1 Staged miniKanren

Staged miniKanren is an extension of miniKanren that supports staging. Staged programming is manual partial evaluation, where offline manual binding time analysis is used to select operations for lifting. In the case of staged miniKanren, we typically choose to lift some unifications and some conditionals.

Given a program and only partially known input data, one can eliminate or simplify certain parts of the code, by performing computations that only involve available data. Depending on the known input, we might also be able to make observations such as whether a loop is degenerate for most iterations and restructure the code accordingly. As a result, we can have a new program that is specialized to the available input.

In order to perform as much computation as possible, it is important to correctly annotate arguments according to their availability. This is done via *binding time analysis*.

The power of staging is manual control of binding time analysis, where one can manually decide which operations are to be lifted. The manual design stems from the observation that automatic binding-time analysis is hard, and that manual control is a powerful compromise that allows the staging user to specify what they want exactly.

1.1 Deferring unifications

In staged miniKanren, some unifications are done in the first stage, while others are quoted out and gets deferred to the second stage. The second stage represents code that is "kept for later", while the first stage is for executing now. Deferring a unification is similar to deferring a command in functional programming.

To defer a unification, we use the `l==` operator:

```
(test (run* (q) (l== q 1) (l== q 2))
      '((_.0 !! ((== _.0 2) (== _.0 1)))))
(test (run* (q) (conde [(l== q 1)] [(l== q 2)]))
      '((_.0 !! ((== _.0 1))) (_.0 !! ((== _.0 2)))))
```

Here, the two unifications get deferred to the second stage. To defer a goal we use the `lift` operator:

```
(test (run* (q) (lift `(conde [(== ,q 1)] [(== ,q 2)])))
      '((_.0 !! ((conde ((== _.0 1)) ((== _.0 2)))))))
```

Lifting unifications `l==` is defined in terms of the more general `lift`.

```
(define fake-evalo (lambda (q n)
                    (fresh ()
                      (l== q n)
                      (l== n n))))
(test
 (run* (q)
       (fresh (c1 c2)
              (lift-scope (fake-evalo q 1) c1)
            (lift-scope (fake-evalo q 2) c2)
            (lift `(conde ,c1 ,c2))))
 '((_.0 !!
       ((conde
         ((== _.0 '1) (== '1 '1))
         ((== _.0 '2) (== '2 '2)))))))
```

1.2 Dynamic variables

An alternative approach to manual lifting of unifications is to introduce dynamic variables, where we annotate some variables by hand, and the lifting automatically follows. To decide whether the unification needs to be deferred we could examine the terms being unified to see whether they contain a dynamic variable. This approach is not very promising as it is unclear how dynamic variables should interact with non-dynamic variables and the rest of the code. For example, if we just say that a term containing a dynamic variable is dynamic, then an expression like '(cdr (cons 5 ,y))' where only 'y' is dynamic will be treated as dynamic, which is not ideal.

1.3 Using staged miniKanren for fuzzing

Fuzzing is used for automated software testing via random high-frequency generation of programs (for testing interpeters and compilers) as well as inputs of a certain format (for testing programs in general). Suppose we have a program whose input is a binary search tree; it is meant to accept binary search trees and reject any other input. A fuzzer for this program would automatically generate trees to test that inputs are accepted and rejected correctly. In general, if we wish to test for soundness, we would like to avoid generating completely irrelevant and incorrect inputs, as the software is probably good enough to reject them [1]. It is more likely that almost correct, or "quasi"-correct inputs will be mistakenly accepted. Thus, ideally, a fuzzer should generate inputs that are, for example, somehow "off by one". In miniKanren, a given acceptor function can be compiled into a relation, and then use the `run` function to fuzz correct inputs. It would be useful to have an infrastructure that takes an arbitrary acceptor relation and makes a relation for fuzzing quasi correct inputs. Consider these examples.

(a) *Mirror trees.* Suppose we have a reverse function that recursively swaps the leaves of a tree. Then define a *mirror* to be a binary tree that is equal to its reverse. Now we can try to define *almost-mirrors* that are almost symmetrical. To fuzz almost-mirrors, we can inject a random choice somewhere in the definition of mirror:

```
(run 1 (q)
(evalo
   `(letrec ((mirror
         (lambda (t)
            (match t
               [(? number? n) n]
               [`(node ,l ,(? number? n) ,r)
                  (list 'node (choice (mirror r) r) n (choice (mirror l)
                     l))]))))
   (equal? ',q (mirror `,q)))
   #t))
```

where `choice` chooses the first argument with probability 90%, for example. Here `mirror` is a function that takes a tree `t` and returns #`t` if `t` is a mirror. The goal `evalo` takes the `letrec` expression setting `mirror` to the recursive mirror function in `(equal? ',q (mirror `,q))` and unifies it with $\#t$, in an empty substitution.

One way to do this without weights is to have a non-deterministic `choice` implemented using `conde`:

```
(define (choice-primo expr env val)
   (fresh (e2 e3)
      (== `(choice ,e2 ,e3) expr)
      (not-in-envo 'choice env)
      (conde
         ((eval-expo e2 env val))
         ((eval-expo e3 env val)))))
```

(b) *Binary search trees.* An quasi binary search tree could be a binary tree where the left leaf of some node is greater than the right one by one.

