BMWk Revisited

Generalization and Formalization of an Algorithm for Detecting Recursive Relations in Term Sequences

Guillaume Le Blanc*

LRI, URA 410 du CNRS, Bâtiment 490, Université de Paris-Sud,
F-91405 Orsay Cedex, France
E-mail : Guillaume.Le-Blanc@lri.fr

Abstract. As several works in Machine Learning (particularly in Inductive Logic Programming) have focused on building recursive definitions from examples, this paper presents a formalization and a generalization of the BMWk methodology, which stems from program synthesis from examples, ten years ago. The framework of the proposed formalization is term rewriting. It allows to state some theoretical results on the qualities and limitations of the method.

1 Introduction

Detecting recursive relations in term sequences is likely to have a number of applications in Machine Learning, Theorem Proving and other fields. During the last seventies and early eighties, an algorithm called BMWk (for Boyer-Moore-Wegbreit-Kodratoff) came out to be very fruitful in the field of program synthesis from examples [5, 7, 8, 19]. Unfortunately, this topic has sunk into oblivion mainly as its protagonists decided to investigate new ways of automatic programming since program synthesis from examples seems too difficult.

Recently, many Machine Learning research workers – especially in the field of Inductive Logic Programming – have come to the conclusion that it is essential to be able to learn recursive definitions [14].

This background compels us to examine the previous works in program synthesis. We studied the two last implementations of the BMWk algorithm [3, 17]. It appears that the methodology is very interesting and in some way close to the work of Lapointe and Matwin [1, 11].

Nevertheless, the BMWk methodology as it is exposed by Fargues [3] and Papon [17] suffers from some drawbacks (no distinction between operators and utilization strategy, unjustified heuristics, limited framework, conception errors) so that a formalization is absolutely necessary.

* This work is supported by the Esprit project BRA ILP n° 6020 and the french MRT through PRC-IA

Moreover, the BMWk algorithms, as opposed to most of those coming from Inductive Logic Programming [16, 18] follow strong directing rules and as LOPSTER [1] or as the algorithm of Idestam-Almquist [4] use fewer examples. These qualities are very interesting since the search space is gigantic.

In the light of works published during the last decade, it is possible to reformulate the BMWk methodology, building a solid formalization and even generalizing it so that it can handle the Inductive Logic Programming usual bench marks. The aim of this paper is to present a formalization and a generalization of the BMWk methodology. As we state in Sect. 6 this results in a powerful methodology able to synthesize complicated functions with several embedded level of recursivity directly from examples built with constructors (e.g. synthesize polynomial functions or exponential functions from examples built with 0 and s, without any other knowledge).

2 Preliminaries

In this Sect., we shall give the notations, definitions and the theoretical requisite propositions. Most vocabulary of this Sect. is defined and exemplified in a survey on term rewriting by Dershowitz and Jouannaud [2].

Let $\mathcal{F} = \{f, g, h, \ldots\}$ be a finite set of function symbols and let $\mathcal{X} = \{u, v, w, x, y, z, \ldots\}$ be a set of variables. Let $\mathcal{T}(\mathcal{F}, \mathcal{X})$ be the set of terms constructed using $\mathcal{F}$ and variables in $\mathcal{X}$ and $\mathcal{T}(\mathcal{F})$ the set of *ground terms* (i.e. terms without variables). Let $\mathcal{V}ar(T)$ be the set of all the variables which appear in the term T. Let $\mathcal{C}$, $\mathcal{C}'$ and $\mathcal{C}''$ be disjoint subsets of $\mathcal{F}$ which are designated to be sets of *constructor* symbols. Usually constructors are named with letters from the beginning of the alphabet $\{b, c, e, \ldots\}$. Constructors stand to represent data types and non-constructors represent functions (e.g. 0 and s are constructors for the natural integers 0, $s(0)$, $s(s(0)))$,...). Lists are built with two constructors **cons** and **nil**, the empty list. Instead of $\mathbf{cons}(x, y)$ and **nil**, we prefer the notation $[x|y]$ and $[]$.

A *substitution* σ or θ is a function from terms to terms which replaces the variables of a term by other terms (e.g. if $\sigma = \{x/a, y/f(x)\}$ and $T = g(x, g(x, y))$ then $T\sigma = g(a, g(a, f(x)))$). A substitution ρ which replaces variables for variables and which is bijective is called a *variable renaming*. Given two terms M and T, if there exists a substitution θ such that $M\theta = T$ we say that the *pattern* M *recognizes* the term T or that M *subsumes* T or that M is a *generalization* of T. The subsumption relation defines an ordering on terms. Given two terms S and T, G is a *least general generalization* (lgg) if G subsumes S and T and if all term which subsumes S and T also subsumes G. Two terms S and T *unify* if there exists a substitution σ such that $T\sigma = S\sigma$.

If L_i and R_i are terms in $\mathcal{T}(\mathcal{F}, \mathcal{X})$, the set of *equations* $\{L_i = R_i\}$ defines a *congruence* on $\mathcal{T}(\mathcal{F}, \mathcal{X})$ if $f(S_1, \ldots, S_n) = f(T_1, \ldots, T_n)$ whenever $S_i = T_i$ for all i, if for all substitution θ, $S\theta = T\theta$ whenever $S = T$ and if it is reflexive, transitive and symmetric. Equations define functions from constructors. For example, the addition can be recursively defined by $s(x) + y = s(x + y)$ and $0 + y = y$. A

destructor is a function which extracts the n^{th} argument of a constructor term; **car** and **cdr** are the destructors of the constructor **cons**. We use the notation a and d instead of **car** and **cdr**; a and d are defined by $a([x|y]) = x$ and $d([x|y]) = y$. We also use the notation $adddd(x)$ for $a(d(d(d(d(x)))))$.

