A Tool for Automatic Code Generation from Schemas

Antonio Gavilanes, Pedro J. Martín, and Roberto Torres

Departamento de Sistemas Informáticos y Computación, Facultad de Informática,
Universidad Complutense de Madrid, 28040 Madrid, Spain
{agav,pjmartin}@sip.ucm.es, r.torres@fdi.ucm.es

Abstract. Algorithm design is one of the more neglected aspects in programming introduction courses. On the contrary, schemas focus on solution construction, since they gather common characteristics of algorithms, so they can be considered as algorithm cognitive units. In this paper, we go beyond the benefits of teaching schemas and we present a tool that incorporates their use. It automatically generates code from the application of schemas, allowing its integration into the class as a useful educational tool.

Keywords: Automatic code generation, schemas, recurrence relations.

1 Introduction

A first year programming course usually is devoted to instructing students on two different aspects: the programming language and the algorithm design. Most of the textbooks focus on the language itself, thus algorithms are scattered along the course to show the language features. As a consequence, the algorithmic knowledge is poorly organized and students find that each problem requires an innovative technique to be solved.

On the contrary, not so frequent trends structure students' instruction around problem analysis and solution construction, by means of teaching schemas [6] [8] [9]. Schemas join the common characteristics of the algorithms that solve a family of problems, thus they can be considered as algorithmic cognitive units that can be applied to build programs. Students must carefully analyze the problem to find the schemas that can be applied to solve it, instead of programming from scratch. This analysis is based on drawing analogies to identify the tasks whose solutions are well known [5]. For instance, when analyzing the query "*is x prime?*" or the calculation of "*trunc($log_2(x)$)*", for a given natural number $x \geq 2$, students should notice that both problems can be similarly solved by using a search schema. Indeed, we can look for the first natural number $y \geq 2$ such that *y divides x*, for the first one, and *pow(2,y)>x*, for the second one, where *pow(x,y)* computes x^y. In fact, students should be able to instance a skeleton like the following one, using Java syntax:

```
y= 2;
found= CONDITION;
while (!found) {
  y= y+1;
  found= CONDITION;
}
```

G. Allen et al. (Eds.): ICCS 2009, Part II, LNCS 5545, pp. 63–73 2009.

where `CONDITION` respectively corresponds to the expressions `x % y == 0`, and `pow(2, y)>x`. After execution, we use "`x is prime iff y=x`" to solve the first problem, and return `y-1` for the second one.

Teaching schemas has great benefits. Regarding students, it improves their ability for abstraction, it avoids a compulsive impulse to write code before knowing what to do; and it standardizes the code that different students could produce. From a teaching point of view, instructors can exploit a broad analysis of the schemas in order to automatically extend their properties to any solution based on them. Hence properties such as correctness, termination or complexity, can be stated once in a theoretical framework, instead of independently analyzing each program. Also, schemas provide students with important insights into the use of other algorithmic units, such as design patterns in later courses.

In this paper we go beyond teaching schemas, since we also take care of how they can be automatically applied to a given problem. Apart from teaching how to instantiate the variable parts of the schema (e.g. `CONDITION` in the examples above), we use a tool to generate the involved code. Thus, when students are asked to solve a problem, firstly they should represent the problem in order to supply it to the tool, and then, choose the proper schema. So the tool allows the student to focus on the schema, not on the syntax of the language, and autonomously to obtain running solutions to the problem, from the code that is automatically generated. Visual and iconic languages, and their programming environments, are also related to code synthesis for programming instruction [2] [3] [13]. But they are based on graphical description of the algorithms, thus our approach has a greater abstraction power, since it requires the specification of the problem instead.

2 Theoretical Framework

In an introductory programming course, schemas can be mainly used to solve two different tasks: *traversing* and *searching*. As we have seen, the primality test is an example of a searching process. The computation of *pow(x,y)* itself can be seen as an example of a traversal from *1* to the natural number *y*. Nevertheless, schemas must also be classified according to the way data are generated. We have been teaching them in three contexts: data built by recurrence relations, data obtained from an array, and data read from a file. Since the schemas involved in the exploration of recurrence equations are the simplest ones, we began developing a tool to solve them.

