A functional perspective on machine learning via programmable induction and abduction

Steven Cheung[1], Victor Darvariu[1], Dan R. Ghica[1], Koko Muroya[1,3], and Reuben N. S. Rowe[2]

[1] University of Birmingham
[2] University of Kent
[3] RIMS, Kyoto University

Abstract. We present a programming language for machine learning based on the concepts of 'induction' and 'abduction' as encountered in Peirce's logic of science. We consider the desirable features such a language must have, and we identify the 'abductive decoupling' of parameters as a key general enabler of these features. Both an idealised abductive calculus and its implementation as a PPX extension of OCaml are presented, along with several simple examples.

1 A principled functional language for machine learning

What is the right programming language for machine learning? This question can be answered in two ways. A first possible answer could take an algorithmic point of view and try to identify those constructs which are most used in machine learning programs, delivering an implementation in which the balance of various optimisation trade-offs favours such constructs. This way of answering the question has been studied quite extensively (e.g. [1, 2]). A different methodological approach to this question is to first put machine learning in a logical perspective, to provide a guideline in the development of a programming language. Connecting deductive systems to computation is a preferred methodology of functional programming language design [3]. This is the methodology we follow in this paper, except we will consider inference systems beyond deduction.

The first step is, therefore, to place machine learning in a logical framework. This is a delicate, somewhat philosophical, undertaking which may be imperfect or incomplete in its representation of the extremely broad spectrum of machine-learning algorithms. However, this step must be made in order to enable the methodological machinery to crank on. We will situate our logical understanding of machine learning within C.S. Peirce's view of deduction, induction, and abduction as the key reasoning processes in the logic of science. This view, espoused in his celebrated paper *Illustrations of the Logic of Science* is perhaps not the only way in which the logic of machine learning, seen as a computational subsidiary to the logic of science, can be understood. In fact Peirce himself revised both his position on the role of induction and abduction, and even the terminology itself. But the clarity and elegance of his *Illustrations*, formal and conceptual, makes it a compelling organising principle.

The second step is to suggest an informal realisability-style correspondence between the logical and the computational, resulting in a programming language for machine learning in which typical algorithms can be expressed concisely and elegantly[4]. This is indeed a common situation when developing functional programming paradigms. The methodological principles invoked above are incorporated into a calculus, which is then implemented as an extension of the OCaml programming language.

Contributions: We give a methodological justification for abductive inference as a logical framework for machine learning. Following an informal realisability argument for abduction we define a functional programming language for machine learning, which we implement as a PPX extension of OCaml. The language relies on a formal calculus of abduction which is studied elsewhere [4].

2 Deduction, induction, abduction

The division of all inference into Abduction, Deduction, and Induction may almost be said to be the Key of Logic. C.S. Peirce

We faithfully follow Peirce's logical analysis of scientific methodology as given in his *Illustrations of the Logic of Science*. Several clarifications of terminology first. The first one is that the term 'logic' must not be confused with 'deductive logic'. We are employing it in its broader sense, that of any system of formal rules employed in carrying out scientific enquiry. Similarly, the term 'induction' must not be confused with 'mathematical' or other kinds of *deductive* induction, but with the Humean principle of '*generalising from examples*' [5]. Finally, the term 'abduction' is not used in *loc. cit.*, but the original term 'hypothesis' was subsequently replaced by the former, which became more popular.

Logical inference rules fall into two broad categories. Some rules, when correctly applied, result in conclusions which are at least as believable as the assumptions. They are 'apodeictic', i.e. beyond debate. These are the 'deductive' rules, the application of general principles to specific cases. Systems of deductive rules relate elegantly to functional programming via correspondences such as Kleene's proofs-as-programs (realisability) [6] or the propositions-as-types correspondence introduced by Curry and Howard [7]. Computation carried out in such deductive functional programming languages produces definitive results. However, these 'analytic' rules, and the computation inspired by them, play no role in the creation of new knowledge.

In contrast, machine learning is a style of computation characterised by the opposite features. It is tentative, in that it produces possibly imprecise or inaccurate results. Yet it is 'ampliative' (or 'synthetic') in that it generates new knowledge. The tentative nature of the results is a necessary consequence of

[4] The Curry-Howard correspondence emphasises types, whereas realisability emphasises proofs. Because we discuss new proof rules, rather than new types, we will prefer the realisability approach.

knowledge generation, which involves heuristics such as generalisation or guessing. The fallibility of machine learning may be unsettling, but it is an allowance we must make for the sake of creativity. The same uncertainty also characterise the synthetic logical rules of scientific discovery, induction and abduction. By formalising them we simply endow existing scientific practice with a computational dimension.

Peirce's presentation of logical concepts is syllogistic. Deduction is the application of a Rule ('*All men are mortal.*') to a Case ('*Socrates is a man.*') in order to produce a Result ('*Socrates is mortal.*'). In contrast, (scientific) induction is the synthesis of a Rule ('*If the ball is struck, it moves.*') out of a Case ('*The ball was struck.*') and a Result ('*The ball moved.*'). Formalised, this rule is deductively uninteresting, $\dfrac{A \wedge B}{A \supset B}$. In scientific practice the rule is slightly different. Induction either generalises from a number of samples $\dfrac{A \wedge B \quad \cdots \quad A \wedge B}{A \supset B}$ or reinforces an existing rule in light of new evidence $\dfrac{A \wedge B \qquad A \supset B}{A \supset B}$. The strength of the evidence can be modelled more precisely either by augmenting the logic with modalities, or quantitatively, by (frequentist) statistical inference or Bayesian belief revision. These lines of inquiry are investigated by a significant literature [8].

