International Conference on Computational Science, ICCS 2010

Toward Interactive Statistical Modeling

Sooraj Bhat[a,1], Ashish Agarwal[b], Alexander Gray[a], Richard Vuduc[a]

[a]College of Computing, Georgia Institute of Technology, Atlanta, GA
[b]Department of Computer Science, Yale University, New Haven, CT

Abstract

When solving machine learning problems, there is currently little automated support for easily experimenting with alternative statistical models or solution strategies. This is because this activity often requires expertise from several different fields (e.g., statistics, optimization, linear algebra), and the level of formalism required for automation is much higher than for a human solving problems on paper. We present a system toward addressing these issues, which we achieve by (1) formalizing a type theory for probability and optimization, and (2) providing an interactive rewrite system for applying problem reformulation theorems. Automating solution strategies this way enables not only manual experimentation but also higher-level, automated activities, such as autotuning.

Keywords:
machine learning, algorithm derivation, interactive modeling, type theory

1. Introduction

Machine learning[2] has become a well-established field whose techniques are widely used in many disciplines across science and engineering. Unfortunately, the programmability of machine learning algorithms has not kept pace with its adoption. Ideally, practitioners should be able to easily experiment with different statistical models as well as alternative algorithms for solving them, but currently this is not the case. Certainly, there are many toolkits available that solve specific subproblems, but several hurdles remain. Because APIs commonly provide efficient algorithms for only a few specific classes of problems, it is necessary to reformulate your instance to conform, if possible. These reformulations often can only be performed manually, which is time-consuming and error-prone. Even after finding a suitable toolkit, later modifying your model may inadvertently change the problem class, which requires finding a new toolkit or even a custom solution. Opportunities for closed-form solutions (usually more efficient) still must be identified manually, which is hard to do pervasively when using toolkits as black-boxes. Furthermore, it is not uncommon for a single machine learning algorithm to require techniques (and thus toolkits) from several different fields, thus exacerbating these hurdles. Toward addressing these issues, we have implemented a system for declaratively specifying input problems and then interactively deriving correct and efficient algorithms for solving them via iterative problem reformulation. Specifically our contributions are the following:

---

[1]Corresponding author. *Email address:* sooraj@cc.gatech.edu
[2]We will refer to machine learning, multivariate statistics, and data mining interchangeably for this paper.

```
θ_curr := <initialize_randomly>;          (μ_0,μ_1) := <initialize_randomly>;
while (<convergence_check>) {             while (<convergence_check>) {
   l(θ) := E_{z~f(Z|X=X̃,Θ=θ_curr)}[          for i = 1 to N do
            log f(X = X̃, Z = z | Θ = θ)]        γ_i := φ(X̃_i;μ_1,1)/(φ(X̃_i;μ_0,1)+φ(X̃_i;μ_1,1));
   θ_curr := arg max_θ l(θ)                   μ_0 := Σ_{i=1}^{N}(1-γ_i)*X̃_i / Σ_{i=1}^{N}(1-γ_i);
}                                            μ_1 := Σ_{i=1}^{N} γ_i*X̃_i / Σ_{i=1}^{N} γ_i;
return θ_curr;                            }
                                          return (μ_0,μ_1);
```

Figure 1: (a) General form of the EM algorithm. (b) EM for the mixture of normals example. $\phi(x; \mu, \sigma^2)$ is the probability density at x of a normal distribution with parameters μ and σ^2.

- We have developed a type-theoretic formalization (syntax, type system, semantics) of a language with constructs for probability and optimization (Section 3). This provides two benefits: (1) it conforms to existing practice of programming language development, allowing us to reap the same benefits of composability and modularity, and (2) it aligns us with modern theorem provers.

- We have implemented an interactive rewrite system for applying problem reformulation theorems (or, *schemas*) that supports the derivation of efficient algorithms that are correct by construction (Section 4). Schemas are written by algorithm designers and describe when and how an expression can be rewritten to an equivalent formulation. Efficiency of the resulting algorithm is achieved through schemas which express a more sophisticated rewrite. Correctness of the derivation relies on the correctness of the individual schema.

Section 4 also discusses problems solved using our system. Our initial target users are machine learning practitioners. As our system and underlying theory matures, we would like to increase the usability of our system and reduce the amount of prerequisite knowledge in order to use it, so that scientists in other disciplines will more easily be able to leverage powerful machine learning techniques. The relation to previous work, notably AutoBayes [1, 2], is covered in Section 5.