2.1 Recurrence Relations

In mathematics, a *recurrence relation* is an equation which defines a sequence recursively: each term of the sequence is defined as a function of the preceding terms [4]. To obtain a unique sequence from a recurrence relation, there must be some initial values that do not depend on other numbers in the sequence. A well-known example of recurrence relation is the Fibonacci sequence given by the equation $f_i = f_{i-1}+f_{i-2}$ and the initial values $f_0=1$, $f_1=1$. The *order* of a recurrence relation is the number of preceding terms occurring in the equation; so the order of the Fibonacci sequence is 2. The *index* of a term is its position in the sequence, beginning from *0*.

We do not intend to solve such relations, as usual in a discrete mathematics course. Actually, we are concerned about generating iterative algorithms to explore the

recurrence sequence. Thus, we do not care about the type of its terms –`float`, `int` or `boolean`-, nor the operators involved in the equation. We will only suppose that the related expressions are valid. As operands of the equation defining f_i, we allow not only the preceding terms of f, but also the index i itself, and the preceding terms of other recurrence relations. In the latter case, it is said that the relations have been simultaneously defined by a *recurrence relation system*. For instance, the system $f_0=1$, $g_0=0$, $f_i=g_{i-1}$, $g_i=f_{i-1}$, defines the characteristic functions of the predicates "*i is even*" (f) and "*i is odd*" (g). The order of a system is the maximum of the orders of its relations. Nevertheless, we will only consider systems whose relations have the same order. Systems not satisfying this condition can be completed by progressing on the relations fallen behind the rest. Finally, we also allow systems where f_i depends on the i-th term of a simultaneous relation. In order to avoid partiality in this case, a topological ordering between the relations of the system is required. For example, equations $f_0=0$, $g_0=0$, $f_i=g_i+f_{i-1}$, $g_i=f_{i-1}+g_{i-1}$ compose a proper system since g_i can be computed before than f_i.

Apart from the types and operators involved in the recurrence relations above presented, the computability they define can be compared to the class of primitive recursive functions [7].

2.2 The Schemas

We use schemas to solve three classic problems involved in the exploration of recurrence relation systems: (1) the *traversal problem*, that calculates the term occurring at a given index, (2) the *unbounded search problem*, which looks for the first term satisfying certain condition, and (3) the *bounded search problem*, which seeks the first term satisfying a condition up to a given index. Among the different schemas that can be designed to solve such problems, we present the following ones using Java syntax.

```
//TRAVERSAL
int i;
DECLARATION
INITIALIZATION
i= CURRENT_INDEX;
while (i<n) {
  i= i+1;
  STEP
}
```

```
//UNBOUNDED SEARCH
int i;
boolean found;
DECLARATION
INITIALIZATION
i= CURRENT_INDEX;
found= CONDITION;
while (!found) {
  i= i+1;
  STEP
  found= CONDITION;
}
```

```
//BOUNDED SEARCH
int i;
boolean found;
DECLARATION
INITIALIZATION
i= CURRENT_INDEX;
found= CONDITION;
while (!found &&(i<bound)){
  i= i+1;
  STEP
  found= CONDITION;
}
```

The three schemas use a `while` sentence to progress on the recurrence system, step by step. The variables `i` and `found` are related to the last computed terms of the system and denote their index and whether they satisfy `CONDITION`, respectively. Capitalized words are "holes" that must be properly replaced depending on the given recurrence system. `DECLARATION` and `INITIALIZATION` must be replaced with the corresponding variables, `CURRENT_INDEX` must be replaced with the order of the system minus one, and `STEP` must be replaced with the code required to progress on the system.

3 The CGR Tool

The CGR tool, which stands for *Code Generation for recurrence Relations*, produces Pascal, C and Java code (the *target* languages) from a specification of the involved recurrence relations. We present how it works by means of examples, and we show how its GUI looks by displaying different snapshots.

