We are all designers.
Programmers, too, want playful, interactive experience.

Synthesizing Programs from Programmer Insight

Ras Bodik
University of California, Berkeley
Reality Check (Programmer’s Apprentice 80’s)

Programmer first develops the spec of the hashtable-based Set S:

\[ \text{delete()}: \; \text{delete}(S,v) = S \setminus \{ v \} \]

Then he tells his insight to the apprentice:

\[ \text{Prgmr:} \; \text{Here's the code.} \]

\[
\text{(DEFINE DELETE (KEY)}
\]
\[
\text{; ; hash table delete routine.}
\]
\[
\text{(PROG (BKT1 BKT2)}
\]
\[
\text{(SETQ BKT1 (TBL (HASH KEY)))}
\]
\[
\text{(SETQ BKT2 (BKT-DELETE KEY BKT1))}
\]
\[
\text{(COND ((NOT (EQ BKT1 BKT2))}
\]
\[
\text{; ; special case when first entry in bucket deleted.}
\]
\[
\text{(STORE (TBL (HASH KEY)) BKT1))))}
\]

\[ \text{Apprentice: I'm sorry but this code is not consistent with your plan. In the special case you are storing the bucket before splicing, rather than the bucket after splicing. Did you mean (STORE (TBL (HASH KEY)) BKT2)?} \]

\[ \text{Prgmr: Yes, fix the code. Also, you can code BKT-DELETE as the standard list deletion algorithm.} \]

\[ \text{Apprentice: Here it is.} \]

\[
\text{(DEFINE BKT-DELETE (KEY BUCKET)}
\]
\[
\text{; ; list deletion routine coded by apprentice.}
\]
\[
\text{(PROG (CURRENT PREVIOUS)}
\]
\[
\text{(SETQ PREVIOUS NIL) ;initialize trailing ptr.}
\]
\[
\text{(SETQ CURRENT BUCKET) ;initialize ptr to current entry.}
\]
\[
\text{LP (COND ((NULL-current)}
\]
\[
\text{(RETURN BUCKET))}
\]
\[
\text{; ; splice out entry.}
\]
\[
\text{(COND (PREVIOUS}
\]

Programmer fixes the plan.

A variant of this we can do today.
aLisp

[Andre, Bhaskara, Russell, ... 2002]
aLisp: learning with partial programs

Problem:
- implementing AI game opponents (state explosion)
- ML can’t efficiently learn how agent should behave
- programmers take months to implement a decent player

Solution:
- programmer supplies a skeleton of the intelligent agent
- ML fills in the details based on a reward function

Synthesizer:
- hierarchical reinforcement learning
What’s in the partial program?

Strategic decisions, for example:

– first train a few peasant
– then, send them to collect resources (wood, gold)
– when enough wood, reassign peasants to build barracks
– when barracks done, train footmen
– better to attack with groups of footmen rather than send a footman to attack as soon as he is trained

[from Bhaskara et al IJCAI 2005]
(defun single-peasant-top ()
  (loop do
    (choose '(call get-gold) (call get-wood))))

(defun get-wood ()
  (call nav (choose *forests*))
  (action 'get-wood)
  (call nav *home-base-loc*)
  (action 'dropoff))

(defun nav (l)
  (loop until (at-pos l) do
    (action (choose '(N S E W Rest))))

this.x > l.x then go West
SKETCH

[Solar-Lezama, Bodik, ... 2006]
SKETCH

Problem:
- 10-line algorithm requires a 1000-line program
- devious corner cases

Solution:
- programmer “sketches” the implementation – this sketch is a template completed by the synthesizer
- programmer also supplies a (slow) reference implementation, which defines desired observable behavior

Synthesizer:
- SAT-based inductive synthesis
Synthesize a concurrent list data structure

The data structure:
- linked list of Nodes
- sorted by Node.key
- with sentries at head and tail

The problem: implement a concurrent remove() method
What’s the programmer insight?

