Inducing Probabilistic Programs by Bayesian Program Merging

Irvin Hwang
Stanford University

Andreas Stuhlmüller
MIT

Noah D. Goodman
Stanford University

Abstract

This report outlines an approach to learning generative models from data. We express models as probabilistic programs, which allows us to capture abstract patterns within the examples without sacrificing the ability to accurately represent given examples. By choosing our language for programs to be an extension of the algebraic data type of the examples, we can begin with a program that generates all and only the examples. We then introduce greater abstraction, and hence generalization, incrementally to the extent that it is explanatory. Motivated by previous approaches to model merging and program induction, we search for such explanatory abstractions using program transformations. We consider two types of transformation: *Abstraction* merges common subexpressions within a program into new functions (anti-unification). *Deargumentation* simplifies functions by reducing the number of arguments. We demonstrate that this approach finds key patterns in the domain of nested lists, including parametrized sub-functions and stochastic recursion.

Contents

1 Introduction

What patterns do you see when you look at figure 1? You might describe the image as a series of trees, where each tree has a large, blueish base and a number of green branches of variable length, with each branch ending in a flower that is either red or purple. Recognizing such patterns is an important aspect of intelligence. Understanding algorithms that can identify patterns contributes to our understanding of human and machine intelligence. One way to approach this pattern recognition problem is as the problem of learning generative models for observed examples: we wish to find a description of the process that gave rise to a set of examples, and we form this description in a rich language for generative processes—a probabilistic programming language. In this document, we build on the representation in [6] and explore a family of algorithms for learning probabilistic programs.

The main components of our approach are as follows: We represent data in terms of algebraic data types, we represent patterns in data as probabilistic programs, we guide search through program space using the Bayesian posterior probability, and we create search moves by detecting repeated computation. Our algorithm (1) turns data into a large program that generates all and only the examples, (2) identifies (approximately) repeated structure in the large program and transforming the program to make sharing explicit, and (3) continues to propose program transformations, accepting only those that increase the posterior probability of the program given the data. The probabilistic programs learned in this manner can be understood as generative models and reasoning about such models can be formulated in terms of probabilistic inference. We illustrate these ideas on list-structured data.

Generative models play a prominent role in modern machine learning and in understanding different classes of models, e.g., Hidden Markov models and probabilistic context-free grammars, and have led to a wide variety of applications. There is a trade-off between the variety of patterns a model class is able to capture and the feasibility of learning models in that class [4]. Much of machine learning has focused on studying classes of models with limited expressiveness in order to develop tractable algorithms for modeling large data sets. Our investigation takes a different approach and explores how learning might proceed in an expressive class of models with a focus on identifying abstract patterns from small amounts of data.

In particular, we represent generative models as programs in a probabilistic programming language. A probabilistic program represents a probability distribution, and each evaluation of the program results in a sample from the distribution. We implement these programs in a subset of the probabilistic programming language Church [2]. These programs can have parameterized functions and recursion, which allow for natural representation of "long-range" dependencies and recursive patterns. We will frame searching this space of models in terms of Bayesian model merging [5] and we demonstrate that this approach can find interesting patterns in the list data domain.

Before we proceed, a note about what this report is and is not: This report *is* a status update on our working system, containing detailed code and illustrative examples. It documents some progress we have made that we believe can be useful more generally. This report *is not* a completed academic work. In particular, it does little to situate our work within the context of previous work (some of which has directly inspired us), provides little high-level discussion, and aims for illustrative examples rather than

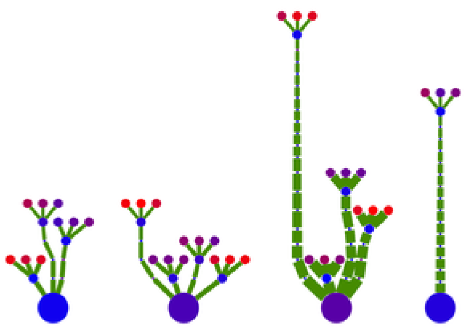


Figure 1: Tree-like objects.

compelling applications.

1.1 Bayesian model merging

Bayesian model merging is a framework for searching through a space of generative models in order to find a model that accurately generates the observed data. The main idea is to search model space through a series of merge transformations and to use the posterior $P(M|D)$ of model M given data D as a criterion for selecting transformations.

We create an initial model by building a program that has a uniform distribution over the training set (*data incorporation*). While this initial model has high likelihood $P(D|M)$, it never generates points outside the training set–it severely overfits the initial data. We generate alternative model hypotheses using program transformations that collapse model structure and that often result in better generalizations.

This technique has been successfully applied to learning artificial probabilistic grammars represented as hidden Markov models, n-grams, and probabilistic context-free grammars [5]. Our work differs in that we use the richer model class of probabilistic programs. This allows us to represent more complex patterns and to use more types of transformations, including transformations that correspond to lossy data compression.

1.2 Bayesian program merging

In order to use Bayesian program merging, we need to specify the following parts:

Data and language We represent data in terms of an algebraic data type, which gives us a way to form initial programs using the type constructors. Our language for programs consists of type constructors and additional operators.

Probabilistic programs We represent distributions over data as probabilistic programs. Program structure corresponds to regularities in the data.

Search Search is based on identifying programs with a high posterior probability and then applying transformations to these programs to continue exploration.

Program transformations Transformations collapse program structure, which can improve generalization and prior probability.

2 Data representation and program language

We assume that we can represent our data using an algebraic data type. This assumption gives us a starting point for program induction, since any data can be directly translated into a program which is a derivation (sequence of data constructor operations) of the data from the type specification.

2.1 Data and program representation in the list domain

We can model the trees shown in figure 1 in terms of nested lists (s-expressions). Each tree consists of nodes, where each node has a size and color along with a list of child nodes.

⟨*tree*⟩ ::= nil

⟨*tree*⟩ ::= (node ⟨*data*⟩ ⟨*tree*⟩ ⟨*tree*⟩ ...)

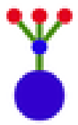
⟨*data*⟩ ::= (data ⟨*color*⟩ ⟨*size*⟩)

⟨*color*⟩ ::= (color [number])

⟨*size*⟩ ::= (size [number])

The following is an example of a tree expression along with a graphical representation.

```
(node (data (color 50) (size 1))
  (node (data (color 30) (size 0.3))
    (node (data (color 225) (size 0.3)))
    (node (data (color 225) (size 0.3)))
    (node (data (color 225) (size 0.3)))))
```



We now have a way of representing data as rudimentary programs. In order to capture interesting patterns, we need a more expressive language. We use the following subset of Church for Bayesian program merging in the tree domain:

⟨*expr*⟩ ::= (begin ⟨*expr*⟩ ⟨*expr*⟩ ...)
$\mid$ (lambda (⟨*var*⟩ ...) ⟨*expr*⟩)
$\mid$ (⟨*expr*⟩ ⟨*expr*⟩)
$\mid$ (define ⟨*var*⟩ ⟨*expr*⟩)
$\mid$ (if (flip [number]) ⟨*expr*⟩ ⟨*expr*⟩)
$\mid$ ⟨*var*⟩
$\mid$ ⟨*primitive*⟩

⟨*var*⟩ ::= V[number] $\mid$ F[number]

⟨*primitive*⟩ ::= uniform-choice $\mid$ list $\mid$ node $\mid$ data $\mid$ color $\mid$ size $\mid$ [number]

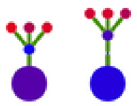
2.2 Data incorporation

The first step of Bayesian program merging is data incorporation. Data incorporation is the creation of an initial model by going through each example in the training set and creating an expression that evaluates to this example (in terms of the algebraic data type constructors). We combine these programs into a single expression that draws uniformly from this list.

```
(define (incorporate-data trees)
  `(lambda () (uniform-choice ,@(map tree->expression trees))))

(define (tree->expression tree)
  (if (null? tree)
      '()
      (pair 'node
            (pair (node-data->expression (first tree))
                  (map tree->expression (rest tree))))))

(define (node-data->expression lst)
  `(data (color (gaussian ,(first (second lst)) 25))
         (size ,(first (third lst)))))
```

