A Simple Complete Search for Logic Programming

Jason Hemann[1], Daniel P. Friedman[2], William E. Byrd[3], and Matthew Might[4]

1 Indiana University, Bloomington, IN 47402, USA
jhemann@indiana.edu
2 Indiana University, Bloomington, IN 47402, USA
dfried@indiana.edu
3 University of Utah, Salt Lake City, UT 84112, USA
Will.Byrd@cs.utah.edu
4 University of Utah, Salt Lake City, UT 84112, USA
might@cs.utah.edu

Abstract

Here, we present a family of complete interleaving depth-first search strategies for embedded, domain-specific logic languages. We derive our search family from a stream-based implementation of incomplete depth-first search. The DSL's programs' texts induce particular strategies guaranteed to be complete.

1998 ACM Subject Classification D.3.2 Language Classifications: Applicative (functional) languages, Constraint and logic languages

Keywords and phrases logic programming, streams, search, Racket, backtracking, relational programming

Digital Object Identifier 10.4230/OASIcs.ICLP.2017.14

1 Introduction

A common logic language implementation technique is the shallowly-embedded, internal domain-specific language (DSL) [12, 8, 4]. In this technique, the logic-language programmer writes in the syntax of the underlying host language and the DSL's operators' behavior are described in terms of the host's semantics. Designers need implement only behaviors not supported natively by the host. For logic languages implemented in functional hosts, these may include backtracking and search, among others.

Here, we present a family of complete interleaving depth-first search strategies induced by an embedding. Each logic program's text induces a particular search strategy. Unlike most other embeddings, our operators provide a complete search without the performance penalties associated with, for example, breadth-first search [12, 8]. We improve on earlier efforts [5] by combining the hand-off of control with relation definition, and in doing so decrease the amount of interleaving while maintaining a complete search. We achieve a minimal placement of interleaving points for arbitrary relation definitions.

We host our embedding in Racket [3], but any eager language with functions as values is equally suited. We deliberately restrict ourselves to a small host language feature set. We rely chiefly on `cons` and `lambda` (λ). The data-structure interpolation operators `‘` and `,` are a shorthand for explicit `cons`es, and the promise and force operators we use are shallow wrappers over function creation and application.

Program

A program consists of zero or more *relations* (predicates, in Prolog parlance) and an initial *goal*. Invoking the first goal may require a call to some relation, which may itself require a call to another relation or relations, etc.

Goals

Goals are implemented as functions that take a *state* and return a *stream* of states. They consist of primitive constraints such as equality (`==`), relation invocations like `(peano q)`, and their closure under operators that perform conjunction, disjunction, and variable introduction.

State

We execute a program p by attempting an initial goal in the context of zero or more relations. The program proceeds by executing a goal in a *state*. The state contains a *substitution* and a *counter* for generating fresh variables. Every program's execution begins with an *initial state* devoid of any constraint information and a variable count 0.

Streams

Executing a goal in a state s/c (connoting a substitution and counter pair) yields a stream. A stream takes one of three shapes. The stream may be empty, indicating the goal cannot be achieved in s/c. A stream may contain one or more resultant states. In this case, each element of the stream is a different (in terms of control flow (i.e., disjunctions); the same state may occur many times in a single stream) way to achieve that goal from s/c. Our streams are not necessarily infinite; there may be finitely many ways to achieve a goal in a given state. We call these first two shapes *mature*, whereas an *immature* stream is a delayed computation that will return a stream when forced.

The final step of running a program is to continually force the resultant stream until it yields a list of answers. Our programs are not guaranteed to terminate. The stream we get from invoking the initial goal may be *unproductive*: repeated applications of `force` will never produce an answer [11]. This is the only potential cause of non-termination; all of the other core operations in our implementation are total.

2 Implementing Depth-first Search

We now implement our interleaving search operators: `disj`, `conj`, `define-relation`, and `call/initial-state`. We omit here the syntactic equality constraint `==` and `call/fresh` (which scopes new logic variables). Interested readers should consult an extended version of this work [6].

