

Sequence Extrapolation

Philip Laird* and Ronald Saul†

A.I. Research Branch

Recom Technologies, Inc.

NASA Ames Research Center

Moffett Field, CA 94035-1000 (U.S.A.)

Abstract

We study sequence extrapolation as a machine-learning problem. Input examples come from a stream of data elements of a single type (integers, strings, etc.). Stream values are presented in order; the problem is to construct a hypothesis that explains the given values and predicts the subsequent ones. We define a new representation language for streams called *elementary stream descriptions* and present a unified algorithm that learns elementary streams over a generous family of data types that includes integers, lists, and strings. The algorithm has been implemented and an application targeted.

Introduction

Learning from experience to predict sequences of discrete symbols is a fundamental problem in machine learning. The problem can be treated statistically as one of learning a time-dependent probability distribution [Laird, 1992], analytically as one of learning a time series $\{f(t) \mid t > 0\}$, or (in this paper) as one of *sequence extrapolation*, where the task is to form a rule for computing the n 'th value from its predecessors. As a motivating example, try to state a rule that accounts for the following sequence of strings and use it to predict the next one: *, !**?, !!*!***?, !!!*!!*!***???, ... We present an algorithm that extrapolates many sequences (including this one) over a large class of datatypes. Easily understood and implemented, the algorithm improves the assortment of *ad hoc* techniques used heretofore.

In prior work, psychologists have studied in some detail the way humans infer rules for *Thurstone sequences*—sequences of alphabetic letters (e.g., ‘‘A C B D C E...’’) in which the alphabetic ordering provides the fundamental relationship from which the rule is formed. Programs that implement models of human performance on this task have also been used to test cognitive theories (e.g., [Kotovsky and Simon, 1973]).

For nearly thirty years programmers have tried to write powerful sequence extrapolation programs. For example, Pivar and Finkelstein [1964] described a Lisp

program that extrapolates integer sequences based on the so-called *method of differences*. Feenstra (cited in [Dewdney, 1986]) later extended the method with some clever heuristics and was able to achieve an ‘‘IQ’’ of about 160 on a published test. Evidently the method of differences is promising for extrapolating integer sequences but hard to extend to datatypes other than integers.

In a little known but interesting study, Persson [Persson, 1966] constructed several sequence extrapolation programs and sought a method for automating the task of constructing a representation in which the extrapolation problem is easy. His strategy was to look for features common to extrapolation problems over different data types and to view the task, not as solving a series of unrelated problem instances, but as learning to solve the general problem more effectively as more instances are solved.

In other related work of note, Hedrick [Hedrick, 1976] used semantic nets to chart relationships among objects in an effort to find a method applicable to sequences of objects over several datatypes. His work, however, seems not to have been continued. Dietterich, Michalski and others [Dietterich and Michalski, 1986; Michalski *et al.*, 1987] built several programs to play the game Eleusis. In this problem the extrapolation rule is nondeterministic in that the next item in the sequence is not uniquely determined (e.g., the rule ‘‘alternating even and odd integers’’ admits many possible sequences).

In recent years, as machine learning research has concentrated on a small number of core problems, interest in sequence extrapolation has subsided. With this paper we hope to revive interest in the problem by (1) giving an elegant algorithm that solves a significant family of extrapolation problems, and (2) showing how it applies to inductive concept learning.

Preliminary Concepts

To motivate our algorithm, let us note some features of the way humans do sequence extrapolation.

- One usually constructs several hypotheses of about the same complexity that explain the first few values of the sequence. As new values arrive, these hypotheses are modified or replaced as needed by more complex hypotheses.
- A small number of unexplained initial values form

*Email: laird@ptolemy.arc.nasa.gov

†Email: saul@ptolemy.arc.nasa.gov

the seed for computing the remainder of the sequence.

- Whether extrapolating sequences of integers, strings, lists, or whatever, we apply largely the same strategy. Indeed, type-independent extrapolation has been a recurring (but elusive) theme in the literature.
- Probably the most striking characteristic is how few examples are needed before we have a hypothesis in whose ability to predict future values we have high confidence. Whereas statistical models of machine learning suggest that thousands of examples should be required to learn a nontrivial family of functions, three or four values in sequence often suffice to learn complex rules within a very large family. (Consider the motivating example above.)
- One can estimate one's confidence that a given hypothesis will correctly predict the next value, based on how well the hypothesis has performed previously. When the confidence is sufficiently high, one may even question an input value that disagrees with the prediction and put more faith in the hypothesis than in the data.

