Cognitive Systems

Project: Automated functional programming

Knowledge gains for ADATE using IGOR2

Neil Crossley
Summer Semester 2007

Contents

1 Background

Writing a program to solve a problem laid out in a specification implies the application of programming skills. As long as these skills are provided by humans, the threshold to access problem-specific programs will remain too high for many computer users. It is the aim of research into automatic program induction to automate program development, removing as much of the human component from development as possible, ultimately entirely. This field promises to alleviate computer users from some of the necessary requirements to perform many programming tasks [1].

This project report focuses on two opposing methods to function synthesis and investigates their potentially complimentary nature. They both utilise specifications that define the behaviour of the target program through examples. The problem is reduced to discovering a function that has correct output for the given input examples. One method is to take the non-deterministic, search intensive approach: to generate, test and refine functions according to a strategy, which is often evolutionary in nature. The other method is deterministic, analysing the given data to directly deduce the target function. Our chosen exemplars for this project are ADATE and IGOR2, respectively non-deterministic and deterministic. ADATE is a search intensive approach, inferring new functions according to its search operators and using a selection and management strategy with evolutionary tendencies [4]. IGOR2 analyses the given information much more thoroughly and closely, taking the more deductive approach in its search for patterns in the examples [2]. Because ADATE has the larger problem space due to a comparative lack of restrictions of both specifications and the used programming language, it is much more capable of inferring programs than IGOR2, at the cost of greater search times. The project aimed to test the hypothesis by means of ADATE and IGOR2, that a nondeterministic method of automatic program construction can consistently benefit from the results of an analytical, deterministic approach.

1.1 ADATE

In this section we shall outline the learning techniques of ADATE to illustrate its strengths upon which its success undoubtedly lies. We shall start with the specification of a problem for ADATE, which should set the scene as we throw light onto the inner workings of ADATE, emphasising the affects on its search bias. With this background, we propose our hypothesis for enhancing ADATE specifications to achieve an improved performance, leading onto an introduction of IGOR2.

ADATE interprets ADATE specification files, which define important characteristics of the problem and additional parameters for configuring an ADATE run. The

specification consists of the following:

- A function f written in ADATE-ML
- A set of data types
- A set of primitive functions written in ADATE-ML
- A set of sample inputs
- An evaluation function

The function declaration of all inferred programs use the function declaration of f, differing only in their respective bodies. The body consists only those structures available in ADATE-ML (a subset of the ML language), the given data types and the given primitive functions. ADATE creates new functions by performing transformations upon an existing function [4].

The sample inputs and the evaluation function fulfil the role of training data typically used in other evolutionary computation approaches [1], which can be as simple as a set of input-output pairs. Input-output pairs can only give a binary evaluation (correct or incorrect), which the evaluation function is also able to produce. However, because it returns a grade for the output, it is possible to reward referred functions for partial correctness. Thus, ADATE can more easily recognise changes that result in functions with fewer errors, with the intention of leading sooner to correct functions. Unfortunately, more finely grained evaluations can also result in more searches in dead-ends and require an understanding of the problem itself, in order to apply at least one heuristic successfully. Considering and that our hypothetical ADATE specifier has neither this knowledge nor the wish to write worse ADATE specifications, our research restricted itself to input-output pairs.

In addition to the evaluation function provided by the specifier, ADATE employs measurements of syntactic complexity and time complexity for further evaluation [4, 5]. These evaluations are built into ADATE and total six – three for time complexity and three for syntactic complexity. All inferred functions are stored in nine grids, each with a different time and syntactic complexity on the axis, providing ADATE with a fairly detailed overview of its population of inferred programs. Using this overview, ADATE is better able to select inferred functions for expansion based on its search heuristic, preferring smaller, quicker functions.

The search operators themselves are transformations, used in reproduction to generate new individuals [4, 5]. ADATE has various types of atomic transformations, identifiable for enabling ADATE to search the problem space of functional programs. These transformations include:

1. R: Replacement changes part of the individual with new expressions
2. REQ: similar to R, it is guaranteed that the new individual does not have a worse evaluation due to the transformation

3. ABST: This transformation abstracts an expression in an individual by replacing the expression with a function call to a new function in a let-block. The body of this function is the replaced expression.

