Combining Techniques of Automatic Program Synthesis

Investigation of Application of Gained Knowledge

Neil Crossley

Otto-Friedrich University, Bamberg

July 7, 2008

Summary

Outline

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.
4. Code.
5. Test.
6. Deploy.
7. Maintain.

Can we automate these processes?

Advantages

Benefits:

- Lower development costs/time.
- Increases availability of solution programs.
    - Lowers skills threshhold
- Greater assistance with repetitious tasks for computer users.

Program Development Processes

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

Outline

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.

```
last: List -> Item
last( [X] ) = X
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

Functional Program Synthesis Techniques

IGOR2

Analytical, induction driven approach.

- Emphasis on pattern searching techniques (ie., 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

Assumed Conditions

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)]
```

Show workings

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 ( ... )
```

Redefined

Definition

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

fun main( Xs ) = f Xs
```

▸ Show IGOR2 solution

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]

fun f Xs = raise D1

fun main( Xs ) =
 case Xs of
    nill => Xs
 | cons( Y1, Y2 ) =>
 case Y2 of
    nill => Xs
 | cons( W1, W2 ) => f Xs
```

▸ Show IGOR2 solution

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
   nill => raise D1
 | cons ( aH, aT ) => (case aT of
    nill => 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

Success

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

Failure

Problem	Type	Time	Efficiency
insert	Unassisted	7.81	176
	Restricted	18.37	240

Every other combination resulted in:

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

▸ Show sort

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

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
   nill => Xs
 | cons( V12CE, V12CF ) =>
 case f( V12CF ) of
   V21C2E =>
 case V21C2E of
   nill => Xs
 | cons( VA355, VA356 ) =>
 case ( VA355 < V12CE ) of
   false => cons( V12CE, V21C2E )
 | true => cons( VA355, f( cons( V12CE, VA356 ) )
```

```
fun f( (aInt, 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

```
shiftR( [] ) = [] shiftR( [X|Y] ) = [last(X|Y) | init(X|Y)]
last( [X] ) = X last( [X|Y] ) = last(Y)
init( [X] ) = [] init( [X|Y] ) = [X| init(Y)]
```

Input Examples: shiftR

Example

1. `shiftR( [] ) = []`
2. `shiftR( [W] ) = [W]`
3. `shiftR( [W,X] ) = [X,W]`
4. `shiftR( [W,X,Y] ) = [Y,W,X]`
5. `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]`
5. `shiftR( [W,X,Y,Z] ) = [Z,W,X,Y]`

Solution

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

Input Examples: shiftR

Example

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

Solution

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

Input Examples: shiftR

Example

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

Solution

1. shiftR([]) = []
2. shiftR([A|B]) = [sub1[A|B]|sub2[A|B]]

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

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. $\text{sub1}(\ [A]\ ) = A$
2. $\text{sub1}(\ [A|B]\ ) = C$

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: 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]) $=$ sub1[B]

Input Examples: last

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. last([A]) = A
2. last([A|B]) = 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]`

Solution

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]`

Solution

1. sub2([A]) = []
2. sub2([A|B]) = [C]

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]`

Solution

1. sub2([A]) = []
2. sub2([A|B]) = [C]

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]`

Solution

1. sub2([A]) = []
2. sub2([A|B]) = sub2[B]

Input Examples: initial

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]`

Solution

1. init([A]) = []
2. last([A|B]) = last[B]

Input Examples: shiftR

Example

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

Solution

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