Example 1 The sequence $\langle 2, 3, 5, 8, \dots \rangle$ over the naturals.

The rule $x_n = x_{n-1} + x_{n-2}$ explains both the 5 and the 8, taking 2 and 3 as initial values, and predicts 13 next. The rule $x_n = x_{n-1} + n$ predicts 3, 5, and 8, taking only 2 as an initial value, and predicts 12 next. Both rules are simple and both predictions, reasonable. A more complicated hypothesis like $x_n = -x_{n-1} + x_{n-2}^2 + 2x_{n-3}$ —while consistent—would not yet be considered.

Example 2 Over the alphabet $\{J, K\}$, the sequence of strings $\langle J, KJK, KJKKJKJ, KJKKJKJKJKKJKJKJ, \dots \rangle$.

Although this appears more difficult than the preceding example, notice that the third string x_3 can be formed by concatenating two occurrences of x_2 to one of x_1 , and this rule correctly predicts x_4 . Note the similarity in the rules: $x_n = x_{n-1} \cdot x_{n-1} \cdot x_{n-2}$ (strings) and $x_n = x_{n-1} + x_{n-1} + x_{n-2}$ (naturals).

Example 3 Over the domain of “dotted” pairs (ordered binary trees), the sequence:

$$\begin{aligned} &((a \cdot b), \\ &((a \cdot a) \cdot ((c \cdot a) \cdot b)), \\ &(((a \cdot a) \cdot (a \cdot a)) \cdot (c \cdot b)), \\ &\dots) \end{aligned}$$

Some aspects of this sequence are immediately predictable—e.g., that each value is $(x_n \cdot y_n)$ whose left part x_n is a pair $(x_{n-1} \cdot x_{n-1})$, and that the right part y_n is of the form $(?? \cdot b)$. But the “??” part corresponds to the sequence $nil, (c \cdot a), c, \dots$, and no predictable pattern is yet emergent. The fourth value in this sequence, therefore, will likely be

$$((((a \cdot a) \cdot (a \cdot a)) \cdot ((a \cdot a) \cdot (a \cdot a))) \cdot (?? \cdot b)),$$

where “??” indicates a part that cannot yet be predicted.

Constructive Datatypes

We require that our extrapolation algorithm apply to many datatypes, not just integers or strings. Let us adopt a fairly conventional formalism for datatypes, but with one added restriction. Let Ω be a finite set of formal operations called *constructors*. Each constructor is assigned a fixed integer *arity* designating the number of formal arguments it takes. The subset of arity zero constructors is assumed to be nonempty and represents the constants, or *generators*, for the type. The set of *expressions* over Ω includes the generators and is closed under application of the constructors Ω .

A type D is a family of equivalence relations over Ω that is compatible with the operations. Thus if d_i and e_i are equivalent expressions ($1 \leq i \leq n$) and f is an n -ary operation in Ω , then $f(d_1, \dots, d_n)$ is equivalent to $f(e_1, \dots, e_n)$.

Examples of familiar datatypes are:

- *Strings* over the alphabet $A = \{a, b\}$, where the only constructor (apart from constants a and b) is the associative binary function $\odot$ concatenating two strings. (Formally, Ω consists of $A \cup \{\odot\}$.) The two expressions $(a \odot b) \odot c$ and $a \odot (b \odot c)$ are equivalent strings since the string operation is associative.
- *Dotted pairs* over the set $A = \{a, b\}$. The constructor $(\cdot)$ makes a pair out of two values. Typical expressions include $a, b, (b \cdot b), (a \cdot (b \cdot a))$, etc.
- *Non-negative integers* generated from $\{0, 1\}$ by the binary constructors $+$ (addition) and $*$ (multiplication). Both constructors are associative and commutative.
- *Thurstone domain*: The letters $A, B, \dots, Z$ together with the constructors *successor* and *predecessor*, e.g., *successor*(A) = B and *predecessor*(B) = A .

One says a constructive datatype is *freely generated* (e.g., [Gallier, 1986]) if each value can be constructed from the generators by exactly one sequence of operations. Dotted pairs, for example, are freely generated, but strings and integers are not: the string abc can be constructed $(ab)c$ or $a(bc)$, and 4 can be constructed as $2 + 2, 3 + 1$, etc.

