in **Innovations in Machine Learning: Theory and Applications,**

Springer-Verlag book, L.C. Jain and D. Holmes, Eds., 2005.

Learning Based Programming

Dan Roth
Dept. of Computer Science
University of Illinois
Urbana, IL 61801, USA
danr@cs.uiuc.edu

Abstract

A significant amount of the software written today, and more so in the future, interacts with naturally occurring data — text, speech, images and video, streams of financial data, biological sequences — and needs to reason with respect to concepts that are complex and often cannot be written explicitly in terms of the raw data observed. Developing these systems requires software that is centered around a semantic level interaction model, made possible via trainable components that support abstractions over real world observations.

Today's programming paradigms and the corresponding programming languages, though, are not conducive for that goal. Conventional programming languages rely on a programmer to explicitly define all the concepts and relations involved. On the other hand, in order to write programs that deal with naturally-occurring data, that is highly variable and ambiguous at the measurement level, one needs to develop a new programming model, in which some of the variables, concepts and relations may not be known at programming time, may be defined only in a data driven way, or may not be unambiguously defined without relying on other concepts acquired this way.

In Learning Based Programming (LBP), we propose a programming model that supports interaction with domain elements at a semantic level. LBP addresses key issues that will facilitate the development of systems that interact with real-world data at that level by (1) allowing the programmer to *name* abstractions over domain elements and information sources – defined implicitly in observed data, (2) allowing a programmer to interact with named abstractions (3) supporting seamless incorporation of trainable components into the program, (4) providing a level of inference over trainable components to support combining sources and decisions in ways that respect domain's or application's constraints, and (5) a compilation process that turns a data-dependent high level program into an explicit program, once data is observed.

This chapter describes preliminary work towards the design of such a language, presents some of the theoretical foundations for it and outlines a first generation implementation of a Learning based Programming language, along with some examples.

1 Introduction

> ...these machines are intended to carry out any operations which could be done by a human computer ... supposed to be following fixed rules. ... We may suppose that these rules are supplied in a book which is altered whenever he is put on to a new job....
>
> A. M. Turing, Computing Machinery and Intelligence, 1950.

The fundamental question that led Turing and others to the development of the theory of computation and of conventional programming languages is the attempt to model and understand the limitation of mechanical computation. Programming languages were then developed to simplify the process of setting explicit rules of computation that digital machines can follow. In the last two decades we have seen the emergence of the field of Machine Learning that studies how concepts can be defined and represented without the need for explicit programming, but rather by being presented with previous experiences. Quoting Leslie Valiant, the theory of machine learning attempts to study models of "what can be learned just as computability theory does on what can be computed".

The goal of Learning Based Programming is to develop the programming paradigm and language that can support machine learning centered systems. We view this as a necessary step in facilitating the development of computer programs that interact with and make inferences with respect to naturally occurring data.

Consider writing a computer program that controls an automatic assistant for analyzing a surveillance tape; the programmer wishes to reason with respect to concepts such as *indoor and outdoor scenes*, the identification of *humans* in the image, *gender* and *style of clothing*, recognizing *known people* (perhaps based on typical gestures or movements) etc. Each of these concepts depend on a large number of hierarchical decisions with respect to the observed input. These might include recognizing the concepts mentioned above, tracking a moving body, associating video with audio, etc. These, in turn, depend in intricate ways on even lower level decisions such as identifying trees or other objects, that are also very involved in terms of the raw input presented. Given that no two inputs observed by this program are ever the *same*, it seems impossible to explicitly write down the dependencies among all factors that may potentially contribute to a decision, or a Turing-like list of definitions and rules the program can follow in its processing. Consequently, a lot of research in the last few years focuses on developing machine learning methods that use observed data to *learn* functions that can reliably recognize these lower level concepts.

Today, developing systems like the one mentioned above resembles programming in *machine language*, as opposed to using an advanced programming language. There is no programming paradigm that supports embedded trainable modules and that allows the development of programs that interact with real worlds data and can abstract away most of the lower level details – including defining features and training classifiers to represent low level predicates – and reason, perhaps probabilistically, at the right level of abstraction. Today, in fact, writing a program for each of the low level tasks mentioned above is a respectable Ph.D. thesis.

Other forms of intelligent access to information can serve as examples for the type of processes future programs will have to support. Consider a program that interacts with free form natural langauge questions presented by users attempting to extract information from some news source, the web or even a knowledge base. E.g., "at what elevation did the Columbia explosion occur?" or, more mundane questions like "Is there public transportation from Campus to Urbana on Sundays?" Naturally, interpreting a natural language question to the extent that it can yield a reliable mapping into a knowledge base query or support extracting an answer from free form documents is a challenging task that requires resolving context-sensitive ambiguities at several levels. A lot of research is done on developing machine learning based approaches that deal with specific context sensitive

ambiguities, ranging from predicting the part-of-speech tag of a word in the context of a given sentence, resolving the sense of a word in a given context, resolving prepositional phrase attachments, inducing a syntactic and semantic parse of a sentence, handling a query presented in the context of a previous one, tolerating user's input errors (e.g. typing `our` instead of `out`, `now` instead of `know`), etc. Each of these specific tasks is a difficult undertaking that qualifies for a Ph.D. thesis. However, there is no programming paradigm that allows people to reason at the desired task level, define and learn the appropriate concepts, chain experience based decisions, and reason with them.

While the development of machine learning libraries is an important stage towards facilitating broader use of software systems that have learning components, algorithmic libraries are only one component in Learning Based Programming. LBP focuses on providing a programmer a way to interact with real world data at the appropriate level of abstraction and reason with induced properties of real world domain elements. Starting from *naming* the key primitive elements that can be *sensed* in a domain, to supporting hierarchical naming via learning operators to enabling reasoning with respect to domain elements and their properties by defining constraints among variables in the domain.

