Inductive Functional Programming Using Incremental Program Transformation

Roland Olsson

A thesis presented to the University of Oslo
in fulfilment of the thesis requirement for the degree of
Doctor Scientiarum
in
Computer Science

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Research Perspectives

The research presented in this part of the thesis belongs to the following areas.

1. Computer-aided software engineering (CASE).
2. Machine learning.
3. Combinatorial optimization.

We will first describe what we want to accomplish from a CASE perspective. However, since we aim for a very high degree of automation, the machine learning and combinatorial search perspectives are more important.

1.1.1 The CASE Perspective

The development of a program to solve a given problem consists of many activities. A software engineer may proceed as follows [Boehm 76].

1. Determine the requirements that the program should satisfy and write a specification.
2. Design algorithms and data structures.
3. Implement the design in a programming language.
4. Test the implementation.

The software engineering literature contains numerous more sophisticated development models such as the spiral model [Boehm 88]. We will use the simple model above to explain our research goal even though it may be too

sequential, phase-oriented and crude for practical purposes. Our research goal is to completely automate phases 2 and 3 and to partially automate phase 4.

We have developed a system, ADATE, that automatically can design, implement and test programs. 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. As we will see, this means that phase 1 i.e., specification, becomes more difficult. In particular, the ADATE user needs to give a specification that supports automatic and incremental program development.

1.1.2 The Machine Learning Perspective

The main part of machine learning is inductive inference, which is the process of finding generally valid rules from a finite number of examples. When inferring programs, the examples may be input-output pairs, which is a primitive form of specification indeed. However, a simple input-output pair specification suffices to illustrate inductive inference of programs.

Example. Assume that a list concatenation algorithm, implemented in Standard ML, is to be inferred. The specification says that the type of the function to be inferred is `'a list * 'a list -> 'a list` and that the empty list `nil` and the list constructor `::` may be used in inferred programs. Also, the specification contains the single input-output pair

```
( ([1,2,3,4],[5,6,7,8,9]), [1,2,3,4,5,6,7,8,9] ).
```

The least complex definition that satisfies this specification is

```
fun append(Xs,Ys) =
  case Xs of
    nil => Ys
  | X1::Xs1 => X1::append(Xs1,Ys)
```

□

We formally define the complexity of a program in Subsection 3.5.1. This formal definition approximates a programmer's intuition regarding program simplicity. There are infinitely many "undesirable" programs that satisfy the specification above e.g.

```
fun append(Xs,Ys) =
  case Xs of
    nil => nil
  | X1::Xs1 =>
  case Xs1 of
    nil => X1::Ys
  | X2::Xs2 => X1::append(Xs1,Ys)
```

However, every undesirable program is more complex than the one we want. This is no coincidence. The small-is-beautiful assumption is also inherent in scientific theory formation. One may even argue that science as we know it today would not exist without it. We discuss the relationship between desirability and complexity in Section 3.2.

Note that the above program easily can be made "desirable" by substituting `Ys` for `nil`. This substitution makes the second `case` redundant. Removing this redundancy gives the simplest desirable `append` definition. These two transformations are actually very simple illustrations of the evolutionary "replacement" (R) transformation presented in Chapter 4.

The specifications employed by ADATE are certainly not restricted to input-output pairs, which are inadequate for many interesting programming tasks. Instead of outputs, ADATE uses an evaluation function, which is defined by the specifier based on the requirements that the outputs must satisfy. For example, the specifier may require that the output from a sorting algorithm is both

1. sorted according to some total ordering and
2. a permutation of the input.

The disadvantage of providing outputs is not evident from this simple example, but becomes quite obvious for more complicated programming tasks e.g. autonomous robot navigation.

1.1.3 The Perspective of Combinatorial Optimization

A combinatorial optimization problem defines a set of solutions and a cost function that determines the quality of each solution. In ADATE, the solutions are correctly typed Standard ML programs that satisfy some additional constraints, for example recursion restrictions. The cost function, which is to be minimized, is called a program evaluation function. Program evaluation functions are automatically defined by ADATE using universal program quality measures such as time complexity and syntactic complexity together with the output evaluation function provided by the specifier.

The search employed by ADATE is a form of local optimization. Given an initial solution, local optimization finds better and better solutions through a

```
fun local_search(S : solution) : solution =
  let
    val N = A minimum cost neighbour of S
  in
    if cost N < cost S then local_search N else S
  end
```

Figure 1.1: Basic local search.

series of incremental changes. In ADATE, these changes are called program transformations. The set of all solutions that can be obtained from a given solution S using only one transformation is called the neighbourhood of S. Local search moves from neighbour to neighbour as long as the cost decreases. When no neighbour of the current solution has lower cost, we are at a local optimum, which of course is not guaranteed to be a global optimum. Figure 1.1 shows one version of local search, where `S` is the current solution. Another version just picks a neighbour of lower, but not necessarily minimal, cost. We employ the former version, which means that the entire neighbourhood is examined before moving to a neighbour.

Most, but not all, "good" transformations yield an increase in program complexity, which means that the search in ADATE usually performs successive augmentation. The following example shows how the `append` program above may be produced using incremental transformations.

Example. Assume that there are two input-output pairs, namely

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

ADATE uses a special constant, `?`, which means "don't-know". Intuitively, it is better to say "don't-know" than give a wrong answer. In this example, we will only employ the replacement (R) transformation, which replaces an expression in a program with a new, small synthesized expression or inserts such an expression into a program. Thus, the neighbourhood of a program is the set of all programs that can be obtained from it using R transformations with a transformation complexity that does not exceed the current transformation complexity limit, which is iteratively deepened by ADATE. The initial program is

```
fun append(Xs,Ys) = ?
```

This program gives a "don't-know" answer for all inputs. It is improved by replacing the `?` with `Ys`, which yields

```
fun append(Xs,Ys) = Ys
```

This program is an improvement since it correctly handles input number 1. The next transformation is an insertion that gives

```
fun append(Xs,Ys) = case Xs of nil => Ys | X1::Xs1 => ?
```

The reason this program is better is that it does not give a wrong output for input number 2 while still being able to handle input number 1. The final and desirable list concatenation program is obtained by replacing the `?` with the synthesized expression `X1::append(Xs1,Ys)`.

Note that we used three transformations and a search space trajectory containing four programs even for this very simple sample inference. However, the search algorithms in ADATE are powerful enough to find the final program using only one single R transformation applied to the initial program. Also, ADATE manages fine without input-output pair number 1. □

The point with the example above is to show that even very simple programs have many better-and-better intermediate forms. For more complicated programs, we do need specifications that give smooth search spaces with chains of gradually improving programs such that there are no weak links requiring too big neighbourhoods.

The search in ADATE is a heavily modified form of the traditional local search algorithms for problems such as graph partitioning, graph colouring, bin packing etc. Some important modifications are

1. Iterative-deepening [Korf 85, Olsson 93] of neighbourhood cardinality.

2. Using three multiple-valued cost functions instead of only one cost function that returns only one numerical value.

3. Maintaining a structured so-called population of programs instead of just one single current program.

The Traveling Salesman problem (TSP) is often considered to be the prototypical "hard" combinatorial optimization problem [Johnson 90]. However, even very large instances of this problem, e.g. one million cities, can be solved within 2% of the optimum with high probability in a few hours of CPU time [Johnson 94]. Therefore, we hope not to scare the reader by saying that the search problem in ADATE is much harder than the TSP. Even though the TSP is very well studied in complexity theory, the practical, experimental results are more significant and not well described by theory. Since it is so difficult to find reasonably exact theoretical bounds on the average time complexity of TSP algorithms, we cannot expect to find such theoretical complexity results for the much more complicated problem of program synthesis as in ADATE. Therefore, our general methodology is empirical end experimental.

The development, debugging and evaluation of ADATE has consumed about 4000 hours of CPU time on an IBM RS6000-590. In spite of careful programming, the debugging phase has consumed more than 80% of this time. We feel that our experimental activity is on the limit between the feasible and the unfeasible using a modern workstation. However, given massively parallel computers and further research and development, it is impossible to tell where this limit will be.

1.2 Design Challenges and Choices

The most important overall design choice in ADATE is the high degree of automation, which is related to the amount of information in ADATE specifications. If a specification contains much explicit information or if system-user interaction is allowed, the system does not need to be particularly autonomous. We will use the explicit information contents in specifications to briefly compare inductive inference systems.

At one end of the spectrum of explicit information contents are systems that use traces of computations [Biermann and Krishnaswamy 76]. At the same end of the spectrum are systems requiring specifications that consist of input-output pairs [Biermann 78, Smith 82, Summers 77] or positive and negative examples as in inductive logic programming [Muggleton and Buntine 88, Muggleton 92, Stahl et. al. 93, Wirth and O'Rorke 92]. In such systems, the input-output pairs or the examples must have a structure that corresponds to a specific algorithm.

At the other end of the spectrum are genetic algorithm (GA) systems [Koza 92] 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. ADATE uses specifications that contain few constraints on the programs to be synthesized and that allow a wide range of correct programs.

Of course, there are many design choices other than the degree of automation and the amount of explicit information in specifications. For each relevant chapter in this part of the thesis, the following listing shows the design choices that are discussed in the chapter. We also give a brief general introduction to each chapter.

Chapter 2. A difficult choice is which language to use for expressing the algorithms that are inferred. We started our program synthesis research believing that the simplest possible language would be most suitable for automatic programming. We looked at polycephalic Turing machines, finite state automatons (sequential nets) and subsets of LISP and PROLOG. Due to lack of time and knowledge, we omitted several candidates

such as Kolmogorov graph machines and neural nets. However, it is now clear that we do not need a simple language, but instead a language that allows simple algorithm formulations that are easy to transform and suitable for combinatorial search.

After many time-consuming bad starts concerning the choice of language, we have found that a subset of Standard ML, which we call ADATE-ML, is superior to the other candidates that we have tried. However, the choice of language depends on the class of algorithms to be inferred. For example, we cannot be sure that ADATE-ML is better than neural nets for pattern recognition applications.

ADATE is also implemented in Standard ML. Additionally, Chapter 2 contains some ML definitions that this implementation employs when manipulating ML programs.

Chapter 3. The form of specifications should be general enough to allow basically any kind of requirement to be formulated. It should also enable the inference system to recognize microscopic program improvements. Traditional predicate logic specifications do not satisfy this last requirement since they are often either "false" or "true", i.e., give an extremely rough search space topography even if they are supplemented with additional measures such as time complexity and syntactic program complexity.

The choice of sample inputs and an output evaluation function as in ADATE is much better than using input-output pairs. For example, the ADATE specification form is easy to adapt to the sort of "environment" simulation employed in artificial life research, but ADATE has primarily been used for the type of problems found in text-books on algorithm design and analysis.

Chapter 4. This chapter presents so-called atomic transformations. The programs in the neighbourhood of the program to be transformed are produced using so-called compound transformations as presented in Chapter 6. A compound transformation is a sequence of one or more "related" atomic transformations.

The choice of atomic transformations was made empirically. We started with the replacement (R) transformation and looked at sample inferences using only this transformation. We also implemented a precursor of ADATE that only employed a limited form of R transformations. The experimental results obtained using this precursor showed the need for a transformation that rearranges `case`-expressions, which is the so-called `case`-distribution (CASE-DIST) transformation. The next step in the evolution of ADATE was the observation that auxiliary functions could be extracted from already synthesized program fragments, which is done by the abstraction (ABSTR) transformation. We then noted that some

of these invented functions could be generalized by adding parameters or changing parameter types, which yielded the embedding (EMB) transformation.

