PLDI: G: Program Synthesis via Reverse Parsing

Vu Le
University of California, Davis
vmle@ucdavis.edu

Abstract

We propose a general program synthesis framework via a process reverse to standard parsing in compilers. Traditionally, a compiler breaks down a program into tokens and applies various optimization techniques to generate lower-level code. In contrast, a synthesizer instantiated from our framework takes a subset of such tokens, generates additional ones to glue the provided tokens into a ranked set of programs in its underlying language.

To demonstrate the *applicability* of our framework, we created three end user-targeted instantiations. In the first instantiation, users use a query interface to find relevant APIs to synthesize smartphone automation scripts. The second instantiation is a natural language front end of TouchDevelop in which users enter English keywords to synthesize matching TouchDevelop snippets. The third instantiation, targeting Microsoft Access users, allows them to synthesize SQL queries by providing only selection columns and their filtering conditions. We illustrate the *usability* of instantiations specialized from our framework via two user studies on the first system. Our results show that users are able to construct intended programs from the core tokens within a reasonable amount of time.

Although their underlying languages are very different, the three instantiations exhibit similar efficiency and effectiveness. This indicates that the framework can be fruitfully applied to other end-user programming domains as well.

1. Introduction

The last few years have seen a tremendous rise in the number of people with access to computational devices, especially smartphones and tablets. This trend is due to falling average prices, increased usability, functionality and programmability that, among other factors, make the devices more accessible to a wider range of consumers. Most people who have or will have access to those devices are non-programmers who are unable to make full effective use of their devices. The application market is unlikely to meet the personalized needs of those users. The economic model in the application market favors the presence of only those applications that would be used "as-is" by a large number of people. This has created a significant opportunity to cater to the needs of the long tail of people looking for personalized applications that only they, or very few other people, would want to use.

Synthesis Framework This paper introduces *a novel, general synthesis framework* that allows users to synthesize programs in any underlying domain-specific languages (DSLs) simply by entering a subset of important tokens in the desire program. The synthesizer designer can instantiate a new synthesizer for a DSL simply by specifying the token kinds that need to be entered by users (for example, function names, literals) and the cost model that ranks programs in the DSL. The framework employs a generic algorithm that gradually unrolls the grammar, plugs in the tokens provided by users, and generates new tokens to synthesize complete programs. It then uses the cost model to rank and return the most likely programs to the user.

Our framework has a few unique aspects. First, it introduces a novel way to express user intent— the core constituent program constructs—to synthesize the intended program. This overcomes weaknesses in other program synthesis techniques such as Programming By Examples, or Programming By Demonstration, which typically are very fragile towards learning looping constructs, not applicable in many domains, or may require many rounds of interaction with users (see Section 5 for details comparison).

Second, our framework provides a unified procedure to create specialized synthesizers for any DSLs. In contrast, state-of-the-art synthesizers such as FlashFill [4] or SmartSynth [7] are highly specialized to their underlying DSLs and not generally applicable to other DSLs. We yet to have a general, domain-independent methodology and technical framework that can apply to any DSLs.

Third, it provides a user-friendly environment to facilitate easy creation of programs. Synthesizers instantiated from this framework relieve users from having to worry about the detailed syntax and semantics of the underlying DSL. These low-level syntactic and semantic details are "precomputed" and incorporated into the synthesizer by its designer during the specialization process. As a result, users only need to identify the core concepts relevant to the construction of their intended programs. In contrast, typical programming models require paying attention to many, often unnatural and tedious, constraints, such as providing arguments to a method call in a specific order (even when the arguments can be disambiguated by use of typing information), or specifying data-flow between APIs (even when one can only consume the output of another), or ensuring syntactic correctness of the program with use of proper keywords and delimiters (even when skipping some of these would not lead to alternative, ambiguous interpretations).

Framework Instantiations Based on this framework, we created three instantiations that target end users. The first instantiation generates smartphone automation scripts from API names (an automation script executes a sequence of actions when a phone event occurs under a certain condition). We made its demo video available at http://sdrv.ms/TxeXmi. The second instantiation allows users to generate TouchDevelop snippets from English keywords in the middle of their programs (demo video at http://sdrv.ms/SVqrV7). The third instantiation synthesizes Microsoft Access SQL queries from selection columns and their filtering conditions. These instantiations demonstrate the general *applicability* of our framework towards different DSLs. The framework works efficiently in all cases where the language either is or is not designed by us.