Abduction is the third and final arrangement of Rule, Case and Result in a distinct inference rule: given a Rule and a Result we infer the Case. The formalisation is the (deductively unsound) $\dfrac{B \qquad A \supset B}{A}$. Peirce acknowledged abduction as the rule leading to the most uncertain, the most speculative, knowledge, but also as the rule with the potential to lead to the creation of the most interesting new knowledge. In the practice of scientific discovery, abduction is the process by which we try to answer the question 'Why?'. This rule may seem extravagantly unsound yet it plays a crucial role in Peirce's philosophical understanding of the logic of scientific discovery.[5]

More succinctly, the roles of induction and abduction can be explained as follows. Induction is a way to mechanically create models of the world from data about the world, whereas abduction is an examination of given models in order to understand why they work.

[5] Abduction is essential in making statements about reality when we only have access to sense-data such as measurements. For example, the Result might be '*The thermometer reads* 10°' with the Rule '*If the temperature is* 10° *then the thermometer reads* 10°'. From these we can abduce the Case, that '*The temperature is* 10°'. Note that this can never be apodeictic because, for example, the thermometer may be broken. Denying abductive reasoning and demanding the certainty of deduction leads to universal scepticism, e.g. Descartes's '*evil demon*' which may subvert our experience of the world.

2.1 Proofs-as-programs for induction

The induction rule $\dfrac{A \wedge B \quad \cdots \quad A \wedge B}{A \supset B}$ has a natural computational interpretation as the coercion of a list of pairs of type $A \times B$ into a function $A \to B$, realisable by a collection of constants $\mathsf{interp}_{A,B} : (A \times B)\,\mathsf{list} \to A \to B$. It is reasonable to expect the function to be both conservative, agreeing with the arguments when specified, and ampliative, supplying new and sensible values of type B when an unknown argument of type A is provided. This is interpolation.

Interpolation can only be computed for certain data types. In the most general case, if the type A is an order, i.e. it is equipped with a comparison function, then the simplest and most general interpolation method is piecewise constant interpolation. The resulting function f is constant on each interval $a_n \leq a < a_{n+1}$ with $a_n, a_{n+1} \in A$ consecutive known points, i.e. $f(a) = b_n$ for $a_n \leq a < a_{n+1}$. If A is an ordered field then the piecewise-constant interpolation can be more sensible, with the segments centred in the known values, so that for $(a_{n-1} + a_n)/2 \leq a < (a_n + a_{n+1})/2$, $f(a) = b_n$. If A is a multi-dimensional vector field then the partition of space into regions based on distance to the given points in which the function value is constant is the Voronoi tessellation [9].

If both A and B are fields then a variety of interpolation methods are available (trading off computational simplicity for precision) from linear interpolation, which approximates the function as a set of line segments, to polynomial or spline interpolations, which produce smooth functions. These methods also apply when A is a vector field ('multivariate interpolation') and even if the premises are countably many, i.e. the list of points is infinite (*streams*), via Whittaker-Shannon interpolation [10].

The alternative presentation of the induction rule, $\dfrac{A \wedge B \qquad A \supset B}{A \supset B}$ is more subtle because its computational interpretation suggests the need to 'update' a function $A \to B$ to take into account a new pair of points $A \times B$. If we situate ourselves in the realm of approximation, we note that a function $f : A \to B$ can be always converted into a list of points $(A \times B)\,\mathsf{list}$ via *sampling*, thus reducing this rule to the previous. Concretely this would involve a family of constants $\mathsf{samp}_{A,B} : (A \to B) \to (A \times B)\,\mathsf{list}$. A realisability-style interpretation of this rule would be more subtle because when relating lists of points and functions, interpolating then sampling at the same inputs produces the original data points, but interpolating from a set of samples in general does not produce the original function.

Finally, if we are in an approximate setting, then the computational interpretation of induction can go beyond interpolation. Interpolation is always accurate at the interpolation points, but functions can be synthesised in ways which reduce rather than avoid error at the sampling points, such as *regression* [11]. Regression is a more robust way of synthesising functions because it recognises the possibility that the sample points may incorporate noise or errors. Interpolation will *over-fit* the function to the points, a problem avoided by regression. In the next section we will see that regression plays a key role in the interpretation of abduction.

A basic 'inductive' core functional programming language for induction could, in principle, be designed on the basis of an applied lambda calculus with lists and special $\mathsf{samp}_{A,B}$ and $\mathsf{interp}_{A,B}$ constants.

2.2 Proofs-as-programs for abduction

Induction, computationally, may be interpreted as the synthesis of functions out of data using techniques such as interpolation or regression. This is potentially useful in the context of data science, but it is not quite machine learning. We will see how abduction fills this role. The interpretation is rather different than in the case of the induction, because in the abduction rule $\frac{B \quad A \supset B}{A}$ we will not think of A and B as data, but rather we will think of A as parameters (P) and B as a rule ($M \supset N$). This 'higher-order' version of abduction is particularly interesting for us: $\frac{M \supset N \quad P \supset (M \supset N)}{P}$.