A particularly difficult stage in this gradual evolution of the ADATE transformations was how to introduce recursive auxiliary functions. However, the current design is both simple and effective.

A major omission in ADATE is the ability to invent and utilize higher order functions. There are no principal problems with these, but they seem to have bad combinatorial properties if used without restrictions and heuristic guidance.

Chapter 5. The R transformation requires the synthesis of new expressions. The choice of expression synthesis techniques has a strong influence on the run times of ADATE since the number of expressions to be examined grows exponentially with expression size i.e., complexity. Therefore, the expression synthesis problem is intractable for large sizes. One may deal with this combinatorial explosion in the following three ways.

1. By ensuring that the size of synthesized expressions is always quite small.
2. By avoiding the synthesis of equivalent expressions.
3. By employing heuristics in order to try the "best" expressions first.

From the very beginning of the research presented in this thesis, we have focussed on ensuring that only very small expressions need to be synthesized.

Informally, we will now try to explain how this is possible by presenting an idealized program induction scenario. Let us define a *grain* to be a subexpression of the program to be inferred. Additionally, a grain is required not to contain any `case` and to have a `case` or a `let` as parent. This means that the grains are the biggest possible `case`-free subexpressions of a program.

Example. Consider the program

```
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 => X1::g Xs2 | false => X1::V1
  in
    g(sort Xs1)
  end
```

The grains are `Xs`, `Xs`, `V1`, `X1::nil`, `X2<X1`, `X1::g Xs2`, `X1::V1` and `g(sort Xs1)`. Note that the biggest grain is `X1::g Xs2` which means that the maximum grain size is 4. □

Let us define the *resolution* of an inference as the maximum number of grains that need to be added to any program P_i in order to improve the value of at least one program evaluation function. We require that each P_i is a program in a suitable "genealogical path" that leads to a final and desirable program. In the ideal scenario, the resolution is one grain. This means that there is a permutation of grains $G_1, G_2, \ldots, G_{\#G}$ such that we can obtain successively better programs by adding one grain G_i at a time with i taking the values $1, 2, \ldots, \#G$. Each added grain G_i is assumed to improve the value of at least one program evaluation function. The following three observations are crucial.

1. Empirically, we have found that many, perhaps even most, functional programs can be written in a form with a very small maximum grain size.
2. It is frequently possible to provide sample inputs and an output evaluation function that give a resolution of one or a few grains.
3. The product of the maximum grain size and the resolution gives an approximate upper bound on the total size of the expressions that need to be synthesized in any compound transformation.

Unfortunately, it is difficult to substantiate these claims by means other than empiricism. The examples in Chapter 8 will give at least some empirical motivation for the claims.

Even though our expression synthesis techniques and heuristics are reasonably simple, they were difficult to implement. Chapter 5 also presents alternative, unimplemented methods, but full exploration of all possible designs would require several additional Ph.D. theses.

Chapter 6. When having chosen atomic transformations, we need to choose how to combine them. For example, after introducing a new function using abstraction (ABSTR), this new function should be used immediately in a subsequent replacement (R). The reason is simply that a function must be used i.e., formally called, at least twice in order to serve any purpose. We define this coupling between atomic transformations using so-called coupling rules, similar to rewrite and production rules, which ADATE employs to automatically generate all possible compound transformations.

Chapter 7. The final choice in the design of ADATE is the overall combinatorial search technique that navigates through the program space by jumping from neighbour to neighbour using compound transformations. There are several combinatorial "meta-heuristics" that may be used as

templates for the overall search. Two such methods are simulated annealing [Kirkpatrick et. al. 83] and tabu search [Glover 89], both of which are based on local optimization. In comparison with plain local optimization methods, both simulated annealing and tabu search are substantially better at escaping from local optima but require much more execution time. Since we already are on the limit of the computationally feasible, we chose a special-designed overall search, which is likely to be at least two orders of magnitude faster than simulated annealing. Unfortunately, we have to sacrifice general and robust search performance in order to reduce the number of examined neighbourhoods and achieve this speed increase. When 10 to 100 times more computing power is available, we will use a randomized search that is better at escaping from local optima. However, the basic population structure and sophisticated iterative-deepening search presented in Chapter 7 will also be needed in future implementations. Therefore, there is practically no material in Chapter 7 that is likely to become obsolete in new versions of ADATE.

Chapter 8. This chapter contains examples of specifications and inferred programs. We used 10 sample specifications to evaluate ADATE. The inferred programs typically consist of less than 30 lines of ADATE-ML code, but are nevertheless mostly non-trivial. As far as we know, there is no other inference system in the literature that can infer even one single of these programs using specifications that contain as little explicit information as ours.

Desirable, non-trivial and unexpected programs were found surprisingly often. One reason is that recursive calls were employed in ways we could not anticipate. Our general impression is that the results of grand scale combinatorial search as in ADATE are unpredictable and that they would be less interesting if they were not.

Some of the 10 specifications, in particular BST deletion and permutation generation, were not easy to write since we had to change them several times in order to enable the overall search method to escape from local optima.

Appendix B shows a "raw" log file from the inference of a list sorting program.

Chapter 9. This chapter discusses categories of inductive inference systems that are related to ADATE, namely inductive logic programming, genetic algorithms (programming) and program transformation. We evaluate each of these categories with respect to seven common criteria.

Chapter 10. This chapter contains merits and drawbacks with ADATE and directions for future research.

Chapter 2

The Language in Which Synthesized Programs are Written

Synthesized programs are written in a purely functional subset, ADATE-ML, of Standard ML. We will first motivate the choice of ML, then compare ADATE-ML with Standard ML and finally explain how ADATE-ML programs are represented using algebraic data types. This explanation lays the ground for the algorithms that transform ADATE-ML programs.

2.1 Advantages of Functional Languages for Inductive Inference

We will first discuss the advantages of ADATE-ML in comparison with languages from the ALGOL family e.g. SIMULA, ADA, MODULA,.... Then, we will compare ADATE-ML with PROLOG and LISP.

For expressing inferred programs, a purely functional language has the following advantages in comparison with ALGOL-like languages.

1. A purely functional language is referentially transparent and has no notion of state nor destructive assignment. Consequently, it is relatively easy to define general, semantics preserving transformations of purely functional programs. Helmut Partsch [Partsch 90, page 263] writes that "experience has shown that it is advisable to do these manipulations on the applicative level as far as possible, thus profiting from the obvious advantages (such as referential transparency) of this level of formulation".

2. Another advantage of a functional language is that a functional program is often much smaller than the corresponding ALGOL-like program. This is particularly important when using search strategies that, in principle, exhaustively search subsets of the space of all programs. The cardinality of such a subset is often much lower for a purely functional language than for practically any other language.

3. ALGOL-like languages have type systems that are rigid and primitive in comparison with algebraic data types and Hindley-Milner polymorphic typing as in ML.

4. A purely functional language uses recursion instead of while-loops. It is easy to reformulate any program that contains while-loops using only recursion, but there are many recursive programs that require clumsy, unnatural formulations if only while-loops are allowed.

5. Most languages in the ALGOL family do not have automatic garbage collection, which necessitates arduous, explicit storage allocation and deallocation.

6. Functions are first-class citizens in functional languages i.e., may be used just like other values. Higher-order functions and λ-expressions provide excellent abstraction facilities, but ADATE-ML does not contain them. However, they may be useful in future versions and certainly are employed over and over again in the ML source code of ADATE itself.

7. Some functional languages, e.g. MIRANDA and HASKELL, have lazy evaluation, which allows more expressive formulations. However, we do not exploit lazy evaluation and use strict evaluation only.

Undoubtedly, LISP and PROLOG are the most popular high level languages for expressing programs that are synthesized by inductive inference systems. The comparison with these languages is more interesting since the advantages of ADATE-ML may be less obvious. Here are some problems with LISP and PROLOG.

1. Both LISP and PROLOG suffer from extremely poor type systems, which might lead to unnecessarily high cardinalities for the program space subsets that are searched. The combinatorial advantage of typing is illustrated by the following example.

   Example. Assume that a syntactically correct program is a string, $a_1 a_2 \ldots a_{\#a}$, of terminal symbols produced by a context-free grammar (CFG). Assume that the a_i's are drawn from sets of terminals that have cardinalities with the geometric average B for large $\#a$. The CFG normally constrains these terminal sets so that B is only a fraction of the total number of different terminals. There are $\Theta(B^{\#a})$ programs of size $\#a$. Type constraints may

be taken into consideration by assuming that t different types reduce B to B/t. Thus, the number of programs of size $\#a$ may be reduced from $\Theta(B^{\#a})$ to $\Theta((B/t)^{\#a})$ through typing.

For example, consider the CFG

$E \longrightarrow$ `a` | `b` | `f` E | `f'` E | `g` E | `g'` E

with the terminal symbols `a, b, f, f', g, g'` and the non-terminal symbol E. There are $\Theta(4^{\#a})$ syntactically correct strings of length $\#a$, i.e., $B = 4$ asymptotically. Assume that an E-string is a preorder listing of an expression and that the functions are typed as follows.

```
a : S    f : S->T    g : T->S
b : T    f': S->T    g': T->S
```

There are two types, `S` and `T`, i.e., $t = 2$. There are $\Theta((B/t)^{\#a}) = \Theta(2^{\#a})$ expressions of size $\#a$ that are both syntactically correct and correctly typed. Of course, there are many grammars and type assignments such that typing does not reduce B to B/t. □

2. PROLOG, but not LISP, lacks scoping and modularization facilities e.g. predicate definitions inside other definitions. Neither does PROLOG have higher order predicates as first class citizens.

3. LISP, but not PROLOG, lacks pattern-matching constructs and relies on test predicates and selectors instead e.g. `null`, `car`, `cdr`. The attempts to rectify this in for example COMMON LISP are quite primitive.

Since language advantages often are hard to prove with mathematical rigour, language discussions are often fruitless. The reader who still feels that SIMULA, ADA, COMMON LISP, SCHEME or PROLOG is superior for inductive inference may want to try to show this by replacing ADATE-ML and reimplementing ADATE. However, we believe that this endeavour would be much more difficult than our own implementation effort.

2.2 The Design of ADATE-ML

One design goal was to remove redundancy from Standard ML, which has many equivalent constructs. Normally, these constructs give programmers freedom of choice, but ADATE-ML should not contain "syntactic sugar" since this may lead to unnecessarily large search spaces. For example, the Standard ML expression

`let val V =` E_1 `in` E_2 `end`

is equivalent to

`(fn V =>` E_1`)` E_2

and to

`case` E_1 `of V =>` E_2`.`

ADATE-ML only allows the last form of this expression. ADATE-ML uses

`case` E `of true =>` RHS_1 `| false =>` RHS_2

instead of

`if` E `then` RHS_1 `else` RHS_2`.`

ADATE-ML also uses `case`-expressions and pattern-matching instead of discriminators and selectors, e.g.

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

is used instead of

```
case null Xs of true => nil | false => tl Xs.
```

Boolean operators can be replaced with `case`-expressions without any significant increase in code size or execution time. They are therefore superfluous and not allowed in ADATE-ML. The boolean expression E_1 `andalso` E_2 can be replaced with

`case` E_1 `of true =>` E_2 `| false => false`.

Similarly, E_1 `orelse` E_2 may be replaced with

`case` E_1 `of true => true | false =>` E_2.

and `not` E can be replaced with

`case` E `of true => false | false => true`.

