ELSEVIER

Artificial Intelligence 74 (1995) 55-81

Artificial Intelligence

Inductive functional programming using incremental program transformation

Roland Olsson *

Department of Computer Science, Østfold College, Os Allé 9, 1777 Halden, Norway

Received April 1993; revised March 1994

Abstract

The paper presents a system, ADATE, for automatic functional programming. ADATE uses specifications that contain few constraints on the programs to be synthesized and that allow a wide range of correct programs. ADATE can generate novel and unexpected recursive programs with automatic invention of recursive auxiliary functions. Successively better programs are developed using incremental program transformations. A key to the success of ADATE is the exact design of these transformations and how to systematically search for appropriate transformation sequences.

1. Introduction

This paper reports on a system, ADATE, that synthesizes recursive Standard ML programs using a specification consisting of sample inputs and an output evaluation function. The name ADATE, Automatic Design of Algorithms Through Evolution, indicates that the goal of the research is automatic invention of new algorithms and not only automatic implementation of algorithms that the ADATE user already knows.

One major dimension along which to differentiate inductive inference systems is the amount of information in the specifications that they employ. At one extreme are systems that use traces of computations [1]. At the same end of the spectrum are systems requiring specifications that consist of input–output pairs [2, 8, 10] or positive and negative examples as in inductive logic programming [6, 7, 9, 12]. In such systems, the input–output pairs or the examples must have a structure that corresponds to a specific algorithm.

* E-mail: Roland.Olsson@hiof.no.

SSDI 0004-3702(94)00042-5

At the other end of the spectrum are genetic algorithm (GA) systems [5] and ADATE, which use specifications such that the ratio between the difficulty of writing a desirable program and the difficulty of specification may be enormous. An important difference between ADATE and GA systems is that the latter are very poor at inferring recursive programs since they use primitive program transformations and an unsystematic search of the program space.

Section 2 explains how to specify programs and gives measures of program quality such as correctness, syntactic complexity and time complexity. These measures are used to guide the search of the program space. Section 3 presents the subset of Standard ML in which inferred programs are written. Programs are synthesized using incremental transformations as discussed in Section 4. These transformations are expression replacement, function abstraction, case-distribution and type embedding. The overall strategy for searching the program space is given in Section 5. This strategy is based on iterative-deepening [4]. Sections 6 and 7 list implementation details and experimental results. The next section discusses inductive inference systems that are related to ADATE. The final section contains merits and drawbacks with ADATE and directions for future research.

2. Specification and selection of programs

A specification implicitly defines a set C of correct programs. The specification writer wants a program chosen from a set D of desirable programs. Some requirements for a specification are:

(1) The specification should be as easy as possible to write and preferably be much simpler than any desirable program.
(2) The specification should facilitate efficient inference.
(3) All desirable programs should be correct according to the specification, i.e., $D \subseteq C$.
(4) A computer should reasonably quickly be able to decide if a given program is correct.

Requirements (1) and (2) are often in conflict. One main goal of the research presented in this paper was to allow specifications to be as simple as possible. The only efficiency goal was that many interesting inferences should be possible using computers that were generally available in 1993.

A specification that satisfies requirement (3) is said to be *loose*. A loose specification does not contain constraints that eliminate desirable programs. ADATE specifications are loose. Even if requirement 4 is satisfied, there are still many specifications that are very simple in comparison with the programs that satisfy them. For example, most of the well-known NP-hard problems can be used to construct such specifications, which employ sample inputs and an output evaluation function.

Example 2.1. Assume that I is a large instance of the traveling salesman problem and that the specification writer knows the minimum length $L_{\min}$ of a Hamiltonian cycle on I. It is easy to construct such an instance in time $O(n^2)$, where n is the total number of nodes. Here is a simple specification of a program P.

Given input I, P is required to output a Hamiltonian cycle C of length $L_{\min}$ in less than $n^2/10^6$ CPU seconds.

Note that it takes time $O(n)$ to check if C is a Hamiltonian cycle of length $L_{\min}$. The correctness of P is thus decidable in time $O(n^2/10^6) + O(n) = O(n^2)$ even though P may be extremely difficult to find.

The *Journal of Algorithms* maintains a list with hundreds of NP-complete problems that can be used to construct similar specifications.

2.1. Specification form

Assume that a specification is to be used to check a synthesized ML program P. P is a definition of a function f which is an approximation of a desirable function.

An ADATE specification consists of

(1) a set of types,
(2) the primitive functions that are to be used in inferred programs,
(3) the type of f,
(4) a set of sample inputs $\{ I_1, I_2, \ldots, I_{\#I} \}$,
(5) an output evaluation function oe, which uses the set $\{(I_1, f(I_1)), \ldots, (I_{\#I}, f(I_{\#I}))\}$ to rate P.

The sample inputs need to be chosen so that incremental inference is facilitated. This means that the inputs should contain sufficiently many special cases. The sample inputs in the specification of a list sorting program may for example include an empty list, a singleton list, a sorted list and a few random lists. One interesting progression of more and more difficult sample inputs would be the problems in mathematics textbooks, ranging from first grade in elementary school up to university level. Even if the specification writer may not need to be as "pedagogical" as the authors of such textbooks, the sample inputs still need to be carefully chosen.

It is important that specifications are not required to be based on input–output pairs. We have identified the following four problems with input–output pair specifications.

Problems.

(1) The choice of output sometimes reflects the particular algorithm that was used to construct it. The specification writer needs to know this algorithm to be able to provide output. An inference system naturally becomes much less useful if the writer is required to know the algorithm to be inferred.
(2) Looseness is lost if the pairs do not include all possible outputs for a given input.
(3) An input–output pair specification grades an output as correct or wrong. It is often desirable to use more than two grades. For example, the grades can be all real numbers in some interval.
(4) It may be too difficult for the user to provide optimal outputs.

Here are four examples such that example number (i) illustrates problem number (i).

Examples.

(1) Consider the specification of a function

```
split : 'a list -> 'a list * 'a list
```

that splits a list `Xs` into a pair of lists `(Ys,Zs)` such that the lengths of `Ys` and `Zs` differ by at most one. The `split` function is useful when implementing merge sort. The input–output pair

```
([1,2,3,4,5,6,7,8], ([1,2,3,4],[5,6,7,8]))
```

obviously reflects the particular algorithm that chooses `Ys` to the first half of `Xs` and `Zs` to the second half. However, the following split algorithm is both simpler and faster.

```
fun split nil = (nil,nil)
  | split (X1::Xs1) = case split Xs1 of (Ys,Zs) => (X1::Zs,Ys)
```

An input–output pair that reflects this algorithm is thus

```
([1,2,3,4,5,6,7,8], ([1,3,5,7],[2,4,6,8])).
```

Instead of giving outputs, it is much better to provide an output evaluation function. Assume that the function `is_perm` is defined so that `is_perm(As,Bs)` means that `Bs` is a permutation of `As`. Given input `Xs` and output `(Ys,Zs)`, the output evaluation function computes

```
is_perm(Xs,Ys@Zs) andalso abs(length Ys - length Zs) <= 1,
```

where `@` is the ML operator for list concatenation.

(2) Problem (2) can be exemplified using the above TSP specification. If the specification only allowed programs that produce a particular pre-determined tour of length $L_{\min}$, a program that produces another tour of length $L_{\min}$ would be regarded as incorrect. The specification would therefore not be loose if such a tour exists.

(3) This example illustrates the usefulness of grades. Consider navigation of a polygon among polygonal obstacles. When computing the output evaluation function one might check if a given path, represented by a series of points and angles of rotation, intersects any obstacle, compute the length and curvature of the path, the amount of rotation along the path and its safety i.e., margin to obstacles.