4. CASE-DIST: This transformation moves a function call surrounding a case expression into each of the case blocks.

5. EMP: Through various embedding techniques, the arguments and/or their types of inner functions are altered.

6. CROSSOVER: A compound transformation from one individual is applied to another.

ADATE does not restrict itself to a single atomic transposition for the creation of new children, but constructs and applies compound transformations instead. A compound transformation is as number of the atomic transformations, which are constructed based on various factors at that stage of the search process. These factors are similar to the heuristics common to all non-deterministic search strategies, such as favouring larger changes the further it is from the goal. Once an individual has been selected for expansion, children are created by applying different compound transformations. Thus, all individuals are characterised by a series of transformations, which, if reversed, would result in the original function. Once generated, an individual is evaluated for insertion into the rest of the population.

With only the minimum necessary background knowledge, such as necessary data types and atomic functions, ADATE is able to find reasonable solutions given time. Additional background knowledge can reduce the required transformations to infer correct solutions, which can also reduce the search time. However, additional background knowledge exacts deeper insights into the problem on behalf of the problem specifier. One would assume, the user would have already detailed their complete knowledge of the problem in the specification, so expecting more would be unreasonable in this process. Without increasing the expected human knowledge requirements, could we introduce additional background knowledge into the problem specifications, resulting in better ADATE searches?

1.2 IGOR2

We acknowledged at this point, the need to apply a second automatic program synthesis method. This process, which would analyse the problem before ADATE, should terminate with results that ideally improve the knowledge about the problem, or at the least not make it worse. We also considered the restriction, that the total search time using assistance should be shorter than or as long as the respective "uninformed" ADATE run. While desireable, we decided it would not be a requirement.

IGOR2 proved to be a worthy candidate as ADATEs preprocessor, being also a method for automatically inducing recursive programs from examples[2]. Thus, the input requirements to solve a problem are very similar to ADATEs own (as opposed to formal input-output specifications [1], for example). More importantly, IGOR2s

capability to improve knowledge about the problem corresponded to our expectations. Employing a term rewriting system, IGOR2 is an analytical approach which detects recurrences and regularities in the examples and generalises them into recursive functions[2]. A representation of the the problem as a set of rules is defined, similar to function declaration using pattern matching in functional programming languages. Each rule has a constructor, known as the left hand side (lhs), and a rewriting relation, the right hand side (rhs). The search process involves finding the smallest set of rules with the most specific possible patterns and the most general rhss[2]. The theory was that IGOR2 would assist ADATE by inducing as many rules as possible and ADATE would benefit from these new, previously unknown and incomplete rules.

Let us consider a simple problem, a function that takes a list and switches the element at every second position with the preceding element. Given the appropriate input data, IGOR2 would conclude the following rules:

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

The first two rules deal with the empty list and all lists containing one term as parameters, making no changes as appropriate. The rhs of the third rule remaps the input terms X and Y and the recursive call on the leftover list XS ensures all pairs of terms are switched. Not a spectacular example, it demonstrates nonetheless the results of a successful search. We want to enhance the ADATE specifications with these inferred rules, supporting our hypothesis for preprocessing the input data in order to gain additional information. The fact that IGOR2, like other strongly analytical approaches, can only successfully solve a more restricted set of problems due to analysis restraints, does not reduce its suitability. When the data is insufficient for the search to completely infer a program, the rules themselves still have validity. ADATE can replace unknown terms with expressions and known terms, not requiring the strict input data, and supposedly arrive at results in a shorter period of time. With the intention of benefiting from this behaviour, the project was planned to investigate, how this analytical preprocessing might be utilised for maximum results.

2 Combining ADATE and IGOR2

The experiment itself consisted of running ADATE with a variety of specifications aimed to inferring solutions for the same problem in the same test environment. The ADATE specifications were compiled using an outdated version of MLTON and executed on a dual-processor Pentium 4 3.0Ghz running Ubuntu 7.10, a Linux distribution. To reduce possible variations in the testing environment, all the various specifications for the same problem were executed during the same computer sitting.

2.1 Assistance Possibilities

Because we wanted to test the effects of providing different combinations of background information, an "unassisted" ADATE specification was created for each problem fulfilling the minimum requirements for solving the it (i.e., specifying lists or atomic functions such as "addition" or "less than"). The input-output examples also included all those of the simplest complexities, not to overly comply with the pedagogical recommendations for ADATE[3], but to satisfy the minimum restraints of IGOR2.

