From Program Synthesis to Optimal Program Synthesis

Joaquin Reyna
Department of Computer Science
University of Texas at El Paso
500 W. University
El Paso, TX 79968, USA
Email: magitek7@gmail.com

***Abstract*—In many practical situations, we know the values of some quantities $x_1, \ldots, x_n$, we know the relations between these quantities, the desired quantity y, and maybe some auxiliary quantities, and we want to estimate y. There exist automatic tools for such estimations – called *program synthesis* tools.**

A program synthesis tool usually generates *a* program for computing y. In many cases, however, several such program are possible, and it is desirable to generate the optimal (e.g., the fastest) program. In this paper, we describe algorithms aimed at such optimal program synthesis.

The problem can be interpreted in logical terms, as assigning fuzzy-style degrees to rules describing relations between variables.

I. Need for Data Processing: A Brief Reminder

One of the main objectives of science: an informal description. One of the main objectives of science is to describe the world.

One of the main objectives of science: formulation in precise terms. In precise terms, the above objective means that we would like to know the values of the numerical characteristics that characterize different objects.

Sometimes, this task is easy. Some of these values, we can simply measure.

In other cases, the task is more complex. There are many physical characteristics which are difficult or even impossible to measure directly, for example,

- the distance to a faraway star, or
- the amount of oil in a well.

Since we cannot measure these difficult-to-measure values y *directly*, we thus measure them *indirectly*: namely, we measure the values of related easier-to-measure characteristics $x_1, \ldots, x_n$, and we find (and list) all possible relations between these characteristics x_i and the desired quantity y (and maybe some auxiliary difficult-to-measure characteristics).

Based on these relations, we design an algorithm $f(x_1, \ldots, x_n)$ that, given the values $x_1, \ldots, x_n$ returns the (estimate for) the desired quantity $y = f(x_1, \ldots, x_n)$.

First example: computations are straightforward. To find the distance to a star, we can use the *parallax method*: we observe the position of this star on the celestial sphere in different seasons, when the Earth is at different sides of the Sun, and then use trigonometry to determine the desired distance.

In this example, the algorithm $f(x_1, \ldots, x_n)$ can be described by explicit formulas – and indeed, these computations were successfully done "by hand" already many centuries ago, when no computers were available.

Second example: computations are complex. To estimate the amount of oil in a well, we perform a large number of seismic experiments, by

- making small explosions at different locations on the Earth surface and
- measuring the time that the corresponding seismic waves took to travel to the sensors placed at different other surface locations.

Based on these times, we:

- solve the partial differential equation describing how seismic waves propagate, and
- thus find the velocity of sound at different locations and at different depths.

We then estimate the amount of oil by computing the volume at which the velocity of sound is consistent with oil.

This is a very simplified version of the actual estimates, but even after this simplification, one san see that the corresponding algorithm $f(x_1, \ldots, x_n)$ (which includes solving partial differential equations as an intermediate step) is very complex. In this case, we need a large number of computations which are very difficult (and often impossible) to do by hand. Thus, we need *data processing*.

This is one of the main reasons why computers were invented in the first place – to perform data processing and thus, to determine the values of difficult-to-measure quantities.

Prediction: another problem for which data processing is needed. Another important goal of science is to *predict* the future state of the world, i.e., in more precise terms, to predict the future values of the desired physical quantities. We want to predict the trajectory of an asteroid approaching Earth, we want to predict the path of a hurricane, etc.

In some cases, we have explicit formulas $y = f(x_1, \ldots, x_n)$ that describe the desired future values in terms of the current values x_i of different quantities. However, in many other cases, we only know some relations between y, x_i, and some additional auxiliary characteristics – such as the future values of other physical characteristics. Our objective is then to use these relations to design an algorithm that transforms the known values $x_1, \ldots, x_n$ into the desired prediction.

Engineering design problems. In engineering, we want to design an object with given characteristics. For example,

- while in science, a typical problem would be to predict a trajectory – e.g., to decide whether a given asteroid will hit the Moon,
- in engineering, a typical problem is to find the starting velocity and the starting direction for which the spaceship reaches the Moon.