(c) *A program processing HTML pages* can be fuzzed with quasi HTML pages containing subtle syntax mistakes.

A potential use case for staged miniKanren is using it for optimizing fuzzers. Thus, the unavailable data in the first example could be the first argument to the `run` function and the argument to `evalo` so that we stage the same program with some known parameters replaced by symbols:

```
(run X (q)
(evalo
   `(letrec ((mirror
          (lambda (t)
             (match t
                [(? Y n) n]
                [`(node ,l ,(? Y n) ,r)
                   (list `node (choice (mirror r) r) n (choice (mirror l) l))]))))
   (equal? ',q (mirror `,q)))
#t))
```

Where we aim to get an optimized fuzzer specialized to generating almost-mirrors, and the Y argument to the resulting function is the type of data stored in the tree.

In general, we wish to explore how we can use staging for optimizing functions into relations. Thus, given a program of the form

```
(run X (q)
(evalo
   `(letrec ((f ...))
   ...)
...))
```

Where some values are symbolic, we might be able to lift operations containing them and generate an optimized program.

1.4 Different types of variables and values

In addition to usual lexical variables in Scheme which can be created using `let`, λ, etc, miniKanren has its own logic variables, which can be bound to values via a *substitution* mapping. These can be *ground*, i.e. associated with a value, or *fresh*. If we unify two fresh variables `X` and `Y`, they both remain fresh. A term is *ground* if it contains variables that are all ground. Logic variables are represented by a vector and can be created using the `fresh` operator. In particular, the `fresh` operator creates a fresh logic variable and binds it to a lexical variable. Initially, the logic variable does not have a value associated with it, but it can become ground through unification. In addition to associating logic variables with values, we can map them to constraints such as disequality and absento.

1.5 Deferring operations

Staging seems to be related to delayed goals, and so it might be potentially useful to integrate the two in a single miniKanren extension. This integration would look as follows. When compiling functions into relations, starting with a conjuction, we evaluate the first goal and the result is incorporated in each substitution, then we move on to the next goal, etc. Now in addition to these we could have an extra step of deferring trigonometric, logarithmic and other tricky unifications, such as `(== X (cos` $\frac{\pi}{4}$`))` or `(== X (+ 3 Z))` where `X, Z` will be known later. Unifying `x` with `cos` π is thus delayed until we have either more information (from a new unification) or appropriate tools for performing, for example, floating-point computations. Even if we have ground arguments we can have various reasons to delay a unification, for example we might want to unify `X` with precisely $\sqrt{2}$ and not its floating-point representation. Dependency analysis would be necessary for the case when we wish to perform the unification as soon as there is more information available.

Then, to make it possible to enter and exit a `run` or a `run`$*$ process at flexible points we might wish to enable feeding and retrieving a constraint stream from it.

2 Current implementation of Staged miniKanren

2.1 Staged interpreter

Suppose our goal is to interpret an expression `expr` and unify the result with `val`. This can be done with `(evalo expr val)`, which calls `eval-expo` with an initial environment. The staged version of `eval-expo` needs to handle staged code. Consider the lifting operator `lift`. The result of applying `lift` to a goal is a lifted goal (so like a goal, it takes a state and returns a state.) In the code below, the first two goals are "`stage?` is true" and "`expr` is a variable". The third goal takes a state `c` and returns the state resulting from applying

```
(lift ‘(u-eval-expo ,expr ,(quasi (walk* env (c->S c))) ,val))
```

to `c`. In particular, we defer calling the unstaged `eval-expo` on the variable `expr` and `val`. Note that the environment now is a substitution map, since the unstaged `eval-expo` takes a substitution as the second argument:

```
(define (eval-expo stage? expr env val)
 ...
  (conde
    ((== stage? #t) (varo expr)
     (lambda (c)
       ((lift ‘(u-eval-expo ,expr ,(quasi (walk* env (c->S c))) ,val))
        c)))
```

Another lifting operator is `l==`, the lifted unification constructor. Below a fresh variable `v` is introduced and unified with `expr`. The unification of `v` with `val` is then deferred via `l` $==$, unlike the first-stage unification `(== expr v)` that we had in the unstaged miniKanren:

```
(define (eval-expo stage? expr env val)
  (conde
    ...
    ((conde
       ((non-varo expr))
       ((== stage? #f)))
     (conde
       ((fresh (v)
          (== `(quote ,v) expr)
          (absento 'closure v)
          (absento 'prim v)
          (not-in-envo 'quote env)
          ((if stage? l== ==) val v)))
```

In the unstaged miniKanren, if `expr` is a number, we unify it with `val`:

```
    (define (eval-expo expr env val)
        ...
        (conde
            ((numbero expr) (== expr val))
            ...))
```

In staged miniKanren, `eval-expo` takes the additional `staged?` argument. If `staged?` is true, then we defer the unification:

```
       ((numbero expr) ((if stage? l== ==) expr val))
```