The binary operators `disj` and `conj` act as goal combinators, and they let us to write composite goals representing the disjunction or conjunction of their arguments.

```
#| Goal × Goal → Goal |#
(define ((disj g1 g2) s/c) ($append (g1 s/c) (g2 s/c)))

#| Goal × Goal → Goal |#
(define ((conj g1 g2) s/c) ($append-map g2 (g1 s/c)))
```

We define `disj` and `conj` in terms of `$append` and `$append-map`. If we define these functions as aliases for the finite-list `append` and `append-map` functions standard to many languages [10], our streams will always be empty or answer-bearing; in fact, they will be fully computed. The result of attempting an `==` goal must be a finite list, of length 0 or 1. If both of `disj`'s arguments are goals that produce finite lists, then the result of invoking `append` on those lists is itself a finite list. If both of `conj`'s arguments are goals that produce finite lists, then the result of invoking `append-map` with a goal and a finite list must itself be a finite list. Invoking a goal constructed from these operators in the initial state returns a list of all successful computations, computed in a depth-first, preorder traversal of the search tree generated by the program.

3 Recursion and `define-relation`

We must enrich our implementation to allow recursive relations. DFS is incomplete for computations with infinite branches. Consider the following stylized Prolog definition of the predicate `peano` that generates Peano numbers.

```
peano(N) :- N = z ; [s R], peano(R).
```

At present there are several obstacles to writing relations like `peano` that refer to themselves or one another in their definitions in our embedding. Suppose we'd used `define` to build a function that we hope would behave like a relation:

```
(define (peano n)
  (disj (== n 'z)
        (call/fresh (λ (r) (conj (== n `(s ,r))
                                 (peano r))))))
```

When we use the `peano` relation in the following program, we hope to generate some Peano numbers. We invoke `(call/fresh ...)` with an initial state. Invoking that goal creates and lexically binds a new fresh variable over the body. The body, `(peano n)`, evaluates to a goal that we pass the state `(() . 1)`. This goal is the disjunction of two subgoals. To evaluate the `disj`, we evaluate its two subgoals, and then call `$append` on the result. The first evaluates to `(((0 . z)) . 1)`, a list of one state.

```
> ((call/fresh (λ (n) (peano n)))
   '(() . 0))
```

Invoking the second of the `disj`'s subgoals however is troublesome. We again lexically scope a new variable, and invoke the goal in body with a new state, this time `(() . 2)`. The `conj` goal has two subgoals. To evaluate these, we run the first in the current state, which results in a stream. We then run the second of `conj`'s goals over each element of the resulting stream and return the result. Running this second goal begins the whole process over again. In a call-by-value host, this execution won't terminate. Simply using `define` in this manner will not suffice.

We instead introduce the `define-relation` operator. This allows us to write recursive relations; with a sequence of uses of `define-relation`, we can create mutually recursive relations. Unlike the other operators, `define-relation` is a macro.

```
(define-syntax-rule (define-relation (defname . args) g)
  (define ((defname . args) s/c) (delay/name (g s/c))))
```

Racket's `define-syntax-rule` gives a simple way to construct non-recursive macros. The first argument is a pattern that specifies how to invoke the macro. The macro's first symbol, `define-relation`, is the name of the macro we define. The second argument is a template to be filled in with the appropriate pieces from the pattern. We do implement `define-relation` in terms of Racket's `define`.

This macro expands a name, arguments, and a goal expression to a `define` expression with the same name and number of arguments and whose body is a goal. It takes a state and returns a stream, but unlike the others we've seen before, this goal returns an immature stream. When given a state `s/c`, this goal returns a promise that evaluates the original goal `g` in the state `s/c` when forced, returning a stream. A promise that returns a stream is itself an immature stream.

`define-relation` does two useful things for us: it adds the relation name to the current namespace, and it ensures that the function implementing our relation is total. It turns out that we will *never* re-evaluate an immature stream. Unlike `delay`, `delay/name` doesn't *memoize* the result of forcing the promise, so it is like a "by name" variant of `delay`. In languages without macros, the programmer could explicitly add a delay at the top of each relation; though this has the unfortunate consequence of exposing the implementation of streams.