ADATE-ML uses `case`-expressions instead of alternative left hand sides in `fun`-definitions. For example, the definition

```
fun l nil = 0
  | l (X1::Xs1) = 1 + l Xs1
```

is written as

```
fun l Xs =  case Xs of nil => 0 | X1::Xs1 => 1 + l Xs1.
```

ADATE-ML does not contain curried functions. The pattern in the left hand side of a `fun`-definition is required to be an n-tuple pattern with $n \geq 1$. 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.

ADATE-ML contains `datatype`-definitions of the following form.

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

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.

If the type of a `case`-analyzed expression E is defined by a `datatype`-definition, the patterns in the left hand sides of `case`-rules must correspond to the alternatives in the `datatype`-definition as follows. Let the `case`-expression be

`case` E `of` *Match*$_1$ `=>` *RHS*$_1$ | ... | *Match*$_n$ `=>` *RHS*$_n$

and assume that the `datatype`-definition for the type of E has the form given above. ADATE-ML requires $n = \#C$ and *Match*$_i$ = C_i(*Tuple-pat*$_i$) where *Tuple-pat*$_i$ is the fully layered tuple pattern for the tuple type $T_{i,1} * \ldots * T_{i,\#T_i}$.

The language restrictions presented so far do not significantly reduce the expressiveness of ADATE-ML. However, as mentioned above, the current version of ADATE-ML contains neither λ-expressions nor higher order functions, both of which are important ingredients of functional programming languages. These ingredients were omitted from ADATE-ML since more complicated program transformations would be required to utilize them effectively. Since these ingredients were omitted, each application of an expression E_1 to an expression E_2, i.e., E_1E_2, is such that E_1 is a function symbol.

The basic syntax of ADATE-ML expressions is specified by the grammar rule for the non-terminal *Exp* in Figure 2.1. In order to keep the grammar simple, it does not describe infix function applications even though these are allowed in ADATE-ML.

Exp $\longrightarrow$
  Id
| (*Exp_list*)
| *Id* (*Exp_list*)
| `case` *Exp* `of` *Rule_list*
| `let` *Dec_list* `in` *Exp* `end`

Id $\longrightarrow$ *All valid alphanumeric identifiers.*

Exp_list $\longrightarrow$
  Exp
| *Exp* , *Exp_list*

Rule_list $\longrightarrow$
  Pat `=>` *Exp*
| *Pat* `=>` *Exp* | *Rule_list*

Pat $\longrightarrow$
  Id
| (*Pat_list*)
| *Id* (*Pat_list*)
| *Id* `as` *Pat*

Pat_list $\longrightarrow$
  Pat
| *Pat* , *Pat_list*

Dec_list $\longrightarrow$ `fun` *Decs*

Decs $\longrightarrow$
  Id Pat `=` *Exp*
| *Id Pat* `=` *Exp* `and` *Decs*

Figure 2.1: The syntax of ADATE-ML expressions.

```
datatype ('a,'b)e =
  app_exp of { func : symbol, args : ('a,'b)e list, exp_info : 'a }
| case_exp of { exp : ('a,'b)e,
    rules : {pat:('a,'b)e,exp:('a,'b)e} list,
    exp_info : 'a }
| let_exp of {
    dec_list :
      {func:symbol,pat:('a,'b)e,exp:('a,'b)e,dec_info:'b} list,
    exp : ('a,'b)e,
    exp_info : 'a }
| as_exp of { var : symbol, pat : ('a,'b)e, exp_info : 'a }

type ('a,'b)d =
  { func : symbol, pat : ('a,'b)e, exp : ('a,'b)e, dec_info : 'b }
```

Figure 2.2: The data types for expressions and `fun`-declarations.

2.3 Basic Definitions for the Manipulation of ADATE-ML Programs

ADATE is implemented in Standard ML. We will also use Standard ML to present algorithms that manipulate ADATE-ML programs. Standard ML was chosen as a presentation language instead of pseudo-notation since the "level" of Standard ML is almost as high as the level of pseudo-notation and since Standard ML has precise and well-defined semantics.

ADATE-ML expressions and `fun`-declarations are represented using the data types `('a,'b)e` and `('a,'b)d` respectively. The type variable `'a` is the type of information that is associated with each expression. The type variable `'b` is the type of information that is associated with each `fun`-declaration. For example, this information can be the types of expressions and functions. Since `fun`-declarations can occur in expressions and vice versa, both the type constructor `e` and the type constructor `d` need `('a,'b)` as argument.

The definitions of `e` and `d` are shown in Figure 2.2. These data types are simple and easy to use when defining program manipulation functions, but may not be theoretically appealing since some values can be constructed in many different ways and since illegal values can be constructed. This is illustrated by the following two examples.

Example. The expression `f(X1,X2)` has at least two representations. The two representations below correspond to the expressions `f(X1,X2)` and `f((X1,X2))`. Assume that we have the binding

```
Args = [ app_exp{func="X1",args=nil,exp_info=NONE},
```

```
        app_exp{func="X2",args=nil,exp_info=NONE} ]
```

The two representations are then

```
app_exp{func="f",args=Args,exp_info=NONE}
```

and

```
app_exp{
   func="f",
   args=app_exp{func="tuple",args=Args,exp_info=NONE}::nil,
   exp_info=NONE}.
```

We always assume that the first form of representation is used. □

Example. The definition of `e` in Figure 2.2 regards patterns as expressions. For example, this means that the data type `('a,'b)e` allows `as`-bindings in places where the grammar in Figure 2.1 does not allow them, e.g. in

```
fun f A = g(A as (B,C)).
```

This `as`-binding is illegal in ML and does not make sense but can still be represented using the data type. □

When manipulating ADATE-ML expressions, it is frequently necessary to specify the positions of subexpressions. The *position* of a subexpression is a list `[` $P_1, P_2, \ldots, P_n$ `]` of natural numbers that correspond to the expression tree path that leads to the subexpression. Number P_i corresponds to going to child number P_i. The left-most child has number 0. The higher order function `pos_fold` shown in Figure 2.3 may be used to define many functions that employ positions. A function `pos_to_sub` that returns the subexpression `Sub` at position `Pos` in an expression `E` can be defined as

```
fun pos_to_sub(E,Pos) = pos_fold( #1, fn Sub => Sub, Pos, E ).
```

In order to produce names for functions and parameters, ADATE maintains a counter that is increased by one each time a new name is needed. If this counter has the value N, a function is called `g`N whereas a parameter is called `V`N. Since the counter contains 60 bits (two `int` values in Standard ML of New Jersey), the name supply is large enough for all practical purposes.

```
fun pos_fold( f : 'c * ('a,'b)e * pos -> 'c, g : ('a,'b)e -> 'c,
      Pos : pos, E : ('a,'b)e ) : 'c =
  case Pos of
    nil => g E
  | P::Ps =>
  case E of
    app_exp{args,...} =>
      f( pos_fold(f,g,Ps,nth(args,P)), E, Pos )
  | case_exp{exp,rules,...} =>
      if P=0 then
        f( pos_fold(f,g,Ps,exp), E, Pos )
      else
        f( pos_fold(f,g,Ps,#exp(nth(rules,P-1))), E, Pos )
  | let_exp{dec_list,exp,...} =>
      if P < length dec_list then
        f( pos_fold(f,g,Ps,#exp(nth(dec_list,P))), E, Pos )
      else
        f( pos_fold(f,g,Ps,exp), E, Pos  )
```

Figure 2.3: The definition of `pos_fold`.

Chapter 3

Specification and Selection of Programs

3.1 Basic Properties of Specifications

A specification implicitly defines a set C of correct programs. A program is correct if and only if it satisfies the specification. The person(s), who wrote the specification, want a program chosen from a set D of desirable programs. The software engineering discipline distinguishes between validation and verification. Validation of a program P means to check if $P \in D$ whereas verification checks if $P \in C$.

Ideally, a specification should be such that

1. $C \neq \emptyset$ (consistency),
2. $D \subseteq C$ (necessity) and
3. $C \subseteq D$ (sufficiency).

The ideal $C = D$ is rarely achieved. A specification that is necessary is sometimes also called "loose" since it does not unnecessarily restrict the set of programs that may be inferred. Many inductive inference systems use specifications that are not necessary, which means that one or more desirable programs are not allowed by the specifications. Usually, such specifications contain extra information that facilitate efficient inference by constraining the search. The basic philosophy of our work is to maximize the ease of specification by minimizing the amount of extra information. ADATE specifications are always necessary.

A major potential problem with practically all specifications employed in inductive inference is that they are not sufficient.

3.2 Attempts to Deal with Lack of Sufficiency

Lack of sufficiency is a fundamental problem that arises in practically all kinds of scientific theory formation and inductive inference. A thorough theoretical treatment of the problem is given by Li and Vitányi in their book on Kolmogorov complexity [Li and Vitányi 93]. They provide an illustration similar to the following. "If a man has seen the sun rise on the eastern side of his house every morning in his entire life, can he use these examples of sunrise to conclude that the sun will rise on the eastern side of the house the next morning?" If one requires examples to be absolutely sufficient, the answer is no, which indicates that it sometimes is unreasonable to insist on sufficiency.

Li and Vitányi discuss Occam's razor principle at length, giving many historical accounts of its importance. They cite many different formulations of this principle. For our purposes, the following version is the most suitable.

> Compute the set of hypotheses that agree as well as possible with the observations. Choose the simplest hypothesis in this set.

For example, they write that Albert Einstein developed his general theory of relativity because he was convinced that the special theory was not the simplest that can explain all observed facts.

In machine learning, there are several interesting theoretical approaches to the problem of constructing highly probable hypotheses, in particular Valiant's model of learning [Valiant 84] and the so-called Occam's razor theorem as stated in [Blumer et. al. 86]. This theorem is based on Valiant's model and Occam's razor principle.

The problem of finding a program $P_{\min}$, that satisfies the specification and has minimum syntactic complexity, is NP-hard even for very simple languages such as regular expressions [Angluin 78]. Occam's razor theorem says that a program P with reasonably small but not necessarily minimal syntactic complexity still is correct with high probability. The theorem is useful since it does not require minimization of syntactic complexity, which means that worst-case polynomial time learning methods are more likely to exist. An interesting question is which languages that may be used to formulate P. Blumer et. al. considers restricted languages such as geometric concepts and Boolean expressions. For the latter language, their result may be described as follows. Assume that the syntactic complexity $s(P)$ is the minimum number of bits required to encode the program P and that n is the number of examples. The theorem assumes that $s(P)$ is $O(s(P_{\min})^k n^\alpha)$, where k and α are constants such that $k \geq 1$ and $0 \leq \alpha < 1$. It is normally assumed that α is substantially less than one so that n^α rapidly becomes much smaller than n. Note that P is assumed to have a reasonably small, but not necessarily minimal, syntactic complexity. Occam's razor theorem says that P still is correct with high probability. The probability of correctness depends on k, α and n as described in [Blumer et. al. 86].

Even if this result is not directly applicable to the inference of general functional programs, it indicates that reasonably small syntactic complexity in combination with well chosen examples can achieve sufficiency with a high probability.

3.3 Specification Form

Some additional 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. 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 thesis 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.

Even if requirement 3 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. 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 $L_{\min}$. Thus, the correctness of P is 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.

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 algebraic data 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.

1. The choice of output sometimes reflects the particular algorithm that was used to construct it. The specification writer may need to know this algorithm to be able to provide appropriate 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.