We derived for each problem the background knowledge from the results generated by IGOR2 for the respective problem. Initially, we allowed IGOR2 to completely infer a correct, complete solution and then undid the last term replacement. In the case of our `switch` example, this results in the following:

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

where Z is an arbitrarily named unknown term. Thus, in every case a new partial function was derived.

Then we created new specifications through feasible alterations of the base specifications by translating the new background knowledge into ADATE-SML (the programming language employed by ADATE) and inserting it. A number of methods for adding the knowledge were available and have been classified as follows:

Redefined: The partial function derived using IGOR2s assistance replaced the function definition of f. A `switch` specification was altered accordingly with the replacement of f:

Listing 1: Function f is redefined

```
fun f( Xs : list ) : list =  case Xs of
    nill => Xs
  | cons( Y1, Y2 ) =>
      case Y2 of
        nill => Xs
    | cons( W1, W2 ) => raise D1

fun main( Xs : list ) : list = f Xs
```

Problem Abstraction: IGOR2 generated a partial solution with exactly one unknown. In this case, we redefined the problem such that ADATE only had to solve this one unknown. This was done by supplying ADATE with the partial solution as an atomic function, replacing the unknown predicate with a function call to f. A `switch` specification received the following changes accordingly:

Listing 2: f now solves only part of the problem

```
fun f( (val1, val2, Xs) : (int * int * list ) ): list =
```

```
  raise D1

fun switch( Xs : list ) : list =  case Xs of
    nill => Xs
  | cons( Y1, Y2 ) =>
      case Y2 of
        nill => Xs
    | cons( W1, W2 ) => f( Y1, W1, W2 )

fun main( Xs : list ) : list = switch Xs
```

Atomic functions: In its attempt to solve a problem, IGOR2 can infer and use sub-functions. Unlike the previous methods, the inferred and incomplete main function definition is not employed in the ADATE specification.

Listing 3: ADATE can now use, but not change, the function last.

```
fun last ( Xs : list ) : int =
  case Xs of
    nill => (raise D1)
  | cons( aH, aT ) =>
      case aT of
        nill => aH
      | cons( aaH, aaT ) =>
          case aaT of nill => aaH
        | cons( V1F0E, V1F0F ) => last( aaT )

fun f ( X : list ) : list= raise D1

fun main( X : list ) : list = f X
```

Inner functions: ADATE-SML allows for nested function definitions and the function f is redefined to include the some or all of the IGOR2 rules as an inner function.

Listing 4: ADATE can now use or change the function last.

```
fun f ( X : list ) : list =
  let
    fun last ( Xs ) =
      case Xs of
        nill => (raise D1)
      | cons( aH, aT ) =>
          case aT of
            nill => aH
          | cons( aaH, aaT ) =>
              case aaT of nill => aaH
```

```
                | cons( V1F0E, V1F0F ) => last( aaT )
    in
      raise D1
    end

fun main( X : list ) : list = f X
```

2.2 Problems

The following problems were used in the experiment to test our theory. A specification of each problem was written for IGOR2 and ADATE, and based on the results from IGOR2, additional ADATE specifications were created according to the methods outlined in 2.1.

insert(X, Y) = Z iff X is a list of elements sorted in an ascending order and Z is a list of elements X + Y sorted in an ascending order.

switch(X) = Y iff the list Y can be obtained from the list X by swapping every element on an odd index in X with the element with the next incremental even index.

sort(X) = Y iff the list Y is a permutation of X with all elements sorted in increasing order.

swap (X) = Y iff the list Y is identical to the list X, except that the first and last element are swapped in around in Y.

IGOR2 inferred one atomic function `last` and inferred that the solution consists of two function that recursively call each other:

```
1. swap( [] ) = []
2. swap( [X|XS] ) = [ last([X|XS]) | sub([X|XS]) ]

3. sub ([X] ) = []
4. sub ( [X,Y|XS] ) = swap( [X | sub([Y|XS])] )
```

The four rules were indeed used, but the complete right hand side of the fourth was replaced with an undefined term. Additionally, the function `last` was also inferred using ADATE.

lasts(X) = Y iff X is a list of lists and Y is a list containing the last element of each list in X in the order those lists appear in X.

shiftR(X) = Y iff the list Y is identical to the list X, except that the last element in X is on the first position in Y and all other elements are shifted one position to the right.