3.1 Example 1. Computing the Vertices of a Regular N-gon

As a first example, consider the problem of calculating the vertices of a regular *N*-gon (*N>2*) for a given side *length>0*. In order to apply the tool, the student must begin defining the recurrence relations required to solve the problem, which basically correspond to express how vertices coordinates develop. The solution we propose is based on the well-known turtle graphics [1]: assuming that the first vertex is placed at an initial arbitrary point, the coordinates of the next vertex yield after properly rotating the direction and moving forward the distance *length*. Let a_i be the recurrence defining the angle that must be rotated. It can be defined by:

$$a_0=0$$
$$a_i=a_{i-1}+2\pi/N, \quad i>0$$

The recurrences x_i and y_i define the coordinates of the successive vertices. If we start at the point `(0, 0)`, they can be defined as follows:

$$x_0=0,\ y_0=0$$
$$x_i=x_{i-1}+length*\cos(a_{i-1}), \quad i>0$$
$$y_i=y_{i-1}+length*\sin(a_{i-1}), \quad i>0$$

Each recurrence relation is provided to the tool by using a dialog box which requests the following inputs from the students: the name of the recurrence, the primitive type (`real`, `integer`, `boolean`) it holds, the expressions for the initial values, and the equation for the recurrence relation (Fig. 1).

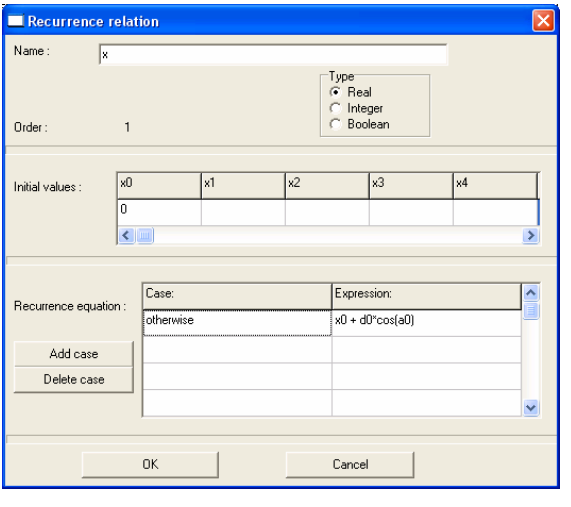


Fig. 1. Defining the recurrence relation `x`

In the generated code, identifiers `f0,…,fm-1` will hold the last generated terms of a given recurrence relation `f` of order `m`; so they must be used to state its recurrence equation. In the example, the identifier `x0` denotes the last term of `x`, and `x0` must be used to state the involved recurrence equation: `x0+length*a0` (Fig. 1). We tell students that they must instance the equation to compute the first new term ($x_1=x_0+length*\cos(a_0)$ in the example), when they provide CGR with the recurrence equations. Once students have defined the recurrences that compose the system, they have to determine which schema must be applied to solve the problem, and which target language (Pascal, C, Java) will be used in the code generation. This information is supplied to the tool by using a new dialog box which requires the names of the recurrences and the chosen schema(s). In our example, we must apply the traversal schema since we ask for all of the *N*-gon vertices (Fig. 2).

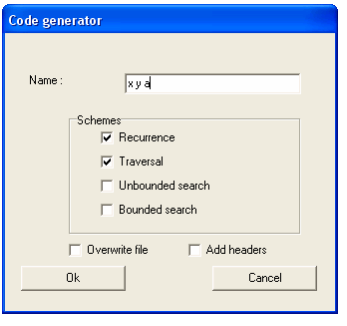


Fig. 2. Asking for an implementation

Finally, we show the generated code for the Java syntax.

```
int i;
float x0, x1, y0, y1, a0, a1;
x0=0; y0=0; a0=0;
i=0;
while (i<N) {
     i=i+1;
     x1 =x0 + d0*Math.cos(a0);
     y1 =y0 + d0*Math.sin(a0);
     a1 =a0 + 2*Math.PI/N;
     x0=x1; y0=y1; a0=a1;
}
```

3.2 Example 2. Carrying Different Weights

A worker is carrying different objects with different weights between two points. The first time he covers the distance, he carries $A>0$ units of weight. The second time, he carries $B>0$ ($A>B$) units of weight. As time goes by, his tiredness increases and he is forced to reduce the weight he can carry, which becomes the minimum between 95% of the last covered distance and 90% of the last distance but one. We pose the problem of determining the number of complete ways the worker can carry out before

he is exhausted, which occurs when the total carried weight exceeds *C*. We define a recurrence relation w_i to express the current weight, by the following equations:

$$w_0=A,\ w_1=B$$

$$w_i=minimum(0.95*w_{i-1},\ 0.90*w_{i-2}),\quad i>1$$

Notice that students must use a definition by cases in order to properly provide the tool with this equation. Thus, condition $0.90*w_0>0.95*w_1$ has to be supplied in the corresponding dialog box (Fig. 3).