Term rewriting gives a way to calculate equalities. If L_i and R_i are terms in $\mathcal{T}(\mathcal{F}, \mathcal{X})$, the set of *rewrite rules* $\{L_i \rightarrow R_i\}$ defines a *rewrite relation* on $\mathcal{T}(\mathcal{F}, \mathcal{X})$ if $f(S_1, \ldots, S_n) \rightarrow f(T_1, \ldots, T_n)$ whenever $S_i \rightarrow T_i$ for all i, if for all substitution θ, $S\theta \rightarrow T\theta$ whenever $S \rightarrow T$ and if it is reflexive and transitive (the notation for the reflexive and transitive closure of $\rightarrow$ is $\stackrel{*}{\rightarrow}$). A *rewrite system* (namely a set of rewrite rules) *terminates* if for any term T there is no infinite chain of rewriting from T. A rewrite system is *confluent* if for any terms T, U and V, there exists a term S such that $U \stackrel{*}{\rightarrow} S$ and $V \stackrel{*}{\rightarrow} S$ whenever $T \stackrel{*}{\rightarrow} U$ and $T \stackrel{*}{\rightarrow} V$.

A well-founded ordering is an ordering with no infinite decreasing chain. A *reduction ordering* is a well-founded ordering $\prec$ on $\mathcal{T}(\mathcal{F}, \mathcal{X})$ such that for all substitution σ and all terms T and T', if $T \prec T'$ then $T\sigma \prec T'\sigma$ and such that for all terms T_i, T and T', if $T \prec T'$ then $f(T_1, \ldots, T_{i-1}, T, T_{i+1}, \ldots, T_n) \prec f(T_1, \ldots, T_{i-1}, T', T_{i+1}, \ldots, T_n)$. The termination of a rewrite relation $\rightarrow$ is proved if it is included into a reduction ordering. The sub-term relation defines a reduction ordering. The sub-term ordering is a particular case of a more powerful class of reduction ordering called *recursive path ordering* [2].

3 Function Synthesis from Examples

First and foremost, we have to expose precisely the problem we want to solve. The BMWk methodology provides a way to compare terms in order to discover recursive relations among them. The main mechanisms that it uses are term matching and least general generalization calculus for two terms. Until now, the framework of BMWk was the LISP language for historical reasons. The examples have been defined as pairs of LISP lists and the solution of the synthesis as a LISP function. Since the BMWk algorithms in fact operate on terms we prefer to avoid this formalism. We use instead the notion of term rewriting [2] which supplies a more general background and which is closer to logic programming.

Given some examples of a function defined on terms by a congruence

reverse$([]) = []$
reverse$([a'']) = [a'']$
reverse$([a'', b'']) = [b'', a'']$
reverse$([a'', b'', c'']) = [c'', b'', a'']$
reverse$([a'', b'', c'', d'']) = [d'', c'', b'', a'']$

the BMWk algorithms generates a recursive definition of that function.

reverse$(x) \rightarrow f(x, x)$
$f([], v) \rightarrow []$
$f([x|y], v) \rightarrow f_1'([x|y], d(v), [a(v)])$
$f_1'([x], u, z) \rightarrow z$

$f_1'([x_1, x_2|y], u, z) \rightarrow f_1'([x_2|y], d(u), [a(u)|z])$
$a([x|y]) \rightarrow x$ and $d([x|y]) \rightarrow y$

Since it does not allow the constant a'', b'', c'' or d'' to appear in the definition of the function and since this problem can not always be solved by variabilizing the constants a'', b'', c'' or d'' as we will see later, the examples are written $f(I\sigma) = O\sigma$, where σ is a substitution which contains all the parts of the example which must not appear in the definition of f. For example, we write $\{x_1/a'', x_2/b''\}, \mathtt{reverse}([x_1, x_2]) \mapsto [x_2, x_1]$ instead of $\mathtt{reverse}([a'', b'']) = [b'', a'']$. $\mathcal{C}''$ will be the set of constructors $\{a'', b'', c'', d''\}$, $\mathcal{C}$ the set of the other constructors of the "inputs" $\{[\_|\_], []\}$ and $\mathcal{C}'$ the remaining constructors of the "outputs" (this set is empty in our example).

Definition 1 Input/Output example. Given $\mathcal{C}$, $\mathcal{C}'$ and $\mathcal{C}''$ three sets of constructor symbols and $\mathcal{X}$ a set of variable symbols, an example (or Input/Output example) of a function f is a triple (σ, I, O) where σ is a ground substitution on $\mathcal{T}(\mathcal{C} \cup \mathcal{C}' \cup \mathcal{C}'', \mathcal{X})$, I is a term from $\mathcal{T}(\mathcal{C}, \mathcal{X})$ and O a term from $\mathcal{T}(\mathcal{C} \cup \mathcal{C}', \mathcal{X})$ such that $\mathcal{V}ar(O) \subset \mathcal{V}ar(I)$. We write $\sigma, f(I) \mapsto O$ for an Input/Output example.

Definition 2 function synthesis from example. Given $\{\sigma_j, I_j \mapsto O_j\}_{j \in J}$ a finite set of Input/Output examples from a function f (which defines a congruence $=$ on terms), synthesizing f from the examples means to build a term rewrite system $\mathcal{R}$ such that for all j, $f(I_j\sigma_j) \rightarrow_{\mathcal{R}} O_j\sigma_j$.

The term rewrite system $\mathcal{R}$ is consistent with respect to f if for all ground terms I and O from $\mathcal{T}(\mathcal{C} \cup \mathcal{C}' \cup \mathcal{C}'')$ such that $f(I) \rightarrow^*_{\mathcal{R}} O$ then $f(I) = O$.