We define a partial ordering $\succeq$ on the expressions of the type D as follows: generators are the minimal elements; and if $y = f(\dots, x_i, \dots)$ for some constructor $f \in \Omega$, then $y \succeq x_i$ for all i . We call a datatype *constructive* if $\succeq$ extends to a partial ordering on the equivalence classes. On the non-negative integers, for example, this rules out multiplication by zero: Since $1 = 1 + 0$, $1 \succeq 0$; but $0 = 0 \times 1$, so $0 \succeq 1$; thus $\succeq$ is not a partial ordering. Disallowing multiplication by zero makes this a constructive type.

Our algorithm, given below, applies to all constructive types and even to composite types (e.g., pairs of strings). We assume that datatypes are recursively distinguishable, so that, for example, there is no confusion between the integer 123 and the string “123”.

Streams

A *stream* is a semi-infinite sequence $\langle s_1, s_2, \dots \rangle$ of values, all of the same datatype. The first element s_1

is called the *head*, and $\langle s_2, s_3, \dots \rangle$ is called the *tail*. We shall formalize *sequence extrapolation as learning a stream*, and to that end we define a new representation language for streams. This language defines our bias for the learning problem; the definition is intentionally crafted so that an efficient learning algorithm can be devised.

Note that the constructors for a type D can be applied to streams of type D : e.g., if $S = \langle s_1, \dots \rangle$ and $T = \langle t_1, \dots \rangle$ are integer streams, then so is $S + T = \langle s_1 + t_1, \dots \rangle$. Note also the constant stream $a = \langle a, a, \dots \rangle$ for $a \in A$.

Before giving the formal definition, we offer a simple example of a stream definition. Over the naturals with addition, the stream $E = \langle 1, 3, 5, \dots \rangle$ of odd naturals can be defined by the following equations: $E = \langle 1 \mid F \rangle$, $F = E + 2$. The first equation states that E is a stream whose head is the number 1 and whose tail is the stream F . F , which is $\langle 3, 5, 7, \dots \rangle$, is defined recursively as $E + 2$.

Definition 1 An *elementary description* (of type D) of a stream S is a block of one or more equations. The block has one of the following three forms:

- (*initial-value definition*) $S = \langle v \mid T \rangle$, where v is an item of type D , followed by an elementary description of T . (Intuitively v represents the head and T , the tail of the stream S .)
- (*functional definition*) $S = f(T_1, \dots, T_k)$, where f is a constructor of arity $k \geq 0$, followed by elementary descriptions of $T_1, \dots, T_k$. (Intuitively this represents the stream S as a function f of the streams T_i .)
- (*equality definition*) $S = U$, where U is the name of a stream in the parent hierarchy of S (see below). (This defines the stream S recursively in terms of another stream U .)

S is said to be the *parent stream* of T (in an initial-value definition) and of T_i ($1 \leq i \leq k$) (in a functional definition). The *parent hierarchy* of T (T_i , etc.) consists of S , the parent of S , its parent, and so on. A stream U named in an equality definition of S must belong to the parent hierarchy of S . ■

For example, over strings the description $S_1 = \{aba \mid S_2\}$, $S_2 = c \odot S_3$, $S_3 = S_1$ denotes the stream $\{aba, caba, ccaba, \dots\}$. Over the naturals the Fibonacci sequence is given by $S_1 = \{1 \mid S_2\}$, $S_2 = \{1 \mid S_3\}$, $S_3 = S_4 + S_5$, $S_4 = S_1$, $S_5 = S_2$. The streams of Examples 1 through 3 (except “??”) all have elementary descriptions.

The language we have defined so far still allows non-sensical definitions, e.g., $Q = R + 1$, $R = Q$. To eliminate these, we require that descriptions satisfy the following additional condition. Define a relation $\sqsupset_1$ such that if T occurs on the right-hand side of a functional or equality definition of a stream S , then $S \sqsupset_1 T$. Let $\sqsupset$ be the reflexive, transitive closure of $\sqsupset_1$. An elementary description for S is called *proper* if for no name T is $T \sqsupset_1 S$ and if $\sqsupset$ is a partial order. (In the preceding example, $Q \sqsupset R$ and $R \sqsupset Q$, so that the definition is not proper.)

The semantics of an elementary description assigns to each S_i in the definition a model stream $[S_i]$ as follows.