To gain deeper understanding regarding the *usability* of synthesizers specialized from our framework, we conducted two user studies on our smartphone instantiation. Our results show that the users are able to (1) construct the intended programs from their core constructs, and (2) select the intended programs from the returned list of most likely programs.

Contributions We make the following main contributions:

- We introduce a novel and general synthesis framework that synthesizes programs from their core constructs. Our framework

Figure 1: The search interface to identify APIs.

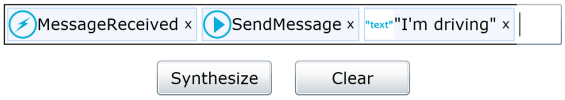


Figure 2: A sample query for Example 1 in our specialization.

benefits both synthesizer designers (*i.e.* easy creation of domain-specific synthesizers from any DSLs) and synthesizer users (*i.e.* easy creation of programs from their core constructs).

- We instantiate our framework to three different domains having different DSLs, which illustrates the general applicability of our framework.
- We conduct two users studies that clearly demonstrate the usability of synthesizers instantiated from our framework.

Next, we use a few examples to illustrate our framework and its three instantiations.

2. Illustrating Examples

We give a high-level overview of our framework via examples from three different domains: smartphone automation scripts, TouchDevelop scripts, and database queries.

Smartphone Automation Scripts We use an example from the tutorial section of App Inventor, a visual programming environment [13] to demonstrate the user-friendliness of this instantiation.

EXAMPLE 1. [No Text While Driving] *When the phone receives a new text message while the user is driving, it replies the message "I'm driving right now. I'll contact you shortly."*

To program this example, App Inventor's users need to identify the relevant APIs, understand their signatures, and use temporary variables to connect these APIs together. In contrast, the users of our specialization only need to specify the APIs via a search interface (Figure 1), in any arbitrary order.

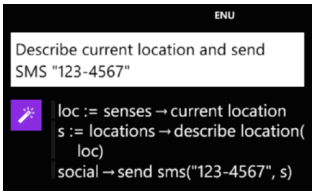
Our approach brings some important benefits. First, users are not required to understand/remember the low-level details/interface of any APIs. For instance, users do not need to know that MessageReceived generates two variables in a specific order, neither what they mean (similarly for SendMessage). Second, users may omit certain unnecessary program constructs, such as temporary variables, that can be automatically synthesized. This lets users focus on the core, high-level aspects of their desired scripts.

Once the user has given the components, the synthesizer unrolls the script grammar, plugging in the provided components and generating glue code to synthesize the desired script (Figure 2).

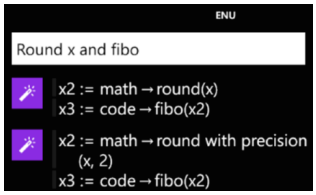
TouchDevelop Snippets We use two programs written by TouchDevelop users to illustrate this instantiation.

EXAMPLE 2. [Send Current Location via SMS] *The user wants to send her current location to the number 123-4567 via SMS.*

EXAMPLE 3. [Round and Fibo Variable] *The user wants to round the local variable* x *and calculate its Fibonacci number.*



(a) A sample query for Example 2.



(b) A sample query for Example 3.

Figure 3: A user use keywords to synthesize TouchDevelop snippets.

To program Example 2 in TouchDevelop, the user needs to identify the constituent APIs and pass the correct arguments to them. In contrast, in our instantiation, she only needs to enter relevant keywords to synthesize the desired snippet (Figure 3a).

Users may also create context-sensitive snippets in the middle of their programs. In Example 3, the user refer to both local variable x and the user-defined function code→fibo in her query (Figure 3b).