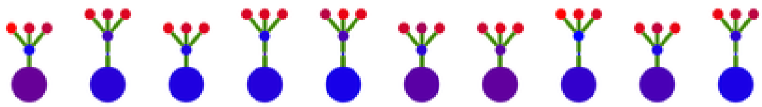
In our implementation, the data are unannotated s-expressions (e.g., `((1) (2))`), which can easily be converted into expressions in terms of the tree type constructors (the tree expression for the above example is `(node (data (color 1) (size 2)))`). In this report, we assume for readability that all data is in tree expression form. An important question we leave for future work is how to perform data incorporation when the data is less structured, e.g., feature vectors. Calling `incorporate-data` on

```
((node (data (color 70) (size 1))
       (node (data (color 37) (size 0.3))
             (node (data (color 213) (size 0.3)))
             (node (data (color 207) (size 0.3)))
             (node (data (color 211) (size 0.3)))))
 (node (data (color 43) (size 1))
       (node (data (color 47 25)) (size 0.1))
             (node (data (color 33) (size 0.3))
                   (node (data (color 220) (size 0.3)))
                   (node (data (color 224) (size 0.3)))
                   (node (data (color 207) (size 0.3)))))))
```



results in the program

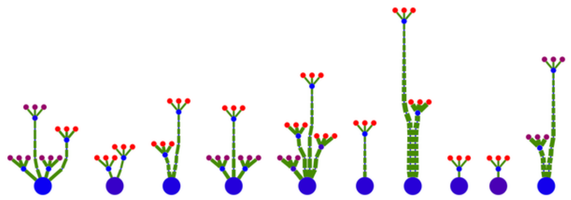
```
(lambda ()
  (uniform-choice
   (node (data (color (gaussian 70 25)) (size 1))
         (node (data (color (gaussian 37 25)) (size 0.3))
               (node (data (color (gaussian 213 25)) (size 0.3)))
               (node (data (color (gaussian 207 25)) (size 0.3)))
               (node (data (color (gaussian 211 25)) (size 0.3)))))
   (node (data (color (gaussian  43)) (size 1))
         (node (data (color (gaussian 47 25)) (size 0.1))
               (node (data (color (gaussian 33 25)) (size 0.3))
                     (node (data (color (gaussian 220 25)) (size 0.3)))
                     (node (data (color (gaussian 224 25)) (size 0.3)))
                     (node (data (color (gaussian 207 25)) (size 0.3))))))))
```



3 Objective: Posterior probability of probabilistic programs

A generative model is a joint probability distribution over latent states and observed data. We can represent such a joint distribution as a program in a probabilistic programming language like Church [2]. The structure of the program, i.e., the decomposition into functions and the flow of control, can capture regularities in the data. We illustrate this idea using a probabilistic program that generates the images in figure 1. Different parts of the program correspond to the different patterns we described earlier (to improve readability, we use semantically meaningful function and variable names instead of `F[number]` and `V[number]` as specified in our grammar).

```
(define (tree)
  (uniform-choice
   (node (body) (branch))
   (node (body) (branch) (branch))
   (node (body) (branch) (branch) (branch))
   (node (body) (branch) (branch) (branch) (branch))))

(define (body)
  (data (color (gaussian 50 25)) (size 1)))

(define (branch)
  (if (flip .2)
      (flower (if (flip .5) 150 255))
      (node (branch-info) (branch))))

(define (branch-info)
  (data (color (gaussian 0 25)) (size .1)))

(define (flower shade)
  (node (data (color (gaussian 0 25)) (size .3))
        (petal shade)
        (petal shade)
        (petal shade)))

(define (petal shade)
  (node (data (color shade) (size .3))))
```



Our programs are a combination of data constructors, control flow operations, and lambda abstractions. When we call `(tree)`, the program shown above first determines the number of branches the tree will have and then creates a large body node that connects these branches. Each branch function recursively connects a series of small nodes together and ends in a call to the flower function, passing it one of two colors. The flower function creates a node with three identically colored "petal" nodes. The program structure reflects patterns such as a flower having three petals and the body having a large base. The compositionality of the language captures relational patterns such as the fact that branches end in flowers. Given that programs can represent such structures, our goal is to transform the initial program generated by data incorporation into such a more structured form.

3.1 Abstract syntax utilities

Some program transformations generate new functions. We call such functions *abstractions*. We represent them using a name, argument variables, and a pattern, i.e., the s-expression that makes up the body of the function. In the following code snippet, we use the `sym` function that creates readable unique symbols for function and variable names.

```
(define (make-abstraction body variables)
  (make-named-abstraction (sym FUNC-SYMBOL) body variables))
(define (make-named-abstraction name body variables)
  (list 'abstraction name variables body))

(define abstraction->name second)
(define abstraction->vars third)
(define abstraction->body fourth)
```

Programs represent the generative models we search over. They consist of a list of abstractions and a body.

```
(define (make-program abstractions body)
  (list 'program abstractions body))
(define program->abstractions second)
(define program->body third)
```

We wrap programs into another data type to keep track of additional information during search. We motivate this in the section on search in more detail. The basic idea is to avoid recomputation of a program's likelihood (an expensive computation) when transformations do not affect a program's semantics.

```
(define (make-program+ program posterior log-likelihood log-prior
    semantics-preserved)
  (list 'program+ program posterior log-likelihood log-prior
        semantics-preserved))
(define program+->program second)
(define program+->posterior third)
(define program+->log-likelihood fourth)
(define program+->log-prior fifth)
(define program+->semantics-preserved sixth)
(define (program+->program-transform semantics-preserved program+ new-program)
  (make-program+ new-program
                 (program+->posterior program+)
                 (program+->log-likelihood program+)
                 (program+->log-prior program+)
                 semantics-preserved))
```

3.2 Posterior probability

Probabilistic programs correspond to probability distributions on observed data, i.e., we can compute the likelihood of a given observation under a program. Given a process for generating programs (a prior), we use Bayes theorem to compute the posterior probability of a program:

$$P(M|D) \propto P(D|M)P(M) \tag{1}$$

Here, $P(D|M)$ is the probability that program M generates data D, the likelihood, and $P(M)$ is the prior probability of program M. We use a prior based on program length:

$$P(M) \propto e^{-\alpha size(M)} \tag{2}$$

```
(define (log-prior program)
  (- (* alpha (program-size program))))
```

This prior biases the search towards smaller programs. Increasing the constant α gives the prior more weight when calculating the posterior, which means that minimizing program size is a more important criterion. A program's size is the number of symbols in the function bodies as well as in the main body.

```
(define (sexpr-size sexpr)
  (if (list? sexpr)
      (apply + (map sexpr-size sexpr))
      1))

(define (abstraction-size abstraction)
  (sexpr-size (abstraction->body abstraction)))

(define (program-size program)
  (let* ([abstraction-sizes (apply + (map abstraction-size
      (program->abstractions program)))]
         [body-size (sexpr-size (program->body program))])
    (+ abstraction-sizes body-size)))
```

Computing the likelihood is the difficult part of the posterior probability computation. Intuitively, we can think of the likelihood as tracking how good a particular program is at producing a set of target data. This is important in search, since it gives precise, quantitative information on whether and how to adjust our hypothesis program. However, since there may be a large number of possible settings for the random choices of a program, this can make determining which choices lead to the observed data difficult. Furthermore, for any given data point, there could be multiple settings that generate this data point; to correctly compute the likelihood, we need to take into account all of them. Often, we cannot compute this quantity exactly. In the following, we describee a way to approximate this computation for list data.

3.3 Likelihood estimation in the list domain

In the case of programs that generate list-structured data, we can estimate the likelihood using a sequential Monte Carlo method (`smc`) that splits the data generation process into generation of the discrete list topology and the continuous value topology. We begin by factoring the problem of generating the data set into the problem of generating each of the data points:

$$P(T|M) \quad = \quad \prod_{t \in M} P(t|M) \tag{3}$$

Here, T is the observed set of trees, t a single tree, and M the generative model (program).

```
(define (log-likelihood trees prog sample-size)
  (apply + (map (lambda (tree) (single-log-likelihood prog sample-size tree))
      trees)))
```

We will estimate the likelihood of a single tree, $P(t|M)$, by evaluating the program many times, restricting the computation to result in the target tree each time. The product of the probabilities of all random choices made during a single evaluation corresponds to the probability of a single possible way of generating the tree. Since there may be multiple ways for a program to generate a given tree, we average these probabilities.

```
(define (single-log-likelihood program popsize tree)
  (let* ([new-program (replace-gaussian (desugar program))]
         [model (eval (program->sexpr new-program))]
         [topology-scores+tree-parameters (compute-topology-scores+evaluate
            model tree popsize)]
         [topology-scores (first topology-scores+tree-parameters)]
         [trees-with-parameters (second topology-scores+tree-parameters)]
         [data-scores (map (lambda (tree-with-parameters) (compute-data-score
            tree tree-with-parameters)) trees-with-parameters)] ;;remove -inf?
         [scores (delete -inf.0 (map + topology-scores data-scores))]
         [score (if (null? scores)
                    -inf.0
                    (apply log-sum-exp scores))])
```

```
    score))
```