- If $S = \langle v \mid T \rangle$, then $[S]$ is the stream whose head is v and whose tail is $[T]$.
- If $S = U$, then $[S] = [U]$.
- If $S = f(T_1, \dots, T_k)$, then $[S]$ is the stream $f([T_1], \dots, [T_k])$.

Theorem An elementary stream description of S has a unique model $[S]$ if the definition is proper. Improper definitions may have no model, one model, or multiple models.

Example 4 The following definition for X of type pair is proper: $X = \langle a \mid Y \rangle$, $Y = (Z \cdot A)$, $Z = X$, $A = a$. The stream (model) corresponding to the name X is $\langle a, (a \cdot a), ((a \cdot a) \cdot a), \dots \rangle$. The stream Y is the tail of X . The stream names are ordered such that $Y \sqsupseteq Z \sqsupseteq X$ and $Y \sqsupseteq A$. Note that $X \not\sqsupseteq Y$.

A stream S that can be represented by an elementary description is called an *elementary stream*. The main result of this paper is an algorithm that learns elementary streams over any constructive datatype.

Induction Spaces

Our algorithm keeps several active hypotheses for the input stream and eliminates hypotheses as soon as they conflict with the input examples. The key to the algorithm's efficiency is the way it stores sets of hypotheses, particularly functional hypotheses. Consider Example 3 above. One of the hypotheses for this sequence is the functional hypothesis $(L \cdot R)$, where L is the stream a , $(a \cdot a), ((a \cdot a) \cdot (a \cdot a)), \dots$ and R is the stream b , $((c \cdot a) \cdot b), (c \cdot b), \dots$. If there are n_L hypotheses for L and n_R hypotheses for R , then we would have to store $n_L n_R$ stream hypotheses if we represented them individually. Instead, the algorithm keeps one set for L , another for R , and the functional hypothesis $(L \cdot R)$ for the stream. In this way it stores only $n_L + n_R + 1$ hypotheses.

We adopt the following representation. Let S be an input stream we want to learn. We maintain a set of hypotheses for S in different formats (initial-value format, equality format, etc.); this set is called an *induction space*. Hypotheses are the elements of the induction space. A *hypothesis* for S is a 3-tuple consisting of a datatype, a confirmed length (equal to the number of input values of S with which the hypothesis has agreed so far), and one of the following formats:

- *initial-value format*: a hypothesis in the form $\langle s_1 \mid \mathcal{H} \rangle$, where s_1 is equal to the head of the stream and $\mathcal{H}$ is the name of an “induction space” (defined below) for the set of possible tail streams.
- *functional format*: a hypothesis in the form $f(\mathcal{H}_1, \dots, \mathcal{H}_k)$, where f is a constructor of arity $k \geq 0$ in Ω and the $\mathcal{H}_i$ are the names of induction spaces.
- *equality format*: a hypothesis of the form $\mathcal{U}$ that is the name of another induction space.
- *unknown format*: This is a special placeholder with a confirmed length of zero representing a stream

whose first element has not yet been presented. We denote this hypothesis by $\square$.

One hypothesis has just one format, but for the stream S there may be several hypotheses with different formats. An *induction space* (or *space*) is a collection of hypotheses in different forms for the same stream. We assign to each induction space assigned a name (" S " for the input stream and arbitrary names for its substreams); without confusion we may assign to the space the same name given to the corresponding stream. Spaces also are assigned a datatype, a parent induction space, and a confirmed length equal to that of the hypotheses in the space (such lengths are all equal).

Example 5 On the dotted-pairs domain suppose the first element of an input stream S_1 is $(a \cdot b)$. The algorithm to follow will create an induction space $\mathcal{S}_1$ for S_1 with type "pair", confirmed length 1, and two hypotheses: an initial-value hypothesis $\langle (a \cdot b) \mid \mathcal{S}_2 \rangle$, and a functional hypothesis $(\mathcal{S}_3 \cdot \mathcal{S}_4)$. The induction space $\mathcal{S}_2$ will eventually contain hypotheses for the tail of S_1 , but since no examples of this tail stream are available, this induction space $\mathcal{S}_2$ is initialized to $\square$, with a confirmed length of zero. The induction spaces $\mathcal{S}_3$ and $\mathcal{S}_4$ contain hypotheses for the left half and right half of S_1 , respectively. The first example of $\mathcal{S}_3$ is " a ", so this induction space contains two hypotheses: the initial-value hypothesis $\langle a \mid \mathcal{S}_5 \rangle$ (where $\mathcal{S}_5$ is initially $\square$) and the constant functional hypothesis $a = \langle a, a, \dots \rangle$. $\mathcal{S}_4$ is like $\mathcal{S}_3$, but with b in place of a . (This example is continued below.)

