Otto-Friedrich-Universität Bamberg

Cognitive Systems Group

Ausarbeitung

Im Rahmen des Bachelor-Seminars

Thema des Seminars

Zum Thema:

Programming by Analogy

Vorgelegt von:

Hieber, Thomas

Betreuer: Prof. Ute Schmid

Bamberg, WiSe 2008/2009

Abstract

Analogies are used in everyday life to fulfil many purposes. People employ them in the fine arts, politicians use them to polarise or emotionalise, we use them to clarify, imagine or draw comparisons. There is no area in which it cannot serve a purpose. This paper is concerned with analogy in the field of computer programs and how intelligent systems can use analogy to work on given programs thus debugging or even improve them and create new programs in the course. The main focus will be on the practical exercise of the process itself explaining Dershowitz' [2] illustration.

Contents

List of Figures

Listings

1 Introduction

> Sixty years ago, American forces made the same type of sacrifice and helped liberate two continents, and made our world a more peaceful place. The men and women of World War II brought honour to the uniform, and to our flag, and to our country. With each passing day their ranks thin, but the peace they built endures. And we will never let the new enemies of a new century destroy with cowardice what these Americans built with courage.[1]

To start off let us consider this statement of George W. Bush from 2005 addressing his soldiers in 2005. It is a typical way how analogies can be employed to create a connection between two concepts in order to get a certain effect. This may not always be as dramatical as in the speech of a politician at war - but the bottom line is that it is not always advisable to throw around with analogies lightly as has been suggested by Schiller:

> The method of drawing conclusions by analogies is as powerful an aid in history, as everywhere else, but it must be justified by an important purpose, and must be exercised with as much circumspection as judgement. [5]

For our purpose it is enough to memorise this fact and we will now be concerned with analogy from a different perspective.

When we humans try to understand concepts which are not intelligible at the beginning we find it easier to find a similar concept we have already understood. A famous example is the *Rutherford Analogy* - here we have the notion of the solar system transfered and applied to the structure of a hydrogen atom. Even though Niels Bohr proved his theory to be incorrect in certain aspects - it still can be seen as a breakthrough in physics. By using the analogy of the solar system Rutherford mapped concepts like ‘central force‘ or ‘orbit‘ from macro- to microcosm allowing Bohr to elaborate this idea to develop the model of the atomic structure which is still valid today.

This seminar paper will be concerned with the role of analogy in a younger scientific discipline which is - like Rutherford almost 100 years ago - in very early stages. Programing by Analogy is nothing else than applying the same principle as Rutherford to any kind of problem. The important fact here is that it is no longer the human but the computer who - with the help of analogy - should be enabled to invent, debug or improve existing computer programs.

For this purpose it is advisable to shortly clarify the concept of analogy which is going to be the concern of chapter 2. After this the practical side will be in focus by running through an example of how a program can be created in the mentioned manner. Finally, some thoughts on the strength and weakness of this methodology will put the concept into the current scientific perspective.

2 Analogy

Before we walk through the example for analogy in programming the concept of analogy together with its relevance in learning needs to be clarified. Before we make the shift to machines it is quite reasonable to examine how humans use analogy especially since the way machines work with it imitate these processes.

2.1 Analogy Defined

> *Analogy* is (1) similarity in which the same relations hold between different domains or systems; (2) inference that if two things agree in certain respects then they probably agree in others.[...] Analogy is [...] central in the study of LEARNING and discovery.

This definition by the MIT Encyclopedia [4] lays out the initial thoughts on how to begin describing Programming by Analogy. Here the field of discovery is the underlying concept - programming is nothing else but discovering algorithms and functions in order to solve a problem. Thus it must be able to apply the help of analogy in order to relate different problem domains in order to transfer an existing solution to one problem into a new domain.

2.2 Analogy and Learning

Gentner has researched 'Structure Mapping' as 'A Theoretical Framework for Analogy' [3] and he concludes that 'Generative analogies can indeed serve as inferential frameworks'. Related to his work were observations on study subjects formulating statements like 'A cigarette is like a time bomb'. His observations were taken to the next level by Eva Wiese in 2008 [6] who tried to elaborate on the cognitive process of learning by analogy.

In the following chapter we will use this concept of analogy on an abstract level as we will see how one program can be used to solve a problem from another domain using relations and similarities in the problem specification.