We take advantage of the fact we can directly compute the probability of a sample from a Gaussian given the parameters. We therefore modify the Gaussian functions in the program to output the mean and variance for a particular node instead of sampling a value from the distribution. The code below also changes the uniform-choice syntactic construct into a uniform-draw, which current Church implementations provide.

```
(define (replace-gaussian program)
  (define (gaussian? sexpr)
    (tagged-list? sexpr 'gaussian))
  (define (return-parameters sexpr)
    `(list 'gaussian-parameters ,(second sexpr) ,(third sexpr)))
  (define (replace-in-abstraction abstraction)
    (make-named-abstraction (abstraction->name abstraction) (sexp-search
        gaussian? return-parameters (abstraction->pattern abstraction))
        (abstraction->vars abstraction)))
  (let* ([converted-abstractions (map replace-in-abstraction
      (program->abstractions program))]
          [converted-body (sexp-search gaussian? return-parameters
              (program->body program))])
    (make-program converted-abstractions converted-body)))

(define (desugar program)
  (define (uniform-choice? sexpr)
    (tagged-list? sexpr 'uniform-choice))
  (define (uniform-draw-conversion sexpr)
    `((uniform-draw (list ,@(map thunkify (rest sexpr))))))
  (define tests+replacements (zip (list uniform-choice?) (list
      uniform-draw-conversion)))
  (define (apply-transforms sexpr)
    (fold (lambda (test+replacement expr)
            (sexp-search (first test+replacement) (second test+replacement)
                expr))
          sexpr
          tests+replacements))
  (define (desugar-abstraction abstraction)
    (make-named-abstraction (abstraction->name abstraction) (apply-transforms
        (abstraction->pattern abstraction)) (abstraction->vars abstraction)))
  (let* ([converted-abstractions (map  desugar-abstraction
      (program->abstractions program))]
          [converted-body (apply-transforms (program->body program))])
    (make-program converted-abstractions converted-body)))

(define (thunkify sexpr) `(lambda () ,sexpr))
```

We evaluate the modified program multiple times to generate trees whose topology (node structure) matches the observed tree and which provide mean and variance of the Gaussians used to generate colors. During the evaluation of the modified programs, we capture the probability for the choices made in generating the topology of the tree.

In the code below, `smc-core` forces the program to generate the desired data. A detailed description of `smc-core` is beyond the scope of this report. In short, the method is an incremental forward sampler; for this reason, we separate out the continuous choices, since forward sampling has probability 0 of generating the observed values.

```
(define (compute-topology-scores+evaluate model tree popsize)
  (let* ([smc-core-arguments (create-smc-core-args model tree popsize)]
          [samples (apply smc-core smc-core-arguments)]
          [repeat-symbol (find-repeat-symbol samples)]
          [samples
```

```
        (fold (lambda (s a)
                (if (member (mcmc-state->addrval s repeat-symbol)
                            (map (lambda (x) (mcmc-state->addrval x
                                repeat-symbol)) a))
                    a (pair s a))) '() samples)]
       [topology-scores (map mcmc-state->score samples)]
       [generated-trees (map mcmc-state->query-value samples)])
  (list topology-scores generated-trees)))
```

We use the Gaussian parameters for each node's color to determine the probability of the observed color values of a tree:

```
(define (compute-data-score tree tree-with-parameters)
    (if (null? tree)
        0
        (+ (single-data-score (node->data tree) (node->data
            tree-with-parameters)) (apply + (map compute-data-score
            (node->children tree) (node->children tree-with-parameters))))))

(define (single-data-score original-data parameterized-data)
  (let* ([color-score (score-attribute (data->color original-data)
      (data->color parameterized-data))]
         [size-score (score-attribute (data->size original-data) (data->size
             parameterized-data))])
    (+ color-score size-score)))

(define (score-attribute original-attribute parameterized-attribute)
  (if (tagged-list? parameterized-attribute 'gaussian-parameters)
      (log (normal-pdf (first original-attribute) (gaussian->mean
          parameterized-attribute) (gaussian->variance
          parameterized-attribute)))
      (if (= (first original-attribute) (first parameterized-attribute)) 0
          -inf.0)))

(define gaussian->mean second)
(define gaussian->variance third)
```

4 Search over program space

We are interested in programs with high posterior probability. Many different search strategies are possible, but in this report, we limit ourselves to beam search using the posterior probability as a search heuristic.

```
(define (beam-search data init-program beam-size depth)
  (let* ([top-transformations
          (sort-by-posterior
           data
           (beam-learn-search-transformations data init-program beam-size
              depth))])
    (if (null? top-transformations)
        init-program
        (program+->program (first top-transformations)))))

(define (beam-search-transformations data program beam-size depth)
  (let ([init-program+ (make-program+ program 0 (log-likelihood data program
     10) (log-prior program) #f)])
    (depth-iterated-transformations (lambda (programs+) (best-n data programs+
       beam-size)) init-program+ depth)))

(define (best-n data programs+ n)
  (max-take (sort-by-posterior data programs+) n))
```

The main part of the search is performed by `depth-iterated-transformations`, which recursively applies program transformations to the best programs at a given search depth and then filters the results to get the best programs for the next depth.

```
(define (depth-iterated-transformations cfilter program+ depth)
  (let* ([transformed-programs+ (apply-and-filter-transformations depth
     cfilter program+)])
    (delete '() (append transformed-programs+
                        (apply append (map (lambda (prog)
                                             (depth-iterated-transformations
                                                cfilter prog (- depth 1)))
                                           transformed-programs+))))))
```

We reduce the amount of computation required when choosing the best programs at each level of search by separating program transformations that preserve semantics from those that do not. Marking programs based on their transformation type allows us to reuse likelihood for a program that was created by a semantics preserving transformation.

```
(define (apply-and-filter-transformations depth cfilter program+)
  (if (= depth 0)
      '()
      (let* ([semantics-preserved-programs+ (apply-transformations program+
          semantic-preserving-transformations #t)]
             [semantics-changed-programs+ (apply-transformations program+
                 semantic-changing-transformations #f)])
        (cfilter (append semantics-preserved-programs+
            semantics-changed-programs+)))))

(define (apply-transformations program+ transformations semantics-preserving)
  (let* ([program (program+->program program+)]
         [transformed-programs (delete '() (concatenate (map (lambda
             (transform) (transform program #t)) transformations)))]
         [transformed-programs+ (map (lambda (program)
             (program+->program-transform semantics-preserving program+
             program)) transformed-programs)])
    transformed-programs+))
```

The posterior distribution is estimated by combining an estimate of the likelihood with the computation of the prior.

```
(define (sort-by-posterior data programs+)
  (let* ([programs (map program+->program programs+)]
         [semantics-flags (map program+->semantics-preserved programs+)]
         [log-priors (map log-prior programs)]
         [log-likelihoods (map (lambda (prog+ semantics-flag)
                                 (if semantics-flag
                                     (program+->log-likelihood prog+)
                                     (log-likelihood data (program+->program
                                        prog+) 10))) programs+
                                        semantics-flags)]
         [posteriors (map + log-priors log-likelihoods)]
         [new-programs+ (map make-program+ programs posteriors log-likelihoods
             log-priors semantics-flags)]
         [posteriors> (lambda (a b) (> (program+->posterior a)
             (program+->posterior b)))])
    (my-list-sort posteriors> new-programs+)))
```

5 Program transformations

We described earlier how patterns in the program representation of data can be viewed as repeated computation. Here we describe how this idea manifests itself as concrete program transformations used in exploring the search space.

5.1 Abstraction

Abstraction is the process of replacing function calls in a program with the body of the function called. Abstraction creates new functions based on syntactic patterns in a program and replaces these patterns with calls to the newly created functions. We can think of this process as creating lambda abstractions that potentially compress the program by removing duplication in the code and acts as a proxy for recognizing repeated computation. The following code fragment outlines this procedure where `compressions` finds the different (lambda) abstractions that can be formed by anti-unifying (partially matching) pairs of subexpressions in a condensed form of the program (only the bodies of the functions and the body of the program). Duplicate abstractions are filtered out, then function calls for each of the abstractions are created.

```
(define (compressions program . nofilter)
  (let* ([condensed-program (condense-program program)]
         [abstractions (possible-abstractions condensed-program)]
         [compressed-programs (map (curry compress-program program)
             abstractions)]
         [prog-size (program-size program)]
         [valid-compressed-programs
          (if (not (null? nofilter))
              compressed-programs
              (filter (lambda (cp) (<= (program-size cp)
                                      (+ prog-size 1)))
                      compressed-programs))])
    valid-compressed-programs))

(define (condense-program program)
  `(,@(map abstraction->body (program->abstractions program))
    ,(program->body program)))

(define (possible-abstractions expr)
  (let* ([subexpr-pairs (list-unique-commutative-pairs (all-subexprs expr))]
         [abstractions (map-apply (curry anti-unify-abstraction expr)
             subexpr-pairs)])
    (filter-abstractions  abstractions)))
```

Aside from the obvious correspondence between repetition in program syntax and repetition in computation, there are two other reasons to believe that this is a useful transformation. In terms of Bayesian model merging we are merging the structure of the model, which potentially leads to models with better generalization properties [5]. The abstraction transformation can also also be interpreted as finding partial symmetries like in the work of Bokeloh et al. [1], which was successful in the domain of inverse-procedural modeling.

The following example illustrates the abstraction transformation on a language slightly different from the tree example.

```
(uniform-choice
 (node
  (node a (node a (node b) (node b)))
  (node a (node a (node c) (node c)))))
```

a possible result of this transformation would be

```
(begin
  (define (F1 V1 V2)
    (node a (node a (node V1) (node V2))))
  (uniform-choice (node (F1 b b) (F1 c c))))
```

The first thing to note is both programs have the same behavior, i.e., this transformation is semantics preserving. Both programs return either `(a (a (b) (b)))` or `(a (a (c) (c)))` with equal probability.