(4) In order to illustrate that it may be problematic to provide optimal outputs, consider choosing random graphs as inputs in the TSP specification. It would then be difficult for the specification writer to provide optimal outputs i.e., Hamiltonian cycles of minimum length.

2.2. *The output evaluation function*

Since the output evaluation function `oe` is of fundamental importance in ADATE, the exact form of `oe` is described below. An inferred program may contain a special constant, `?`, that needs to be considered when defining `oe`. A `?` constant means "don't-

know". A correct output is better than a don't-know output which in turn is better than a wrong output. Let the type of f be `input_type -> output_type`. The domain type of `oe` is

```
(input_type * output_type exec_result) list,
```

where `exec_result` is defined as

```
datatype 'a exec_result = ? | too_many_calls | some of 'a.
```

The outcome of the computation of $f(I_i)$ is

- `?` if any `?`-constant was evaluated,
- `too_many_calls` if the call count limit, which is discussed in Section 6, was exceeded, and
- `some` O_i otherwise.

ADATE calls `oe` with an argument `Execute_result` which is a list of the form $[(I_1, R_1), \ldots, (I_{\#I}, R_{\#I})]$, where each R_i is the outcome of the computation of $f(I_i)$. The range type of `oe` is `cwd list * real list` where `cwd` is defined as

```
datatype cwd = correct | wrong | dont_know
```

If the call `oe Execute_result` returns `(Cs,Grades)`, element number i in `Cs` corresponds to (I_i, R_i) in `Execute_result`. `Grades` is a list of floating point numbers which is to be minimized according to the usual lexicographic ordering on lists. `Grades` may for example have the form `[Grade_1,Grade_2]`, where `Grade_1` is more important than `Grade_2`.

Example 2.2. Consider the specification of a program that simplifies polynomials. Assume that simplification of a polynomial `Xs`, e.g. $3X^2 + 4X + 8X^2 - 5X + 4 - X^2 + 8$, yields a polynomial `Ys`, e.g. $12 + 10X^2 - X$. For a given polynomial `Xs` the user may need to determine how good an output `Ys` is without knowing any optimal output nor any way of computing one. Assume that the function `eval_pol` is defined so that the call `eval_pol(Pol,Z)` evaluates the polynomial `Pol` with the integer `Z` substituted for the variable in the polynomial, e.g. `eval_pol`$(X^3+X^2+1, 3) = 37$. Note that `eval_pol` is easier to define than a function that simplifies polynomials. `Grades` is a singleton list `[Grade]` such that `Grade` is the sum of the lengths of all correct output polynomials.

If `M` and `N` are the number of terms in `Xs` and `Ys` respectively, `oe` checks that `eval_pol(Xs,X) = eval_pol(Ys,X)` for all integers `X` in $1, \ldots, M + N$. This check obviously suffices to ensure that `Xs` and `Ys` are equivalent since `Xs@Ys` cannot contain terms of more than $M + N$ different degrees. A polynomial is represented as a list of (coefficient,exponent) pairs. The complete definition of `oe`, including the auxiliary `eval_pol` definition, is shown in Fig. 1. This definition looks complicated in comparison with a polynomial simplification program. However, the structure of `oe`-definitions is basically the same for all specifications, even if much more complicated programs are specified.

```
fun eval_pol(Pol,Z) =
  case Pol of
    nil => 0
  | (Coeff,Exponent)::Pol =>
  Coeff * int_pow(Z,Exponent) + eval_pol(Pol,Z)
      handle _ => 0

fun oe(Execute_result : (input_type * output_type exec_result) list)
    : cwd list * real list =
  let
    val Zs = map(fn(Xs,R) =>
      case R of
        ? => (dont_know,0)
      | too_many_calls => (wrong,0)
      | some Ys =>
      let val M = length Xs val N = length Ys in
        if (N<=1 orelse N<M) andalso
           forall(fn X => eval_pol(Xs,X)=eval_pol(Ys,X), fromto(1,M+N))
        then
          (correct,N)
        else
          (wrong,0)
      end,
      Execute_result)
  in
    (map(#1,Zs), [real(int_sum(map(#2,Zs)))])
  end
```

Fig. 1. The output evaluation function for polynomial simplification.

2.3. *The program evaluation functions*

ADATE uses the sample inputs $I_1, \ldots, I_{\#I}$ and the output evaluation function oe to compute three program evaluation functions pe_1, pe_2 and pe_3 that supplement the program rating provided by oe with measures of syntactic complexity and time complexity.

The syntactic complexity is defined as follows. Let $N_1, \ldots, N_{\#N}$ be all nodes in the tree representations of all expressions in the program. Due to scoping constraints, the number of symbols that may occur in node N_i is limited to some number m_i. For simplicity and speed, type constraints are ignored when computing syntactic complexity, which is defined as $\sum_{i=1}^{\#N} \log_2 m_i$.

The function f and the let-functions defined in a program P are called during the computation of $f(I_1), \ldots, f(I_{\#I})$. The time complexity measure for P is the total number of such calls.

Let

- N_c = the number of correct outputs,
- N_w = the number of wrong outputs,
- S = the syntactic complexity,

Table 1
The definitions of pe_1, pe_2 and pe_3

i	Value returned by pe_i
1	$-N_c$ :: Grades @ $[N_w, S, T]$
2	$-N_c$ :: Grades @ $[N_w, T, S]$
3	$[N_w, -N_c]$ @ Grades @ $[S, T]$

- T = the total call count.

The three program evaluation functions are defined in Table 1. A program P is considered to be better than a program Q according to pe_i if and only if $pe_i(P)$ comes before $pe_i(Q)$ in the lexicographic ordering of lists. For example, the program evaluation function pe_1 prefers correctness to small syntactic complexity which in turn is preferred to low call count.

3. The functional language in which inferred programs are written

Inferred programs are written in a subset of Standard ML without currying, boolean operators, `if`-expressions and references. All functions are viewed as having a single argument which is a tuple. The ML subset consists of `datatype`-definitions, `fun`-definitions, `case`-expressions and `let`-expressions.

The pattern in the left-hand side of a `fun`-definition is restricted to a single tuple pattern. A tuple pattern is always required to be fully layered which means that names are introduced for all possible parts of a tuple pattern. For example, the type `((int * int) * int) * int` corresponds to a pattern like `(A as (B as (C,D),E),F)`. Requiring tuple patterns to be fully layered often leads to the introduction of superfluous names. This problem is more aesthetic than practical.

The alternatives in a `case`-expression are required to exactly correspond to the alternatives in the `datatype`-definition for the type of the expression that is analyzed. The only constructor allowed in the pattern of a `case`-alternative in addition to tuple constructors is a single occurrence of the corresponding constructor in the `datatype`-definition. `case`-expressions are used instead of `let`-expressions that introduce functions of arity zero. Thus, the `case`-expression `case` E_1 `of V =>` E_2 is used instead of the `let`-expression `let val V =` E_1 `in` E_2 `end`.

In ML, any expression E_1 may be applied to an expression E_2 provided that the type of E_2 matches the domain type of E_1, but ADATE only produces applications where E_1 is a function symbol.

4. The program transformations

A *compound* transformation is the composition of a sequence of *atomic* transformations. The program evaluation functions pe_1, pe_2 and pe_3, which are used to determine whether a program is to be kept or discarded, are only applied to programs resulting from compound transformations. Assume that program P_{i+1} is produced from program

P_i with an atomic transformation t_i. A compound transformation that produces $P_{\#\,t+1}$ from P_1 will be written $t_1 t_2 \ldots t_{\#\,t}$.

The initial program only consists of a single ? and thus gives a don't-know output for all inputs. The final program is evolved from the initial program through a sequence of compound transformations.