2. Motivating Example: Maximum Likelihood Estimation

We will use *maximum likelihood estimation* (MLE) as our motivating example. Parameter estimation is the activity of finding parameters for a statistical model which "best explain" observed data; in MLE, the "best" parameters are the ones which maximize the likelihood function. For a model with parameter Θ, random variable X, and observed data $\widetilde{X}$, the likelihood is simply a function of θ, equal to the probability density at $\widetilde{X}$, assuming Θ to be equal to the supplied θ: $L(\theta) \triangleq f(X = \widetilde{X} | \Theta = \theta)$. MLE can thus be summarized as $\theta_{\text{best}} = \arg \max_\theta L(\theta)$. Closed-form solutions for θ_{best} exist for simple models. Take for example a dataset of numbers $\widetilde{X}_1, \ldots, \widetilde{X}_N$ which we believe are drawn independently and identically from a normal distribution, whose parameters (mean μ and variance σ^2) we would like to estimate. With some calculus we can show that $\mu_{\text{best}} = \frac{1}{N} \sum_{i=1}^{N} \widetilde{X}_i$ and $\sigma^2_{\text{best}} = \frac{1}{N} \sum_{i=1}^{N} (\widetilde{X}_i - \mu_{\text{best}})^2$.

Now suppose we change our minds (perhaps based on visualizing the data) and decide to model these same points as if drawn from a mixture of two normals. This process can be understood as flipping a coin to select which normal to use and then drawing a point from the selected normal (for simplicity we will consider an unbiased coin and unit variance for the normals). Suddenly, there does not exist a closed form for the MLE estimates. In fact, we have to use an entirely different algorithm, called the *expectation maximization* (EM) algorithm. More accurately, EM is an algorithm template; different models have different instantiations of the algorithm. EM leverages the fact that maximizing the likelihood is more tractable if certain hidden variables Z are assumed to be known (in our example, these are the results of the coin flip, $Z_i \in \{0, 1\}$, for each $\widetilde{X}_i$). In essence, EM iteratively alternates between fabricating values for the hidden variables and using those values to re-estimate the parameters. EM is numerically robust and is guaranteed to make progress toward a local maximum. If at this point we decided to further revise our model, say to a mixture between a normal and an exponential distribution, we would again have to derive an algorithm from scratch. Clearly, automation is called for.

3. A Language for Probability and Optimization

We choose to develop our formalization in the context of type theory. There are several reasons for this. First, type theory disciplines our language design; it obligates us to precisely capture the semantics of constructs such as probability distributions. Second it promotes correctness by ruling out ill-behaved programs by definition. Third, we wish to develop a system in which it will be possible to mechanically prove schemas correct, rather than trusting schema writers.[3] Types are a path to constructing proofs as demonstrated by modern theorem proving systems such as Coq [4] or Isabelle [5], which are grounded in advanced dependent type theories. We envision implementing our language directly in these systems and proving schemas correct by proving the associated reformulation theorems, leveraging the existing ecosystem of automated theorem proving. The following is the syntax of our language:

$$T ::= \mathtt{bool} \mid \mathtt{int} \mid \mathtt{real} \mid T_1 \times \ldots \times T_n \mid T \to T \mid \mathtt{prob}\, T \mid \mathtt{pdf}\, T \qquad \text{types} \qquad (1a)$$

$$E ::= x \mid \lambda x : T \,.\, E \mid E\, E \mid \mathtt{fix}\, E \qquad\qquad\qquad\quad \text{functional core} \qquad (1b)$$

$$\mid \mathtt{true} \mid \mathtt{false} \mid \neg E \mid E \vee E \mid E \wedge E \qquad\qquad\qquad \text{Booleans} \qquad (1c)$$

$$\mid r \mid E + E \mid E \times E \mid E^E \mid \log E \qquad\qquad\qquad\qquad \text{reals} \qquad (1d)$$

$$\mid (E_1, \ldots, E_n) \mid E.k \qquad\qquad\qquad\qquad\qquad\qquad\quad \text{tuples} \qquad (1e)$$

$$\mid \mathtt{if}\, E\, \mathtt{then}\, E\, \mathtt{else}\, E \mid E = E \mid E \le E \qquad\qquad\qquad \text{relational} \qquad (1f)$$

$$\mid \mathtt{bernoulli}\, E \mid \mathtt{normal}\, E\, E \mid \ldots \mid \mathtt{delta}\, E \qquad\qquad \text{distributions} \qquad (1g)$$

$$\mid \mathtt{let}\, x \sim E\, \mathtt{in}\, E \qquad\qquad\qquad\qquad\qquad\quad \text{random variables} \qquad (1h)$$

$$\mid \mathbb{E}_{x \sim E}[E] \mid \mathtt{condition}\, E\, \mathtt{in}\, E \mid \mathtt{pdf}\, E \qquad\qquad\quad \text{probability} \qquad (1i)$$

$$\mid \mathtt{argmax}_{x_1 : T_1, \ldots, x_n : T_n}\{E \mid E\} \qquad\qquad\qquad\qquad\quad \text{optimization} \qquad (1j)$$