3 Programming by Analogy - A Practical Approach

3.1 Program Correction and Debugging

We will begin with an example on real-division which is to be solved programmatically. The programmer has designed a specification (listing 1) which may be the input for a machine. Furthermore the expert has provided an implementation (listing 2) of his specification which - as we will come to find out - is not entirely correct.

Now before we start with the example right away we have to introduce some of the constructs used in the code samples. It is assumed that the reader knows about concepts like conditionals and loops, so they are not going to be included in the following enumeration. Dershowitz describes the less obvious statements as follows:

1. **assert**: contains 'hard facts' like input specification
2. **varying** q : indicates that of the variables in the specification, only q may be set by the program
3. **suggest**: describes the intent of the following code
4. **purpose**: contains the programmer's contention that the preceding code actually achieves the desired relation

First of all let us have a look at the specification (listing 1). The second line gives us the context - the range of quotient q will be between 0 and 1 since $c \leqq d$. The task formulated by **achieve** is to find the quotient of c and d within some fixed tolerance e.

```
D1:  begin comment real-division specification
assert 0 ≦ c < d, e > 0
achieve |c/d − q| < e varying q
end
```

Listing 1: Real Division Specification

According to this specification let us assume a programmer went ahead and produced something like this:

```
T1:  begin comment suggested real-division program
B1:  assert 0 ≦ c < d, e > 0
purpose |c/d − q| < e
purpose q ≦ c/d < q + s, s ≦ e
(q, s) := (0, 1)
loop L1:  suggest q ≦ c/d < q + s
until s ≦ e
purpose q ≦ c/d < q + s, 0 < s < s[L1]
if d ∗ (q + s) ≦ c then q := q + s fi
s := s/2
repeat
suggest q < c/d < q + s, s ≦ e
E1:  suggest |c/d − q| < e
end
```

Listing 2: Real Division Buggy Implementation

The **purpose** statement in line 3 takes the target instruction along with the input definition from the specification (lines 1-3). Label **E1** concludes the program with the programmer stating that the target has been achieved once the control reaches this point. The actual computation takes place between lines 4 and 12 where the body of a loop will be concerned with the calculations. But even before this the programmer has broken down the main goal from line 3 into two subgoals beginning in line 4:

```
purpose q ≦ c/d < q + s, s ≦ e
```

Listing 3: Real Division Subgoal

If these subgoals are satisfied the target relation holds, as enforced by the **suggest** statement in the line before **E1**. In order to achieve the new goals the programmer initialises the values

$q = 0$ and $s = 1$.

Now we come to the loop which is now extracted in order to have a clear view at things:

```
loop L1:  suggest q ≦ c/d < q + s
until s ≦ e
purpose q ≦ c/d < q + s, 0 < s < s[L1]
if d ∗ (q + s) ≦ c then q := q + s fi
s := s/2
repeat
```

Listing 4: Real Division Loop

As we can see from the **suggest** statement at the beginning of the loop it is the intention of the programmer to keep the first subgoal invariant over the loop while converging the second target

$s \rightarrow 0$

which at the same time forms the termination condition for the loop (line 2). The purpose statement in the loop's body formulates the goal of the following lines that is to fulfil the loops **suggest** statement and, in order to finally fulfil the termination condition it ensures that at the end of the loop the value of s must be smaller than the value of s at position **L1** right at the beginning. This is the job of the body-statement in line five, the conditional in line four computes the quotient q.

3.1.1 Program Evaluation

Now the next step is to validate our program in respect to correctness. As a first step we would like to find out about what the program does. For this Dershowitz uses **assert** statements as invariants. The first assertion we can easily make is that s must be less than e by the time we reach label **E1**. Since the termination condition of the preceding loop is exactly $s \leqq e$ this assertion is justified.

Step 1 Now let us consider the assertions we could make at the beginning of the loop. Since we alter the value of s in every iteration by the pattern

$s := S/2$

we can assert at label **L1** that in every iteration s *must* be at least twice as big as e or, in case of the very first iteration $s = 1$. This leads to a first assertion at label **L1**:

$s = 1 \vee 2s > e.$

`if` $d * (q + s) \leqq c$ `then` $q := q + s$ `fi`

Listing 5: Real Division Loop Conditional

Step 2 For the next assertion we first have to have a look at the conditional statement (listing 5) within the loop body. The conditional validates true in case the condition

$d * (q + s) \leqq c$

is met and q is set to $q + s$.