The assistance that proved most successful was atomic function generation. IGOR2 inferred that two atomic functions were necessary and assigned examples

defining examples. Although IGOR2 could have inferred them, these functions were inferred using ADATE and have been named `last` and `init`.

shiftL(X) = Y iff the list Y is identical to the list X, except that the first element in X is on the last position in Y and all other elements are shifted one position to the left.

2.3 Results

The results were ascertained by analysing the log files produced to document an ADATE run. To effectively compare the specifications we evaluated each according to the time taken to generate the most correct functions. Because ADATE infers many incorrect programs in the search process, we restricted our focus to those programs that:

- were tested by ADATE against the complete set of given training examples.
- terminated for each training example.
- generated a correct output for each training example .

This allowed us to achieve a meaningful overview of the performance of the specifications. While an analysis of the inferred programs with poorer performce provides insights into the learning process, it is outside of our scope.

Nonetheless, ADATE generates a very complete overview of inferred programs in the log files. For the analysis of the ADATE runs we needed only the following information:

- the elapsed **time** since the start of the search until the creation of the program.
- the breakdown of the results the function produced for the examples, which in our case is the number of results evaluated as correct, incorrect or timed-out. Due to our accepted evaluation restrictions, we filtered out all inferred functions which did not attain 100% correct results with the test examples.
- an ADATE time evaluation of the inferred function. This is the equivalent to the execution time taken of the function for all the test examples. We refer to this as its runtime **efficiency**.

Because all the specifications designed to solve the same problem included exactly the same examples, we could now compare the respective runs with each other. Table 1 is a summary comparing the most efficient solutions of the unassisted specifications with those of the best specification for the same problem. Included is the problem name, the specification type (either unassisted or the type of assistance), the creation time of the solution, the runtime efficiency.

In the Appendix are all the tables with the complete listing of the results from which this summary was extracted. Additionally, the column "Improvement" is included which is a procentual running comparison of the efficiency of solutions for a given specification. With this, we can more easily collate the improved efficency of solutions with the time taken to generate them.

Table 1: Summary of the most efficient solutions

Problem	Type	Solution #	Time	Efficiency	Recursive calls
insert	Unassisted	2 of 2	7.81	176	1
	Abstraction	1 of 2	18.37	240	1
switch	Unassisted	1 of 1	4.34	302	1
	Abstraction	1 of 1	0.47	344	1
	Redefined	1 of 1	3.96	302	1
sort	Unassisted	5 of 6	457.99	2487	3
	Abstraction	5 of 6	225.13	2849	2
swap	Unassisted	4 of 4	292.05	1076	2
	Abstraction and functions	4 of 4	41.43	685	1
lasts	Unassisted	3 of 4	260.34	987	2
	Abstraction	2 of 2	6.25	1116	2
shiftR	Unassisted	1 of 1	8.85	239	1
	Redefined, functions	1 of 1	1.79	239	0
shiftL	Unassisted	1 of 1	4.17	221	1
	Abstraction	1 of 2	0.61	281	1

2.4 Analysis

In this section, we outline the observable trends in the data, also making reference to impressions gained the unsuccessful specifications. Thus, having clearly explained the limits to the success, we are able to outline an outlook for further development in this field and its potential rewards.

Attention should be drawn to the uniformly quicker inference times in achieved by the assisted ADATE specifications in Table 1, with the noteable exception of `insert`, which can be consider a resounding failure. Two assisted specifications (`swap` and `shirftR`) resulted in better results that were also inferred sooner, whereas the remaining assisted specifications produced results between 14% and 27% less efficient than their unassisted counterparts. All in all, one could summarise, that this some relatively small comparative inefficiency is more than compensated by the drastically reduced search time, just over 41 times quicker in the case of `lasts`. One can easily summarise, gaining additional knowledge from the provided information, even using automatic analytical methods such as IGOR2, assists ADATE in producing acceptable results quicker than otherwise would be the case.

Howeve, the method is not yet complete. We have not included the poor results the other types of assistance produced with ADATE. For completeness of the experiment, each method of injecting the gain knowledge from IGOR2 into ADATE was carried out and analysed. Without exception, the combinations not listed above in this report produced only to worse results. There were two types of runs: either the ADATE produced results exactly like those from the unassisted specification but with a (in

some cases considerable) delay, or it produced in comparison relatively slowly relatively inefficient solutions. From these runs we are aware, that the new knowledge can in fact hinder the search process. In the case of `insert`, none of the applicable methods proved successful at all and the best of them is presented.