4.1. Expression synthesis

The synthesis of new expressions is fundamental for the transformation of a program. A simple form of expression synthesis is enumerative and exhaustive production of type correct expressions containing a fixed set of function symbols. Expressions are synthesized in order of increasing size. The size of an expression is the number of nodes in the tree representation of the expression. ADATE first synthesizes all expressions of size 1, then all expressions of size 2 and so on. Since the number of expressions normally grows exponentially with size, great care must be taken to keep the size small and to heuristically identify the most promising expressions. The mere thought of such exponential growth undoubtedly acts as a mental barrier that is hard to overcome when trying to understand the limitations of ADATE.

ADATE employs the following heuristics.

(1) Assume that

> `case` E `of` $Match_1$ `=>` $Unknown_1$ | $\cdots$ | $Match_n$ `=>` $Unknown_n$

is a partially synthesized `case`-expression where each $Unknown_i$ is a "dummy" constant that later is to be replaced with a synthesized expression.

The program to be transformed contains a subexpression *Sub* that is to be replaced by a finished synthesized expression. In order to check if the incomplete `case`-expression should be discarded, *Sub* is replaced with the expression

> (`case` E `of` $Match_1$ `=>` $Unknown_1$ | $\cdots$ | $Match_n$ `=>` $Unknown_n$;
> *Sub*).

The resulting program is then executed for all sample inputs. An expression is said to be *activated* if and only if it was evaluated during this execution. The entire `case`-expression is discarded if only one $Unknown_i$ was activated and the corresponding $Match_i$ does not contain any variable. Otherwise, the finished `case`-expression is produced by replacing each activated $Unknown_i$ with a synthesized expression and each non-activated $Unknown_i$ with a ?.

(2) Consider the synthesis of a recursive call $g(A_1, A_2, \ldots, A_n)$ occurring in the declaration `fun` $g(V_1, V_2, \ldots, V_n) = \cdots$. At least one A_i is required to be "smaller" than the corresponding V_i. A_i is "smaller" than V_i if and only if A_i occurs in an RHS_k in a `case`-expression

> `case` V_i `of` $Match_1$ `=>` RHS_1 | $\cdots$ | $Match_m$ `=>` RHS_m

and

(a) A_i is a proper subexpression of $Match_k$ or

(b) $Match_k$ contains a variable W such that A_i is "smaller" than W.

4.2. The atomic transformations

There are four atomic transformations namely replacement, abstraction, case-distribution and embedding.

4.2.1. Replacement

Replacement is the only atomic transformation that may change the semantics of a program. A replacement substitutes a synthesized expression *Synt* for a subexpression *Sub* of the program. During the synthesis of *Synt*, subexpressions of *Sub* may be reused as subexpressions of *Synt*. When *Sub* itself, but no other subexpression of *Sub*, is reused, the replacement can be viewed as an insertion of a synthesized expression.

Example 4.1. Consider the inference of a list sorting program. Assume that the sample inputs are I_1 = `[]`, I_2 = `[10]`, I_3 = `[10,20,30,40]`, I_4 = `[50,20,60,20,40]` and I_5 = `[10,20,50,40]`. In one out of many possible inferences of `sort`, each compound transformation except the last consists of a single replacement. The initial program is `fun sort Xs = ?`.

(1) Replacing the ? with the synthesized expression

```
case Xs of nil => Xs | X1::Xs1 => ?
```

is the first compound transformation which gives

```
fun sort Xs = case Xs of nil => Xs | X1::Xs1 => ?
```

(2) The next compound transformation is another ?-replacement that yields

```
fun sort Xs =
  case Xs of nil => Xs
  | X1::Xs1 => case Xs1 of nil => Xs | X2::Xs2 => ?
```

(3) The third compound transformation, which also is a ?-replacement, gives

```
fun sort Xs =
  case Xs of nil => Xs
  | X1::Xs1 =>
  case Xs1 of nil => Xs
  | X2::Xs2 => case X2<X1 of true => ? | false => Xs
```

(4) The program above gives correct output for inputs I_1, I_2 and I_3, ?-output for I_4 and wrong output for I_5. The program is improved to also give ?-output for I_5 by replacing `Xs1` with the synthesized expression `sort Xs1`, which yields

```
fun sort Xs =
  case Xs of nil => Xs
  | X1::Xs1 =>
  case sort Xs1 of nil => Xs
  | X2::Xs2 => case X2<X1 of true => ? | false => Xs
```

The final compound transformation is shown in Section 4.2.2.

In order to discriminate between replacements, ADATE employs a special program evaluation function pe_{REQ} which returns $-N_c$::Grades @ [N_w]. A replacement that does not increase the pe_{REQ} value will be denoted by REQ whereas an ordinary replacement will be denoted by R. If a compound transformation contains several replacements, ADATE usually requires that one or more of the replacements are REQs. REQs are found by trying Rs and selecting the ones that do not increase the pe_{REQ} value. Normally, only a small fraction of the Rs meet this requirement. The REQs are sorted according to the pe_{REQ} value to give preference to the best REQs.

4.2.2. Abstraction

An abstraction introduces a `let`-function with a definition based on a subexpression E of the program to be transformed. The transformation schema is

$H(E_1, E_2, \ldots, E_n) \longrightarrow$
`let fun g(`$V_1, V_2, \ldots, V_n$`) = `$H(V_1, V_2, \ldots, V_n)$` in g(`$E_1, E_2, \ldots, E_n$`) end,`

where g is a new function. Matching E against $H(E_1, E_2, \ldots, E_n)$ may be viewed as higher-order unification that finds values for the variables $H, E_1, E_2, \ldots, E_n$.

Example 4.2. One possible unifier of $H(E_1, E_2)$ and `a(b(c(d),e),f(x))` is E_1 = `c(d)`, E_2 = `f(x)` and H = `fn(X1,X2) => a(b(X1,e),X2)`.

An abstraction is done by choosing $E_1, \ldots, E_n$ to disjoint subexpressions of E.

Example 4.3. The last compound transformation in the inference of `sort` has the form ABSTR REQ REQ R where ABSTR denotes an abstraction transformation. Consider the last `sort` program given above. The ABSTR has $n = 1$, E_1 = `sort Xs1` and

$H(E_1)$ =
```
case sort Xs1 of nil => Xs
| X2::Xs2 => case X2<X1 of true => ? | false => Xs
```

The program produced by the ABSTR is thus

```
fun sort Xs =
  case Xs of nil => Xs
  | X1::Xs1 =>
  let fun g V1 =
    case V1 of nil => Xs
    | X2::Xs2 => case X2<X1 of true => ? | false => Xs
  in
    g(sort Xs1)
  end
```

The first REQ replaces the third occurrence of `Xs`. The second REQ replaces the fourth occurrence of `Xs`. Assume that these occurrences for pedagogical reasons are labeled `Xs'` and `Xs''`. The program above is then written as

```
fun sort Xs =
  case Xs of nil => Xs
  | X1::Xs1 =>
  let fun g V1 =
    case V1 of nil => Xs'
    | X2::Xs2 => case X2<X1 of true => ? | false => Xs''
  in
    g(sort Xs1)
  end
```

The first REQ replaces `Xs'` with the synthesized expression `X1::nil`. This preserves equivalence since `Xs'` always is a singleton. The second REQ replaces `Xs''` with the synthesized expression `X1::V1`. Equivalence is preserved since `Xs''` always is sorted. The R then finally replaces the ? with the synthesized expression `X2::g Xs2` which yields a correct sorting program i.e.,

```
fun sort Xs =
  case Xs of nil => Xs
  | X1::Xs1 =>
  let fun g V1 =
    case V1 of nil => X1::nil
    | X2::Xs2 => case X2<X1 of true => X2::g Xs2 | false => X1::V1
  in
    g(sort Xs1)
  end
```

4.2.3. case-*distribution*

This transformation is based on the following schema, where h denotes a function symbol.

$$h(A_1,\ldots,A_i,\ \texttt{case}\ E\ \texttt{of}\ Match_1\ \texttt{=>}\ E_1\ \texttt{|}\ \cdots \texttt{|}\ Match_n\ \texttt{=>}\ E_n, A_{i+1},\ldots,A_m) \longleftrightarrow$$

$$\begin{array}{l}\texttt{case}\ E\ \texttt{of}\\ \quad Match_1\ \texttt{=>}\ h(A_1,\ldots,A_i,E_1,A_{i+1},\ldots,A_m)\\ \quad\vdots\\ \quad \texttt{|}\ Match_n\ \texttt{=>}\ h(A_1,\ldots,A_i,E_n,A_{i+1},\ldots,A_m)\end{array}$$

Note that the schema may be used both left-to-right and right-to-left. If the schema is used left-to-right and some E_k is ?, $h(A_1,\ldots,A_i,E_k,A_{i+1},\ldots,A_m)$ is changed to ?.

