Program Synthesis - Toward mathematically generated programs

Mikaël Mayer, Viktor Kuncak

June 5, 2009

Abstract

Code synthesis is very frequently used in industries, to generate low-level languages from higher-level ones like UML, to generate GUI, forms, and database support. The Program Synthesis problem is also some kind of code synthesis, where the high-level language is expressed formally and mathematically. It aims at generating a program and its proof of at least correctness, given a set of formal specifications.

In this paper, our approach to this problem is to first describe the associated mathematical problem, and which subset of program synthesis based on Hoare triples we are able to solve.

Second we will describe all the components of our algorithm, as with the type of specifications we worked on, namely based on integer variables, constants, function and predicate symbols.

Third we will present some examples for which we were able to give a complete solution program, like the 2-INT-MAX, 4-INT-SORT and the 2-INT-GCD inner-loop problem.

1 Introduction

What are the current programming methods ? They depend on the purpose, and on the performance, security or correctness constraints. If one concentrates on the correctness constraints, there are several methods to achieve them.

- Method 1: Write the code, test it manually.
- Method 2: Write the code, write some tests, run the tests, change the code until it works.
- Method 2_{bis}: (Test-driven development) Write first some failing tests, then write only the code to make them work.

Industrially, a lot of companies use test-driven development [10], because it proved to be quite powerful, extensible, and reliable. However some rare cases not covered by tests can happen, and this can be a source of software bug.

Ideally, we would like to have specifications which prevent any bugs from happening, and this can be achieved through mathematical methods.

- Method 3: Write the code, write formal specifications, prove manually or with an automated prover that the code meets the specifications.
- Method 3_{bis}: (Specification-Driven Development) Write first failing formal specifications, then update the code to meet specifications.
- Method 4: (Program Synthesis) Write down formal specifications, and let the machine infer the code which meets specifications [6].

This is the hard general goal of Program Synthesis. It tries to reproduce human thinking when it comes to write program, in a more basic and computing way.

2 Problem statement

The problem is the following:

We have some constrained input, and we are asked to produce an output so that some constraints and properties are satisfied[1].

We assume that the input is on the form of constants typically named $a, b, c \ldots$, and that the requested output is on the form of variables named $x, y, z, a', b' \ldots$.

For example, provided an integer $a \in \mathbb{Z}$, we could be asked to compute its integer square root, i.e. the integer x such that $x^2 \leq a < (x+1)^2$.

As it, this problem formulation is too strong. Indeed, in the previous example, the possibility that $a < 0$ makes it impossible to solve; however, we would like a way to report a solution in the case $a \geq 0$.

2.1 Formal Problem statement

In all generality, it is about to solve the second-order logic problem of finding a function f such that $R(f)$ holds, where R is a predicate over functions, meaning for example "f computes the maximum of its arguments" or "f returns its argument sorted".

We use a simplified but still powerful approach where the predicate R can be expressed with an Hoare triple[4], and the function as a program (see Appendix A).

2.1.1 Conventions and first approach

We decided to adapt the standard convention of the Hoare triple to our needs.

$\{P\}_{[a,b]}\ C\ \{Q\}_{[a,b,x,y]}$ means that

- P, Q are formulas (conjunctions are coma-separated)
- P contains variables a, b (acting as constants).
- Q contains variables a, b, x, y
- C is a program only modifying x and y
- "If P holds, then after the execution of C, Q will hold".

Ex. 1: $\{a = 2\}_{[a]}\ x \leftarrow a + 1\ \{x = 3,\ a \leq 2\}_{[a,x]}$ is a valid Hoare triple.

Ex. 2: $\{a = 2\}_{[a]}\ x \leftarrow a - 1\ \{x \neq 1\}_{[a,x]}$ is not a valid Hoare triple.

For readability reasons, we will sometimes omit the indices containing variables.

We are interested in the *satisfiability* of Hoare triples; that is, given P and Q, find a command C such that $\{P\}\ C\ \{Q\}$ holds; we represent this problem by $\{P\}\ C^?\ \{Q\}$.

This formulation is too strong for the Program Synthesis problem. Indeed,

$$\{\}_{[a]}\ C^?\ \{x^2 \leq a < (x+1)^2\}$$

is not satisfiable, contrary to

$$\{a \geq 0\}_{[a]}\ C^?\ \{x^2 \leq a < (x+1)^2\}$$

.