This language is an extension of our previous work on optimization [6]. The syntax being defined is called the *abstract syntax*, which is meant to capture the core mathematical principles. This is distinct from *concrete syntax*, which defines how programs are represented in text. Our focus is always on the abstract syntax. Next, we define which phrases in the language are well-formed and what they mean. We use a type system to reject ill-formed, nonsensical expressions. We make use of *judgments*, which are a commonly used notation for describing typing and evaluation relations (see Equation 2 for an example). Judgments can be read as if-then rules; if the statement above the bar is true, then we can conclude that the statement below the bar holds.

3.1. General features

We define general features (Equations 1a–1f, except $\mathtt{prob}\, T$ and $\mathtt{pdf}\, T$, discussed next) in a standard way [7], with some minor novelties. We define $\mathtt{int}$ to be a subtype of $\mathtt{real}$ and have implemented *depth subtyping* for products. This allows using a pair of integers when a pair of reals is expected, for instance. The meta-variable r refers to numeric literals (such as 1729 and −40) and includes the mathematical constants e and π. Our programs involve real numbers, which raises the issue of computing over them. This is a fundamental challenge being pursued by others in various contexts [8, 9] and, while beyond the scope of our current investigations, is a topic we wish to support.

3.2. Probability

Measure theory [10] is the basis of modern probability theory and unifies the concepts of discrete and continuous probability distributions. It develops a principled way of assigning sizes to subsets of interest. We develop our formalization within this framework.

Take a set X. A σ-algebra Σ over X is a nonempty set of subsets of X (including X), closed under complement and countable union. In the context of probability, Σ represents the set of events we care to consider. Function μ is a probability measure if: $\mu(\emptyset) = 0$; $\mu(X) = 1$; $\mu(E) \ge 0$ for $E \in \Sigma$; and $\mu(\bigcup_{i \in I} E_i) = \sum_{i \in I} \mu(E_i)$ for a countable number of pairwise disjoint sets $E_i \in \Sigma$. The type $\mathtt{prob}\, T$ is the type of probability measures over the type T; the corresponding

---

[3]All compilers suffer from this issue: the correctness of a compiler is always subject to the correctness of the program transformations implemented in the compiler. The exception is *verified compilers*, such as Compcert [3], which formally verify the transformations.

σ-algebra is implicitly defined. Measures over product spaces `prob` $(T_1 \times \ldots \times T_n)$ represent joint distributions. The type `pdf` T is the type of probability densities over T. Specifically, it is the type of Radon-Nikodym derivatives over T with respect to the (implicitly defined) underlying measure. The term *probability density* has a broad meaning here; it covers the cases of countable types (where it refers to the probability mass function) and "mixed" cases, such as products that contain both discrete *and* continuous types. A good overview of these concepts is available in [11].

Constructs from Equation 1g correspond to the base measures available in the language, such as the normal or Bernoulli distributions (we model the latter as a measure over Booleans, instead of $\{0, 1\}$) . The expression `delta` E constructs a "singleton" measure, where all probability mass is centered on E. The construct `let` $x \sim E_1$ `in` E_2 introduces random variables, just as `let` $x = E_1$ `in` E_2 introduces regular variables. The latter form is often found in functional programming languages; we have not provided it in the syntax here, but we will define a syntactic shorthand for it later. In the latter, subsequent occurrences of x can be freely replaced by E_1. However, as we will see, a random variable x cannot be replaced by the distribution characterizing it. On the other hand, these two kinds of variables are similar in that they are known quantities, albeit a functional one in the case of random variables. In contrast, variables introduced by `argmax` are those whose values are unknown and must be searched for using an appropriate algorithm. The set of probability measures, together with the `delta` and `let` constructs, forms a mathematical structure from category theory known as a *monad*, and our formalization follows the standard monadic treatment, developed as early as Giry [12] and eloquently summarized by Ramsey & Pfeffer [11]. This approach is desirable because monads are a well understood structure with existing theory and applications to programming languages. To discuss well-formedness, we introduce the relation $\Gamma \vdash E : T$, which holds if E has the type T under the context Γ, where Γ is simply a list of variable names and their associated types. The typing rule for `let` is interesting:

$$\frac{\Gamma \vdash E_1 : \text{prob } T_1 \quad \Gamma, x : T_1 \vdash E_2 : \text{prob } T_2}{\Gamma \vdash \text{let } x \sim E_1 \text{ in } E_2 : \text{prob } T_2} \tag{2}$$