The transformation can be described as refactoring subexpressions that partially match in a program with a function whose body is the common parts of the matching subexpressions. In the above example the subexpressions that partially match are `(node a (node a (node b) (node b)))` and `(node a (node a (node c) (node c)))`. The common subexpression is `(node a (node a (node x) (node y)))`, the function created using this common subexpression is `F1` and the original subexpressions are refactored as `(F1 b b)` and `(F1 c c)`.

An abstraction transformation can be created for each pair of subexpressions that have a partial match. In the case of `(+ (+ 2 2) (- 2 5))` the following pairs of subexpressions have a partial match: `[2,2]`, `[(+ 2 2), (- 2 5)]`, `[(+ 2 2), (+ (+ 2 2) (- 2 5))]`.

5.1.1 Anti-unification

The process of finding a partial match between two expressions is called anti-unification. One way to understand the process is in terms of the syntax trees for the expressions. Every s-expression can be thought of as a tree where the lists and sublists of the s-expression make up the interior nodes and the primitive elements of the lists (e.g., symbols, numbers) are the leaves. The tree in figure 2 corresponds to the expression `(+ (+ 2 2) (- 2 5))`. Finding a partial match between two expressions can be thought of as finding a common subtree between their tree representations.

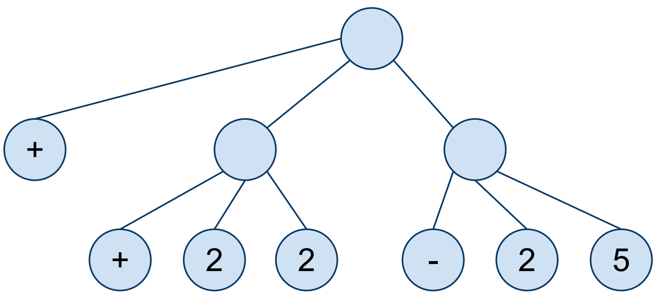


Figure 2: An expression represented as a tree.

Anti-unification proceeds by recursively comparing two expressions, A and B (performed by `build-pattern`). If A and B are the same primitive that primitive is returned. If A and B are lists $(a_1, \ldots, a_n)$ and $(b_1, \ldots, b_n)$ of the same length then a list C $(c_1, \ldots, c_n)$ where each element c_i is the anti-unification a_i and b_i is returned. For all other cases when A and B do not match a variable is returned.

```
(define anti-unify
  (lambda (expr1 expr2)
    (begin
      (define variables '())

      (define (add-variable!)
        (set! variables (pair (sym (var-symbol)) variables))
        (first variables))
```

```
    (define (build-pattern expr1 expr2)
      (cond [(and (primitive? expr1) (primitive? expr2)) (if (equal? expr1
          expr2) expr1 (add-variable!))]
            [(or (primitive? expr1) (primitive? expr2)) (add-variable!)]
            [(not (eqv? (length expr1) (length expr2))) (add-variable!)]
            [else
             (let ([unified-expr (map (lambda (subexpr1 subexpr2)
                 (build-pattern subexpr1 subexpr2))
                                      expr1 expr2)])
               unified-expr)]))
    (let ([pattern (build-pattern expr1 expr2)])
      (list pattern (reverse variables))))))
```

Example We illustrate the process of anti-unification on the expressions `(+ (+ 2 2) (- 2 5))` and `(+ (- 2 3) 4)` The first step is to compare the root of the trees and make sure we have lists of the same size, in this case they are both of size three so we have a matching roots and a partial match of `(* * *)` where the *'s are yet to be determined. Now we recursively attempt to match the three subexpressions + with +, `(+ 2 2)` with `(- 2 3)`, and `(- 2 5)` with 4. Since + and + are both primitives that are the same they match, and our partial match is now `(+ * *)`. Comparing `(+ 2 2)` and `(- 2 3)` we see that they are both lists of size 3 and so they match giving us a total partial match of `(+ (* * *) *)`. Again we recursively match subexpressions of `(+ 2 2)` and `(- 2 3)`, i.e., + to -, 2 to 2, and 2 to 3. Since + and - are primitives that do not match we replace them with a variable to get a total partial match of `(+ (V1 * *) *)` Likewise after comparing 2 to 2 and 2 to 3 we get `(+ (V1 2 V2) *)` as our partial match. In the final comparison of `(- 2 5)` and 4 we check can see there is no match because one `(- 2 5)` is a list and 4 is a primitive so the final result of our anti-unification is `(+ (V1 2 V2) V3)`.

5.1.2 Refactoring programs

Now that we have created lambda abstractions from the partial matches between two subexpressions of some program, P, we can use these abstractions to refactor P, possibly compressing it. The idea will be to take the pattern of the abstraction (the body of the lambda abstraction) and replace any subexpressions of P that fit the pattern. A subexpression that fits the pattern is replaced by an application of the lambda abstraction. We first do this replacement for each of the functions in P and then the body, resulting in a program P'. After all the replacements are made, the lambda abstraction is inserted into the set of functions for P' and this is our refactored program.

```
(define (compress-program program abstraction)
  (let* ([compressed-abstractions (map (curry compress-abstraction
      abstraction) (program->abstractions program))]
         [compressed-body (replace-matches (program->body program)
             abstraction)])
    (make-program (pair abstraction compressed-abstractions)
                  compressed-body)))

(define (compress-abstraction compressor compressee)
  (make-named-abstraction (abstraction->name compressee)
                          (replace-matches (abstraction->body compressee)
                              compressor)
                          (abstraction->vars compressee)))
```

Finding and replacing pattern matches of an abstraction F in an expression E proceeds recursively by taking the body of F and trying to match it to the expression on E. A match between the body of the abstraction and the expression is determined by unification. If a match exists, then the application of F is returned where the arguments to F are refactored (if F has arguments). If the match does not hold then E is returned with each of its subexpressions refactored. If E is a primitive and not a match it is returned.

```
(define (replace-matches s abstraction)
  (let ([unified-vars (unify s
```

```
                              (abstraction->body abstraction)
                              (abstraction->vars abstraction))])
    (if (false? unified-vars)
        (if (list? s)
            (map (lambda (si) (replace-matches si abstraction)) s)
            s)
        (pair (abstraction->name abstraction)
              (map (lambda (var) (replace-matches (rest (assq var
                  unified-vars)) abstraction))
                (abstraction->vars abstraction))))))
```

Example In the case of `(+ (+ 2 2) (- 2 5))` the partial match resulting from anti-unification between the subexpressions `[(+ (+ 2 2) (- 2 5)), (+ 2 2)]` is `(+ V1 V2)`. We refactor the original expression `(+ (+ 2 2) (- 2 5))` in terms of `(+ V1 V2)` by creating a function `(define (F1 V1 V2) (+ V1 V2))` and applying it in the expression. For example one application could be `(F1 (+ 2 2) (- 2 5))` and another could be `(+ (F1 2 2) (- 2 5))`. In general, we apply the function everywhere possible and get a refactored program such as:

```
(+ (+ 2 2) (- 2 5))
=>
(begin
  (define (F1 V1 V2) (+ V1 V2))
  (F1 (F1 2 2) (- 2 5)))
```

The input to the refactoring procedure is a function, F, created from anti-unification and an expression, E, which will be refactored in terms of F. In the example above F was `(define (F1 V1 V2) (+ V1 V2))`, E would be `(+ (+ 2 2) (- 2 5))`, and the result of refactoring was

```
(begin
  (define (F1 V1 V2) (+ V1 V2))
  (F1 (F1 2 2) (- 2 5)))
```

In the example `(+ (+ 2 2) (- 2 5))` matches `(+ V1 V2)` so an application of F1 to refactored arguments `(+ 2 2)` and `(- 2 5)` is returned resulting in `(F1 (F1 2 2) (- 2 5))`[1].

5.1.3 Unification

Determining whether there is a match between a function body and an expression is known as unification [3]. Earlier we described anti-unification which can be thought of as a process to create a pattern from two expressions, unification can be viewed as the opposite process of seeing if and how a given pattern fits onto an expression. The return value of unification is a list of assignments for the variables of the function that would make the function the same as the expression.