Microsoft Access SQL Queries We have encountered many questions regarding the SQL syntax during our study on Access user forums. We use one of them to demonstrate the specialization of our framework to database queries.[1]

EXAMPLE 4. [Aggregation with Join] *The user wants to join three tables and apply aggregate functions on them. Below is the intended query:*

```
SELECT A.id, B.id, COUNT(C.id), SUM(C.qty)
FROM (A INNER JOIN B ON A.id = B.a_id)
      INNER JOIN C ON B.id = C.b_id
GROUP BY A.id, B.id
```

The user, despite having some prior SQL experience, got stuck because he missed column A.id in the GROUP BY clause. In fact, novice users who have no prior experience would find it even more challenging to construct well-formed SQL queries, especially when these queries involve JOIN and GROUP BY clauses. To make matters worse, the returned error messages, if any, are often cryptic to most users. The above experienced user had to seek help because he could not understand the error message.

Under our SQL specialization, users only need to focus on the core elements of their desired queries, and need not worry about additional syntactical constraints that can be synthesized automatically. For the above example, they only declare the four elements that they wish to have in the results, namely A.id, B.id, COUNT(C.id), SUM(C.qty). The synthesizer generates additional constructs to combine the input elements into the intended query. Note that users can provide the function and column separately (*e.g.*, COUNT, C.id instead of COUNT(C.id)).

3. Synthesis Framework

In this section, we formally define the synthesis framework and discuss its algorithm.

3.1 Definition

A synthesizer of a DSL $\mathcal{L}$ instantiated from our framework is a tuple $\langle G, T, \hat{T}, \text{SA}, \mathcal{C} \rangle$ where:

- G is a specialized context-free grammar whose terminal set Σ is split into two disjoint sets: the *user-supplied* Σ_1 and the *system-generated* Σ_2.
- $T : Token \rightarrow \Sigma_1$ is the tokenizer that maps a token to a terminal symbol. We use it to identify the token type of a provided token to place it to the right position in the expansion.

[1] http://www.accessforums.net/queries/sql-query-sum-inner-join-group-7305.html

Algorithm 1: Compute k most likely programs from a set of tokens U provided by the user.

```
1  begin
2      P ← {}                                   /* top programs queue */
3      Q ← {⟨S, U⟩}                             /* worklist queue */
4      while Q ≠ {} do
5          ⟨ℓ̃, U_ℓ̃⟩ ← Q.Dequeue()               /* process a task */
6          if C(ℓ̃) > Min_{P∈P} C(P) then continue   /* skip */
7          if ℓ̃ is concretized then
8              if |U_ℓ̃| = 0 and |P| < k then
9                  P.Enqueue ⟨ℓ̃, C(ℓ̃)⟩
10         else                                 /* continue to unroll ℓ̃ */
11             α ← choose(GrammarSymbols(ℓ̃))
12             if α ∈ Σ1 then
13                 U_α ← {u ∈ U_ℓ̃ | T(u) = α})
14                 foreach w ∈ U_α do
15                     ℓ̃′ ← ℓ̃[w/α]
16                     if SA(ℓ̃′) then
17                         Q.Enqueue(⟨ℓ̃′, U_ℓ̃ \ {w}⟩)
18             else if α ∈ Σ2 then
19                 foreach w ∈ T̂(α) do
20                     ℓ̃′ ← ℓ̃[w/α]
21                     if SA(ℓ̃′) then
22                         Q.Enqueue(⟨ℓ̃′, U_ℓ̃⟩)
23             else                             /* unroll non-terminal */
24                 R_α ← {r | α → r ∈ Simplify(R, U_ℓ̃)}
25                 foreach r ∈ R_α do
26                     ℓ̃′ ← ℓ̃[r/α]
27                     if SA(ℓ̃′) then
28                         Q.Enqueue(⟨ℓ̃′, U_ℓ̃⟩)
29     return P /* worklist empty, return result */
```

- $\hat{T} : \Sigma_2 \rightarrow Token$ is the detokenizer that generates tokens for system-generated terminals. It is responsible for generating glue code that combines the supplied tokens together.
- SA is the semantic checker that eliminates undesired programs.
- $\mathcal{C}$ is the cost model that ranks programs in G.