An analysis of the ADATE runs that failed to confirm our hypothesis confirmed the existence of several trends within the data, on which we base the following observations explaining their respective failures:

Redefined: Redefining the function f resulted in failure in two observable ways. In the first, ADATE quickly evaluated the given function so poorly, that it inferred the empty function and from that point on never again used the given information. This resulted in a run similar to that produced by the unassisted specification. The second type of failure was caused because ADATE attempted to test the many different possibilities which could be inferred from the start program. Its preference for smaller and quicker functions naturally meant ADATE began by reducing the function f, and worked its way from there out. Thus, this attempt to assist ADATE either gave it information which either was completely unusable or exposed an unnecessarily large search space.

Problem Abstraction: This method guaranteed the preservation of the gained knowledge, but increased the overhead of the search process accordingly. An increased execution time for each example, even for the most simple inferred functions, inhibits ADATEs ability to quickly infer and evaluate functions. Additionally, one can only apply this at all if there is only a single undefined term, which was only the case due to our assumptions made in the derivation of the knowledge.

Atomic functions: All attempts at the problem `sort` with various combinations of assistance using the atomic functions `min` (which returns the smallest element in a list) and `butmin` (which returns a list without its smallest element) could not produce satisfactory results compareable with the unassisted ADATE runs. It was conclusive that atomic functions should be provided when necessary, but could possibly increase both the search space and the execution time (and thus search time) to the detriment of the search.

Inner functions: Worse than just redefining the function f, attempting to provide nested functions drastically increased the search space and severely inhibited the performance of the ADATE search.

We would like to conclude this report by stating that automatic, search intensive program synthesis methods can reap benefits from automatic preprocessing, such as analytical program synthesis methods. However, there is no single method to apply the gain that guaranteed a benefit or even similar results are always obtained. Further investigation is necessary to search for the necessary clues by which the additional knowledge can be correctly applied.

2.5 Conclusions

The results of our investigation shows that the application of analytical orientated function synthesis to enhance the knowledge of the problem results in benefits for a non-deterministic search algorithm. In our examples the method of "Problem Abstraction" achieved the most success. However, because this was not resounding in every case, in some even worse than no assistance, one either needs to hazard such consequences, or develop a system to apply the correct method of assistance for a given problem. Further investigation is needed to explain why a particular method is so successful to the ends that characteristics of the problem could conclusively identify this relationship.

It is also possible to optimise the collaboration between ADATE and IGOR2 by fine tuning the behaviour of ADATE. For example, it is counter-intuitive that we can redefine the function `f` to include knowledge we know to be correct, yet ADATE will quickly ignore this and start from scratch if its performance based heuristics decide that it is supposedly better (see critic above to the method "Redefined"). Protecting such improvements using "Problem Abstraction" obviously works and attributes to its success, but this is only a work-around with its own restrictions. It would be better if ADATE would assume the given function is the starting point and search only from their - however, ADATE currently regresses somewhat too quickly, losing this valuable and timesaving knowledge. From the perspective of providing user readable functions, it could be beneficial if ADATE provided mechanisms to influence the selection (or evaluation) of inferred programs based on their structure. Not to mention, the assistance from IGOR2 would not run counter to the purely size/performance selection mechanisms and could thus possibly lead more quickly to conforming and correct functions.

As it stands, the combination of the two opposing techniques brings benefits to the performance of ADATE, however a proven method to achieve consistently such results is not yet available and requires further investigation. These insights should bring this field one step closed to the unified application of completely different yet complementary approaches to synthesis recursive functional programs.

References

[1] A. W. Biermann, G. Guiho, and Y. Kodratoff. An overview of automatic program construction techniques. In A. W. Biermann, G. Guiho, and Y. Kodratoff, editors, *Automatic Program Construction Techniques*, pages 3–30. Macmillan Publishing Company, New York, 1984.

[2] Emanuel Kitzelmann. Data-driven induction of recursive functions from input/output-examples. In Emanuel Kitzelmann and Ute Schmid, editors, *Proceedings of the ECML/PKDD 2007 Workshop on Approaches and Applications of Inductive Programming (AAIP'07)*, pages 15–26, 2007.