The term rewrite system $\mathcal{R}$ is complete if for all ground terms I and O from $\mathcal{T}(\mathcal{C} \cup \mathcal{C}' \cup \mathcal{C}'')$ such that if $f(I) = O$ then there exists a ground term O' such that $f(I) \rightarrow^*_{\mathcal{R}} O'$.

The term rewrite system $\mathcal{R}$ is correct if it is complete and consistent.

Given a set of examples $\{\sigma_j, I_j \mapsto O_j\}_{j \in J}$ it would be nice to be able to consider the rewrite system $\{f(I_j) \rightarrow O_j\}_{j \in J}$. Alas, this rewrite system is not necessarily consistent with respect to f. In particular, it can be inconsistent with respect to the examples. Let us see an example. Given two examples from `member`, with $\sigma = \{x/a'', y/b'', z/c''\}$:

$$\sigma, \mathtt{member}(x, [y, x]) \mapsto \mathtt{true} \quad \text{and} \quad \sigma, \mathtt{member}(z, [x, y]) \mapsto \mathtt{false}$$

the rewrite rule $\mathtt{member}(z, [x, y]) \rightarrow \mathtt{false}$ is contradictory to the examples (because the first example implies that $\mathtt{member}(a'', [b'', a'']) = \mathtt{true}$).

The method exposed in this paper only handles the case where the rewrite system is consistent with respect to the example.

Restriction 3. Given two examples $\sigma, f(I) \mapsto O$ and $\sigma', f(I') \mapsto O'$ and $=$ the syntactic equality, if there exists θ such that $f(I'\theta) = f(I\sigma)$ then $O'\theta = O\sigma$.

The functions `reverse`, `plus`, `times` among others (as opposed to `member`) meet this restriction ($\mathtt{reverse}([x, y]) = [y, x]$ no matter if $x = y$ or $x \neq y$).

In order to apply the BMWk methodology, the examples need a last transformation which consists in replacing the variables of O by their position in I. This position is given by the destructors. For example, $\mathbf{reverse}([x,y]) \rightarrow [y,x]$ is turned into $\mathbf{reverse}(x) \rightarrow f(x,x)$ and $f([x,y],v) \rightarrow [a(d(v)),a(v)]$ (since $ad([x,y]) = y$ and $a([x,y]) = x$). Since this transformation is not necessarily unique, we state a new restriction.

Restriction 4. Given an example $\sigma, f(I) \mapsto O$ the variables of O must have a single position in I.

Generally this restriction holds when the previous one holds. Also the functions `reverse`, `plus`, `times` bear this restriction.

If both restrictions hold, the example $\sigma, f(I) \mapsto O$ is turned into $f(I,v) \rightarrow O\theta$, where θ replaces each variable of O with its position in I. Assume, now, we have a set of rewrite rules $\{f(C_i,v) \rightarrow F_i\}_{i \in I}$.

4 Recursivity

Given a set of rewrite rules $\{f(C_i,v^m) \rightarrow F_i\}_{i \in I}$, where v^m is the variable vector $v_1,\ldots,v_m$, applying the BMWk methodology roughly consists in *"matching two consecutive terms."* The most important restriction of the algorithms described ten years ago was to only deal with linearly ordered examples. As Jouannaud and Kodratoff pointed out, the inputs of the examples (i.e. the C_i) have to belong to an *ascending linear domain* [5]. This condition still remains when they propose a methodology for two variables functions [6]. In his Ph. D. thesis, Papon [17] considers recursive relations with a step greater than one but still retains the linearity condition.

Because of these restrictions, the methodology was only able to handle recursive relations such that $f(s^k(n))$ is a function of $f(n)$ or $f([x|l])$ is a function of $f(l)$.

We would like to deal with more complicated definitions. For example, the Fibonnacci function is given by $f(ss(n)) = f(s(n)) + f(n)$, $f(s0) = s0$ and $f(0) = s0$. We can improve the methodology by defining the notion of *recursive scheme.*

Definition 5 recursive scheme. Given $\preceq$ a reduction ordering on $\mathcal{T}(\mathcal{C},\mathcal{F},\mathcal{X})$, a recursive scheme with respect to $\preceq$ is a finite set $\{M_\alpha \mapsto \{R_{\alpha\beta}\}_{1 \leq \beta \leq m_\alpha}\}_{\alpha \in A}$ where the M_α and the $R_{\alpha\beta}$ are terms from $\mathcal{T}(\mathcal{C},\mathcal{X})$ such that for all α and all β, $R_{\alpha\beta} \prec M_\alpha$. The M_α are called the patterns of the recursive scheme and the $R_{\alpha\beta}$ the recursive calls of M_α.

The notion of recursive scheme specifies which kind of recursive relations will be investigated by the BMWk methodology. The ordering condition is necessary to ensure that the relation given by the method can be calculated (i.e. the calculus terminates). The recursive scheme for the Fibonnacci function is $\{ss(x) \mapsto \{s(x),x\}, s(0) \mapsto \phi, 0 \mapsto \phi\}$.

Given a set of examples, it is not possible to detect a relation corresponding to any recursive scheme. So, it is necessary to adapt in some way the recursive scheme to the examples.

Definition6 recursive scheme compatibility. Given a set of terms $\{C_i\}_{i\in I}$, C_i is recursively recognized by a pattern if it is recognized and if the recursive calls also recognize some C_j with the same substitution up to a variable renaming.

A recursive scheme is compatible with a set of terms if each pattern recursively recognizes at least one C_i.

This definition is necessary because a relation like $f(ss(x)) = f(s(x)) + f(x)$ can not be established without some examples like $f(s^{n+2}(0))$, $f(s^{n+1}(0)$ and $f(s^n(0))$.