Random variable x is distributed according to a probability measure E_1 of type `prob` T_1 and is then assumed to have type T_1 in the body E_2, i.e., it is given the same type as a *non*-probabilistic variable of type T_1. Intuitively, this is allowed because E_2 is forced to be a measure, so x can never "escape" a probabilistic context. To discuss the meaning of expressions, we introduce the notation $[\![E]\!]$, which for an expression E in our language represents the meaning, or *denotation*, of that expression in terms of usual math. The meaning of the `let` construct is

$$[\![\text{let } x \sim E_1 \text{ in } E_2]\!] = \lambda R \,\textbf{.}\, \int_R [\![\lambda x : T_1 \,\textbf{.}\, E_2]\!] \, d[\![E_1]\!] \tag{3}$$

where $R \subseteq T_2$. The definition is given in terms of *abstract integration*. Abstract integration of a function f w.r.t. a measure μ is written $\int f \, d\mu$. The right hand side is a measure: it takes a region of interest R, and returns its "size", after factoring in the contribution of E_1. Expectation has a similar typing rule

$$\frac{\Gamma \vdash E_1 : \text{prob } T \quad \Gamma, x : T \vdash E_2 : \text{real}}{\Gamma \vdash \mathbb{E}_{x \sim E_1}[E_2] : \text{real}} \tag{4}$$

with the only difference being that E_2 must be a `real` expression. Its definition is standard:

$$[\![\mathbb{E}_{x \sim E_1}[E_2]]\!] = \int [\![\lambda x : T \,\textbf{.}\, E_2]\!] \, d[\![E_1]\!] \tag{5}$$

The expression `condition` E_1 `in` E_2 has type `prob` T when E_1 has type `bool` and E_2 has type `prob` T. The definition is equally straightforward:

$$[\![\text{condition } E_1 \text{in } E_2]\!] = \begin{cases} [\![E_2]\!] & \text{when } [\![E_1]\!] \text{ is true} \\ \lambda R \,\textbf{.}\, 0 & \text{otherwise} \end{cases} \tag{6}$$

When the guard of the conditional is false, the definition returns the "zero measure", i.e., a measure that always returns zero. To represent densities and likelihoods, we have the construct `pdf` E which is well formed when E has type `prob` T and is defined to be the Radon-Nikodym derivative of the probability measure E. A density can be applied to an argument of compatible type, yielding a real number, and is equal to the density of E_1 at E_2.

3.3. Optimization

Our previous work on formalizing optimization supported expressive constraints and reformulations of those constraints to standard forms [6]; however, the `argmax` construct was not nestable within other expressions, a feature necessary for the EM algorithm. To this end, we introduce a new typing rule for `argmax`:

$$\frac{\Gamma, x_1 : T_1, \ldots, x_n : T_n \vdash E_1 : \texttt{real} \quad \Gamma, x_1 : T_1, \ldots, x_n : T_n \vdash E_2 : \texttt{bool}}{\Gamma \vdash \texttt{argmax}_{x_1:T_1,\ldots,x_n:T_n}\{E_1 \mid E_2\} : T_1 \times \ldots \times T_n} \tag{7}$$

with the semantics being that `argmax` evaluates to the value in $T_1 \times \ldots \times T_n$ that makes the objective E_1 reach its maximum attainable value, under the constraint that E_2 must be true.

3.4. Homogeneity of source and target languages

As will be seen in the next section, the source and target languages for our program transformations are the same, unlike typical compilers. We take a declarative specification of a problem which may contain constructs like `argmax` which need to be "solved for", and iterate towards a reformulation which only contains fragments in the functional, or "executable", subset of the language. The primary benefit of this approach is that it allows us to cleanly represent and interactively inspect intermediate stages of derivation, mixing declarative and functional, without any semantic problems. Doing this with a target language—say, C++—with different semantics than our source language would be problematic, as expressions in C++ have a different meaning than in math, thus many reformulation theorems we are accustomed to become invalid. This ability to elegantly support partial derivations is useful not only for human inspection but also for automated procedures performing algorithm search. These procedures would need to reason about the search tree of possible algorithms, the internal nodes of which are partial derivations.

4. Interactively Rewriting Problems

Having defined the language, we now have a representation for reasoning about problems involving probability and optimization. In this section we discuss the system we have implemented that allows users to iteratively and interactively reformulate problems to an executable form. Because of the flexibility of our input language, we do not yet completely understand how to accomplish fully automatic algorithm derivation for arbitrary input programs. Interactivity is thus an indispensable feature not only for assisted algorithm derivation, but also for debugging schemas and automated derivation procedures.