The next assumption is based on a feature of conditionals per se - most of the time they are intended to hold in more than just one case and so we conclude that

$d * q <= c$

holds even though the conditional does not validate true. Note that this suggestion is independent of the value s so we do not cause any harm turning our educated guess into a suggestion on label **L1**. When we look for hard evidence for this guess, we find that it is true right from the beginning, since the initial assertion at label **B1** states that

$0 \leqq c$

and the initialisation of value q evaluates it to *0*. So when the conditional test fails, q

is not altered and this justifies not only our suggestion, it also turns it into an invariant which we can add to our assertion at label **L1**.

Step 3 In case the *then* path is not taken, the condition

$c < d * (q + s)$ holds.

Additionally s is divided by two, q remains the same - so we can alter our new condition to

$c < d * (q + 2s)$.

When we look at our program outside of the loop, we find that the following statements hold:

- $c < d$
- $d > 0$
- $q > 0$ (initialised with 0 and never decreased in value)
- $s > 0$ (initialised with 1 and then constantly divided by two)

This leads to the assertion, that

$c < d * (q + 2s)$

is valid before the loop is entered no matter what happens in the conditional. At some point it might be different but this does not matter to the overall output of the loop. Furthermore, the values q and s will never be less than zero, which means that even after the loop has finished, our condition still holds. Therefore we can add it as a new invariant to our **assert** statement at label **L2**, which now, combining step 1, step 2 and step 3 reads as follows:

`L1:` `assert` $d * q \leqq c, c < d * (q + 2s), s = 1 \vee 2s > e$

Listing 6: Assert at L1

It should have become clear that

$d * q <= c$ and $c < d * (q + 2s)$

hold even after the loop has terminated, so together with the termination condition (which has now also become true) they implicate the following:

$|c/d - q| < 2e$

Listing 7: Assert at E1

This is now added as final assertion at the end of the program:

```
T1:  begin comment suggested real-division program
B1:  assert 0 ≦ c < d, e > 0
purpose |c/d − q| < e
purpose q ≦ c/d < q + s, s ≦ e
(q, s) := (0, 1)
loop L1:  assert d ∗ q ≦ c, c < d ∗ (q + 2s), s = 1 ∨ 2s > e
suggest c/d < q + s
until s ≦ e
purpose q ≦ c/d < q + s, 0 < s < s[L1]
if d ∗ (q + s) ≦ c then q := q + sfi
s := s/2
repeat
assert q < c/d < q + 2s, s ≦ e
suggest c/d − q + s
E1:  assert |c/d − q| < 2e
suggest |c/d − q| < e
end
```

Listing 8: Annotated Buggy Implementation of Real Division

As you can see from the last two lines, the **assert** statement at label **E1** does not correspond to the suggested output one line below. This means we have found out, that our program is buggy.

3.1.2 Debugging

Now we will be going to use analogy for the first time since we need to correct an erroneous program output to meet our original specification goal, which reads

suggest $|c/d - q| < e$.

In conclusion, we have to find an analogy between the output of our buggy program and the desired target output:

$|c/d - q| < 2e => |c/d - q| < e$

It is very obvious to see the difference here, so we can shrink the current analogy to the important part:

$2e \Rightarrow e$

In order to reach the desired output, we have to find a substitute for e such that our transformation holds in the whole program. This is very straightforward, we simply transform

$e \Rightarrow e/2$

and apply this transformation to the program and get the following output:

$(q, s) := (0, 1)$
`loop L2:` `assert` $d * q <= c, c < d * (q + 2s), s = 1 \vee 2s > e$
`until` $s <= e/2$
`purpose` $q <= c/d < q + 2s, 0 < s < s[L2]$
`if` $d * (q + s) <= c$ `then` $q := q + s$ `fi`
$s := s/2$
`repeat`
`assert` $q < c/d < q + 2s, 2s <= e$

Listing 9: Transformed Buggy Implementation of Real Division

Note the change in the purpose statement in line 4 which now correctly describes what happens within the loop body ($s := s/2$), the assert statement in line 8 does likewise, since now

$2s \leqq e$.