Recall that a synthesis problem is a finite set of rewrite rules $\{f(C_i, v^m) \rightarrow F_i\}_{i\in I}$ (Cf. Sect. 3). Given a recursive scheme $\mathcal{RS} = \{M_\alpha \mapsto \{R_{\alpha\beta}\}_{1\leq\beta\leq m_\alpha}\}_{\alpha\in A}$, I_α is a subset of the set of the indexes of the C_i that M_α recursively recognizes. We also assume that i belongs to some I_α if C_i is recursively recognized by some pattern.

We say that $\mathcal{RS}$ is complete with respect to $\{C_i\}_{i\in I}$ if each C_i is recursively recognized by at least one pattern and that it is consistent if each C_i is recursively recognized by at most one pattern. If $\mathcal{RS}$ is complete and consistent, the I_α form a partition[2] of I.

5 The BMWk Methodology

Now, we can expose the mechanisms of the BMWk algorithms. It can be stated as four operators or inference rules.

5.1 Example

Let us consider first a simple case, the synthesis of the `reverse` function which reverses the order of the items in a list. After the preliminary transformation exposed in Sect. 3 the problem becomes:

$f([], v) \rightarrow []$
$f([x_1], v) \rightarrow [a(v)]$
$f([x_2, x_1], v) \rightarrow [ad(v), a(v)]$
$f([x_3, x_2, x_1], v) \rightarrow [add(v), ad(v), a(v)]$
$f([x_4, x_3, x_2, x_1], v) \rightarrow [addd(v), add(v), ad(v), a(v)]$
$f([x_5, x_4, x_3, x_2, x_1], v) \rightarrow [adddd(v), addd(v), add(v), ad(v), a(v)]$
with $\texttt{reverse}(x) \rightarrow f(x, x)$.

Let us call $F_0, F_1, \ldots, F_5$ the right hand sides of the rules. Given the recursive scheme $\{[x|y] \mapsto \{y\}, [] \mapsto \phi\}$ we try to find $f([x|y], v)$ as a function of $f(y, v')$.

[2] All index $i \in I$ belongs to one and a single I_α

This can be achieved if there exists a substitution θ_i such that $F_{i+1} = F_i\theta_i$. Alas the matching between F_i and F_{i+1} fails because there is no substitution θ such that $[] = [a(v)]\theta$.

The generalizing rule (see Sect. 5.2 for technical details) is designed for this kind of situation. Let $G_{i+1} = \text{lgg}(F_i, F_{i+1})$, where "lgg" means the least general generalization ($G_4 = \text{lgg}(F_3, F_4) = [add(u_1), ad(u_1), a(u_1)|u_2]$). Since F_{i+1} is an instance of G_{i+1}, it is sufficient to find a recursive relation among the G_i to have a recursive definition for the function f. Now, the G_i sequence is more simple than the F_i sequence and it is probably possible to find a recursive relation among the G_i even if it is not possible among the F_i.

When generalizing new variables appear which are a priori different for each of the G_i because the least general generalization is defined up to a variable renaming. Their renaming (so that they become the same for each G_i) is quite complex and although Papon [17] and Kodratoff have studied it, it needs further work.

In our example, the new variables are called u_1 and u_2. Let f'_1 be the function which calculates the generalizations and let h_1 and h_2 be the functions which calculate the substitutions on u_1 and u_2. The calculus of f can be decomposed into the set of rules:

$$\begin{cases} h_1([x_1], v) \rightarrow d(v) \\ h_1([x_2, x_1], v) \rightarrow d(v) \\ h_1([x_3, x_2, x_1], v) \rightarrow d(v) \\ h_1([x_4, x_3, x_2, x_1], v) \rightarrow d(v) \\ h_1([x_5, x_4, x_3, x_2, x_1], v) \rightarrow d(v) \end{cases} \quad \begin{cases} h_2([x_1], v) \rightarrow [a(v)] \\ h_2([x_2, x_1], v) \rightarrow [a(v)] \\ h_2([x_3, x_2, x_1], v) \rightarrow [a(v)] \\ h_2([x_4, x_3, x_2, x_1], v) \rightarrow [a(v)] \\ h_2([x_5, x_4, x_3, x_2, x_1], v) \rightarrow [a(v)] \end{cases}$$

which defines the substitutions on u_1 and u_2,

$f'_2([], v) \rightarrow []$
$f'_1([x_1], u_1, u_2) \rightarrow u_2$
$f'_1([x_2, x_1], u_1, u_2) \rightarrow [a(u_1)|u_2]$
$f'_1([x_3, x_2, x_1], u_1, u_2) \rightarrow [ad(u_1), a(u_1)|u_2]$
$f'_1([x_4, x_3, x_2, x_1], u_1, u_2) \rightarrow [add(u_1), ad(u_1), a(u_1)|u_2]$
$f'_1([x_5, x_4, x_3, x_2, x_1], u_1, u_2) \rightarrow [addd(u_1), add(u_1), ad(u_1), a(u_1)|u_2]$

which corresponds to the generalized part (using the new variables u_1 and u_2),

$f([x|y], v) \rightarrow f'_1([x|y], h_1([x|y], v), h_2([x|y], v))$
$f([], v) \rightarrow f'_2([], v)$

the recursive definition for f.

The previous transformation consists in applying the generalizing rule. Now, since h_1 and h_2 are constant it is possible replace their rules with:

$h_1(x, v) \rightarrow d(v)$
$h_2(x, v) \rightarrow [a(v)]$

this process is called identifying.

