Combining Techniques of Automatic Program Synthesis
Investigation of Application of Gained Knowledge

Neil Crossley
Otto-Friedrich University, Bamberg

July 7, 2008
Summary

1 Introduction
   Overview of Program Development
   Importance of Specifications

2 Experiment Description
   Objective
   Approach

3 Results
   In Brief
   Conclusions

4 Current work
1. Introduction
   Overview of Program Development
   Importance of Specifications

2. Experiment Description
   Objective
   Approach

3. Results
   In Brief
   Conclusions

4. Current work
### Program Development Processes

1. **Analysis.**
2. **Specification.**
3. **Design.**
4. **Code.**
5. **Test.**
6. **Deploy.**
7. **Maintain.**
Program Development Processes

1. Analysis.
2. Specification.
3. Design.
5. Test.
6. Deploy.
7. Maintain.

Can we automate these processes?
Advantages

Benefits:

• Lower development costs/time.
• Increases availability of solution programs.
  • Lowers skills threshold
• Greater assistance with repetitious tasks for computer users.
Program Development Processes

1. Analysis.
2. Specification.
3. Design.
5. Test.
6. Deploy.
7. Maintain.
Outline

1. Introduction
   Overview of Program Development
   Importance of Specifications

2. Experiment Description
   Objective
   Approach

3. Results
   In Brief
   Conclusions

4. Current work
Automatic Program Synthesis

Assumed available

- A description of the program behaviour.
- Programming skills.
- Testing capability.
Automatic Program Synthesis

Assumed available
- A description of the program behaviour.
- Programming skills.
- Testing capability.

Important Questions
- How descriptive is the specification?
- How is the target program extracted out of the specification?
How descriptive is the specification? I

(In-)formal (natural) language

Expressing the target behaviour using a known syntax with known semantics.

Example (last)

Return the last element in the given list.

\[\text{last: } \text{List} \rightarrow \text{Item}\]
\[\text{last( [X] ) = X}\]
\[\text{last( [X|Y] ) = last(Y)}\]
How descriptive is the specification? II

Other: by examples
The characteristics are expressed using positive (or negative) examples.

Example (last)

last: List -> Item
last( [X] ) = X
last( [X,Y] ) = Y
last( [X,Y,Z] ) = Z

last: List -> Item
last( [X] ) = X
last( [X|Y] ) = last(Y)
## Formal vs Examples

### Formal
- Specification language as complex as programming language.
- High demands on the specifier.

### By examples
- Examples characterise the problem space of the target program.
- Is more intuitive for many users.
- Insufficient examples lead to either insufficient or no results.
How is the target program extracted out of the specification?

**Analytically**

Analysis techniques on the examples commonalities to directly infer a solution program.

- Deterministic
- Reduced problem space due to expression constraints

**Generate and test**

Systematically generate and evaluate potential solutions becomes the search for one or many correct solution programs.

- Non-deterministic
- Reduced problem space due to search constraints and operators.
Outline

1. Introduction
   - Overview of Program Development
   - Importance of Specifications

2. Experiment Description
   - Objective
   - Approach

3. Results
   - In Brief
   - Conclusions

4. Current work
Functional Program Synthesis Techniques

**IGOR2**

Analytical, induction driven approach.
- Emphasis on pattern searching techniques (i.e., anti-unification)
- Specifications in MAUDE
- Solutions are a constructor system

**ADATE**

“Automatic Design of Algorithms Through Evolution”
- Specifications in ML
- Solutions in ADATE-ML, subset of ML.
Objective

Origin

Jumpstart ADATE with quick inferences from IGOR2

Investigate

• Is it possible to use both ADATE and IGOR2 together?
• Can we improve ADATE search runs using IGOR2?
Outline

1 Introduction
   Overview of Program Development
   Importance of Specifications

2 Experiment Description
   Objective
   Approach

3 Results
   In Brief
   Conclusions

4 Current work
The situation involving IGOR2

- Insufficient examples.
- Results in an incomplete rule.
- Terminated incomplete.
Simulated Termination with Failure in IGOR2

Example

1. `switch( [] ) = []`
2. `switch( [X] ) = [X]`
3. `switch( [X,Y | XS] ) = [Y,X | switch(XS)]`
Simulated Termination with Failure in IGOR2

Example

1. switch( [] ) = []
2. switch( [X] ) = [X]
3. switch( [X,Y | XS] ) = Z

Show workings
ADATE Specifications

- Unassisted.
- Redefined.
- Top level restriction.
- Additional atomic functions.
- Inner functions.
Unassisted Specification

Definition

A normal ADATE specification, which solves the problem without aid from IGOR2.