But this means also, that the program now differs from what the programmer suggested, especially the **purpose** statement has changed. This can be mended by applying

$s \Rightarrow s/2$

and so the loops termination condition becomes

$s/2 \leqq e/2 = s \leqq e$,

the conditional

if $d * (q + s/2) \leqq c$ then $q := q + s/2$ fi.

The assignment becomes a bit tricky, since we cannot apply a transformation on an expression, which is what appears on the left hand side of

$s =: 1$.

Here Dershowitz applies a trick, saying that by this assignment we mean

achieve $s/2 = 1$ varying s

which resolves to

achieve $s = 2$ varying $s \Rightarrow s =: 2$.

The same has to be done with the assignment inside the loop. Here the goal

achieve $s/2 = (s'/2)/2$ varying s (s' being the value of variable s before. the goal)

resolves to

$s := s/2$.

When we now take a look at the loop body as it is at the moment we can apply a simple facelift by reversing the order of the statements

```
if d * (q + s/2) ≦ c then q := q + s/2 fi
s := s/2
```

Listing 10: Loop Body

becomes to

```
s := s/2
if d * (q + s) ≦ c then q := q + s fi
```

Listing 11: Loop Body Reversed

Having applied all these transformations to the buggy program, we have successfully corrected it into this:

```
D2:  begin comment real-division program
B2:  assert 0 ≦ c < d, e > 0
purpose |c/d − q| < e
purpose q ≦ c/d < q + s, s ≦ e
(q, s) := (0, 2)
loop L2:  assert d * q ≦ c, c < d * (q + 2s), s = 2 ∨ 2s > e
until s <= e
purpose q ≦ c/d < q + s, 0 < s < s[L1]
s := s/2
if d * (q + s) ≦ c then q := q + s fi
repeat
assert q < c/d < q + 2s, s ≦ e
E2:  assert |c/d − q| < e
end
```

Listing 12: Correct Implementation of Real Division

When we compare this final version to our initial one, we can see that they are very much alike apart from the different initialisation of s and a switched order of the two loop body statements.

We have successfully used an analogy between an incorrect program and a correct specification target to generate a valid working program. This is the first way analogy can be used for automated programming - in order to refer back to the meaning of analogy we might say that we have replaced one erroneous piece from our original domain (buggy program) with the help of comparing it to another domain in which we assumed the correct solution to be. This has led us into the new domain where we now correctly calculate real division.

3.2 Program Inference by Analogy

Dershowitz next step is to infer the calculation of cube-roots from the just completed real-division. In order to get to the more relevant points of his example this will only be summarised very shortly.

```
C3:  begin comment cuberoot specification
assert a ≧ 0, e > 0
achieve |a^{1/3} − r| < evarying r
end
```

Listing 13: Cube Root Specification

From the specification in listing 13 an analogy to *assert* | *c / d - q* | < *e* is sought and found in:

- $q \Rightarrow r$
- $c/d \Rightarrow a^{1/3}$

This translates into these transformations:

- $q \Rightarrow r$
- $u/v \Rightarrow u^{1/3}$
- $c \Rightarrow a$

These transformations are applied to the real-division code sample and finally result in a correct (listing 16) program example: [1]

```
C3:  begin comment cube-root program
B3 assert a > 0, e > 0
(r, s) := (0, a + 1)
loop L3:  assert r ≦ a^{1/3} < r + s
until s ≦ e
s := s/2
if (r + s)^3 ≦  a then r := r + s fi
repeat
E1:  assert |a^{1/3} − r| < e
end
```

Listing 14: Cube Root Implementation

3.2.1 A Little More Complicated

The next example we have to consider is a little more complex, but all the more important since the obvious analogy between real-division and cube-root is not a very representative example. If one should like to infer array-search from the cube-root program for instance, he would run into a lot of problematic issues and so this seems like the perfect example to take the process we have just run through with an easy example to the next level.

[1] read more in the paper [2] (pp. 402-404)

The new specification would look like this:

```
A4:  begin comment array-search specification
assert u <= v ⊃ A[u] <= A[v], A[u] ∈ A
achieve A[z] = b varying z
end
```

Listing 15: Array Search Specification

Here we have an array of integers in ascending order and further the most important ground fact that the number we are looking for is contained within the array. As the array indexes may be real numbers, we define a convention to access the index like this:

$A[u] = A[\lfloor u \rfloor]$

Currently we are facing the problem that our new goal does not fit the output specification of the cube-root program (listing 16), so we have to play around with what we have in order to assimilate the structure of output and new goal. We begin with our new goal which can be reformulated into:

achieve $A[z] \leqq b \leqq A[z]$ varying z.