Sometimes, there are ready algorithms for such design. In many other cases, however, we only know

- the relations between different characteristics, and
- the desired values of some of these characteristics.

We need to find the values of the set characteristics for which we will achieve the desired goals.

II. Program Synthesis: Successes and Open Problems

A general problem. In all the above situations:

- we know the values of some of these quantities $x_1, \ldots, x_n$, and
- we know the relations between these quantities, the desired quantity y, and maybe some auxiliary quantities.

Our objective is to use the known values and the known relations to find the values of the desired quantities.

How this problem is traditionally solved. Traditionally, in science and engineering, researchers and practitioners use their own creativity to come up with an algorithm $f(x_1, \ldots, x_n)$ that

- inputs the values of the known quantities, and
- outcomes an estimate for the desired quantity y.

Designing such an algorithm is a very difficult task.

The main idea of program synthesis. It turns out that in many cases, we can have a system that automatically analyzes the relation and synthesizes the desired program. Such automatic systems are called systems for *program synthesis*; see, e.g. [12].

Toy example. Let us use a simple example of a triangle to illustrate how program synthesis works.

A triangle is described by its angles A, B, C and side lengths a, b, c. We know the following relations between them:

- $A + B + C = \pi$ (the sum of the angles is 180°, or π radians),
- $a^2 + b^2 - 2 \cdot a \cdot b \cdot \cos(C) = c^2$ and similar expressions for a and b (cosine theorem), and
- $\dfrac{a}{\sin(A)} = \dfrac{b}{\sin(B)} = \dfrac{c}{\sin(C)}$ (sine theorem).

Now we can ask all kinds of questions:

- If we know a, b and c, can we determine A? and if yes, how?
- If we know a, b and A, how to compute b? and if yes, how?

Preparing for the program synthesis. First, we analyze which quantities are directly computable from which.

Suppose that we have a relation $F(A, B, C) = 0$. If we know all of these values but one (for example, A and B), then we have an equation with one unknown, from which in general we can compute C. So, if we already know A and B, then we are able to compute C. We will describe this implications, for short, as $A, B \to C$.

Similarly, if we know A and C, then we can compute B, and from B and C we can compute A.

So each equation leads to as many computability relations as there are unknowns in it. In our case we get three computability relations: $A, B \to C$; $A, C \to B$; and $B, C \to A$.

Example. In the triangle case, the relations turn into the following formulas:

- $A, B \to C$; $B, C \to A$; $A, C \to B$; (these three come from the equation $A + B + C = \pi$)
- $A, a, b \to B$; $A, a, B \to B$; …(from sine theorem), and
- $a, b, C \to c$; $a, b, c \to C$; $a, c, C \to b$; $b, c, C \to a$; …(from cosine theorem).

Wave algorithm for program synthesis. Let us describe a natural algorithm for deciding whether the desired quantity y can be computed based on the known quantities x_i.

According to this algorithm,

- We first mark the variables that we know.
- Then, when we look at all the rules, find those, for which all the conditions are marked and the conclusion is not, and mark the conclusion.
- Then we repeat the same procedure again and again.

After each iteration,

- either we did not add anything – which means that we are done (nothing else can be computed),
- or we add at least one marked variable.

Since there are finitely many variables, this process will eventually stop:

- If the desired y is marked, then we *can* compute it,
- else we *cannot* compute y.

Once we have a sequence of rules that lead to computing y, we can combine the corresponding algorithms and come up with a program for computing y based on x_i.

Example. Suppose that in the triangle, we know A and B, and we want to compute C and a.

Then, according to the algorithm, we first mark A and B.

There is only one rule whose conditions are marked: the rule $A, B \to C$. So, we mark C.

On the second iteration, we find three rules whose conditions are marked: $A, B \to C$; $B, C \to A$; and $B, C \to A$, but their conclusion have already been marked. So, we stop.