- Only necessary help functions (if any).
- Many examples.
- Templates for other specifications.
Unassisted Specification

Example

[type declarations]
[help functions]

fun f ( ... ) : myType = raise D1

fun main ( ... ) : myType = f ( ... )
The function definition of $f$ is rewritten to implement the rules known to IGOR2.

- This $f$ is inserted into starting population.
Redefined Specification

Example

[type declarations]
[help functions]

fun f Xs =
  case Xs of
    nil => Xs
  | cons( Y1, Y2 ) =>
   case Y2 of
     nil => Xs
   | cons( W1, W2 ) => raise D1

fun main( Xs ) = f Xs
Top Level Restriction

Definition

The function definition of main is rewritten to implement the rules known to IGOR, using $f$ to solve the unknown rule.

- The task becomes much smaller.
Restricted Specification

Example

[**type declarations**]
[**help functions**]

```haskell
fun f Xs = raise D1

fun main ( Xs ) =
    case Xs of
      null => Xs
    | cons ( Y1, Y2 ) =>
      case Y2 of
        null => Xs
      | cons ( W1, W2 ) => f Xs
```
Additional atomic function’s

Definition

(Some) additional discovered functions are implemented in the specification.

- Used only as needed.
Specification with an additional atomic function

Example

[type declarations]
[help functions]
fun last (aList : list) : int = case aList of
  nil ⇒ raise D1
| cons (aH, aT) ⇒ (case aT of
    nil ⇒ aH
  | cons (aaH, aaT) ⇒ lastFun (aT))

fun f Xs = raise D1

fun main (Xs) = f Xs
Inner functions

Definition

(Some) additional discovered functions are implemented as a part of the function $f$.

- Used only as needed.
Specification with an inner function

Example

[type declarations]
[help functions]

fun f Xs =
  let
  last ( ... ) = ...
  in
  raise D1
end

fun main( Xs ) = f Xs
Outline

1. Introduction
   Overview of Program Development
   Importance of Specifications

2. Experiment Description
   Objective
   Approach

3. Results
   In Brief
   Conclusions

4. Current work
### Success

<table>
<thead>
<tr>
<th>Problem</th>
<th>Type</th>
<th>Time</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>switch</td>
<td>Unassisted</td>
<td>4.34</td>
<td>302</td>
</tr>
<tr>
<td></td>
<td>Restricted</td>
<td>0.47</td>
<td>344</td>
</tr>
<tr>
<td></td>
<td>Redefined</td>
<td>3.96</td>
<td>302</td>
</tr>
<tr>
<td>sort</td>
<td>Unassisted</td>
<td>457.99</td>
<td>2487</td>
</tr>
<tr>
<td></td>
<td>Restricted</td>
<td>225.13</td>
<td>2849</td>
</tr>
<tr>
<td>swap</td>
<td>Unassisted</td>
<td>292.05</td>
<td>1076</td>
</tr>
<tr>
<td></td>
<td>Restricted</td>
<td>41.43</td>
<td>685</td>
</tr>
<tr>
<td></td>
<td>Restricted and functions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lasts</td>
<td>Unassisted</td>
<td>260.34</td>
<td>987</td>
</tr>
<tr>
<td></td>
<td>Restricted</td>
<td>6.25</td>
<td>1116</td>
</tr>
<tr>
<td>shiftR</td>
<td>Unassisted</td>
<td>8.85</td>
<td>239</td>
</tr>
<tr>
<td></td>
<td>Redefined, functions</td>
<td>1.79</td>
<td>239</td>
</tr>
<tr>
<td>shiftL</td>
<td>Unassisted</td>
<td>4.17</td>
<td>221</td>
</tr>
<tr>
<td></td>
<td>Restricted</td>
<td>0.61</td>
<td>281</td>
</tr>
</tbody>
</table>
Failure

<table>
<thead>
<tr>
<th>Problem</th>
<th>Type</th>
<th>Time</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>Unassisted</td>
<td>7.81</td>
<td>176</td>
</tr>
<tr>
<td></td>
<td>Restricted</td>
<td>18.37</td>
<td>240</td>
</tr>
</tbody>
</table>

Every other combination resulted in:

- the same results, only delayed.
- drastically worse times.
Failure

Reasons in short

Redefined
- Too many possible variations to explore.

Restriction
- Contrary to ADATEs search techniques.

Additional atomic functions
- Expands the search space right from the start.