4.1. Schemas

Schemas are a core notion in our system. A schema is a rewrite rule that embodies a problem reformulation theorem. Logically, a schema represents a theorem that an expression can be reformulated into another expression, e.g., $\forall a, b : \texttt{real}, a \times b = 0 \mapsto a = 0 \vee b = 0$. The theorem may have a precondition which states when the reformulation is applicable. Operationally, a schema is a metalevel operation that takes an expression and returns a reformulated expression, throwing an exception (*failing*) in cases where the precondition is not met. In our system, schemas are implemented as functions in the OCaml programming language.

We have implemented over 100 schemas, covering topics such as basic arithmetic and computer algebra, Boolean logic, equation manipulation (e.g., factoring, root finding), calculus, probability, and optimization. For example, consider the following schemas for the `let` construct:

$$\texttt{let } x \sim (\texttt{delta } E_1) \texttt{ in } E_2 \mapsto E_2[x := E_1] \qquad \texttt{let\_1} \tag{8}$$

$$\texttt{let } x \sim E \texttt{ in } (\texttt{delta } x) \mapsto E \qquad \texttt{let\_2} \tag{9}$$

$$\texttt{let } y \sim (\texttt{let } x \sim E \texttt{ in } E_x) \texttt{ in } E_y \mapsto \texttt{let } x \sim E \texttt{ in } (\texttt{let } y \sim E_x \texttt{ in } E_y) \qquad \texttt{let\_3} \tag{10}$$

We assume all expressions are well-formed. These schemas correspond directly to the three monad laws, in the context of probability measures. The first one states that if a random variable comes from a measure that can only take on one value, then you can replace all occurrences of the variable with that value. The second one essentially states that the delta measure is the identity of the `let` construct. The last one states that a nested random variable

x used to define another random variable *y* can instead be defined beforehand; this gives us the power to linearize definitions of models, which will turn out to be a handy operation. This schema is correct even when *x* and *y* are the same identifier; lexical scoping guarantees that uses of the identifier occurring in E_y (after the rewrite) will refer to the innermost definition, which in this case is *y*. Due to the first law, we define `let` $x = E_1$ `in` E_2 to be shorthand for `let` $x \sim$ `delta` E_1 `in` E_2.

4.2. Schema combinators

In addition to schemas which represent reformulations, we also provide *schema combinators* which take schemas as input and return a new schema. These are implemented as higher-order OCaml functions. For instance, the schema `s1 <|> s2` sets up a pair of alternative schemas to try: it first tries to apply the schema `s1`, and tries to apply `s2` if and only if `s1` fails. Additionally, `rewrite s` returns a new schema that repeatedly applies `s` to an expression in a bottom-up fashion until no further applications of `s` are possible. These combinators provide a basic facility for authoring more complicated schemas; more mature rewriting schemes are possible [13, 14]. We can combine the schemas from before: `let let_simpl = rewrite (let_1 <|> let_2 <|> let_3)`. This is a schema that simplifies and normalizes models to a linearized form where all the random variables are in the same scope. This supports important operations such as finding all the hidden variables of a model to complete a joint distribution, a necessary step of the EM algorithm.

Currently, the process of iterative problem reformulation is largely manual, supported by some automated schemas authored in the combinator style. Much like computer algebra systems, we have automated reformulation for certain subsets of the input language. As our understanding of problem classes deepens over time, we can incorporate more general procedures.

4.3. The interactive top-level

We have implemented our interactive system in the functional language OCaml, which is well-suited to the task of language analysis. Our interactive top-level is simply a specially loaded OCaml top-level interpreter. We load the first example from Section 2:

```
# load one_normal;;
- : Syntax.expr =
argmax{mu : real, ss : real}{
  pdf (normal mu ss) 99 * pdf (normal mu ss) 281
  * pdf (normal mu ss) 111 | 0 <= ss}
```

The function `load` loads the example into a the session manager, which allows undo and replay of commands. The example is just an OCaml value representing its abstract syntax tree (AST). The current state of the derivation is printed after each command via a custom pretty-printer; we have edited the whitespace of the output for clarity. We restrict the examples to a dataset of three numbers for ease of exposition. In this case, the user has already written out the factored likelihood. The optimization constrains the variance σ^2 to be positive.