The only remaining problem is the synthesis of f'_1. Following the new recursive scheme $\{[x_1, x_2|y] \mapsto \{[x_2|y]\}, [x] \mapsto \phi\}$, it is possible to match F'_i with F'_{i-1} (the right hand sides of the rules defining f'_1) since F'_i is an instance of F'_{i-1}. This yields the constant substitution $\{u_1/d(u_1), u_2/[a(u_1)|u_2]\}$. The functions

g'_1 and g'_2 calculate these substitutions. Since they are constant, the identifying rule applies and leads to the rewrite system:

$g'_1(x, u_1, u_2) \rightarrow d(u_1)$
$g'_2(x, u_1, u_2) \rightarrow [a(u_1)|u_2]$
$f'_1([x], u_1, u_2) \rightarrow u_2$
$f'_1([x_1, x_2|y], u_1, u_2) \rightarrow f'_1([x_2|y], g'_1([x_1, x_2|y], u_1, u_2), g'_2([x_1, x_2|y], u_1, u_2))$
$h_1(x, v) \rightarrow d(v)$
$h_2(x, v) \rightarrow [a(v)]$
$f'_2([], v) \rightarrow []$
$f([x|y], v) \rightarrow f'_1([x|y], h_1([x|y], v), h_2([x|y], v))$
$f([], v) \rightarrow f'_2([], v)$

Finally, replacing constant functions by their values and adding the definition of the destructors a and d, the system becomes more readable:

$\texttt{reverse}(x) \rightarrow f(x, x)$
$f([], v) \rightarrow []$
$f([x|y], v) \rightarrow f'_1([x|y], d(v), [a(v)])$
$f'_1([x], u_1, u_2) \rightarrow u_2$
$f'_1([x_1, x_2|y], u_1, u_2) \rightarrow f'_1([x_2|y], d(u_1), [a(u_1)|u_2])$
$a([x|y]) \rightarrow x$ and $d([x|y]) \rightarrow y$

The algorithm described in the thesis of Papon [17] synthesizes this program since the examples have an *ascending linear domain* [5] (i.e. a domain which is described by some predicates $\pi_i(x) = \pi_0(d^i(x))$ where d is made up of destructors).

Notice finally that the methodology needs a minimum number of four examples to yield this result.

5.2 Inference Rules

This section is rather technical and can be skipped by the reader who does not need a deep understanding of the BMWk methodology.

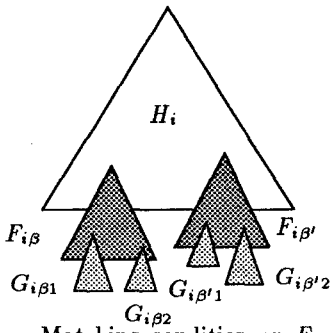
Assume that we have a set of rewrite rules and a set of recursive schemes (see the notations Sect. 4). When necessary we will choose a compatible recursive scheme to apply one of the following inference rules:

Identifying to synthesize the constant function on some domains.

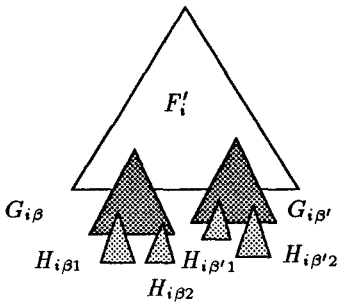
Given a recursive scheme without recursive calls.

$$\frac{\{f(C_i, v^m) \rightarrow F_i\}_{i \in I}}{\{f(M_\alpha, v^m) \rightarrow F_\alpha\}_{\alpha \in A}} \text{ if } \forall \alpha, i \quad i \in I_\alpha \Rightarrow F_\alpha = F_i$$

This means that if it is possible to group some C_i in a set where $f(C_i, v^m)$ is constant and has the value F_α and if the pattern M_α recognizes these C_i (e.g. M_α is the least general generalization of the C_i) then it is possible to replace the rules $f(C_i, v^m) \rightarrow F_\alpha$ by a single one $f(M_\alpha, v^m) \rightarrow F_\alpha$. This corresponds to infer by induction that $f(C, v^m)$ is constant for all term C recognized by M_α. In particular, if f is constant on all the examples it is possible to infer that f is constant on any entry.



Matching condition on F_i



Generalizing condition on F_i

Matching to detect recursive relations and synthesize a recursive definition. If it is possible to find some instances of $F_{i\beta}$ and $F_{i\beta'}$ into each term F_i (Cf. the figure *"Matching condition on F_i"*) and if C_i is greater than $C_{i\beta}$ and $C_{i\beta'}$ then some recursive relation like $f(n) = h(n, f(n-1, g_{\beta 1}(n), g_{\beta 2}(n)), f(n-2, g_{\beta' 1}(n), g_{\beta' 2}(n)))$ holds. The purpose of the matching rule is to detect such a recursive relation.

Given a recursive scheme complete with respect to $\{C_i\}_{i\in I}$, if for all α and all i in I_α, $F_i = H_i[u_\beta / F_{i\beta}[v_1/G_{i\beta 1}, \ldots, v_m/G_{i\beta m}]]_{1\leq\beta\leq m_\alpha}$ (Cf. figure), then the following inference rule applies:

$$\frac{\{f(C_i, v^m) \rightarrow F_i\}_{i\in I}}{\begin{array}{c}\{h_\alpha(C_i, v^m, u^{m_\alpha}) \rightarrow H_i\}_{\alpha\in A, i\in I_\alpha} \\ \{g_{\alpha\beta k}(C_i, v^m) \rightarrow G_{i\beta k}\}_{\alpha\in A, i\in I_\alpha, 1\leq\beta\leq m_\alpha, 1\leq k\leq m} \\ \{f(M_\alpha, v^m) \rightarrow h_\alpha(M_\alpha, v^m, f(R_{\alpha\beta}, \mathbf{g}_{\alpha\beta}(M_\alpha, v^m))^{m_\alpha})\}_{\alpha\in A}\end{array}}$$

