Lifted inference: normalizing loops by evaluation

Oleg Kiselyov
FNMOC
Monterey, CA, USA
oleg@pobox.com

Chung-chieh Shan
Rutgers University
Piscataway, NJ, USA
ccshan@cs.rutgers.edu

Abstract

Many loops in probabilistic inference map almost every individual in their domain to the same result. Running such loops symbolically takes time sublinear in the domain size. Using normalization by evaluation with first-class delimited continuations, we lift *inference procedures to reap this speed-up without interpretive overhead. To express nested loops, we use multiple control delimiters for metacircular interpretation. To express loops over a powerset domain, we convert nested loops over a subset to unnested loops.*

1. Introduction

Probabilistic inference is a popular and powerful way to make decisions under uncertainty in AI [19]. The basic approach is to specify a probability distribution and devise procedures to answer queries on the distribution, such as to compute or approximate the expected value of a random variable. The typical query concerns a *conditional* distribution: the condition restricts the query to those possible worlds that match the evidence we have observed and the action we are planning. For example, having specified a probability distribution over how people transmit diseases, we might then compute the risk of my being infected with swine flu, conditional on known diagnoses and my plan to take a stroll.

Many distributions used for realistic decision-making concern numerous individuals (such as people, diseases, vehicles, or discourse referents) yet place few specific conditions on these individuals. For example, there are numerous people in the world, yet few are known to have or not have a disease. To express such distributions compactly and answer queries on them efficiently, AI researchers have moved from propositional to first-order representations and devised algorithms to match [10]. In particular, *lifted* inference algorithms take advantage of logic variables to avoid repeated computation [20, 4, 17, 22].

Our goal in this paper is to lift inference *automatically*. To establish this goal, the rest of this introduction first uses an example to explain lifted inference.

The examples of distributions in this paper are all finite. A finite distribution is a finite set of possible worlds, each assigned a weight. (The weights should sum up to one, but we need not worry about that.) As in modal logic, different propositions – known as Boolean random variables – may hold in each world. For example, to study a certain disease, we may postulate the Boolean random variables I, whether the disease is infectious, and $S(x)$, whether the person x has the disease. There are as many variables of the first-order form $S(x)$ as there are individuals, say n. Because any assignment of Boolean values to these $n+1$ variables is in principle possible, our distribution contains 2^{n+1} worlds, one for each assignment.

A typical way to define a distribution is to specify the weight of a world as a product of *factors*. Each factor is a function of the values of a few random variables. In the example, we may specify that the weight of a world is the product of the following $2n$ factors: '4 if $I \wedge \neg S(x)$' for each x (because it is more usual to not have an infectious disease than to have it) and '9 if $\neg I \wedge \neg S(x)$' for each x (because it is *much* more usual to not have a *non*-infectious disease than to have it).

The probability $\Pr(A)$ of a proposition A is the total weight of the worlds where A holds. For example, if we care about Bob, we might want to find $\Pr(S(b))$, the probability that Bob is sick. By definition, it is the total weight of the worlds where $S(b)$ holds. More often, however, we want our probabilistic deliberations to take some observed evidence into account. For example, suppose we know that Alice is healthy. We might then want to find the *conditional* probability that Bob is sick *given* that Alice is healthy:

$$\Pr(S(b) \mid \neg S(a)) = \frac{\Pr(S(b) \wedge \neg S(a))}{\Pr(\neg S(a))}. \qquad (1)$$

In words, this ratio is the proportional weight of the worlds where Bob is sick among the worlds where Alice is healthy.

Conditions imposed by observed evidence, such as $\neg S(a)$, are typically specified as additional factors by which to scale the worlds' weights. In the example, knowing that Alice is healthy, we can add the factor '0 if $S(a)$'. To find the conditional probability (1), it is standard to compute the numerator $\Pr(S(b) \wedge \neg S(a))$ as well as $\Pr(\neg S(b) \wedge \neg S(a))$; the sum of the two probabilities is the denominator. Formally, we want to compute the number

$$\sum_{I} \sum_{\substack{S(z) \\ \text{for each } z \neq b}} \Big(\big(\textstyle\prod_{I \wedge \neg S(x)} 4\big) \big(\textstyle\prod_{\neg I \wedge \neg S(x)} 9\big) \big(\textstyle\prod_{S(a)} 0\big) \Big) \quad (2)$$

for each of the 2 possible values of $S(b)$, true and false. The product above is the weight of a world; the sum turns it into a probability. Each $\prod$ loops over propositions that hold; for example, the product $\prod_{I \wedge \neg S(x)} 4$ is 1 if I is false and 4^m if I is true, where m is the number of healthy people. Because we only care about (the conditional probability of) the variable $S(b)$, we sum over assignments to all other variables: the first $\sum$ loops over the 2 possible values of I; the second $\sum$ loops over the 2^{n-1} possible values of $S(z)$ for every z except b (so the product is not in the scope of z). In other words, we project away all variables but $S(b)$.

If we compute (2) by naively enumerating worlds, it would take $O(2^n)$ time due to the second $\sum$. Using the well-known technique of *variable elimination* [6], we can instead compute (2) by executing the query plan

$$\begin{aligned} \sum_{I} & \Big(\big(\textstyle\prod_{I \wedge \neg S(b)} 4\big) \big(\textstyle\prod_{\neg I \wedge \neg S(b)} 9\big) \Big) \\ & \Big(\sum_{S(a)} \big(\textstyle\prod_{I \wedge \neg S(a)} 4\big) \big(\textstyle\prod_{\neg I \wedge \neg S(a)} 9\big) \big(\textstyle\prod_{S(a)} 0\big) \Big) \\ & \Big(\prod_{x \notin \{a,b\}} \sum_{S(x)} \big(\textstyle\prod_{I \wedge \neg S(x)} 4\big) \big(\textstyle\prod_{\neg I \wedge \neg S(x)} 9\big) \Big). \end{aligned} \quad (3)$$

In this formula, each $\sum$ loops only over the 2 possible values that a single Boolean random variable can take, not the 2^{n-1} possible values that $n-1$ Boolean random variables can take. Therefore, this query plan runs much faster, yet distributivity guarantees that the result is the same.