We implement schemas as OCaml functions, and entire modules of schemas are available to the user through OCaml's namespace mechanism. For convenience, they are imported automatically on startup. Now we can start entering commands:

```
# apply(rewrite pdf_base <&> argmax_log <&> log_simpl <&> argmax_add);;
- : Syntax.expr =
argmax{mu : real, ss : real}{
  -1.500000 * log ss + (99 - mu)^2/ss * -0.500000 + (281 - mu)^2/ss *
  -0.500000 + (111 - mu)^2/ss * -0.500000 | 0 <= ss}
```

Here we have sequenced schemas with the `<&>` combinator. This series of rewrites unfolds the definition of probability density for basic measures, takes the logarithm of the objective function, simplifies the resulting expression, and drops constants from the maximization. Additionally, a simplification schema, consisting of basic numeric, Boolean, and equation simplifications, is run at the end of any schema entered at the prompt. Next,

```
# apply(gradient_opt <&> rewrite undistr <&> rewrite factors_0);;
- : Syntax.expr =
argmax{mu : real, ss : real}{
  -1.500000 * log ss + (99 - mu)^2/ss * -0.500000 + (281 - mu)^2/ss *
```

```
   -0.500000 + (111 - mu)^2/ss * -0.500000
 | 0 <= ss && 0 = 491 + -3 * mu && 0 = -1.500000 * ss + (99 - mu)^2 *
   0.500000 + (281 - mu)^2 * 0.500000 + (111 - mu)^2 * 0.500000}
```

This series of rewrites enforces the constraint that the maximum must occur at a critical point, and then factors the resulting constraint. Being able to treat constraints as formal objects enables this kind of expressive constraint manipulation. Finally,

```
# apply(rewrite (linear_root <|> subst_known));;
- : Syntax.expr =
argmax{mu : real, ss : real}{
  -1.500000 * log ss + (99 - mu)^2/ss * -0.500000 + (281 - mu)^2/ss *
  -0.500000 + (111 - mu)^2/ss * -0.500000
  | mu = 163.666667 && ss = 6907.555556}
```

Here we can finish by repeatedly alternating between solving linear equations and substituting known variables (i.e., of the form $x = r$ for real number r) back into the constraint. As we can see, the optimization constraint is now a conjunction of equality constraints, which represents the solution of the parameters. Now, we would like to update our model to be a mixture of two normals. Solving this requires the EM algorithm, which we describe next.

4.4. The Expectation-Maximization schema

As sketched in Figure 1, the EM algorithm is an iterative algorithm with one major operation performed each iteration: maximizing the expectation of a joint likelihood. The following is the OCaml function that generates this step:

```
let em_step e = match e with
  | Argmax(context, App(Pdf model, x_obs), constraint) ->
      let model' = de_shadow (let_simpl model) in
      let theta  = names context in
      let theta' = fresh_n theta () in
      let joint  = make_joint model' in
      let condl  = make_conditional model' in
      let condl' = substs (List.map var theta') theta condl in
      let z      = fresh "Z" () in
      let body = Expect(z,condl',Log (App(Pdf joint,Tup[x_obs; Var z]))) in
        Argmax(context, body, constraint)
  | _ -> fail "Not the maximization of a likelihood."
```

This is not a schema by itself (it does not represent a reformulation between the input and output arguments) but is the primary component in the general EM schema. First, we pattern match on the expression to see if it is the maximization of a likelihood. Next we normalize the model and rename variables so no name-shadowing occurs. Then, we extract the names of the parameters θ and generate a unique copy for use as θ_{curr}. The function `make_joint` scans a model with hidden random variables and returns the corresponding joint measure. The next two lines construct the conditional distribution of the hidden variables on the observed data and substitute in the correct copy of parameters to use, yielding $f(Z|X = \widetilde{X}, \Theta = \theta_{\text{curr}})$. Finally, all the components are arranged (`App` and `Tup` are the AST constructors for function application and tuple construction). We load the mixture of normals example:

```
# load mixture_of_normals;;
- : Syntax.expr =
argmax{mu0 : real, mu1 : real}{
  pdf (let pick = let z ~ bernoulli 0.500000 in
                  if z then normal mu1 1 else normal mu0 1 end in
       let x1 ~ pick in
       let x2 ~ pick in
       let x3 ~ pick in
         delta (x1, x2, x3))
  (99, 281, 111)}
```

We select `mu1` when `z` is true in order to be consistent with the example in Figure 1 and the intuition that `true` should correspond to `1`. Here we can see compositional construction of models; `pick` represents the model of a single number and is subsequently used to model the dataset. The typical pattern for constructing a model is to have a series of `let`