Mathematically, it corresponds to give a constructive proof to the problem :

$$\forall (a,b).\ P_{[a,b]} \Rightarrow \exists (x,y).\ Q_{[(a,b),(x,y)]}$$

that is, to find a computable function f such that

$$\forall (a,b).\ P_{[a,b]} \Rightarrow Q_{[(a,b),f(a,b)]}$$

2.1.2 Better approach

A better approach is to find R and C in the Hoare triple $\{P,\ R^?\}\ C^?\ \{Q\}$ such that it becomes valid. Of course, it always satisfiable: one can choose $R = \mathit{False}$ and $C = \mathit{Skip}$ and the Hoare triple is obviously valid.

A solution $\{P,\ R_1\}\ C_1\ \{Q\}$ is said to be better than another solution $\{P,\ R_2\}\ C_2\ \{Q\}$ if R_1 is weaker than R_2 [1].

One can quickly remark that there cannot be two independent best solutions. Indeed, if $\{P,\ R_1\}\ C_1\ \{Q\}$ and $\{P,\ R_2\}\ C_2\ \{Q\}$ both hold, then

$$\{P,\ R_1 \vee R_2\}\ \text{if}(R_1)\ \text{then}\ C_1\ \text{else}\ C_2\ \{Q\}$$

is a better solution than both of the previous ones. Even better, if $\{P,\ R\}\ C_1\ \{Q\}$ and $\{P, \neg R\}\ C_2\ \{Q\}$ both hold, then using a rule similar to propositional resolution [3, 2],

$$\{P\}\ \text{if}(R)\ \text{then}\ C_1\ \text{else}\ C_2\ \{Q\}$$

[1] A is weaker than B if the implication $B \Rightarrow A$ holds. *true* is the weakest proposition.

holds, and this is the best solution.

This last step is the main step of our algorithm: If we find a solution $\{P,\ R_1\}\ C_1\ \{Q\}$, we try to solve $\{(P,\ \neg R_1),\ R^?\}\ C^?\ \{Q\}$, and then recombine the solutions.

2.2 Our approach

It is often easier to solve problems if they are in normal form. Therefore we chose the following normal form, where repetitions are expressed with *.

```
PostCondition ::=
  ([Not] SinglePostCondition)*

SinglePostCondition ::=
    F(FTerm*) =  FTerm*)
  | F(FTerm*) =  F(FTerm*)
  | (CTerm*)  =  (0*)
  | (CTerm*) >=  (0*)
  | (CTerm*) >   (0*)
  | P(Var*)
  | Constant '|' Var

CTerm ::= ((K '*' Var)* - K)
FTerm ::= F(FTerm*) | CTerm
Var   ::= OutputVar | InputVar
F     ::= 'Function symbol'
P     ::= 'Predicate symbol'
K     ::= 'Integer'
```

As you can observe, we directly introduced the tuples, so that we can easily write function symbols which return a set of values. The ">" symbol over tuple is interpreted as the lexicographic order over the tuples.

If we have disjunctions, implications, equivalences, we are currently distributing them during the problem solving process.

Function symbols in affine combination like $y + F(a+x) = 1$ are replaced by OutputVars: $\{y+x42 = 1,\ F(a+x) = (x42)\}$.

We do not have support for universally or existentially quantified formulas in the post-conditions. This could be added in the future.

3 The derivation process

"Derivation" has here the same meaning as "program synthesis" or "program generation".

3.1 Problem structures

An "abstract" problem has a list of available preconditions plus axioms, and the post-condition formula.

Typically, problems have other sub-problems and a parent which they interact with.

We are dealing here with 6 kind of problems:

- **NewProblem**: It is the most general problem to solve. It will eventually have a list of sub-problems whose solution is also a solution of the NewProblem.

$$\{R^?\}_{[a,b]}\ C^?\ \{Q\}$$

- **AssignmentProblem**: An assignment-based problem. It is the base of any solution. It contains assignments from Input to Output Variables.

$$\{3 \mid a\}_{[a]}\ x \leftarrow \frac{a}{3}\ \{3x - a = 0\}$$

- **AssignmentAndNewProblem**: A wrapper containing an AssignmentProblem and a NewProblem. If the NewProblem is solved with a certain precondition, it will replace the variables by their values in the precondition before passing it to its parent.

