A few synthesizers and their algorithms

Ras Bodik
University of California, Berkeley
aLisp

[Andre, Bhaskara, Russell, ... 2002]
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]
Fragment from the aLisp program

(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
check for conflicts
...
It’s synthesis from partial programs

correctness criterion

partial program

synthesize

merge

complete program

completion
SKETCH

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

sketch →
aLisp

- reward function
- aLisp partial program

hierarchical reinforcement learning

learnt choice functions
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.
SMARTedit*

[Lau, Wolfman, Domingos, Weld 2000]
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

```latex
@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 “)
...

---

%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},
}
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()
    ...
  }
}
```
Version Space for SMARTedit
SMARTedit*

demonstration(s) ➔ version space

macrot template ➔ algebra

completed macro(s) ➔ set of macro

parameters

input file ➔ run the macro

run the macro ➔ processed file
Denali

[Joshi, Nelson, Randall  PLDI 2002]

synthesis with automated theorem proving
Denali

Problem:
- scalable super-optimizer (previous ones: gen-and-test)

Solution:
- spec: the program, given as an instruction sequence
- process: write down instruction-equivalence axioms; rewrite the spec in all possible ways, using E-graphs; pick fastest one

Solver:
- no solver; new programs obtained by rewriting, as in the Simplify theorem prover (Nelson-Oppen)
- (given a candidate $P$, SAT solver computes $P$’s exec time)
\texttt{\textit{Input}}

\begin{verbatim}
proc byteswap4 : [ a : int ] -> int =
 var r : int \in
  r := 0 ;
  r<0> := a<3> ;
  r<1> := a<2> ;
  r<2> := a<1> ;
  r<3> := a<0> ;
 res := r ;
\end
\end{verbatim}

Figure 3: Envisioned program for 4-byte swap. $w<i>$ denotes byte $i$ of word $w$, that is, \texttt{selectb}(w,i). Our current prototype requires a parenthesized input syntax in the style of figure 6.
Output

```
// Register Map: {a=$16, r=$1, \res=$0, 0=$31}
byteswap4:
   exbl  $16, 1, $2   # 0, U1 ; $2 = 000y
   insbl $16, 3, $3   # 0, U0 ; $3 = z000
   nop             # 0
   nop             # 0
   insbl $2, 2, $2  # 1, U1 ; $2 = 0y00
   extbl $16, 3, $4 # 1, U0 ; $4 = 000w
   nop             # 1
   nop             # 1
   or   $4, $3, $3  # 2, L0 ; $3 = z00w
   extbl $16, 1, $4 # 2, U1 (unused)
   extbl $16, 2, $4 # 2, U0 ; $4 = 000x
   nop             # 2
   insbl $4, 1, $4  # 3, U0 ; $4 = 00x0
   or   $2, $3, $2  # 3, L0 ; $2 = zy0w
   nop             # 3
   nop             # 3
   or   $4, $2, $0  # 4, U0 ; $0 = zyxw
   ret ($26)       # 4, L0
   nop             # 4
   nop             # 4
.end byteswap4
```

Figure 4: Generated EV6 assembly program for four byte swap. The unused instruction is necessary: if it were a nop, the following extbl instruction would be scheduled on the wrong cluster.
Axioms: equiv of instruction sequences

\[(\forall x, y :: \text{add64}(x, y) = \text{add64}(y, x))\]
\[(\forall x, y, z :: \text{add64}(x, \text{add64}(y, z)) = \text{add64}(\text{add64}(x, y), z))\]
\[ (\forall x :: \text{add64}(x, 0) = x) \]

\[(\forall a, i, j, x :: i = j \]
\[ \forall \text{select}(\text{store}(a, i, x), j) = \text{select}(a, j)) \]

\[(\forall w, i :: \text{insbl}(w, i) = \text{selectb}(w, 0) \ll 8 \times i) \]
E-graph matching: find equiv programs

\[(\forall k, n :: k \times 2^n = k \ll n)\]
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)
Are these two also about partial programs?

- correctness criterion
- partial program
- synthesizer
- completion
- merge
- complete program
SMARTedit*

demonstration(s) → version space algebra → set of macro parameters

macro template → completed macro(s) ←

input file → run the macro → processed file
Prospector

have, want query

jungloid template + API

shortest path search

ranked jungloids

user selection

desired jungloid
Turn partial synthesis around?

correctness criterion

partial program

correctness check

angelic partial program

demonstrations

partial program

synthesizer

completion

synthesizer

angelic demonstration

synthesizer

completion