[3] J. Roland Olsson. The art of writing specifications for the ADATE automatic programming system. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla,

Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 278–283, University of Wisconsin, Madison, Wisconsin, USA, 22-25 1998. Morgan Kaufmann.

[4] Roland Olsson. Inductive functional programming using incremental program transformation. *Artificial Intelligence*, 74(1), 1995.

[5] Geir Vattekar. Adate user manual. Technical report, Ostfold University College, 2006.

A Appendix

The following includes all the IGOR2 solutions for each problem and the corresponding ADATE best solution without and with assistance from IGOR2.

A.1 Specifications and Solutions

A.1.1 insert

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

Listing 5: The best ADATE solution from the unassisted specification.

```
fun f( Ys, Xs ) =
 case Xs of
   nill => cons( Ys, Xs )
 | cons( V37C, V37D ) =>
 case ( Ys < V37C ) of
   false => cons( V37C, f( Ys, V37D ) )
 | true => cons( Ys, Xs )
```

Listing 6: The changes to the specification.

```
fun f( (Zs , Ys, Xs) : int * int * list ) : list = raise D1

fun insert( (toInsert , aList) : int * list ) : list = case aList of
          nill => cons (toInsert , nill)
        | cons ( aH, aT ) => f( toInsert , aH, aT)

fun main( (Ys, Xs) : int * list ) : list = insert( Ys, Xs)
```

Listing 7: The best generated solution with assistance.

```
fun f( Zs, Ys, Xs ) =
 case ( Zs < Ys ) of
   false => f( Ys, Zs, Xs )
 | true => cons( Zs, insert( Ys, Xs ) )
```

A.1.2 switch

```
switch( [] ) = []
switch( [X] ) = [X]
switch( [X,Y|Z] ) = [Y,X|switch(Z)]
```

Listing 8: The best ADATE solution from the unassisted specification.

```
fun f Xs =
 case Xs of
   nil => Xs
 | cons( V1511, V1512 ) =>
 case V1512 of
   nil => Xs
 | cons( V34A2, V34A3 ) => cons( V34A2, cons( V1511, f( V34A3 ) ) )
```

Listing 9: The changes to the specification using abstraction.

```
fun f( (val1, val2, Xs) : (int * int * list ) ): list = raise D1

fun switch( Xs : list ) : list =  case Xs of
    nil => Xs
  | cons( Y1, Y2 ) =>
  case Y2 of
    nil => Xs
  | cons( W1, W2 ) => f( Y1, W1, W2 )

fun main( Xs : list ) : list = switch Xs
```

Listing 10: The best generated solution with abstraction.

```
fun f( val1, val2, Xs ) =
 cons( val2, cons( val1, switch( Xs ) ) )
```

Listing 11: The specification after redefining the function f.

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

Listing 12: The best generated solution with abstraction.

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

A.1.3 sort

```
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] )]
```

Listing 13: The best ADATE solution from the unassisted specification.

```
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 ) ) )
```

Listing 14: The changes to the specification.

```
fun f( (aInt, aList) : (int * list) ) : list = raise D1

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

fun main( Xs : list ) : list = sort Xs
```

Listing 15: The best generated solution with assistance.

```
fun f( aInt, aList ) =
 case sort( aList ) of
   V60E7D =>
 case V60E7D of
   nil => cons( aInt, V60E7D )
 | cons( V57A28, V57A29 ) =>
 case ( V57A28 < aInt ) of
```

```
    false => cons( aInt, V60E7D )
  | true => cons( V57A28, f( aInt, V57A29 ) )
```

A.1.4 swap

```
swap( [] ) = []
swap( [X|Y] ) = [last([X|Y]) | sub([X|Y)]

sub( [X] ) = []
sub( [X,Y|Z] ) = swap( [X| sub([Y|Z]) ] )

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

Listing 16: The best ADATE solution from the unassisted specification.

```
fun f Xs =
 case Xs of
   nil => Xs
 | cons( V144C, V144D ) =>
 case V144D of
   nil => Xs
 | cons( V63EC5, V63EC6 ) =>
 case f( V63EC6 ) of
   nil => cons( V63EC5, cons( V144C, V63EC6 ) )
 | cons( V66B8B, V66B8C ) =>
     cons( V66B8B, cons( V63EC5, f( cons( V144C, V66B8C ) ) ) )
```

Listing 17: The changes to the specification.