The Sequence Extrapolation Algorithm

The algorithm receives the sequence $\langle s_1, s_2, \dots \rangle$ of values of a stream S and incrementally constructs an induction space of consistent descriptions of S . The algorithm also calls upon a halting criterion that determines when to stop and declare success. Note that values of *only* the stream S are given the algorithm: any substreams that are part of the elementary description of S are inferred by the algorithm.

Hypotheses for S are developed in the induction space called *MAIN-SPACE*, initially consisting of only $\square$. The algorithm ensures that a proper partial order is maintained among stream names in the induction space by assigning a numerical value called the *delay* to each new stream symbol and ensuring that equality forms refer only to symbols with smaller delay. Finally, this algorithm assumes that the elements in the input stream are presented without any errors, and that the elements belong to a freely generated datatype (a restriction which we remove subsequently).

The output is an induction space of hypotheses consistent with the input stream from which elementary descriptions may be extracted.

The Extrapolation Algorithm:

1. Initialize *MAIN-SPACE* to $\{\square\}$. The confirmed length is zero, and the delay is one. The assigned name is that of the input stream (here: S).
2. For $i := 1, 2, 3, \dots$ until the termination condition holds: Set *MAIN-SPACE* :=

EXTEND(*MAIN-SPACE*, s_i , i), where s_i = the i 'th value of the input stream S . Increment the confirmed length of *MAIN-SPACE* by one.

3. Output *MAIN-SPACE*.

The *EXTEND* Routines: Let $\mathcal{H}$ be a space, x an input item, and $i \geq 0$. The procedure *EXTEND*($\mathcal{H}$, x , i) replaces each hypothesis $H \in \mathcal{H}$ by zero or more hypotheses formed by extending H to account for the value x :

For each hypothesis $H \in \mathcal{H}$, $\mathcal{H} := (\mathcal{H} - \{H\}) \cup \text{EXTEND-HYP}(H, x, i, \mathcal{H})$

The routine *EXTEND-HYP*($H, x, i, \mathcal{H}$) is defined by cases on H :

- Case: H is $\square$.
Return the following set of hypotheses, each with a confirmed length of one:
 - The initial-value stream $\langle x \mid \mathcal{H}' \rangle$. The tail stream is assigned a fresh name $\mathcal{H}'$, a confirmed length of zero, and a delay one more than that of $\mathcal{H}$. It is initialized to contain only the hypothesis $\square$.
 - The existing stream name $\mathcal{U}$, provided $\mathcal{U}$ is in the parent hierarchy of $\mathcal{H}$, the delay of $\mathcal{U}$ is less than that of $\mathcal{H}$, and its initial element u_1 is equal to x .
 - The functional hypothesis $f(\mathcal{H}_1, \dots, \mathcal{H}_k)$, $k \geq 0$, where¹ $x = f(x_1, \dots, x_k)$, $f \in \Omega$ is a constructor, and $x \neq x_j$ for any j . Each $\mathcal{H}_j$ is a new space initialized to contain only $\square$ and immediately extended by calling *EXTEND*($\mathcal{H}_j$, x_j , 1). $\mathcal{H}$ is the parent space of each $\mathcal{H}_j$, and the delay of each $\mathcal{H}_j$ is equal to that of $\mathcal{H}$.
- Case: H is an initial-value hypothesis.
Let $H = \langle v \mid \mathcal{H}' \rangle$; return the set $\{\langle v \mid \text{EXTEND}(\mathcal{H}', x, i - 1) \rangle\}$. Increase the confirmed length of H by one.
- Case: H is $\mathcal{U}$, the name of another stream.
If $x = u_i$, return $\{H\}$ with a confirmed length of i ; otherwise, return the empty set (in effect, deleting H as a hypothesis).
- Case: H is a functional hypothesis.
Let $H = f(\mathcal{H}_1, \dots, \mathcal{H}_k)$, $k \geq 0$. If there exist no values $x_1, \dots, x_k$ such that $x = f(x_1, \dots, x_k)$ and $x \neq x_i$ (for any i), return the empty set (in effect, deleting H as a hypothesis). Else, let $x = f(x_1, \dots, x_k)$; return, with a confirmed length of i , the hypothesis $f(\text{EXTEND}(\mathcal{H}_1, x_1, i), \dots, \text{EXTEND}(\mathcal{H}_k, x_k, i))$.