Sequential `remove()`:

- **Insight 1**: for efficiency, use *fine-grain* locking
  - lock individual *Nodes* rather than the whole list
- **Insight 2**: Maintain a sliding window with two locks
On to mechanics ... ?

\[ -\infty \rightarrow a \rightarrow b \rightarrow c \rightarrow +\infty \]
Capture the insight in a sketch

```c
#define comp { | ((cur|prev)(.next)? | null)
    (== | !=) ((cur|prev)(.next)? | null) |}
#define loc { | (cur | prev | tprev) (.next)? |}

void Remove(int in){
    Node cur = this.head, prev = null;
    lock({| cur(.next)? |});
    while( cur.val < in){
        Node tprev = prev;
        reorder {
            if(comp) lock( loc );
            if(comp) unlock( loc );
            prev = cur;
            cur = cur.next;
        }
    }
    if( cur.val == in ){ prev.next = cur.next; } 
    unlock( {|| cur(.next)? |} );
    unlock( {|| prev(.next)? |} );
}
```
SKETCH generates a correct program

```java
void Remove(int in){
    Node cur = this.head, prev = null;
    lock( cur );
    while( cur.val < in){
        Node tprev = prev;
        if(prev != null){
            unlock(prev);
        }
        prev = cur;
        cur = cur.next;
        lock(cur);
    }
    if( cur.val == in ){ prev.next = cur.next; }
    unlock( cur );
    unlock( prev );
}
```

SKETCH finds correct conditions and expressions

SKETCH orders these statements correctly
SKETCH: just two constructs

spec:

```c
int foo (int x)
{
    return x + x;
}
```

sketch:

```c
int bar (int x) implements foo
{
    return x << ??;
}
```

result:

```c
int bar (int x) implements foo
{
    return x << 1;
}
```

Assertions can also be used to state safety properties
It’s synthesis from partial programs
SKETCH

ref implementation → SAT-based inductive synthesizer → hole values

sketch →
aLisp

reward function \rightarrow \textit{hierarchical reinforcement learning} \rightarrow \text{learnt choice functions}

aLisp partial program \rightarrow
First problem with partial programming

Where does specification of correctness come from? Can it be developed faster than the program itself?

Unit tests (input, output pairs) sometimes suffice.

Next two projects go in the direction of saying even less.
Prospector

[Mandelin, Bodik, Kimelman 2005]
Software reuse: the reality

Using Eclipse 2.1, parse a Java file into an AST

```java
IFile file = ...
ICompilationUnit cu = JavaCore.createCompilationUnitFrom(file);
ASTNode node = AST.parseCompilationUnit(cu, false);
```

Productivity < 1 LOC/hour  Why so low?

1. follow expected design? two levels of file handlers
2. class member browsers? two unknown classes used
3. grep for ASTNode? parser returns subclass of ASTNode
Problem:

APIs have 100K methods. How to code with the API?

Solution:

Observation 1: many reuse problems can be described with a have-one-want-one query $q=(h,w)$, where $h,w$ are static types, eg ASTNode.

Observation 2: most queries can be answered with a jungloid, a chain of single-parameter “calls”. Multi-parameter calls can be decomposed into jungloids.

Synthesizer:

Jungloid is a path in a directed graph of types+methods.

Observation 3: shortest path more likely the desired one
Integrating synthesis with IDEs

• How do we present jungloid synthesis to programmers?
• Integrate with IDE “code completion”

Queries: 
(IFile, ASTNode)
(IEditorPart, ASTNode)
SMARTedit*

[Lau, Wolfman, Domingos, Weld 2000]
**SMARTedit**

**Problem:**
- creation of editor macros by non-programmers

**Solution:**
- user demonstrates the steps of the desired macro
- she repeats until the learnt macro is unambiguous
- *unambiguous* = all plausible macros transform the provided input file in the same way

**Solver:**
- version space algebra
An editing task: EndNote to BibTex

%0 Journal Article
%1 4575
%A Richard C. Waters
%T The Programmer's Apprentice: A Session with KBEmacs
%@ 0098-5589
%V 11
%N 11
%P 1296-1320
%D 1985
%R http://dx.doi.org/10.1109/TSE.1985.231880
%I IEEE Press

@article{4575,
    author = {Waters, Richard C.},
    title = {The Programmer's Apprentice: A Session with KBEmacs},
    journal = {IEEE Trans. Softw. Eng.},
    volume = {11}, number = {11}, year = {1985},
    issn = {0098-5589},
    pages = {1296--1320},
    doi = {http://dx.doi.org/10.1109/TSE.1985.231880},
    publisher = {IEEE Press}, address = {Piscataway, NJ, USA},
}

Demonstration = sequence of program states:

1) cursor in (0,0) buffer = “%0 …” clipboard = “”
2) cursor in ^ buffer = “%0 …” clipboard = “”
3) ...