$$\begin{gathered}\{(3 \mid a),\ R_1^?\}_{[a]} \\ \left[\begin{array}{rcc} \{3|a\}_{[a]} & x{\leftarrow}(\frac{a}{3}) & \{3x{=}a\} \\ \{3x{=}a,\ R^?\}_{[a,x]} & C_1^? & \{F(x){=}y\} \end{array}\right] \\ \{F(x) = y, 3x = a\}\end{gathered}$$

- **SkipProblem**: A problem containing an empty command and a weakest precondition. It is useful for problems without solution or with trivial ones.

$$\{a = b\}_{[a,b]}\ \mathit{Skip}\ \{a - b \leq 0\}$$

- **IfThenElseProblem**: An If-Then-Else based problem. It is the base of all problems needing branching logic. It contains a computable condition, and two sub-problems: one for the "if-then" clause, and the other for the "else" clause. It is solved as soon as both subproblems are solved, and preconditions are reported.

$$\begin{array}{c}\text{if}(a > 0)\\ \{a > 0,\ R_1^?\}_{[a]}\ C_1^?\ \{x \geq a,\ x \geq 0\}\\ \text{else}\\ \{a \leq 0,\ R_2^?\}_{[a]}\ C_2^?\ \{x \geq a,\ x \geq 0\}\end{array}$$

- **WhileProblem**: A while-loop based problem. It contains an loop invariant, a loop variant, a guard condition and a subproblem whose solution would give the inner loop of the while-loop Problem. We did not implement this kind of problem in our algorithms, because it would require further research, like invariant generation, etc.
  However we will give afterwards an example of inner-loop problem, that could be contained in a WhileProblem.

3.2 Message exchange

Once a problem has a solution, if it is not the top level problem, it reports itself plus its precondition to its parent.

The parent decides afterwards how to deal with this solution, if it need more solutions as in the IfThenElseProblem, or if one solution is enough as for the NewProblem (see after the main algorithm).

If a problem decides that it is solved, or finished, it notifies all its children to do/be the same, even those which are still looking for a solution.

3.3 Implementation part

We implemented the process of solution-finding in a breadth-first search way, with a "processor" object containing all NewProblems currently looking for a solution, and processing them one by one.

But this is transparent; externally, everything happens as each problem was solving itself.

3.4 Main algorithm

Let us present our complete one-pass algorithm for deriving programs. It is working on the problems that need to be solved, namely NewProblems. The strength and weakness of this algorithm is to generate less complex problems from a given NewProblem.

1. If the format of the post-condition is not suitable for computing, create and start one equivalent NewProblem's with the post-condition transformed to make it suitable. Return.

2. If the post-condition contains disjunctions or "self-expandable" post-conditions like "is a permutation of", expand them, and start all subproblems. Return.

3. If all computable post-conditions are not satisfiable, set precondition as false and mark as solved. Notify parent. Return.
   . . . else

4. $A \leftarrow$ affine post-conditions

5. $A_0 \leftarrow$ Solve A, get assignments from InputVars to OutputVars.

6. If A_0 contains assignments, generate an AssignmentAndNewProblem containing the assignment part (already solved), and the remaining post-conditions in a NewProblem, start it, return.
   . . . else

7. If there are only computable post-conditions containing InputVariables and no OutputVariables, without function or predicate symbols, mark the problem as solved with these post-conditions as preconditions. Return.
   . . . else

8. For each non affine equality post-condition $A = B$, match the members to generate new problems.
   Example: $F(x + a, G(b))) = F(y - b, G(x + a))$ will yield a problem where this post-condition is replaced by $\{x+a = y-b,\ G(b) = G(x+a)\}$

Example: $\{F(a+x) = G(b)\}$ is not processed, neither are $\{F(a+x) = y\}$, nor $\{G(x,y) = F(a)\}$

9. For each non affine post-condition, match it with each available precondition, and start the new problems resulting from such a match.
   Example: With $\{F(a+x) = G(b)\}$, if we have in the preconditions that
   $$\forall u.\ a = 1 \Rightarrow F(b+u) = G(u)$$
   we will generate a new problem where $F(a+x) = G(b)$ is replaced by:
   $$\{a = 1,\ a + x = b + u41,\ b = u41\}$$
   where $u41$ is an instantiation of the axiom variable.

10. For each inequality post-condition $A \neq B$, $A > B$, $A \geq B$, look for a neighbouring solution.
    Example: From $\{x + a < b\}$, it will start the subproblem $\{x + a + 1 = b\}$. From $\{x \neq a\}$, it will start the subproblems $\{x = a + 1\}$ and $\{x = a - 1\}$

Now that we have the main algorithm, each problem needs to handle incoming solutions from its subproblems.