This technique achieves exact inference exponentially faster than naively enumerating worlds, but there is plenty of room for improvement: because basic variable elimination operates over ground propositions only and does not explicitly represent logic variables such as x, it executes the last line in this query plan literally by repeating $n-2$ times the same computation $\sum_{S(x)} \big(\prod_{I \wedge \neg S(x)} 4\big) \big(\prod_{\neg I \wedge \neg S(x)} 9\big)$. In contrast, lifted variable elimination [20, 4] operates over first-order propositions and explicitly represents logic variables such as x, so it executes that computation just once then raises the result to the $(n-2)$-th power.

In general, lifted inference gains efficiency by exploiting the symmetry in a distribution, because a realistic distribution often combines a uniform model of a large population with sparse conditions on specific individuals like Alice and Bob. This idea seems general, yet only specific inference methods (variable elimination and *belief propagation* [22]) have had lifted variants devised. It is thus natural to ask, can an arbitrary inference procedure be lifted automatically?

As a first affirmative answer to this question, we show in this paper how an inference procedure expressed in terms of a general loop primitive can be lifted automatically using normalization by evaluation (NBE) with first-class delimited continuations. We contend that the abstract interpretation of loops is the key to lifted inference.

The next section describes two parts of the problem – transforming the loop body into the form of a top-level switch and taking advantage of this representation in iterating. On a simple example of a single loop we explain how to automatically convert, or reify, its body from a function over the loop variable to a switch data structure. Section 3 extends the reification to nested loops, revealing its metacircularity. Section 4 parameterizes reification over the type of the loop domain. Section 5 handles loops over tuples and over all subsets of the domain. We then conclude. We have already reviewed the related work on lifted inference; related work on NBE is discussed in the text. We have implemented our reification algorithms in OCaml and quote code fragments in the paper. The complete code is available online at `http://okmij.org/ftp/lift-reduce/`.

2. Symbolic execution of MapReduce loops

In this section, we show how to perform lifted inference using a technique for speeding up MapReduce loops. Whereas the original work on MapReduce [5] focused on parallel and distributed computation, we exploit the structure of a certain class of loop bodies – those that treat most elements of the domain uniformly. In §2.1, we represent this class of loop bodies so that MapReduce can take advantage of their structure and achieve extraordinary speed-ups. In §2.2, we use NBE to generate this efficient representation automatically from loop bodies written as ordinary OCaml functions.

We favor clarity over generality in this expository section. For simplicity, we assume that loops are not nested and that they range implicitly over a fixed domain of 100 individuals. Nested loops are described in §3; in §4 the loop variables are no longer restricted to a fixed type; and §5 deals with a domain other than that of individuals.

2.1. MapReduce in sublinear time

This section explains how to represent loops so that they can be evaluated in sublinear time.

Although the example in §1 only uses sum and product loops, other inference tasks and methods require different

aggregate operations. To abstract over the variety of aggregate operations, we require that all loops be expressed in terms of MapReduce [5]. A MapReduce loop specifies

- the domain to iterate over (for now, simply the set of all 100 individuals);
- the body, a function that *maps* individuals to results;
- the *reduction* operator, a commutative and associative binary function that combines two results into one; and
- the result when the domain is empty.

The empty result should be the identity of the reduction operator. The last two items specify a *commutative monoid*; therefore, we group them in the following data structure:

```
type 'r monoid = {union : 'r -> 'r -> 'r;
                  empty : 'r}
```

For example, addition on integers forms a monoid.

```
let sum_monoid = {union = (+); empty = 0}
```

The result of the loop, as computed by MapReduce, is defined by the following function.

```
let map_reduce_gen :
  'i domain -> 'r monoid -> ('i -> 'r) -> 'r
 = fun domain monoid f ->
    fold_left
     (fun acc x -> monoid.union (f x) acc)
    monoid.empty domain
```

In words, MapReduce maps the loop body `f` over all individuals of the `domain` and reduces the results using the `monoid`. A sum loop is a MapReduce loop with `sum_monoid`. For example,

```
map_reduce_gen people sum_monoid
  (fun x -> 1 + if x = "alice" then 0 else 1)
```

We assume a domain `people` of 100 individuals identified by distinct names: `"alice"`, `"bob"`, `"carol"`, ..., `"äijö"`. The sum loop above produces the result 199.

Whether or not we evaluate MapReduce loops using parallel or distributed computing, we can apply the following optimization. The body of the sum loop above maps every one of the 100 individuals to the integer 2, except `alice`. Since addition is associative and commutative, we rearrange the loop to add the result for `alice` to the sum of 99 copies of the default result 2, giving us $1+99\times 2=199$.

In this example, because the reduction operator is just addition on integers, we can easily use multiplication to reduce multiple copies of the same result. In general, as long as reduction is associative as we require, we can reduce m copies of a result r to a single result, notated mr, using $O(\log m)$ reductions [15, §4.6.3]. In particular, the familiar 'fast power' algorithm generalizes to an arbitrary monoid:

```
let rec repeat monoid r = function
  | 0 -> monoid.empty
  | n when n mod 2 = 0 ->
      repeat monoid (monoid.union r r) (n/2)
  | n ->
      monoid.union r (repeat monoid r (n-1))
```

Thus, we evaluate the sum loop in *sublinear time* and avoid enumerating the majority of the domain. This optimization is the essence of lifted inference and the core of this paper.