Similarly, if it is a symbol, then depending on the value of `staged?`, we either look up the symbol immediately or defer to the second stage:

```
       ((symbolo expr) (lookupo stage? expr env val))
```

where `lookupo` is the function

```
(define (lookupo stage? x env t)
  (fresh (y b rest)
    (== `((,y . ,b) . ,rest) env)
    (conde
      ((== x y)
       (conde
         ((fresh (v) (== `(val . ,v) b) (== v t)))
         ((fresh (rec-fold? lam-expr)
            (== `(rec ,rec-fold? . ,lam-expr) b)
            (conde
              ((== rec-fold? #t) (== `(call ,x) t))
              ((== rec-fold? #f) (== `(closure ,lam-expr ,env) t)))))))
      ((=/= x y)
       (lookupo stage? x rest t)))))
```

If `expr` is a function (which we check by unifying `‘(lambda ,x ,body)` with `expr` in the first stage) and `stage?` is true, then we defer the unification of `val` with the closure `‘(closure (lambda ,x, body), env)` to the second stage:

```
((fresh (x body)
   (== ‘(lambda ,x ,body) expr)
   ((if stage? l== ==) ‘(closure (lambda ,x ,body) ,env) val)
```

The rest of this clause is the same as in the unstaged miniKanren. The variable `x` can either be variadic:

```
(conde
   ((symbolo x))
```

or multi-argument:

```
   ((list-of-symbolso x)))
```

Finally, we check that `‘lambda` is not in the environment:

```
(not-in-envo ’lambda env)))
```

Here, the expression is an operation, so we unify `expr` with ‘(, *rator*., *rands*), where `rator` stands for “operator” and `rands` stands for “operands”:

```
((fresh (rator x rands body env^ a* res)
   (== ‘(,rator . ,rands) expr)
```

Then we have the goal that `x` is a symbol that is a variadic argument.

```
   (symbolo x)
```

We also need an environment `res` in which the body of the operator will be evaluated. This is the environment in the closure `rator`, concatenated with the association of the symbol `x` with the pair `val` and `a*`, where `a*` is the result of evaluating the operands `rands`. The operator is a relation, so we pass `val` as an argument in addition to the `rands`.

```
   (== ‘((,x . (val . ,a*)) . ,env^) res)
```

The operator is a lambda taking `x` and returning `body`. In particular, we evaluate unstaged `rator` and unify it with a closure.

```
   (eval-expo #f rator env ‘(closure (lambda ,x ,body) ,env^))
```

We then evaluate the body of the operator, by calling `eval-expo` on `body`, passing `stage?` argument, and the `val` argument. Note that we evaluate `rator` with the `stage?` argument set to false, as here we do not defer evaluating the operator itself; the staging applies to evaluating the body of `rator`. Thus, we do pass the original `stage?` argument when we evaluate the *body* of the operator in the extended environment:

```
(eval-expo stage? body res val)
```

The operands are evaluated using `eval-listo` and are unified with `a*`.

```
(eval-listo rands env a*)))
```

If the argument to the operand is a list, we have similar goals but use `ext-env*o` to extend the environment `env^`:

```
((fresh (rator x* rands body env^ a* res)
   (== `(,rator . ,rands) expr)
   (eval-expo #f rator env `(closure (lambda ,x* ,body) ,env^))
   (eval-listo rands env a*)
   (ext-env*o x* a* env^ res)
   (eval-expo stage? body res val)))
```

If `stage?` is true, and `rator` evaluates to some `(call ,p-name)`, then we can lift the goal `(,p-name . ,a*) ,val`. Here `a*` is the list of the evaluated `rands` and `val` is unified the result of the operation.

```
((fresh (rator rands a* p-name)
   (== stage? #t)
   (== `(,rator . ,rands) expr)
   (eval-expo #f rator env `(call ,p-name))
   (eval-listo rands env a*)
   (lift `((,p-name . ,a*) ,val))))
```

If `rator` is a symbol and `stage?` is true, then we do not evaluate `rands` and unify with an `a*`. The unstaged evaluation of the entire `expr` is lifted, and we pass `(quasi (walk* env (c->S c)))` as the environment to `u-eval-expo` ("unstaged `eval-expo`").

```
((fresh (rator rands p-name)
    (== stage? #t)
    (== `(,rator . ,rands) expr)
    (symbolo rator)
    (eval-expo #f rator env `(sym . ,p-name))
    (lambda (c)
      ((lift `(u-eval-expo ',expr ,(quasi (walk* env (c->S c))) ,val))
       c))))
```

If `rator` is a primitive operator, we can find its id by unifying with `(prim . ,prim-id)` and evaluate it on the evaluated operands `a*`.

```
((fresh (rator x* rands a* prim-id)
   (== `(,rator . ,rands) expr)
   (eval-expo #f rator env `(prim . ,prim-id))
   (eval-primo prim-id a* val)
   (eval-listo rands env a*)))
```