```
fun lastFun (aList : list) : int = case aList of
                          nil => raise D1
                        | cons ( aH, aT ) => (case aT of
                                  nil => aH
                                | cons (aaH, aaT) => lastFun ( aT )
                        )


fun f( (int1 , int2 , bList ): int * int * list ) : list = raise D1


fun sub( Xs: list ) : list = case Xs of
          nil => raise D1
        | cons ( aH, aT ) => case aT of
                    nil => nil
```

```
                | cons ( aaH, aaT ) => f (aH, aaH, aaT)

fun swap( Xs: list) :list = case Xs of
          nill => Xs
        | cons ( Hx , Tx) =>
                cons ( lastFun Xs , sub Xs )



fun main( Xs : list ) : list = swap Xs
```

Listing 18: The best generated solution with assistance.

```
fun f( int1 , int2 , bList ) =
 case bList of
   nill => cons( int1 , bList )
 | cons( V1B75, V1B76 ) => cons( int2 , f( int1 , V1B75, V1B76 ) )
```

A.1.5 lasts

```
lasts( [] ) = []
lasts( [[X]|Y] ) = [X| lasts(Y) ]
lasts( [X|Y] | Z] ) = lasts([Y|Z])
```

Listing 19: The best ADATE solution from the unassisted specification.

```
fun f X =
 case X of
   nill_list => nill
 | cons_list( V913, V914 ) =>
 case V913 of
   nill => (raise NA_192D)
 | cons( V192E, V192F ) =>
 case V192F of
   nill => cons( V192E, f( V914 ) )
 | cons( V1930, V1931 ) =>
     f(
       cons_list(
         case V1931 of
           nill => V192F
         | cons( V129B51, V129B52 ) => V1931,
         V914
         )
       )
```

Listing 20: The changes to the specification.

```
fun f ( (intHead, tailList, restLists) : (int*list* list_of_lists )) : list =
  raise D1

fun lasts_body ( Xs : list_of_lists ) : list = case Xs of
          nill_list => nill
        | cons_list( headList, tailLists ) => (case headList of
                  nill => raise D1
                | cons ( aH, aT ) => case aT of
                          nill => cons ( aH , lasts_body ( tailLists ) )
                        | cons (aaH, aaT) => f ( aH, aT, tailLists )
        )


fun main( Xs : list_of_lists ) : list = lasts_body Xs
```

Listing 21: The best generated solution with assistance.

```
 fun f( intHead, tailList, restLists ) =
  case tailList of
    nill => (raise NA_282)
  | cons( V283, V284 ) =>
      lasts_body(
        cons_list(
          case V284 of nill => tailList | cons( V285, V286 ) => V284,
          restLists
          )
        )
```

A.1.6 shiftR

```
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)]
```

Listing 22: The best ADATE solution from the unassisted specification.

```
 fun f X =
  case X of
    nill => X
```

```
  | cons( VB5B8, VB5B9 ) =>
  case f( VB5B9 ) of
    nil => X
  | cons( VB5BA, VB5BB ) => cons( VB5BA, cons( VB5B8, VB5BB ) )
```

Listing 23: The changes to the specification.

```
fun last ( Xs : list ) : int =
  case Xs of
    nil => (raise D1)
  | cons( aH, aT ) =>
  case aT of
    nil => aH
  | cons( aaH, aaT ) =>
  case aaT of nil => aaH | cons( V1F0E, V1F0F ) => last( aaT )

fun init ( X : list ) : list =
  case X of
    nil => (raise D1)
  | cons( V12BB, V12BC ) =>
  case V12BC of
    nil => nil
  | cons( V12BD, V12BE ) =>
  case V12BE of
    nil => cons( V12BB, V12BE )
  | cons( V76D5, V76D6 ) => cons( V12BB, cons( V12BD, init( V12BE ) ) )

fun f ( X : list ) : list= case X of
          nil => X
        | cons ( aH, aT ) => raise D1

fun main( X : list ) : list = f X
```

Listing 24: The best generated solution with assistance.

```
 fun f X =
  case X of
    nil => X
  | cons( V26EA, V26EB ) => cons( last( X ), init( X ) )
```

A.1.7 shiftL

```
shiftL( [] ) = []
shiftL( [X] ) = [X]
shiftL( [X,Y|Z] ) = [Y| shiftL([X|Z]) ]
```

Listing 25: The best ADATE solution from the unassisted specification.