A case-distribution transformation consists of one or more applications of the schema. Each application marks the h and the case that were used. In the most general case, the first application may use any h and case such that the h is the parent or a child of the case. Subsequent applications are only allowed to use hs and cases that are marked or that have at least one marked child or parent in the expression tree. The purpose of marking is to only allow "related" applications of the schema.

4.2.4. Embedding

An embedding generalizes the type of a `let`-function. Two examples of embeddings are to add an argument to the function or to change an argument of type `'a` to one of type `'a list`. Assume that the `let`-function to be embedded has the definition

`let fun g(`$V_1, V_2, \ldots, V_n$`) =` *RHS* `in` *Exp* `end.`

In its most general form, an embedding inserts a synthesized type expression into the type expression for g. When the type of g changes, the types of functions occurring in *RHS* and *Exp* may need to change too. Changing these types may make it necessary to change other types and so on. Since this "chain reaction" makes it a bit difficult to choose which types to change, a simplified form of embedding that avoids chain reactions is described below.

The data type definitions provided by the specification writer are used for embedding. The allowed data type definitions are a subset of ML data type definitions and have the following form.

`datatype (`$'a_1, 'a_2, \ldots, 'a_{\#a}$`)` *Type_constructor* `=`
C_1 `of` $T_{1,1} * T_{1,2} * \cdots * T_{1,\#T_1}$ |
C_2 `of` $T_{2,1} * T_{2,2} * \cdots * T_{2,\#T_2}$ |
$\vdots$
$C_{\#C}$ `of` $T_{\#C,1} * T_{\#C,2} * \cdots * T_{\#C,\#T_{\#C}}$

Each $'a_i$ is a type variable, each C_j is a constructor and each $T_{j,k}$ is the type of argument number k of constructor C_j.

A given `datatype`-definition may be used to embed a type T only if T matches some $T_{j,k}$. The types T and $T_{j,k}$ are considered to match only if a function with domain type $T_{j,k}$ may be applied to an object of type T according to the typing rules of ML.

Example 4.4. The `datatype`-definition for lists is

```
datatype 'a list = nil | :: of 'a * 'a list
```

Since $T_{2,1}$ is the type variable `'a`, which matches any type, this definition may be used to embed any type. For example, embedding the type `'b bin_tree` yields the type `('b bin_tree) list`.

Tuple types are predefined and given special treatment. A tuple type $T_1 * \cdots * T_n$ can be embedded in two ways.

(1) The new type is $T_1 * \cdots * T_n * 'a$, where $'a$ is a fresh type variable.
(2) An index i is chosen and the type T_i is embedded using a `datatype`-definition as described above.

The embedding of a proper subtree of some T_i is not allowed. Using the type constructor `bin_tree`, the tuple type `int list * bool` may for example be embedded to `(int list) bin_tree * bool` but not to `(int bin_tree) list * bool`. This restriction simplifies the translation between an expression of the old type and the corresponding expression of the new type as described below.

The only tuple types that may be embedded are the domain and the range of g. Note that all embeddings given below preserve semantics and completely avoid chain reactions. The following schemas use a special constant, `?_emb`, to denote an expression to be synthesized as part of an embedding transformation.

Embedding the domain of g

Assume that the domain type of g is $T_1 * \cdots * T_n$ and that the `datatype`-definition for lists is to be used. The two ways of embedding tuple types given above are now used as follows.

(1) $T_1 * \cdots * T_n$ to $T_1 * \cdots * T_n *' a$. Each call of the form $g(E_1, \ldots, E_n)$ is changed to $g(E_1, \ldots, E_n, ?\_emb)$

(2) $T_1 * \cdots * T_i * \cdots * T_n$ to $T_1 * \cdots * T_i$ `list` $* \cdots * T_n$.

(a) Each call $g(E_1, \ldots, E_i, \ldots, E_n)$ is changed to $g(E_1, \ldots, E_i::?\_emb, \ldots, E_n)$.

(b) *RHS* is replaced by `case` V_i `of nil => ?_emb | ` $X::Xs$ `=>` (*RHS* with X substituted for V_i), where X and Xs are fresh variables.

Embedding the range of g

Assume that the range type of g is $T_1 * \cdots * T_n$ and that the `datatype`-definition for lists is to be used. The two ways of embedding tuple types given above are now used as follows.

(1) $T_1 * \cdots * T_n$ to $T_1 * \cdots * T_n * 'a$.

(a) Each call $g(\ldots)$ is changed to

```
case g(...) of X as (X1,...,Xn,Xn+1) => (X1,...,Xn).
```

(b) The *RHS*, which in this case is assumed to have the form $(E_1, \ldots, E_n)$, is changed to $(E_1, \ldots, E_n, ?\_emb)$. If $n = 1$ and E_1 is a `case`-expression, `case`-distribution is used to move `?_emb` downwards until no `?_emb` has a `case`-expression as sibling. This is illustrated by the `del_min` example below.

(2) $T_1 * \cdots * T_i * \cdots * T_n$ to $T_1 * \cdots * T_i$ `list` $* \cdots * T_n$.

(a) If $n = 1$, each call $g(\ldots)$ is changed to

```
case g(...) of nil => ?_emb | X::Xs => X.
```

If $n \geqslant 2$, each call $g(\ldots)$ is changed to

```
case g(...) of X as (X1,...,Xi,...,Xn) =>
 case Xi of nil => ?_emb
 | Y::Ys => (X1,...,Xi-1,Y,Xi+1,...,Xn).
```

(b) The *RHS*, which in this case is assumed to have the form $(E_1, \ldots, E_i, \ldots, E_n)$, is changed to $(E_1, \ldots, E_i::?\_emb, \ldots, E_n)$. If E_i is a `case`-expression, `case`-distribution is employed to move `?_emb` downwards until no `?_emb` has a `case`-expression as sibling.

The datatype-definition for lists was used above in order to make the presentation less abstract. In case (2)(a) for embedding of the domain and in case (2)(b) for embedding of the range, the constructor :: was used to translate an expression E_i of type T_i to an expression of type T_i list. In general, the datatype-definition may contain several types $T_{j,k}$ that match T_i. For each such $T_{j,k}$, E_i may be translated to $C_j(?\_emb, \ldots, E_i, \ldots, ?\_emb)$ where E_i is argument number k. It is of course also straightforward to generalize case-analysis to datatype-definitions other than the one for lists. The same (j, k) must be used for all translations in the same embedding. This restriction ensures that the system knows which case-alternative to use for translation in case (2)(b) for embedding of the domain and in case (2)(a) for embedding of the range.

