Interactive Interface Programming by Example and Constraint-Based Synthesis.

Mikaël Mayer
I&C, EPFL

Abstract—To produce software, the best modern programming paradigms aim at providing testing frameworks, verification systems and code synthesis based on specifications. Nevertheless, programming by example or constraints is still a significantly more user-friendly, but also more challenging way of producing programs. This paper first shows the potential of such systems for programs and string processing through three research papers. Subsequently, we present the research proposal, which is about exploring similar systems in graphical environments. We already successfully developed a game engine for Android, which can be on-the-fly programmed by providing input and output examples.

Index Terms—Program synthesis, Programming by example, Software verification, Interactive programming, EDIC, EPFL

I. INTRODUCTION

PROGRAMMING by example consists in describing programs through their intended behavior rather than their implementation. Closely related to constraint programming [1], it helps the user to communicate her needs in terms of input and output expectations.

Modern ways of formally proving the correctness of programs include verification and synthesis systems based on formal specifications. Type-checking systems also help to generate reliable code, and assist the programmer [2]. All these systems have the benefit to add meta-programming structures to programs, such as pre-conditions, invariants and post-conditions. Such predicates provide a verifiable meaning to the program.

Sometimes, the meaning is not fully specified or entirely determined. Programming by example allows users to provide more examples and counter-examples of what they need. Considering this, debugging is simply another refinement step, which consists in providing more examples about how the behavior should be, and retrieving the causal contradictions. Asking why or why not about program behaviors is indeed an example of efficient debugging[3].

We would like to improve the role of the computer to suggest and help the user to express her needs. This is a tricky task because it implies to structure the knowledge so that the computer can learn by inductive reasoning and generalization.

While current researches aim at providing an easier access to program functionalities [4], programming environments still dictate what the user will try and achieve. Building easy-to-use frameworks is challenging because of the simplicity and the incompleteness of meaning representations versus the intended complex behavior of the system.

Therefore, what is the efficiency of programming by example? What are the limits and domains for which the computer can still learn? Furthermore, which interfaces provide the best communication between the user’s intentions and the machine?

We predict that, in the future, most programming environment will be graphical-oriented, with a minimal but relevant user interaction. Therefore, we will now review three important papers in this field. The first explains how to retrieve programs from explicit trace examples. The second explains how to retrieve regular sets from explicit examples and counter-examples, and the third is a recent work about finding string processing programs from very few input and output examples.

II. LEARNING PROGRAMS FROM TRACES

Learning Programs from Traces using Version Space Algebra (T. Lau, 2003) [5]

There are many tasks in programming that programmers themselves would like to automate. Because procedural languages are widely used, this paper describes how to retrieve the code of a “hidden” procedural program based on the visible modification of its environment during its execution.

As a result, by using examples and specifications provided by a human user, this algorithm was used to synthesize
Repetitive text-editing actions.

<table>
<thead>
<tr>
<th>Type of examples provided</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial, intermediate, and final states of integer variables, program step identification.</td>
<td>A single-loop program with a fixed number of statements matching the precise trace.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Trace 1</th>
<th>Trace 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