Our goal, thus, is to evaluate MapReduce loops in a way that automatically discovers if the loop body maps most individuals in the domain to the same value, and that uses repeated reduction to evaluate such a loop in sublinear time. To achieve this goal, we have to represent the additional structure of the loop body – that it maps all individuals, except for a select few, to a single default value. We introduce the *switch* data structure:

```
type 'r switch
  = Default of 'r
  | Case of indiv * 'r * 'r switch

type indiv = string
```

The meaning of a switch (of type `'r switch`) is a mapping from individuals to results (of type `'r`). The switch represents the mapping essentially as a list of excepted individuals along with their particular results, terminated with the default result. (The list of 'cases' with the 'default' at the end is reminiscent of the 'switch' statement in C.) We can convert a switch to the mapping it represents:

```
let rec reflect : 'r switch -> (indiv -> 'r)
  = function
  | Default r -> fun _ -> r
  | Case (x,r,fsw) -> fun x' ->
      if x = x' then r else reflect fsw x'
```

Thus, if `fsw` is the switch `Case ("alice",1, Default 2)`, then `reflect fsw "alice"` is 1 whereas `reflect fsw "bob"` is 2.

By the way, it is straightforward to add laziness to the switch data structure, so that results and cases are computed on demand. We omit this trivial optimization in this paper.

To take advantage of the representation of the loop body as a switch, we rewrite MapReduce as follows:

```
let map_reduce_sw monoid fsw =
  let rec loop card = function
    | Default r -> repeat monoid r card
    | Case (_,r,tail) ->
        monoid.union r (loop (card-1) tail)
  in loop 100 fsw
```

The type of this function is `'r monoid -> 'r switch -> 'r`. Instead of enumerating every individual in the domain as `map_reduce_gen` does (which takes linear time), the new code processes the default case using repeated reductions, then adds the particular results one by one. For simplicity, we assume that the switch never excepts any individual twice, nor any individual outside the domain; all switches we produce have this property. Therefore, we only need to calculate the cardinality of the domain without the

excepted individuals. We use the latter number for repeated reductions. We can evaluate our example sum loop faster:

```
map_reduce_sw sum_monoid
  (Case ("alice",1,Default 2))
```

Likewise, we speed up the query plan (3): the outermost product (everything except $\sum_I$) can be specified as 2 special cases for a, b along with the result for all other individuals. It is easy to evaluate such a loop in sublinear time, because the default result can be computed once and reduced repeatedly. That would accomplish our goal of lifted inference.

2.2. Reifying a function to a switch

In the rest of this paper, we use NBE to make it convenient to represent loop bodies efficiently as switches. More specifically, we generate switches from ordinary (and possibly separately compiled) OCaml functions.

In principle, we can write switches by hand. Our trivial sum loop above already shows the drawback of such hand coding. To convert the loop body in functional form `fun x -> 1 + if x = "alice" then 0 else 1` to a switch, we have to manually lift the `if` statement and repeat the increment operation. The algorithm of the loop body is no longer perspicuous in the resulting switch. In our introductory example (3), the special cases a and b of the product loop arise for different reasons – we *observe* Alice's health and we *care* about Bob – perhaps from different modules in a large AI program. It is a tedious and error-prone chore to convert this product to a switch by collecting the special cases manually. This chore becomes unbearable as we move to nested loops in §3.

A better approach is to design a domain-specific language (DSL) for loop bodies, embed it in OCaml, and write an interpreter that evaluates a DSL expression to a switch. A DSL can be embedded into OCaml *initially* and *deeply*, by defining a data structure for the DSL's abstract syntax tree, or *finally* and *shallowly*, by defining a function for each expression form of the DSL [12, 18, 1]. Embedded in these two ways, the body of our trivial sum loop would look like

```
Add (Int 1) (If  (Equ (Var "x") (Ind "alice"))
                 (Int 0) (Int 1))
add (int 1) (if_ (equ (var "x") (ind "alice"))
                 (int 0) (int 1))
```

respectively. In this paper, we follow the NBE strategy of taking [13] the final embedding to its extreme, a *very shallow* embedding in which the DSL extends the host language OCaml. For example, we write the body of our sum loop as

```
1 + if equ x (Val "alice") then 0 else 1
```

using the native OCaml conditional operator and integer literals. In fact, the loop body can be any OCaml code and use any compiled OCaml library. We reuse OCaml's implementation rather than writing our own interpreter. Another advantage, perhaps the most important one, is discussed in §3.

We want to turn a loop body from a function to a switch. If we use the initial or final DSL embedding, we would write a partial evaluator that takes the loop variable as the dynamic input and computes a switch as a residual program. In the very shallow embedding, the loop body is a regular OCaml function that takes the loop variable as argument. We use NBE to partially evaluate this function without writing a full partial evaluator. In particular, since the loop variable ranges over a sum type of individuals, we perform NBE using first-class delimited continuations [16, 2].

Informally, we apply the function to a distinguished value of the loop variable. Imagine that this application happens in a debugger, so that a breakpoint is triggered when the distinguished value is used, such as compared to a constant. The debugger can report the constant used in the comparison and resume from the breakpoint in two ways: either by assuming that the loop variable is equal to the constant or by assuming that it is not equal. The resumed function may trigger another breakpoint by using the distinguished value again. A transcript of the debugging session is a search tree of alternative resumptions that describes how the loop body uses the loop variable and hence reifies [9, 7] the body as a data structure. If the only operations on the loop variable are equality comparisons, then the tree is actually a switch.

In the rest of the section, we turn this outline into code. The outline poses three problems: producing a distinguished value for the loop variable; triggering a breakpoint when this value is used; handling breakpoints from within the program itself. We solve the first problem by extending the domain of the loop variable. The body of a loop over individuals should take an argument of the sum type `var`:

```
type var = Val of indiv | Var
```

A `var` is either a concrete individual such as `Val "alice"` or an abstract individual `Var`, our distinguished value. (Because this section deals with unnested loops only, there can be only one loop variable in scope and we need only one distinguished value.)