Here we unify `expr` with a `letrec` expression. The `bindings*` fresh variable unifies with the bindings in the expression and `letrec-body` unifies with the body.

```
((fresh (bindings* letrec-body out-bindings* env^)
   (== `(letrec ,bindings*
          ,letrec-body)
       expr)
```

The `bindings*` are evaluated using `letrec-bindings-evalo` with the result being `out-bindings*`. We unify `stage?` with true and use `lift-scope` to generate staged code in the body of the `letrec` expression. Then the new `letrec` expression `(letrec ,out-bindings* (fresh () . ,c-letrec-body))` is lifted using the `lift` operator.

```
   (letrec-bindings-evalo bindings* out-bindings* env env^ env^)
   (not-in-envo 'letrec env)
   (== stage? #t)
   (fresh (c-letrec-body)
     (lift-scope
      (eval-expo #t letrec-body env^ val)
      c-letrec-body)
     (lift `(letrec ,out-bindings*
              (fresh () . ,c-letrec-body))))))
  ...
 ))))
```

Additionally, the staged interpreter has these functions.

The `mapo` *function* unifies `xs` with a pair `xa` and `xd`, unify `ys` with a pair `ya` and `yd`. The function `fo` is applied with the arguments `xa` and `ya`, and we apply the `mapo` relation to the tails of the two lists.

```
(define (mapo fo xs ys)
  (conde
    ((== xs '()) (== ys '()))
    ((fresh (xa xd ya yd)
       (== xs (cons xa xd))
       (== ys (cons ya yd))
       (fo xa ya)
       (mapo fo xd yd)))))
```

The `mapo` function is used to make a list of symbols.

```
(define (make-list-of-symso xs ys)
  (mapo (lambda (x y) (== y (cons 'sym x))) xs ys))
```

Goals for checking that `x` is a variable or a non-variable are introduced:

```
(define (varo x)
  (lambda (c)
    (if (var? (walk* x (c->S c)))
        c
        #f)))
(define (non-varo x)
  (lambda (c)
    (if (var? (walk* x (c->S c)))
        #f
        c)))
```

The `fix-l==` function lifts `t` if `(car t)` is `==` or `=/=`.

```
(define fix-l==
  (lambda (t)
    (if (and (pair? t)
             (or (eq? '== (car t))
                 (eq? '=/= (car t))))
        (list (car t) (quasi (cadr t)) (quasi (caddr t)))
        t)))
```

The `quasi` *function.* If `t` is a variable, return that variable. If `t` is a pair and the first entry is a symbol, return the second entry, otherwise apply `quasi` to both entries. If `t` is null, return `'()`. Otherwise `(list 'quote t)` is returned.

```
(define quasi
  (lambda (t)
    (cond
      ((var? t) t)
      ((and (pair? t) (eq? (car t) 'sym)) (cdr t))
      ((pair? t) (list 'cons (quasi (car t)) (quasi (cdr t))))
      ((null? t) ''())
      (else (list 'quote t)))))
```

The function `walk-lift` walks `C` and lifts all unifications.

```
(define walk-lift
  (lambda (C S)
    (map fix-l== (walk* (reverse C) S))))
```

Lifting a goal adds the goal to the `C` constraint store.

```
(define lift
```

```
  (lambda (x)
    (lambdag@ (c : S D A T C L)
      `(,S ,D ,A ,T ,(cons x C) ,L))))
```

Lifting a scope. This function takes a goal `g` and a variables `var` and makes another goal, which, when applied it to a state `c`, applies `g` to a modified `c` where the `C` store is empty.

```
(define lift-scope
  (lambda (g out)
    (lambdag@ (c : S D A T C L)
      (bind*
       (g `(,S ,D ,A ,T () ,L))
       (lambdag@ (c2 : S2 D2 A2 T2 C2 L2)
         ((fresh ()
            (== out (walk-lift C2 S2)))
          `(,S ,D ,A ,T ,C ,L)))))))
```

The `l==` *operator* takes arguments $\texttt{e}_1$ and $\texttt{e}_2$ and creates a lifted goal `(lift `(== ,e1 ,e2))`.

```
(define l== (lambda (e1 e2) (fresh () (lift `(== ,e1 ,e2)))))
```

The `dynamic` *function* takes a symbol `xs` and returns a goal that, given a state, appends it to the `L` field.

```
(define dynamic
  (lambda xs
    (lambdag@ (c : S D A T C L)
      `(,S ,D ,A ,T ,C ,(append L xs)))))
```

2.2 Staged miniKanren

In the canonical implementation of miniKanren, a *state* `c` is an object containing

- The *substitution* mapping `S` used to hold substitutions resulting from unifications `==`. The substitution is accessed by `c->S`.
- The *disequality* mapping `D` used to keep track of disequalities associated with variables. It is accessed by the function `c->D`.
- The *absento* mapping `A` used to handle the `absento` goals and accessed by `c->A`.
- The *types* mapping `T` used to keep track of constraints such as `symbolo` and `numbero`.

In staged miniKanren, the state object has two additional components, accessed by `c->C` and `c->L` respectively:

- The *code* component `C` is a list of deferred goals.
- The `L` component holding dynamic variables.