2 Learning Based Programming

Learning Based Programming (LBP) is a programming paradigm that extends conventional programming languages by allowing a programmer to write programs in which some of the variables are not explicitly defined in the program. Instead, these variables can be *named* by the programmer that may use language constructs to define them as trainable components. That is, these variables are defined, possibly recursively, as outcomes of trainable components operating on data sources supplied to the program.

Conventional programming languages allow the design and implementation of large scale software systems and rely on a programmer to explicitly define all the concepts and relations involved, often hierarchically. In LBP we develop a programming model that supports building large scale systems in which some components cannot be explicitly defined by a programmer. Realizing that some of the variables, concepts and relations may not have an explicit representation in terms of the observed data, or may not be defined unambiguously without relying on other higher level concepts, LBP provides mechanisms that allow seamless programming using data-defined concepts.

Learning Based Programming allows the programming of complex systems whose behaviors depend on naturally occurring data and that require reasoning about data and concepts in ways that are hard, if not impossible, to write explicitly. The programmer can reason using high level concepts without the need to explicitly define all the variables they might depend on, or the functional dependencies among them. It supports reasoning in terms of the information sources that might contribute to decisions; exactly what variables are extracted from these information sources and how to combine them to define other variables is determined in a data-driven way, via a learning operator the operation of which is abstracted away from the programmer. The key abstraction is that of *naming*, which allows for re-representation of domain elements in terms other concepts, low level *sensors* or higher level concepts that are functions of them. Naming is supported in LBP by allowing programming in terms of data-defined, trainable components. This makes an LBP program, formally, a family of programs; yet, an explicit program is eventually generated by a compilation process that takes as additional input the data observed by the program.

The rest of this article describes the technical issues involved in developing this programming paradigm. The technical description focuses on the knowledge representations underlying LBP, since this is a key issue in developing the language.

3 The LBP Programming Model

A computer program can be viewed as a system of definitions. Programming languages are designed to express definitions of variables in terms of other variables, and allow the manipulations of them so that they represent some meaningful concepts, relations or actions in a domain. In traditional programming systems the programmer is responsible for setting up these definitions. Some may be very complicated and may require the definition of a hierarchy of functions. In LBP, a *naming* process allows the declaration of variables in terms of other variables without setting up the explicit functional dependencies; explicit dependencies are being set up only at a later stage, via interactions with data sources, using a learning process that is abstracted away from the programmer. LBP provides the formalism and an implementation that allows a programmer to set up definitions only by declaring the target concepts (the variables being defined) and the information sources a target concept might depend on, without specifying exactly how. Variables defined this way might have interacting definitions (in the sense that x is a data-defined variable that depends on y, and y is a data-defined variable that depend on x), and there might be some constraints on simultaneous values they can take (e.g., Boolean variables x and y cannot take the same value).

Associated with data sources supplied to an LBP program is a well defined notion of a *sensor*. A sensor is a functional (defined explicitly below) that provides the abstraction required for an LBP program to interact with data. A given domain element comes with a set of sensors that encode the information a program may know about this element. That is, sensors "understand" the raw representation of the data source at some basic level, and provide a mechanism to define and represent more abstract concepts with respect to these elements. For example, for programs that deal with textual input, there might exist basic predicates that *sense* a "word", the relation "before" between words (representing that the word u is before word w in the sentence), and a "punctuation mark", in the input. Other sensors might also be supplied, that provide the program with more knowledge, such as "part-of-speech tag of a word", or a sensor that identifies upper case letters. Primitive sensors might be provided with the data, or programmed to reflect an understanding of the input data format. All other variables will be defined in terms of the output of these sensors, some explicitly and some just *named*, and defined as trainable variables.

Unlike conventional programming models, the notion of an *interaction* is thus central in LBP. An LBP program is data driven. That is, some of the variables participating in the declaration of a new variable, as well as the exact ways new variables depend on other variables, are determined during a data-driven compilation process, which is implemented via machine learning programs and inference algorithms. An LBP program is thus a set of possible programs. The specific element in this set of possible programs that will perform the computation is determined either in a *data driven compilation* stage or at run-time. The notion of data-driven compilation is fundamental to LBP, which has, conceptually, two compilation stages. The first is a machine learning based compilation in which the program interacts with data sources. In this stage, the LBP code, which may contain data-defined variables, is converted into a "conventional" program – although many of the predicates may have complex definitions the programmer may not fully understand (and need not look at). This is done by "running" the machine learning algorithm and inference algorithms on the LBP code and the data pointed to from it. The second compilation stage is the conventional stage of converting high level code to machine code.

Specifically, an LBP program extends traditional programming in the following ways:

- It allows to define *types* of variables, that may give rise to (potentially infinitely) many variables, the exact name of which is determined in a data-driven way.
- Variables (and types of variables) are *named* as abstractions over domain elements; they are defined as the outcome of trainable components and their explicit repre-

sentation is learned in terms of the sensory input.

- It supports a well defined notion of *interaction* with data sources, via sensor mechanism that abstracts the raw information available in the data source.
- It allows the definition of *sets* of data-defined and possibly mutually constrained variables; these take values that are optimized globally via inference procedures.
- Data-driven compilation converts an LBP program into a conventional program by first defining the optimal set of learning tasks required, and a training stage that results in the generation of an explicit program.

Although LBP is formalized in the traditional manner, in the sense that it is given a syntax and operational semantics, we focus here on describing an abstract view of some of the constructs, without providing the syntax of our implementation, so that the general view of the language and programming with it is clear.