3.4.1 NewProblem gets a solved subproblem

Let us consider the case when a NewProblem N, with preconditions $PreCond$ and a formula $PostCond$ to prove, is notified by one of its children A that A solved $PostCond$, and A provides a precondition K such that

$$K = K_1 \wedge K_2 \ldots \wedge K_n$$

- If K is $true$ or $PreCond \Rightarrow K$, then N reports itself plus precondition $true$ to its parent.
- If K is $false$ or $PreCond \Rightarrow \neg K$
  - If ($N \not\Leftarrow A$ and the list of sub-problems is not empty), just return.
  - Else N reports itself plus precondition $false$ to its parent.

  ...else there are no trivial preconditions.
- N stores A as a potential solution if nothing better, with its precondition.
  N generates an IfThenElseProblem as sub-problem like the following, and starts it.

```
if $K_1$ then
  if $K_2$ then
    ...
    if $K_n$ then
      $A$
    else
      $\{Precond,\ K_1, \ldots \neg K_n\}_{[]}\ C_n^?\ \{PostCond\}$
    end if
    ...
  else
    $\{Precond,\ K_1,\ \neg K_2\}_{[]}\ C_2^?\ \{PostCond\}$
  end if
else
  $\{Precond,\ \neg K_1\}_{[]}\ C_1^?\ \{PostCond\}$
end if
```

In practice, when we generate an IfThenElseProblem, we are currently removing all other problems that were in the N sub-problem list, in order to cut down the complexity. This could be enhanced a lot by having priorities and good heuristics.

3.4.2 AssignmentAndNewProblem gets its NewProblem solved

Let us consider the case when an AssignmentAndNewProblem $S_N = \{S,\ N\}$ gets a report from one of its children. The assignment is denoted S and the NewProblem N.

Assuming that S is already resolved, the report comes from N with a *precondition* K.

S_N computes the weakest precondition of the command of S and the *post-condition* K, and report it to its parent.

Example: If the assignment problem is

$$\{\}_{[a,b]}\ x \leftarrow a\ \{x = a\}$$

and if the NewProblemSolved is the following

$$\{x = a\}_{[a,b,x]}\ Skip\ \{x = a,\ x \geq a,\ x \geq b\}$$

then it will report to its parent to have been solved with the precondition R such that $\{R\}_{[a]}\ x \leftarrow a\ \{x = a,\ x \geq a,\ x \geq b\}$ which is solved with

$$R\ =\ \{a = a,\ a \geq a,\ a \geq b\}$$

3.4.3 IfThenElseProblem gets a subproblem solved

Let us consider the case when an IfThenElseProblem I gets a report from one of its children, either the if-problem IP or the else-problem EP.

First, it stores the precondition. Then if both IP and EP are solved, it reports to its parent to have been solved with the conjunction of the two preconditions.

3.4.4 Affine constraint solver

Our affine constraint solver is able to deal with specific of affine constraints. It needs that at most one OutputVar appear in the equations, so $\{a - b = 1, x + a = b\}$ is solvable, but we did not consider problems like $\{x + y = 1, x - y = a\}$, which would require an extra effort.

The affine constraint solver:

- Generates assignments commands.

  $$\{R^?\}_{[a,b]}\ C^?\ \{-4a + 2x + 2b = 1\}$$

  will yield the exact solution

  $$\{\}_{[a,b]}\ x \leftarrow 2a - b + 1\ \{-4a + 2x + 2b = 1\}$$

- Generates necessary preconditions

  $$\{R^?\}_{[a,b]}\ C^?\ \{x = a,\ x = b\}$$

  will yield the exact solution

  $$\{a = b\}_{[a,b]}\ x \leftarrow a\ \{x = a,\ x = b\}$$

- Generates (not fully yet) divisibility constraints

  $$\{R^?\}_{[a]}\ C^?\ \{2x - a = 0\}$$

  will be solved as :

  $$\{2 \mid a\}_{[a]}\ x \leftarrow a/2\ \{2x - a = 0\}$$

- Does not (yet) solve equations containing more than two output variables: $\{R^?\}\ C^?\ \{x + y = 1\}$ is such an example, not constrained enough for our solver.

4 Examples

4.1 Presburger arithmetic support

In this formalism, we are able to produce solution for a subset of Presburger arithmetic.