We need to trigger a breakpoint whenever the abstract individual is used. That is easy with a bit of cooperation from the programmer of the loop body, who should use custom operations to deal with the loop variable. (The cooperation is not required if we can overload the operations on the type `var`, for example, with Haskell-like type classes.) For example, to compare the loop variable `x` with `alice`, the programmer should write `equ x (Val "alice")` rather than `x = "alice"`. The function `equ` is our custom equality comparison. It returns a regular OCaml Boolean, ready for `if` to test. It is inconvenient to have to use these custom operations, but the programmer cannot forget to use them because the type checker would complain about `x = "alice"`, that `"alice"` has the type `indiv` and `x` has the different type `var`. The particular type `var` for the loop variable also lets us control how the loop variable may be

used and hence what constraints on it may be imposed. In this paper, we only deal with equality comparisons and equality constraints, so we provide a single primitive operation `equ` on arguments of type `var`. We are working on generalizing to inequality or arithmetic constraints.

To debug the body, we use the *delimited control operators* `shift` and `reset` [3]. The function `reset` takes a thunk, runs it, and, if no breakpoint is hit, returns its result. For example, `reset (fun () -> 42)` evaluates to 42. To trigger a breakpoint, we use the function `shift`: evaluating `shift (fun k -> exp)` interrupts the running thunk and makes the nearest dynamically enclosing `reset` return the result of the expression *exp*. For example, the expression

```
reset (fun () -> 42 + shift (fun k -> 1 + 2))
```

evaluates to 3. The argument `k` is bound to the *delimited continuation*, a function that resumes the computation from the breakpoint. To tell if the thunk hits a breakpoint or finishes normally, it is common to define a sum data type like

```
type 'r req
  = Done of 'r
  | Compare of var * var * (bool -> 'r req)
```

so that, if `reset (fun () -> Done exp)` returns the value `Done r`, then `r` is the final result of *exp*. (The name `req` is short for 'request'.) Using `shift`, we define our custom equality predicate `equ` to trigger a breakpoint:

```
let equ (x : var) (y : var) : bool =
  shift (fun k -> Compare (x,y,k))
```

Evaluating `equ x y` makes the nearest dynamically enclosing `reset` return the value `Compare (x,y,k)`. The first two components are the arguments passed to `equ`. The last component of `Compare` is a function that can be applied to a Boolean `b` to resume from the call to `equ` with the comparison result `b`. Because the resumed computation may hit another breakpoint before finishing normally, the type of this last component recursively refers to `'r req`.

We are all set to normalize loop bodies from functions to switches. We define the function `reify`:

```
let reify (f : var -> 'r) : 'r switch =
  let var = Var in
  let rec loop_known w = function
    | Done r -> r
    | Compare (Var,Var,k) ->
        loop_known w (k true)
    | Compare (Var,Val y,k) ->
        loop_known w (k (y = w))
    | Compare (Val y,Var,k) ->
        loop_known w (k (y = w))
    | Compare (Val x,Val y,k) ->
        loop_known w (k (x = y)) in
  let rec loop ws = function
    | Done r -> Default r
    | Compare (Var,Var,k) ->
        loop ws (k true)
    | Compare (Var,Val y,k) ->
        make_case ws k y
    | Compare (Val y,Var,k) ->
        make_case ws k y
    | Compare (Val x,Val y,k) ->
        loop ws (k (x = y))
  and make_case ws k y =
    if List.mem y ws then loop ws (k false)
    else Case (y,loop_known y (k true),
               loop (y::ws) (k false))
  in loop [] (reset (fun () -> Done (f var)))
```

As its type indicates, `reify` takes a function mapping the loop variable of type `var` to the result of type `'r`, and returns the corresponding switch. Our reification algorithm is summarized in the last line of the code: we use `reset` to debug the application of the loop body to the distinguished value `var` of the loop variable. The debugger reports all attempts to compare two values of type `var`.

At the heart of `reify` is the internal function `loop` of the type `indiv list -> 'r req -> 'r switch`. (Let us overlook the first argument for a moment.) The function is the 'debugging script' that *interacts* with the loop body being debugged. An occurrence of `equ x y` in the body asks a question, whether two values of type `var` are equal. This question triggers a breakpoint and makes `reset` return the value `Compare (x,y,k)`, which specifies the two values asked about and the function `k` for answering the question. The computation may produce more questions, to be answered in turn, so `loop` is recursive.

The 'debugging script' `loop` is a straightforward case analysis. If the debugger returns `Done r`, then the loop body is finished without using the loop variable, so we get the default result `r` of the body. If the loop body asks to compare two distinguished values, then we return the answer `true`: two loop variables must be equal because we assume that only one is in scope. To compare two concrete values `Val x` and `Val y`, we simply compare the individuals `x` and `y`. Admittedly, `equ` has enough information to handle these two cases on its own without triggering a breakpoint. We introduce this optimization in §4.

The two remaining cases of `loop` are the most interesting: they answer a question like `equ x (Val "alice")`, comparing the loop variable to a concrete value `y`. Such a question gives rise to a special `Case` of the generated switch. The `Case` maps the individual `y` to a particular result, which we compute by answer the question with `true`. To compute the rest of the switch, we answer the question with `false`.

We have overlooked so far the first argument to `loop`. If the loop body asks the same question twice, we must give consistent answers. The first argument to `loop` is the list of individuals for which `loop` has already received equality questions and answered in the negative. This list is a memo

table of sorts: when the same question comes again, `loop` answers `false` right away.

The function `loop_known` is a variant of `loop` for when we answer `true` to an equality comparison between the loop variable and a concrete individual `w`. Because equality is transitive, any further comparison between the loop variable and a concrete individual `y` can only be answered by simply comparing `y` to `w`. Thus `loop_known w` computes the result of the loop body on `w`. From the point of view of partial evaluation, `loop_known` propagates positive information resulting from equality comparison whereas `loop` propagates negative information [24].