```
fun f Xs =
 case Xs of
   nil => Xs
 | cons( V115F, V1160 ) =>
 case V1160 of
   nil => Xs
 | cons( V1B1D, V1B1E ) => cons( V1B1D, f( cons( V115F, V1B1E ) ) )
```

Listing 26: The changes to the specification.

```
fun f ( Xs : list ) : list = raise D1

fun igorshiftLbody (Xs : list) : list =
        case Xs of
          nil => Xs
        | cons ( HD1 , TL1 ) => case TL1 of
                  nil => Xs
                | cons ( HD2, TD2 ) => cons ( HD2, f Xs)

fun main( X : list ) : list = igorshiftLbody X
```

Listing 27: The best generated solution with assistance.

```
fun f Xs =
 case Xs of
   nil => (raise NA_52C)
 | cons( V52D, V52E ) =>
 case V52E of
   nil => (raise NA_52F)
 | cons( V530, V531 ) => igorshiftLbody( cons( V52D, V531 ) )
```

A.2 Results in Detail

Table 2: insert(X,Y)

	Solution #	Time	Efficiency	Improvement(%)	Recursive calls
Unassisted	1 of 2	6.62	190	-	1
	2 of 2	7.81	176	7.37	1
Problem Abstraction	1 of 2	18.37	240	-	1
	2 of 2	86.89	340	-41.67	2

Table 3: switch(X) = Y

	Solution #	Time	Efficiency	Improvement(%)	Recursive calls
Unassisted	1 of 1	4.34	302	-	1
Redefined	1 of 1	3.96	302	-	1
Abstraction	1 of 1	0.47	344	-	1

Table 4: sort(X) = Y

	Solution #	Time	Efficiency	Improvement(%)	Recursive calls
Unassisted	1 of 6	43.06	17958		4
	2 of 6	48.12	4165	76.81	3
	3 of 6	105.61	3715	10.8	3
	4 of 6	315.48	10231	-175.4	3
	5 of 6	457.99	2487	33.06	3
	6 of 6	571.06	2694	-8.32	3
Problem Abstraction	1 of 6	167.74	2975		2
	2 of 6	167.75	89597	-29.12	4
	3 of 6	196.46	13130	-3.41	3
	4 of 6	200.18	63050	-20.19	3
	5 of 6	225.13	2849	0.04	2
	6 of 6	244.43	3056	-0.03	2

Table 5: swap (X) = Y

	Solution #	Time	Efficiency	Improvement(%)	Recursive calls
Unassisted	1 of 4	10.91	11816	-	2
	2 of 4	106.75	9903	16.19	2
	3 of 4	277.58	1160	88.29	3
	4 of 4	292.05	1076	7.24	2
Problem Abstraction	1 of 4	12.74	2388	-	2
with `last`	2 of 4	16.07	1720	27.97	2
	3 of 4	37.95	995	45.83	1
	4 of 4	41.43	685	28.27	1
infer `last`	1 of 2	0.08	102	-	1
	2 of 2	0.60	72	29.41	1

Table 6: lasts(X) = Y

	Solution #	Time	Efficiency	Improvement (%)	Recursive calls
Unassisted	1 of 4	180.26	1922	-	2
	2 of 4	182.73	1922	0	2
	3 of 4	260.34	987	48.65	2
	4 of 4	263.75	1142	-15.7	2
Abstraction	1 of 2	0.04	1272	-	2
	2 of 2	6.25	1116	12.26	2

Table 7: shiftR(X) = Y

	Solution #	Time	Efficiency	Improvement (%)	Recursive calls
Unassisted	1 of 1	8.85	239	-	1
Redefined with `last` and `init`	1 of 1	1.79	239	-	0
infer `last`	1 of 2	0.08	102	-	1
	2 of 2	0.60	72	29.41	1
infer `init` run	1 of 3	0.27	189	-	1
	2 of 3	0.66	184	2.64	1
	3 of 3	9.07	150	18.47	1

Table 8: shiftL(X) = Y

	Solution #	Time	Efficiency	Improvement (%)	Recursive calls
Unassisted	1 of 1	4.17	221	-	1
Abstraction	1 of 2	0.61	281	-	1
	2 of 2	2.72	281	0	1