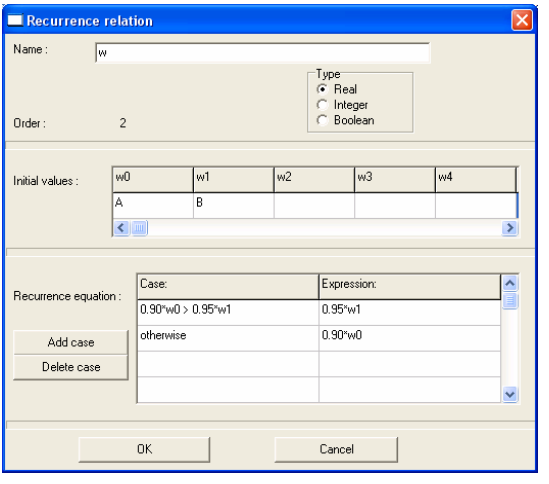


Fig. 3. Defining the recurrence relation w

Since we look for the first time the total carried weight exceeds *C*, we introduce a recurrence relation ac_i to hold the total carried weight:

$$ac_0=A,\ ac_1=A+B,$$
$$ac_i=ac_{i-1}+w_i,\ i>1$$

Observe that ac_i depends on w_i, thus the generated code must progress on the recurrence w before progressing on ac. The tool warns the student about such situation, and supplies a right ordering when required (Fig. 4 on the left).

Students must apply the unbounded search schema, thus the search condition has to be provided before generating the code (Fig. 4 on the right). Since we use the expression ac1>C, the value i-1 will finally return the number of complete ways before the worker becomes exhausted.

Fig. 4. Left: Advising that a proper ordering is required. Right: Asking for the search condition.

Then the generated Java code is the following:

```
int i; boolean found;
float w0, w1, w2, ac0, ac1, ac2;
w0=A; w1=B; ac0=A; ac1=A+B;
i=1;
found=ac1>C;
while (!found) {
  i=i+1;
  if (0.90*w0 > 0.95*w1) w2= 0.95*w1;
  else w2= 0.90*w0;
  ac2=ac1 + w2;
  w0= w1; w1= w2; ac0= ac1; ac1= ac2;
  found= ac1>C;
}
```

3.3 Integrating the Tool into the Course

We present the tool in the classroom after teaching the schemas for recurrence relations, which usually takes place at the end of the first out of two trimesters. Thus, the students have already been taught about their properties and about how schemas must be applied to specific problems. We have also presented some variations of the schemas (e.g. loops controlled by a counter or by a boolean expression), which are compared each other in order to gain the insights on them.

The tool is introduced to solve some of the problems they have manually coded previously. Students get really surprised when they notice that the tool solves the problems instantaneously. For the instructor, the tool can be used to prove that schema application is a real systematic process.

One hour is basically enough to explain the tool features. Next, the tool is uploaded to the web to make it public. Then students are encouraged to autonomously apply the tool to a selection of problems, as an optional lab assignment.

4 Evaluation of Schemas and the Tool

4.1 Study Framework

For the last two years, we have been analyzing the influence of using schemas on the development of students' programming skills, when teaching an introductory programming course during the first year of a Software Engineering degree, which applies the usual CS-first approach [12]. The study population was integrated in 7 groups each year, of around 70 students each; some of the groups (3 the first year, and 1 the second) studied schemas, while the others studied in a traditional language-oriented approach. Since students are randomly assigned to the groups, the groups are comparable with respect to the students' programming capabilities. We begin comparing the two approaches according to the academic success of the students, not considering other factors as the diversity of teachers and exams.