3.1 Knowledge Representations for LBP

The key abstraction in LBP is that of a *Relational Generation Function (RGF)*. This is a notion that allows LBP to treat both explicitly defined variables and (types of) variables that are data-defined via learning algorithms, as basic, interchangeable building blocks. In order to define RGFs we need to introduce two of the basic representational constructs in LBP – *relational variables* and a *structural instance space*. An LBP program can be viewed as a program for manipulating, augmenting and performing inferences with respect to elements in the structural instance space. The augmentation can be explicit (changing the instance space) or implicit, and amounts to re-representation of (properties of) domain elements as functions of other (properties of) domain elements, typically done via learning operators incorporated into the program. From the programmer's perspective, though, handling the knowledge representation is done in the same way regardless of whether elements are readily available in the input (via sensors; see below), explicitly defined, or named via learning or inference operators.

A structured instance is a multi-labeled graph based representation of data; nodes are place holders for individuals in the domain; edges represent relations among individuals; and labels (possibly multi labels) on nodes and edges represent properties of these. Raw data elements the program interacts with are viewed by the program via its associated *sensors* (to be defined below), that reveal the information available in the input. The knowledge representation described below, which underlies LBP, draws on our previous work for knowledge representations that support learning [4, 5, 6, 11, 19].

For example, given a natural language *sentence* in the input, the readily available information might be the words and their order (via a `word` sensor and a `before(a,b)` sensor). The programmer might want to reason with respect to higher level predicates such as the syntactic category of the word (e.g., noun, verb), the semantic sense of a word, or its role in the sentence (e.g., subject, object). Reasoning at that level constitutes augmentation of the structured instance by adding labels, nodes (e.g., representing noun phrases) and edges to the graph. LBP allows the programmer to define these and reason with the higher level predicates. This *naming* operation can be viewed as re-representing the input in terms of higher level concepts, the definition of which in terms of the readily available sensors cannot be explicitly given by the programmer[1]. Similarly, a *document* in the input might be represented as a list of words (same sensors as above), but the programmer might want to reason about it also as a list of *sentences*, or as a structured document with a *title* and

[1] Clearly, an explicit definition of the syntactic role of a word in the sentence cannot be given and is likely to depend on the value of the syntactic role of neighboring words; the learning operator will be used to abstract away the explicit definition, acquired in the data-driven compilation process.

paragraphs[2].

We define below a collection of "relational" variables, termed so to emphasize that they can actually stand for relational (quantified) entities. The relational variables are all formulae in a relational language $\mathcal{R}$ that we now define[3]. We then describe how a programmer can initiate a data driven generation of these variables, rather than explicitly define all of them, how they can be manipulated, and how they can be defined in terms of others. For most of the discussion we treat relational variables as Boolean variables, taking only values in $\{0, 1\}$. It will be clear that these variables can also take real-values (in the range $[0, 1]$).

The data sources the program interacts with constitute the *domain* of the relational language $\mathcal{R}$. A domain consists of elements and relations among these elements. Predicates in the domain either describe the relation between an element and its attributes, or the relation between two elements.

Definition 1 (Domain) *A domain $\mathcal{D} = \langle \mathcal{V}, \mathcal{E} \rangle$ consists of a set of typed elements, $\mathcal{V}$, and a set of binary relations $\mathcal{E}$ between elements in $\mathcal{V}$. An element is associated with some attributes and two elements may have multiple relations between them. When two elements have different sets of attributes, we say that the two elements belong to different types.*

Definition 2 (Instance) *An instance is an interpretation [13] which lists a set of domain elements and the truth values of all instantiations of the predicates on them.*

Each instance can be mapped to a directed graph. In particular, each node represents an element in the domain, and may have multiple labels, and each link (directed edge) denotes the relation that holds between the two connected elements. Multiple edges may exists between two given nodes.

The relational language $\mathcal{R}$ is a restricted (function free) first order language for representing knowledge with respect to a domain $\mathcal{D}$. The restrictions on $\mathcal{R}$ are applied by limiting the formulae allowed in the language to those that can be evaluated very efficiently on given instances (Def. 2). This is done by (1) defining primitive formulae with a limited scope of quantifiers, and (2) defining general formulae inductively, in terms of primitive formulae, in a restricted way that depends on the relational structures in the domain. The emphasis is on locality with respect to the relational structures that are represented as graphs.

We omit many of the standard FOL definitions and concentrate on the unique characteristics of $\mathcal{R}$ (see, e.g., [13] for details). The vocabulary in $\mathcal{R}$ consists of constants, variables, predicate symbols, quantifiers, and connectives. Constants and predicate symbols vary for different domains. In particular, for each constant in $\mathcal{R}$, there is an assignment of an element in $\mathcal{V}$. For each k-ary predicate in $\mathcal{R}$, there is an assignment of a mapping from $\mathcal{V}^k$ to {0,1} ({true, false}).

Primitive formulae in $\mathcal{R}$ are defined in the standard way, with the restriction that all formulae have only a single predicate in the scope of each variable. Notice that for primitive formulae in $\mathcal{R}$, the *scope* of a quantifier is always the unique predicate that occurs within the atomic formula. We call a variable-free atomic formula a *proposition* and a quantified atomic formula, a *quantified proposition* [11]. Clearly, for any Boolean function $f : \{0, 1\}^n \rightarrow \{0, 1\}$ on n variables, if $F_1, F_2, \ldots F_n$ are primitive formulae in $\mathcal{R}$, then $f(F_1, F_2, \ldots, F_n)$ is also a formula. The informal semantics of the quantifiers and connectives is the same as usual. Relational variables in $\mathcal{R}$ receive their "truth values" in a data

[2]Even segmenting a document into sentences is a non trivial and context sensitive task, as a large number of the "periods" that occur in documents do not represent sentence delimiters. Similarly, identifying the structure of the document, e.g., title, abstract, introduction and other sections is delegated to named variables that are defined in a data driven way.