We have achieved both the goal of reifying a MapReduce loop body into a switch data structure and the goal of making MapReduce take advantage of that structure to run loops in sublinear time. We finish this section with a complete example. We first define membership testing in a list, using our custom `equ` predicate:

```
let rec member x = function
  | [] -> false
  | (h::t) -> equ x h || member x t
```

We can now very quickly evaluate the loop

```
let xs = [Val "alice"; Val "bob"; Val "alice"]
in map_reduce_sw sum_monoid
    (reify (fun x ->
     (if member x xs then 0 else 1) *
     (if equ x (Val "alice") then 4 else 1) *
     (if equ x (Val "carol") then 0 else 1)))
```

yielding 97. In general, we can speed up the loop

```
map_reduce_gen people monoid
  (fun x -> f (Val x))
```

by replacing it with `map_reduce monoid (reify f)`.

The rest of the paper generalizes the achieved result removing the simplifying assumptions of unnested loops and of the fixed, atomic type of the loop domain.

3. Metacircular interpretation for nested loops

In this section, we generalize our MapReduce speed-up technique to nested loops. This generalization is crucial for probabilistic inference. It requires and reveals that the NBE underlying our reification technique is metacircular. In contrast, the generalization would be far less straightforward using the two other methods briefly considered in §2.2 for embedding a DSL for MapReduce loops.

The assumption that only one loop variable is in scope, built into `reify` in §2.2, implies that any two loop variables are equal to each other. This leads to spectacularly wrong results for nested loops. For example, the following

```
map_reduce_sw sum_monoid (reify (fun x ->
  map_reduce_sw sum_monoid (reify (fun y ->
    if equ x y then 0 else 1))))
```

evaluates to zero. More disturbing are subtly wrong results for nested loops even if we never compare two loop variables together:

```
map_reduce_sw sum_monoid (reify (fun x ->
  map_reduce_sw sum_monoid (reify (fun y ->
    if equ x (Val "alice") ||
       equ y (Val "bob")
    then 0 else 1))))
```

We expect the expression to yield $99 \times 99 = 9801$ but obtain only 9800. The problem occurs because the inner loop considers the outer loop variable x to be identical to the inner loop variable y. Such a confusion of variables occurs when nesting other naive NBE procedures such as naive automatic differentiation [23].

To tell loop variables apart, we need an infinite supply of distinguished values of the loop variable. Whenever we are about to reify a new loop body, we should create a unique distinguished value for its loop variable. We redefine the type `var` as follows:

```
type var = Val of indiv | Var of unit ref
```

Since reference cells are generative, the expression `ref ()` creates a unique value that is physically equal (in the sense of `(==)`) only to itself.

We have yet to deal with a tougher problem: by definition, loop nesting means that a loop body – an argument to `reify` – contains `reify` itself. We need to debug a debugger. The outer debugger's breakpoints may appear in the inner debugger and in the program it debugs. The inner debugger should distinguish its own breakpoints from those of the outer debugger [8]. In other words, our DSL for writing loop bodies should be expressive enough to write the function `reify` in it – it has to be amenable to self-interpretation. Writing a self-interpreter in a typed language is a remarkably difficult problem [21]. The very shallow encoding saves the day: our DSL is OCaml itself with the library function `equ` to compare loop variables. Thus, to make `reify` nestable – or, metacircular – we just have to replace its calls to `(=)` by calls to `equ` whenever we compare values that may be outer loop variables. Since loop variables have a distinguished type `var`, such a replacement is easy, being type-driven. Below is the new code for `reify`. (We elide `loop_known`, which is a mere variation of `loop`.)

```
let reify (f : var -> 'r) : 'r switch =
  let var = Var (ref ()) in
  let rec loop_known w = function ... in
  let rec loop ws = function
    | Done r -> Default r
    | Compare (x,y,k) when x == var &&
                           y == var ->
        loop ws (k true)
    | Compare (x,y,k) when x == var ->
        make_case ws k y
    | Compare (y,x,k) when x == var ->
```

```
      make_case ws k y
  | Compare (x,y,k) ->
      loop ws (k (equ x y))
and make_case ws k y =
  if member y ws then loop ws (k false)
  else Case (y,loop_known y (k true),
             loop (y::ws) (k false))
in loop [] (reset (fun () -> Done (f var)))
```

The type of the first argument to `loop` changed from `indiv list` to `var list`, so we use the function `member` defined at the end of §2.2 to test membership. Instead of assuming that any `Var` is our loop variable, we check if something is our loop variable by comparing it physically against the `var` we created. This way, we treat any variables of outer loops as concrete individuals. In particular, a special `Case` in the generated switch may contain an outer loop variable as its first component `y`. Therefore, we need to redefine the `switch` data type, so that the first component of `Case` is of the type `var`:

```
type 'r switch
  = Default of 'r
  | Case of var * 'r * 'r switch
```

This `reify` delegates the questions that do not concern its own loop variable to some parent `reify`. So, we need an outermost 'debugging script' to answer questions about actually concrete individuals:

```
let top thunk =
  let rec loop = function
    | Done r -> r
    | Compare (x,y,k) -> loop (k (x = y))
  in loop (reset (fun () -> Done (thunk ())))
```

All our MapReduce computations must be within a thunk executed by `top`.

We finish the section with a complete example of evaluating a nested loop. We count the ordered pairs (x,y) of individuals such that $x \neq$ `alice`, $y \neq$ `bob`, and $x \neq y$.

```
top (fun () ->
map_reduce_sw sum_monoid (reify (fun x ->
  map_reduce_sw sum_monoid (reify (fun y ->
    if equ x (Val "alice") ||
       equ y (Val "bob") || equ x y
    then 0 else 1)))))
```

This expression evaluates to the desired result 9703. We can glean how this counting works by removing the outer `map_reduce_sw sum_monoid` so to compute the result of reifying the outer loop:

```
Case (Val "alice", 0,
  Case (Val "bob", 99, Default 98))
```

Our loop bodies never directly compare `x` to `bob`, yet the outer `reify` correctly makes a special case for when `x` is `bob`! This special case arises from the use of `member` by the inner `reify` to decide whether $y \neq$ `bob` entails $x \neq y$. Multiple layers of interpreters thus combine to process the constraints and count their satisfying assignments, without converting them globally to a normal form [14].