The input to unification is a function F, and an expression E. Unification occurs recursively by checking whether the body of F and E are lists of the same size. If they are the same size then unification returns the unification of each of the subexpressions. If they are not the same size or only one of them is a list unification returns false. In the case where both expressions being unified are primitives true is returned if they are equal and false otherwise. In the case where the function expression of the unification is a variable an assignment is returned, i.e., the variable along with the other expression passed to unification. At the end a check is made to see if any unifications have returned false, in which case unification of F and E returns false. There is also a check that any variable that is repeated in F has the same value assigned to it for each place it appears. If this is not the case unification returns false. If unification is not false then an assignment for each unique variable of F is returned.

```
(define unify
  (lambda (s sv vars)
    (begin
```

[1] Observe the function `F1` is basically the addition function between two numbers and so refactoring the program in terms of this function does compress the original expression at all. An example where refactoring can compress an expression would be if the abstraction between `[(+ (+ 2 2) (- 2 5)), (+ 2 2)]` had been `(define (F1 V1) (+ V1 V1))`, then the refactored expression would be `(+ (F1 2) (- 2 5))`, which is one character smaller than `(+ (+ 2 2) (- 2 5))`.

```
(define (variable? obj)
  (member obj vars))

;;deals with a variable that occurs multiple times in sv
(define (check/remove-repeated unified-vars)
  (let* ([repeated-vars (filter more-than-one (map (curry all-assoc
      unified-vars) (map first unified-vars)))])
    (if (and (all (map all-equal? repeated-vars)) (not (any false?
        unified-vars)))
        (delete-duplicates unified-vars)
        #f)))

(cond [(variable? sv) (if (eq? s 'lambda) #f (list (pair sv s)))]
      [(and (primitive? s) (primitive? sv)) (if (eqv? s sv) '() #f)]
      [(or (primitive? s) (primitive? sv)) #f]
      [(not (eqv? (length s) (length sv))) #f]
      [else
       (let ([assignments (map (lambda (si sj) (unify si sj vars)) s
           sv)])
         (if (any false? assignments)
             #f
             (check/remove-repeated (apply append assignments))))])))))
```

Example The example listed earlier where the function F is `(define (F1 V1 V2) (+ V1 V2))` and the s-expression being refactored is `(+ (+ 2 2) (- 2 5))` is used to illustrate unification. Since `(+ (+ 2 2) (- 2 5))` and `(+ V1 V2)` are the same length unification is applied to the subexpression pairs `[+,+]`, `[(+ 2 2), V1]`, `[(- 2 5), V2]`.

Unification between + and + returns nothing since neither is a variable and they match. Unification between `(+ 2 2)` and V1 returns the assignment of `(+ 2 2)` to V1 and likewise for `(- 2 5)` and V2. So the function F1 matches the s-expression `(+ (+ 2 2) (- 2 5))` with variable assignments `V1:=(+ 2 2)` and `V2:=(- 2 5)`.

If the expression being unified with F1 had been `(- (+ 2 2) (- 2 5))` then unification between the outer - of the expression and the + of F1 would have returned false and the unification would have failed.

5.1.4 Summary

Abstraction is a program transformation that identifies repeated computation in a program by finding patterns in its syntax. These patterns in syntax correspond to patterns in the data. We can interpret this in terms of repeated computation by recognizing that repeated syntax corresponds to repeated sequences of operations. This might seem like a limited notion of pattern, but it is worth contemplating the role of lambda abstraction in the lambda calculus and the expressiveness of this language despite (or perhaps because of) its simplicity.

The process of abstraction follows two main steps (1) create abstractions from common subexpressions in a program via anti-unification (2) compress the program with the abstractions by replacing instances of the abstraction with a function application via unification. Going back to our tree example we illustrate the process on the program in the data incorporation section. Anti-unification found seventeen possible matches or abstractions. The abstractions that produce the smallest compressed program and the fifth smallest are

```
(abstraction F1 (V1 V2)
             (data (color (gaussian V1 25)) (size V2)))

(abstraction F1 (V1 V2 V3 V4)
             (node (data (color (gaussian V1 25)) (size 0.3))
                   (node (data (color (gaussian V2 25)) (size 0.3)))
                   (node (data (color (gaussian V3 25)) (size 0.3)))
                   (node (data (color (gaussian V4 25)) (size 0.3)))))
```

The following are the programs compressed using these abstractions the first being of size 55 and the second being 66.

```
(program
 ((abstraction F1 (V1 V2)
               (data (color (gaussian V1 25)) (size V2))))
 (uniform-choice
  (node (F1 70 1)
        (node (F1 37 0.3) (node (F1 213 0.3))
              (node (F1 207 0.3)) (node (F1 211 0.3))))
 (node (F1 43 1)
       (node (F1 47 0.1)
             (node (F1 33 0.3) (node (F1 220 0.3))
                   (node (F1 224 0.3)) (node (F1 207 0.3))))))

(program
 ((abstraction F1 (V1 V2 V3 V4)
               (node (data (color (gaussian V1 25)) (size 0.3))
                     (node (data (color (gaussian V2 25)) (size 0.3)))
                     (node (data (color (gaussian V3 25)) (size 0.3)))
                     (node (data (color (gaussian V4 25)) (size 0.3))))))
 (uniform-choice
  (node (data (color (gaussian 70 25)) (size 1))
        (F1 37 213 207 211)))
 (node (data (color (gaussian 43 25)) (size 1))
       (node
        (data (color (gaussian 47 25)) (size 0.1))
        (F1 33 220 224 207))))
```

The last abstraction captures the flower pattern we mentioned earlier. Intuitively, it seems like we can capture even more structure and replace the variables for the petal colors with a fixed value since they are all similar. Instead of explaining the data as being drawn from several Gaussians with slightly different means, we would like to say the data comes from only one Gaussian with a single mean. We address this issue in the next section with another type of program transformation.

5.2 Deargumentation

Deargumentation is a program transformation that takes a function in a program and removes one of its arguments. The variable that is removed is redefined inside the function in terms of all the values it has been assigned (i.e., the values passed to the function for that variable during an application). We can create different program transformations by allowing multiple schemes for redefining a variable, via `replacement-function`, as a function of its instantiations.

```
(define (deargument replacement-function program abstraction variable)
  (let* ([new-abstraction (remove-abstraction-variable replacement-function
      program abstraction variable)])
    (if (null? new-abstraction)
        '()
        (let* ([program-with-new-abstraction (program->replace-abstraction
            program new-abstraction)]
               [new-program (remove-application-argument
                   program-with-new-abstraction abstraction variable)])
          new-program))))
```

The abstraction whose variable is being removed keeps the same body, but the variable removed is now assigned a value within the body instead of having its value passed in as an argument.

```
(define (remove-abstraction-variable replacement-function program abstraction
    variable)
  (let* ([variable-instances (find-variable-instances program abstraction
      variable)]
         [variable-definition (replacement-function program abstraction
             variable variable-instances)])
    (if (equal? variable-definition NO-REPLACEMENT)
```

```
        '()
        (let* ([new-body `((lambda (,variable) ,(abstraction->body
            abstraction)) ,variable-definition)]
                [new-variables (delete variable (abstraction->vars
                    abstraction))])
          (make-named-abstraction (abstraction->name abstraction) new-body
              new-variables)))))

(define (program->abstraction-applications program target-abstraction)
  (define (target-abstraction-application? sexpr)
    (if (non-empty-list? sexpr)
        (if (equal? (first sexpr) (abstraction->name target-abstraction))
            #t
            #f)
        #f))
  (let* ([abstraction-bodies (map abstraction->body (program->abstractions
      program))]
         [possible-locations (pair (program->body program)
             abstraction-bodies)])
    (deep-find-all target-abstraction-application? possible-locations)))

(define (deep-find-all pred? sexp)
  (filter pred? (all-subexprs sexp)))

(define (all-subexprs t)
  (let loop ([t (list t)])
    (cond [(null? t) '()]
          [(primitive? (first t)) (loop (rest t))]
          [else (pair (first t) (loop (append (first t) (rest t))))])))

(define (find-variable-instances program abstraction variable)
  (let* ([abstraction-applications (program->abstraction-applications program
      abstraction)]
         [variable-position (abstraction->variable-position abstraction
             variable)]
         [variable-instances (map (curry ith-argument variable-position)
             abstraction-applications)])
    variable-instances))

(define (ith-argument i function-application)
  (list-ref function-application (+ i 1)))
```

After the abstraction has been adjusted, any place it was used (i.e., the function was applied) needs to be changed so that no values are passed in to the variable that was removed.

```
(define (remove-application-argument program abstraction variable)
  (define (abstraction-application? sexpr)
    (if (non-empty-list? sexpr)
        (equal? (first sexpr) (abstraction->name abstraction))
        #f))
  (define (change-application variable-position application)
    (define (change-recursive-arguments argument) ;;in case one of the
        arguments is an application of the abstraction currently being
        deargumented
      (if (abstraction-application? argument)
          (change-application variable-position argument)
          argument))
    (let* ([ith-removed (remove-ith-argument variable-position application)])
      (map change-recursive-arguments ith-removed)))
  (let* ([variable-position (abstraction->variable-position abstraction
      variable)]
```

```
        [program-sexpr (program->sexpr program)]
        [changed-sexpr (sexp-search abstraction-application? (curry
           change-application variable-position) program-sexpr)]
        [new-program (sexpr->program changed-sexpr)])
    new-program))

(define (remove-ith-argument i function-application)
  (append (take function-application (+ i 1)) (drop function-application (+ i
     2))))

(define (sexp-search pred? func sexp)
  (if (pred? sexp)
      (func sexp)
      (if (list? sexp)
          (map (curry sexp-search pred? func) sexp)
          sexp)))
```

In the following, we demonstrate that this transformation is useful for generalizing a program to have recursive structure as well as dealing with continuous data types that may be distorted with noise, for example the color data of the tree examples.