The `C` component holds second stage goals and is extended whenever we use a lifting operator such as `lift`, `lift-scope` or `l==`.

The `lconde` macro is for lifting `conde` expressions. We let `r` be the list consisting of the results of applying all clauses to `c2`, where `c2` is the same as `c` except the `C` list is empty.

```
(define-syntax lconde
  (syntax-rules ()
    ((_ (g0 g ...) (g1 g^ ...) ...)
     (lambdag@ (c : S D A T C L)
       (let ((r (let ((c2 `(,S ,D ,A ,T () ,L)))
                  (append (all-of (bind* (g0 c2) g ...))
                          (all-of (bind* (g1 c2) g^ ...)) ...))))
         ((lift `(conde ,@(map (lambda (c3) (walk-lift (c->C c3) (c->S c3))) r))) c))))))
```

The `walk-lift` replaces variables in `C` with their values according to the substitution `S`:

```
(define walk-lift
  (lambda (C S)
    (map fix-l== (walk* (reverse C) S))))
```

where `walk*` finds the values of all variables in a list and returns a list with the found substitutions:

```
(define walk*
  (lambda (v S)
    (case-value (walk v S)
      ((x) x)
      ((av dv)
       (cons (walk* av S) (walk* dv S)))
      ((v) v))))
```

The `walk` function finds the value of a variable in a substitution:

```
(define walk
  (lambda (u S)
    (cond
      ((and (var? u) (assq u S)) =>
       (lambda (pr) (walk (rhs pr) S)))
      (else u))))
```

The `bind` function binds goals by consecutively applying goals to a state. The `bind*` function applies goals to a stream of states:

```
(define-syntax bind*
  (syntax-rules ()
    ((_ e) e)
```

```
    ((_ e g0 g ...) (bind* (bind e g0) g ...))))
(define bind
  (lambda (c-inf g)
    (case-inf c-inf
      (() (mzero))
      ((f) (inc (bind (f) g)))
      ((c) (g c))
      ((c f) (mplus (g c) (lambdaf@ () (bind (f) g)))))))
```

The `post-unify-=/=` function is used in the definition of the `=/=` goal constructor:

```
(define =/=
  (lambda (u v)
    (lambdag@ (c : S D A T)
      (cond
        ((unify u v S) => (post-unify-=/= S D A T))
        (else (unit c))))))
```

Here even though the goal is a disequality between `u` and `v`, these are first unified in the `S` store. Then, the prefix of the resulting substitution `S+` is taken to be a disequality extension `D+` to `D`. In particular, we need to subsume redundancy among constraints `A`, `T` and `D`. For example, if x is absent in y then x is not equal to it, so disequality between x and y holds automatically. Similarly, a number is not a symbol, etc. In the unstaged miniKanren, these subsumptions are done with the `post-unify` function:

```
(define post-unify-=/=
  (lambda (S D A T)
    (lambda (S+)
      (cond
        ((eq? S+ S) (mzero))
        (else (let ((D+ (list (prefix-S S+ S))))
                (let ((D+ (subsume A D+)))
                  (let ((D+ (subsume T D+)))
                    (let ((D (append D+ D)))
                      (unit `(,S ,D ,A ,T))))))))))
```

In the staged miniKanren, we can additionally handle disequality between dynamic variables. In particular, we can partition the new disequalities according into two sets. Let `C+D-` be this partition. The first set in `C+D-` contains the constraints where one of the variables is dynamic (hence the `C+` part in the name). The second set doesn't contain dynamic variables. Now we can take the dynamic set and write all pairs in it as disequality goals again. The resulting deferred code is then appended to the `C` (second-stage code) component of the state:

```
(define post-unify-=/=
  (lambda (S D A T C L)
```

```
...
              (let ((dynamic? (lambda (x) (memq x L))))
                 (let ((C+D- (partition (lambda (v)
                                          (or (dynamic? (car v))
                                              (dynamic? (cdr v))))
                                        (apply append D+))))
                   (let ((C+ (map (lambda (v) `(=/= ,(car v) ,(cdr v)))
                                  (car C+D-))))
                     (let ((D (append D+ D)))
                       (let ((C (append (reverse C+) C)))
                         (unit `(,S ,D ,A ,T ,C ,L))))))))))))))
```

The disequality constraints involving dynamic variables can thus be deferred to the second stage.

Subsumed constraints. It's possible that a constraint, say A is satisfied whenever some other constraint B is. In this case, the constraint B *subsumes* the constraint A [2].

3 Related work: mixed computation and partial evaluation

3.1 Mathematical premise of partial evaluation

A mathematical premise behind partial evaluation is a theorem stating that there exists a computable function that maps from recursive partial functions of two variables to recursive partial functions of one variable via A *recursive function* is any function that can be computed by a Turing machine. A *recursive partial function* is a partial function that is also recursive.

Suppose we enumerate all Turing machines, so that P_x is the Turing machine with index x (which is called the *Gödel* number of P_x). Then let $\phi_x^{(k)}$ denote the partial function of k variables computed by P_x. Then we have the following theorem [3].