expressions ending with a `delta` expression that "returns" the quantity of interest; hidden variables are those that do not get returned (e.g., the `z` in `pick`). We now apply the EM schema:

```
# apply em_alg;;
- : Syntax.expr =
letrec em_func mu0' mu1' {...} =
  if {...}
  then (mu0',mu1')
  else em_func argmax{mu0 : real, mu1 : real}{
     E{Z ~ let z ~ bernoulli 0.500000 in
            let x1 ~ if z then normal mu1' 1 else normal mu0' 1 end in
            let z1 ~ bernoulli 0.500000 in
            let x2 ~ if z1 then normal mu1' 1 else normal mu0' 1 end in
            let z0 ~ bernoulli 0.500000 in
            let x3 ~ if z0 then normal mu1' 1 else normal mu0' 1 end in
              condition (x1, x2, x3) = (99, 281, 111) in delta (z0, z1, z)}[
       log (pdf (let z ~ bernoulli 0.500000 in
                 let x1 ~ if z then normal mu1 1 else normal mu0 1 end in
                 let z1 ~ bernoulli 0.500000 in
                 let x2 ~ if z1 then normal mu1 1 else normal mu0 1 end in
                 let z0 ~ bernoulli 0.500000 in
                 let x3 ~ if z0 then normal mu1 1 else normal mu0 1 end in
                   delta ((x1, x2, x3), (z0, z1, z)))
              ((99, 281, 111), Z))]} in
  em_func 0.231 0.395
```

First, because we do not have while-loops in our language, we formulate EM as a recursive algorithm. We do this with our fixpoint operator `fix` and the standard shorthand for defining recursive functions, `letrec` $f\ x_1\ \ldots\ x_n = e \equiv$ `fix` $(\lambda\ f\ x_1\ \ldots\ x_n\ .\ e)$. Distracting bookkeeping (denoted by `{...}`) necessary for the termination criteria has been elided. The EM algorithm requires the initial estimate of the parameter to be randomized. For now, we seed the initial estimates with values that are randomly generated at derivation time, instead of at run-time. This is a convenient means for achieving randomization in the context of a purely functional language such as ours. A more proper solution that we plan to implement involves adding to the language the necessary machinery for representing randomization (while maintaining purity); this increases the complexity of the reformulation and is beyond the scope of this example. We see that the model basically appears twice in the code; this is done for ease of analysis. In general, we have found a tension between representations good for analysis versus those good for interactive usability, an issue not present for typical compilers. This is a topic we wish to explore further.

We can see the benefit of having a representation for mathematical expressions: we have a way to once-and-for-all state the EM algorithm. All that differs between applications of EM to different models are the details from this point forward about how to expand the definitions of expectation and density functions. The latter steps resemble the schemas we used for the first example. So far, our system has the capability to find closed form solutions for MLE of basic measures and a few simple mixtures of these basic measures, all fully manually. However, similar problems generally have similar derivation scripts (lists of schemas), so taking an existing script and adapting it to solve a closely related model is not overly burdensome in practice.

5. Related & Future Work

The pioneering AutoBayes system automated the derivation of maximum likelihood as well as Bayesian estimators for a significant class of statistical models [1, 2]. It is written in Prolog and also implements a *schema-based synthesis* approach to code generation which also focuses on correctness by construction. It covers a much wider variety of statistical models than our system and can generate efficient, commented C++ code as output. Our system represents an alternative exploration of the design space of statistical derivation systems, designed to be interactive (as opposed to fully automated), strictly logical (AutoBayes makes use of Prolog's extra-logical `cut` operator, whereas we use schema combinators), and fully typed.

SPIRAL is a successful system that derives tuned, high-performance implementations of linear digital signal processing transforms via feedback-driven optimization of the generated code and exploitation of the mathematical structure of the transforms [15]. As our system, it exploits domain specific knowledge in order to apply desired

reformulations. It considers a narrower niche of mathematics than our system, but does so to great effect. Also, its focus is not on algorithmic exploration by the user (which is often desirable in a machine learning setting), but rather aggressive optimization. As we expand our spectrum of schemas from manual towards automatic, we hope to incorporate SPIRAL-style feedback-based optimization.

FLAME is a methodology for systematically deriving high-performance algorithms for a class of dense linear algebra operations [16]. It simultaneously achieves correctness and efficiency: a constructive proof of correctness accompanies each derivation, and the derivation scheme formalizes and employs well-established heuristics for how to decompose matrix problems in terms of submatrices. It focuses on a single pattern of reformulation common to many high-performance linear algebra algorithms. FLAME inspires some correctness-ensuring aspects of our proposed system.

There is a variety of work that uses the monadic framework as a way to structure ordinary libraries that implement a shallow embedding of probabilistic reasoning into a host language [17, 18, 19, 11]. These libraries do not implement a new language, but instead provide an API for an existing language, such as OCaml or Haskell, that allows for naturally expressing probabilistic computations, such as sampling or exact inference of discrete probability distributions. The usability comes from the monadic structure. This approach is beneficial in that it allows them to inherit features from the host language, such as type-checking and interoperability with other existing libraries. Unfortunately, it also comes at a price; these toolkits cannot reason about continuous probability distributions or perform deep analysis, such as reformulating, via the EM algorithm, a computation stated in the host language. Our approach incorporates both features.