Typically, the language $\mathcal{L}$ is already equipped with a context-free grammar G', a tokenizer T and a semantic checker SA. The synthesizer designer only needs to: (1) identify a split of Σ of G' into Σ_1 and Σ_2; (2) implement $\hat{T}$ that generates glue code; and (3) define $\mathcal{C}$ that ranks $\mathcal{L}$ programs.

3.2 Algorithm

At a high level, our synthesis algorithm keeps unrolling the grammar symbols to find programs that uses all user-provided tokens U. Algorithm 1 describes this process. It maintains a worklist $\mathcal{Q}$ of *abstract programs* (the current result of unrolling the grammar) and their corresponding sets of *unused user-supplied tokens* (line 3). In processing a worklist item, if the abstract program $\tilde{\ell}$ is fully unrolled and it has consumed all tokens in U, we have found a program and add it to the result list (lines 7–9). Otherwise, the abstract program is not fully unrolled. It has a to-be-concretized grammar symbol α (line 11), which might be one of the followings:

- A user-supplied terminal symbol: We replace α with one of the matching tokens in U (lines 12–17).

Script P	::=	$I\ E\ F*\ C\ Stmt+$
Parameter I	::=	**input** '(' **i** (, **i**)* ')' $\mid \epsilon$
Event E	::=	**when** Event $RList \mid \epsilon$
Return List $RList$	::=	**returns x** (, **x**)* $\mid \epsilon$
Conversion F	::=	Conversion '('$AList$')' $RList$ ';'
Arg. List $AList$	::=	a (, a)* $\mid \epsilon$
Condition C	::=	**if** '(' Π ($\wedge$ Π)* ')' **then** $\mid \epsilon$
Predicate Π	:=	Predicate '('$AList$')'
	$\mid$	a_1 **in** $a_2 \mid a_1 = a_2$
Statement $Stmt$	::=	$S \mid$ **foreach x in** a **do** $Stmt+$ **od** ';'
Atomic Stmt S	::=	$A \mid F$
Action A	::=	Action '('$AList$')' $RList$ ';'
Argument a	::=	**i** $\mid$ **x** $\mid$ *literal*

Figure 4: The smartphone automation script language. The Σ_1 terminals are underlined while the Σ_2 terminals are in **bold**. **i** is a script parameter, **x** is a temporary variable.

- A system-generated terminal symbol: We use the detokenizer $\hat{T}$ to generate a token for α and replace it. For example, if the temporary variable symbol is in Σ_2 and we reach its symbol, the $\hat{T}$ will generate a temporary variable (lines 18–22).
- A nonterminal symbol: We replace α with its "best" right-hand-side production (*w.r.t.* $\mathcal{C}$) and enqueue the rest to process later (lines 23–28).

The process continues until the worklist is empty. The cost model is used to skip unrolling an abstract program $\tilde{\ell}$ if it estimates that the lower bound cost of all programs unrolled from $\tilde{\ell}$ is greater than those of the programs in the current result list (line 6). One unique aspect of our algorithm is illustrated by the two alternatives in expanding α (the first and second bullets above). One is a tokenization step (*abstraction*) that selects a user-supplied token to expand a terminal symbol. The other is a detokenization step (*concretization*) that selects a system-generated token to expand a terminal symbol. To the best of our knowledge, this combination has not been formalized and explored in the literature.

4. Evaluation

We now evaluate two aspects of our framework: (1) its general applicability via three instantiations having different underlying DSLs, and (2) its instantiations' usability via two user studies on the smartphone instantiation.

4.1 Instantiations

A synthesizer designer can use our framework to build domain-specific synthesizers. We demonstrate this process via the instantiation of three DSLs: the smartphone automation language in SmarthSynth [7], the TouchDevelop language, and the Microsoft Access SQL language.

Automation Script Synthesizer Figure 4 shows the language of smartphone automation scripts. The language is fairly simple with the event at the top, followed by the optional conditionals, and a sequence of phone actions. Nested loop is allowed.