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

```
( [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 predetermined 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.

3.4 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 Subsection 3.5.2, 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. For example, `Grades` may have the form `[Grade_1,Grade_2]`, where `Grade_1` is more important than `Grade_2`.

Example. 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

```
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
```

Figure 3.1: The output evaluation function for polynomial simplification.

check 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 Figure 3.1. 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. □

3.5 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, time complexity, error locality and lineage.

3.5.1 Syntactic Complexity

We define the syntactic complexity of a program P as $-\log_2 Pr(\xi = P)$ bits, where the random variable ξ is defined on a program space Φ. Let $\varphi(P) = Pr(\xi = P)$ be a predetermined distribution on Φ. Intuitively, the distribution φ should be such that $\varphi(P) > \varphi(Q)$ holds for all programs P and Q in Φ such that P is "simpler" than Q. In order to ensure that φ is a probability distribution, we should also have $0 \leq \varphi(P) \leq 1$ for all P in Φ and $\sum_{P \in \Phi} \varphi(P) = 1$.

Here are three ways of choosing Φ.

- Φ_1 = The set of all lexically correct programs.
- Φ_2 = The set of all syntactically correct programs.
- Φ_3 = The set of all type correct programs.

It is assumed that $\Phi_3 \subseteq \Phi_2 \subseteq \Phi_1$. The traditional choice when performing data compression is $\Phi = \Phi_1$. For example, using the so-called universal prior distribution for positive integers [Rissanen 82], we could define $\varphi(P) = K 2^{-log^* n}$, where the positive integer n is the number of lexemes in P and K is a normalizing constant such that $\sum_{P \in \Phi_1} \varphi(P) = 1$. The function $\log^*$ is defined as

$$\log^* n = \log n + \log\log n + \log\log\log n + \ldots$$

where the sum only includes the positive terms.

When performing data compression, Robert Cameron [Cameron 88] showed that $\Phi = \Phi_2$ gives better compression than $\Phi = \Phi_1$. Since the choice $\Phi = \Phi_3$ gives unnecessarily complicated and slow computation of syntactic complexity, we will choose $\Phi = \Phi_2$.

Cameron's encoding is lossless, which means that an encoded program can be decoded so that an exact copy of the original program is obtained. The

syntactic complexity estimate used by ADATE is based on lossy encoding. In particular, the exact choice of variable names is neglected. For example, the expressions

`case Xs of A1::As1 =>` RHS_1

and

`case Xs of B1::Bs1 =>` RHS_2

are viewed as having the same encoding if the substitution { `A1=B1, As1=Bs1` } unifies RHS_1 and RHS_2.

The complexity estimation algorithm partitions the nodes in expression trees into four classes namely `let`-nodes, `case`-nodes, other internal nodes and leaves. The complexity of an expression is computed by a preorder traversal of the expression tree. Let Pr_c be the probability that the next node to be encoded during such a traversal belongs to class c. Usually, ADATE employs the ad hoc choices $Pr_{\text{let}} = 0.025$, $Pr_{\text{case}} = 0.15$ $Pr_{\text{internal}} = 0.325$ and $Pr_{\text{leaf}} = 0.5$. High confidence estimation of these probabilities would require a rather large sample of typical inferred programs. Since it was difficult to find such a sample before ADATE was implemented, the ad hoc probabilities above were chosen. Experimentally, these probabilities have led to adequate syntactic complexity based differentiation of programs. Therefore, there is no compelling reason to change them in accordance with the sample of synthesized programs that now are available. Additionally, this sample is still too small to allow statistically justifiable estimation of universal probabilities since estimation using a small sample will yield estimates that are too tailored to the sample.

The scope rules of ML determine the set of symbols that may occur in a node. It is assumed that all symbols in the set have the same probability of occurring in the node. This means that $-\log_2(1/N) = \log_2 N$ bits are required to encode a symbol if the symbol set has cardinality N. Let

$N_{\text{internals}} =$ The number of different symbols that may occur in an internal node.

$N_{\text{leaves}} =$ The number of different symbols that may occur in a leaf.

$s(E) =$ The syntactic complexity of an expression E.

In principle, syntactic complexity is defined as follows.

$s(E) = -\log_2 Pr_{\text{leaf}} + \log_2 N_{\text{leaves}}$ if E is a leaf.

$s((E_1, \ldots, E_n)) = -\log_2 Pr_{\text{internal}} + \log_2 N_{\text{internals}} + \sum_{i=1}^{n} s(E_i)$.

$s(h\ E) = -\log_2 Pr_{\text{internal}} + \log_2 N_{\text{internals}} + s(E)$.

$s($`case` E `of` $Match_1$ `=>` E_1 | $\ldots$ | $Match_n$ `=>` $E_n) = -\log_2 Pr_{\text{case}} + s(E) + \sum_{i=1}^{n} s(E_i)$.

s(`let fun g(`V_1`,`V_2`,...,`V_n`) = `E_1` in `E_2` end`)$= -\log_2 Pr_{\texttt{let}} + s(E_1) + s(E_2)$.

Note that neither the number of components in a tuple nor the number of rules in a `case`-expression are encoded. It is assumed that these numbers can be determined using type information or, alternatively, that their contribution to the syntactic complexity is negligible.

The actual definition of syntactic complexity is somewhat more complicated than the one above. The expression E that is analyzed in a `case`-expression

`case` E `of` $Match_1$ `=>` E_1 `|` ... `|` $Match_n$ `=>` E_n

very rarely contains `let`-expressions or other `case`-expressions when n is two or more. Therefore, we have chosen $Pr_{\texttt{let}} = 0.0025$ and $Pr_{\texttt{case}} = 0.015$ inside `case`-analyzed expressions when n is two or more.

The exact definition of syntactic complexity is shown in Appendix A.

3.5.2 Time Complexity

A natural measure of time complexity is the total execution time required to compute $f(I_1), \ldots, f(I_{\#I})$. In practice, it is difficult to measure this time with sufficient accuracy since few computers have timers with sufficiently high resolution, e.g. 1 microsecond or less.

Another time complexity measure is the total number of function calls that are made during the computation of $f(I_1), \ldots, f(I_{\#I})$. This measure is also somewhat impractical since it would require much time to increase a counter each time a function is called. Therefore, ADATE only keeps track of the number of calls to the function f and the `let`-functions that are defined in a program P. Thus, the time complexity measure for P is the total number of such calls.

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 $f(I_i)$. Thus, the upper limit on the total number of calls is $200\#I$. The fixed 200 limit is somewhat arbitrary and may in the future need to be replaced by an iterative-deepening scheme.

3.5.3 Error Locality

We will first define the problem of computing error locality, then show that this problem is NP-hard and finally present a simple approximation algorithm that works well in practice.

For a given program P, let

$N_c =$ The number of correct outputs.

$N_w =$ The number of wrong outputs.

$N_d =$ The number of don't-know outputs.

Naturally, the sum $N_c + N_w + N_d$ equals the number $\#I$ of sample inputs. By replacing subexpressions of P with `?`-constants, we can decrease N_w and increase N_d. Usually, such replacements also decrease N_c. The error locality measure only considers replacements that give $N_w = 0$. Such replacements are assumed to eliminate all errors in the program P. Out of all replacements that give $N_w = 0$, the best replacements are the ones that maximize N_c. Let N_c' be the maximum. Intuitively, the difference $\#I - N_c'$ indicates how much transformation work that remains to be done in order to obtain a completely correct program. Therefore, this difference should be as small as possible.

Example. Consider a list sorting program `sort`. 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]`. Both of the following two programs have $N_c = 3$ and $N_w = 2$.

$P_1 =$
```
fun sort Xs = Xs
```

$P_2 =$
```
fun sort Xs =
      case Xs  of
        nil => Xs
      | X1::Xs1 =>
      case Xs1 of
        nil => Xs
      | X2::Xs2 => Xs
```

However, P_2 is better than P_1 since P_2 has $N_c' = 2$ whereas P_1 has $N_c' = 0$. □.

Since the semantics of the `?`-constant is such that $h(?) =$ `?` for all functions h, it is only necessary to consider replacing right hand sides of `case`-rules and `fun`-definitions with `?`-constants. Let

Correct = The set of all correct outputs.

Locations = The set of all right hand sides of `case`-rules and `fun`-definitions.

Wrong = The set of all wrong outputs.

Assume that *Correct*, *Locations* and *Wrong* are the three node partitions in a tripartite graph. There is an edge between an output R and a location *RHS* if and only if R changes to `?` when *RHS* is replaced by a `?`.

Example. The program P_2 above contains 5 right hand sides, namely

$RHS_1 =$ `case Xs of nil => Xs | X1::Xs1 =>`
`case Xs1 of nil => Xs | X2::Xs2 => Xs.`

$RHS_2 =$ `Xs` (position `[1]`).

$RHS_3 =$ `case Xs1 of nil => Xs | X2::Xs2 => Xs.`

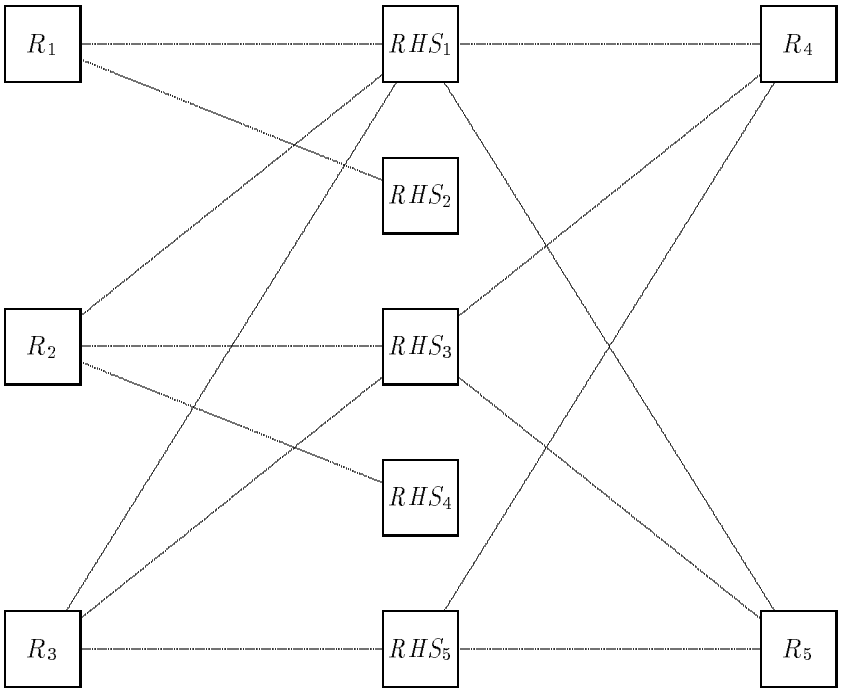


Figure 3.2: The tripartite graph for P_2.

$RHS_4 =$ `Xs` (position `[2,1]`).

$RHS_5 =$ `Xs` (position `[2,2]`).

Assume that output R_i corresponds to input I_i. The three node partitions are

$Correct = \{R_1, R_2, R_3\}$.

$Locations = \{\ RHS_1,\ RHS_2,\ RHS_3,\ RHS_4,\ RHS_5\ \}$.

$Wrong = \{R_4, R_5\}$.

The tripartite graph is shown in Figure 3.2. N_c' can be computed by finding a subset of *Locations* that is connected to all nodes in *Wrong* and to a minimum number of nodes in *Correct*. N_c' is then N_c minus this minimum number. In the `sort` example, such a subset is $\{RHS_5\}$ which gives

$$N_c' = N_c - \#\{R_3\} = 3 - 1 = 2.$$

We will now show that the problem of computing N_c' unfortunately is NP-hard. Using the style of [Garey and Johnson 79], we state the decision version of this problem as follows.

INSTANCE: An undirected tripartite graph $G = (V, E)$ and a positive integer K. Let *Correct*, *Locations* and *Wrong* be the three node partitions.

QUESTION: Is there a subset L of *Locations* such that

$$(\forall w \in \mathit{Wrong}.\exists l \in L.\{w, l\} \in E) \wedge |\{c : c \in \mathit{Correct} \wedge \exists l \in L.\{c, l\} \in E\}| \leq K?$$

We prove that this problem is NP-complete by a reduction from the NP-complete MINIMUM COVER problem in [Garey and Johnson 79]. The MINIMUM COVER problem is

INSTANCE: A collection C of subsets of a finite set S and a positive integer K.

QUESTION: Is there a collection C' such that

$$C' \subseteq C \wedge |C'| \leq K \wedge \forall s \in S.\exists \sigma' \in C'.s \in \sigma'?$$

Given an instance of MINIMUM COVER, the corresponding instance of our problem is constructed as follows. Choose *Locations* so that there is a bijection between *Locations* and C, i.e., each location l corresponds to one and only one subset $\sigma \in C$. Choose *Correct* so that there is a bijection between *Correct* and *Locations*. The edges between *Correct* and *Locations* are given by this bijection. Choose *Wrong* so that there is a bijection between S and *Wrong*. There is an edge between $w \in \mathit{Wrong}$ and $l \in \mathit{Locations}$ if and only if the element $s \in S$ that corresponds to w is in the subset $\sigma \in C$ that corresponds to l. It is now easy to see that the question for MINIMUM COVER has the answer 'yes' if and only if the question for our problem has the answer 'yes'.

Ideally, the computation of the error locality measure should take only a small fraction of the time required to compute $\{f(I_1), \ldots, f(I\#I)\}$. Since f is often reasonably efficient, e.g. linear in the size of its input, we want to compute error locality in linear time. Most likely, there is no worst-case polynomial time algorithm for computing error locality since this problem is NP-hard. The program evaluation functions employ many measures other than the error locality measure, which only occasionally is needed in order to differentiate programs. Therefore, it is not necessary to choose an algorithm that always gives the "best" error locality measure.

All the errors in a synthesized program are often located in a single *RHS* that does not contain any other *RHS*. For such a program, the best error locality measure is obtained by replacing this single *RHS* with a `?`-constant. Therefore, ADATE uses an algorithm that approximates N'_c by

$$N_c - |\{c : c \in \mathit{Correct} \wedge \{c, l\} \in E\}|,$$

where l is chosen to a member of *Locations* such that the approximation is maximized and such that l is connected to all nodes in *Wrong*. In the `sort` example, we obviously have $l = \mathit{RHS}_5$. This algorithm is very efficient and gives sufficiently good approximations sufficiently often for all inferences that have been run.

i	Value returned by pe_i
1	$-N_c$ :: `Grades @` N_w :: $-N_c'$ :: S :: *Lineage* `@ [`T`]`
2	$-N_c$ :: `Grades @` N_w :: $-N_c'$ :: T :: *Lineage* `@ [`S`]`
3	N_w :: $-N_c$ :: `Grades @` $-N_c'$ :: S :: *Lineage* `@ [`T`]`

Table 3.1: The definitions of pe_1, pe_2 and pe_3.

3.5.4 Lineage

The lineage measure for a program P considers the parent, say $\overline{P}$, of P. Intuitively, a program with a good parent is preferable to a program with a poor parent. Let

$\overline{N_c}$ = The number of correct outputs produced by $\overline{P}$.

$\overline{N_w}$ = The number of wrong outputs produced by $\overline{P}$.

$\overline{\texttt{Grades}}$ = The grades produced by the output evaluation `oe` when applied to the outputs produced by $\overline{P}$.

The lineage measure *Lineage* is simply defined as

$$-\overline{N_c}\texttt{::}\overline{\texttt{Grades}}\ \texttt{@ [}\overline{N_w}\texttt{]}.$$

A typical inference proceeds by adding one `case`-expression at a time, which means that $cc(P) = 1 + cc(\overline{P})$, where the function cc counts the number of `case`-expressions. If an identity transformation is employed to produce P from $\overline{P}$, we will get $P = \overline{P}$, $N_c = \overline{N_c}$, $N_w = \overline{N_w}$ and `Grades` $= \overline{\texttt{Grades}}$. Therefore, an identity transformation (or one that makes a small trivial change) will make the lineage measure meaningless. This is avoided by choosing $\overline{N_c}$, $\overline{N_w}$ and $\overline{\texttt{Grades}}$ to the N_c, N_w and `Grades` values of the initial program if $cc(\overline{P}) \geq cc(P)$. Thus, the lineage measure gives preference to "genealogies" with strictly increasing cc values.

3.5.5 The Definitions of pe_1, pe_2 and pe_3

Let

S = The syntactic complexity.
T = The total call count.

The three program evaluation functions are defined in table 3.1. 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.

Chapter 4

The Atomic Transformations

We will first explain the concepts atomic transformations, compound transformations and expression synthesis.

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.

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. The most frequently used atomic transformation, replacement, employs expression synthesis. Since the requirements for expression synthesis are determined by this and other atomic transformations, we will wait with a more detailed discussion of expression synthesis until all atomic transformations have been presented. Expression synthesis is so important and complicated that the entire Chapter 5 is dedicated to it. Chapter 6 explains how to build compound transformations from atomic transformations.

Section 4.1 gives schemas for each of the atomic transformations which are

R. Replacement.

REQ. Replacement that does not make the program "worse".

ABSTR. Abstraction.

CASE-DIST. `case`-distribution.

EMB. Embedding.

Since REQ is a special case of R, it may seem to be superfluous. However, REQ transformations are so common that an enormous reduction in synthesis time usually is achieved by identifying them as special cases. The most difficult atomic transformation to design is EMB. Our choices of schemas and algorithms for EMB are open in the respect that they may be substantially changed in the future.

Section 4.2 deals with algorithms that perform atomic transformations. This section is more mathematical and technical than Section 4.1.

4.1 Atomic Transformation Schemas

The schemas are presented in a form that relies on higher order matching, which is a special case of higher order unification. The difference between matching and unification is as follows. When unifying two terms T_1 and T_2, both T_1 and T_2 may contain variables. When matching T_1 against T_2, only T_2 may contain variables.

Example. Assume

$$T_1 = \texttt{a(b(g(d),e), c)}$$

and

$$T_2 = H(E_1, E_2).$$

Matching T_1 against T_2 yields 12 unifiers if we assume that each argument of H must occur in the λ-body of H. If the order of the arguments of H does not matter, there are only 6 non-equivalent unifiers, namely

1. $\{\ H = \lambda(X,Y).\texttt{a}(X,Y),\ E_1 = \texttt{b(g(d),e)},\ E_2 = \texttt{c}\ \}$
2. $\{\ H = \lambda(X,Y).\texttt{a(b}(X,Y),\texttt{c)},\ E_1 = \texttt{g(d)},\ E_2 = \texttt{e}\ \}$
3. $\{\ H = \lambda(X,Y).\texttt{a(b}(X,\texttt{e}),Y)\ E_1 = \texttt{g(d)},\ E_2 = \texttt{c}\ \}$
4. $\{\ H = \lambda(X,Y).\texttt{a(b(g}(X),Y),\texttt{c)}\ E_1 = \texttt{d},\ E_2 = \texttt{e}\ \}$
5. $\{\ H = \lambda(X,Y).\texttt{a(b(g}(X),\texttt{e}),Y)\ E_1 = \texttt{d},\ E_2 = \texttt{c}\ \}$
6. $\{\ H = \lambda(X,Y).\texttt{a(b(g(d)},X),Y),\ E_1 = \texttt{e},\ E_2 = \texttt{c}\ \}$

A schema has the form

$$LHS \longrightarrow RHS,$$

where a subexpression of the program to be transformed is matched against LHS.

4.1.1 Replacement

Replacement is the only transformation that may change the semantics of a program. The general replacement schema is

$$H(E_1, E_2, \ldots, E_n) \longrightarrow G(E_1, E_2, \ldots, E_n),$$

where G is an expression that is synthesized as a part of a replacement transformation. The special case $n = 0$ simply means that an entire subexpression of the program is replaced with a newly synthesized expression. The special case $n = 1$ and $H = \lambda X.X$ may be viewed as an insertion of a newly synthesized expression. These two special cases are the most common forms of replacement. Here is an example that illustrates the special case $n = 0$.

Example. 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]`

$I_5 =$ `[10,20,50,40]`

In one out of many possible inferences of `sort`, each compound transformation except the last consists of a single replacement with $n = 0$. For each compound transformation, we will give the position Pos of H, the synthesized expression G, the resulting program and its N_c and N_w values. Program number 1 is the initial program.

1. `fun sort Xs = ?`

   $N_c = 0$ $N_w = 0$

2. $Pos =$ `[]`

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

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

   $N_c = 1$ $N_w = 0$

3. $Pos =$ `[2]`

   $G =$ `case Xs1 of nil => Xs | X2::Xs2 => ?`

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

$N_c = 2$ $N_w = 0$

4. $Pos =$ `[2,2]`

   $G =$ `case X2<X1 of true => ?  | false => Xs`

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

   $N_c = 3$ $N_w = 1$

5. $Pos =$ `[2,0]`

   $G =$ `sort Xs1`