Inner functions
- Too many possible variations to explore.
Outline

1. Introduction
   Overview of Program Development
   Importance of Specifications

2. Experiment Description
   Objective
   Approach

3. Results
   In Brief
   Conclusions

4. Current work
Conclusions

**Project**
- Combination is successful.
- Criteria for the successful method is unknown.

**ADATE**
- Needs better support for “background knowledge”.

Master thesis

Questions

• When is which method appropriate?
• Unknown categories of problems?
• Avoid trial-and-error?

Aim

The method of inserting IGOR2 knowledge into ADATE which guarantees no worse results.
Master thesis

Under development

- More problems.
- More complex. (more non-iterative and of varied O-complexities)
- More analysis. (ie, rules count, complexity order)
sort( [] ) = []
sort( [X|Y] ) = [ min([X|Y]) | sort( butmin([X|Y]) ) ]

min( [X] ) = X
min( [X,Y|Z] ) = if X < Y then min( [X|Z] ) else min( [Y|Z] )

butmin( [X] ) = []
butmin( [X,Y|Z] ) = if X < Y then [Y| butmin( [X|Z] )] else [X| butmin( [Y|Z] )]
fun f Xs =
case Xs of
  nil  => Xs
| cons(V12CE, V12CF) =>
case f(V12CF) of
  V21C2E =>
case V21C2E of
    nil  => Xs
| cons(VA355, VA356) =>
case (VA355 < V12CE) of
  false  => cons(V12CE, V21C2E)
| true  => cons(VA355, f(cons(V12CE, VA356)))
fun f( (alnt, aList) : (int * list) ) : list = raise D1

fun sort (Xs : list) : list = case Xs of
  nill => Xs
  | cons( Y1, Y2 ) => f( Y1, Y2 )

fun main( Xs : list ) : list = sort Xs
shiftR Solution

\[
\begin{align*}
\text{shiftR}(\ [\] ) &= \ [\] \\
\text{shiftR}(\ [X|Y] ) &= \ [\text{last}(X|Y) \mid \text{init}(X|Y)] \\
\text{last}(\ [X] ) &= X \\
\text{last}(\ [X|Y] ) &= \text{last}(Y) \\
\text{init}(\ [X] ) &= \ [\] \\
\text{init}(\ [X|Y] ) &= [X \mid \text{init}(Y)]
\end{align*}
\]
Example

1. \( \text{shiftR}( \[] ) = \[] \)
2. \( \text{shiftR}( \[W] ) = [W] \)
3. \( \text{shiftR}( \[W,X] ) = [X,W] \)
4. \( \text{shiftR}( \[W,X,Y] ) = [Y,W,X] \)
5. \( \text{shiftR}( \[W,X,Y,Z] ) = [Z,W,X,Y] \)

Solution
Input Examples: shiftR

Example

1. shiftR( [] ) = []
2. shiftR( [W] ) = [W]
3. shiftR( [W,X] ) = [X,W]
4. shiftR( [W,X,Y] ) = [Y,W,X]

Solution

1. shiftR( [] ) = []
2. shiftR( [A|B] ) = C
Input Examples: shiftR

Example

1. $\text{shiftR}(\quad) = \quad$
2. $\text{shiftR}(\text{[W]}\quad) = \text{[W]}$
3. $\text{shiftR}(\text{[W,X]}\quad) = \text{[X,W]}$
4. $\text{shiftR}(\text{[W,X,Y]}\quad) = \text{[Y,W,X]}$
5. $\text{shiftR}(\text{[W,X,Y,Z]}\quad) = \text{[Z,W,X,Y]}$

Solution

1. $\text{shiftR}(\quad) = \quad$
2. $\text{shiftR}(\text{[A|B]}\quad) = \text{C}$
**Input Examples: shiftR**

### Example

1. \( \text{shiftR( [] )} = [] \)
2. \( \text{shiftR( [W] )} = [W] \)
3. \( \text{shiftR( [W,X] )} = [X,W] \)

### Solution

1. \( \text{shiftR( [] )} = [] \)
2. \( \text{shiftR( [A\mid B] )} = [\text{sub1}[A\mid B] | \text{sub2}[A\mid B]] \)
Input Examples: sub1

Example

1. $\text{sub1}( [W] ) = W$
2. $\text{sub1}( [W,X] ) = X$
3. $\text{sub1}( [W,X,Y] ) = Y$
4. $\text{sub1}( [W,X,Y,Z] ) = Z$