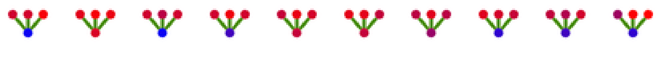
5.2.1 Compactly representing noisy data

One issue with abstraction is that anything other than perfect equality results in the creation of a variable when matching two expressions. So if we have an expression such as `(+ 2 2)` and `(+ 2 2.01)` and we know there is some noise in the system then we may want to treat these expressions as if they were the same, i.e., inverse-inline them to `(define (F1) (+ 2 2))` or `(define (F1) (+ 2 2.01))` rather than `(define (F1 x) (+ 2 x))`. We can do this using a Deargumentation transformation along with a `replacment-function` called `noisy-number-replacment` that replaces the variable with the mean of its instances.

```
(define (noisy-number-replacement program abstraction variable
    variable-instances)
  (if (all (map number? variable-instances))
      (my-mean variable-instances)
      NO-REPLACEMENT))
```

Going back to the tree example, suppose we have the program following program.

```
(begin
  (define flower
    (lambda (V1 V2 V3 V4)
      (node (data (color (gaussian V1 25)) (size 0.3))
            (node (data (color (gaussian V2 25)) (size 0.3)))
            (node (data (color (gaussian V3 25)) (size 0.3)))
            (node (data (color (gaussian V4 25)) (size 0.3))))))
  (uniform-choice
   (flower 200 213 207 211))
  (flower 33 220 224 207))
```



Applying the Deargumentation transform to the `flower` function and variabe `V2` would first result in identifying instances of `V2`, which are 213 and 220. The abstraction, `flower`, is now changed using `remove-abstraction-variable` to get:

```
(define flower
  (lambda (V1 V3 V4)
    ((lambda (V2)
       (node (data (color (gaussian V1 25)) (size 0.3))
```

```
             (node (data (color (gaussian V2 25)) (size 0.3)))
             (node (data (color (gaussian V3 25)) (size 0.3)))
             (node (data (color (gaussian V4 25)) (size 0.3)))))
     216.5)))
```

The whole program is now adjusted to incorporate the new version of `flower` using `remove-application-argument`. This creates a potentially simpler model (if there were more applications of flower with a likelihood that is similar to the original, which can be observed in the data generated by the transformed model).

```
(begin
  (define flower
    (lambda (V1 V3 V4)
      ((lambda (V2)
         (node (data (color (gaussian V1 25)) (size 0.3))
               (node (data (color (gaussian V2 25)) (size 0.3)))
               (node (data (color (gaussian V3 25)) (size 0.3)))
               (node (data (color (gaussian V4 25)) (size 0.3)))))
       216.5)))
  (uniform-choice
   (flower 200 207 211))
  (flower 33 224 207))
```

Now if we had applied the Deargumentation transform to the `flower` function to variabe `V1` whose variable instances are not similar we would get the following abstraction. Which creates a program that generated data unlike the original program.

```
(define flower
  (lambda (V2 V3 V4)
    ((lambda (V1)
       (node (data (color (gaussian V1 25)) (size 0.3))
             (node (data (color (gaussian V2 25)) (size 0.3)))
             (node (data (color (gaussian V3 25)) (size 0.3)))
             (node (data (color (gaussian V4 25)) (size 0.3)))))
     116.5)))

(begin
  (define flower
    (lambda (V2 V3 V4)
      ((lambda (V1)
         (node (data (color (gaussian V1 25)) (size 0.3))
               (node (data (color (gaussian V2 25)) (size 0.3)))
               (node (data (color (gaussian V3 25)) (size 0.3)))
               (node (data (color (gaussian V4 25)) (size 0.3)))))
       116.5)))
  (uniform-choice
   (flower 213 207 211))
  (flower 220 224 207))
```

5.2.2 Merging identical variables

The abstraction transformation creates a function with variables where the variables represent parts of an expression that differ between specific abstraction instances. It might be the case that variables with different names in the abstraction really ought to be treated as the same variable, i.e., we would like to allow a variable to occur in multiple places within an abstract expression. We can address this with a Deargumentation transform that uses the the following replacement function. This function basically checks that all variable instances are equal or have the same type if they are numbers to allow for some flexibility.

```
(define (same-variable-replacement program abstraction variable
    variable-instances)
  (let* ([possible-match-variables (delete variable (abstraction->vars
      abstraction))])
    (find-matching-variable program abstraction variable-instances
        possible-match-variables)))

(define (find-matching-variable program abstraction variable-instances
    possible-match-variables)
  (define (my-equal? a b)
    (if (and (pair? a) (pair? b))
        (and (my-equal? (car a) (car b)) (my-equal? (cdr a) (cdr b)))
        (if (and (number? a) (number? b))
            #t
            (eq? a b))))
  (if (null? possible-match-variables)
      NO-REPLACEMENT
      (let* ([hypothesis-variable (uniform-draw possible-match-variables)]
             [hypothesis-instances (find-variable-instances program
                 abstraction hypothesis-variable)])
        (if (my-equal? hypothesis-instances variable-instances)
            hypothesis-variable
            (find-matching-variable program abstraction variable-instances
               (delete hypothesis-variable possible-match-variables))))))
```

The basic idea is to check whether two variables take on the same values in each application of the function and if they do, they can be considered as the same variable. Putting this in terms of the Deargumentation transform we start with some variable of an abstraction we are trying to remove and check the other variables to see whether they match. Variables are randomly selected and if one matches it is returned as the definition for the variable that is being removed.

Going back to the tree example, suppose we have the abstraction:

```
(define flower
  (lambda (V1 V2 V3 V4)
    (node (data (color (gaussian V1 25)) (size 0.3))
          (node (data (color (gaussian V2 25)) (size 0.3)))
          (node (data (color (gaussian V3 25)) (size 0.3)))
          (node (data (color (gaussian V4 25)) (size 0.3))))))
```

Applying the Deargumentation transform with the `same-variable-replacment` to variable `V2` can give us a program that generates data similar to the original if the `V2` is matched to a variable that is a "petal", e.g., `V3`.

```
(begin
  (define flower
    (lambda (V1 V3 V4)
      ((lambda (V2)
         (node (data (color (gaussian V1 25)) (size 0.3))
               (node (data (color (gaussian V2 25)) (size 0.3)))
               (node (data (color (gaussian V3 25)) (size 0.3)))
               (node (data (color (gaussian V4 25)) (size 0.3)))))
       V3)))
  (uniform-choice
   (flower 200 207 211))
  (flower 33 224 207))
```

If the variable does not match a petal, e.g. V1, then we get a program that generates data unlike the original program.

```
(begin
  (define flower
    (lambda (V2 V3 V4)
      ((lambda (V1)
         (node (data (color (gaussian V1 25)) (size 0.3))
               (node (data (color (gaussian V2 25)) (size 0.3)))
               (node (data (color (gaussian V3 25)) (size 0.3)))
               (node (data (color (gaussian V4 25)) (size 0.3)))))
       V2)))
  (uniform-choice
   (flower 213 207 211))
  (flower 220 224 207))
```

5.2.3 Inducing recursive functions

We illustrate how recursive patterns can be discovered using Deargumentation and a `replacement-function` called `recursion-replacement` that redefines a variable as a choice between a recursive call and a non-recursive call. The basic idea is to check whether variable instances are calls to the function being deargumented. If they are then there is a recursion. The probability of recursing is a function of how many variable instances are recursive function calls. We also check to see whether this transformation creates an infinite loop. This approach is admittedly very limited in what it can capture, but we use it as a starting point to illustrate how recursion might be identified.

```
(define (recursion-replacement program abstraction variable variable-instances)
  (let* ([valid-variable-instances (remove has-variable? variable-instances)]
         [recursive-calls (filter (curry abstraction-application? abstraction)
             valid-variable-instances)]
         [non-recursive-calls (remove (curry abstraction-application?
             abstraction) valid-variable-instances)]
         [terminates (terminates? program (abstraction->name abstraction)
             non-recursive-calls)])
    (if (or (null? valid-variable-instances) (null? recursive-calls) (not
        terminates))
        NO-REPLACEMENT
        (let* ([prob-of-recursion (/ (length recursive-calls) (length
            valid-variable-instances))])
          `(if (flip ,prob-of-recursion) ,(first recursive-calls)
              (uniform-choice  ,@non-recursive-calls))))))

(define (terminates? program init-abstraction-name non-recursive-calls)
  (define abstraction-statuses (make-hash-table eq?))

  (define (initialize-statuses)
    (let ([abstraction-names (map abstraction->name (program->abstractions
        program))])
      (begin
        (map (curry hash-table-set! abstraction-statuses) abstraction-names
            (make-list (length abstraction-names) 'unchecked))
        (hash-table-set! abstraction-statuses init-abstraction-name
            'checking))))

  (define (status? name)
    (hash-table-ref abstraction-statuses name))
```

```
  (define (set-status! name new-status)
    (hash-table-set! abstraction-statuses name new-status))

  (define (terminating-abstraction? abstraction-name)
    (cond [(eq? (status? abstraction-name) 'terminates) #t]
          [(eq? (status? abstraction-name) 'checking) #f]
          [(eq? (status? abstraction-name) 'unchecked)
           (begin
             (set-status! abstraction-name 'checking)
             (if (base-case? (program->abstraction-body program
                 abstraction-name))
                 (begin
                   (set-status! abstraction-name 'terminates)
                   #t)
                 #f))]))

  (define (base-case? sexpr)
    (cond [(branching? sexpr) (list-or (map base-case?  (get-branches sexpr)))]
          [(application? sexpr) (if (any-abstraction-application? sexpr)
                                    (and (terminating-abstraction? (operator
                                        sexpr)) (all (map base-case? (operands
                                        sexpr))))
                                    (all (map base-case? (operands sexpr))))]
          [else #t]))
  (begin
    (initialize-statuses)
    (list-or (map base-case? non-recursive-calls))))
```