We implement `define-relation` as a macro, since it is critical that the expression `g` not be evaluated prematurely: we need to delay the invocation of `g` in `s/c`. Under call-by-value, a function would (prematurely) evaluate its argument and would not delay the computation.

This solves the non-termination of relation invocations. When `peano` is defined by `define-relation`, the goal `(peano n)` immediately returns an immature stream when invoked. We can also write recursive relations whose goals quite clearly will never produce answers.

```
(define-relation (unproductive n)
  (unproductive n))
```

We now redefine `$append` and `$append-map`, augmenting them with support for immature streams.

```
(define ($append $1 $2)
  (cond
    ((null? $1) $2)
    ((promise? $1) (delay/name ($append (force $1) $2)))
    (else (cons (car $1) ($append (cdr $1) $2)))))
```

If the recursive argument to `$append` is an immature stream, we return an immature stream, which, when forced, continues appending the second to the first. Likewise, in `$append-map`, when `$` is an immature stream, we return an immature stream that will continue the computation but still forcing the immature stream. Rather than `delay/name`, `force`, and `promise?`, we could have used `(λ () ...)`, procedure invocation, and `procedure?`. Using λ to construct a procedure delays evaluation, and `procedure?` would be our test for an immature stream.

```
#| Goal × Stream → Stream |#
(define ($append-map g $)
  (cond
    ((null? $) '())
    ((promise? $) (delay/name ($append-map g (force $))))
    (else ($append (g (car $)) ($append-map g (cdr $))))))
```

After these changes, we must do something special when we invoke a goal in the initial state, as this can now produce an immature stream instead of an empty or answer-bearing stream such as in the following example.

```
> ((call/fresh (λ (n) (peano n)))
   '(() . 0))
#<promise>
```

4 call/initial-state

At the very least, we would like to know if our programs are *satisfiable* or not. That is, we would hope to get at least one answer if one exists, and the empty list if there are none. The `call/initial-state` operator ensures that if we return, we return with a list of answers.

```
#| Maybe Nat⁺ × Goal→ Mature |#
(define (call/initial-state n g) (take n (pull (g '(() . 0)))))
```

`call/initial-state` takes an argument `n` which represents the number of answers to retrieve. `n` may just be a positive natural number, in which case we return at most that many answers. Otherwise, we provide `#f`, indicating our embedding should return *all* answers. It also takes a goal as an argument. The function `pull` takes a stream as argument, and if `pull` terminates, it returns a mature stream. As streams may be unproductive, it is not always possible to produce a mature stream. As a result, `pull`, and consequently `take` and `call/initial-state`, are partial functions. These are the only partial functions in our implementation.

```
#| Stream → Mature |#
(define (pull $) (if (promise? $) (pull (force $)) $))
```

`take` receives the mature stream that is the result of `pull` and, `n`, the argument dictating whether to return all, or just the first n elements of the stream.

```
#| Maybe Nat⁺ × Mature → List |#
(define (take n $)
  (cond
    ((null? $) '())
    ((and n (zero? (- n 1))) (list (car $)))
    (else (cons (car $) (take (and n (- n 1)) (pull (cdr $)))))))
```

Our embedding is now capable of creating, combining, and searching for answers in infinite streams.

```
> (call/initial-state 2
    (call/fresh (λ (n) (peano n))))
'((((0 . z)) . 1) (((1 . z) (0 . (s 1))) . 2))
```

Rather than always returning a list implementation of non-deterministic choice, we either have no values, a value now (possibly more than one), or something we can search later for a value. `pull`, since it forces an actual value out of a promise, is akin to run in the delay monad. `take` bears a similar relationship to run in the list monad.