The computational interpretation is as follows. Given a parametrised function $f : P \to (M \to N)$ and a non-parametric function $g : M \to N$ we want to find the values of parameter $p : P$ which makes functions $f\,p : M \to N$ and $g : M \to N$ be 'as similar as possible'. One can think of g as an external phenomenon, an experiment, or an oracle, and f as a model. By abduction we need to find the best parameter values for the model so that the model instance $f\,p$ best explains g. The process of 'abducing the best parameters' of a generic model is a general instance of a *machine learning situation*.

Concretely, a programming language would require a family of constants $\mathsf{abd}_{P,A} : A \to (P \to A) \to P$, where A is (usually) a function type. The informal semantics of $\mathsf{abd}\,m\,f$ is the calculation of a parameter $p : P$ so that a defined measure of distance between $f\,p$ and the reference external function m is minimised. This is a generic optimisation problem which can be approached in different ways depending on the types involved.

If P, the type of the parameters, is a discrete data type then combinatorial optimisation algorithms can be used to compute p. The literature on the topic is substantial [12]. If P is a vector space then numerical approximations such as gradient descent can be used. There is an even broader literature in this area [13] going back to Cauchy's pioneering work on numeric solutions to systems of equations. Note that if the model $P \to A$ is a smooth (differentiable) function and if the programming language has reflection [14] then gradient descent can be made very efficient by computing the differential of the function automatically [15], otherwise it can be computed numerically [16].

An abductive programming language, i.e. a simply-typed lambda calculus extended with a family of abduction primitives ($\mathsf{abd}_{P,A}$), would offer the advantage of highly simplifying machine-learning programming by hiding the search or optimisation mechanisms from the programmer. For example, considering an oracle with signature $r : \mathsf{float} \to \mathsf{float}$, a linear-regression model l_c of r would be constructed as follows (in a generic functional syntax):

$$l\,(p_1, p_2)\,x = p_1 \times x + p_2$$

$$(p'_1, p'_2) = \mathsf{abd}_{\mathsf{float}\times\mathsf{float},\mathsf{float}\to\mathsf{float}}\, r\, l$$
$$l_c = l(p'_1, p'_2)$$

The resulting function l_c is the concrete model, obtained from the abstract model l instantiated with abduced parameters (p'_1, p'_2).

From the point of view of this computational interpretation we can see how induction and abduction are subtly different. Both involve the synthesis of new functions, but the mechanisms are distinct. Whereas induction synthesises a function out of data using a fixed, built-in, transparent mechanism, abduction provides the means of adjusting the parameters of a programmer-supplied function to best-fit an oracle. Induction and abduction can interact to create a reference model out of sampled data rather than using the external process directly, which would be inconvenient. The combined induction-abduction function, using a collection of data points $d : (\mathsf{A} \times \mathsf{B})\,\mathsf{list}$ is defined as $\mathsf{indabd}\, d\, f = \mathsf{abd}_{\mathsf{P},\mathsf{A}\to\mathsf{B}}\,(\mathsf{interp}_{\mathsf{A},\mathsf{B}}\, d)\, f$.

There is a similarity between this style of programming and TensorFlow [17], a successful machine-learning library, with some differences. Our 'abduction' corresponds to 'training', but we impose no syntactic distinctions between a function used as an argument to abduction or as applied to an argument. In TensorFlow the programmer needs to explicitly create 'sessions' in which a model ('computation graph') can be either trained or evaluated, separately. Such distinctions are generally unpleasantly low-level.

3 Programming with induction-abduction

The considerations above highlight a programming idiom which from a realisability perspective relates induction and abduction with certain useful programming constructs. We are reassured by the resemblance of the inductive-abductive style of programming with established frameworks such as TensorFlow. We are proposing in fact an idealised version of such frameworks. The question we ask is how can a conventional functional programming language be improved or extended with inductive-abductive constructs.

Induction does not present a challenge. The interp family of constants are simply extrinsic functions of the requisite signature. Abduction is also introduced in the same way, but programming with it turns out to be inconvenient. The real programming language design challenge is wrestling with the bureaucratic burden of parameter management. In this section we will describe at some length our rationale for language design, with the actual language to follow. We iterate through key problems and informally present partial solutions, in order to highlight the challenge faced. The definitive solution, which addresses all these problems will be presented in the next section.

The key requirement, which will drive most of the language design, is the fact that abduction must rely on a fixed and generic optimisation algorithm. A model $P \to A \to B$ must accommodate a generic optimisation algorithm over the space P of parameters and a norm for type B. For numeric optimisations such as gradient descent, the space P of parameters is commonly a vector space. The linear regression example becomes:

$$\begin{aligned}
&l : \mathsf{vec} \to \mathsf{float} \to \mathsf{float}\\
&l\,v\,x = v[0] \times x + v[1]\\
&v' = \mathsf{indabd}\,r\,l\\
&l_c = l\,v'
\end{aligned}$$