with $\mathbf{g}_{\alpha\beta}(x, v^m) = g_{\alpha\beta 1}(x, v^m), \ldots, g_{\alpha\beta m}(x, v^m)$ and $f(R_{\alpha\beta}, \mathbf{g}_{\alpha\beta}(M_\alpha, v^m))^{m_\alpha}$ $= f(R_{\alpha 1}, \mathbf{g}_{\alpha 1}(M_\alpha, v^m)), \ldots, f(R_{\alpha m_\alpha}, \mathbf{g}_{\alpha m_\alpha}(M_\alpha, v^m))$. This rule applies each time the $F_{i\beta}$ match some disjoint subterms of F_i. Such a matching does not always exist. If it fails only on few examples it is possible to slightly modify the recursive scheme in order to deal with the failure cases as particular cases.

Generalizing to prepare the examples before applying the matching rule when it does not apply. It is not always possible to find embedded instances of $F_{i\beta}$ in F_i in order to apply the matching rule. But it could be possible to find some term $F'_{i\beta}$ "which looks like" $F_{i\beta}$. In that case it is possible to eliminate the differences between $F'_{i\beta}$ and $F_{i\beta}$ when replacing $F'_{i\beta}$ by $G_{i\beta}$, the least general generalization of $F'_{i\beta}$ and $F_{i\beta}$ (Cf. the figure *"Generalizing condition on F_i"*). The idea beyond this technique is to replace the wrong F_i sequence by the better sequence obtained when elimating the subterms H_{ijk} (Cf. figure). This very important idea has been discovered by Kodratoff [7]. Given a recursive scheme complete with $\{C_i\}_{i\in I}$, for all α and all i in I_α let $F_i = F'_i[u_\beta / F'_{i\beta}]_{1\leq\beta\leq m_\alpha}$. Given $G_{i\beta}$ the least general generalization of $F_{i\beta}$

and $F'_{i\beta}$, and $F'_{i\beta} = G_{i\beta}[u_1/H_{i\beta 1}, \ldots, u_p/H_{i\beta p}]$, the following inference rule applies:

$$\frac{\{f(C_i, v^m) \to F_i\}_{i\in I}}{\begin{array}{c}\{h_{\alpha\beta k}(C_i, v^m) \to H_{i\beta k}\}_{\alpha\in A, i\in I_\alpha, 1\leq\beta\leq m_\alpha, 1\leq k\leq p_\alpha} \\ \{f'_\alpha(C_i, v^m, u^{p_\alpha}) \to F'_i[u_\beta/G_{i\beta}]_{1\leq\beta\leq m_\alpha}\}_{\alpha\in A, i\in I_\alpha} \\ \{f(M_\alpha, v^m) \to f'_\alpha(M_\alpha, v^m, \mathbf{h}_{\alpha\beta}(M_\alpha, v^m)^{m_\alpha})\}_{\alpha\in A}\end{array}}$$

with notations similar to those of the matching rule. This rule applies when the matching rule does not, so that it constitutes in some way an attempt to force the matching. When generalizing, new variables appear. The renaming of these variables is a real problem that Papon explains in his thesis [17].

Composing to cut a complex problem into two sub-problems.

$$\frac{\{f(C_i, v^m) \to F_i\}_{i\in I}}{\begin{array}{c}\{g(C_i, v^m, u) \to G_i\}_{i\in I} \\ \{h(C_i, v^m) \to H_i\}_{i\in I} \\ f(x, v^m) \to g(x, v^m, h(x, v^m))\end{array}} \text{ if } F_i = G_i[u/H_i]$$

This inference rule splits the terms F_i into two subterms G_i and H_i. This rule is difficult to use because many cuts are a priori possible, whereas most of them lead nowhere.

6 Results

Now, we shall state a few general results about the qualities and limitations of the BMWk methodology. All the following results apply to any implementation of the method. The formalization we gave has the advantage to yield to very precise results on synthesized functions. We state below the most important ones.

Proposition 7 inference termination. *For all sets of examples and all choices of recursive schemes there is no infinite chain of inferences.*

Proof. Build a well-founded ordering which decreases with the inferences.

This result is very interesting because it ensures that the choice of a *"wrong"* inference or a *"wrong"* recursive scheme has no other consequence than delaying the final result. We can also show that the number of recursive schemes compatible with the domain is finite, that the number of variable renamings for the generalization is finite and that the number of uses of the composing rule is also finite thus there is a finite number of choices. Alas the search space is quite big.

Proposition 8 correction on the example. *Given $\sigma, I \mapsto O$ an example and $\mathcal{R}$ a rewrite system calculated by the BMWk methodology, $I\sigma \xrightarrow{*}_{\mathcal{R}} O\sigma$.*

Proof. Show that if $\mathcal{R} \vdash \mathcal{R}'$ then the relation $\xrightarrow{*}_{\mathcal{R}}$ is a subset of $\xrightarrow{*}_{\mathcal{R}'}$.

This means that the examples can be calculated from the recursive definition given by the method. It was not always the case with the algorithm of Fargues [3] and Papon [17].

Proposition 9 termination. *The rewrite system given by the BMWk methodology terminates if all the recursive schemes are valid with respect to the same recursive path ordering (in particular if it is the subterm ordering).*

Proof. Expand the ordering so that it decreases on the calculated rewrite rules.

According to the last proposition, the BMWk methodology restricted to subterm ordering generates term rewrite systems which terminate (i.e. for all possible evaluation strategies, the use of the calculated recursive definitions always leads to a value).