```
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
```

   $N_c = 3$ $N_w = 0$

   The final compound transformation is shown in Subsection 4.1.2. □

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 is denoted by REQ whereas an ordinary replacement is denoted by R. If a compound transformation contains several replacements, ADATE usually requires that one or more of the replacements are REQ's. REQ's are found by trying R's and selecting the ones that do not increase the pe_{REQ} value. Normally, only a small fraction of the R's meet this requirement. The REQ's are sorted according to the pe_{REQ} value to give preference to the best REQ's.

4.1.2 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`,...,`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.

Example. The last compound transformation in the inference of `sort` has the form ABSTR REQ REQ R. 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
```

Thus, the program produced by the ABSTR is

```
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 second occurrence of `Xs`. The second REQ replaces the third 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
```

□

4.1.3 case-distribution

This transformation is based on the following schema.

H`(` `case` E `of` $Match_1$ `=>` E_1 `|` $Match_2$ `=>` E_2 `|` ... `|` $Match_n$ `=>` E_n `)` $\longleftrightarrow$
`case` E `of` $Match_1$ `=>` $H(E_1)$ `|` $Match_2$ `=>` $H(E_2)$ `|` ... `|` $Match_n$ `=>` $H(E_n)$.

Note that the schema may be used both left-to-right and right-to-left. If it is used left to right and some E_i is `?`, an expression H(`?`) is produced. Such an expression is immediately replaced by `?`.