A program continuing the same actions on the remaining data.

```plaintext
<table>
<thead>
<tr>
<th>&lt;HTML&gt;</th>
<th>$&lt;HTML&gt;</th>
<th>$&lt;HTML&gt;</th>
<th>$&lt;HTML&gt;</th>
<th>$&lt;HTML&gt;</th>
<th>$&lt;HTML&gt;</th>
<th>$&lt;HTML&gt;</th>
<th>$&lt;HTML&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:While</td>
<td>a</td>
<td>&gt;</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2:</td>
<td>a</td>
<td>=</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:</td>
<td>b</td>
<td>+=</td>
<td>a%2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Consider the version space $ConstInt$ consisting of the set of constant functions $\lambda x.C$ for every integer $C$. When it accepts the first input/output example $0 \rightarrow 5$, this version space collapses to the unique function $\lambda x.5$. If it accepts the other example $1 \rightarrow 6$, it collapses to an empty set.

Consider the version space $AddInt$ consisting of the set of adding functions $\lambda x.x + C$. Applied to the first example, it yields the unique function $\lambda x.x + 5$, which is then compatible with the second example.

A version space algebra $\langle V, \bigcup, \Join, \langle, \rangle \rangle$ allows the composition of such typed version spaces with the following algebraic operations:

**Union:** $\bigcup$

The functions of the union version space are the functions belonging to at least one of the two spaces. In our previous examples, the union version space $ConstInt \bigcup AddInt$ would succeed in finding a function consistent with the previous two examples because such a function can be found in $AddInt$.

**Join:** $\Join$

The functions of a join version space are formed by a combination on the cross-product of two versions spaces. For example, if variables $i$ and $j$ are modified sequentially, a program can be described as the join of the two corresponding version space assignments $f$ and $g$: “$i = f(i,j,\ldots)$” $\Join$ “$j = g(i,j,\ldots)$”.

**Transform:**

The functions of a transform version space are formed by a lazy transformation of the functions belonging a version space. For example, a triplet version space $(\text{Condition}) \Join (\text{ConstInt}) \Join (\text{ConstInt})$ can be transformed to a single statement using the function $(c,t,f) \Rightarrow if(c) \text{ goto } t \text{ else goto } f$.

The transformation is lazy, which means that it is done when the functions are retrieved.

To infer the structure of the program from the examples, Lau first uses the following version space for simple programs. An implicit $\bigcup$ union version space is defined for multiple definitions.
The program version space

\[
\langle \text{Program} \rangle ::= \langle \text{Statement} \rangle \cdot \cdot \cdot \langle \text{Program} \rangle
\]

The program version space

\[
\langle \text{Statement} \rangle ::= \langle \text{PrimitiveStatement} \rangle
\]

Primitive statement modifying variables

\[
\langle \text{Statement} \rangle ::= \text{If}(\langle \text{Condition} \rangle) \cdot \cdot \cdot \langle \text{ConstInt} \rangle \cdot \cdot \cdot \langle \text{ConstInt} \rangle
\]

A branching condition with two sub-programs addresses

\[
\langle \text{Statement} \rangle ::= \text{While}(\langle \text{Condition} \rangle) \cdot \cdot \cdot \langle \text{ConstInt} \rangle \cdot \cdot \cdot \langle \text{ConstInt} \rangle
\]

A branching loop with two sub-programs addresses

Primitive statements are also unions and joins over expressions using addition, modulo, array variables and so on. Conditions are expressed under a comparison between a variable and another.

Once this version space is defined, because it can generate any program, it can “parse” any trace to finally rebuild a program equivalent to the original one.

Learning algorithm

A learning algorithm incrementally builds and refines a program or a representation of it as the user provides examples. The more the user provides examples, the more likely the learning algorithm will produce the correct program.

The learning algorithm takes a example of a trace, and for every trace step, it uses it to reduce the version space of the corresponding line. Lau shows that this process converges quickly, because a lot of information can be extracted from the trace. For example, knowing that two non-consecutive statements are executed consecutively helps the algorithm to decide that there is a control flow. Depending on the variable state, the version space will generate suitable conditions. If multiple statements are possible at the end of the algorithm, the most probable one is selected. For loops of unknown length, the learning algorithm tries versions spaces of increasing size until it finds a correlation with the trace.

Results and discussion

On average, their system requires 5.1 trace examples to reach 100% accuracy. For their text-based scenarios where the trace is generated by a user, the algorithm needs from 0.8 to 18 iterations to produce a generalizing program. These results are quite impressive, because it first shows that the more the programs are executed, the more the algorithm can reconstruct them. Second, such algorithms provides a higher potential than macros-recording for text actions. This opens the doors to clever suggesters and by-example macros systems. Lau also mentioned the possibility of increasing the version space up to the point that there would be as few information as only input and output values.