Implicit parameters Parametrising models by vectors makes for ugly syntax. Moreover, composite models must be constructed using operations which are manually lifted to manage parameters. Consider a model for confidence bounds which involves two linear functions (a simple weighted regression [18]):

$$\begin{aligned}
&\mathit{bound}\,v\,x = (l\,v[\mathit{0} : \mathit{1}]\,x, l\,v[\mathit{2} : \mathit{3}]\,x)\\
&v' = \mathsf{indabd}\,r\,\mathit{bound}\\
&\mathit{bound}_c = \mathit{bound}\,v'
\end{aligned}$$

where $v[m : n]$ means taking a slice of the vector v from m to n and r some suitable reference data. The model has four parameters, which must be distributed to the two linear bounds. However, there is a problem with this approach. Explicit decomposition of the parameter vector worsens the syntactic overhead. More seriously, mistakes in the slicing of the vector can lead to runtime errors which, since they involve sizes, cannot be prevented by a simple type system. Instead, we would prefer this:

$$\begin{aligned}
&\mathit{bound}\,x = (l\,x, l\,x)\\
&v' = \mathsf{indabd}\,r\,\mathit{bound}\\
&\mathit{bound}_c = \mathit{bound}\,\{v'\}
\end{aligned}$$

The vector-parameter is an implicit parameter. When a concrete model is produced from the instantiation of the abstract model with the abducted parameters, they are explicitly provided. However, managing the implicit parameter in the lifted term formers is more complex than in languages which support such feature syntactically [19].

In terms of implementation, it may seem possible to handle parameters in a monadic style, but we shall see soon why this solution would not be satisfactory when other, more subtle, requirements are taken into account.

Linear parameters Let us now turn to an issue that drives the ultimate design of the language, illustrated by our running examples:

$$\begin{aligned}
&\mathit{bound}_{\mathit{1}}\,x = (\lambda h.(h\,x, h\,x + \mathit{1}))\,l\\
&\mathit{bound}_{\mathit{2}}\,x = (l\,x, l\,x + \mathit{1})
\end{aligned}$$

What is the dimension of the parameter vector for $bound_1$ and $bound_2$? In the case of the former, it is quite obvious that the vector has two components. In the latter, it depends on whether the programmer intended the two occurrences of the function l to be abducted separately or together. We think there is a case for the parameters to be abducted together, so that both functions have two parameters, not only in an attempt to conform to a beta law which is often expected by functional programmers, but also to allow the programmer to keep control of how many parameters are independently adjustable during abduction. In this example, abduction will always lead to boundaries delimited by two lines 1 unit apart, i.e. fixed confidence bounds of a linear regression. In contrast, the weighted regression model can be defined with two separate parameter vectors (each of size two), $v_0 = [1; 0]$ and $v_1 = [1; 0]$:

$$bound\, x = (l\,\{v_0\}\,x, l\,\{v_1\}\,x)$$

In a final streamlining of the syntax, we omit parameter vectors altogether and we only indicate by $\{-\}$ that a constant is to be interpreted as a parameter, leaving the vector of parameters everywhere implicit:

$$\begin{aligned} l\,x &= \{1\} \times x + \{0\} \\ l'\,x &= \{1\} \times x + \{0\} \\ bound\,x &= (l\,x, l'\,x). \end{aligned}$$

The linearity of parameter occurrences will be always observed, so that for example terms $(\lambda x.x + x)\{0\}$ and $\{0\} + \{0\}$ are distinct, because they have one, respectively two, parameters.

In general, the parameters of a model may be contributed by both arguments and free variables, which means that parameters must be discovered not only in the body of the function and its arguments, but also in closures. Linearity and the need for 'deep' search of parameters indicates a simple syntactic solution to be unlikely.

In conclusion, we want our running examples to be, ideally, written as:

$$\begin{aligned} &l\,a\,b\,x = a \times x + b \\ &g\,x = (l\,\{1\}\,\{0\})\,x \\ &f\,x = (l\,\{1\}\,\{0\})\,x \\ &f'\,x = (l\,\{1\}\,\{0\})\,x \\ &bound\_w\,x = (f\,x, f'\,x) \\ &bound\_l\,x = (g\,x, g\,x + 1) \end{aligned}$$

where $bound\_w$ is the four-parameter weighted regression model, and $bound\_l$ the two-parameter unit confidence interval of a linear regression.

4 The abductive calculus

We have argued that the construction of a parametrised model, which is the second input of the abduction $\mathsf{abd}_{P,A} : A \to (P \to A) \to P$, can be tedious or erroneous in the absence of implicit parameter management. We would like parameters to be implicit in construction of a parametrised model, and to be collected as a unique and opaque type for abduction.

The previous section illustrated the construction of models with implicit parameters that are represented by specially annotated constants $\{k\}$. To enable abductive programming we propose *decoupling* as a key feature. This consists of a family of constants $\mathsf{dec}_{V_a,A} : A \to \big((V_a \to A) \times V_a\big)$ where V_a is an abstract type indexed by a unique name a (or 'atom'), as a mechanism to collect implicit parameters and prepare an explicitly parametrised model.

Informally, $\mathsf{dec}\, m$ computes a pair $(l : V_a \to A, p : V_a)$ by collecting all implicit parameters in $m : A$ as the vector p and turning the model m (with implicit parameters) into a parametrised model $l : V_a \to A$, a function on parameter-vectors. The name a is shared by the parameter-vector p and the parametrised model l, making type V_a unique for the model. For our leading example, where π_1 is the first projection,

$$\begin{aligned}
&l\, x = \{1\} \times x + \{0\}\\
&bound\, x = (l\, x, l\, x + 1)\\
&bound_p = \pi_1(\mathsf{dec}\, bound)\\
&p' = \mathsf{indabd}\, r\, bound_p\\
&bound_c = bound_p\, p'
\end{aligned}$$

In this section we give an overview presentation of the *abductive calculus*, an extension of the simply-typed lambda-calculus for abductive decoupling [4]. Let $\mathbb{A}$ be a set of names (or *atoms*). Let $(\mathbb{F}, +, -, \times, /)$ be a (fixed) field and V_a an $\mathbb{A}$-indexed family of opaque vector types. The types T of the calculus are defined by the grammar $T ::= \mathbb{F} \mid V_a \mid T \to T$ where $a \in \mathbb{A}$.