Example. Consider the inference of a function `bst_del` that deletes an element from a binary search tree (BST). Binary trees are represented with the the following data type

```
datatype 'a bin_tree =
  bt_nil | bt_cons of 'a  * 'a bin_tree * 'a bin_tree
```

We will exemplify `case`-distribution using the final compound transformation in one of many possible inferences of a correct `bst_del` program. This compound transformation, which has the form CASE-DIST ABSTR REQ R, starts with the following program.

```
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 =>
  case LeXs of
    bt_nil => RiXs
  | bt_cons(RoLeXs,LeLeXs,RiLeXs) =>
  case RiXs of
    bt_nil => LeXs
  | bt_cons(RoRiXs,LeRiXs,RiRiXs) =>
      bt_cons(
        case LeRiXs of
          bt_nil => RoRiXs
        | bt_cons(RoLeRiXs,LeLeRiXs,RiLeRiXs) => ?,
        LeXs,
        RiRiXs )
```

The CASE-DIST moves the last occurrence of `case` outwards, which yields the program

```
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 =>
  case LeXs of
    bt_nil => RiXs
  | bt_cons(RoLeXs,LeLeXs,RiLeXs) =>
  case RiXs of
```

```
    bt_nil => LeXs
  | bt_cons(RoRiXs,LeRiXs,RiRiXs) =>
  case LeRiXs of
    bt_nil => bt_cons(RoRiXs,LeXs,RiRiXs)
  | bt_cons(RoLeRiXs,LeLeRiXs,RiLeRiXs) => ?
```

The ABSTR has $n = 2$ and $H(E_1, E_2)$ equal to the last `case`-expression. The following program is produced by the ABSTR.

```
fun bst_del(I as (X,Xs)) =
  ...
  case RiXs of
    bt_nil => LeXs
  | bt_cons(RoRiXs,LeRiXs,RiRiXs) =>
  let fun g(Ys,Y) =
    case Ys of
      bt_nil => bt_cons(Y,LeXs,RiRiXs)
    | bt_cons(RoLeRiXs,LeLeRiXs,RiLeRiXs) => ?
  in
    g(LeRiXs,RoRiXs)
  end
```

The REQ changes the last occurrence of `RiRiXs` to `bst_del(Y,RiXs)`, which gives the program

```
fun bst_del(I as (X,Xs)) =
  ...
  case RiXs of
    bt_nil => LeXs
  | bt_cons(RoRiXs,LeRiXs,RiRiXs) =>
  let fun g(Ys,Y) =
    case Ys of
      bt_nil => bt_cons(Y,LeXs,bst_del(Y,RiXs))
    | bt_cons(RoLeRiXs,LeLeRiXs,RiLeRiXs) => ?
  in
    g(LeRiXs,RoRiXs)
  end
```

Finally, the R produces a correct `bst_del` program by replacing the `?` with

`g(LeLeRiXs,RoLeRiXs)`.

Note that the retention of "old" variable names in $H(E_1, E_2)$ means that the variables that designate the root and the left and the right subtrees of `Ys` have misleading names i.e., `RoLeRiXs`, `LeLeRiXs` and `RiLeRiXs` instead of `RoYs`, `LeYs` and `RiYs`. □

4.1.4 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`,...,`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. However, this design is not as complete and definitive as the designs of the other atomic transformations.

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` $T_{1,1} * T_{1,2} * \ldots T_{1,\#T_1}$ |
C_2 `of` $T_{2,1} * T_{2,2} * \ldots T_{2,\#T_2}$ |
$\vdots$
$C_{\#C}$ `of` $T_{\#C,1} * T_{\#C,2} * \ldots T_{\#C,\#T_{\#C}}$

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. 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 $T_1 * \ldots * T_n$ can be embedded in two ways.

1. The new type is $T_1 * \ldots * T_n * 'a$, 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. For example, using the type constructor `bin_tree` the tuple type `int list * bool` may 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 $T_1 * \ldots * T_n$ 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. $T_1 * \ldots * T_n$ to $T_1 * \ldots * T_n *' a$. Each call of the form `g`$(E_1, \ldots, E_n)$ is changed to `g`$(E_1, \ldots, E_n,$ `?_emb`$)$
2. $T_1 * \ldots * T_i * \ldots * T_n$ to $T_1 * \ldots * T_i$ `list` $* \ldots * T_n$.
   (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 $T_1 * \ldots * T_n$ 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. $T_1 * \ldots * T_n$ to $T_1 * \ldots * T_n *' a$.
   (a) Each call `g`$(\ldots)$ is changed to

   `case g`$(\ldots)$ `of` X `as` $(X_1, \ldots, X_n, X_{n+1})$ `=>` $(X_1, \ldots, X_n)$.

   (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. $T_1 * \ldots * T_i * \ldots * T_n$ to $T_1 * \ldots * T_i$ `list` $* \ldots * T_n$.
   (a) If $n = 1$, each call `g`$(\ldots)$ is changed to

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

If $n \geq 2$, each call `g(...)` is changed to

`case g(...) of` X `as` $(X_1, \ldots, X_i, \ldots, X_n)$ `=>`
`case` X_i `of nil => ?_emb | Y::Ys =>` $(X_1, \ldots, X_{i-1}, Y, X_{i+1}, \ldots, X_n)$.

(b) The *RHS*, which in this case is assumed to have the form

$$(E_1, \ldots, E_i, \ldots, E_n),$$

is changed to

$$(E_1, \ldots, E_i\texttt{::?\_emb}, \ldots, E_n).$$

If E_i is a `case`-expression, `case`-distribution is invoked to move the `?_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. 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. □

4.2 Atomic Transformation Algorithms

The transformation algorithms operate with two concepts, *work* and combinatorial *cost*. The *work* is the approximate number of programs to be produced. The *cost* is a measure of transformation complexity.

The cost of a transformation is the reciprocal probability of the transformation as determined by some prior probability distribution. Intuitively, this prior distribution specifies the probability that a transformation produces a "good" program. Assume that the program to be transformed is a declaration D and that the transformation of D produces declarations $D_1, D_2, \ldots, D_n$. The cost-probability relationship is

$$Pr(D_i) \cdot cost(D_i) = 1,$$

where each D_i has probability $Pr(D_i)$ and cost $cost(D_i)$.

A proper probability distribution on the set { D_1, D_2, ..., D_n } must satisfy $0 \leq Pr(D_i) \leq 1$ and $\sum_{i=1}^{n} Pr(D_i) = 1$. We assume that each D_i has positive probability, which means that each cost is well-defined. The requirement $\sum_{i=1}^{n} 1/cost(D_i) = 1$ is not only motivated by the desire to have a proper probability distribution, but also by the need to ensure that the costs of different types of atomic transformations can be directly compared. For example, an ABSTR transformation with a cost of 200 should be as "complex" as a REQ transformation with a cost of 200. We assume that probability is inversely proportional to transformation complexity. When an algorithm needs to make m sequential choices, i.e., choose a path of length m in a decision tree, we assume that the total probability is the product of the probabilities of each choice.

Each type of transformation T in { R, REQ, ABSTR, CASE_DIST, EMB } could be implemented in a purely functional language using a function that returns a list of produced programs. Since very many programs may be produced, such lists may require unreasonably much space. Therefore, we have chosen an implementation style that is not purely functional but still rather elegant. Each type of transformation T is implemented by a function with a declaration of the basic form

```
fun T_trfs( D : dec, Cost_limit :  real,
            emit :  dec * atomic_trf_record list * real -> unit
            ) :  unit = ...
```

```
datatype ty_exp =
  ty_var_exp of ty_var
| ty_con_exp of symbol * ty_exp list

type ty_schema = { schematic_vars : ty_var list, ty_exp :  ty_exp }

type ty_env = (symbol * ty_schema) list

type exp = (ty_exp option,ty_schema option)e
type dec = (ty_exp option,ty_schema option)d
```

Figure 4.1: The ML representation of typed expressions and declarations.

The declaration `D` is to be transformed using a cost that does not exceed `Cost_limit`. The type `dec` is the type of typed declarations. Similarly, `exp` is the type of typed expressions. These types are specializations of the polymorphic types `('a,'b)d` and `('a,'b)e` that were given in Figure 2.2. Following [Peyton Jones 87], the type of the function introduced by a `fun`-declaration is described using a type schema that contains so-called schematic variables. Please see [Peyton Jones 87] for more details concerning the representation and inference of types. Figure 4.1 shows the ML declarations of `dec` and `exp`. It is necessary to use the types `dec` and `exp` instead of the types `('a,'b)d` and `('a,'b)e` since the atomic transformation algorithms need to know the types of expressions and functions.

For each produced declaration D_i, T`_trfs` makes the call `emit(`D_i`,Records,Cost)`. `Records` is a list that describes the atomic transformation that transformed `D` to D_i. `Cost` is the cost of this transformation. Note that the `emit` function is called only because of its side-effects, which are unacceptable in purely functional programming.

An implementation of T`_trfs` should normalize costs, i.e., ensure that

$$\sum_{i=1}^{n}(1/cost(D_i)) = 1.$$

The ADATE implementation uses a fixed, unnormalized cost measure, say *cost'*. The normalized cost is determined by finding a normalizing factor K and choosing $cost(D_i) = K \cdot cost'(D_i)$ for each i in $1, \ldots, n$. For some types of atomic transformations, the normalization problem is non-trivial. Here is a simple example that exhibits the difficulty of computing the normalizing factor K.

Example. Assume that $cost'(D_i) = 10i$, which gives $cost(D_i) = 10Ki$. For a given `Cost_limit`, we want to find n and K such that `Cost_limit` $= 10Kn$ and $\sum_{i=1}^{n}(1/(10Ki)) = 1$ i.e., $\sum_{i=1}^{n}(1/i) = 10K$. A simple integration approximation

gives $\ln n = 10K$. Thus, we need to solve the equation $n \ln n =$`Cost_limit` for n, which is difficult to do analytically. □.

In general, K cannot be computed analytically. Therefore, we sometimes employ a simple numerical equation solver in order to find K. The difficulty of computing K depends on the type of atomic transformation that is under consideration. Normalization is further discussed in conjunction with each atomic transformation algorithm that is presented below.

4.2.1 The R Transformation Algorithm

This algorithm needs to choose the number n of reused expressions $E_1, \ldots, E_n$, the synthesized expression G, the position of H and the position of each E_i. ADATE uses only the following three kinds of replacements.

1. $n = 0$.
2. $n = 1$ and $H = \lambda X.X$.
3. $n = 1$ and $H \neq \lambda X.X$.

Therefore, the R transformation algorithm needs to make the following choices.

1. The kind of replacement, i.e., 1, 2 or 3.
2. The position of H. For replacement of kind 3, it is also necessary to choose the position of E_1.
3. The synthesized expression G.

We will now discuss the (unnormalized) costs associated with each of these three choices.

1. We have chosen costs 1, 1 and 2 for replacements of kinds 1, 2 and 3 respectively. These cost choices have been found empirically, but we have too limited transformation statistics to claim that they are not "ad hoc". Unfortunately, this holds for most prior choices of costs in ADATE.
2. Let `Top_pos` be the position of H, which is of course the same as the position of $H(E_1, \ldots, E_n)$. The set of all possible values of `Top_pos` is partitioned into the following three classes.
   (a) $H(E_1, \ldots, E_n)$ is a `?`.
   (b) $H(E_1, \ldots, E_n)$ is the right hand side of a `case`-rule.
   (c) $H(E_1, \ldots, E_n)$ is some other expression.

A `Top_pos` in class (a) is more likely to lead to a "good" replacement than a `Top_pos` in class (b), which in turn is more likely than a `Top_pos` in class (c). Therefore, we have chosen the costs 1, 5 and 25 for classes (a), (b) and (c) respectively.

For replacements with $n = 1$, i.e., replacements of kinds 2 or 3, we do not allow E_1 to be a leaf in the expression tree. The reason is that a leaf is considered too small to be worth reusing. Let `Bottom_pos` be the position of E_1. For replacements of kind 2, we obviously have `Bottom_pos` = `Top_pos`. For replacements of kind 3, we require that `Top_pos` is a proper prefix of `Bottom_pos`, i.e., a prefix such that `Bottom_pos` $\neq$ `Top_pos`. Each `Bottom_pos` is assigned the same cost.