However, there are still some challenges that Lau discovered and where more research can be performed. First, the exhaustive generation is fortunately bounded by the decomposition of the problem in sub-problems. Typical programs have six lines of code. As Lau says, if fewer information is given or if the requested algorithm is greater, this remains an open problem.

Second, the implementation behind the version space algebra is not clearly explained. It lacks some formal definitions that would help to generalize it to other classes of programs. We present page 8 an example of such version space implementation that could be more easily generated.

Third, this algorithm lacks the possibility of interactive example-providing mechanism. Because all the provided examples are mostly randomly generated, such learning algorithm are useful more in theory than in practice. Having a way to question the synthesizer with specific inputs and outputs, or having the synthesizer to ask for feedback would be easier for users, rather than only using whole random traces.

This leads us to the second part of this paper review, which presents a comparatively older but still useful method to create automata with the help of interactive example providing.

III. Learning Regular Sets from Queries


This older article by Angluin presents how to interactively recover an regular set. The user provides a “teacher” which can tell if a string is accepted or not, and distinguishing examples or counter-examples for another regular set if both do not describe the same set. The interesting part relies in the fact that the generation is optimal, i.e. there cannot be a smaller automaton recognizing the regular set than the generated one. A description of this task can be found in Fig. 2.

As for the previous paper, the algorithm learns the regular set iteratively. This means that it has an intermediate representation which converges to a final representation. The final representation is, as explained, the optimal solution.

Instead of dealing with version spaces which would exhaustively generate all possible regular sets for heavy testing, this approach is analytical. No string or code is generated unless it is needed. The main loop consists in updating a table representing the information about which strings are accepted and which are not. Each information added by a counter-example or a membership query does not only reduce the search space, but it also has a direct effect for the constructing table. This process eventually converges to the solution.

The size of the table representing the final regular set is polynomial in \( m \) and \( n \), where \( n \) is the number of different states in the minimum acceptor and \( m \), the maximum length of any counterexample string provided by the teacher during the running of \( L^* \).

Application to context-free grammars: This teaching/learning approach can be applied to context-free grammars in Chomsky Normal Form (see Fig. 2). In this case, the teacher provides the set of terminal and non-terminal symbols to the learner. The teacher answers queries about if a chain of terminal symbols can be derived from a non-terminal symbol, and counter-examples if a candidate grammars are not equivalent to the requested one.

Finding the grammar of the teacher is much simpler than for the regular set. Because the learner knows the set of symbols, it generates a super-grammar by generating all possible derivation rules. For each counter-example provided by the
A interactive teacher which can:
- Test if a string matches the requirements.
- Test if a model describes all matching strings.
- Yield an counter-example if it is not the case.

<table>
<thead>
<tr>
<th>Type of examples provided</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>An interactive teacher which can:</td>
<td>A regular automaton matching all requirements.</td>
</tr>
<tr>
<td>- Test if a string matches the requirements.</td>
<td></td>
</tr>
<tr>
<td>- Test if a model describes all matching strings.</td>
<td></td>
</tr>
<tr>
<td>- Yield an counter-example if it is not the case.</td>
<td></td>
</tr>
</tbody>
</table>

A interactive teacher which can:
- Provide the set of terminal symbols, non-terminal symbols and the start symbol of the required grammar.
- Test if a string of terminals can be derived from a given non-terminal symbol.
- Test if a given grammar is equivalent to the required one.
- Yield a counter-example if it is not the case.

Terminal symbols: a,c, other: A,C
OK: ac, aacc, aaacce . . .
Not ok: acc, cac, aac, a, . . .
Counter-examples:
for (00)* : 0011
for (0011)* : 1100

Fig. 2. Learning Regular Sets. On the left, the queries to the teacher and their answers in bold. On the right, the resulting automaton or context-free grammar

teacher, the learner recursively refines the grammar by finding and removing a rule that causes a contradiction.

In the first part of the following discussion, we provide an example for which this last method for context-free grammars is applicable.

Discussion - NFA Extension: This result is not explicitly mentioned by Angluin, but because it is a direct and easier application of his algorithm to explain, we would like to mention it.

The result for context-free grammars is applicable to an algorithm learning a regular set using a non-deterministic finite state automaton (NFA). Indeed, NFA can be seen as a particular context-free grammar in Chomsky Normal form where all transitions of type $s_0 \rightarrow a_1 s_1$ can be seen as the set of two rules: $S_0 := A_1 S_1$ and $A_1 := a_1$; and if $S_1$ is a final state then the rule $S_1 := \epsilon$ is added.