Theorem 1 (Kleene's s-m-n theorem)**.** *Let $m, n \geq 1$ and fix $x, y_1, \ldots, y_m$. Then there is a recursive function s_n^m of $m+1$ variables such that for all $x, y_1, \ldots y_m$,*

$$\lambda z_1, \ldots, z_n[\phi_x^{(m+n)}(y_1, \ldots y_m, z_1, \ldots, z_n)] = \phi^{(n)}_{s_n^m(x, y_1, \ldots, y_m)}.$$

Proof. Suppose $m = n = 1$. We have a class of all possible partial functions of one variable, sending $z \mapsto \lambda z[\phi_x^{(2)}(y, z)]$ for some x, y. This can be viewed as a formal characterization for a class recursive partial functions of one variable. By Basic Result in [3], we can have a procedure f for going from this characterization of functions to the original characterization used to enumerate all Turing machines. Then, by Church's Thesis, there exists a recursive function f of two variables such that

$$\lambda z[\phi_x^{(2)}(y, z)] = \phi_{f(x,y)}.$$

We can then set s_1 to be the function f. Church's Thesis is the generally accepted proposal that Turing and Church characterizations of algorithms agree with the intuitive notions of algorithm and computable functions. $\square$

3.2 Original works by Ershov

Staged programming can be understood as manual partial evaluation, which is the idea that even the most efficient algorithm can be made faster if we specialize it to certain inputs. It was developed by Futamura [4] in the 70-s. Around the same time, Ershov [**?**] and his students explored a related idea of mixed computation. He described it as the process of jointly transforming programs and data in a partially specified setting in order to increase efficiency.

In the first works [**?**], the formal definition was as follows: a mixed computer was defined as a software processor that receives as input some representation of the program and part of the input data and receives at the output a transformed program, part of the result and data that require additional processing. Ershov distinguished functional and operational approaches to this problem. In the functional approach, we are given a function `f` of variables `x, y`. Let us instantiate x with value v_x, and let g be the function $\mathtt{y} \mapsto \mathtt{f}(\mathtt{v_x}, \mathtt{y})$. A *mixed computation* is then a process for deriving a program P_{v_x}, called a *projection* of `P` onto `x`, for computing `g`. We call this kind of computation mixed because concrete computation involving `x` is mixed with code generation. In the operational approach, our starting point is a program P composed of elementary steps. These steps are partitioned into disjoint sets A and B, where A are *permissible* steps and B are *suspensible* steps. Then by *mixed computation* we mean computing the permissible steps and generating a *residual* program from the set B.

A notable application of these ideas is that we can treat programs and even languages themselves as input data. Formally, let us represent an implementation language L by a domain $\mathbf{D}$, a set of programs $\mathbf{P}$, and a semantics V, so $L = (\mathbf{D}, \mathbf{P}, V)$. Assume that $\mathbf{D}$ has a free component $\mathbf{X}$ and a bound component $\mathbf{Y}$, so that we can project programs onto the $\mathbf{X}$ component. Let $l = (\Xi, \Pi, \sigma)$ be a source language, where Ξ is the data domain, Π is the set of all programs written in l and σ is semantics, so that, for all $\pi \in \Pi$ and $\xi \in \Xi$,

$$\sigma(\pi, \xi) = \pi(\xi).$$

Then one can use projections to approach the following problems:

(a) *Source language $l = (\Xi, \Pi, \sigma)$, program $\pi \in \Pi$ are known, input $\xi \in \Xi$ is unknown.* Thus,

$l, \pi \in X$ and $\xi \in Y$ and we wish to compile π into $obj \in \mathbf{P}$ such that obj agrees with π on all inputs $\xi \in \Xi$.

(b) *Source language l is known, program π is unknown.* Thus, $l \in X$ and $\pi \in Y$, and we are looking for a compiler $comp \in \mathbf{P}$ such that $comp(\pi)(\xi) = obj(\xi)$ for all $\xi \in \Xi$.

(c) *Source language l, program π and input ξ are all unknown.* We are then looking for a compiler making compilers, i.e. a program $c \in \mathbf{P}$ such that $c(\sigma)(\pi)(\xi) = comp(\pi)(\xi)$ for all σ, π, ξ.

An interpreter $\mathbf{int} \in \mathbf{P}$ for l computes $\pi(\xi)$ i.e. $\mathbf{int}(\pi, \xi) = \sigma(\pi, \xi)$. Define also $\mathbf{mix}$ to be a program that computes the projection of $p \in \mathbf{P}$ onto known inputs $x \in \mathbf{X}$, so

$$\mathbf{mix}(p, x, \mathbf{Y}) = V(\text{mix}, (p, x, \mathbf{Y}) = p_x(\mathbf{Y}).$$