Example 4.5. Consider the inference of a program del_min : int list -> int list that deletes one occurrence of the smallest integer in a list. Since an empty list does not have a smallest element, it is natural for del_min nil to evaluate to ?. If ADATE was given a function min that finds the smallest element in a list or a function delete_one that deletes one occurrence of an element from a list, the inference would be trivial. An important point is that ADATE is given neither of these functions, which means that it is required to invent corresponding "auxiliary functionality". The sample inputs are I_1 = [10], I_2 = [1,2,3,4,5,6,7,8,9,10] and I_3 = [5,9,45,46,28,3,11,10,30,23]. Here is one of many possible inferences of del_min. The initial program is fun del_min Xs = ?.

(1) The first compound transformation is a single R which gives

```
fun del_min Xs =
  case Xs of nil => ?
  | X1::Xs1 =>
  case Xs1 of nil => nil | X2::Xs2 => ?
```

(2) The second compound transformation has the form ABSTR EMB REQ R. The ABSTR gives

```
fun del_min Xs =
  let fun g Ys =
    case Ys of nil => ?
    | X1::Xs1 =>
    case Xs1 of nil => nil | X2::Xs2 => ?
  in
    g Xs
  end
```

The range of g is then embedded so that the type of g is changed from int list -> int list to int list -> int list * int. Application of schema (1) for embedding of the range and the accompanying case-distribution gives

```
fun del_min Xs =
  let fun g Ys =
    case Ys of nil => ?
    | X1::Xs1 =>
    case Xs1 of nil => (nil,?_emb) | X2::Xs2 => ?
  in
    case g Xs of V as (Zs,Z) => Zs
  end
```

Note that the case-distribution changes each of the two occurrences of ? to (?,?_emb) which in turn immediately is replaced by ?. The type of each of the two occurrences of ? naturally changes from int list to int list * int. The EMB is then finished by replacing the single occurrence of ?_emb with X1. Note that this program still has $N_c = 1$ and $N_w = 0$. The REQ yields a program with $N_c = 2$ and $N_w = 0$, namely

```
fun del_min Xs =
  let fun g Ys =
    case Ys of nil => ?
    | X1::Xs1 =>
    case Xs1 of nil => (nil,X1) | X2::Xs2 =>
    case g Xs1 of V as (Ws,W) =>
    case X1<W of true => (Xs1,X1) | false => ?
  in
    case g Xs of V as (Zs,Z) => Zs
  end
```

Note that the REQ is facilitated by input I_2. The R then produces the final program by replacing the last ? with (X1::Ws,W). The final program has $N_c = 3$ and $N_w = 0$.

This inference is unusually short since it only consists of two compound transformations.

4.3. *Synthesis of compound transformations*

Recall that a compound transformation is a sequence $t_1 \ldots t_{\#t}$ where each atomic transformation t_i is one of the following.

- R. Replacement.
- REQ. Replacement that does not make the program "worse".
- ABSTR. Abstraction.
- CASE-DIST. case-distribution.
- EMB. Embedding.

4.3.1. *Compound transformations forms*

The choice of an atomic transformation t_i, $i \geqslant 2$, depends on the previously chosen transformations $t_1 \ldots t_{i-1}$. No transformation except the first may be chosen freely. The

dependency is specified with so-called coupling rules which are employed to produce all possible compound transformation *forms*.

Example 4.6. Consider the last compound transformation in the inference of `sort` presented above. The *form* of this compound transformation is ABSTR REQ REQ R where both the REQs and the R are coupled to the ABSTR as described below.

Assume that $t_1 \ldots t_{i-1}$ have been chosen so far and that t_i is to be chosen next. A "weak" coupling rule $t' \rightarrow t''$ means that t_i may be chosen to t'' if $t' \in \{t_1, \ldots, t_{i-1}\}$. A "strong" coupling rule $t' \Rightarrow t''$ means that t_i may be chosen to t'' if $t' = t_{i-1}$. When a rule $t' \rightarrow t''$ or $t' \Rightarrow t''$ is used with t' equal to some t_k, t_i is said to be coupled to t_k. If a t'' is followed by an ! mark in a coupling rule, no subsequent transformation may be coupled to t''. No rule may be used more than once during the production of a form, which means that there are a finite number of possible forms. These forms are computed immediately after system start up and remain unchanged during the entire execution.

Transformation t_1 is chosen to R, REQ, ABSTR, CASE-DIST or EMB. A form is required to have $t_{\#\, t} = \text{R}$ and $t_i \neq \text{R}$ for each $i < \#\, t$. Each transformation t_i, $i \geqslant 2$, is chosen with one of the coupling rules below. Each t'' in a coupling rule is constrained by the applicability requirement listed after each rule.

(1) REQ $\Rightarrow$ R. The R is applied in the expression introduced by the REQ.
(2) REQ $\Rightarrow$ ABSTR. The ABSTR is such that the expression introduced by the REQ occurs in the $H(E_1, \ldots, E_n)$ used by the ABSTR but not entirely in H.
(3) ABSTR $\rightarrow$ R. The R is applied in the right-hand side $H(V_1, \ldots, V_n)$ of the `let`-definition introduced by the ABSTR.
(4) (a) ABSTR $\rightarrow$ REQ! or (b) ABSTR $\rightarrow$ REQ! REQ!. The REQ(s) are applied in $H(V_1, \ldots, V_n)$.
(5) ABSTR $\Rightarrow$ EMB!. The `let`-function introduced by the ABSTR is embedded.
(6) CASE-DIST $\Rightarrow$ ABSTR. The ABSTR is such that the root of $H(E_1, \ldots, E_n)$ was marked by the CASE-DIST.
(7) CASE-DIST $\Rightarrow$ R. The R is such that the root of the expression *Sub*, which is replaced by the R, was marked by the CASE-DIST.
(8) EMB $\rightarrow$ R. The R is applied in the right-hand side of the definition of the embedded function.

Combining these 8 rules in all possible ways yields 22 forms. For example, the form ABSTR REQ REQ R is produced by first choosing t_1 to ABSTR and then applying coupling rules (4b) and (3). The rule set above was found empirically and may need to be extended.

Since coupling rules normally focus a compound transformation within a small part of the program, they are particularly important for the transformation of very large programs. For example, assume that a program contains N subexpressions and that an ABSTR is applied so that $H(V_1, \ldots, V_n)$ contains N_{RHS} subexpressions. Consider the form ABSTR REQ REQ R. Each of the four transformations needs to choose at least one subexpression. Without coupling, there would be about $N^4/2$ such choices whereas there are about $N \cdot N_{RHS}^3/2$ choices with coupling, which means that coupling is particularly important for small N_{RHS}/N ratios. The denominator 2 is used since the first REQ

and the second REQ may be interchanged without changing the result of a compound transformation. The actual number of choices is often much smaller than $N \cdot N_{RHS}^3/2$ since REQs only are found for a small fraction of the N_{RHS} subexpressions.

4.3.2. The algorithm that uses the forms to produce programs

The algorithm operates with two concepts, *work* and combinatorial *cost*. The *work* is the approximate number of programs to be produced from the current program. The *cost* is a measure of the complexity of a compound transformation. For each choice made by the algorithm, the cost is multiplied by the number of alternatives that the algorithm has chosen between.

Let W_{tot} be the total work goal. Let N_{forms} be the number of forms of compound transformations. The number of programs to be produced using a specific form is chosen to $W_{\text{tot}}/N_{\text{forms}}$. W_{tot} is thus uniformly distributed on the forms.