As a result, C is marked, which means that we can compute C. Moreover, we know how to compute C: C was obtained from a rule $A, B \to C$ that stems from $A + B + C = \pi$, so we must solve an equation $A + B + C = \pi$, in which A and B are known, and C is the only unknown.

As for a, it is not marked, and therefore, cannot be computed.

Comment. It should be mentioned that while the above description captures the main idea of program synthesis, the actual program synthesis algorithms implemented, e.g., [12], are more complex.

Boolean logic interpretation of the wave algorithm. The program synthesis problem can be reformulated in *logical* terms. Namely, we can interpret each rule $A, B \to C$ that stem from the relations as a propositional formula $A\&B \to C$ with variables $A, B, \ldots$ that can take the values "true" or "false":

- "true" means that we can compute the corresponding variable, and
- "false" means that we cannot.

So our knowledge can be represented as a set of propositional formulas that include all the rules and all the atoms A that represent the known variables x_i.

We want to know whether the value y is computable, or, in the propositional terms, whether the variable that corresponds to y is true. So, in logical terms, we want to know whether these variables are deducible from the knowledge base.

We can therefore use logical deduction tools to check whether y is deducible from x_i; see, e.g., [6], [7], [8], [9], [10], [11], [12], [13].

Triangle example. In the triangle example, we have a knowledge base $A\&B \to C$; $B\&A \to C$; $\ldots$; A; B, and we want to know whether C and a follow from these formulas.

An example of a practical application. The logical deduction tools have been used to automate program synthesis in space missions such as the NASA Cassini mission to Saturn; see, e.g., [6], [7], [8], [9], [10], [11], [13].

Limitations of the Boolean logic approach. There exist cases in which this logical approach does not work. Let us consider the case when we want to know the values of two unknowns y_1 and y_2, and we know two relations between them: $y_1+y_2-1 = 0$ and $y_1 - y_2 - 2 = 0$. In this case, we can determine both y_1 and y_2, because we have a system of two linear equations with two unknowns.

However, in this example, the above logical approach will not work; indeed:

- The first equation will translate into two rules $Y_1 \to Y_2$, $Y_2 \to Y_1$, where propositional variables Y_i correspond to y_i.
- The second equation will lead to these same rules.

From these two formulas we cannot logically conclude that Y_1 is true (because if Y_i are both false, still both rules are true), and therefore, we cannot conclude that y_i are computable.

Possibility to use fuzzy-type logic. In [1], it was shown that such examples can be handled if we replace the original Boolean logic with a more complex fuzzy-type logic.

Remaining problem. The existing program synthesis algorithms produce *a* program for computing y. In many cases, there are several possible ways of computing y.

In the triangle example, if we know A, B, a, b, and c, and we want to compute C, then we have several alternatives:

- we can use the rule $A, B \to C$ corresponding to the relation $A + B + C = \pi$, and find C as $\pi - A - B$;
- alternatively, we can use the sine rule $A, a, c \to C$ corresponding to the relation $\dfrac{a}{\sin(A)} = \dfrac{c}{\sin(C)}$, and compute C as $C = \arcsin\left(\dfrac{c \cdot \sin(A)}{a}\right)$;
- we can use even more complex formulas related to the cosine rule.

In such situations, it is desirable to come up with the *fastest* program – or, more generally, a program that spends the smallest amount of resources.

This may include the resources needed to measure the input characteristics x_i.

In other words, instead of program synthesis, we need *optimal* program synthesis.

What we do in this paper. In this paper, we interpret the problem in logical terms, as assigning fuzzy-style degrees to rules describing relations between variables. As a result, we develop algorithms aimed at optimal program synthesis.

III. Towards Algorithms for Optimal Program Synthesis

Formulation of the problem. Let us assume that for each known variables a and for each rule r of the type $a, b \to c$, we know the corresponding *weight* $w(a)$ or $w(r)$ – describing the amount of resources that are needed

- to measure the value a or
- to compute c based on the known values a and b.

To estimate the amount of resources needed to perform a sequence of measurements and computations, we simply add the corresponding weights.

Our objective is to select, among all paths that lead to the desired quantity y, the path with the shortest overall weight.