4. Polymorphism and optimization

We have so far assumed that all loop variables range over the same type of individuals. In this section, we remove this limitation, so that loop variables may have any type `'i var`, parameterized over the type `'i` of the domain to iterate over. The `equ` predicate becomes a polymorphic function over `'i`, and `reify` now handles functions from any loop domain. This generalization turns out an optimization.

We start by describing an inefficiency of the nested loop reification in §3. The comparison `equ x (Val "alice")` of the loop variable `x` triggers a breakpoint handled by the nearest dynamically enclosing `reify`. If `x` turns out not to be the innermost loop variable, then this `reify` re-executes `equ x (Val "alice")` to trigger a breakpoint handled by an outer `reify`, which goes through the same moves. Thus, it may take many delegating steps to answer a comparison with an outer variable that occurs in an inner body. One may wish to instead send the question straight to the `reify` in charge of that variable (or, if two variables are compared, to the `reify` in charge of the inner of the two variables). For this purpose, a `shift` expression enclosed by several `reset`s needs to designate at run time a particular `reset` for which to trigger a breakpoint. Multi-prompt delimited continuations [11] give us exactly this ability. We can create arbitrarily many *prompts* to pass to `reset` and `shift` as an additional argument. The function `shift` delivers the breakpoint to the nearest `reset` with the same prompt, bypassing any intervening `reset`s with other prompts.

For `equ` to specify which `reify` to deliver the breakpoint to, we create a new prompt each time `reify` is called and store the prompt in the distinguished value for the corresponding loop variable. To compare two loop variables to see which one has inner scope, we also store in the distinguished value a generation counter that is incremented every time `reify` is called. Below are the new definitions of the types `var`, `req`, and `switch`.

```
type 'i var
  = Val of 'i
  | Var of generation * 'i req prompt
and 'i req = Compare of 'i var * (bool -> 'i req)
type generation = int
type ('i,'r) switch
  = Default of 'r
  | Case of 'i var * 'r * ('i,'r) switch
```

It turns out that sending each question straight to the prompt in charge relieves all calls to `reify` from having to handle questions of the same type, so we can and do parameterize the types `var`, `req`, and `switch` by the type `'i` of the loop

domain. Also, a `Compare` question only has two components now because it no longer needs to include the name of the inner variable to compare against.

We turn to the new `equ`. It used to be a one-liner, but now it has to determine to which `reify` to direct the comparison question and so has to analyze its arguments.

```
let equ (x : 'i var) (y : 'i var) : bool =
  match (x,y) with
  | (Var (gx,px), Var (gy,py)) ->
      if gx > gy then
        shift px (fun k -> Compare (y,k))
      else if gx < gy then
        shift py (fun k -> Compare (x,k))
      else true
  | (Val x, Val y) -> x = y
  | (Var (_,px), y) | (y, Var (_,px)) ->
      shift px (fun k -> Compare (y,k))
```

To compare a loop variable with a constant (the last line of the code), `equ` extracts the prompt `px` from the variable and passes it to `shift` to direct the question. To compare two loop variables, `equ` compares the counters `gx` and `gy` to decide which one is inner. Finally, to compare two concrete values, `equ` has no prompt to send the question to and performs the comparison itself using `(=)`. We gain a welcome optimization: `equ` now compares two concrete values without triggering any breakpoint, and we no longer need `top`.

The types of `reify` and `reflect` are now as follows.

```
reify   : ('i var -> 'r) -> ('i,'r) switch
reflect : ('i,'r) switch -> ('i var -> 'r)
```

Their complete code is available online. The final example of §3 runs as before (without the obsolete call to `top`).

5. Looping over a powerset domain

As illustrated in §1, probabilistic inference often calls for looping (especially summing) over Boolean assignments to $\Omega(n)$ random variables at once. In the opening example, we used distributivity in variable elimination to avoid looping over the exponentially many assignments. However, this strategy is not available when the individuals in the domain interact more tightly. For example, suppose our distribution comprises factors that involve two individuals at once, such as '2 if $I \wedge S(x) \wedge S(y)$', for each x and y such that $x \neq y$. We want to find the probability that Bob is sick, which has the form

$$\sum_{I} \sum_{\substack{S(z) \\ \text{for each } z \neq b}} \left(\prod_{I \wedge S(x) \wedge S(y)} 2\right) \cdots . \tag{4}$$

Unlike in (2) and (3), distributivity cannot turn the sum of products into a product of sums, because the product $\prod_{I \wedge S(x) \wedge S(y)} 2$ is a loop over *pairs* of individuals.

In general, many probabilistic models postulate local (especially pairwise) interactions among a large population of individuals. To evaluate such models, we need to loop over all *subsets* of the domain and – inside that loop's body – loop over *tuples* of individuals in the subset. If we only wanted to loop over tuples of individuals in the entire domain, then nested loops do the job nicely. We have already seen an example: the code at the end of §3 loops over ordered pairs (x,y) and distinguishes the special cases where $x =$ `alice`, $y =$ `bob`, or $x = y$. Adding an outer loop over subsets of the domain makes the computation much harder to perform tractably. The problem is reminiscent of NBE with higher-order functions and sum types. Counting arguments have been used formally to deal with some cases of the problem [4, 17].

Our approach is to convert a loop over tuples into a loop over individuals. That is, we provide a construction that turns every loop over tuples into an unnested loop over individuals, such that the result of the original loop can be read off easily from the result of the new loop. We detail this construction in §5.1, then show in §5.2 how to use it to answer queries like (4).

5.1. From loops over tuples to loops over individuals

We first illustrate our construction using the loop over pairs in §3. That loop counts the pairs (x,y) such that $x \neq$ `alice`, $y \neq$ `bob`, and $x \neq y$. Our goal is to express the same count using an unnested loop.