Terms t are defined by the grammar $t ::= x \mid \lambda x^{T'}.t \mid t\,t \mid \underline{k} \mid t \,\$\, t \mid \{\underline{k}\} \mid \mathsf{A}_{a,T'}(f,x).t$, where T and T' are types, f and x are variables, $\$ \in \Sigma$ binary primitive operations, $k \in \mathbb{F}$ field elements, and $a \in A$ names. The novel syntactic elements of the calculus are provisional constants $\{\underline{k}\}$, which serve as implicit parameters in the above discussion, and a family of type- and name-indexed abductive decoupling functions $\mathsf{A}_{a,T'}(f,x).t$. Decoupling functions are the implicational form of the decoupling operation dec; they can be syntactically related by $(\mathsf{A}_{a,T'}(f,x).t)\, u \equiv (\lambda(f,x).t)\, (\mathsf{dec}_{V_a,T'}\, u)$ in the presence of tuples. We opt for the implicational form to make the scope of names explicit, as we see in the type system below.

Let $A \subset_{\mathrm{fin}} \mathbb{A}$ be a finite set of names, Γ a sequence of typed variables $x_i{:}T_i$, and $\boldsymbol{p}$ a sequence of elements of the field $\mathbb{F}$ (i.e. a vector over $\mathbb{F}$). We write $A \vdash \Gamma$ if A is the support of Γ. The typing judgements are of shape: $A \mid \Gamma \mid \boldsymbol{p} \vdash t : T$, and typing derivation rules are given below.

$$\frac{A \vdash \Gamma, T}{A \mid \Gamma, x : T \mid - \vdash x : T} \qquad \frac{A \mid \Gamma, x : T' \mid \boldsymbol{p} \vdash t : T}{A \mid \Gamma \mid \boldsymbol{p} \vdash \lambda x^{T'}.t : T' \to T}$$

$$\frac{A \mid \Gamma \mid \boldsymbol{p} \vdash t : T' \to T \quad A \mid \Gamma \mid \boldsymbol{q} \vdash u : T'}{A \mid \Gamma \mid \boldsymbol{p}, \boldsymbol{q} \vdash t\, u : T}$$

$$\frac{A \vdash \Gamma \quad k \in \mathbb{F}}{A \mid \Gamma \mid - \vdash \underline{k} : \mathbb{F}} \qquad \frac{A \mid \Gamma \mid \boldsymbol{p} \vdash t_1 : T_1 \quad A \mid \Gamma \mid \boldsymbol{q} \vdash t_2 : T_2 \quad \$: T_1 \to T_2 \to T \in \Sigma}{A \mid \Gamma \mid \boldsymbol{p}, \boldsymbol{q} \vdash t_1 \; \$ \; t_2 : T}$$

$$\frac{A \vdash \Gamma}{A \mid \Gamma \mid p \vdash \{\underline{p}\} : \mathbb{F}} \qquad \frac{A, a \mid \Gamma, f : V_a \to T', x : V_a \mid \boldsymbol{p} \vdash t : T \quad A \vdash \Gamma, T', T}{A \mid \Gamma \mid \boldsymbol{p} \vdash \mathsf{A}_{a,T'}(f, x).t : T' \to T}$$

Note that the rules are linear with respect to the parameters $\boldsymbol{p}$. In a derivable judgement $A \mid \Gamma \mid \boldsymbol{p} \vdash t : T$, the vector $\boldsymbol{p}$ gives the collection of all provisional constants from t. The decoupling function $\mathsf{A}_{a,T'}(f, x).t$ binds name a, so it requires in its typing a unique vector type V_a collecting all the provisional constants. The typing rule for the function limits the scope of the name a, so that this vector type V_a cannot be used outside of the scope of the function. As a consequence the vector type V_a is unique to the decoupling function. Variables f and x bound by the function share the type V_a but this type cannot be mixed with parameters produced by other decouplings, as they may result in vectors with different numbers of elements. This is discussed in Sec. 6.

Employing the straightforward extension by tuples (and lists) and the syntactic sugar $\mathsf{let}\ x = u\ \mathsf{in}\ t \equiv (\lambda x.t)\, u$, our leading example can be written in the abductive calculus as below.

$$\begin{aligned}
&\mathsf{let}\ l = \lambda a.\lambda b.\lambda x.a \times x + b\ \mathsf{in}\\
&\mathsf{let}\ f = l\,\{1\}\,\{0\}\ \mathsf{in}\\
&\mathsf{let}\ \mathit{bound} = \lambda x.(f\,x, f\,x + 1)\ \mathsf{in}\\
&\mathsf{let}\ \mathit{update} = \mathsf{A}(\mathit{bound}_p, \_).\mathit{bound}_p\ (\mathsf{indabd}\ r\ \mathit{bound}_p)\ \mathsf{in}\\
&\mathit{update}\ \mathit{bound}
\end{aligned}$$

Its operational semantics is specified using a variation on the Geometry of Interaction [20] which relies on graph rewriting [21]. Using this semantics the calculus is proved as sound (well-typed programs terminate with a value).