In the Presburger arithmetic, checking the validity of a formula can be reduced[5] to check the validity of many formulas like:

$$\exists y. \bigwedge_{i=1}^{L} a_i < y \wedge \bigwedge_{j=1}^{U} y < b_j \wedge \bigwedge_{i=1}^{D} K_i \mid (y + t_i)$$

In our logic, y would be an OutputVar, a_i and b_i would be InputVar, and it would be about to solve the problem:

$$\{R^?\}\ C^?\ \{\bigwedge_{i=1}^{L} a_i < y,\ \bigwedge_{j=1}^{U} y < b_j,\ \bigwedge_{i=1}^{D} K_i \mid (y + t_i)\}$$

Our algorithm does not deal well with divisibility constraints, so let us consider the problem :

$$\{R^?\}_{[a_1,\ldots,b_U]}\ C^?\ \{\bigwedge_{i=1}^{L} a_i < y,\ \bigwedge_{j=1}^{U} y < b_j\}$$

This would be solved by our tool with one possible solution:

$$\{\bigwedge_{i=1}^{L} a_i < a_1+1,\ \bigwedge_{j=1}^{U} a_1+1 < b_j\}\ y \leftarrow a_1+1\ \{\dots\}$$

Knowing that one of the last steps to arrive to the first formula was to introduce y such that $y = Mx$, we would need a further assignment $x \leftarrow \frac{y}{M}$.

This shows that there is still some work to improve our algorithm to produce the best solution from Presburger arithmetic constraints.

4.2 The 2-MAX problem

The statement of this problem is the following:

$$\{R^?\}_{[a,b]}\ C^?\ \{(x = a \vee x = b),\ x \geq a,\ x \geq b\} \quad (1)$$

Let us go through algorithm step by step to find the solution. First, we introduce two new subproblems by converting to a normal form, and splitting the disjunction:

$$\{R^?\}_{[a,b]}\ C^?\ \{x-a=0,\ x-a\geq 0,\ x-b\geq 0\} \quad (2)$$

$$\{R^?\}_{[a,b]}\ C^?\ \{x-b=0,\ x-a\geq 0,\ x-b\geq 0\} \quad (3)$$

Solving the affine equations of problem (2) would yield a new problem (4):

$$\begin{array}{c}\{R^?\}\\ x \leftarrow a\\ \{x=a,\ R^?\}_{[a,b,x]}\ C^?\\ \{x-a=0,\ x-a\geq 0,\ x-b\geq 0\}\end{array} \quad (4)$$

Problem (4) does not contain any more OutputVar, so all the post-conditions should appear in the preconditions.

$$\begin{array}{c}\{R^?\}\\ x \leftarrow a\\ \{x=a,\ (x-a\geq 0,\ x-b\geq 0)\}_{[a,b,x]}\\ \{x-a=0,\ x-a\geq 0,\ x-b\geq 0\}\end{array} \quad (5)$$

To compute the global precondition of (5), we need to replace x by its value a in the generated precondition $x-a \geq 0, x-b \geq 0$. This give a precondition $a-a \geq 0, a-b \geq 0$, which after simplification reduces to $a-b \geq 0$ in (6).

$$\begin{array}{c}\{a-b\geq 0\}\\ x \leftarrow a\\ \{x=a,\ (x-a\geq 0,\ x-b\geq 0)\}_{[a,b,x]}\\ \{x-a=0,\ x-a\geq 0,\ x-b\geq 0\}\end{array} \quad (6)$$

So far, (6) is a solved sub-problem solution of (2) and therefore solution of (1). From the precondition, problem (1) will start a new if-problem (7):

$$\begin{array}{c}\{R^?\}\\ \text{if}(a-b\geq 0)\\ (6)\\ \text{else}\\ \{a-b<0,\ R^?\}_{[a,b]}\ C^?\\ \{(x=a \vee x=b),\ x\geq a,\ x\geq b\}\end{array} \quad (7)$$

By similar means, the else-problem of (7) will be solved by (8), and this time giving the solution $x \leftarrow b$, but as we know that $a < b$, no more condition is generated:

$$\begin{array}{c}\{a-b<0,\ R^?=true\}\\ x \leftarrow b\\ \{x=b,\ (x-a\geq 0,\ x-b\geq 0)\}_{[a,b,x]}\\ \{x-b=0,\ x-a\geq 0,\ x-b\geq 0\}\end{array} \quad (8)$$