Example 5 (Continued) Suppose the second value of S_1 is $(a \cdot a)$. The algorithm revises the space $\mathcal{S}_1$ and its subsidiaries as follows:

¹The expression $x = f(\mathcal{H}_1, \dots, \mathcal{H}_k)$ is unique, since by assumption the type is freely generated. See "Non-freely Generated Types", below.

$$\begin{aligned}
S_1 &= \left\{ \begin{array}{l} \langle (a \cdot b) \mid S_2 \rangle \\ (S_3 \cdot S_4) \end{array} \right\} & S_2 &= \left\{ \begin{array}{l} \langle (a \cdot a) \mid \square \rangle \\ (S_7 \cdot S_8) \end{array} \right\} \\
S_3 &= \left\{ \begin{array}{l} \langle a \mid S_5 \rangle \\ \mathbf{a} \end{array} \right\} & S_4 &= \left\{ \begin{array}{l} \langle b \mid S_6 \rangle \\ \mathbf{b} // [\text{deleted}] \end{array} \right\} \\
S_5 &= \left\{ \begin{array}{l} \langle a \mid \square \rangle \\ \mathbf{a} \\ S_3 \end{array} \right\} & S_6 &= \left\{ \begin{array}{l} \langle a \mid \square \rangle \\ \mathbf{a} \end{array} \right\} \\
S_7 &= \left\{ \begin{array}{l} \langle a \mid \square \rangle \\ \mathbf{a} \end{array} \right\} & S_8 &= \left\{ \begin{array}{l} \langle a \mid \square \rangle \\ \mathbf{a} \end{array} \right\}
\end{aligned}$$

Note that the space S_2 (the tail stream of S_1) has been extended to two hypotheses: an initial-value and a functional hypothesis ($S_7 \cdot S_8$). The first value of S_7 and of S_8 is $\mathbf{a}$. Note also that the constant stream $\mathbf{b}$ has been eliminated from the space S_4 since it is inconsistent with the value $\mathbf{a}$ in the second half of the second example. Finally, observe that S_3 occurs as an equality hypothesis in the space S_5 because the parent space of S_5 is S_3 and the first element of S_3 is $\mathbf{a}$. S_3 is *not* a hypothesis in the space S_6 because S_3 is not in the parent chain for S_6 .

Non-freely Generated Types. The preceding algorithm breaks input values s_i into functional components $s_i = f(s_{i_1}, \dots, s_{i_k})$ in order to find functional hypotheses. When datatypes are not freely generated, there may be more than one way to do so—e.g., the integer 4 can be written $1+3$ or $2+2$ or $3+1$. In such cases the preceding algorithm can be viewed as non-deterministic: if an external oracle were available to tell it which functional form $A+B$ was the “right” one, the algorithm would require no modification. Lacking such an oracle, however, the algorithm must consider all possible constructions.

Our approach is retain the single functional hypothesis $f(A, B)$ but replace each of the spaces A and B by a *tree of spaces* representing the ways of constructing the example. Consider the integer stream $\langle 3, 4, \dots \rangle$, for example. These first two values actually represent *six possible pairs of examples* to a functional form like $A+B$: for the left halfstream $A = \langle a_1, a_2, \dots \rangle$, a_1 could be 1 or 2 and a_2 could be 1, 2, or 3; and for $B = \langle b_1, b_2, \dots \rangle$, the corresponding values of b_1 would be 2 or 1 and, for b_2 , 3, 2, or 1. Thus instead of the single hypothesis of the form $A+B$, we must maintain six. Not all of these hypotheses will remain consistent with the subsequent examples. For lack of space we cannot give the full algorithm here, but the main point is that when the datatype is not freely generated, the same algorithm applies with some additional bookkeeping. Naturally, however, the number of hypotheses grows much more rapidly than it does for freely generated types.