The abstract machine represents a term as a graph along the edges of which a token travels. The routing of the token is defined by language-specific rules, as are the rewrite rules. The presence of the token indicates unambiguously what rule must apply, defining in effect a particular reduction strategy.

The 'dynamic rewriting' style of the operational semantics has several advantages which we discuss below.

Sharing of provisional constants in the graph model is naturally represented by making provisional-constant nodes with several incoming edges. In a term model the same can only be achieved by introducing auxiliary names, a more complicated formalism. Moreover, provisional constants cannot be copied or discarded, but only shared at run-time, restrictions which are naturally reflected in a graph model, unlike in a term model.

Decoupling is a complex, dynamic runtime operation which in the graph model is surprisingly easy to formulate. Representing code and environment jointly as a single graph removes the need for complex lookups in the formulation of this rule. This is not as convenient in conventional abstract machines, because code and environment are separate, hindering the formulation of rules involving both, especially the collecting and sharing of provisional constants.

In the presence of provisional constants, a program can be interpreted in both extensional and intensional ways. For example, an extensional interpretation of $\{1\} + \{2\}$ is a value 3, while its intensional interpretation is 'a computation of summation, given two provisional constants that are currently 2 and 3.' The graph-rewriting abstract machine has an ability to handle both interpretations at the same time, by separating the flow of computation (the graph) and the input-output behaviour (the token).

The same 'two-in-one' graph representation also enables a direct proof of program equivalence by means of bisimulation, notably by dealing with the congruence property in terms of sub-graphs.

5 DecML, a functional language for machine learning

Terms in the abductive calculus can be evaluated on-line in an experimental graph-rewriting engine implementing the semantics directly.[6] Additionally, we want to implement a fragment of the abductive calculus as an extension of an existing real-world functional programming language rather than as a totally new language. The major challenge of implementing the abductive calculus is to extract parameters, especially from closures. The semantic definition, which is a global reference-chasing operation similar to garbage collection, is not easily implementable. It seems to require a deep and undesirable intervention on the runtime of the language. We will therefore pursue an alternative strategy, by building a runtime structure for parameter management which is maintained by instrumentation of the native code. This will make it possible to actually maintain the model in a form in which decoupling is a trivial operation.

We implement the abductive calculus as a language extension to OCaml along with the translation from it to standard OCaml under the PPX framework [22, Sec. 7.23].[7] In addition to OCaml terms t_{OCaml} the abductive calculus is extended with new terms $t ::= t_{\text{OCaml}} \mid [\%\mathsf{pc}\ k] \mid [\%\mathsf{lift}\ t]$ where, k are (floating-point) constants. If $t : A$ then $[\%\mathsf{model}\ t] : (dict \to A){*}dict$. The boundary between 'pure' OCaml and the abductive terms is indicated by $[\%\mathsf{model}\ t]$, with abductive code inside the marker.

The tag $[\%\mathsf{model}\ t]$ ensures the code t is evaluated as a term of the abductive calculus, and presents the result to the ambient OCaml code as a model with decoupled parameters. This will require a combination of syntactic transformations and runtime instrumentations. $[\%\mathsf{pc}\ k]$ defines a provisional constant,

[6] http://bit.ly/abd-vis
[7] https://github.com/DecML/decml-ppx

while lifting [%lift k] allows identifiers from outside of the abductive fragment, including most OCaml operators, to be used as (trivial) models.

We define a translation $\lceil - \rceil$ for terms $t : A$ of the extended abductive calculus, into terms t'_{OCaml} of 'lifted' types $(\mathit{dict} \to A) * \mathit{dict}$ of OCaml. The first projection is the model, a function parameterised by a dictionary of parameters, which is also given, as the second projection. The translation accumulates the parameters from its sub-terms by merging their dictionaries, and it 'lifts' the syntactic constructs (abstraction, application, etc.) so that they match the lifted types.

The dictionary is a simple data structure that associates each provisional constant to a unique key. By using dictionaries instead of vectors, merging two sets of parameters is easy since we no longer need to consider their order inside the parametrised function. Below are the required dictionary operations in the target language: *empty* denotes an empty dictionary, *new_key* is an operation that returns a new global key, *create_dict* creates a single element dictionary from a key and a float, *lookup* returns the value that is associated with a key in a dictionary and *merge* combines two compatible dictionaries by joining them.

$$\begin{array}{ll}
\mathit{empty} : \mathit{dict} & \mathit{new\_key} : \mathit{unit} \to \mathit{key} \\
\mathit{new\_dict} : \mathit{key} \to \mathit{float} \to \mathit{dict} & \mathit{lookup} : \mathit{key} \to \mathit{dict} \to \mathit{float} \\
\mathit{merge} : \mathit{dict} \to \mathit{dict} \to \mathit{dict} &
\end{array}$$