3. Synthesized expressions are numbered in the order in which they are synthesized, which is in order of increasing size. The cost of expression number i is i.

Figure 4.2 shows the ML definition of a function `R_trfs` that implements the `R` transformation. We will now explain this implementation. In addition to the parameters `D`, `Cost_limit` and `emit`, which were discussed above, `R_trfs` has the parameters `poses_ok` and `Min_once`.

The function parameter `poses_ok` is used to impose additional constraints on positions. For replacements with $n = 0$, we must have `poses_ok( Top_pos, nil )`. If $n = 1$, we require `poses_ok( Top_pos, [Bottom_pos] )`.

The parameter `Min_once` specifies symbols that are required to occur at least once in the synthesized expression G. For example, when having done an ABSTR transformation, we require that the introduced `let`-function is used at least once in some G. It is, after all, rather meaningless to have a `let`-function that is used only once. `Min_once` is a list of lists of symbols

$$\texttt{[[}S_{1,1}, S_{1,2}, \ldots\texttt{], [}S_{2,1}, S_{2,2}, \ldots\texttt{], } \ldots\texttt{, [}S_{\#S,1}, S_{\#S,2}, \ldots\texttt{]]}$$

For each i in $1, 2, \ldots, \#S$, at least one $S_{i,j}$ in `Min_once` must occur in G.

Before starting to find R transformations of `D`, `D` is executed for all sample inputs $\{I_1, \ldots, I_{\#I}\}$ that are given in the specification. A subexpression of `D` is said to be *activated* if and only if it was evaluated during this execution. An R transformation, that replaces a subexpression that is not activated, is pointless. The call `add_not_activated_exps_dec D` executes `D` for all sample inputs and replaces each non-activated subexpression with a special *Not_activated* constant.

We first normalize the costs of positions. The normalized cost of an R transformation is obtained by multiplying the normalized cost of the positions with the normalized cost of G. The local variable `Interval_width` is the normalization factor for position costs. This factor is computed by simulating a "dry" run of the R transformation algorithm before doing the "real" run. The dry run does not synthesize expressions. Its only purpose is to find the normalization factor.

```
fun R_trfs( D : dec, Cost_limit : real,
      poses_ok : pos*pos list -> bool, Min_once : symbol list list,
      emit : dec*atomic_trf_record list*real->unit ) : unit =
  let
    val D as {func,pat,exp,dec_info} = add_not_activated_exps_dec D
    val Interval_width = ref 0.0
    fun run(Dry_run:bool) =
  let
    fun replace'(Top_pos,Bottom_poses,Cost) =
      if Dry_run then
        Interval_width := !Interval_width + 1.0/Cost
      else
      let
        val Cost = Cost * !Interval_width
        fun emit'(New_D,G_cost,Not_activated_syms) =
          emit( New_D, R{top_pos=Top_pos,bottom_poses=Bottom_poses,
            not_activated_syms=Not_activated_syms}::nil,
            Cost*G_cost )
      in
        Replace.replace( D, Top_pos, Bottom_poses,
          Cost_limit/Cost, Min_once, emit')
      end
    val All_poses = all_poses_in_preorder exp
    val R0_top_poses =
      filter( fn Top_pos => poses_ok(Top_pos,nil), All_poses )
  in
    (* Replacement of kind 0. *)
    map( fn Top_pos =>
      replace'(Top_pos,nil,
        real(length R0_top_poses)*top_pos_class_cost(exp,Top_pos) ),
      R0_top_poses );
(* Code for replacements of kinds 1 and 2 has been omitted here. *)
  end
  in
    run true;
    run false; ()
  end (* fun R_trfs *)
```

Figure 4.2: The ML implementation of R transformations.

A run calls the function `replace'` with all allowed `Top_pos` and `Bottom_poses` values. If a real run is in progress, the function `replace'` in turn calls the auxiliary function `Replace.replace`, which synthesizes G expressions and inserts them into `D`. The implementation of this auxiliary function is discussed together with expression synthesis in Chapter 5.

The local variable `All_poses` is the list of all positions in `D`, excluding positions of *Not_activated* constants.

Figure 4.2 contains code for replacement of kind 0 but not for kinds 1 and 2. The code for kind 1 is straightforward. The code for kind 2 is complicated by scope checking, which is necessary since symbols that occur in E_1 may be defined in H. Since H is removed, there may be "illegal" occurrences in E_1 of symbols defined in H. The scope checking code is straightforward but long-winded. Therefore, we have omitted it in Figure 4.2.

4.2.2 The REQ Transformation Algorithm

The most common form of REQ is a semantics preserving replacement of a small subexpression with a small, synthesized expression. All replacements that are employed when trying to find REQs have $n = 0$, i.e., do not reuse any E_i's. This is motivated by the empirical observation that other kinds of replacements only rarely are useful REQs.

REQs are implemented by a function `N_REQ_trfs` which has parameters as follows.

```
fun N_REQ_trfs( No_of_REQs : int, D : dec, Cost_limit : real,
      REQ_cost_limit : real, top_pos_ok : pos->bool,
      emit : dec*atomic_trf_record list*real->unit ) : unit = ...
```

The parameters `D`, `Cost_limit` and `emit` have already been described.

Sometimes, ADATE needs to apply a sequence of REQs to a program. The parameter `No_of_REQs` specifies how many REQs that are to be applied in sequence. For example, if `No_of_REQs` $= 2$, the initial declaration `D` is transformed to another declaration, say `D'`, by the first REQ. The second REQ then transforms `D'` to yet another declaration which is emitted.

The parameter `REQ_cost_limit` says how complex replacements that are to be employed in order to find REQs. Usually, `REQ_cost_limit` is many times greater than `Cost_limit`.

The function parameter `top_pos_ok` is such that `top_pos_ok Top_pos` holds if and only if the expression at position `Top_pos` should be allowed to be replaced when trying to find REQs.

We will now discuss the implementation of `N_REQ_trfs`. Assume that `Enumeration` is a list of (position, expression) pairs `[` (Pos_1,E_1)`, (`Pos_2`,`E_2`), ... ]` such that the replacement of the expression at position Pos_i with E_i is a REQ. Also assume that `Enumeration` is sorted in order of increasing pe_{REQ} values, i.e., with

the best REQs first. There are two main problems to solve, namely

1. How to find an appropriate value of `Enumeration`.
2. How to use `Enumeration` to compute sequences of REQs.

We will first attack problem 1 and then problem 2.

Computing `Enumeration`

Problem 1 is solved using the following function.

```
fun find_REQs( D : dec, REQ_cost_limit : real,
      top_pos_ok : pos -> bool,
      emit : pos*exp*real list->unit ) : unit =...
```

This function makes the call

`emit(` Pos_i`,` E_i`,` pe_{REQ}`-`$value_i$ `)`

for all REQs found using a cost that does exceed `REQ_cost_limit`. The emitted triples are inserted into a priority queue that is sorted according to the pe_{REQ}-value with the size of E_i appended. The size is appended in order to give preference to small REQs if the pe_{REQ}-values are equal. Since the queue is implemented as a heap, each insertion takes time $O(\log N)$, where N is the heap cardinality. ADATE has a predetermined, fixed upper limit on heap cardinality. This limit should be set as high as the amount of memory in the computer allows. An upper limit of 1000 is actually more than enough for all examples that have been run so far. When the upper limit is exceeded, the worst REQ is deleted from the heap, which takes time $O(\log N)$. When the heap has been constructed, it is very simple to convert it to `Enumeration` in time $O(N \log N)$.

The implementation of `find_REQs` is somewhat more complicated. One complication is that different REQs may yield identical programs. Naturally, only one of these identical programs should be emitted. Here is an example with REQs that produce identical programs.

Example. Assume that the following program is to be transformed.

```
fun f X = g1(g2(g3(X,a)))
```

Also assume that the (position,expression) pair `([0,0,1],b)` describes a REQ, i.e., that replacement of `a` with `b` does not increase pe_{REQ}. This REQ yields the program

```
fun f X = g1(g2(g3(X,b)))
```

There are three other REQs that produce the same program, namely

1. `([0,0],g3(x,b))`,

2. `([0],g2(g3(x,b)))` and

3. `([],g1(g2(g3(x,b))))`.

Each of these three REQs is more complex than `([0,0,1],b)`. □

ADATE finds the least complex REQ first in a class of REQs that produce identical programs by trying positions in postorder.

A simple way to avoid the emission of identical programs is to store each produced program in a hash table and immediately discard a new program if it occurs in the table. This method requires too much space since each program may require 10 or more times as much space as a (position,expression) pair that specifies a REQ. Instead of storing programs in the hash table, we store fingerprints of the right hand sides of programs. A fingerprint is an integer computed by a hash function `exp_hash` of type `('a,'b)e -> int`.

Since ADATE uses this technique for other purposes as well, it is important enough to be worth analyzing. Two expressions E_1 and E_2 are assumed to be identical if and only if `exp_hash`(E_1) =`exp_hash`(E_2). This assumption is invalid if and only if `exp_hash`(E_1) =`exp_hash`(E_2) and $E_1 \neq E_2$. We will now show that the probability that the assumption is invalid can be made negligibly small. Let n be the cardinality of the range of `exp_hash`. Using b bits to represent fingerprints gives $n = 2^b$. Assume that `exp_hash` has such a good spread that every fingerprint is equally probable. It is then reasonable to assume that two different expressions have the same fingerprint with probability $1/n$. For example, choosing b to 64 bits means that this probability is 2^{-64} which is negligibly small.

Another issue in the implementation of `find_REQ` is how to distribute `REQ_cost_limit` on the allowed positions i.e., the positions specified by the `top_pos_ok` predicate. We have chosen to use a five times higher cost limit for positions of `?`-constants than for other positions. Assume that n is the total number of allowed positions and that $n_?$ of these positions specify `?`-constants. The requirement that costs are normalized then gives a cost limit of

$$\frac{5\texttt{REQ\_cost\_limit}}{n + 4n_?}$$

for positions of `?`-constants and a cost limit of

$$\frac{\texttt{REQ\_cost\_limit}}{n + 4n_?}$$

for other positions.

Using `Enumeration` to Compute Sequences of REQs

We are now ready to tackle problem 2, i.e., the construction of REQ sequences and their normalized costs. Let

(Pos_1,E_1), (Pos_2,E_2), ..., $(Pos_{\texttt{No\_of\_REQs}},E_{\texttt{No\_of\_REQs}})$

be a REQ sequence i.e., a sequence of (position, expression) pairs such that the pe_{REQ} value does not increase when all expressions at positions Pos_1, ..., $Pos_{\texttt{No\_of\_REQs}}$ have been replaced with the corresponding E_i's. Note that one or more (Pos_i,E_i) replacements might increase pe_{REQ} even though all `No_of_REQs` replacements taken together do not increase pe_{REQ}. As illustrated by the following example, the assumption that each replacement in a REQ sequence is a REQ may lead to an enormous reduction of the number of combinations that need to be checked for REQ-hood.