Indeed, let us assume that the teacher of an unknown regular set $H$ already provides the set of states of $H$, the sub-set of final states, and a query $\text{MEMBER}(Q, s)$ which returns yes if the string $s$ leads to a final state if the starting point is $Q$. For example, $q_0$ and $q_1$ are the accepting states of a 3-state automaton:

$$
\begin{array}{c}
\text{start} \\
\downarrow \\
q_0 \\
\downarrow \\
q_1 \\
\downarrow \\
q_2
\end{array}
$$

Then the algorithm can learn the regular set using a deterministic refinement of a NFA in the following way. For that, let us generate the NFA that consists of the same states and the same final states, and all possible transitions: for each pair of node $(n_1,n_2)$ and each letter $s$ in the alphabet $A$, it contains the transition $n_1 \rightarrow s^* n_2$. Therefore, this NFA recognizes a subset of $A^*$ which is a necessarily a superset of the teacher’s recognized language $H$. The corresponding complete NFA from the previous specifications would be the following:

$$
\begin{array}{c}
\text{start} \\
\downarrow \\
q_0 \\
\downarrow \\
q_1 \\
\downarrow \\
q_2
\end{array}
$$

To refine this NFA to discover $H$, we ask for a counter-example $a_1...a_n$ that belongs to the NFA but not to $H$. From this counter-example, we extract a sequence of states $s_0 \rightarrow a_1 s_1 \rightarrow a_2 ... \rightarrow a_n s_n$ where $s_n$ is a final state and $s_0$ is the starting state. We then ask for $\text{MEMBER}(s_1,a_2...a_n)$. If the answer is yes, this mean that the edge $s_0 \rightarrow a_1 s_1$ should not exist, so we remove it. If the answer is no, we recursively continue for $\text{MEMBER}(s_2,a_3...a_n)$. We will eventually find an edge to remove, or we will finish with the last transition query $\text{MEMBER}(s_{n-1},a_n)$ which will be no. In this case, the last edge should be removed.

For the previous example, if the teacher provides that 101 is not in $H$, then because the path $q_0 \rightarrow^1 q_1 \rightarrow^0 q_1 \rightarrow^1 q_2$ exists, we ask for $\text{MEMBER}(q_1,01)$. If the teacher returns false, we ask for $\text{MEMBER}(q_1,1)$. If this is false again, we need to remove the edge 1 from $q_1$ to $q_2$; had it been true, we would have removed the loop 0 from $q_1$, and so on.

Because we remove an edge at each iteration, this process terminates and yields an automaton recognizing the unknown regular set $H$.

Discussion - Comparisons: It would be interesting to explore the relationship between the non-mentioned pumping lemma on regular sets and this algorithm. The final size of the table is likely to be related to the constant $p$ for which every word in the language of size $> p$ can be decomposed as $uv^w$ such that $uv^w$ is part of the language and $w$ is of size $< p$. Namely, this algorithm splits the problem into accepted
prefixes and accepted postfixes until the table is closed.

In our research, we will also focus on such analytical results. Having an analytical polynomially-bounded solver such as the one we developed for Comfy [7] is always beneficial for practical reasons. Whenever it will be possible, having a strong theory behind graphical examples will surely keep the research “safe”. We would like our research not to be a set of good-looking features that would please users, but to be based on strong theoretical foundations to improve the current state-of-the-art.

This approach has the advantage of communication between the learner and the teacher. Lau was describing a synthesis system which used random traces; Angluin the synthesis system continues to explicitly ask for more examples if it did not find the regular set. The third example about automating string processing we are going to investigate is able to deal with a better communication system. Even if it requires few examples to run, it can tell if some examples are potentially wrong.

IV. AUTOMATING STRING PROCESSING

Automating string processing in spreadsheets using input / output examples (S. Gulwani, 2011) [8]

According to Gulwani, string processing is one of the most repetitive and most common task for “more than 500 million people worldwide [using] spreadsheets”. Such processing includes name/phone extraction, formatting dates to another format, and so on. However, most of them struggle to use scripting languages, macros or regular expressions to accomplish their specific tasks. Many cases presented in help forums show that users still rely on the help of human experts. Helping users sometimes lasts a few days.

Gulwani presents an algorithm to synthesize expressions operating on strings, with conditionals and loops, to match a given set of input/output examples. It runs in real time, converges quickly to a solution, and even handle noisy inputs. Contrary to the earlier step-by-step programming-by-demonstration techniques from Lau, this algorithm only requires inputs and outputs, not intermediary steps. This provides the user a great flexibility to submit multiple formats and implicit examples to the algorithm (see Fig. 3).

Solving steps

The steps described to solve this programming challenge are the following:

1) For each input | and output string s, generate all programs converting | to s.
2) Regroup/intersect the set of programs generated from two examples if their intersection is not empty
3) Partition the programs and find excluding conditions on the inputs to generate a switch statement.

For step 1), the simplified and specialized programming language that is used is displayed in Fig. 4. It is worth to note the simplicity of this functional language: it does not contain any unbounded assignment.