Then

$$p(x, y) = p_x(y).$$

From the above equations we can derive that

$$\mathbf{mix}(\mathbf{int}, \pi, \Sigma) = \mathbf{int}_{\pi}(\Sigma),$$

and

$$\mathbf{int}_{\pi}(\Sigma) = ob(\Sigma),$$

i.e. to compile a program we project the interpreter onto the source code. We also have

$$\mathbf{mix}_{\mathbf{int}}(\Pi, \Sigma) = comp(\Pi, \Sigma),$$

i.e. to make a compiler we project $\mathbf{mix}$ onto the interpreter. For the third projection, we have

$$\mathbf{mix}_{\mathbf{mix}}(\mathbf{int}, \Pi, \Sigma) = \mathbf{mix}(\mathbf{mix}, \mathbf{int}, (\Pi, \Sigma)) = \mathbf{mix}_{\mathbf{int}}(\Pi, \Sigma)) = comp(\pi, \Sigma),$$

i.e. to make a compiler-compiler we project $\mathbf{mix}$ onto itself.

3.3 Monovariant and Polyvariant binding time analysis

Automated binding time analysis can be monovariant and polyvariant. The general idea is as follows: a monovariant binding time analyzer declares a variable dynamic if it is dynamic in at least one calculation. We would like to transform the original program to make more computations available by copying both the program code and the data as needed. Consider this fragment of the interpreter for evaluating expressions [5]:

```
(define (eval expr mem)
  (cond
    ((isConst? expr) (fetch-constant expr))
    ((isVar? expr) (lookup (fetch-Var expr) mem))
    ((isOp? expr)
      (apply-Op
        (fetch-Op expr)
        (eval* (fetch-Args expr) env)))))
(define (eval* expr* env)
  (if (null? expr*)
    '()
    (cons (eval (car expr*) env)
          (eval* (cdr expr*) env))))
```

Here `expr` is the available representation of the interpreted expression, `mem` is a delayed memory state. In the monovariant binding time analysis, we have to consider the value of the `eval` function as always delayed, since in the `isVar?` clause it is assigned the value of the delayed mapping `mem`. Hence, interpreting the expression `(x + 3) * (7 - 2)` yields the deferred code

```
(apply-Op '*
  (cons
    (apply-Op '+
      (cons
        (lookup 'x env)
        (cons 3 '())))
    (cons
      (apply-op '-
        (cons 7 (cons 2 '())))
      '())))
```

Our problem is that the second addition is not executed even though both arguments 7 and 2 are known. Thus, such a mixed interpreter does not allow for performing permissible computations, which is essentially its purpose.

It's possible to automatically transform the `eval` function to the following form. The `dbt-eval` function is used to determine the availability of the expression and the `eval-static` function to calculate the fully accessible expression [5]:

```
(define (eval expr env)
  (cond
    ((isCst? expr) (fetch-Cst expr))
    ((isVar? expr) (lookup (fetch-Var expr) env))
    ((isOp? expr)
      (apply-Op
        (fetch-Op expr)
        (if (eq? (dbt-eval* (fetch-Args expr))
            'static)
            (static-eval* (fetch-Args expr))
```

```
          (eval* (fetch-Argse) env))))))
```

The `dbt-eval` *function* annotates expression as static or dynamic:

```
(define (dbt-eval expr)
  (cond
    ((isCst? expr) 'static)
    ((isVar? expr) 'dynamic)
    ((isOp? expr)
      (bt-apply-Op
      'static
      (dbt-eval* (fetch-Argu expr))))))
```

The `static-eval` *function* evaluates an expression that is known to be static:

```
(define (static-eval expr)
  (cond
    ((isCst? expr) (fetch-Cst expr))
    ((isVar7 expr) (lookup (fetch-Var expr) env))
    ((isOp? expr)
      (apply-Op
        (fetch-Op expr)
        (static-eval* (fetch-Argue))))))
```

The `eval*` *function* performs binding time analysis on a list of expressions `expr*` and applies `static-eval` where possible to reduce expressions.

```
(define (eval* expr* env)
  (if (null? expr*)
    '()
    (cons
      (if (eq? (dbt-eval (car expr*)) 'static)
        (static-eval (car expr*))
        (eval (car expr*) env)
      (if (eq? (dbt-eval* (cdr expr*)) 'static)
        (static-eval* (cdr expr*))
        (eval* (cdr expr*) env))))
```

The `dbt-eval*` *function* annotates a list of expressions `expr*`, marking the empty list as static:

```
(define (dbt-eval* expr*)
  (if (null? expr*)
  'static
  (bt-cons (dbt-eval (car expr*))
  (dbt-eval* (cdr expr*)))))
```

Finally, the `static-eval*` *function* applies static evaluation on a list of expressions `expr*`:

```
(define (static-eval* expr*)
```

```
    (if (null? expr*)
    '()
    (cons  (static-eval (car expr*))
           (static-eval* (cdr expr*)))))
```

While we have a larger code as a result, and the `eval` function performs checks that do not affect the result, the `dbt-eval` and `eval-static` functions are fully ground and reduced during the specialization stage, resulting in a better object code for the same example:

```
(apply-Op '*
    (cons
        (apply-Op '+
            (cons
                (lookup 'x env)
                '(3)))
        '(5)))
```

Already it is evident that automatic binding time analysis can be quite challenging and suboptimal. For programs more complex than a calculator, it might indeed be more advantageous to give the binding time annotation power to the programmer, as it is done in staged miniKanren.