Given the current program P, a specific form F and a cost limit C, assume that

```
comp_synt(F : form, C : real, P : program, Emit : program -> unit) : unit
```

is an ML function that makes the call `Emit` $\mathtt{P}_i$ for each program $\mathtt{P}_i$ that can be produced from P using form F with a cost less than C. For each form F, the synthesis algorithm makes calls `comp_synt(F,3.0,P,Emit)`, `comp_synt(F,9.0,P,Emit)`, `comp_synt(F,27.0,P,Emit)`, ... until $W_{\text{tot}}/N_{\text{forms}}$ programs have been produced using F. Thus, the cost limit C is deepened iteratively [4] with a branching factor of 3. It is not possible to use the same final cost limit for all forms since the cost-work relationship varies too much from form to form.

5. The overall search for an appropriate program

Using compound transformation forms as described in the previous section, an expansion of a parent program P produces children programs $\mathtt{P}_1$, $\mathtt{P}_2, \ldots, \mathtt{P}_{W_{\text{tot}}}$.

The overall search uses a population of programs. The population is initialized to a single program that consists of a single ?. The population is partitioned into classes such that all programs in a class contain the same number of `case`-expressions. The purpose of this partitioning is to maintain diversity and to ensure that programs with low `case` counts are expanded before programs with high `case` counts.

Each class contains three programs. Program number i in class number c is the best program found so far according to program evaluation function $\mathtt{pe}_i$ that contains exactly c `case`-expressions. Assume that c_{best_i} is the `case` count of the best program found so far as judged by $\mathtt{pe}_i$. ADATE tries to avoid futile expansions by only expanding programs with a `case` count that does not exceed

$$\lceil \max(1.2c_{best_1}, 1.2c_{best_3}) \rceil.$$

The `case` count c_{best_2} is omitted since $\mathtt{pe}_2$ prefers low call count to small syntactic complexity. If the arguments of the max function above also included $1.2c_{best_2}$, this preference may lead to very big programs through sequences of R-transformations that unfold function calls.

The search is run iteratively with $W_{tot} = 10^4$ for the first iteration, $3 \cdot 10^4$ for the second iteration, $9 \cdot 10^4$ for the third iteration and so forth, i.e., $W_{tot} = 3^{i-1} \cdot 10^4$ for iteration number i.

A program is eligible for expansion only if it is better than all its ancestors according to at least one pe_i and has not been expanded thus far during the current iteration, which terminates when no program is eligible. Out of all eligible programs, a program with minimum `case` count is chosen as the next to be expanded.

6. Additional implementation details

In order to reduce run times, the implementation uses additional restrictions on atomic transformations. The implementation of replacement is such that only 0 or 1 subexpressions of the "old" expression *Sub* are reused in the synthesized expression *Synt*. When one subexpression *Sub_sub* is to be reused, the expression synthesis algorithm uses a special variable χ to represent *Sub_sub*. If the synthesis algorithm produces the expression *Raw_synt*, the finished synthesized expression *Synt* is *Raw_synt* with *Sub_sub* substituted for χ. Assuming that no A or E contains any `case`, the implementation chooses *Raw_synt* according to one of the following expression forms.

(1) E.

(2) `case` A `of` $Match_1$ `=>` E_1 | $\cdots$ | $Match_n$ `=>` E_n

(3)
```
case A of
    Match_1 => E_1
  ⋮
  | Match_i => case A' of Match'_1 => E'_1 | ··· | Match'_n' => E'_n'
  ⋮
  | Match_n => E_n
```

Thus, *Raw_synt* contains 0, 1 or 2 `case`s such that each `case` occurrence either is the root or a child of the root of *Raw_synt*. If N expressions are to be synthesized, $N/3$ *Raw_synt*s are chosen according to form (1), $N/3$ according to form (2) and $N/3$ according to form (3).

The implementation uses the following three types of replacements.

(1) a pure replacement that does not reuse any part of *Sub*,

(2) an insertion that only reuses *Sub* itself,

(3) a replacement that reuses one subexpression of *Sub* but not *Sub* itself.

If N replacements are to be done, 40% are of type (1), 40% of type (2) and 20% of type (3).

Abstraction transformations are restricted to introduce only `let`-functions of arities 1 or 2. If N abstractions are to be done, $N/2$ have arity 1 and $N/2$ have arity 2.

Since an inferred program P may have very bad time complexity, the number of calls to functions defined in P needs to be limited. The current version of ADATE uses a call count limit of 200 when computing $f(I_i)$. The upper limit on the total number of calls is thus 200# I. The fixed 200 limit is somewhat arbitrary and may in the future need to be replaced by an iterative-deepening scheme.

7. Sample specifications, inferred programs and run times

Polynomial simplification. This problem was discussed in Section 2.2. The specification consisted of

(1) The type `int` and the type declaration `datatype 'a list = nil | :: of 'a * 'a list`.
(2) The primitives `= : int * int -> bool` and `+ : int * int -> int`.
(3) The type of the function to be inferred i.e., `(int*int) list -> (int*int) list`. Recall that a polynomial is represented as a list of (coefficient,exponent) pairs.
(4) The following sample inputs.

```
I1 = []
I2 = [(3,2)]
I3 = [(3,2), (5,2), (12,2), (11,2)]
I4 = [(57,0), (71,4), (37,3), (117,1), (13,2), (19,4),
      (31,0), (53,1), (67,3), (87,4)]
```

(5) The output evaluation function shown in Fig. 1.

Note that these four sample inputs were chosen to facilitate incremental inference. I_1 is an empty polynomial. I_2 consists of only one term. All terms in I_3 have the same degree. I_4 is a "random polynomial". Thus, I_1, I_2 and I_3 are special cases which it may be advantageous to learn to simplify before trying to simplify general polynomials such as I_4.

With this specification, ADATE inferred a polynomial simplification program which below is shown exactly as it was printed by the system.

```
fun f (V3_0) =
  case V3_0 of
    nil => V3_0
  | ((V4996_0 as (V4997_0, V4998_0)) :: V4999_0) =>
  let
    fun g5011724_0 (V5011725_0) =
      case V5011725_0 of
        nil => (V4996_0 :: nil)
      | ((V5000_0 as (V5001_0, V5002_0)) :: V5003_0) =>
      case (V5002_0 = V4998_0) of
        true => (((V4997_0 + V5001_0), V4998_0) :: V5003_0)
      | false =>
      (V5000_0 :: g5011724_0(V5003_0))
  in
    g5011724_0(f(V4999_0))
  end
```

This program is equivalent to the one below in which identifiers generated by the system have been replaced by more readable identifiers.

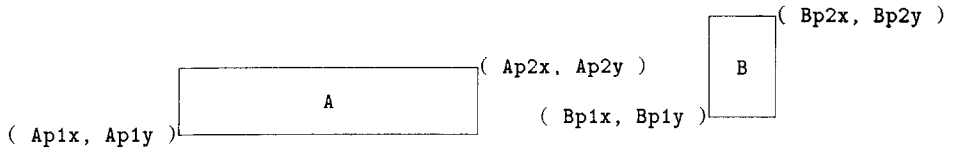

Fig. 2. Two non-intersecting rectangles and their coordinates.

```
fun simplify Xs =
  case Xs of nil => nil
  | (X1 as (X1c,X1e)) :: Xs1 =>
  let fun g Ys =
    case Ys of nil => X1::nil
    | (Y1 as (Y1c,Y1e)) :: Ys1 =>
    case Y1e = X1e of true => (X1c+Y1c, X1e) :: Ys1
    | false => Y1 :: g Ys1
  in
    g(simplify Xs1)
  end
```

The auxiliary function g, which was invented by the system, is such that the call g Ys tries to merge X1 with a term in Ys

Rectangle intersection. This is one of the few problems for which an input–output pair specification is adequate. The rectangles may be viewed as windows occurring in a graphical user interface. The overlap between a foreground window and a background window needs to be updated when the latter is moved into the foreground, i.e., made entirely visible. Each rectangle is represented by a pair of points which in turn are pairs of integers specifying the coordinates of the lower left and the upper right corners. Fig. 2 shows the representation of two rectangles A and B.