[3]In recent work we have developed two different, but equivalent ways to represent elements in the language $\mathcal{R}$. We provide here the operational definition, but a well defined syntax and semantics for the language can also be defined via feature description logic [5].

driven way, with respect to an observed instance. Given an instance x, a formula F in $\mathcal{R}$ has a unique truth value, *the value of F on x*, defined inductively using the truth values of the predicates in F, and the connectives' semantics.

3.1.1 Relation Generation Functions

Writing a formula in $\mathcal{R}$ provides one way to define new variables. We view a formula F in $\mathcal{R}$ as a *relation* $F : x \rightarrow \{0, 1\}$, which maps the instance x to its truth value. A formula is *active* in x if it has truth value *true* in this instance. We denote by X the set of all instances, the *instance space*. A formula $F \in \mathcal{R}$ is thus a relation over X, and we call it a *relational variable*. From this point we use formula and relation interchangeably.

Example 1 *Consider an instance x represented simply as a collection of nodes, that is, as an unordered collection of place holders with two attributes, "word" and "tag" each. Let the list of words be: he, ball, the, kick, would. Then some active relations on this instance are word(he), word(ball), and tag(DET). In this case we could say that the variable* `word(he)` *has value* 1 *in the instance. If* `object` *was another attribute of nodes in the domain, or if there was a way to compute this predicate given the input instance,* `object(z)` *would be another variable. It represents an unbound formula that would evaluate to true if a word exists in the input instance which is an object of a verb in it.*

We now define one of the main constructs in LBP. The notion of *relation generating functions* allows an LBP programmer to define a collection of relational variables without writing them explicitly. This is important, in particular, in the situations for which we envision LBP is most useful; that is, over very large (or infinite) domains or in on-line situations where the domain elements are not known in advance, and it is simply impossible to write down all possible variables one may want to define. However, it may be possible for the programmer to define all "types" of variables that might be of interest, which is exactly the aim of formalizing the notion of an RGF. As we will see, RGFs will allow a unified treatment of programmer defined (types) of variables and data-defined (types) of variables, which is a key design goal of LBP.

Definition 3 (Relation Generation Function) *Let $\mathcal{X}$ be an enumerable collection of formulae over the instance space X. A relation generation function (RGF) is a mapping $G : X \rightarrow 2^{\mathcal{X}}$ that maps an instance $x \in X$ to a set of all elements in $\mathcal{X}$ that satisfy $\chi(x) = 1$. If there is no $\chi \in \mathcal{X}$ for which $\chi(x) = 1$, $G(x) = \phi$.*

RGFs provide a way to define "types" of relational variables (formulae), or to parameterize over a large space of variables. A concrete variable (or a collection thereof) is generated only when an instance x is presented. Next we describe more ways to do that in LBP.

The family of relation generation functions for $\mathcal{R}$ are RGFs whose output are variables (formulae) in $\mathcal{R}$. Those are defined inductively, via a *relational calculus*, just like the definition of the language $\mathcal{R}$. The relational calculus is a calculus of symbols that allows one to inductively compose relation generation functions. Although we do not specify the syntax of the calculus here, it should be clear that this is the way an LBP programmer will introduce his/her knowledge of the problem, by defining types of formulae they think are significant. The alphabet for this calculus consists of (i) basic RGFs, called *sensors* and (ii) a set of connectives. While the connectives are part of the language and are the same for every alphabet, the *sensors* vary from domain to domain.

A sensor is the mechanism via which domain information is extracted. Each data source comes with a set of "sensors" that represent what the program can "sense" in it. For example, humans that read a sentence in natural language "see" a lot in it including, perhaps, its syntactic and semantic analysis and even some association to visual imagery. A program might "sense" only the list of words and their order.

A sensor is the LBP construct used to abstract away the way in which information about an instance has been made available to the program. The information might be readily available in the data, might be easily computed from the data, might be a way to access and external knowledge source that may aid in extracting information from an instance, or might even be a previously learned concept.

An LBP programmer programs sensor definitions that encode the knowledge available about the input representation; that is, they parse the input and read from it the type of information that is readily available.

Definition 4 (Sensor) *A* sensor *is a relation generation function that maps an instance x into a set of atomic formulae in $\mathcal{R}$. When evaluated on instance x, a sensor s outputs all atomic formulae in its range which are* active.

Once sensors are defined, the relational calculus is a calculus of symbols that allows one to inductively compose relation generation functions with sensors as the primitive elements. The relational calculus allows the inductive generation of new RGFs by applying connectives and quantifiers over existing RGFs (see [4] for details). Several mechanisms are used in the language to define the operations of RGFs and their design. These include (1) a *conditioning* mechanism, that restricts the range of an RGF to formulae in a given set (or those which satisfy a given property), (2) a *focus* mechanism that restricts the domain of an RGF to a specified part of the instance and (3) a naming mechanism that allows an easy manipulation of RGFs and a simple way to define new RGFs in terms of existing ones. The operation of the RGFs is defined inductively starting with the definitions of the sensors, using these mechanisms and standard definitions of connectives. We provide here the definition of the *focus* mechanism, which specifies a subset of elements in an instance on which an RGF can be applied.