Thus, the generated program, without all of its proved formulas content would look like 9:

$$\begin{array}{c}\{true\}\\ \text{if}(a-b\geq 0)\\ x \leftarrow a\\ \text{else}\\ x \leftarrow b\\ \{(x=a \vee x=b), x\geq a, x\geq b\}\end{array} \quad (9)$$

5 Solved Problems

5.1 The n-SORT problem

The n-SORT problem stands for the problem of sorting n integers. It can be expressed formally by this

# integers	# steps to find solution
1	2 steps
2	10 steps
3	62 steps
4	752 steps

Table 1: Steps needed to solve the n-Sort problem

problem, where $i \in [1, n]$

$$\{R^?\}_{[\{a_i\}_i]}\ C^?\ \{\{a_i\}_i = \{x_i\}_i,\ x_1 \leq x_2 \leq \ldots \leq x_n\}$$

Contrary to the general sort problem, it only accept a fixed number of integers. Although one can observe that sorting one or two integers is very easy, it is less trivial for 3 or 4 integers.

As our program is not specialized for this kind of work, this is why we are not getting compelling results compared to specialized approaches [9].

Anyway, our paradigm was strong enough to deal with this problem, and we provide the generated code in Appendix B .

Table 1 presents the number of steps needed to generate n-SORT problems. A step is counted as one pass of the main algorithm and does not include the message exchange, which is always bounded.

We also copy-pasted the code generated for 4-SORT into a scala[7] function to "test" it, although it might look quite useless to test a code which has been generated and proved.

Of course, it gave the expected results:

```
4-SORT: (8,3,5,4) => (3,4,5,8)
4-SORT: (8,5,2,1) => (1,2,5,8)
4-SORT: (3,5,2,1) => (1,2,3,5)
4-SORT: (1,5,2,1) => (1,1,2,5)
4-SORT: (1,5,2,3) => (1,2,3,5)
```

5.2 The GCD inner loop problem

The mathematical formulation of the GCD inner loop problem is the following. The "+" sign in the axioms is not a typography error, the problem is really solved like that.

$$\begin{array}{l}\{\forall u.\forall v.\ GCD(u,v) = GCD(v,u),\\ \forall u.\forall v.\ GCD(u,v) = GCD(u,v+u),\\ c \geq 0,\ d \geq 0,\ GCD(c,d) = GCD(a,b)),\\ c \neq 0, R^?\}_{[c,d,a,b]}\\ C^?\\ \ \{c' \geq 0,\ d' \geq 0,\ GCD(c',d') = GCD(a,b),\\ (c',\ d') < (c,\ d)\}\end{array} \tag{10}$$

This problem contains among others an invariant:

$$\{c \geq 0,\ d \geq 0,\ GCD(c,d) = GCD(a,b)\}$$

a decreasing variant:

$$(c, d)$$

and a guard:

$$c \neq 0$$

If this problem is solved, it can be put in the inner loop of the following algorithm to compute the GCD:

```
c = a
d = b
while(c != 0) {
  // here a solution from c, d to c', d'
  c = c'; d = d'
}
// At this point, d contains gcd(a, b)
```

Our algorithm was able to derive such a code in 61 steps.

```
if(-c+d >= 0) {
  d' = -c+d; c' = c; x49 = c; x48 = d
} else {
  c' = d; d' = c
}
```

One remarks that some unexpected variables appear, due to the instantiation of the axioms. They could be easily removed by refining the rendering process.

6 Conclusions

We were able to automatically synthesize some programs for simple and less simple specifications.

A full program synthesizer would have some terrific applications. First, it would not only tell where the errors are in a program, but also suggest corrections. If we manage to have formal specifications, then it would even not be worth to write the code, the synthesizer might do it.

6.1 Further work

To enhance such a program synthesizer, one could:

- Add specific structures to deal with equalities, rather than matching them.
- Set up priority on problems. More constrained problems should have a greater priority
- Add back-jumping techniques[8] to cut-off some research branches.
- Change the order of the conditions for the if-then-else deployment ? (see section 3.4.1)
- Add support for object-oriented programming
- Add support for other computable functions.