Only a few students decided to try CGR, despite of the extra mark that had been added to their final grade in case they would have solved some of the problems posed

in the lab assignment. Although they were not forced to solve all of them, they were asked to apply each of the three schemas at least once; hence, they should classify the chosen problems before trying to solve them, exploiting one of the schemas benefits. The assignment also included a satisfaction survey that students should fill in order to evaluate the tool.

4.2 Discussion about Academic Success

Table 1 reports the pass rates of the seven groups. The schemas-groups rates are displayed in boldface. The table points out that the schemas rates occupied the highest places, especially in 2007-2008.

Table 1. Pass rates

Group	1	2	3	4	5	6	7
2006-2007	31.7	**51.2**	51.4	**49.1**	**47.8**	27.7	39.7
2007-2008	29.7	**43.3**	25.9	25.9	23.5	40.2	51.5

4.3 Discussion about How Students Used the Tool

The programming assignment consisted of a list of 10 problems: six of them required the unbounded search schema (US), two the bounded search one (BS), and two the traversal one (T). Each student had to solve from 3 to 5 problems. The number of students that participated was 24 in 2007-2008. Table 2 shows the results we obtained. For each problem, we have studied four variables, which have been displayed in rows: the schema solving the problem (A), the number of students that chose the schema rightly|wrongly (B), the number of students that defined the recurrences rightly|wrongly (C), and the number of students that defined the condition of the (unbounded search or bounded search) schema rightly|wrongly (D).

Table 2. Report on the students' solutions

	1	2	3	4	5	6	7	8	9	10
A	US	US	BS	US	US	T	BS	T	US	US
B	1|0	2|0	11|9	6|2	12|3	19|0	11|0	12|0	4|0	14|0
C	1|0	2|0	16|3	8|0	13|2	12|6	10|0	9|3	2|1	13|1
D	1|0	2|0	5|5	5|2	10|3		10|0		2|1	14|0

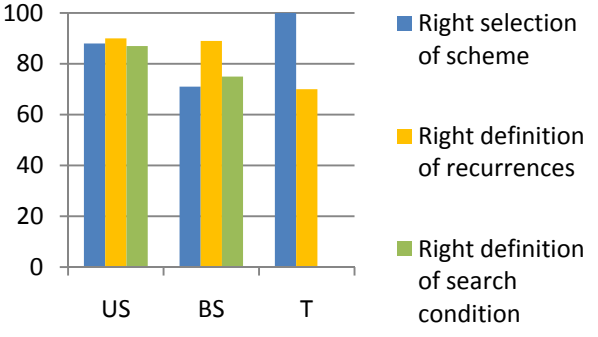
Table 3 summarizes the previous results, showing the success rates related to variables B, C and D for each schema. The average of these rates for the three schemas is 87% for the variable B, 84% for C, and 83% for D.

Thus, a descriptive analysis of the results shows that students usually choose the right schema, define the recurrence relations properly, and provide the tool with the correct search condition.

In order to analyze Table 3 more deeply, we have compared the percentages of the three schemas by pairs [11]. This study reports that the schemas US versus T, and BS

versus T, are different at the 95.0% confidence level (P<0.05), regarding variable B. Actually we conclude that problems based on traversals are more easily guessed, since the success rate for this schema is much higher than for the search ones.

Table 3. Analysis of students' solutions



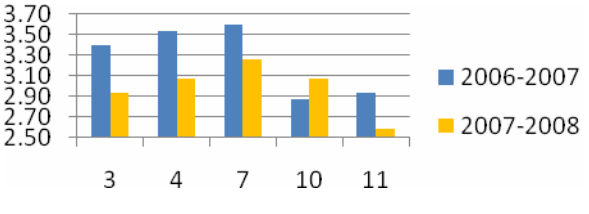
4.4 Discussion about Students' Compliance

The satisfaction survey was composed of 14 assertions. Students' answers ranged from 1 [total disagreement] to 5 [total agreement]. We focus on the most relevant questions:

3. "To define the relations involved in is an easier task than to code from scratch".
4. "To find out the relations takes less time than to code from scratch".
7. "The use of the CGR tool is suitable to solve problems in a first-year programming course".
10. "The CGR tool is useful to teach a first-year programming course".
11. "The CGR tool is a helpful tool to autonomous learning".
12. "Your programming background was broad before starting the course"

Table 4 shows the averaged answers to the previous questions in the two years. Notice that almost all of them exceed the middle value (3), thus students seem to be grateful for using the tool. We especially appreciate answers to question 3, 4 and 7, since they indicate that students find the tool useful to build programs.