Proposition 10 confluency. *Given a recursive scheme $\mathcal{RS}$ used by the methodology, if there does not exist any pair of patterns which unify and if there does not exist any variable which has more than one position in a given pattern then the term rewrite system built by the BMWk methodology is confluent.*

Proof. Technical, show that the term rewrite system is *orthogonal* (a class of rewrite systems which are known to be confluent).

The confluency property ensures that the rewrite rules define a function (i.e. given an input the calculated recursive definition leads to at most one output). The hypothesis on the recursive scheme is not very restrictive so that it can hold in most cases.

Proposition 11 class of the synthesized functions. *The BMWk algorithm restricted to subterm ordering and natural integers can not synthesize a function which increases faster than the exponential functions but it can synthesize exponential functions.*

This means that for any function f synthesized by the BMWk algorithm restricted to subterm ordering and natural integers, there exists a constant c such that $|f(x)| \leq c^{|x|}$. The methodology restricted to unary functions on natural integers synthesizes exactly the functions $f(n) = \sum_i P_i(n)c_i^n$ where P_i is a polynomial with positive rational coefficients and c_i a positive natural integer constant. This result is very intuitive because of the analogy between matching and substraction so that $f(n)$ is a solution of some linear equation $f(n+k) = \sum_{i<k} a_i f(n+i)$ where a_i is an integer.

The methodology is able to synthesize other functions like Ackermann's function. But it is quite improper to insist on this fact because only the recursive scheme of Ackermann's function is complicated and the methodology gives no indication of how to choose the recursive scheme. More important is the fact that functions such as $f(x) = x^x$ can not be synthesized with the subterm ordering restriction (the definition $x^{y+1} = x \times x^y$ is validated by the subterm ordering).

It is quite difficult to define in the general case the class of the synthesized functions even with the subterm ordering restriction. Generally, when the subterm ordering restriction holds, BMWk can not synthesized functions which increase faster than the exponential functions but there exists some pathological cases (functions which increase arbitrarily fast). Moreover, BMWk can not synthesized functions which increase too slowly (logarithm,square root,...).

7 Related Works

In program synthesis from example, Summers [19] was the first to introduce the idea of comparing two terms F_i and F_{i+1} in order to discover that the first one is an instance of the second one ($F_{i+1} = F_i\theta$). Then Fargues showed that the substitution θ may be considered i dependent so that the same search process should apply with θ instead of F and Kodratoff refines the method considering G_i the least general generalization of F_i and F_{i+1} in order to deal with the case where θ does not exist [3, 7]. Then Kodratoff and Papon introduced a variable renaming algorithm for the G_i set [8, 17] to implement the idea of Fargues and Kodratoff.

Up to now, there exists only two implementations of BMWk [3, 17]. These implementations contain a conception error concerning the generalizing rule (see Sect. 5.2) which entails that the synthesized program does not always calculate all the examples. Despites this problem these algorithms are usually able to synthesize unary functions on linear domains (lists, integers, stacks,...). Introducing the notion of recursive scheme, we made a strong generalization of the method which allows to synthesize functions of any arity and needing several recursive calls so that it is now possible to synthesize functions on other data types such as trees, heaps, terms,...Nevertheless, this is not yet implemented. Moreover, our formalization (Sect. 5.2) allows to correct the conception error in the algorithms of Fargues and Papon (Proposition 8).

Recently, in Inductive Logic Programing several works came close to the BMWk methodology. Ling pointed out the necessity of learning recursive predicates[3] [14] and several algorithms have been proposed to achieve this task ([16, 18] among others). Most of these algorithms need numerous examples and even more counter-examples. Also, Ling has proposed an algorithm to learn from *"good examples"* [12, 13]. The notion of *"good examples"* is very close to our work since the BMWk methodology is able to learn from very small sets of examples. The algorithm of Lapointe and Matwin [1, 9, 10] could be seen in some way as a particular case of the work of Fargues although they now develop it in a different way. Finally, Idestam-Almquist [4] showed how to use subterm positions to discover recursive relations. This is close to the idea of Summers which consists in replacing output constants by their position in the input.

Among all the Inductive Logic Programing works, those of Lapointe and Matwin are the closest to BMWk. These algorithms are strongly driven by the

[3] Following the work of Kleene, Ling emphasized that some theories are not finitelly axiomatisable without using recursive definitions.

examples so that they avoid useless calculus and also use the notion of subterm unification. BMWk is more powerful since the class of the functions it synthesizes is larger. For example, it can synthesize all the polynomial functions from examples only containing the constructors 0 and s (this entails that the definitions BMWk built could contain an arbitrary number of recursivity levels) whereas the algorithm of Lapointe and Matwin can not synthesizes such functions without using some background knowledge. The BMWk methodology restricted to natural integer is based on the principle that if f is a polynomial function of degre d then $f(n+1) - f(n)$ is a polynomial of degre $d-1$. This analogy is possible since $u = v\theta$ entails $|u| = |v| + |\theta|$. It is the original idea which is behind the BMWk methodology.

All the functions that can be synthesized with the algorithms of Lapointe and Matwin [1, 9, 10] can be synthesized by BMWk. The main difference between their work and BMWk is rather technical. With the notations of Sect. 5.2, their algorithms try to find some relations of the form $F_i = H_i[F_{i-k}]$ with $H_i = H'_i\theta^k$ and $H'_i = \sigma^l[H'_{i-l}]$ where θ and σ correspond to what they call some generating terms. BMWk search for relations of the form $F_i = H_i[F_{i-1}\theta_i]$ where H_i and θ_i could verify some similar relations or may be constant, which is obviously more general. The reasons why BMWk use the substitution θ_i which does not appear in the work of Lapointe and Matwin are described in Sect 3 (non-instanciated examples, positions of input variables in the output). The other differences are that some of the algorithms of Lapointe and Matwin can use background knowledge whereas BMWk does not[4] and that they usually does not need consecutive examples whereas BMWk does.