3.4 Different types of partial evaluation

Programs can be specialized using online and offline partial evaluation. With online partial evaluation, we decide to reduce an expression and perform the reduction at the same time. The offline technique involves annotating code using binding-time analysis, and performing the program transformations later [6]. The accuracy of BTA depends on whether it is monovariant and polyvariant: the former assigns terms the safest binding time, or the join of all binding times involved, which may result in a loss of data, while the latter is more precise and flexible. Finally, there is manual binding-time analysis, used by evaluators such as Pink [7], which allows one to annotate programs by hand. Manual binding time analysis is used in Staged miniKanren to lift unifications by hand.

3.5 Lightweight Modular Staging - an alternative staging method

Quasi-quotation is commonly used to distinguish between the parts of the computation that need to be performed now and those that are to be deferred to later stages. Lightweight modular staging [8] uses an alternative approach where types represent different binding times. Thus, for a type `T`, we write `Rep[T]` to denote a *representation* of `T`. A term of type `Rep[T]` is not computed yet, however we know that the result of the future computation will have type `T`. An advantage of this approach is easier staging of practical algorithms such as the Fast Fourier Transform.

Another use case is specializing matrix multiplication to a matrix, where the result of the staging depends on how dense the matrix is. In particular, assume the matrix is known and the vector is unknown. Suppose the matrix is all zeros. Then instead of iterating through every row we can just have our specialized program return zero. Now suppose only the first row is non-zero. Then our specialized program still does not need to iterate through every row, as we can simply left-multiply the vector by the first row. On the other hand, if most entries in the matrix are non-zero, such a simplification might not be advantageous. This kind of specialization (multiplication with a known matrix and an unknown vector) is applicable in Markov models. Note that this process can be optimized using other staging systems as well.

3.6 Using supercompilation techniques for miniKanren.

One possible direction for staging is supercompilation, which is a method for creating an efficient residual program given partial input. While supercompilation includes partial evaluation, it involves a deeper transformation of the original program [9]. The basic idea is as follows. We view a program as a computing machine with certain states, or stages, and configurations, or sets of states. We have a basic language for states and an extended language for representing states. Both states and stages are described by expressions. These can be *ground*, if they describe precise states, or *nonground*, if they describe states representing precise states. A supercompiler then analyses computation histories and attempts to generalize them via the extended language.

Supercompilation involves a transition from the basic system to a metasystem whose object of study is the basic system. A *metacode* is a mapping M from general expressions in the extended language to expressions in the metasystem having two properties:

1. M is homomorphic, in the sense that $M(E_1E_2) = M(E_1)M(E_2)$ where E_1 and E_2 are concatenated expressions.

2. M is injective, meaning no two expressions map to the same object expression.

Metacoded E is denoted $M(E) = \mu E$ and, since M is injective, we can *demetacode* an expression, written $\mu^{-1}E$.

The generalized history is represented by a graph of states and transitions. The goal is then to perform transformations on the graph, such as reducing configurations, generalizing two configurations, and constructing subgraphs. A *strategy* is an algorithm for deciding which transformation needs to be taken at every moment during the creation of the graph. The possibility of applying supercompilation techniques in miniKanren has been explored in [10].

References

[1] Kyle Dewey, Jared Roesch, and Ben Hardekopf. Fuzzing the rust typechecker using clp (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 482–493. IEEE, 2015.

[2] William E Byrd. Relational programming in minikanren: techniques, applications, and implementations. 2010.

[3] Hartley Rogers Jr. Theory of recursive functions and effective computability. 1987.

[4] Yoshihiko Futamura. Partial computation of programs. In *RIMS Symposia on Software Science and Engineering*, pages 1–35. Springer, 1983.

[5] Mikhail A Bulyonkov. Extracting polyvariant binding time analysis from polyvariant specializer. In *Proceedings of the 1993 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 59–65, 1993.

[6] Niels H Christensen and Robert Glück. Offline partial evaluation can be as accurate as online partial evaluation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(1):191–220, 2004.

[7] Nada Amin and Tiark Rompf. Collapsing towers of interpreters. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–33, 2017.

[8] Tiark Rompf. *Lightweight modular staging and embedded compilers: Abstraction without regret for high-level high-performance programming.* PhD thesis, Citeseer, 2012.

[9] Valentin F Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(3):292–325, 1986.

[10] Maria Kuklina and Ekaterina Verbitskaia. Supercompilation strategies for minikanren.

[11] A.P. Ershov. Mixed computation: potential applications and problems for study. *Theoretical Computer Science*, 18(1):41 – 67, 1982.

[12] Michele Bugliesi, Evelina Lamma, and Paola Mello. Partial deduction for structured logic programming. *The Journal of Logic Programming*, 16(1-2):89–122, 1993.

[13] Andrew M Pitts. Nominal logic, a first order theory of names and binding. *Information and computation*, 186(2):165–193, 2003.