If we assume $x \neq y$, then we can express the body of this loop over tuples as a *meta-switch*, a switch on x – of type `int switch switch` – whose results are switches on y:

```
Case (Val "alice", Default 0,
Case (Val "bob"  , Default 1,
Default (Case (Val "bob", 0, Default 1))))
```

The second `Case` makes sense because if x is Bob then our assumption that $x \neq y$ guarantees the final result to be 1.

This meta-switch contains 3 switches. Consequently, our new loop, when applied over a set S of individuals, results in a record $(r; m_1, m_2, m_3; r_1, r_2, r_3)$ with 7 components:

- The first component r is the result we want – that of applying the original loop over $S \times S$. In particular, when S is empty, r is just 0, the empty result of the original loop. Also, when S is a singleton set $\{x\}$, r is just 0, the result of the original loop over $\{(x,x)\}$.
- The next 3 components m_1, m_2, m_3 are natural numbers that count how many elements of S are sent by the meta-switch to each of the 3 switches. That is, m_1 counts how many elements of S are `alice` (either 0 or 1), m_2 counts how many are `bob` (again either 0 or 1), and m_3 counts how many are neither.
- The last 3 components r_1, r_2, r_3 are the results of applying the 3 switches as loops over S. In this example, r_1 is always 0, r_2 is the cardinality of S, and r_3 is the number of elements of S that are not `bob`.

Thus, the empty result of the new loop is $(0;0,0,0;0,0,0)$. The new reduction operator is more interesting: given the results $(r;m_1,m_2,m_3;r_1,r_2,r_3)$ and $(r';m'_1,m'_2,m'_3;r'_1,r'_2,r'_3)$ for two sets S and S', it computes the combined result

$$(r+r'+m_1r'_1+m_2r'_2+m_3r'_3+m'_1r_1+m'_2r_2+m'_3r_3;\\ m_1+m'_1,m_2+m'_2,m_3+m'_3;r_1+r'_1,r_2+r'_2,r_3+r'_3) \quad (5)$$

for the disjoint union of S and S'. In the first component of this result, r is the result of the original loop over $S\times S$, r' is the result over $S'\times S'$, $m_1r'_1+m_2r'_2+m_3r'_3$ is the result over $S\times S'$, and $m'_1r_1+m'_2r_2+m'_3r_3$ is the result over $S'\times S$.

Following this description, the body of the new loop maps `alice` to $(0;1,0,0;0,1,1)$, `bob` to $(0;0,1,0;0,1,0)$, and all other individuals to $(0;0,0,1;0,1,1)$. Applying the loop over a domain of 100 individuals, including `alice` and `bob`, gives the result $(9703;1,1,98;0,100,99)$, whose first component 9703 is the result of the original loop, as desired.

We now turn to the general construction. We focus on looping over pairs, but the idea generalizes to tuples.

To begin, we reify the original loop body, which operates on a tuple, into a meta-switch. We do so using the `reify` function we already implemented, but with a twist: to generate the inner switches under the assumption that the components of the tuple are distinct, we generalize `reify` to a new function `reify_without`, which can be called with an argument list `ws` of individuals that the loop does not range over. The implementation of `reify_without` is just like `reify`, except the empty list `[]` on the last line is replaced by the new argument `ws`. If `f` is a loop body over pairs, a binary function of type `var -> var -> 'r`, then the following expression computes the meta-switch.

```
reify (fun x -> reify_without [x] (f x))
```

If the meta-switch contains k switches, then the new loop, when applied over a set S of individuals, results in a record $(r;m_1,\ldots,m_k;r_1,\ldots,r_k)$ with $1+2k$ components:

- The first component r is the result we actually want – that of applying the original loop over $S\times S$.
- The next k components $m_1,\ldots,m_k$ are natural numbers that count how many elements of S are sent by the meta-switch to each of the k switches.
- The last k components $r_1,\ldots,r_k$ are the result of applying the k switches as loops over S.

Thus, if r_0 is the empty result of the original loop, then the empty result of the new loop is $(r_0;0,\ldots,0;r_0,\ldots,r_0)$. The reduction operator combines $(r;m_1,\ldots,m_k;r_1,\ldots,r_k)$ and $(r';m'_1,\ldots',m'_k;r'_1,\ldots',r'_k)$ to form the result

$$(r\oplus r'\oplus m_1r'_1\oplus m_2r'_2\oplus m_3r'_3\oplus m'_1r_1\oplus m'_2r_2\oplus m'_3r_3;\\ m_1+m'_1,m_2+m'_2,m_3+m'_3;r_1\oplus r'_1,r_2\oplus r'_2,r_3\oplus r'_3). \quad (6)$$

Here $\oplus$ denotes the original reduction operator and juxtaposition denotes repeated reduction.

Following the description above, it is easy to implement a pair of functions `pairs` and `read_off` with the signature

```
type 'r pairs = 'r * (int * 'r) list
val pairs : 'r monoid * (var -> var -> 'r)
    -> 'r pairs monoid * (var -> 'r pairs)
val read_off : 'r pairs -> 'r
```

satisfying the basic law:

```
   map_reduce_gen dom monoid (fun x ->
    map_reduce_gen dom monoid (f x))
=  let (monoid',f') = pairs (monoid,f) in
   read_off (map_reduce_gen dom monoid' f')
```

5.2. From loops over individuals to loops over sets

By itself, the `pairs` construction only gives us a new way besides nested loops to loop over pairs of individuals in the entire domain. To loop over subsets of the domain, as demanded by the queries (2) and (4), we need to use distributivity as in §1. However, it is less obvious how to use distributivity to answer (4) than to answer (2), because the results produced by the new loop are no longer just numbers that can be added. Our final hurdle is to combine the new loop's results over different subsets of the domain.

Algebraically speaking, the query plan (3) uses distributivity in the commutative semiring of probabilities. To answer the more complex query (4), we need distributivity in a commutative semiring whose multiplication is the reduction operation produced by `pairs`. Because this multiplication is not just that on numbers, the corresponding addition cannot just be that on numbers. Instead, we extend a multiplicative monoid *freely* to a semiring, by taking formal sums of elements of the multiplicative monoid. In other words, we put multiple results produced by the new loop into a bag.