Furthermore, positions p in input strings are either represented by an integer offset, or by the cth position of the boundary of two regular expressions r1 and r2. This way of providing indexes over a string provides a huge set of possibilities to describe extracted tokens in the output string, while being restrictive enough to provide meaningful positions. Because position descriptors either have a meaning corresponding to the content of the string where it is located (Pos) or are statically bound to the left or the right of the string (CPos(−k) is the position of the kth character from the right of the string), there is no such complicated and unused thing such as “the position following a polynomial equation”.

<table>
<thead>
<tr>
<th>Type of examples provided</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>A set of examples where the data is formatted differently and can even be noisy.</td>
<td>A program that transforms all input for all outputs, and a suggestion for the faulty examples.</td>
</tr>
<tr>
<td>&quot;Automated String Processing&quot; → &quot;ASP&quot;</td>
<td>Loop (Aw: Concatenate (SubStr2(v1, UpperTok, w)) ) which means: “Concatenate each wth occurrence of capitals in the original string for w from 1 until it stops.”</td>
</tr>
<tr>
<td>&quot;Programming by Example&quot; → &quot;PE&quot;</td>
<td>A set of input and output strings to generalize.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type of examples provided</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>A set of examples where the data is formatted differently and can even be noisy.</td>
<td>A program that transforms all input for all outputs, and a suggestion for the faulty examples.</td>
</tr>
<tr>
<td>&quot;Automated String Processing&quot; → &quot;ASP&quot;</td>
<td>Loop (Aw: Concatenate (SubStr2(v1, UpperTok, w)) ) which means: “Concatenate each wth occurrence of capitals in the original string for w from 1 until it stops.”</td>
</tr>
<tr>
<td>&quot;Programming by Example&quot; → &quot;PE&quot;</td>
<td>A set of input and output strings to generalize.</td>
</tr>
</tbody>
</table>

Fig. 3. On the left, the list of input and output strings. On the right, a generated program transforming all input strings to their respective output strings.
String expr $P := Switch((b_1, e_1), \ldots, (b_n, e_n))$

Bool $b := d_1 \lor \ldots \lor d_n$

Conjunct $d := \pi_1 \land \ldots \land \pi_n$

Predicate $\pi := \neg Match(v, r, k)$

Trace expr $e := Concatenate(f_1 \ldots f_n)$

Atomic expr $f := Substr(v_1, p_1, p_2)$

Position $p := CPos(k)|CPos(r_1, r_2, c)$

Integer expr $c := k \mid k_1 w + k_2$

Regular expression $r := TokenSeq(T_1, \ldots, T_m)$

Token $T := \{1\} \cup \{S\}$

Character class $C := [a - z]|[0 - 9]|[a - Z]\ldots$

SpecialToken $S := Colon, Coma, Start, \ldots$

Fig. 4. Syntax of string expressions for automating string processing

Fig. 5. Efficient program set representation for the concatenation or join operation $\bullet^*$

Efficiently describing multiple programs

If a string has a size of $n$, then the number of its substrings is $n(n+1)/2$. Because it is not exponential, it is useful in practice to find the sets of programs generating each of these substrings and to concatenate them afterwards. Programs are stored in a big two-dimensional table $W[k_1, k_2]$ where $k_1 < k_2$ are indexes describing substrings of the input $\sigma$.

Although the representation of Fig. 4 describes a single program, Gulwani managed to extend this representation to describe succinctly a set of programs. For each expression, arguments are replaced by sets of programs, excepted for the Concatenate expression which is replaced by a direct acyclic graph (DAG) to efficiently handle the multidimensional join of sub-programs (see Fig. 5).

This representation of multiple joints is comparable to the $\otimes$ notation in section II, although it is implemented differently. The intersection of program sets can be computed efficiently, for example by using an algorithm similar to the intersection of two regular automata. The size can also be computed recursively without having to enumerate all programs.

With this factored representation, computing the size or intersecting two sets can be recursive and non-exponential. The size function is used to order the programs by their size. This is necessary to ensure a reasonable speed of execution even for a large amount of input/output strings decompositions.

Loop functions are created by generalization. Given two programs generating two consecutive substrings from the output $s[k_1, k_2]$ and $s[k_2, k_3]$, if those two programs can be unified modulo a certain integer expression $w$ to a single program $e$, then this program is generalized to a loop $Loop(\lambda w. e)$. This loop is then run on the input $\sigma$. If the whole run matches a substring $s[k_1, k_4]$ from the output $s$, then the expression is kept for generating this particular substring in the table $W[k_1, k_4]$.

Discussion

This recent paper is a direct answer to the need of millions of Excel users worldwide. As for Angluin’s algorithm for regular sets, Gulwani’s algorithms are clearly explained. Programs are quickly computed and relevant to the case studies encountered in Excel forums, even without adding new special tokens although this would be possible. Furthermore, the feedback provided for errors in input/output examples describes this work as very interesting for scientists in this domain.