Latency

Consider the following two descriptions of the stream $S_1 = \langle 1, 2, 3, \dots \rangle$,

$$\begin{aligned}
S_1 &= \langle 1 \mid S_2 \rangle & S_1 &= \langle 1 \mid S_2 \rangle \\
S_2 &= S_3 + S_4 & S_2 &= \langle 2 \mid S_3 \rangle \\
S_3 &= 1 & S_3 &= S_4 + S_5 \\
S_4 &= S_1 & S_4 &= 2 \\
&& S_5 &= S_1
\end{aligned}$$

The first description assumes only the first value as an initial value and predicts all the rest, whereas the second

description assumes the first *two* values as initial values and predicts the rest. Ordinarily we prefer the first for reasons of economy (Occam's razor). The *latency* of a description measures the amount of assumption built into it—higher latency means that more initial values are required before the hypothesis can begin making predictions.

Definition 2 Let S be a proper elementary description. The *latency* $\lambda(S)$ is defined recursively as follows.

- If $S = \langle v \mid T \rangle$, then $\lambda(S) = 1 + \lambda(T)$.
- If $S = f(T_1, \dots, T_k)$ ($k \geq 0$), then $\lambda(S) = \max_{1 \leq i \leq k} \lambda(T_i)$ if $k > 0$, or 1 if $k = 0$. In particular, if S is a constant stream, then $\lambda(S) = 1$.
- If $S = U$, then $\lambda(S) = 1$. ■

Example 6 In the following definition for X of type *pair* over the generators $\{a, b\}$,

$$\begin{aligned}
X &= \langle (b \cdot b) \mid Y \rangle \\
Y &= (Z \cdot W) \\
Z &= \mathbf{b} \\
W &= X
\end{aligned}$$

the latency $\lambda(X)$ is 2, and $\lambda(Y) = \lambda(Z) = \lambda(W) = 1$. ■

We may want to extract from an induction space the hypotheses of minimum latency. Induction spaces are intentionally structured so that extracting the hypotheses of smallest latency is easily accomplished with a simple branch-and-bound algorithm. For example, in the space of Example 5 after the second instance $\langle a \cdot a \rangle$ has arrived, there are two hypotheses of minimum latency ($\lambda = 2$): (1) $S_1 = \langle (a \cdot b) \mid S_2 \rangle$, $S_2 = (S_7 \cdot S_8)$, and $S_7 = S_8 = \mathbf{a}$; and (2) $S_1 = (S_3 \cdot S_4)$, $S_3 = \mathbf{a}$, $S_4 = \langle b \mid S_6 \rangle$, $S_6 = \mathbf{a}$. Both hypotheses assume all or part of the first example as an initial value and predict the remaining values.

An Application for Sequence Extrapolation

Is the extrapolation algorithm practical, and if so, for what applications? We have implemented the algorithm in Common Lisp taking full advantage of the object-oriented features for handling different datatypes. We also use lazy evaluation to delay exploring a hypothesis until all lower-latency hypotheses in the space have been ruled out. The implementation confirms what we have also proved theoretically elsewhere [Laird and Saul, 1992]: that the algorithm is efficient (polynomial in the size of the input examples) for freely generated datatypes and streams of bounded latency, but inefficient for non-freely generated types and for unbounded latency.

An application where sequence extrapolation can play a significant role is *inductive concept learning*, the task of formulating a general concept from examples and counterexamples. An area of active research, inductive programming has been applied to such problems in biology, structural engineering, and qualitative modeling, with modest success.

Most approaches (such as FOCL [Pazzani and Kibler, 1992], FOIL [Quinlan, 1990], GOLEM [Muggleton and Feng, 1992], etc.) depend exclusively on generalization

from ground examples. While this approach sometimes works well, it requires an enormous number of examples and searches through many possible generalizations. At the same time, these methods throw away much of the structure available in the examples. As a very simple example, imagine an arithmetic concept $\Delta(X, Y)$ defined by the following examples:

X :	0	1	0	2	1	0	3	2	1	0	4	3	...
Y :	0	0	1	0	1	0	0	1	2	3	0	1	...
Delta :	2	2	2	2	3	2	2	4	4	2	2	5	...

The structure of this concept becomes much more discernible if arranged so that the sequential patterns emerge:

Y	0	1	2	3
X :				
0	2	2	2	2
1	2	3	4	5
2	2	4	6	8
3	2	5	8	11

Almost immediately we can recognize that (for all integers X) $\Delta(X, 0) = 2$, $\Delta(X, 1) = \Delta(X - 1, 1) + 1$, $\Delta(X, 2) = \Delta(X - 1, 2) + 2$, ... Our sequence extrapolation algorithm can easily find each of these patterns. Then by generalizing from the patterns $\Delta(X, 0)$, $\Delta(X, 1)$, $\Delta(X, 2)$, etc., instead of from the ground examples, we see that

$$\Delta(X, Y) = \Delta(X - 1, Y) + Y,$$

with an initial value of $\Delta(0, Y) = 2$. Fewer than twenty examples have sufficed to infer a simple hypothesis for the concept. Surely this is more akin to the way humans infer concepts than the methods based solely on massive generalization.