Variable x stands for any OCaml identifier, including constants, and k for a float. The translation is indexed by a set V of variables bound in the model:

$$\begin{array}{rll}
\lceil x \rceil_V = & (\mathit{fst}\, x, \mathit{snd}\, x) & (\text{if } x \notin V) \\
\lceil x \rceil_V = & (\lambda\_.x, \mathit{empty}) & (\text{if } x \in V) \\
\lceil \%\mathsf{lift}\ t \rceil_V = & (\lambda\_.t, \mathit{empty}) & (\text{where } t \text{ is a pure OCaml term}) \\
\lceil \%\mathsf{pc}\ k \rceil_V = & (\lambda q.\mathit{lookup}\ L\ q, \mathit{new\_dict}\, L\, k) & (\text{where } L = \mathit{new\_key}\,()) \\
\lceil (t_1, t_2) \rceil_V = & (\lambda q.((F_1\, q), (F_2\, u)), \mathit{merge}\, P_1\, P_2) & (\text{where } (F_i, P_i) = \lceil t_i \rceil_V) \\
\lceil t_1\ t_2 \rceil_V = & (\lambda q.((F_1\, q)\, (F_2\, u)), \mathit{merge}\, P_1\, P_2) & (\text{where } (F_i, P_i) = \lceil t_i \rceil_V) \\
\lceil \lambda x.t \rceil_V = & (\lambda q.\lambda x.F\, q, P) & (\text{where } (F, P) = \lceil t \rceil_{V \cup \{x\}})
\end{array}$$

In the defintion of %lift, by a 'pure' OCaml term we mean a term with no abductive syntax or types. In concrete DecML syntax, the leading example is:

$$\begin{array}{l}
\mathsf{let}\ (*), (+), j, z, i = [\%\mathsf{lift}\,(*.)], [\%\mathsf{lift}\,(+.)], [\%\mathsf{pc}\,1.0], [\%\mathsf{pc}\,0.0], [\%\mathsf{lift}\,1.0]\ \mathsf{in} \\
\mathsf{let}\ l = [\%\mathsf{model}\,\mathsf{fun}\, x \to j * x + z]\ \mathsf{in} \\
\mathsf{let}\ (bp, p) = [\%\mathsf{model}\,\mathsf{fun}\, x \to (l\, x), (l\, x + i)]\ \mathsf{in} \\
\mathsf{let}\ p' = \mathit{indabd}\, r\, bp\ \mathsf{in} \\
\mathsf{let}\ bc = bp\, p'\ \mathsf{in} \ldots
\end{array}$$

Induction-abduction *indabd* is not implemented as a constant, but it can be any function of the right type, implementing a generic optimisation algorithm such as gradient descent. Parameter p can be supplied to this function as an argument, if necessary, to seed the optimisation algorithm with an initial point.

This final observation also explains our decision to use the %model annotation to lift only parts of an OCaml program rather than the whole program. Inside the abductive fragment all operations are lifted to manage parameters, which makes them less efficient and interferes with compiler optimisation. But once a model is created in the decoupled form then it can be processed in the more efficient ambient language. It is particularly important that abduction, which dominates computationally any machine learning program, can be executed natively and efficiently. The mixing of pure and instrumented code, on the other hand, can lead to subtle typing problems which could be difficult for the programmer to understand and fix. The limited access PPX has to typing information makes it a challenge to give further assistance on this matter, so a future version of this language may require substantial re-engineering.

6 Related and further work

Our work has been heavily influenced by TensorFlow [17]. We are aiming to provide an idealised, functional version of this framework. DecML, by using the decoupling mechanism, avoids the need to represent parameters using imperative state, while recognising their importance (‘variables’ in TensorFlow terminology) as a distinct language entity. We also recognise the dual-use of models, in direct and training mode, but we prefer to not make this semantic distinction syntactic. As a shallow embedding of a DSL into Python, TensorFlow must sometimes use rather heavy-going constructs such as that of a ‘session’, which we can afford to completely elide. Moreover, by presenting a language extension rather than an embedded DSL we avoid a host of well-known problems and pitfalls [23–25].

For reasons of space and presentational focus we have also glossed over another significant distinction between inductive-abductive programming and TensorFlow. In the former, abduction is given as a fixed, language-specific, construct whereas in the latter the abduction (search and optimisation) algorithm is programmable. Of course, fixing the abduction algorithm and assuming that certain types come with fixed norms is impractical. Prolog is an example of an abductive programming language in which abduction is implemented as a fixed resolution algorithm, which significantly narrows the applicability of the language to practical problems.

However, if the programming language we are extending is rich enough (such as OCaml), then the induction-abduction extrinsics can be simply programmed as normal library functions. The same applies to programming the norm functions explicitly. In fact a fixed abduction construct is not actually defined in the abductive calculus [4], nor is it in DecML! We will briefly discuss some further language design considerations for programmable abduction, which are already included in the abductive calculus but not yet implemented in DecML.

The key requirement is that parameters are collected as an opaque *vector* type, the key data type required by the formalisation of generic numeric optimisation problems. Since parameter collection and slicing mechanisms are complex, the dimension of abducted parameter vectors and even the order of coordinates are impossible to anticipate at compile-time. Abstracting away these details is therefore required, and any vector needs to be uniquely associated with the model which it parametrises. Making abduction programmable also explains why we prefer to extract, rather than discard, the current values of the parameters. They are often used by programmers to seed the search and optimisation algorithms, using domain-specific knowledge.