On another aspect, Gulwani clearly underlines the trade-off between the knowledge of the algorithm or the expressiveness of the language, and the efficiency or completeness of the algorithms. For example, this algorithm cannot generate the string corresponding to characters positioned at prime number positions, because this would be irrelevant.

Concerning our research, if we decide to implement one or many ideas from this paper, our tokens will likely to be graphical aspects of software features. We therefore aim to follow and extend their notation systems with the same level of clarity and user-oriented goals for our synthesis problems. In a similar way, we would like to take care of the needs of our users to design our theory.

V. Research Proposal

We focus the research on improving the computer interaction and understanding of the process of programming, in order to assist the programmer in both high and low abstractions. As an example, we aim at providing users more comfortable programming environments.

Synthesizing program from specifications is at the core of our research. Although specifications can be formulated as logical formulas, we will also investigate the huge potential of partial specifications, such as input/output examples.

To achieve our goals, we propose to advance the state of the art in two domains.

- First, we would like to introduce implicit programming to a graphical system, by continuing the work this year which resulted in an Android application named Pong Designer. Because implicit programming requires a good knowledge of the user’s way of thinking, we plan to design user studies to comply with a various range of mental models.
- Second, we would like to contribute to the state-of-the-art systems about programming by example, by developing interactive tools to generate accurate code from input and output specifications, and by generalizing and abstracting the first approach to other classes of problems and interfaces.

A. Graphical Framework

Programming on tablets is a very active research field. The constraint that the input is restrained to fingers is challenging, although the consumer market is huge and heterogeneous.
We intend to show that the efficiency of specifying graphical behaviors by the finger is comparable to writing source code, if not more enjoyable than the latter. The first approach includes features such as on-the-fly event generation and interactive code (see Fig. 6).

We are already encountering a number of challenges concerning this graphical interface. First, we would like to discover the most efficient ways of programming with the finger. State-of-the-art systems notably include TouchDevelop [9], which provides users means to write code efficiently; and a fast API browser and auto-completion for touch systems[4]. Our approach aims at working directly on the graphical output rather than on a code which will be hidden afterwards.

Bret Victor [10], [11] provides design guidelines for future programming environment. We already implement two of his ideas, which are the principles “create by reacting” and “see the state”. The user can already on-the-fly select events such as collisions or out-of-screen, and show to our game engine how the objects should behave with multiple input and output examples. To see the state, we display all the information directly on the game. Even the time is simply displayed as a measurable success for one of the experiments, we would like to start a web community to share the user-made creations over one of the systems. Following this, we plan to create a corpus for further case studies, and respond to the demands of the users.

C. Timeline

During the first year, we plan to continue several experiments dealing with user interfaces for programming by demonstration in several domains, including the current one about programming games by example on tablets. By conducting user studies, we hope to be able to understand the needs of the users we are targeting, such as students or even younger child aged 10. We hope that our research will contribute to the teaching of programming in an enjoyable way, similarly to FunFonts [14], the Khan Academy [15] or Scratch [16]. During the second and third year, if as expected we encounter a measurable success for one of the experiments, we would like to start a web community to share the user-made creations over one of the systems. Following this, we plan to create a corpus for further case studies, and respond to the demands of the users.

REFERENCES


def x = Identity () // x => x
def c0 = ConstantByJump (0) // 0, 1, -1, 2, ...
def c1 = ConstantByJump (1) // 1, -1, 2, -2, ...
def cp2 = ConstantIncrement (2) // 2, 3, 4, ...

// | is the union version space which // outputs the best of its arguments // +, /, * are joins version space
def P1 = (c0 + c1 * x |
          (c1 * x) / cp2 |
          c0 + c0 * x + c1 * x * x)
def P2 = (c0 |
          P1)

// div means "divides"
def B1 = (cp2 div P1 |
          P1 \= 0)

// |? means "or alternatively (lazily)"
def B : Generator [Function [Boolean]] = (B1 |
          (B1 && B) |
          (B1 || B))

// Resulting version space
def F1 : Generator [Function [Int]] = (P2 |
          ite (B, P2, P2))

// Main result: we can retrieve functions // providing some examples.
println (F1.stream
          .filter (_, (1) => 2)
          .filter (_, (-2) => -4)
          .head.toReadableString)
2 * x

Fig. 7. A code to define the version space of integer to integer functions in Scala. This version does not contain any analytical acceleration, but is merely an aggressive enumeration. This enumeration will be optimized later when we will better understand the need of the users.