We can simplify the description by only assigning weights to edges. In the standard program synthesis, a variable that can be measured is assumed to be known.

Actually, in practice, we can only measure those variables that we need for deriving y, and we do not have to measure the values of all other variables. However, in the standard program synthesis, we do not take into account the resources that go into measurements, so we do not have to distinguish between measurements that we actually do and measurements that we an potentially perform.

However, in our (more realistic) formulation, we do take measurement efforts into account. To properly describe these efforts, we:

- denote by 0 the starting point, where no resources have been spent and thus, no measurements have been made, and
- describe each potentially measurable variable a by a rule $0 \to a$ with the weight $w(a)$ (describing how many resources we need to spend to measure its value).

In this new description, weights are only assigned to rules.

Logical interpretation of weights. We can interpret these weights as *degrees* in the style of fuzzy logic (see, e.g., [4], [5]): if the derivation takes a long time, this means that we should not be using this implication unless it is absolutely necessary.

So, we can interpret $1 - d$ as the "degree of confidence" in using this rule.

A possible solution: exhaustive search. In the toy example of a triangle, we can simply enumerate all possible paths and select the one with the smallest weight.

However, the number of possible paths grows exponentially with the number of variables. In the practical applications (like the space navigation problem mentioned earlier), we have dozens and hundreds of variables, so exhaustive search will take too long. For example, for $n \approx 300$, 2^n computations require longer time that the lifetime of the Universe :-(

Simplest case: rules of the type $a \to b$. To look for faster algorithms, let us start with the simplest case when all the rules have the form $a \to b$, i.e., when all the rules only have one input. This simplest case can be described by a *directed graph* in which nodes are variables, and variables a and b are connected by an edge if there is a rule $a \to b$. Non-negative weights are now assigned to edges.

In this case, estimating y simply means that we have a sequence of rules $0 \to a \to \ldots \to b \to y$, i.e., that we have a path from the starting node 0 to the desired node y. The optimal program corresponds to the *shortest path* from 0 to y.

Efficient algorithms for the simplest case. Efficient algorithms are known for computing shortest path in a graph; see, e.g., [2]. Thus, we can use these algorithms to solve the above simplest case of the optimal program synthesis problem.

Our objective is to generalize these algorithm to the more general case. In view of this objective, let us describe these algorithms in some detail.

Most of the efficient shortest path algorithms are based on the following *dynamic programming* idea. We want to find the length of the shortest path from the fixed node 0 to different nodes y. The shortest path cannot visit a node twice: otherwise, we can shorten it by cutting out the part between the two visits. Thus, on a graph with n nodes, we only need to consider paths with $\le n-1$ edges. For each node x, let $d_k(x)$ denote the shortest of the paths from 0 to x that have $\le k$ edges (∞ in no such path exists).

Then, for $k = 0$, we have $d_0(0) = 0$ and $d_0(x) = \infty$ for all $x \neq 0$.

For $k > 0$, the shortest path of length $\le k$:

- either has the length $\le k-1$, in which case its length is equal to $d_{k-1}(x)$;
- or has length exactly k, in which case it spends $k-1$ edges to get to some node y and then the last edge to get from y to x; in this case, its length is

$$d_{k-1}(y) + w(y \to x).$$

Thus, the length $d_k(x)$ of the shortest path with $\le k$ edges is equal to the smallest of these values:

$$d_k(x) = \min\left(d_{k-1}(x), \min_y (d_{k-1}(y) + w(y \to x))\right).$$

For each k and for each x, we need a linear time to compute this expression (by counting all ys). For each k, we need to repeat this computation for each of n nodes x – which requires $n \cdot n = O(n^2)$ time.

Finally, as we have mentioned, the shortest path is equal to $d_{n-1}(x)$, we need to repeat all computations for all $k = 1, \ldots, n-1$ – so the overall time is $(n-1) \cdot O(n^2) = O(n^3)$. This is thus indeed a polynomial-time algorithm.