The specification contained

(1) The type `int` and the type declaration `datatype 'a option = none | some of 'a`.
(2) The primitive `< : int * int -> bool`.
(3) The type of the function to be inferred. The type is

```
((int * int) * (int * int)) * ((int * int) * (int * int)) ->
((int * int) * (int * int)) option.
```

(4) A set of 50 sample inputs consisting of each pair of rectangles such that the big rectangle in Fig. 3 is either the first or the second rectangle and such that the other is one of the 25 small rectangles.
(5) An output evaluation function that knows the correct output for each sample input.

The value returned by a correct rectangle intersection program is `none` if the two input rectangles do not intersect and `some C` if their intersection is the rectangle C.

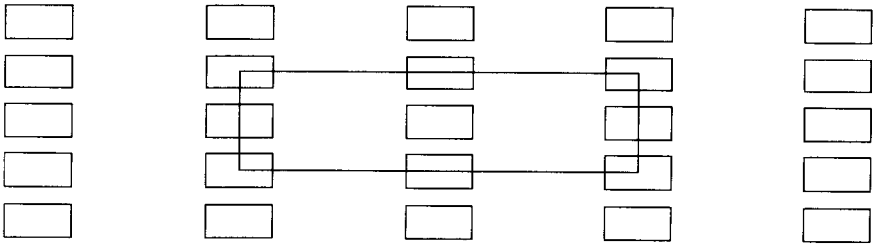

Fig. 3. The set of input rectangles.

After renaming, the inferred program is as follows.

```
fun rect_is(I as (A as (Ap1 as (Ap1x,Ap1y),
                        Ap2 as (Ap2x,Ap2y)),
                  B as (Bp1 as (Bp1x,Bp1y),
                        Bp2 as (Bp2x,Bp2y)))) =
  case Ap1x<Bp2x of
    true => (case Ap2x<Bp1x of true => none
              | false =>
              case Ap1y<Bp2y of
                true => (case Ap2y<Bp1y of true => none
                          | false =>
                          some((case Bp1x<Ap1x of true => Ap1x
                                  | false => Bp1x,
                                  case Ap1y<Bp1y of true => Bp1y
                                  | false => Ap1y),
                               (case Bp2x<Ap2x of true => Bp2x
                                  | false => Ap2x,
                                  case Ap2y<Bp2y of true => Ap2y
                                  | false => Bp2y)))
              | false => none)
  | false => none
```

If two input rectangles A and B intersect, the output of this program is

```
some((max(Ap1x,Bp1x),max(Ap1y,Bp1y)),
     (min(Ap2x,Bp2x),min(Ap2y,Bp2y)))
```

This algorithm is not obvious even though both the algorithm and the specification are simple.

BST deletion. The problem is to delete an element from a binary search tree with integers in the nodes. The specification contained

(1) The type `int` and the type declaration `datatype 'a bin_tree = bt_nil | bt_cons of 'a * 'a bin_tree * 'a bin_tree`.

(2) The primitive `< : int * int -> bool`.

(3) The type of the function to be inferred, i.e., `int * int bin_tree -> int bin_tree`.

(4) Eight sample inputs. Assume that the element `X` is to be deleted from the BST `Xs` and that `bt_cons(X,Ls,Rs)` is a subtree of `Xs`. The inputs cover the following four cases.

Ls	Rs
`bt_nil`	`bt_nil`
`bt_nil`	`bt_cons(_,_,_)`
`bt_cons(_,_,_)`	`bt_nil`
`bt_cons(_,_,_)`	`bt_cons(_,_,_)`

(5) An output evaluation function that uses inorder listing and deletion for lists to check that the correct element is deleted. Note that it is possible to define this function without knowing any good way to delete an element from a BST. The output evaluation function oe uses the following auxiliary definitions.

```
fun inorder bt_nil = nil
  | inorder(bt_cons(RoXs,LeXs,RiXs)) =
    inorder LeXs @ RoXs::inorder RiXs

fun depth bt_nil = 0
  | depth(bt_cons(_,LeXs,RiXs)) = 1+max(depth LeXs,depth RiXs)

fun delete_one(_,nil) = nil
  | delete_one(X,Y::Ys) = if X=Y then Ys else Y::delete_one(X,Ys)
```

Given input `(X,Xs)` and output `Ys`, oe checks that

```
inorder Ys = delete_one(X,inorder Xs) andalso
depth Ys <= depth Xs.
```

If the depth requirement `depth Ys <= depth Xs` is omitted, ADATE infers a BST deletion program that produces very unbalanced outputs. With the depth requirement, the following program was inferred.

```
fun bst_del(I as (X,Xs)) =
  case Xs of bt_nil => Xs
  | bt_cons(RoXs,LeXs,RiXs) =>
  case RoXs<X of true => bt_cons(RoXs,LeXs,bst_del(X,RiXs))
  | false =>
  case X<RoXs of true => bt_cons(RoXs,bst_del(X,LeXs),RiXs)
  | false =>
  let fun g Ys =
    case Ys of bt_nil => LeXs
    | bt_cons(RoYs,LeYs,RiYs) =>
    case LeYs of bt_nil => bt_cons(RoYs,LeXs,bst_del(RoYs,RiXs))
    | bt_cons(RoLeYs,LeLeYs,RiLeYs) => g LeYs
```

```
in
  g RiXs
end
```

The most innovative part of this program is the `let`-expression, which determines what to do when the element to be deleted has been found.

BST insertion. This problem is to insert an integer into a binary search tree. In addition to the `datatype`-definition for binary trees, the specification contained the relation < on integers. No auxiliary function was needed.

List reversal. The specification contained the `datatype`-definition for lists. An auxiliary function, that inserts an element last in a list, was inferred.

List intersection. The problem is to compute the intersection of two lists of integers. The specification contained the `datatype`-definition for lists and the relation = on integers. An auxiliary function, that checks if an element occurs in a list, was inferred.

List delete min. The problem is to delete exactly one occurrence of the minimum element in a list. The specification contained the `datatype`-definition for lists and the relation < on integers. The sample inputs and the inferred program were presented in Section 4.2.

Permutation generation. The problem is to compute all permutations of a list of integers. The specification contained the `datatype`-definition for lists and the function @ that concatenates two lists. The output evaluation function measured the number of different permutations occurring in the output and checked that the output only consisted of permutations. The inferred program contains two auxiliary functions.

List sorting. The specification contained the `datatype`-definition for lists and the relation < on integers. The sample inputs and the inferred program were presented in Section 4.2.

List splitting. The specification contained the `datatype`-definition for lists. The output evaluation function was described in Section 2.1.

The run times shown in Table 2 were obtained using the Standard ML of New Jersey compiler and a SUN SparcStation 10. Note that the table shows the times required to find correct programs. In general, there is no guarantee that a correct program also is small and efficient.

8. Related work

The inference of LISP programs from input–output pairs is surveyed by D.R. Smith [8]. Smith writes that all the methods in his survey stem from Summer's [10] insight

Table 2
Run times

Problem	Run time in days:hours
Polynomial simplification	0:7
Rectangle intersection	1:18
BST deletion	7:12
BST insertion	3:5
List reversal	0:10
List intersection	6:3
List delete min	8:8
Permutation generation	9:5
List sorting	1:12
List splitting	0:7

that a semi-trace of a computation can be constructed from well chosen input–output pairs. Summer's THESYS system then uses the semi-trace to construct the corresponding LISP program.