5 Interleaving, Completeness, and Search

Although we can now create and manage infinite streams, we cannot manage them as well as we'd like. Consider what happens in the following program execution:

```
> (call/initial-state 1
    (call/fresh (λ (n) (disj (unproductive n)
                             (peano n)))))
```

We wish the program to return a stream containing the `n`s for which `peano` holds and in addition the `n`s for which `unproductive` holds. We know from Section 3 that there are no `n`s for which `unproductive` holds, but infinitely many for `peano`. The stream should contain only ns for which peano holds. It's perhaps surprising, then, to learn that this program loops infinitely.

Streams that result from using unproductive will always be, as the name suggests, unproductive. When executing the program above, such an unproductive stream will be the recursive argument $1 to $append. Unproductive streams are necessarily immature. According to our definition of $append, we always return the immature stream. When we force this immature stream, it calls $append on the forced stream value of (the delayed) $1 and $2. Since unproductive is unproductive, this process continues without ever returning any of the results from peano.

Such surprising results are not solely the consequence of goals with unproductive streams. Consider the definition of church.

```
(define-relation (church n)
  (call/fresh (λ (b) (conj (== n `(λ (s) (λ (z) ,b)))
                           (peano b)))))
```

The relation church holds for Church numerals. Using a newly created variable b, it constructs a list resembling a lambda-calculus expression whose body is the variable b. It uses peano to generate the body of the numeral. We can thus use it to generate Church numerals in a manner analogous to our use of peano. But consider the following program, wherein the resulting stream is productive, but only contains elements for which peano holds.

```
> (call/initial-state 3
    (call/fresh (λ (n) (disj (peano n)
                             (church n)))))
'((((0 . z)) . 1) (((1 . z) (0 . (s 1))) . 2)
  (((2 . z) (1 . (s 2)) (0 . (s 1))) . 3))
```

Under the default Racket printing convention, "." is suppressed when it precedes a "(". We retain the "." for legibility – Racket's current-print parameter controls this behavior.

Our implementation of $append in Section 3 induces a depth-first search. Depth-first search is the traditional search strategy of Prolog and can be implemented quite efficiently. As we've seen though, depth-first search is an *incomplete* search strategy: answers can be buried infinitely deep in a stream. The stream that results from a disj goal produces elements of the stream from the second goal only after exhausting the elements of the stream from the first.

```
#| Stream × Stream → Stream |#
(define ($append $1 $2)
  (cond
    ...
    ((promise? $1) (delay/name ($append (force $1) $2)))))
```

As a result, even if answers exist microKanren may fail to produce them. We will remedy this weakness in $append, and provide microKanren with a simple complete search. We want microKanren to guarantee each and every answer should occur at a finite position in the stream. Fortunately, this doesn't require a significant change.

```
#| Stream × Stream → Stream |#
(define ($append $1 $2)
  (cond
    ...
    ((promise? $1) (delay/name ($append $2 (force $1))))))
```

That's it. This one change to the promise? line of $append is sufficient to make disj *fair* and to transform our search from an incomplete, depth-first search to a complete one.

Interestingly, we haven't reconstructed a particular, single complete search strategy. Instead, the search strategy of microKanren programs is program- and query-specific. The particular definitions of a program's relations, together with the goal from which it's executed, dictates the order we explore the search tree. By contrast, Spivey and Seres implement breadth-first search, also a complete search, in a language similar to microKanren [12].

Relying on non-strict evaluation simplifies their implementation; manually managing delays would make the call-by-value version less elegant than their implementation. Even excepting that, their implementation requires a somewhat more sophisticated transformation than does ours. Kiselyov et al. describe a different mechanism to achieve a complete search, but they too rely on non-strict evaluation [9]. We achieve a simpler implementation of a complete search by using the delays as markers for interleaving our streams.

6 Conclusion and Related Work

There has been extensive research on logic programming implementation [1]. Spivey and Seres's [12] present a Haskell embedding of a language quite similar to microKanren. They begin with depth-first search language, and through transformations derive an implementation of breadth-first search.