Users use a special query interface to search for and enter all API names (which might be events, conditions, and actions - these are Σ_1) that are related to the intended program. Note that they need not to worry about these API's internal details such as the number of arguments, the type of each argument and their relative order. The synthesizer generates additional constructs (keywords, temporary variables, arguments, looping and conditional structures - these are

Program P	::=	S+
Stmt S	::=	$\underline{if}$ ($\underline{not}$)? E '{' S+ '}' ';'
	|	$\underline{while}$ ($\underline{not}$)? E **do** '{' S+ '}' ';'
	|	$\underline{for}$ $\mathbf{x} := 0$ **to** E **do** '{' S+ '}' ';'
	|	$L := E$ ';'
Lvalue L	::=	$\mathbf{x}$ | **localVar** | **globalVar**
Expr E	::=	E $mathOp$ E ($mathOp$ E)* | E $comOp$ E
	|	C | $\underline{globalVar}$ | $\underline{localVar}$ | $\underline{literal}$
Call Expr C	::=	$\underline{userFunc}$ ('(' a (',' a) * ')')?
	|	$\underline{builtinFunc}$ ('(' a (',' a) * ')')?
Argument a	::=	E | **defaultArg**

Figure 5: The language of TouchDevelop instantiation. $\mathbf{x}$ is a temporary variable, **defaultArg** is an argument's default value.

Σ_2) to synthesize the intended programs. We use a rule-based cost model to rank the results. It contains a few expert rules derived from scripts collected from user forums (for example, one rule states that a script is not likely if it contains an unused variable). The synthesizer also suggests likely missing tokens to users, which happens when some provided tokens are not used in the final program. The likely missing tokens are those that make these unused tokens used.

TouchDevelop Synthesizer Our synthesizer generates code for a sub-language of TouchDevelop, which is shown in Figure 5. We only concentrate on the most popular language constructs that end users are familiar with. Users may invoke the synthesizer in the middle of their programs to synthesize snippets relevant to their existing code. They simply enter a few English keywords, and a ranked list of synthesized snippets are shown for them to select. The synthesizer has an NLP layer that maps keywords into TouchDevelop tokens, which are fed into the core synthesizer to generate matching snippets.

In this instantiation, Σ_1 are the system and user-defined API names, system variables (*e.g.*, current location, the color red), user-defined global and local variables, and keywords such as for, if, and while. The cost model, which is an enhanced version of the automation script synthesizer's cost model, incorporates statistics about API usage from the large TouchDevelop user code base. We use the built-in TouchDevelop's semantic checker. The synthesizer has been released since version 2.10 of TouchDevelop.

Database Query Synthesizer We choose Microsoft Access SQL as our target because it has a large novice user base who might benefit the most from our specialization. Users synthesize queries by providing only the elements in the SELECT clause and the filtering conditions. The system generates additional code to construct well-formed queries, and presents the most likely ones to the users.

Figure 6 shows the syntax of our sub-language of Access SQL. A query Q must contain at least the SELECT and FROM clauses. The SELECT clause contains either table columns or aggregation functions operating on the columns. Tables are joined via inner join, left join or right join operations.

Users only need to provide the select column name $\underline{table.col}$, the aggregate function $\underline{aggfunc}$, and the filtering expression $\underline{expr}$, which constitute the set Σ_1. The remaining terminals (which belong to Σ_2) along with appropriate connections are synthesized automatically. Note that there are two types of columns in the grammar: one in S (provided by users) and the other in C (generated by the synthesizer). Our cost model contains a few rules to reflect common queries created by users in the forums. The semantic checker rejects queries that are not well-formed under the SQL semantics.

Q	::=	**SELECT** S ($,S$) *
		FROM **table** (J **JOIN** **table** **ON** C O C) *
		(**WHERE** $\underline{expr}$ (**AND** $\underline{expr}$)*)?
		(**GROUP BY** C ($,C$)*)?
		(**HAVING** $\underline{expr}$ (**AND** $\underline{expr}$)*)?
S	::=	$\underline{table.col}$ | $\underline{aggfunc}$ '(' $\underline{table.col}$ ')'
C	::=	**table.col**
J	::=	**INNER** | **LEFT** | **RIGHT**
O	::=	= | <> | > | < | >= | <=

Figure 6: The language of Microsoft Access instantiation.