Solution
Input Examples: sub1

Example

1. sub1([W]) = W
2. sub1([W,X]) = X
3. sub1([W,X,Y]) = Y
4. sub1([W,X,Y,Z]) = Z

Solution

1. sub1([A]) = A
2. sub1([A|B]) = C
Input Examples: \text{sub1}

\begin{itemize}
  \item \text{Example}
    \begin{enumerate}
      \item \text{sub1}( [W] ) = W
      \item \text{sub1}( [W,X] ) = X
      \item \text{sub1}( [W,X,Y] ) = Y
      \item \text{sub1}( [W,X,Y,Z] ) = Z
    \end{enumerate}
  \item \text{Solution}
    \begin{enumerate}
      \item \text{sub1}( [A] ) = A
      \item \text{sub1}( [A|B] ) = C
    \end{enumerate}
\end{itemize}
Input Examples: sub1

Example
1. \( \text{sub1}( [W] ) = W \)
2. \( \text{sub1}( [W,X] ) = X \)
3. \( \text{sub1}( [W,X,Y] ) = Y \)
4. \( \text{sub1}( [W,X,Y,Z] ) = Z \)

Solution
1. \( \text{sub1}( [A] ) = A \)
2. \( \text{sub1}( [A|B] ) = \text{sub1}[B] \)
### Example

1. \( \text{sub1( } [W] \text{ ) } = W \)
2. \( \text{sub1( } [W,X] \text{ ) } = X \)
3. \( \text{sub1( } [W,X,Y] \text{ ) } = Y \)
4. \( \text{sub1( } [W,X,Y,Z] \text{ ) } = Z \)

### Solution

1. \( \text{last( } [A] \text{ ) } = A \)
2. \( \text{last( } [A|B] \text{ ) } = \text{last}[B] \)
Input Examples: sub2

Example

① sub2( [W] ) = []
② sub2( [W,X] ) = [W]
③ sub2( [W,X,Y] ) = [W,X]
④ sub2( [W,X,Y,Z] ) = [W,X,Y]
Example

1. $\text{sub2}( [W] ) = []$
2. $\text{sub2}( [W, X] ) = [W]$
3. $\text{sub2}( [W, X, Y] ) = [W, X]$
4. $\text{sub2}( [W, X, Y, Z] ) = [W, X, Y]$

Solution

1. $\text{sub2}( [A] ) = []$
2. $\text{sub2}( [A|B] ) = [C]$
### Example

1. \( \text{sub2( [W] )} = [] \)
2. \( \text{sub2( [W,X] )} = [W] \)
3. \( \text{sub2( [W,X,Y] )} = [W,X] \)
4. \( \text{sub2( [W,X,Y,Z] )} = [W,X,Y] \)

### Solution

1. \( \text{sub2( [A] )} = [] \)
2. \( \text{sub2( [A|B] )} = [C] \)
**Example**

1. \( \text{sub2}( [W] ) = [] \)
2. \( \text{sub2}( [W,X] ) = [W] \)
3. \( \text{sub2}( [W,X,Y] ) = [W,X] \)
4. \( \text{sub2}( [W,X,Y,Z] ) = [W,X,Y] \)

**Solution**

1. \( \text{sub2}( [A] ) = [] \)
2. \( \text{sub2}( [A|B] ) = \text{sub2}[B] \)
Example

1. \( \text{sub2}( [W] ) = [] \)
2. \( \text{sub2}( [W,X] ) = [W] \)
3. \( \text{sub2}( [W,X,Y] ) = [W,X] \)
4. \( \text{sub2}( [W,X,Y,Z] ) = [W,X,Y] \)

Solution

1. \( \text{init}( [A] ) = [] \)
2. \( \text{last}( [A|B] ) = \text{last}[B] \)
Input Examples: shiftR

Example

1. $\text{shiftR( } [\ ] \text{ )} = [\ ]$
2. $\text{shiftR( } [W] \text{ )} = [W]$
3. $\text{shiftR( } [W,X] \text{ )} = [X,W]$
4. $\text{shiftR( } [W,X,Y] \text{ )} = [Y,W,X]$
5. $\text{shiftR( } [W,X,Y,Z] \text{ )} = [Z,W,X,Y]$

Solution

1. $\text{shiftR( } [\ ] \text{ )} = [\ ]$
2. $\text{shiftR( } [A|B] \text{ )} = [\text{last}[A|B]|\text{init}[A|B]]$
3. $\text{last( } [A] \text{ )} = A$
4. $\text{last( } [A|B] \text{ )} = \text{last}[B]$
5. $\text{init( } [A] \text{ )} = [\ ]$
6. $\text{last( } [A|B] \text{ )} = \text{last}[B]$