Comment. Once we computed the values $d_{n-1}(x)$, we can also find the shortest paths: if the minimum of $d_k(x)$ is attained for $d_{k-1}(y) + w(y \to x)$, this means that the previous step before last should be the rule $y \to x$. Similarly, we can find the best rule leading to y, etc.

A seemingly natural extension of this idea to a more general case. Since the above dynamic programming idea works for the simplest case, it seems reasonable to try it for the general case as well. Specifically, for each variable x, let $d_k(x)$ be the smallest amount of resources that we need to compute x by using $\le k$ rules.

Similarly to the above, for $k = 0$, we have $d_0(0) = 0$ and $d_0(x) = \infty$ for all $x \neq 0$. For $k > 0$, we have

$$d_k(x) = \min\left(d_{k-1}(x), d'_k(x)\right),$$

where

$$d'_k \stackrel{\text{def}}{=}$$

$$\min_{a,\ldots,b \to x} \left(d_{k-1}(a) + \ldots + d_{k-1}(b) + w(a, \ldots, b \to x)\right),$$

and the minimum is taken over all rules that result in x.

The above seemingly natural approach does not always lead to an optimal program synthesis. Let us show that this approach does not always work. Let us assume that we have rules $0 \to a$, $a \to b$, $a \to c$, and $b, c \to d$, all with weight 1. Then,

- for $k = 0$, we have

$$d_0(0) = 0, \quad d_0(a) = d_0(b) = d_0(c) = \infty;$$

- for $k = 1$, we have

$$d_1(0) = 0, \quad d_1(a) = 1, \quad d_1(b) = d_1(c) = d_1(d) = \infty;$$

- for $k = 2$, we have

$$d_1(0) = 0, \; d_1(a) = 1, \; d_1(b) = d_1(c) = 2, \; d_1(d) = \infty;$$

- for $k = 3$, we have

$$d_1(0) = 0, \;\; d_1(a) = 1, \;\; d_1(b) = d_1(c) = 2, \;\; d_1(d) = 5;$$

- after that, the values $d_k(x)$ do not change.

We are thus tended to conclude that the length of the shortest path to d is 5. However, we can compute d by using only 4 resource units:

- first, we spend one unit to measure a, i.e., to use the rule

$$0 \to a;$$

- then, we spend two units to apply the rules

$$a \to b \text{ and } a \to c;$$

- finally, we spend the last (fourth) unit to apply the rule

$$b, c \to d.$$

Analysis of the situation. The reason why the above algorithm over-estimated is that when had inputs b and c, we added their costs – without taking into account that computing b and c has a common part – the part of measuring a. As a result, in our computations, we counted the resources needed to measure a twice:

- once as part of estimating resources needed to compute b, and
- second time as part of estimating resources needed to compute c.

Solution to the problem: a more efficient algorithm. Let us first consider the case when all the rules use either a single symbol or a pair in the left-hand side – i.e., when all the rules have either a form $a \to b$ or a form $a, b \to c$.

In this case, instead of only iteratively estimating the cost of computing each individual *quantity* a (as in the shortest path algorithm), we also iteratively estimate the cost of computing all possible *pairs*. In other words, for each k, in addition to the values $w_k(a)$, we also estimate the smallest cost $d_k(a, b)$ of computing both a and b in k rules. In general, we are computing values $d_k(A)$, where A is either a single variable or a pair of variables.

To be able to perform computations, we need to expand the original set of rules covering variables to rules covering pairs. Specifically: we generate a new rule $A \to x$ if

- either x is already an element of C (in this case, the weight is 0),
- or there is an original rule r of the type $C \to x$ with $C \subseteq A \cup \ldots \cup B$ (i.e., in which available inputs A, ..., B include inputs needed for this rule); the new rule gets the weight $w(r)$.

Similarly, we generate a new rule $A, \ldots, B \to X$ if every element $x \in X$ is:

- either already in the inputs $x \in A \cup \ldots \cup B$,
- or is covered by a rule r_x of the type $C_x \to x$ with $C_x \subseteq A \cup \ldots \cup B$.

The weight of the new rule is then defined as the sum of the weights of these original rules.