Definition 5 *Let E be a set of elements in a given instance x. An RGF r is* focused *on E if it generates only formulae in its range that are active in x due to elements in E. The focused RGF is denoted by $r[E]$.*

In particular, the focus mechanism is the one which allows the LBP programmer to define segments of the input, which, once re-represented via the RGFs, form the "examples" presented to the learning operator. There are several ways to define a focus set. It can be specified explicitly or described indirectly by using the structure information (i.e., the links) in the instance. For example, when the goal is to define an RGF that describes a property of a single element in the domain, e.g., part-of-speech tag of a word in a text fragment, focus may be defined relative to this single element. When the goal is to define an RGF the describes a property of the whole instance (e.g., distinguish the mutagenicity of a compound), the focus can simply be the whole instance.

The discussion above exemplified that defining RGFs is tightly related to the structure of the domain. We now augment the relational calculus by adding structural operations, which exploit the structural (relational) properties of a domain as expressed by the links. RGFs defined by these structural operations can generate more general formulae that have more interactions between variables but still allow for efficient evaluation and subsumption, due to the graph structure.

3.1.2 Structural Instance Space

We would like to support the ability of a programmer to deal with naturally occurring data and interact with it at several levels of abstractions. Interacting with domain elements at levels that are not readily available in the data and reasoning with respect predicates that are (non trivial) functions of the data level predicates, require that learning operators can induce a representation of the desired predicates in terms of the available ones. The goal of the following discussion is to show how this is facilitated within LBP.

LBP makes it possible to define richer RGFs, as functions of sensors and recursively, as functions of named variables. These, in turned, provide a controlled but enriched representation of data and thus supports the induction of data elements properties via the learning operators. While this is a general mechanism within LBP, each use of it can be viewed in an analogous way to a feature space mapping commonly done in machine learning (e.g., via kernels) [6, 10]

LBP supports these mappings via the abstraction of data elements the system interacts with into a *structural instance space*. The structural instance space is an encoding of domain elements as "sensed" by the system as graphs; thus, RGFs can operate on instances in a unified way and an LBP programmer can deal with the manipulation and augmentation of the instances and with inference over them.

We mentioned that each instance can be mapped into a directed graph. The following definition formalizes it. In particular, each node represents an element in the domain, and may have multiple attributes, and each link (directed edge) denotes the relation that holds between the two connected elements. Multiple edges may exist between two given nodes.

Definition 6 (Structured Instance) *Let x be an instance in the domain $\mathcal{D} = \langle \mathcal{V}, \mathcal{E} \rangle$. The* structured instance *representation of x is a tuple $(\mathcal{V}, \mathcal{E}_1, \mathcal{E}_2, ...\mathcal{E}_k)$ where $\mathcal{V}$ is a set of typed elements, and $\mathcal{E}_i \subseteq \mathcal{E}$ is a set of binary relations of a given type. The graph $G_i = (\mathcal{V}, \mathcal{E}_i)$, is called the ith* structure *of the instance x and is restricted to be an acyclic graph on V.*

Example 2 (NLP) *A structured instance can correspond to a sentence, with $\mathcal{V}$, the set of words in the sentence and $\mathcal{E} = \mathcal{E}_1$ describing the linear structure of the sentence. That is, $(v_i, v_j) \in \mathcal{E}_1$ iff the word v_i occurs immediately before v_j in the sentence.*

Example 3 (VP) *A structure instance can correspond to a gray level representation of an image. $\mathcal{V}$ may be the set of all positions in a 100×100 gray level image and $\mathcal{E} = \mathcal{E}_1$ describes the adjacency relations top-down and left-right in it. That is, $(v_j, v_k) \in \mathcal{E}_1$ iff the pixel v_j is either immediately to the left or immediately above v_k.*

Alternatively, one could consider a subset of $\mathcal{V}$ which corresponds to those nodes in $\mathcal{V}$, those which are deemed interesting (e.g.,[1]) and $\mathcal{E} = \mathcal{E}_2$ would represent specific relations between these that are of interest. It will be clear later that LBP can allow to easily define or induce new attributes that would thus designate a subset of $\mathcal{V}$.

The relational calculus discussed in Sec. 3.1.1 can now be augmented by adding structural operations. These operations exploit the structural properties of the domain as expressed in each graph G_is in order to define RGFs, and thereby generate non-atomic formulae that may have special meaning in the domain. Basically, structural operations allow us to construct RGFs that conjunct existing RGFs evaluated at various nodes of the structural instances. A large number of regular expression like operations can be defined over the graphs. For computational reasons, we recommend, and exemplify below, only operations along chains in the graphs of the structured instance. However, we note that multiple edges types can exist simultaneously in the graph, and therefore, while very significant computationally, we believe that this does not impose significant expressivity limitations in practice.

Example 4 (NLP) *Assume that the structured instance consists of, in addition to the linear structure of the sentence (G_1), a graph G_2 encoding functional relations among the words. RGFs can be written that represent the Subject-Verb relations between words, or the Subject-Verb-Object chain.*

Example 5 (VP) *An RGF can be written that defines an edge relation in an image, building on a sensor s producing active relations for pixels with intensity value above 50. More*

general operations can be defined or induced to indicate whether a collection of nodes form an edge.

We exemplify the mechanism by defining two structural operators that make use of the chain structure in a graph; both are types of "collocation" operators.

Definition 7 (collocation) *Let $s_1, s_2, \ldots s_k$ be RGFs for $\mathcal{R}$, g a chain-structured subgraph in a given domain $\mathcal{D} = (\mathcal{V}, \mathcal{E})$. $colloc_g(s_1, s_2, \ldots s_k)$ is a restricted conjunctive operator that is evaluated on a chain of length k in g. Specifically, let $v_1, v_2, \ldots v_k \in \mathcal{V}$ be a chain in g. The formulae generated by $colloc_g(s_1, s_2, \ldots s_k)$ are those generated by $s_1[v_1]\&s_2[v_2]\& \ldots \&s_k[v_k]$, where*

1. *by $s_j[v_j]$ we mean here the RGF s_j is focused to $\{v_j\}$, and*
2. *the $\&$ operator means that formulae in the output of $(s\&r)$ are active formulae of the form $F \wedge G$, where F is in the range of s and G is in the range of r. This is needed since each RGF in the conjunction may produce more than one formula.*

The labels of links can be chosen to be part of the generated features if the user thinks the information could facilitate learning.