This means in other words, that we are only interested in solutions, which establish an equality in this inequation. For our purposes we will have to add one more thing, which seems to come out of left field but you will be able to understand the purpose. Since this is an inequation of integers we may add random numbers at any 'bigger' side, so lets just do this here:

achieve $A[z] \leqq b \leqq A[z] + 1$ varying z.

Since we know an output invariant of the cube-root program 16 to hold, we can take it and use it as our analogy with a little twist:

assert $r \leqq a^{1/3} < r + e$ varying z.

Note that we have replaced s by e which we can do in this inequation since $e > s$. Now that our current output and the new goal have been defined, we can establish our analog transformation like this:

$r \leqq a^{1/3} < r + e \Rightarrow A[z] \leqq b \leqq A[z] + 1$

The necessary single transformations are now easily compiled - this works as we have seen before:

- $r \Rightarrow A[z]$
- $a \Rightarrow b^3$
- $e \Rightarrow 1$

Applying these transformations to our cube-root program (listing 16) leaves us with the following new program:

```
X4:  begin comment proposed array-search program
B4 assert b^3 ≧ 0, 1 > 0
(A[z], s) := (1, b^3 − 1)
loop L4:  assert A[z] ≦ b < A[z] + s
until s ≦ 1
s := s/2
if (A[z] + s)^3 ≦  b^3 then A[z] := A[z] + s fi
repeat
E1:  assert A[z] ≦ b ≦ A[z] + 1
end
```

Listing 16: Array Search First Implementation

At this point we find a serious problem in the *then* part of the conditional, for one since z, not the array itself is an output variable, so it cannot be altered by the program. Furthermore can there be no warranty that we can assign this variable even if we tried to. The following reformulation of the assignment follows the one used debugging our initial real-division program (chapter 3.1.2).

achieve $A[z] = A[z'] + s$ varying z

How can we know for sure that z' is in the array? Since we cannot, there has to be another way formulating our goal. Now we have to remember the way, array indexes can be accessed which allows us to alter the current goal.

achieve $A[\lfloor z \rfloor] = b$ varying z

Dershowitz now uses an inversion to replace this indexing function by a function *pos(U,u)* which calculates the position of u in U. This new formula now reverses the current goal, but the goal remains the same:

achieve $pos(A, b) = \lfloor z \rfloor$ varying z

This still means that we are looking for the position of b, we have just found a way to express our goal a little differently. Of course the introduced function is only used temporarily and will not be in the final program. For now we can take this part and insert it into our goal before we applied the transformations earlier and so we get the new modified goal together with our old output invariant from cube-root:

goal: achieve $pos(A, b) \leqq z < pos(A, b) + 1$ varying z

output: achieve $r \leqq a^{1/3} < r + e$

When we now compare this new goal to the cube-root output invariant we find another issue which forces us to do some more modification. For now let us just consider the first part of them, namely the $\leqq$ relation:

$pos(A, b) \leqq z$ and $r \leqq a^{1/3}$

The crucial difference between them is the position of the output variables which are interchanged. One quite obvious solution to this problem would be to transform the relation operators. Still, in the first part of the equations we would like to transform

$r \leqq a^{1/3} \Rightarrow z \geqq pos(A, b)$

We can achieve this by using:

- $r \Rightarrow z$
- $\leqq \Rightarrow \geqq$
- $a^{1/3} \Rightarrow pos(A, b)$

When we now apply these transformations to the remaining part of our **output** we get a second set of equations to match:

$pos(A, b) < z + e \Rightarrow pos(A, b) + 1 > z$

After some transformation to have the function *pos(A,b)* standing alone we get

$pos(A, b) < z + e \Rightarrow pos(A, b) > z - 1$

We now add two more transformations in order to have the complete terms matched:

- $< \Rightarrow >$
- $e \Rightarrow -1$

These transformations are now applied to our cube-root program resulting in a new, simplified suggestion.