The last important difference between BMWk and the Inductive Logic Programming usual methods is that it can not use instanciated examples (see Sect. 3) whereas the later usually only use instanciated ones.

8 Conclusion

Our work confers a new youth to the old BMWk algorithms. This is achieved by separating four fundamental operators from the calculus strategy. Introducing the notion of recursive scheme, we have done a strong generalization with respect to the *"state of the art"* of the BMWk methodology. This results in gain of power. Since our formalization uses the framework of term rewriting, we are able to state precise results on the calculated recursive definitions. Last, but not least, the BMWk methodology supplies a restricted search space and needs very few examples so that it is likely to be very useful in Machine Learning (in particular in Inductive Logic Programming).

The methodology is also interesting because it can be improved. First, to handle example sets which contain $\sigma, I \mapsto O$ and $\sigma', I' \mapsto O'$ such that there exists θ, $I\theta = I'\theta$ and $O\theta \neq O'\theta$ (our current work). This allows to deal with the **member** predicate, for example. By the way, it is probably possible to supply

[4] Nevertheless BMWk is able to synthesize the auxilliary knowledges that it needs.

background knowledge to the methodology in order to synthesize functions like **sort** which use the knowledge of the predifined $<$ predicate. Finally, it is very easy to extend the methodology to the synthesis of relations (more than one output is associated to a given input, $\mathbf{in}([a'', b'', c']) = a''$ or b'' or c'').

Acknowledgements

I would like to gratefully thank Céline Rouveirol, Jean-François Puget and José de Siqueira for their helpful advices and Yves Kodratoff my thesis advisor.

References

1. David W. Aha, Charles X. Ling, Stan Matwin, and Stephane Lapointe. Learning singly-recursive relations from small datasets. In F. Bergadano, Luc De Raedt, Stan Matwin, and S. Muggleton, editors, *Proceedings of the IJCAI workshop on inductive logic programming*, Chambéry, France, 1993.
2. Nachum Dershowitz and Jean-Pierre Jouannaud. *Handbook of Theorical Computer Science*, volume B, chapter Rewrite Systems, pages 243–320. MIT Press, 1990.
3. Jean Fargues. Une méthodologie pour la synthèse de programme : application à la synthèse à partir d'exemples. Thèse, Université Pierre et Marie Curie (Paris VI), June 1978.
4. Peter Idestam-Almquist. Recursive anti-unification. In Stephen Muggleton, editor, *Proceedings of the third international workshop on Inductive Logic Programming*, pages 241–253, Ljubljana, Slovenia, March 1993. J. Stefan Institute.
5. Jean-Pierre Jouannaud and Yves Kodratoff. Characterization of a class of functions synthesized from examples by a SUMMER's like method using a BMW matching technique. In *Proceedings of the 6th IJCAI*, pages 440–447, 1979.
6. Jean-Pierre Jouannaud and Yves Kodratoff. *A methodology for two variable functions synthesis from examples.* CRIN université de Nancy, 1979.
7. Yves Kodratoff and Jean Fargues. A sane algorithm for the synthesis of LISP functions from example problems : the Boyer and Moore algorithm. In *Proceedings of the AISB meeting*, pages 169–175, Hamburg, 1978.
8. Yves Kodratoff and Éric Papon. A system for program synthesis and program optimization. In *Proceedings of the AISB meeting*, pages 1–10, Amsterdam, 1980.
9. Stéphane Lapointe, Charles Ling, and Stan Matwin. Constructive inductive logic programming. In Stephen Muggleton, editor, *Proceedings of the third international workshop on Inductive Logic Programming*, pages 255–264, Ljubljana, Slovenia, March 1993. J. Stefan Institute.
10. Stéphane Lapointe, Charles Ling, and Stan Matwin. Constructive inductive logic programming. In Ruzena Bajcsy, editor, *Proceedings of the 13th IJCAI*, volume Vol. 2, pages 1030–1036. Morgan-Kaufmann, August 1993.
11. Stéphane Lapointe and Stan Matwin. Induction de programmes logiques récursifs fondée sur la sous-unification. In PRC-IA GRECO, editor, *Actes des 1$^{\text{ères}}$ Journées Francophones d'Apprentissage et d'Explication des Connaissances*, pages 3–14. AFIA AFCET, PRC-IA GRECO, April 1992.
12. Charles Xiaofeng Ling. Inductive learning from good examples. In *Proceedings of the 12th IJCAI*, volume Vol. 2, pages 751–756, Sydney, Australia, August 1991. Morgan-Kaufmann.

13. Charles Xiaofeng Ling. *Inductive Logic Programming*, chapter Inductive learning from good examples, pages 113–129. APIC. Turing Institute Press, academic press edition, 1992.
14. Charles Xiaofeng Ling. Inventing necessary theoretical terms to overcome representation bias. In *Proceedings of the ML92 workshop on biases in inductive learning*, Aberdeen, Scotland, July 1992.
15. Stephen Muggleton, editor. *Inductive Logic Programing.* APIC. Turing Institute Press, academic press edition, 1991.
16. Stephen Muggleton and C. Feng. *Inductive Logic Programming*, chapter Efficient induction of logic programs, pages 281–298. APIC. Turing Institute Press, academic press edition, 1992.
17. Éric Papon. Algorithmes de détection de relations de récurrence – application à la synthèse et à la transformation de programmes. Thèse, Université de Paris-Sud, April 1981.
18. Ross J. Quinlan. Learning logical definition from relations. *Machine Learning Journal*, Vol. 5(3):239–266, 1990.
19. P.D. Summers. A methodology for LISP program construction from examples. *Journal of the ACM*, Vol. 24:161–175, 1977.