Desired macro:

move(to after string “%A “)
...

...
Version space = space of candidate macros

Version space expressed in SKETCH (almost):

```c
#define location { | wordOffset(??) | rowCol(??,??)
  | prefix(“??”) | ... |}

repeat ?? times {
  switch(??) {
    0:   move(location)
    1:   insert({| “??” | indent(??,”??”) |})
    2:   cut()
    3:   copy()
    ...
  }
}
```
Are these two also about partial programs?

correctness criterion → synthesizer → completion

partial program → merge ← complete program
Prospector

have, want query

jungloid template + API

shortest path search

user selection

ranked jungloids

desired jungloid
SMARTedit*

- Demonstration(s)
- Macro template
- Input file
- Run the macro
- Version space algebra
- Completed macro(s)
- Set of macro parameters
- Processed file
SKETCH

ref implementation → "SAT-based inductive synthesizer" → hole values

sketch →
First problem with partial programs

How to supply a specification?

**Reference implementation or Correctness check:**
all input-output pairs

**Unit testing:**
sparse input-output pairs

**Prospector:**
Approximate! From full correctness to type safety

**SmartEdit**:
tell the user if valid completions diverge on any input
Second problem with partial programs

Early insight is too fuzzy to express in partial programs

If only we had a demonstration of the program! The demonstration would reveal the insight.

Could an executable oracle demonstrate it for us?
Angelic Programming

[Galenson, Chandra, Bodik  ongoing]
Demonstrations by an oracle

Say we know that we **want** to reverse a linked list...

...but not **how**

So we ask an oracle how to do it.

Once it shows us how...

...we can mimic it with a deterministic program.

But how do we tell the oracle what to demonstrate?
Synthesize demonstration

correctness check → synthesizer → completion = demonstration

partial program
Oracular choice operator

A new kind of ‘choice’ in the partial program

```
x := !! -- oracular choice of value
y := foo(x)
assert(y is prime) -- correctness check
```

!! returns a value that makes the program correct

- !! possesses oracular look-ahead
- can be implemented with backtracking [Floyd 1967]
- more scalable with SAT (reduced to ??-synthesis in SKETCH)
Creating a demonstration of list reversal

// !! evaluates to a value so that all assertions pass

```
reverse(list) {
  while (!!)
    !!.next = !!
  reversedList = !!
  assert(reversedList is reversal of list)
  return reversedList
}
```
Are there other ways to turn things around?
Variants on partial-program synthesis

- correctness criterion → synthesizer → completion
- partial program → completion
- correctness check → synthesizer → angelic demonstration
- angelic partial program → synthesizer
- demonstrations → completion
- partial program → synthesizer
Conveying insight with partial programs

Partial programs can communicate programmer insight.

Suitable synthesis algorithm completes the mechanics.

Ideas exist for dealing with some key challenges:

What is the desired behavior?
How to capture early insight and refine it gradually?

End-user programming, API-level coding, too, may be decomposable into partial program and completion.
Credits

Students
- Gilad Arnold
- Chris Jones (Mozilla)
- Joel Galenson
- Casey Rodarmor
- Lexin Shan
- Armando Solar-Lezama (MIT)
- Liviu Tancau (Google)
- Nicholas Tung

Professors
- Bob Brayton
- Koushik Sen
- Sanjit Seshia

Other Collaborators
- Satish Chandra (IBM)
- Kemal Ebcioglu (IBM)
- Alan Mischenko (UCB)
- Rodric Rabbah (IBM)
- Mooly Sagiv (Tel Aviv)
- Vijay Saraswat (IBM)
- Vivek Sarkar (Rice)
- Martin Vechev (IBM)
- Eran Yahav (IBM)