$(A[z], s) := (1, pos(A, b)^3 + 1)$
`loop L5:` `suggest` $z \geqq pos(A, b) > z + s$
`until` $s \geqq -1$
$s := s/2$
`if` $z + s \geqq pos(A, b)$ `then` $z := z + s$ `fi`
`repeat`
`E5:` `suggest` $z \geqq pos(A, b) \geqq z - 1$
`end`

Listing 17: Array Search Second Implementation

Note that we can have no **assert** statements since we have transformed relational operators which might cause parts of the program not to work as intended. So we are going to validate what we have so far and in conclusion we will take the function *pos* out of the program. If we rip the **suggest** statement at label **E5** apart, we can write it like this:

$pos(A, b) \leqq z, z - 1 < pos(A, b)$

and we can transform it into

$pos(A, b) \leqq z, z < pos(A, b) + 1$

which is exactly our desired goal **goal**, which is not affected by either part of the loop condition. A little more problematic are the loop invariants

$z \geqq pos(A, b)$ and $pos(A, b) > z + s$

which are not instantiated, since the function *pos* is a replacement for the position of b within A. In order to have our invariants initialised so they hold at the time when the loop is entered, we have to introduce a new subgoal:

achieve $z \geqq pos(A, b), pos(A, b) > z + s$ varying z,s

We know from the beginning that $b \in A$, which means

$b \in A[1], ..., A[n]$.

Because of this fact we can fulfil the first invariant by

$z = n$.

At the same time the second invariant will hold as soon as

$z + s = 0$,

which in combination with the assignment of z results in

$s = -z = -n$.

Before we ultimately replace the instantiation we have to check if the termination condition will still hold after that. For this we only have to check if

$(-n/2)* \geqq -1$,

which will certainly be the case, eventually.

The one thing left to do is to replace the occurrences of the function *pos*. For this let us again have a look at the meaning of the **achieve** statement formulated before:

$z + s < pos(A, b) \leqq z$

This invariant expresses the fact that we are trying to approximate to our searched element b from the left. $z+s$ becomes gradually smaller in value as s converges to -1. But we are trying to establish an equation between the two positions $A[z + s]$ and $A[z]$, so in order to get this, we have to add 1 to the indexing on the left side:

$A[z + s + 1]$

As soon as $s \geqq -1$ (termination of the loop) this becomes:

$A[z]$

and we have successfully established our equation.

The second part of the **achieve** statement enforces the position of $A[z]$ to be lower than the one to its right in the array. Since we are not approximating from the right side, we have to mind that this always has to hold - so we just add 1 to the position we are looking for:

$A[z + 1]$

This results in the loop invariant

assert $A[z+s+1] \leqq b < A[z+1]$

The conditional test within the loop checks if the left side of our position of b is actually bigger, which has to be denied. The statement accordingly looks like this:

if $A[z+s+1] > b$ then $z := z+s$ fi

This means that the conditional takes care that our first loop invariant will hold after every loop iteration.

Now we can replace the function *pos* accordingly and get the following final solution to the array-search problem:

```
A5:  begin comment array-search program
assert u <= v ⊃ A[u] <= A[v], A[u] ∈ A
(z, s) := (n, −n)
loop L5:  assert A[z + s + 1] ≦ b ≦ A[z + 1]
until s ≧ −1
s := s/2
if A[z + s + 1] > b then z := z + s fi
repeat
E5:  assert A[z] = b
end
```

Listing 18: Array Search Correct Implementation

3.2.2 The Next Level

Let us now summarise what we have done so far - we have found programs to solve these three problems:

- real division (D2 - listing 12)
- cube root calculation (C3 - listing 16)
- array search (A5 - listing 18)

Taking a closer look at the way all three of them work reveals a surprising feature: they all use *binary-search*. In every loop body we find the value of s divided by 2, which is exactly what *binary-search* does - splitting the problem space in two. Now wouldn't it be nice if we could extract this technique in order to develop a schema which uses this technique in an unspecific context? As this sounds quite compelling let us have a look at what we know about *D2* and *C3*:

- $q \Leftrightarrow r$
- $u/v \Leftrightarrow u^{1/3}$

- $c \Leftrightarrow a$
- $u * v \Leftrightarrow v^3$

We are now going to abstract the variables/functions in these analogies. Both q and r are output variables, so we can abstract them under a new variable z. We can do the same for c and a which both are inputs. The other two transformations are both binary functions, so we can introduce abstract functions for them which we call γ and δ. In conclusion, when we apply the abstraction to our *D2*, we get:

- $q \Rightarrow z$
- $u/ \Rightarrow \gamma(u, v)$
- $c \Rightarrow x$
- $u * v \Rightarrow \delta(u, v)$

We must now apply these rules to the specification (listing 1) which after that reads:

achieve $|\gamma(x, d) - z| < e$ varying z

Since we are now only interested in a *schema*, not a program, let this **achieve** statement be the output specification for it. Before we continue it has to be said, that even though we are looking for a general schema we cannot be very general here, since γ and δ are multiplication and division functions in *D2*, so we cannot instantiate them randomly as the procedures of *D2* are based on multiplication and division only. This is why we have to specify the conditions under which our new schema corresponds to the specified output. The loop invariant translated according to our current transformations would look like this:

$\delta(d, z) \leqq x, x < \delta(d, z + s)$

If we initialised this with the statement from the program $(z, s) := (0, 2)$, we would get:

$\delta(d, 0) \leqq x, x < \delta(d, 2)$

This may be correct for δ being multiplication, but this will not always be the case in a more general schema. So we have to find a way to influence a part of the inequation in order to balance it out if necessary. We can do this by simply changing s to a variable which may be set by the program. Then we have to replace the initialisation statement by a new subgoal:

achieve $\delta(d, z) \leqq x, x < \delta(d, z + s)$ varying z,s.

By this we have left the initialisation of the loop invariant unspecified meaning we will have to ensure that it holds nevertheless. Since the original program employed the reciprocity of multiplication and division, we have to ensure that whatever happens within the loop body, the relation between the functions δ and γ remains the same. If we define this relation by matters of equivalence, we might put it this way:

$\delta(w, u) \leqq v \equiv u \leqq \gamma(v, w)$

This has to be put before the new subgoal for the loop is defined and after that we have ensured to maintain the relation between the functions and successfully derived a schema for *binary-search* for $\gamma(x, d)$:

```
S6:  begin comment binary-search schema
B6:  assert e > 0, δ(w,u) ≦ v ≡ u ≦ γ(v,w)
achieve δ(d,z) ≦ x, x < δ(d,z+s) varying z,s.
loop L6:  assert δ(d,z) ≦ x, x < δ(d,z+s)
until s <= e
s := s/2
if δ(d,z+s) ≦ x then z := z+s fi
repeat
E2:  assert |γ(x,d) − z| < e
end
```