The Probabilistic Modeling Toolkit (PMTK) is a Matlab library that supports a variety of models and priors for Bayesian inference and MAP/MLE estimation [20]. It gives users the ability to directly construct *graphical models*, which allows for deeper analysis than other libraries, precisely because the graphical models serve as a language representation, comparable to the abstract syntax trees of our system. PMTK also supports the EM algorithm, but the user needs to provide an implementation of the main iteration steps for any model outside of what has been defined in their library.

A primary outcome of formalizing mathematical problems in the type-theoretic style is the ability to formally and mechanically manipulate them. This is an important ability for interactive exploration, but to fully leverage this new ability, we hope to move in the direction of autotuning [21] in order to generate machine learning algorithms that are not only computationally efficient (as is the common use-case for autotuning), but also of high quality (e.g., achieve low classification error).

5.1. Acknowledgments

References

[1] B. Fischer, J. Schumann, AutoBayes: A system for generating data analysis programs from statistical models, Journal of Functional Programming 13 (03) (2003) 483–508.

[2] A. G. Gray, B. Fischer, J. Schumann, W. Buntine, Automatic Derivation of Statistical Algorithms: The EM Family and Beyond, in: S. Becker, S. Thrun, K. Obermayer (Eds.), Advances in Neural Information Processing Systems (NIPS) 15 (Dec 2002), MIT Press, 2003.

[3] X. Leroy, Formal verification of a realistic compiler, Communications of the ACM 52 (7) (2009) 107–115.

[4] B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J. Filliâtre, E. Giménez, H. Herbelin, et al., The Coq proof assistant reference manual, INRIA, version 6 (11).

[5] T. Nipkow, L. Paulson, M. Wenzel, Isabelle/HOL–A Proof Assistant for Higher-Order Logic, volume 2283 of LNCS (2002).

[6] A. Agarwal, S. Bhat, A. Gray, I. E. Grossmann, Automating mathematical program transformations, in: Practical Aspects of Declarative Languages (PADL), 2010.

[7] B. Pierce, Types and programming languages, The MIT Press, 2002.

[8] P. Potts, A. Edalat, M. Escardo, Semantics of exact real arithmetic, in: LICS '97., 12th Annual IEEE Symp. on Logic in Comp. Sci., Warsaw, 1997, pp. 248–257.

[9] A. Nanevski, G. Blelloch, R. Harper, Automatic generation of staged geometric predicates, in: Proceedings of the sixth ACM SIGPLAN International Conference on Functional programming, ICFP 2001, ACM, Florence, Italy, 2001, pp. 217–228.

[10] W. Rudin, J. Cofman, Principles of mathematical analysis, McGraw-Hill New York, 1964.

[11] N. Ramsey, A. Pfeffer, Stochastic lambda calculus and monads of probability distributions, ACM SIGPLAN Notices 37 (1) (2002) 154–165.

[12] M. Giry, A categorical approach to probability theory, Categorical Aspects of Topology and Analysis 915.

[13] E. Visser, Stratego: A language for program transformation based on rewriting strategies, Lecture Notes in Computer Science 2051 (2001) 357–362.

[14] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martı-Oliet, J. Meseguer, J. Quesada, Maude: Specification and programming in rewriting logic, Theoretical Computer Science 285 (2) (2002) 187–243.

[15] M. Puschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, et al., SPIRAL: Code generation for DSP transforms, Proceedings of the IEEE 93 (2) (2005) 232–275.

[16] P. Bientinesi, J. A. Gunnels, M. E. Myers, E. S. Quintana-Ortí, R. A. van de Geijn, The science of deriving dense linear algebra algorithms, ACM Transactions on Mathematical Software 31 (1) (2005) 1–26.
URL http://doi.acm.org/10.1145/1055531.1055532

[17] M. Erwig, S. Kollmansberger, Functional Pearls: Probabilistic functional programming in Haskell, Journal of Functional Programming 16 (01) (2005) 21–34.

[18] O. Kiselyov, C. Shan, Embedded probabilistic programming, in: Working conf. on domain specific lang, Springer, 2009.

[19] S. Park, F. Pfenning, S. Thrun, A probabilistic language based upon sampling functions, in: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM New York, NY, USA, 2005, pp. 171–182.

[20] K. Murphy, pmtk3: Probabilistic modeling toolkit for matlab/octave, version 3, http://code.google.com/p/pmtk3/.

[21] R. W. Vuduc, Autotuning, in: D. Padua (Ed.), Encyclopedia of Parallel Computing, Springer, 2010.