4.2 Usability Measurements

We now report the results on measuring the usability of our smartphone specialization. We picked the benchmarks for the smartphone domain from [7]. It contains 50 smartphone automation scripts having various complexity. All the experiments was run on a machine with an Intel® i7 2.66 GHz CPU and 4 GB of RAM.

The audience of our first user study is a class of freshman students at UC Davis. We built a website and asked them to participate. We gave the students a short 15 minutes tutorial. Each student was randomly assigned either the even half or the odd half of our benchmarks (25 problems each). Eleven students completed all their assigned problems (4 in the even set and 7 in the odd set).

API Selection The participants provided the correct set of APIs and literals for nearly 80% of the benchmarks without using our inferring missing APIs feature. The precision jumps to 93% when this feature was enabled. Fig. 7 shows the result for each user. The results indicate that our system is easy to use. Note that the students who took part in our study were first-time users of our system and all the benchmarks were drawn from the other tools' help forums where users were struggling with these problems and seeking help.

For 28 out of 50 benchmarks, every user provided the correct set of APIs and literals. For 21 out of the remaining 22 benchmarks, only one user (out of 4 or 7 users in the odd/even benchmark sets) failed to provide the complete set. This is statistically insignificant. The only benchmark that has more than one user providing an incomplete set is benchmark 1. In this benchmark, two users missed SendMessage while the other one missed the ReadText. We suspect that the query was so long that the users forgot to enter those APIs.

Time Taken by the User We measured the completion time for each benchmark by marking the time of first keystroke to the synthesis interface and the moment when the user hit the search button. Figure 8 shows that on average, a user needed 50 seconds to program a benchmark. The actual time may be smaller because users may be distracted during their tasks in their own environments.

Selecting the Desired Script Once a user has provided APIs and literals, our system synthesizes the top scripts and presents them to the user. She must be able to choose her desired script from the top list. We conducted an additional user study to measure how easy it is for users to distinguish the top script candidates.

The participants were a group of 7 students at UC Davis. We gave them each 50 questions. Each question consisted an English description of a benchmark problem and two choices: (1) A script that either matches or does not match the given English description, and (2) The statement that "The above script does not match the description." For the first choice, we chose the correct script half of the time, and randomly chose an incorrect script in the top 5 for the other half. We illustrated the language by providing each participant three examples and their matching descriptions.

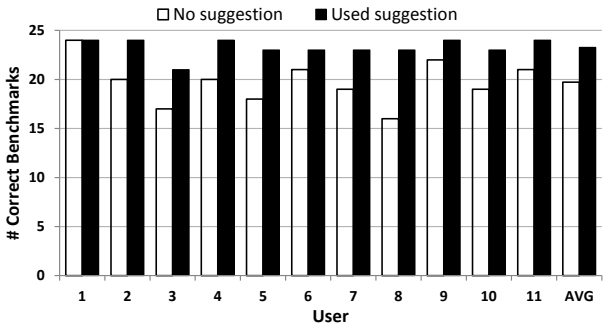


Figure 7: The number of problems that the users did correctly without/with the inferring missing API feature.

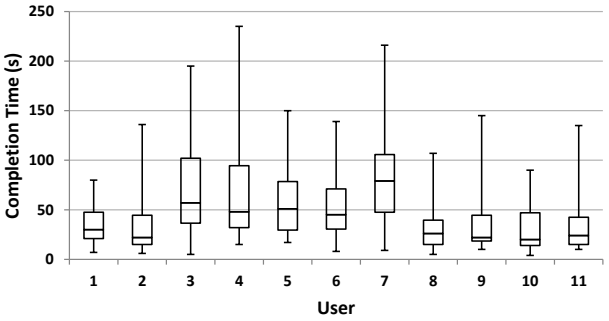


Figure 8: The time each user uses for one benchmark.

The participants were correct 97% of the time, and spent 26.5 seconds per problem on average. The results show that users can easily select her desired script from the top scripts.

5. Related Work

The field of end-user programming is gaining increasing attention, and a variety of techniques have been proposed including Programming by Example (PBE) [8], Programming by Demonstration (PBD) [2], Visual programming [1, 5, 14], Keyword Programming [9–12].