Listing 19: Binary Search Schema Abstracted from Real Division

To wrap all the steps up at one glance, take a look at figure 1

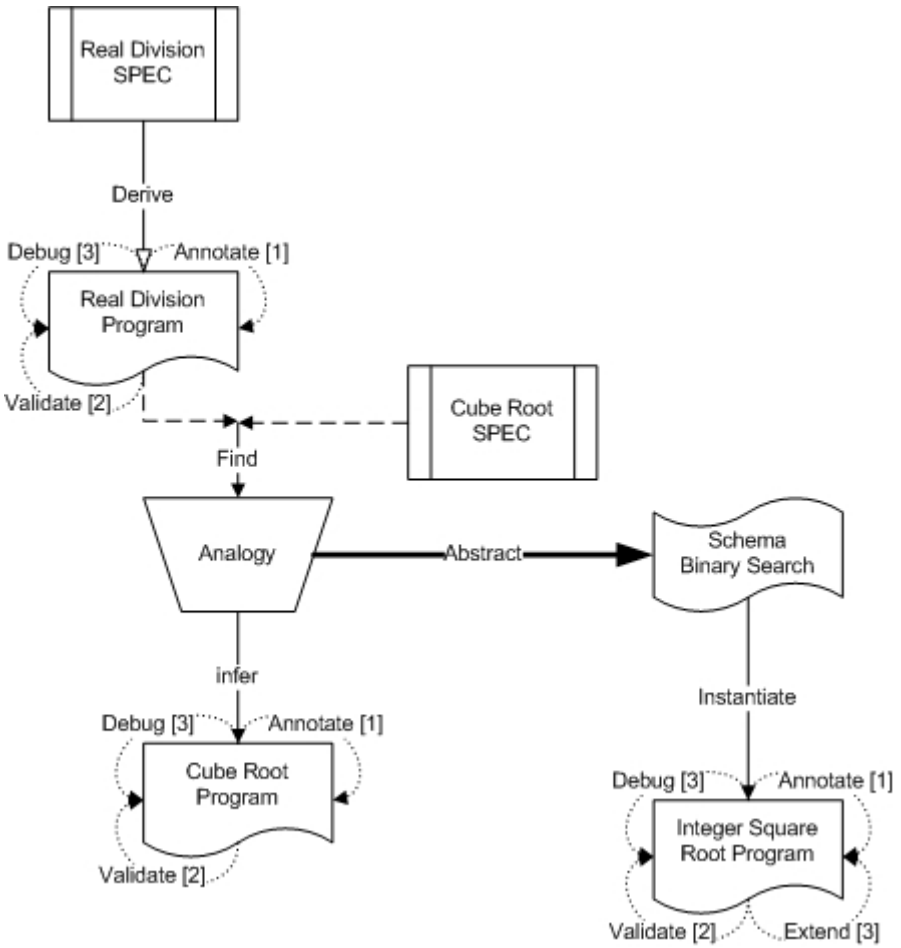

Figure 1: Schematic Impression of the Running Example

3.2.3 Further Reading

The remaining steps in the Dershowitz paper are **instantiation** and **extension** [2]. These shall be quickly summarised before we begin to draw a conclusion on the subject as we have come to see it.

Instantiation takes the derived schema of *binary-search* and tries to find an analogy to a new problem specification (here *integer square root*) and infer a working program which also uses *binary-search* as core technique.

Extension is a necessary discipline as soon as a suggested program still doesn't solve a problem after all possible transformations have been applied. This means, that now the algorithm of the program itself has to be modified, in Dershowitz' example it is the insertion of a second loop.

[2] read more in the paper [2] (pp. 410 - 415)

4 Analogy - Analysis

As we have seen, analogy in programming is absolutely possible and it may even be able to produce novel or unexpected outputs. Remember the final program for array-search which is not a quite obvious way to solve this problem. In fact, if there would exist a computer system which imitated these operations as we have carried them out before the scenario of benefits seems endless. New, more efficient algorithms for existing problems or for such that still haven't been solved. Even better, imagine the superiority over state-of-the-art machine learning systems: Since the user would be able to provide a set of programs to provide analogies, the search space may be restricted very efficiently.

At the other hand - how do we know, which analogies to provide for something we don't know a lot about? After going through Dershowitz' example you might agree with this thesis in general - but you will nevertheless have noticed the drawbacks:

There are passages along the way, which are hard to schematize for a machine like the redefinition of the array-indexing function along with the necessary modifications to the specification. This would require large background knowledge as well as complex problem-solving algorithms along the way. This is for sure a cause of slowdowns in the calculation. Another, even more problematic issue is formulated by Dershowitz himself - the *hidden analogy*. This means that even though specifications might differ on the symbolic level, they may still contain analogies. This is quite hard for a machine to detect, unless the user rewrote the specifications accordingly. The same problem arises vice versa - let the programs be symbolically analog, without being analog semantically.

While other parts like finding and applying transformations, as well as validation seem already to be tailored for machines, it is still problematic to take the concept of Programming by Analogy as suggested by Dershowitz and develop a working system. But, as he himself suggested - it might be valuable in combination with other techniques which has yet to be proven.

References

[1] George W. Bush. President Commemorates 60th Anniversary of V-J Day, August 2005. `http://www.whitehouse.gov/news/releases/2005/08/20050830-1.html`.

[2] Nachum Dershowitz. Programming by Analogy. *Machine Learning*, 2:393–421, 1986.

[3] Dedre Gentner. Structure-Mapping: A Theoretical Framework for Analogy. *Cognitive Science*, 7(2):155–170, Apr-Jun 1983.

[4] Robert A. Wilson; Frank C. Keil. *The MIT Encyclopedia of the Cognitive Sciences*. MIT Press, 2001.

[5] Friedrich Schiller. *Sämtliche Werke, Band 4*. DTV, 2004.

[6] Eva Wiese; Uwe Konerding; Ute Schmid. Mapping and inference in analogical problem solving – As much as needed or as much as possible? In *Proceedings of the 30th Annual Conference of the Cognitive Science Sociecty*, pages 927–932. Mahwah, NJ : Lawrence Erlbaum, 2008.