Example 8.1. Assume that the input–output pairs are `([1],1)`, `([1,2],2)` and `([1,2,3],3)`. If the input is $\texttt{Xs}_i$, each output $\texttt{Y}_i$ can be described as follows using Standard ML notation.

$$\texttt{Y}_1 = \texttt{hd Xs}_1, \quad \texttt{Y}_2 = \texttt{hd(tl Xs}_2\texttt{)}, \quad \texttt{Y}_3 = \texttt{hd(tl(tl Xs}_3\texttt{))}.$$

THESYS notes that $\texttt{Y}_i$ equals $\texttt{Y}_{i-1}$ with `tl` $\texttt{Xs}_i$ substituted for $\texttt{Xs}_{i-1}$. This recurrence relation is then employed to infer a function that finds the last element in a list.

The inference method used by THESYS is highly specialized and requires that the structure of the input–output pairs directly corresponds to a specific program.

Unfortunately, this also holds for inductive logic programming (ILP) systems. The four problems with input–output pair specifications, which were presented in Section 2.1, also apply to ILP specifications. A particularly interesting development in ILP is the invention of new predicates, which is reviewed by Irene Stahl [9]. A new predicate is introduced using so-called intra-construction, which is based on inverse resolution. When executing a logic program, a resolution step corresponds to a function call in a functional program. Intuitively, inverse resolution corresponds to "inverse function call" i.e., replacing an instantiation of the right-hand side of a function definition with the corresponding instance of the left-hand side. As described in Section 4.2.2, this is done by an abstraction transformation, which is therefore analogous to predicate invention. However, the abstraction transformation was developed independently of any previous work, including predicate invention.

One major difference between abstraction and predicate invention is the choices that need to be made to determine the initial definition of the invented function or predicate. Many ILP systems that do predicate invention, e.g. CIGOL [7] and SIERES [12], ask the user to confirm the usefulness of an invented predicate. Another criteria of usefulness that is employed is the size of the resulting program. Irene Stahl concludes

that "Additionally, the experimental evaluation of systems performing predicate invention in ILP is almost lacking".

The specifications used by GA systems [5] are similar to ADATE specifications. The main program transformation is crossover, i.e., random exchange of subexpressions between two programs. This is a primitive program transformation indeed.

Crossover is only effective if the schema theorem [3] is applicable. In general, this means that if a large expression E is to be inferred, it should primarily be composed from first- or higher-order subexpressions $E_1, E_2, \ldots, E_n$ such that the fitness advantage of each E_i can be measured independently of each E_j with $j \neq i$. Unfortunately, practically all recursive programs consist of coupled E_i's.

Example 8.2. Consider the following ML list concatenation program, which is written using `if` and selectors instead of `case` in order to make it resemble Koza's LISP style [5].

```
fun @(Xs,Ys) = if null Xs then Ys else hd Xs::@(tl Xs,Ys)
```

The right-hand side can be written as E_1E_2 with

E_1 = `fn As => if null Xs then Ys else As`

and

E_2 = `hd Xs::@(tl Xs,Ys)`.

The fitness advantage of E_2 can obviously not be measured unless the base case of the recursion is properly handled. Thus, E_2 has a positive effect on fitness only if it appears in conjunction with E_1 or some equivalent expression.

This so-called "subexpression coupling problem" means that crossover is an extremely inefficient program transformation when recursive programs are to be inferred. Therefore, it is quite natural that Koza's book does not list any inferred programs that contain explicit recursive calls.

The inability to infer recursive programs is most unfortunate since recursion is of fundamental importance in LISP and functional programming. In general, it seems to be equally difficult for Koza's system to produce iterative programs. This means that the current form of the system is unlikely to ever become an effective tool for general purpose programming.

9. Conclusions and future work

The main advantages of ADATE are

(1) The abstraction transformation can invent auxiliary functions, which the user might be unaware of.
(2) The embedding transformation can change the type of a function in order to make the function more general.
(3) Specifications are loose.

(4) The ability to automatically invent nontrivial algorithms.

The main disadvantage is the long run times. The systems for induction of logic programs reviewed in Section 8 are much faster. However, they do need to acquire much more knowledge from the users.

ADATE finds "good" programs through a combination of thorough testing and attempted minimization of syntactic complexity. There is no guarantee that ADATE will find a program that is optimal according to some program evaluation function pe_i. For example, if ADATE always guaranteed to find a correct program of minimum syntactic complexity, run times would in general grow exponentially with complexity. The ability to give such a guarantee would therefore have little practical value. Fortunately, many users are satisfied with a program that is correct and reasonably small and fast, but not necessarily the smallest nor the fastest. This situation is analogous to the one for many NP-hard problems, where a solution within say 1% of the optimum can be found in polynomial time with high probability, even though the worst case time complexity for finding an optimal solution is exponential.

Some possible improvements are

(1) to generalize embedding to arbitrary insertions into type expressions;
(2) to generalize abstraction so that higher-order functions can be invented;
(3) to add more heuristics to the algorithms that synthesize expressions and compound transformations;
(4) to significantly improve run times by implementing ADATE on a high performance massively parallel computer.

All programs inferred so far are rather small. The most important future work is to study the inference of large programs. Recall that m_i is the number of symbols that may occur in node N_i in an expression tree. A potential problem with inference-in-the-large is that m_i grows with the number of ancestor `let-` and `case-`nodes, since such nodes introduce new symbols. More experimentation is needed to determine if the scoping rules of Standard ML suffice to keep m_i small or if additional symbol selection techniques are required. A related question is the use of library functions versus the invention of functions on-the-fly i.e., if the system should rely on a general toolbox or on the construction of specialized tools as needed. In comparison with human programmers, a system for inference-in-the-large is likely to rely less on general tools since the use of such tools seems to be combinatorially expensive.

Acknowledgements

I am indebted to Olaf Owe at the University of Oslo for providing constructive comments on a draft of this paper. The most important influence on the work presented in the paper comes from the functional programming community. This work would never have been done without the foresight shown by Åke Wikström of Chalmers when teaching functional programming [11] to freshmen, one of which I had the fortune to be.

References

[1] A.W. Biermann and R. Krishnaswamy, Constructing programs from example computations, *IEEE Trans. Softw. Eng.* **2** (1976) 141–153.

[2] A.W. Biermann, The inference of regular LISP programs from examples, *IEEE Trans. Syst. Man Cybernet.* **8** (1978) 585–600.

[3] J.H. Holland, *Adaptation in Natural and Artificial Systems* (University of Michigan Press, Ann Arbor, MI, 1976).

[4] R.E. Korf, Depth-first iterative-deepening: an optimal admissible tree search, *Artif. Intell.* **27** (1985) 97–109.

[5] J.R. Koza, *Genetic Programming* (MIT Press, Cambridge, MA, 1992).

[6] S.H. Muggleton, Inductive logic programming, in: S. Muggleton, ed., *Inductive Logic Programming* (Academic Press, London, 1992) 4–21.

[7] S.H. Muggleton and W. Buntine, Machine invention of first-order predicates by inverting resolution, in: *Proceedings Fifth International Conference on Machine Learning*, Ann Arbor, MI (1988) 339–352.

[8] D.R. Smith, A survey of the synthesis of LISP programs from examples, in: A.W. Biermann, G. Guiho and Y. Kodratoff, eds., *Automatic Program Construction Techniques* (Macmillan, New York, 1982) 307–324.

[9] I. Stahl, Predicate invention in ILP—an overview, in: *Proceedings European Conference on Machine Learning* (Springer Verlag, Berlin, 1993) 313–322.

[10] P.D. Summers, A methodology for LISP program construction from examples, *J. ACM* **24** (1977) 161–175.

[11] Å. Wikström, *Functional Programming Using Standard ML* (Prentice Hall, Englewood Cliffs, NJ, 1987).

[12] R. Wirth and P. O'Rorke, Constraints for predicate invention, in: S. Muggleton, ed., *Inductive Logic Programming* (Academic Press, London, 1992) 299–318.