Hinze [7, 8] and Kiselyov et al. [9] implement backtracking with asymptotic performance improvements over stream-based approaches like that used in microKanren and the works cited above. These context-passing implementations are also more complicated to understand and to implement. We chose to use streams in part to more easily communicate ideas.

The fair search operators in Kiselyov et al.'s LogicT monad provide the basis of the interleaving search in earlier miniKanren implementations. The LogicT transformer augments an arbitrary monad with backtracking and control operators similar to those we use. We have access to the whole logic program in our embedding and carefully control interleaving in recursions; therefore we can use less frequent interleaving and maintain a complete search.

Our development led us to a number of interesting, still-open problems. Hinze [7] shows list-based implementations of nondeterminism to be asymptotically slower than a continuation-based "context-passing" implementation. We would like to combine our manual control of delays with a context-passing implementation à la Hinze and Kiselyov et al. [9]. Earlier work by Wand [13] and Danvy et al. [2] in relating models of backtracking has provided a starting point.

While `define-relation` is sufficient to ensure our search is complete, it in general causes more interleaving than necessary. For instance, mutually-recursive relations only need one interleaving point between them, and we don't need to interleave at all deterministic relations. We could statically "push down" the delays into the body of a relation, reducing the amount of interleaving we perform while retaining a complete search. We would also like to mechanically prove the correctness of our search with a dependently-typed implementation whose types encode our fairness properties.

Acknowledgements. We thank our reviewers, both known and anonymous, for their comments and suggestions. This material is partially based on research sponsored by DARPA under agreement number AFRL FA8750-15-2-0092 and by NSF under CAREER grant 1350344. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

1 Isaac Balbin and Koenraad Lecot. *Logic Programming: A Classified Bibliography*. Springer Science & Business Media, 2012.

2 Olivier Danvy, Bernd Grobauer, and Morten Rhiger. A unifying approach to goal-directed evaluation. *New Generation Computing*, 20(1):53, 2002. URL: http://dx.doi.org/10.1007/BF03037259, doi:10.1007/BF03037259.

3 Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Design Inc., 2010. http://racket-lang.org/tr1/.

4 Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. *The Reasoned Schemer*. MIT Press, Cambridge, MA, 2005.

5 Jason Hemann and Daniel P. Friedman. μKanren: A minimal functional core for relational programming. In *Scheme 13*, 2013. URL: http://schemeworkshop.org/2013/papers/HemannMuKanren2013.pdf.

6 Jason Hemann, Daniel P. Friedman, William E. Byrd, and Matthew Might. A small embedding of logic programming with a simple complete search. In *Proceedings of DLS '16*. ACM, 2016. URL: http://dx.doi.org/10.1145/2989225.2989230.

7 Ralf Hinze. Deriving backtracking monad transformers. In *ACM SIGPLAN Notices*, volume 35, pages 186–197. ACM, 2000.

8 Ralf Hinze. Prolog's control constructs in a functional setting: Axioms and implementation. *International Journal of Foundations of Computer Science*, 12(02):125–170, 2001.

9 Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers: (functional pearl). In Olivier Danvy and Benjamin C. Pierce, editors, *Proceedings of the 10th ACM SIGPLAN ICFP*, pages 192–203. ACM, September 2005.

10 Olin Shivers. List Library. Scheme Request for Implementation. SRFI-1, 1999. URL: http://srfi.schemers.org/srfi-1/srfi-1.html.

11 Ben A. Sijtsma. On the productivity of recursive list definitions. *ACM Trans. Program. Lang. Syst.*, 11(4):633–649, October 1989. URL: http://doi.acm.org/10.1145/69558.69563, doi:10.1145/69558.69563.

12 JM Spivey and Silvija Seres. Embedding Prolog in Haskell. In E. Meier, editor, *Haskell 99*, 1999.

13 Mitchell Wand and Dale Vaillancourt. Relating models of backtracking. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*, ICFP '04, pages 54–65, New York, NY, USA, 2004. ACM. URL: http://doi.acm.org/10.1145/1016850.1016861, doi:10.1145/1016850.1016861.