The program we transform is `(begin (node (node a)))` and we can apply the abstraction transformation to get the following:

```
(begin
  (define F1
    (lambda (x)
      (node x))
    (F1 (F1 a))))
```

Now we apply Deargumentation and remove F1's argument. The first step is to change the definition of F1 (via `remove-abstraction-variable`) so that x is drawn from a distribution of past instances of the argument like so:

```
(begin
  (define (F1 x)
    ((lambda (x) (node x)) (if (flip .5) (F1 a) a)))
  (F1 (F1 a)))
```

Here the instances of x (i.e., what was passed into the function F1) are `'(F1 a)`. The final step is to remove the argument from F1 and any applications of F1 resulting in the program using `remove-application-argument`.

```
(begin
  (define (F1)
    ((lambda (x) (node x)) (if (flip .5) (F1) a)))
  (F1))
```

5.2.4 Inducing noisy data constructors

We generated colors for our trees using Gaussians and one motivation for this would be to model the random processes in the environment. There is another, more subtle, reason to introduce a noise process into our programs related to representation. Randomness in data constructors (such as `color`) can potentially allow for compact representations of patterns in the presence of noise. To get a sense of this idea look at figure 3. Without a noisy data constructor (i.e., if there were no Gaussian inside calls to

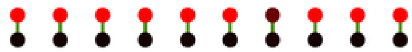


Figure 3: Some trees.

`color`) we might create the following generative model:

```
(if (flip)
 (node (data (color 20) (size .5)) (node (data (color 255) (size .5))))
 (node (data (color 20) (size .5)) (node (data (color 105) (size .5)))))
```

This seems like a big penalty as far as model representation size goes since the probability of drawing the darker colored object may be incredibly small. By having a noisy constructor we can model the data more compactly as simply

```
(node (data (color 20) (size .5)) (node (data (color (gaussian 255 25) (size
    .5)))))
```

The use of the noisy color constructor in our programs was built into the `incorporate-data` function as noted earlier. Here we give an example of how the compactness of noisy constructors might be used to "learn" them from data.

We make the following minor adjustments to `incorporate-data` and `noisy-number-replacement`. The first is to not have `incorporate-data` automatically add a call to `gaussian`.

```
(define (node-data->expression lst)
  `(data (color (gaussian ,(first (second lst)) 25))
         (size ,(first (third lst)))))
```

becomes

```
(define (node-data->expression lst)
  `(data (color ,(first (second lst))) (size ,(first (third lst)))))
```

Instead of having `noisy-number-replacement` return the sample mean in a deargument move we have it return a call to the `gaussian` function and use the sample mean and variance as parameters.

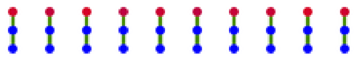
```
(define (noisy-number-replacement program abstraction variable
    variable-instances)
  (if (all (map number? variable-instances))
      (my-mean variable-instances)
      NO-REPLACEMENT))
```

becomes

```
(define (noisy-number-replacement program abstraction variable
    variable-instances)
  (if (all (map number? variable-instances))
      (let* ([instances-mean (my-mean variable-instances)]
             [instances-deviation (sqrt (sample-variance variable-instances))])
        `(gaussian ,instances-mean ,instances-deviation))
      NO-REPLACEMENT))
```

We use the following program to generate three node structures where there is some variance in the color of the third node.

```
(define (three-node shade)
(node (data (color 0) (size .3))
      (node (data (color 0) (size .3))
            (node (data (color (gaussian shade 10)) (size .3))))))

(list (three-node 200) (three-node 200) (three-node 200) (three-node 200)
    (three-node 200) (three-node 200) (three-node 200) (three-node 200)
    (three-node 200) (three-node 200))
```

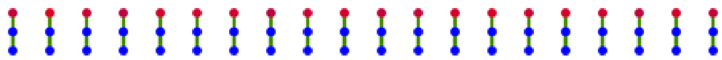


Data incorporation gives us a program that uniformly chooses between ten generated three node structures.

```
(lambda ()
  (uniform-choice
   (node (data (color 0) (size 0.3))
         (node (data (color 0) (size 0.3))
               (node (data (color 209.0) (size 0.3)))))
   (node (data (color 0) (size 0.3))
         (node (data (color 0) (size 0.3))
               (node (data (color 196.0) (size 0.3)))))
                 ⋮
   (node (data (color 0) (size 0.3))
         (node (data (color 0) (size 0.3))
               (node (data (color 206.0) (size 0.3)))))))
```

Bayesian program merging with the above modifications results in a compressed program that creates a function for the three node structure that adds noise to the color of the third node. We ran the system with $\alpha = 3$, beam width 1, and depth 10 for this example.

```
(begin
    (define F2 (lambda (V2) (data (color V2) (size 0.3))))
    (define F1
      (lambda ()
        ((lambda (V1)
           (node (F2 0) (node (F2 0) (node (F2 V1)))))
          (gaussian 202.3 9.286190463980603))))
    (lambda ()
      (uniform-choice (F1) (F1) (F1) (F1) (F1) (F1) (F1)
        (F1) (F1) (F1))))
```



The discussion above hints at the benefits of using probabilistic data constructors and probabilistic programs in general with respect to representation, but understanding the full implications of such a design decision and its impact on program induction are left as future work.

5.2.5 Deargumentation as program induction

In some sense the problem we are trying to address with the Deargumentation is program induction itself. We can view the values for each variable as data generated by some process we would like to identify, i.e., there is some sort of common computation between variable values that we would like to represent as a program. In the case of the noisy-number transformation we can view the generation of values for a single variable as coming from the same process, here we restrict the program representing this process to a Gaussian. Similarly, in the case of the recursive transformation we can view the generation of values for a single variable as coming from a the same process, but we restrict the program representing this process to be the function being deargumented. In the case of the same variable transformation we can view the generation of values for multiple variables as coming from the same process and we implicitly use the program for one variable as the generating process for the other.

Hence, it is natural to ask whether we could reformulate the Deargumentation transform as recursive calls to the Bayesian program merging procedure in an attempt to find programs for generating the values of the variables. One can think of this as recursively squeezing out the randomness in the data where the invented abstractions create a separation between the "structured" parts of the data (the fixed common sub-expressions) and the random parts of the data (the variables in the expression patterns). As an abstraction is applied one gets more data for the random parts and could conceivably attempt to learn a program to generate this data, i.e., separate out even more structure.

6 Examples

6.1 Single color flower

We use the program below to generate ten instances of flower that have the same color.

```
(define (flower shade)
  (node (data (color (gaussian shade 25)) (size .3))
        (petal shade)
        (petal shade)
        (petal shade)))

(define (petal shade)
  (node (data (color (gaussian shade 25)) (size .3))))

(repeat 10 (lambda () (flower 20)))
```

An expression for these ten flowers was created using data incorporation and this expression was compressed into the program below. The function `F1` is a function that takes no arguments and creates a flower with petals that are all the same color. We ran the system with $\alpha = 1$, beam width 1, and depth 10 for this example.

```
(begin
  (define F2
    (lambda (V5)
      (data (color (gaussian V5 25)) (size 0.3))))
  (define F1
    (lambda ()
      ((lambda (V4)
         ((lambda (V2)
            ((lambda (V1)
               ((lambda (V3)
                  (node (F2 V1) (node (F2 V2))
                        (node (F2 V3)) (node (F2 V4))))
                17.2))
             32.9))
          2.0))
       19.7)))
  (lambda ()
    (uniform-choice (F1) (F1) (F1) (F1) (F1) (F1) (F1)
                    (F1) (F1) (F1))))
```

6.2 Multiple color flower

Here we use a similar program to generate ten instances of flower that have alternating colors.

```
(define (flower shade)
  (node (data (color (gaussian 0 25)) (size .3))
        (petal shade)
        (petal shade)
        (petal shade)))

(define (petal shade)
  (node (data (color (gaussian shade 25)) (size .3))))

(repeat 10 (lambda () (flower (if (flip) 100 220))))
```