Example 6 *When applied with respect to the graph g which represents the linear structure of a sentence, $colloc_g$ generates formulae that corresponds to n-grams. E.g., given the fragment "Dr John Smith", RGF* colloc(word, word) *extracts the bigrams* `word(Dr)-word(John)` *and* `word(John)-word(Smith)`. *When the labels on the links are shown, the features become* `word(Dr)-before-word(John)` *and* `word(John)-before-word(Smith)`. *If the linguistic structure is given instead, features like* `word(John)-SubjectOf-word(builds)` *may be generated. See [7] for more examples.*

Similarly to $colloc_g$, one can define a *sparse* collocation as follows:

Definition 8 (sparse collocation) *Let $s_1, s_2, \ldots s_k$ be RGFs for $\mathcal{R}$, g a chain structured subgraph in a given domain $\mathcal{D} = (\mathcal{V}, \mathcal{E})$. $scolloc_g(s_1, s_2, \ldots s_k)$ is a restricted conjunctive operator that is evaluated on a chain of length n in g. Specifically, let $v_1, v_2, \ldots v_n \in \mathcal{V}$ be a chain in g. For each subset $v_{i_1}, v_{i_2}, \ldots v_{i_k}$, where $i_j < i_l$ when $j < l$, all the formulae: $s_1[v_{i_1}]\&s_2[v_{i_2}]\& \ldots \&s_k[v_{i_k}]$, are generated.*

Example 7 *Given the fragment "Dr John Smith", the features generated by RGF scolloc(word,word) are* `word(Dr)-word(John)`, `word(Dr)-word(Smith)`, *and* `word(John)-word(Smith)`.

Notice that while primitive formulae in $\mathcal{R}$ have a single predicate in the scope of each quantifier, the structural properties provide a way to go beyond that, but only in a restricted way dictated by the RGF definition and the defined graph structure of the domain elements. Structural operations allow us to define RGFs that constrain formulae evaluated on different objects without incurring the cost usually associated with enlarging the scope of free variables. This is done by enlarging the scope only as required by the structure of the instance.

3.2 Interaction

An LBP program interacts with its environment via data structures that are *structured instance*s along with a set S of *sensors*. A structured instance is an implementation of Def. 6. It is a graph which consists of nodes (place holders) labeled with attributes – a list of *properties* (e.g., `word`, `tag`, `intensity`) that are predicates that hold in this instance. The

structure of the instance is described by labeled edges, where the labels represent relations between the corresponding nodes. Multiple edges may exist between nodes.

In order for a program to process a domain element it needs to have an understanding of the domain and how it is represented. There is a need to "parse" the input's syntactically at some level. This is one aspect of what is represented by sensors.

Along with observations which represent, for example, a sentence or an image, the program has a set of sensors which can operate on the observation and provide the information the program "sees" in the input. Sensor use their understanding of the syntactic representation of the input to extract information directly from the input – word, tag, word order or intensity value – but can also utilize outside knowledge in processing the input; for example, a `vowel` sensor (which outputs an active variable if its focus word starts with a vowel), needs to use some information to determine its output. An IS-A sensor may need to access an external data structure such as wordnet in order to return an active value (e.g., when focused on "desk" it might return "furniture" among the active variables).

An LBP programmer thus assumes that domain elements come with parsing information – a set of sensors – or it needs to code these in order to build on it in further processing of the data. Via the sensor mechanism LBP supports mapping real world data (text, images, biological sequences) into structured instances. From that point, programming proceeds as usual, and may consist of manipulating conventional variables, defining new RGFs as functions of sensors, evaluating them on observations, or defining new RGFs without supplying explicit definition for how they are computed, which we discuss next.

3.3 Learning Operators in LBP

The design of the learning operators is one of the most important components in LBP and we outline some of the basic ideas on that below. We deliberately disregard a large number of related theoretical issues that are research issues in machine learning and learning theory. The goal of LBP is to build on the success of research in these areas and provide a vehicle for developing systems that use it. The design allows for easy ways to incorporate newly developed learning algorithms and even learning protocols into the language. Conceptually, we would like to think of the learning operators in LBP as the multiplication operation in conventional programming languages in that the specific details of the algorithms may differ according to the type of the operands and are typically hidden from the programmer. At the same time, we expect that the availability of the LBP vehicle will motivate the study of new learning algorithms and protocols as well as open up possibilities for experimental studies in learning that are difficult to pursue today.

The knowledge representations discussed before provide a way to generate and evaluate intermediate representations efficiently, given an observation. An RGF maps an observation into a set of variables, and is thus a *set function*. When defined using the operator Ł below, an RGF would be used mostly as a multi-valued function.

Definition 9 *Let X be the instance space, $C = \{c_1, \ldots c_k\}$ a discrete set of labels. A* set-function *$G : X \rightarrow C$ maps an instance $x \in X$ to a set $c \subseteq C$ of labels. G is a multi-valued function if $|c| = 1$.*

In practice, our implementation allows G to return real values and, once normalized, can be viewed as supplying a probability distribution over the set elements. This is sometimes important for the inference capabilities discussed later.