In our implementation, we represent a bag as a finite map from elements to natural numbers. For efficiency, we must detect duplicate elements and represent them by counting.

```
type 'r bag = ('r, int) PMap.t
```

Bags form a monad, that of nondeterminism.

```
val unit : 'r -> 'r bag
val bind : 'r bag -> ('r -> 's bag) -> 's bag
```

Equipped with the union operation, bags also form a commutative monoid – the additive monoid of the semiring.

```
val bag_add : 'r bag monoid
```

Furthermore, we can loop over the contents of a bag; in other words, we can homomorphically map a bag monoid to another monoid. This operation is efficient, using repeated reduction in the target monoid.

```
val reduce_bag : 'r monoid -> 'r bag -> 'r
```

Finally, any monoid structure on the bags' elements (such as produced by `pairs`) gives rise to another monoid structure on the bags – the multiplicative monoid of the semiring.

```
val bag_mul : 'r monoid -> 'r bag monoid
```

To demonstrate the use of these algebraic structures, we return to the example in §5.1. Provided that `int` is the type of arbitrary-precision integers, the following code computes the sum

$$\sum_S \prod_{x,y \in S,\, x \neq \texttt{alice},\, y \neq \texttt{bob},\, x \neq y} 3, \tag{7}$$

where S ranges over all subsets of the domain.

```
let f x y = if equ x (Val "alice") ||
               equ y (Val "bob") || equ x y
            then 0 else 1 in
let (m',f') = pairs (sum_monoid, f) in
let m'' = bag_mul m' in
reduce_bag sum_monoid
 (bind
   (map_reduce_sw m'' (reify (fun x ->
     bag_add.union m''.empty (unit (f' x)))))
   (fun r -> unit (power 3 (read_off r))))
```

6. Conclusions

We have applied NBE to make MapReduce loops run in sublinear time in case the loop bodies process most of the domain elements uniformly. NBE with first-class delimited continuations automatically reifies a loop body into a switch data structure, exposing the uniformity of iteration. Our technique applies to nested loops and to loops over tuples and over all subsets of the domain.

Our use of first-class delimited continuations to simulate symbolic execution with backtracking has the advantage that `reify` incurs no interpretive overhead. That is, any code that does not compare individuals runs just as fast inside `reify` as outside. This lack of a slowdown is especially helpful for probabilistic inference, which often involves heavy numerical computations.

Currently our technique only applies to loop bodies whose sole operation on loop variables is testing their equality with constants and each other. Many classes of probabilistic inference have loops of that form. We are working on other loop-variable constraints, such as numeric inequalities and ranges.

Acknowledgments We thank Rodrigo de Salvo Braz for posing the problem of lifted inference to us, and Dylan Thurston for discussing loops over a powerset domain.

References

[1] J. Carette, O. Kiselyov, and C.-c. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, in press.

[2] O. Danvy. Type-directed partial evaluation. In *POPL*, pages 242–257, 1996.

[3] O. Danvy and A. Filinski. A functional abstraction of typed contexts. Technical Report 89/12, DIKU, University of Copenhagen, 1989.

[4] R. de Salvo Braz, E. Amir, and D. Roth. Lifted first-order probabilistic inference. In Getoor and Taskar [10], pages 433–451.

[5] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[6] R. Dechter. Bucket elimination: A unifying framework for probabilistic inference. In M. I. Jordan, editor, *Learning in Graphical Models*. MIT Press, Cambridge, 1999.

[7] A. Filinski. Representing monads. In *POPL*, pages 446–457, 1994.

[8] A. Filinski. Representing layered monads. In *POPL*, pages 175–188, New York, 1999. ACM Press.

[9] D. P. Friedman and M. Wand. Reification: Reflection without metaphysics. In *Lisp and Functional Programming*, pages 348–355, 1984.

[10] L. Getoor and B. Taskar, editors. *Introduction to Statistical Relational Learning*. MIT Press, Cambridge, 2007.

[11] C. A. Gunter, D. Rémy, and J. G. Riecke. A generalization of exceptions and control in ML-like languages. In *FPCA*, pages 12–23, 1995.

[12] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4es):196, Dec. 1996.

[13] O. Kiselyov and C.-c. Shan. Embedded probabilistic programming. In *DSL Working Conference*, pages 360–384, 2009.

[14] J. Kisyński and D. Poole. Constraint processing in lifted probabilistic inference. In *UAI*, 2009.

[15] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, third edition, 1997.

[16] J. L. Lawall and O. Danvy. Continuation-based partial evaluation. In *Lisp and Functional Programming*, pages 227–238, 1994.

[17] B. Milch, L. S. Zettlemoyer, K. Kersting, M. Haimes, and L. P. Kaelbling. Lifted probabilistic inference with counting formulas. In *AAAI*, pages 1062–1068, 2008.

[18] T. Æ. Mogensen. Self-applicable online partial evaluation of the pure lambda calculus. In *PEPM*, pages 39–44, 1995.

[19] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

[20] D. Poole. First-order probabilistic inference. In *IJCAI*, pages 985–991, 2003.

[21] T. Rendel, K. Ostermann, and C. Hofer. Typed self-representation. In *PLDI*, 2009.

[22] P. Singla and P. Domingos. Lifted first-order belief propagation. In *AAAI*, pages 1094–1099, 2008.

[23] J. M. Siskind and B. A. Pearlmutter. Perturbation confusion and referential transparency: Correct functional implementation of forward-mode AD. In *Draft Proceedings of the 17th International Workshop on Implementation and Application of Functional Languages*, 2005.

[24] M. H. B. Sørensen, R. Glück, and N. D. Jones. Towards unifying deforestation, supercompilation, partial evaluation, and generalized partial computation. In *ESOP*, pages 485–500, 1994.