In order to prevent erroneous uses, unique and opaque vector types must be generated for each model so that vectors produced by abduction can only be used with the original model. The opaqueness prevents access to the representation, i.e. to the bases or the individual coordinates. Only operators which are symmetric under permutations of bases will be allowed. Mathematically, they correspond to symmetric tensors, but formulated in a programmer-friendly way. This is a real, but not onerous, restriction on the way the generic optimisation algorithms are used. Indeed, if generic optimisation algorithms are to be used at all it is difficult to imagine how (or why) we may want to program them so that different axes are treated differently in the search space. As an extra bonus, linear vector operations are efficiently programmable and easily parallelisable on specialised architectures such as GPUs.

Our proposal focusses on the correspondence between inductive-abductive inference rules and programming constructs in order to extract methodological principles for the design of a machine-learning-oriented programming language. However, these correspondences are only pursued informally. A rigorous realizability definition for induction and abduction is likely to be an interesting and instructive mathematical exercise. We plan to pursue it in the future.

Besides realisability, Curry-Howard-style correspondences can also be pursued by refining the type system to distinguish between the various modalities arising out of inductive and abductive reasoning. The distinction between definite and tentative (or approximate) values can be handled by epistemic logics which distinguish between 'known' and 'believed' statements. This can lead to types for inductive-abductive programming which can track the epistemic status of results of computations. More subtly, the analytic vs. synthetic distinction can also be modelled by type systems [26] which can prove useful in the context of machine learning. This remains a longer-term project.

References

1. Chu, C.T., Kim, S.K., Lin, Y.A., Yu, Y., Bradski, G., Olukotun, K., Ng, A.Y.: Map-reduce for machine learning on multicore. In: Advances in neural information processing systems. (2007) 281–288
2. King, D.E.: Dlib-ml: A machine learning toolkit. Journal of Machine Learning Research **10**(Jul) (2009) 1755–1758

3. Huet, G.: Deduction and computation. In: Fundamentals of Artificial Intelligence. Springer (1986) 38–74
4. Muroya, K., Cheung, S., Ghica, D.R.: Abductive functional programming, a semantic approach. CoRR **abs/1710.03984** (2017) (Submitted for publication).
5. Howson, C.: Hume's problem: Induction and the justification of belief. Clarendon Press (2000)
6. Kleene, S.C.: On the interpretation of intuitionistic number theory. The Journal of Symbolic Logic **10**(4) (1945) 109–124
7. Howard, W.A.: The formulae-as-types notion of construction. To HB Curry: essays on combinatory logic, lambda calculus and formalism **44** (1980) 479–490
8. Sheridan, F.: A survey of techniques for inference under uncertainty. Artificial Intelligence Review **5**(1-2) (1991) 89–119
9. Aurenhammer, F.: Voronoi diagramsa survey of a fundamental geometric data structure. ACM Computing Surveys (CSUR) **23**(3) (1991) 345–405
10. Marks, R.: Introduction to Shannon sampling and interpolation theory. Springer Science and Business Media (2012)
11. Vaughn, B.K.: Data analysis using regression and multilevel/hierarchical models. Journal of Educational Measurement **45**(1) (2008) 94–97
12. Ehrgott, M., Gandibleux, X.: A survey and annotated bibliography of multiobjective combinatorial optimization. OR-Spektrum **22**(4) (2000) 425–460
13. Snyman, J.: Practical mathematical optimization: an introduction to basic optimization theory and classical and new gradient-based algorithms. Volume 97. Springer Science and Business Media (2005)
14. Smith, B.C.: Reflection and semantics in Lisp. In: POPL, ACM (1984) 23–35
15. Rall, L.B.: Automatic differentiation: Techniques and applications. Springer (1981)
16. Lyness, J.N., Moler, C.B.: Numerical differentiation of analytic functions. SIAM Journal on Numerical Analysis **4**(2) (1967) 202–210
17. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., et al.: Tensorflow: Large-scale machine learning on heterogeneous distributed systems. arXiv preprint arXiv:1603.04467 (2016)
18. Cleveland, W.S.: Robust locally weighted regression and smoothing scatterplots. Journal of the American statistical association **74**(368) (1979) 829–836
19. Lewis, J.R., Launchbury, J., Meijer, E., Shields, M.B.: Implicit parameters: Dynamic scoping with static types. In: POPL. (2000) 108–118
20. Girard, J.Y.: Geometry of interaction 1: Interpretation of System F. Studies in Logic and the Foundations of Mathematics **127** (1989) 221–260
21. Muroya, K., Ghica, D.R.: The dynamic Geometry of Interaction machine: A call-by-need graph rewriter. In: Computer Science Logic. (2017) 32:1–32:15
22. Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., Vouillon, J.: The OCaml system release 4.02 (2013)
23. Svenningsson, J., Axelsson, E.: Combining deep and shallow embedding for EDSL. In: Trends in Functional Programming. (2012) 21–36
24. Scherr, M., Chiba, S.: Implicit staging of EDSL expressions: A bridge between shallow and deep embedding. In: ECOOP, Uppsala, Sweden. (2014) 385–410
25. Gibbons, J., Wu, N.: Folding domain-specific languages: deep and shallow embeddings (functional pearl). In: ICFP, Gothenburg, Sweden. (2014) 339–347
26. Martin-Löf, P. In: Analytic and Synthetic Judgements in Type Theory. Springer Netherlands, Dordrecht (1994) 87–99