Appendices

A Program commands

For our needs, a program is a command, and it is defined as below:

```
Command ::=
   Var = [Term]
 | While(Formula) { Command }
 | if(Formula) Command else Command
 | Command; Command
 | Skip

Term ::= Var | K
       | k*Var + Term
       | Var/k + Term

Var  :;= [Literal]

Formula ::= Formula && Formula
          | Formula || Formula
          | Term == Term
          | Term > Term
          | ...

K ::= [Integers]
```

B The 3,4-SORT programs

Here is one automatically generated 3-SORT algorithm.

```
// Problem was solved in 62 steps
//{requires nothing else
if(c-b >= 0) {
  if(b-a >= 0) {
    z = c; y = b; x = a
  } else {
    if(c-a >= 0) {
      z = c; y = a; x = b
    } else {
      z = a; y = c; x = b
    }
  }
} else {
  if(c-a >= 0) {
    z = b; y = c; x = a
  } else {
    if(b-a >= 0) {
      z = b; y = a; x = c
    } else {
      z = a; y = b; x = c
    }
  }
}
//}ensures (({x,y,z}={a,b,c}&&(y >= x))&&(z >= y))
```

Here is one automatically generated 4-SORT algorithm.

```
// Problem was solved, in 752 steps
//{requires nothing else
if(d-c >= 0) {
  if(c-b >= 0) {
    if(b-a >= 0) {
      w = d; z = c; y = b; x = a
    } else {
      if(c-a >= 0) {
        w = d; z = c; y = a; x = b
      } else {
        if(d-a >= 0) {
          w = d; z = a; y = c; x = b
        } else {
          w = a; z = d; y = c; x = b
        }
      }
    }
  } else {
    if(d-b >= 0) {
      if(c-a >= 0) {
        w = d; z = b; y = c; x = a
      } else {
        if(b-a >= 0) {
          w = d; z = b; y = a; x = c
        } else {
          if(d-a >= 0) {
            w = d; z = a; y = b; x = c
          } else {
            w = a; z = d; y = b; x = c
          }
        }
      }
    } else {
      if(c-a >= 0) {
        w = b; z = d; y = c; x = a
      } else {
        if(d-a >= 0) {
          w = b; z = d; y = a; x = c
        } else {
          if(b-a >= 0) {
            w = b; z = a; y = d; x = c
          } else {
            w = a; z = b; y = d; x = c
          }
        }
      }
    }
  }
} else {
  if(d-b >= 0) {
    if(b-a >= 0) {
      w = c; z = d; y = b; x = a
    } else {
      if(d-a >= 0) {
        w = c; z = d; y = a; x = b
      } else {
        if(c-a >= 0) {
          w = c; z = a; y = d; x = b
        } else {
          w = a; z = c; y = d; x = b
        }
      }
    }
  } else {
    if(c-b >= 0) {
      if(d-a >= 0) {
        w = c; z = b; y = d; x = a
      } else {
        if(b-a >= 0) {
          w = c; z = b; y = a; x = d
        } else {
          if(c-a >= 0) {
            w = c; z = a; y = b; x = d
          } else {
            w = a; z = c; y = b; x = d
          }
        }
      }
    } else {
      if(d-a >= 0) {
        w = b; z = c; y = d; x = a
      } else {
        if(c-a >= 0) {
          w = b; z = c; y = a; x = d
        } else {
          if(b-a >= 0) {
            w = b; z = a; y = c; x = d
          } else {
            w = a; z = b; y = c; x = d
          }
        }
      }
    }
  }
}
//}ensures ((({x,y,z,w}={a,b,c,d}&&(y >= x))&&(z >= y))&&(w >= z))
```

References

[1] R.G. Dromey and D. Billington. Stepwise program derivation. *School of Computing and Information Technology*, 1991.

[2] Harald Ganzinger. Logic for computer science, 2002. http://www.mpi-inf.mpg.de/~uwe/lehre/logic/.

[3] John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, New York, NY, USA, 2009.

[4] C. A. R. Hoare. *An axiomatic basis for computer programming*. Communications of the ACM, 1969.

[5] V. Kuncak and M. Rinard. The first-order theory of sets with cardinality constraints is decidable. *MIT CSAIL Technical Report*, 2004.

[6] NASA. Robust software engineering, 1994-2008. http://ti.arc.nasa.gov/tech/rse/publications/program-syn/.

[7] M. Odersky and al. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[8] Wikipedia. Backjumping. http://en.wikipedia.org/wiki/Backjumping.

[9] Wikipedia. Comparison sort. http://en.wikipedia.org/wiki/Comparison_sort.

[10] Wikipedia. Test-driven development. http://en.wikipedia.org/wiki/Test-driven_development.