Table 4. Students' answers to the selected assertions



In order to know whether the tool can be considered a useful tool we have analyzed the answers of 2007-2008 in depth. Concretely, we have studied whether any previous programming background affects the answers. Thus, we have applied the χ^2 test between answers to questions 3, 4, 7, 10 and 11 versus answers to question 12. Since the range of answers to these questions was too wide for 24 surveys, we have grouped the answers in order to safely apply this test. Thus, answers 1 and 2 have been replaced with 1, 4 and 5 with 5, and answer 3 has been ruled out. Then we have finally applied the independence tests over five 2x2 matrices. The study only reveals that answers to question 7 and 12 are not independent at the 95.0% confidence level ($P<0.05$) [10]. Concretely, on the one hand, 80.0% of the students with a broad background answered 1 to question 7, while 20.0% of them answered 5; on the other hand, 21.4% of the students with a narrow background answered 1 to question 7, while 78.6% of them answered 5. In consequence, we can conclude that the more previous background, the less they think that the tool is helpful to solve problems. In our opinion, students with previous programming skills do not like the tool because they feel uncomfortable when they cannot appeal to their own programming schemas.

5 Conclusions

Methodologies based on schemas are becoming popular for teaching programming in introductory courses. They focus on algorithm design instead of the language syntax, and use schemas as algorithm cognitive units. In this paper, we have presented a tool for programming using schemas. In order to solve a given programming problem, the student defines a recurrence relation system, selects the proper schema and the tool automatically generates the code that solves the problem in the target language. In this way, our tool allows the integration of methodologies based on schemas into the subject of the course.

We have also experimentally studied the influence of using schemas on the development of students' programming skills, and we have analyzed the students' compliance with the tool. The results we have obtained reveal that students assimilate schemas well. Regarding the tool, students find it suitable to solve problems and helpful to autonomous learning.

References

1. Abelson, H., di Sessa, A.A.: Turtle Geometry. MIT Press, Cambridge (1981)
2. Calloni, B., Bagert, D.: Iconic Programming Proves Effective for Teaching the First Year Programming Sequence. In: SIGCSE 1997, pp. 262–266. ACM Press, New York (1997)
3. Carlisle, M., Wilson, T., Humphries, J., Hadfield, S.: RAPTOR: A Visual Programming Environment for Teaching Algorithmic Problem Solving. In: SIGCSE 2005, pp. 176–180. ACM Press, New York (2005)
4. Grimaldi, R.P.: Discrete and Combinatorial Mathematics. Addison Wesley, Reading (2003)
5. Muller, O.: Pattern Oriented Instruction and the Enhancement of Analogical Reasoning. In: ICER 2005, Seatle, Washington, USA (2005)

6. Muller, O., Haberman, B., Ginat, D.: Pattern-Oriented Instruction and its Influence on Problem Decomposition and Solution Construction. In: ITiCSE 2007, pp. 151–155. ACM Press, New York (2007)
7. Odifreddi, P.G.: Classical Recursion Theory. North Holland, Amsterdam (1992)
8. Scholl, P.C., Peyrin, J.P.: Schémas Algorithmiques Fondamentaux. Séquences et iteration. Masson (1991)
9. Soloway, E.: Learning to program = learning to construct mechanisms and explanations. Comm. ACM 29(9), 850–858 (1986)
10. SPSS v.15, SPSS Inc. (1989-2006), `http://www.spss.com`
11. STATISTICA v. 7.1. SatSoft, Inc. (2005), `http://www.statsoft.com`
12. The Joint Task Force for Computing Curricula. Software Engineering 2004 (August 2004)
13. Watts, T.: The SFC Editor: A Graphical Tool for Algorithm Development. JCSC 20(1), 73–85 (2004)