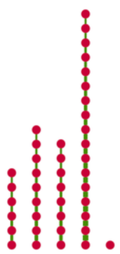
Bayesian program merging results in the following program. The abstraction `F2` corresponds to `petal` and `F1` corresponds to `flower`. We ran the system with $\alpha = 1$ for this example, beam width 1, and depth 10.

```
(begin
  (define F2
    (lambda (V5)
      (data (color (gaussian V5 25)) (size 0.3))))
  (define F1
    (lambda (V2)
      ((lambda (V1)
         ((lambda (V3)
            ((lambda (V4)
               (node (F2 V1) (node (F2 V2)) (node (F2 V3))
                     (node (F2 V4))))
             V2))
          V1))
       V2)))
  (lambda ()
    (uniform-choice (F1 91.0) (F1 85.0) (F1 254.0)
                    (F1 234.0) (F1 82.0) (F1 243.0) (F1 104.0))))
```

6.3 Recursion

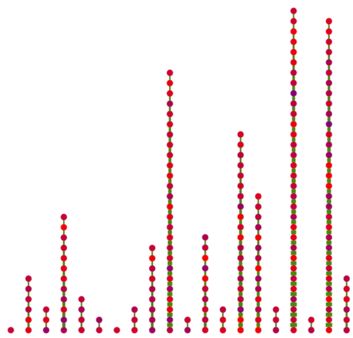
We use the following program to generate a tree made of a single line of nodes. We ran the system with $\alpha = 1$ for this example, beam width 1, and depth 10.

```
(define (line)
  (if (flip .2)
      (node (data (color 200) (size .5)))
      (node (data (color 200) (size .5)) (line))))
```



Bayesian program merging finds the following recursion.

```
(begin
  (define F3 (lambda () (lambda () (F1))))
  (define F2
    (lambda () (data (color (gaussian 200 25)) (size 0.5))))
  (define F1
    (lambda ()
      ((lambda (V1) (node (F2) V1))
       (if (flip 8/9) (F1) (node (F2))))))
  (lambda ()
    (uniform-choice
     (list (F3) (F3) (F3) (F3) (lambda () (node (F2)))))))
```

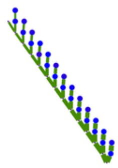


The recursions have to be of a certain length before the tradeoff between prior and likelihood favors the recursive programs, which are usually smaller. We can adjust where this tradeoff happens using the size constant, α, in the prior.

6.4 Vine

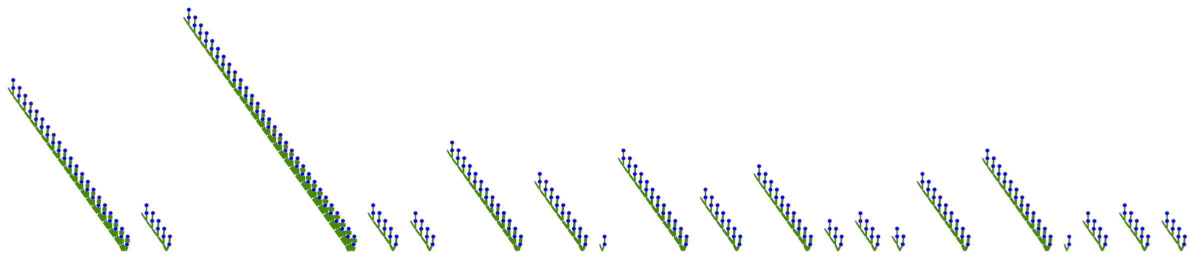
Here we use Bayesian program merging to model a vine with flowers from a single instance. This example demonstrates multiple types of program transformations used on the same data. We ran the system with $\alpha = 1$ for this example, beam width 1, and depth 10.

```
(define (vine)
  (if (flip .1)
      (node (data (color 100) (size .1)))
      (node (data (color 100) (size .1)) (vine) (flower))))

(define (flower)
  (node (data (color (gaussian 20 25)) (size .3))
        (petal 20)
        (petal 20)))

(define (petal shade)
  (node (data (color (gaussian shade 25)) (size .3))))
```



Here we see `F1` corresponds roughly to `vine` and is passed a single color for the flower parts and the size has been fixed within `F1`, `F2` corresponds to the data part of the flower.

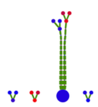
```
(begin
  (define F3 (lambda () (F2 100 0.1)))
  (define F2
    (lambda (V4 V5)
      (data (color (gaussian V4 25)) (size V5))))
  (define F1
    (lambda (V2)
      ((lambda (V3)
         ((lambda (V1)
            (node (F3) V1
                  (node (F2 V2 0.3) (node (F2 V3 0.3)))))
          (if (flip 12/13) (F1 1.0) (node (F3)))))
       7.0)))
  (lambda ()
    (uniform-choice (F1 -14.0))))
```



6.5 Tree

This example demonstrates learning both a parameterized function and a recursion and applying these functions in multiple places within a program. The following functions were used to generate some flower patterns and a tree that consists of two branches each of which ends in a different color flower. We ran the system with $\alpha = 3$, beam width 1, and depth 10 for this example. With $\alpha = 1$ we do not get the right tradeoff between prior and likelihood to make introducing recursion worhtwhile although We expect the system could learn a similar program with $\alpha = 1$ given training examples with more branch instances, since the amount of program compression would increase.

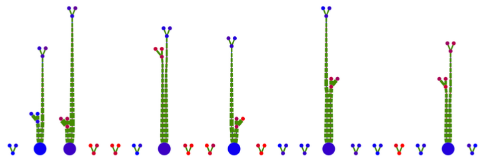
```
(define tree
  (lambda ()
    (uniform-choice
     (node (body) (branch) (branch)))))

(define (body)
  (data (color (gaussian 50 25)) (size 1)))

(define (branch)
  (if (flip .1)
      (uniform-choice (flower 20) (flower 220))
      (node (branch-info) (branch))))

(define (branch-info)
  (data (color (gaussian 100 25)) (size .1)))

(define (flower shade)
  (node (data (color (gaussian shade 25)) (size .3))
        (petal shade)
        (petal shade)))

(define (petal shade)
  (node (data (color (gaussian shade 25)) (size .3))))
```



Bayesian model merging produces a program with a similar structure to the original generating program. `F3` plays a similar role to the `flower` function by taking a single color as argument and creating three nodes of size .3 with the passed in color. `F2` is a function that creates a branch that ends in a flower with either blue petals or red petals.

```
(begin
   (define F4 (lambda (V9 V10) (node (F1 V9 0.1) V10)))
   (define F3
     (lambda (V7)
       ((lambda (V6)
          ((lambda (V8)
             (node (F1 V6 0.3) (node (F1 V7 0.3))
```

```
                    (node (F1 V8 0.3))))
            V7))
         V7)))
(define F2
  (lambda (V3 V4)
    ((lambda (V5) (F4 V3 (F4 V4 V5)))
     (if (flip 9/11)
         (F2 121.0 135.0)
         (uniform-choice (F4 39.0 (F3 47.0)) (F3 187.0))))))
(define F1
  (lambda (V1 V2)
    (data (color (gaussian V1 25)) (size V2))))
(lambda ()
  (uniform-choice (F3 2e1) (F3 235.0)
                  (node (F1 34.0 1) (F2 108.0 99.0) (F2 134.0 85.0))
                  (F3 36.0))))
```



7 Conclusion

We presented Bayesian program merging, an approach to inducing generative models from data. The main idea of this approach is to directly translate the data into a program and to then compress this program by identifying repeated computations. We perform this compression of repeated computation using program transformations. We determine the sequence of transformations by using the posterior probability of the program as a search heuristic.

Future improvements of the system described in this paper include more sophisticated search strategies, more efficient ways of computing the likelihood, and the development of a more robust method for identifying recursive patterns. Another direction is adaptation of this style of probabilistic program induction to the online setting.

More generally, we can ask whether the idea of regularity in data as repeated computation in the generative process can be used to identify other useful program transformations or, whether it can be applied in a more systematic fashion than motivating a collection of disparate search moves.

A different and important question is what happens when the data incorporation step is less immediate and training data is not directly computed in terms of some algebraic data type and whether one can impose some semblance of structure on unstructured data in order to use these methods for learning.

There are many other barriers to overcome before probabilistic program induction can compete with state-of-the-art machine learning algorithms on real world problems, but the increased potential for capturing rich patterns and less dependence on human engineering make research in the subject a worthy pursuit.

References

[1] Martin Bokeloh, Michael Wand, and Hans-Peter Seidel. A connection between partial symmetry and inverse procedural modeling. *ACM Trans. Graph.*, 29(4), 2010.

[2] N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: a language for generative models. *Uncertainty in Artificial Intelligence 2008*, 2008.

[3] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12:23–41, January 1965.

[4] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition, 2003.

[5] Andreas Stolcke and Stephen M. Omohundro. Inducing probabilistic grammars by bayesian model merging. In *Proceedings of the Second International Colloquium on Grammatical Inference and Applications*, pages 106–118, London, UK, 1994. Springer-Verlag.

[6] A. Stuhlmueller, J. B. Tenenbaum, and N. D. Goodman. Learning structured generative concepts. *Proceedings of the Thirty-Second Annual Conference of the Cognitive Science Society*, 2010.