An operator $\mathcal{E}x$ is defined and used to evaluate an RGF (or a collection thereof). Given an RGF(s) and an instance as input, it generates a list of all formulae specified by the RGFs that are active in the instance. This operator can be viewed as converting domain elements, typically, using specifically focused RGFs, into examples that conventional learning algo-

rithm (hidden inside the learning operator) can make use of. $\mathcal{E}x$ has several parameters that we do not describe here, which allow, for example, to associate a weight with the produced formulae. The resulting representation is an extension of the infinite attribute model [2], that has been used in the SNoW learning architecture [3]. This representation does not have an explicit notion of a label. Any of the variables (features) in the example can be treated as a label. If needed, the programmer can define an RGF to focus on part of the observation and use it as a label or can generate an RGF specifically to label an observation using external information.

The goal of the knowledge representation and the formalism discussed so far is to enable the next step, namely the process of *learning* representations of RGFs from observations. The Ł operator learns a definition of a multi-valued RGF in terms of other variables. It receives as input a collection $r_1, \ldots r_k$ of RGFs, along with a collection of structured instances and a set of values (names of variables), $T = \{t_1, \ldots, t_k\}$, that determines its range. Elements in the set T are called the target variables. Notice that a new RGF is defined to be a function of the *variables* produced by RGFs in its domain, but the variables themselves need not be specified; only "types" of variables are specified, by listing the RGFs.

$$T \doteq \text{Ł}(T, r_1, r_2, \ldots r_k, instances).$$

This expression defines a new RGF that has the variables' names in T as its possible values. Which of these values will be active when T is evaluated on a future instance is not specified directly by the programmer, and will be learned by the operator Ł.

Ł is implemented via a multi-class learner (e.g., [**?**, 9]). When computing Ł, the RGFs $r_1, \ldots r_k$ are first evaluated on each instance to generate an example; the example is then used be Ł to update its representation for all the elements in T as a function of the variables that are currently active in the example. The specific implementation of the operator Ł is not important at this level. The key functional requirement that any implementation of Ł needs to satisfy is that it uses the infinite attribute model (otherwise, it needs to know ahead of time the number of variables that may occur in the input.) Thus, examples are represented simply as lists of variables and the operator Ł treats each of the targets as an autonomous learning problem (an inference operator, below, will handle cases of *sets* of variables that receive values together.) It is the responsibility of the programmer to make sure that the learning problem set up by defining T as above is well defined. Once so defined, T is a well defined RGF that can be evaluated by the operator $\mathcal{E}x$. Moreover, each of the variables in the target set T is an RGF in itself, only that it has only a single variable in its range.

The operator Ł has several modes which we do not describe here. We note, though, that from the point of view of the operator Ł, learning in LBP is always supervised. However, this need not be the case at the application level. It is quite possible that the supervision is "manufactured" by the programmer based on an application level information (as is common in many natural language applications [8]), or by another algorithm (e.g., an EM style algorithm or a semi-supervised learning algorithm). As mentioned earlier, we purposefully abstract away the learning algorithm itself and prefer to view the Ł operator essentially as the multiplication operation in conventional programming languages in that the specific details of the algorithm used are hidden from the programmer.

3.4 Inference

We have discussed so far the use of LBP to learn single variables. The capabilities provided by LBP at this level are already quite significant in that it significantly simplifies the generation of programs that interact with data and rely on a hierarchy or learning operators.

However, making decisions in real world problems often involve assigning values to *sets* of variables where a complex and expressive structure can influence, or even dictate, what assignments are possible. For example, in the task of labeling part-of-speech tags to the

words of a sentence, the prediction is governed by the constraints like “no three consecutive words are verbs.” Another example exists in scene interpretation tasks where predictions must respect constraints that could arise from the nature of the data or task specific conditions.

The study of how these joint predictions can be optimized globally via inference procedures, and how these may affect the training of the classifiers is an active research area. In LBP, the learning task is decoupled from the task of respecting the mutual constraints. Only after variables of interest are learned they used to produce global output consistent with the structural constraints.

LBP supports two general inference procedure a sequential conditional model (e.g., as in [17, 15]) and a more general optimization procedure implemented via linear programming [18].

LBP is designed so that global inference for a collection of variables is an integral part of the language. The programmer needs to specify the variables involved which will be, in general, induced variables whose exact definition is determined in run time by LBP. The constraints among them can be either observed from the data (as in a sequential process) or specified as rules. In either case, an inference algorithm will be used to make global decision over the learned variables and constraints.

3.5 Compilation

An LBP program is a set of possible programs, where the specific element in the space of all programs is determined by a process of *data driven compilation*. This novel notion of data-driven compilation is fundamental to LBP. It combines optimization ideas from the theory of compilation with machine learning based components – RGFs defined in the program are becoming explicit functions as a result of the interaction of the program with data.

Developing a learning centered systems today requires an elaborate process of chaining classifiers. For example, in the context of natural langauge applications, extracting information from text requires first learning a part of speech (POS) tagger. Then, the available data is annotated using the learned POS tagger, and that data is used to (generate features and) learn how to decompose sentences into phrases. The new classifiers are then used to annotate data that is, in turn, used to learn tags required for the information extraction task. This sequential pipelining process has some conceptual problems, that have to do with error accumulation and avoiding inherent interactions across layers of computations. These can be dealt with algorithmically, and the inference capabilities provided by LBP will allow programmers to develop approaches to this problem.

Here, however, we would like to address the software engineering issue raised by this sequential process. Observing the computation tree required to extract the specific phrases of interest, for example, may reveal that most of the computation involved in this process is superfluous. Only a small number of the evaluations done are actually needed to support the final decisions. In traditional programming languages, compilation methods have been developed that can optimize the program and generate only the code that is actually required. LBP provides a framework within which this can be done also for learning intensive programs. We intend to develop optimization methods in the context of learning based programming that will perform analogous tasks and determine what needs to be learned and evaluated in the course of the data driven compilation.