Then, we apply the same algorithm to the new nodes A.

Comment. While we increase the number of rules, this increase is still polynomial, so we still get a polynomial-time algorithm.

Example. Let us show that in the above example, the new algorithm lead to the correct estimate $d(d) = 3$. Indeed, with pairs, we now have rules

- $0 \to a$ (of weight 1),
- $a \to \{b, c\}$ (of weight 2), and
- $\{b, c\} \to d$ (of weight 1).

Applying these rules one after another we get the desired shortest path 3.

General case. In general, we may have rule that have three inputs like $a, b, c \to d$, or even $k > 3$ inputs. In this case, we have to consider sets A consisting of 3 (or, correspondingly, k) variables.

Comment. For each k, the resulting algorithm is still polynomial. Since the size is usually limited, we thus have still a polynomial algorithm.

However, it is worth mentioning that the computation time grows as n^k – i.e., exponentially with k. This exponential growth is inevitable, since we are facing a general problem of finding a shortest path in a hyper-graph (= graph with "edges" of the type $a, \ldots, b \to c$), and this problem is known to be NP-hard [3].

Acknowledgments

The author is thankful to Vladik Kreinovich and Steve Roach for valuable discussions.

References

[1] D. E. Cooke, V. Kreinovich, and S. A. Starks, "ALPS: A Logic for Program Synthesis (motivated by fuzzy logic)", *Proceedings of the IEEE International Conference on Fuzzy Systems FUZZ-IEEE'98*, Anchorage, Alaska, May 4–9, 1998, Vol. 1, pp. 779–784.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, MIT Press, Cambridge, Massachusetts, 2009.

[3] G. F. Italiano and U. Nanni, "On line maintenance of minimal directed hypergraphs", In: *Proceedings of the 3rd Convegno Italiano di Informatica Teorica*, Mantova, Italy, World Science Press, 1989, pp. 335–349.

[4] G. Klir and B. Yuan, *Fuzzy Sets and Fuzzy Logic: Theory and Applications*, Upper Saddle River, New Jersey: Prentice Hall, 1995.

[5] H. T. Nguyen and E. A. Walker, *First Course on Fuzzy Logic*, CRC Press, Boca Raton, Florida, 2006.

[6] S. Roach, *TOPS: Theory Operationalization for Program Synthesis*, PhD Thesis, University of Wyoming, August 1997.

[7] S. Roach, "Constructing decision procedures for domain specific deductive synthesis", *Proceedings of the Dagstuhl Seminar 01221 "Can Formal Methods Cope With Software Intensive Systems?"*, International Conference and Research Center for Computer Science, Dagstuhl, Germany, May 2001.

[8] S. Roach, "Logic-based program synthesis: state-of-the-art and future trends", *Proceedings of the 2002 American Association for Artificial Intelligence AAAI Spring Symposium on Logic-Based Program Synthesis*, Stanford University, Palo Alto, California, March 2002.

[9] S. Roach and J. Van Baalen, "Experience report on automated procedure construction for deductive synthesis", *Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, Edinburgh, UK, September 2002, pp. 69–78.

[10] S. Roach and J. Van Baalen, "Automated procedure construction for deductive synthesis", *Journal of Automated Software Engineering*, 2005, Vol. 12, No. 4, pp. 393–414.

[11] S. Roach, J. Van Baalen, and M. Lowry, "Meta-Amphion: scaling up high assurance deductive program synthesis", *Proceedings of the IEEE High Integrity Software Symposium*, Albuquerque, New Mexico, October 15–16, 1997, pp. 81–93.

[12] E. Tyugu, *Knowledge-Based Programming*, Addison-Wesley, Wokingham, England, 1988.

[13] J. Van Baalen and S. Roach, "Using decision procedures to accelerate domain-specific deductive synthesis systems", in P. Flener (ed.), *Proceedings of the 8th International Workshop on Logic Programming Synthesis and Transformation LOPSTR'98*, Manchester, UK, Lecture Notes in Computer Science, Vol. 1559, Springer-Verlag, 1999, pp. 61–70.