Concluding Observations

Sequence extrapolation can be formalized as a problem of learning streams from their initial values. We have shown how the family of elementary streams over constructive datatypes can be effectively learned, using the Extrapolation Algorithm.

While elementary streams are a reasonably expressive family and many common sequences are elementary, many more sequences (e.g., the primes) are not elementary streams. The primary value of this work is to demonstrate that sequence extrapolation can be studied in much greater generality than heretofore, and that there exists a simple and efficient algorithm for at least an interesting class of sequences.

Elsewhere [Laird and Saul, 1992; Laird et al., 1993] we analyze the algorithm and prove it correct. As noted above, its complexity depends on the latency of the target stream and on whether the datatype is freely generated. We have also developed a model that quantifies how one's confidence in a hypothesis grows with continued success in predicting sequence values.

Empirically one finds that very few input values (e.g., five to ten) are needed to identify an elementary stream. Unfortunately, the sample size analysis turns out to be quite difficult in general, but for one particular datatype

we have calculated the sample size bound exactly. For this case the sample size is, indeed, very small—unlike statistical concept learning, where even very constrained concept classes require hundreds or thousands of examples.

We have also studied the problem of sequence extrapolation when the input values may contain errors. We hope to be able to report these results and some applications in the future.

Acknowledgments

The inspiration for this work came from Peter S. Dunning, who first discovered some intriguing relationships between string sequences and integer sequences. The authors thank Peter Dunning, Peter Friedland, Mike Lowry, Steve Minton, David Thompson, and Richard Waters for their suggestions.

References

[Dewdney, 1986] A. K. Dewdney. Computer recreations. *Scientific American*, pages 14–21, 1986.

[Dietterich and Michalski, 1986] T. Dietterich and R. Michalski. Learning to predict sequences. In R. S. Michalski et al., editor, *Machine Learning: An AI Approach, Vol. II*. Morgan Kaufmann, 1986.

[Gallier, 1986] J. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving*. Harper and Row, 1986.

[Hedrick, 1976] C. Hedrick. Learning production systems from examples. *Artificial Intelligence*, 7:21–49, 1976.

[Kotovsky and Simon, 1973] K. Kotovsky and H. Simon. Empirical tests of a theory of human acquisition of concepts for sequential patterns. *Cognitive Psychology*, 4:399–424, 1973.

[Laird and Saul, 1992] P. Laird and R. Saul. The extrapolation of elementary sequences. Technical Report FIA-92-31, NASA Ames Research Center, AI Research Branch, 1992.

[Laird et al., 1993] P. Laird, R. Saul, and P. Dunning. A model of sequence extrapolation. (submitted), 1993.

[Laird, 1992] P. Laird. Discrete sequence prediction and its applications. In *Proc., 9th National Conference on Artificial Intelligence*. AAAI, 1992.

[Michalski et al., 1987] R. Michalski, H. Ko, and K. Chen. Qualitative prediction: the SPARC/G methodology for inductively describing and predicting discrete processes. In *Current Issues in Expert Systems*. Academic Press, 1987.

[Muggleton and Feng, 1992] S. Muggleton and C. Feng. Efficient induction of logic programs. In *Inductive Logic Programming*, pages 281–298. Academic Press, 1992.

[Pazzani and Kibler, 1992] M. Pazzani and D. Kibler. The utility of knowledge in inductive learning. *Machine Learning*, 9:57–98, 1992.

[Persson, 1966] S. Persson. *Some Sequence Extrapolation Programs: A study of representation and modeling in inquiry system*. PhD thesis, University of California, Berkeley, 1966. Also printed as Stanford University Computer Science Department Technical Report # CS50, 1966.

[Pivar and Finkelstein, 1964] M. Pivar and M. Finkelstein. Automation, using LISP, of induction on sequences. In E. Berkeley and D. Bobrow, editors, *The Programming Language LISP*. Information International, Inc., 1964.

[Quinlan, 1990] R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.