4 Related Work

There are several research efforts that are related to the LBP research program. Perhaps the most related program, at least superficially, is "Yet Another Learning Environment" (YALE) developed at the University of Dortmund (http://yale.cs.uni-dortmund.de). Yale provides a unified framework for diverse machine learning applications. However, YALE is very different from LBP, both conceptually and functionally. It does not attempt to deal with raw data but rather with preprocessed data, it does not have feature extraction and example extraction mechanisms, and it does not deal with relational features as in LBP. It does not plan to support inference and does not have any view on compilation, and clearly not on data-driven compilation. Essentially, none of the issues presented in Sec. 3 are handled by YALE, which is a well designed package for integrative use of machine learning libraries, but not a programming paradigm. Other extensive machine learning libraries, like MLC++, are different for the same reasons. There are some other programs that have some conceptual similarity to LBP, but they typically represent more restricted efforts at developing programming languages for specific applications, such as constructing robot control software [20, 12] or stochastic programs [14, 16].

5 Discussion

The step represented by the Learning Based Programming paradigm is essential in order to develop large scale systems that acquire the bulk of their knowledge from raw, real world data and behave robustly when presented with new previously unseen situations.

This line of research can be viewed both as an end point of a line of research in machine learning – building on the maturity level the machine learning community has reached in understanding and in the design of classifiers - and as a beginning of a promising new programming discipline. At the same time, we expect that the availability of the LBP vehicle will motivate the study of new learning protocols as well as open up possibilities for experimental studies in intelligent interaction that are difficult to pursue today.

While the difficulties of developing machine learning centered systems are apparent to most researchers, there isn't yet a real research effort to develop an appropriate programming paradigm. In the preliminary development of LBP we attempt also to place this direction on the research agenda, given that developing a mature programming language is a huge effort that may require a community wide participation.

We believe that more research into fundamental issues from learning, compilation and software engineering perspectives, as well as a more mature implementation of LBP and several test cases of using it are required before we can determine the success of this approach as well as some of the key directions to follow up on.

6 Acknowledgments

This research is supported by NSF grants ITR-IIS-0085836, ITR-IIS-0085980 and IIS-9984168 and an ONR MURI Award.

References

[1] S. Agarwal, A Awan, and D. Roth. Learning to detect objects in images via a sparse, part-based representation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(11):1475–1490, 2004.

[2] A. Blum. Learning boolean functions in an infinite attribute space. *Machine Learning*, 9(4):373–386, 1992.

[3] A. Carlson, C. Cumby, J. Rosen, and D. Roth. The SNoW learning architecture. Technical Report UIUCDCS-R-99-2101, UIUC Computer Science Department, May 1999.

[4] C. Cumby and D. Roth. Relational representations that facilitate learning. In *Proc. of the International Conference on the Principles of Knowledge Representation and Reasoning*, pages 425–434, 2000.

[5] C. Cumby and D. Roth. Learning with feature description logics. In *Proc. of the International Conference Inductive Logic Programming*, 2002.

[6] C. Cumby and D. Roth. Feature extraction languages for propositionalized relational learning. In *IJCAI Workshop on Learning Statistical Models from Relational Data*, 2003.

[7] Y. Even-Zohar and D. Roth. A classification approach to word prediction. In *Proc. of the Annual Meeting of the North American Association of Computational Linguistics (NAACL)*, pages 124–131, 2000.

[8] A. R. Golding and D. Roth. A Winnow based approach to context-sensitive spelling correction. *Machine Learning*, 34(1-3):107–130, 1999.

[9] S. Har-Peled, D. Roth, and D. Zimak. Constraint classification for multiclass classification and ranking. In *The Conference on Advances in Neural Information Processing Systems (NIPS)*. MIT Press, 2002.

[10] R. Khardon, D. Roth, and R. Servedio. Efficiency versus convergence of boolean kernels for on-line learning algorithms. In *The Conference on Advances in Neural Information Processing Systems (NIPS)*. MIT Press, 2001.

[11] R. Khardon, D. Roth, and L. G. Valiant. Relational learning for NLP using linear threshold elements. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 911–917, 1999.

[12] H. J. Levesque, A. BB, C. DD, D. EE, and E. FF. Golog: a login programming language for dynamic domains. *Journal of Logic Programming*, 31:59–84, 1997.

[13] J. W. Lloyd. *Foundations of Logic Progamming*. Springer-Verlag, 1987.

[14] D. McAllester, D. Koller, and A. Pfeffer. Effective bayesian inference for stochastic programs. In *Proceedings of the National Conference on Artificial Intelligence*, 1997.

[15] A. McCallum, D. Freitag, and F. Pereira. Maximum entropy Markov models for information extraction and segmentation. In *Proceedings of ICML-00*, Stanford, CA, 2000.

[16] A. Pfeffer. Ibal: An integrated bayesian agent language. In *Proceedings of the International Joint Conference of Artificial Intelligence*, 2001.

[17] V. Punyakanok and D. Roth. The use of classifiers in sequential inference. In *The Conference on Advances in Neural Information Processing Systems (NIPS)*, pages 995–1001. MIT Press, 2001.

[18] V. Punyakanok, D. Roth, W. Yih, and D. Zimak. Semantic role labeling via integer linear programming inference. In *Proc. the International Conference on Computational Linguistics (COLING)*, Geneva, Switzerland, August 2004.

[19] D. Roth and W. Yih. Relational learning via propositional algorithms: An information extraction case study. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1257–1263, 2001.

[20] S. Thrun. A framework for programming embedded systems: initial design and results. Technical Report CMU-CS-98-142, CMU Computer Science Department, 1998.