The PBE and PBD techniques construct a program from a set of input/output examples and from a set of complete traces, respectively. These forms of intent reflect *semantic* properties of the desired program. In contrast, users of our framework synthesize programs by providing the core program constructs, which reflect *syntactic* properties of the desired program. As a result, while most existing work on PBE/PBD employs version space algebra based search techniques [3, 4] over the underlying grammar, our synthesizer performs A* directed search over the underlying grammar.

PBE techniques are not applicable in some domains like smartphone scripting because of the statefulness of programs. PBD techniques require a greater burden on part of the user (in terms of the user having to provide inputs for each of the components in the trace, which becomes especially tricky when the inputs are environment triggered, as in case of various events). Furthermore, PBD techniques are quite fragile in presence of loops and hence have not yet been very successful [6]. Since our system is token-based, we do not need to worry about the problem of environmental inputs and robust loopy demonstrations.

Visual programming environments visualize programming constructs as puzzle-like pieces, and the users program by selecting, configuring, and connecting these pieces together. Although the aforementioned programming environments have significantly improved user-friendliness for users, they still require users to think and act like programmers. In particular, users need to understand the low-level interfaces of the pieces in order to configure and connect them correctly. In contrast, our system provides much greater automation, since our users need to provide only a important subset of the required pieces. Moreover, since users do not have to specify the connections between pieces, they need not concern themselves with low-level programming details.

The goal in keyword programming is to generate type-correct code that matches a given sequence of keywords. [9–12]. However, these techniques have been restricted to discovering a single method call expression in Java programs [11], or a single command in domain-specific web scripting languages like Chickenfoot [10] or Koala [9], which have a small number of possible function trees.

Recently, SmartSynth developed new techniques to let user synthesize a smartphone script from its description [7]. However, SmartSynth still faces some practical issues because it attempts to convert an entire description to a script. In particular, it assumes that an API corresponds to a continuous text chunk in the description, which is not always the case. In contrast, our framework does not suffer from the mapping ambiguity, because user identifies the APIs herself. More importantly, the technique described in SmartSynth is specialized to the underlying DSL while our framework is domain-independent and can be instantiated to any DSLs, which we have demonstrated in previous sections.

6. Conclusion

Traditionally, PL has catered to the needs of the elite class of programmers, primarily because only they had access to programmable computing devices. The trend is now changing with hundreds of millions of end users having access to those devices today, and the number is expected to rise to a billion in a few years. We believe that our *novel, systematic and general* approach can be fruitfully applied to many application domains of end-user programming.

References

[1] S. Cooper. The design of Alice. *Trans. Comp. Educ.*, 2010.

[2] A. Cypher, editor. *Watch What I Do – Programming by Demonstration*. MIT Press, 1993.

[3] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, 2011.

[4] S. Gulwani, W. Harris, and R. Singh. Spreadsheet data manipulation using examples. In *Commun. ACM*, 2012.

[5] M. Kölling. The Greenfoot programming environment. *Trans. Comput. Educ.*, 2010.

[6] T. Lau. Why PBD systems fail: Lessons learned for usable AI. In *CHI 2008 Workshop on Usable AI*, Florence, Italy, 2008.

[7] V. Le, S. Gulwani, and Z. Su. Smartsynth: Synthesizing smartphone automation scripts from natural language. In *MobiSys*, 2013.

[8] H. Lieberman, editor. *Your wish is my command: programming by example*. Morgan Kaufmann Publishers Inc., 2001.

[9] G. Little, T. A. Lau, A. Cypher, J. Lin, E. M. Haber, and E. Kandogan. Koala: capture, share, automate, personalize business processes on the web. In *CHI*, pages 943–946, 2007.

[10] G. Little and R. C. Miller. Translating keyword commands into executable code. In *UIST*, pages 135–144, 2006.

[11] G. Little and R. C. Miller. Keyword programming in Java. In *ASE*, pages 84–93, 2007.

[12] G. Little, R. C. Miller, V. Chou, M. Bernstein, T. Lau, and A. Cypher. Sloppy programming. 2010.

[13] MIT. App Inventor for Android. http://appinventor.mit.edu/.

[14] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai. Scratch: programming for all. *Commun. ACM*, 2009.