Example. For illustration purposes only, assume that N_R replacements are to be tried for each element in a REQ sequence of length `No_of_REQs`. If we had to try all combinations of N_R replacements for each element, there would be $N_R^{\texttt{No\_of\_REQs}}$ replacement sequences that would need to be checked for REQ-hood. Sometimes, about one of every one hundred replacements is a REQ, which means that the probability Pr_{REQ} that a replacement is a REQ is about 10^{-2}. The assumption that each replacement in a REQ sequence can be required to be a REQ reduces the total number of combinations from $N_R^{\texttt{No\_of\_REQs}}$ to $(N_R Pr_{\text{REQ}})^{\texttt{No\_of\_REQs}}$. Note that we usually have $\texttt{No\_of\_REQs} \leq 2$ and that the probability Pr_{REQ} depends on the position, the program and the specification. □

We assume that each (Pos_i,E_i) is a REQ. This means that a sequence

(Pos_1,E_1), (Pos_2,E_2),..., $(Pos_{\texttt{No\_of\_REQs}},E_{\texttt{No\_of\_REQs}})$

can be constructed by choosing each (Pos_i,E_i) from the list `Enumeration`.

There are some obvious constraints on positions. If Pos_j is the next position to be added to the sequence, the sequence must not contain any position Pos_i such that

1. Pos_i is a prefix of Pos_j or
2. Pos_j comes before Pos_i in preorder, which will be written $Pos_j < Pos_i$.

The second condition only allows one permutation of a sequence.

The last topic in the implementation of `N_REQ_trfs` is the computation of the normalized cost of a sequence. Let $x(Pos_i, E_i)$ be the order number of (Pos_i, E_i) in `Enumeration`. The unnormalized cost of the sequence

(Pos_1,E_1), (Pos_2,E_2),..., $(Pos_{\texttt{No\_of\_REQs}},E_{\texttt{No\_of\_REQs}})$

is chosen to

$$\prod_{i=1}^{\texttt{No\_of\_REQs}} (x(Pos_i, E_i) + A),$$

where A is a constant factor that is chosen to 3 in the current implementation. The purpose of A is, for example, to say that a REQ with order number 1 is not

5 times as cheap as a REQ with order number 5 but $(5+3)/(1+3)$ =2 times as cheap.

Normalization is somewhat complicated. We use the following abbreviations.

$n =$ `No_of_REQs` = The number of REQs to be applied in sequence.

$N =$ The cardinality of `Enumeration`.

Using simplified constraints on positions, we will first quickly compute two approximations of the normalizing factor K and then choose the best of these as the initial value of K in a more time consuming and precise iterative search that employs the two position constraints given above.

The Two Approximations of K. The two approximations are computed as follows. Let $(x_1, \ldots, x_n)$ be the order numbers of the REQs in a sequence of length n. The position constraints are approximated by requiring $x_i < x_{i+1}$ for $i = 1, \ldots, n-1$. We want to find a normalizing factor K such that

$$\sum_{(x_1,\ldots,x_n) \in D_n(\texttt{Cost\_limit}/K)} \frac{1}{(x_1 + A) \cdot \ldots \cdot (x_n + A)} = K,$$

where the summation domain is

$$D_n(c) = \{(x_1, \ldots, x_n) \mid 1 \leq x_i \leq N, x_i < x_{i+1}, (x_1 + A) \cdot \ldots \cdot (x_n + A) \leq c\}.$$

The sum is approximated using the n-dimensional integral

$$I_n(c) = \int \ldots \int_{D_n(c)} \frac{1}{(x_1 + A) \cdot \ldots \cdot (x_n + A)} dx_1 \ldots dx_n.$$

We will now relax constraints in $D_n(c)$ in order to simplify the integration. Let $\pi_1 \ldots \pi_n$ be a permutation of $x_1 \ldots x_n$. Consider the integration domain obtained by requiring $\pi_1 < \pi_2 < \ldots < \pi_n$ instead of $x_1 < x_2 < \ldots < x_n$. Due to symmetry, the integral over this domain has the value $I_n(c)$. Since there are $n!$ different permutations of $x_1 \ldots x_n$, there are $n!$ such domains that are disjoint. If we remove the constraints $x_1 < x_2 < \ldots < x_n$ from $D_n(c)$, we multiply the integral by $n!$. Therefore, it is sufficient to consider integration over the domain

$$\{(x_1, \ldots, x_n) \mid 1 \leq x_i \leq N, (x_1 + A) \cdot \ldots \cdot (x_n + A) \leq c\}.$$

It is straightforward to integrate over this domain for $n = 1$ and $n = 2$. For $n \geq 3$, we get problems with overlapping sub-domains under the hyperbolic surface

$$(x_1 + A) \cdot \ldots \cdot (x_n + A) = c,$$

which seems to require the use of a so-called "inclusion-exclusion" formula that is quite complicated for large n. Since we only want to quickly compute an initial approximation of K, we choose to further simplify the domain. The following two simplifications are used

$$D'_n(c) = \{(x_1, \ldots, x_n) \mid 1 \leq x_i \leq N\}.$$

$$D''_n(c) = \{(x_1, \ldots, x_n) \mid 1 \leq x_i, (x_1 + A) \cdot \ldots \cdot (x_n + A) \leq c\}.$$

Let I' and I'' be the integrals over the domains D' and D'' respectively. Both $I'_n(c)$ and $I''_n(c)$ are overestimates of $I_n(c)$, which means that the corresponding values of K, say K' and K'', also are overestimates. We choose to initially approximate K with the smallest of these factors i.e.,

$$K_{initial} = \min(K', K'').$$

The next issue is the computation of I' and I''. Using repeated one-dimensional integration we get

$$I'_n(c) = \int_1^N \frac{1}{x_1 + A} \int_1^N \frac{1}{x_2 + A} \int_1^N \cdots \int_1^N \frac{1}{x_n + A} dx_n \ldots dx_2 dx_1,$$

which equals

$$(\ln \frac{N + A}{1 + A})^n.$$

The computation of I'' is more difficult. The definition is

$$I''_n(c) = \int \cdots \int_{D''_n(c)} \frac{1}{(x_1 + A) \cdot \ldots \cdot (x_n + A)} dx_1 \ldots dx_n.$$

Since $1 \leq x_i$ and $(x_1 + A) \cdot \ldots \cdot (x_n + A) \leq c$, we have

$$1 \leq x_n \leq \frac{c}{(1 + A)^{n-1}} - A,$$

where the upper limit is obtained by setting each x_i to 1 for $i = 1, \ldots, n - 1$. For a fixed value of x_n, the upper limit on the other variables is given by

$$(x_1 + A) \cdot \ldots \cdot (x_{n-1} + A) \leq \frac{c}{x_n + A}.$$

This means that $I''_n(c)$ equals

$$\int_1^{\frac{c}{(1+A)^{n-1}} - A} \frac{1}{x_n + A} (\int \cdots \int_{D''_{n-1}(\frac{c}{x_n + A})} \frac{1}{(x_1 + A) \cdot \ldots \cdot (x_{n-1} + A)} dx_1 \ldots dx_{n-1}) dx_n.$$

We obtain the following recurrence relation.

$$I''_n(c) = \int_1^{\frac{c}{(1+A)^{n-1}} - A} \frac{1}{x + A} I''_{n-1}(\frac{c}{x + A}) dx.$$

The base case is

$$I''_1(c) = \int_1^{c-A} \frac{1}{x + A} dx = \ln \frac{c}{1 + A}.$$

We employed this recurrence relation to compute $I_2''(c)$ and $I_3''(c)$, which lead to the induction hypothesis

$$I_n''(c) = \frac{1}{n!}(\ln \frac{c}{(1+A)^n})^n.$$

Assuming that this equality holds for n, we want to prove that it holds for $n+1$. The recurrence relation gives

$$I_{n+1}''(c) = \int_1^{\frac{c}{(1+A)^n}-A} \frac{1}{x+A} I_n''(\frac{c}{x+A})dx.$$

The induction hypothesis yields

$$\begin{aligned} I_{n+1}''(c) &= \textstyle\int_1^{\frac{c}{(1+A)^n}-A} \frac{1}{x+A}\frac{1}{n!}(\ln \frac{c}{(x+A)(1+A)^n})^n dx \\ &= \textstyle\frac{1}{n!}\left[-\frac{1}{n+1}(\ln \frac{c}{(x+A)(1+A)^n})^{n+1}\right]_1^{\frac{c}{(1+A)^n}-A} \\ &= \textstyle\frac{1}{(n+1)!}(-(\ln 1)^{n+1} + (\ln \frac{c}{(1+A)^{n+1}})^{n+1}) \\ &= \textstyle\frac{1}{(n+1)!}(\ln \frac{c}{(1+A)^{n+1}})^{n+1} \text{ Q.E.D.} \end{aligned}$$

In order to find K' and K'', we need to solve the equations

$$\frac{1}{n!}(\ln \frac{N+A}{1+A})^n = K'$$

and

$$\frac{1}{(n!)^2}(\ln \frac{\texttt{Cost\_limit}}{K''(1+A)^n})^n = K''.$$

The first equation is already solved. The second equation can be solved very quickly using Newton-Raphson iteration. In order to do this, we define

$$y(z) = \frac{1}{(n!)^2}(\ln \frac{\texttt{Cost\_limit}}{z(1+A)^n})^n - z,$$

which means that we want to find K'' such that $y(K'') = 0$. Let z_i be the K'' approximation produced after iteration number i. Newton-Raphsons' formula is

$$z_{i+1} = z_i - \frac{y(z_i)}{\frac{dy}{dz}(z_i)}.$$

Computing the derivative and simplifying yields

$$z_{i+1} = z_i + \frac{z_i - \frac{(\ln \frac{\texttt{Cost\_limit}}{(1+A)^n z_i})^n}{(n!)^2}}{-1 - \frac{n(\ln \frac{\texttt{Cost\_limit}}{(1+A)^n z_i})^{n-1}}{z_i (n!)^2}}.$$

To prevent some floating point overflows, we want an initial approximation z_0 that is a lower bound on K''. Since we must have

$$\frac{\texttt{Cost\_limit}}{z(1+A)^n} > 1,$$

we choose

$$z_0 = \frac{\texttt{Cost\_limit}}{(1+\epsilon)(1+A)^n},$$

where ϵ is chosen to 0.1. The search is terminated when

$$|\frac{z_{i+1}}{z_i} - 1| < 10^{-3}.$$

The Exact Search for K. Starting from

$$K_{initial} = \min(K', K''),$$

we now want to do a computation of K that does not use approximations such as the ones that were employed to find $K_{initial}$. Since this exact search may be quite time consuming if we employ a poor initial approximation of K, we do need an initial approximation that is as good as $K_{initial}$.

Figure 4.3 shows the definition of a function `choose_order_nos` which first is employed to compute the sum of cost reciprocals with the parameter `Dry_run` = `true` and then to emit programs resulting from REQ sequences with `Dry_run` = `false`. In the ADATE source code the definition of `choose_order_nos` occurs inside the definition of `N_REQ_trfs`. Therefore, Figure 4.3 contains variables that are parameters to `N_REQ_trfs`, namely `Cost_limit`, `No_of_REQs` and `emit`. There are also some other variables that are global with respect to Figure 4.3, namely `Enumeration` and `pe_REQ_D`. The latter is the pe_{REQ} value of the program `D` that is to be transformed. `No_of_REQs_left` is the number of REQs that remain to be determined, which initially is `No_of_REQs`. `So_far` is the partial REQ sequence that has been chosen so far. `D_so_far` is the result of applying `So_far` to `D`. `Cost_so_far` is the unnormalized cost of `So_far`. `K` is a tentative normalizing factor. The REQ-hood of `So_far` is checked if and only if `Evaluate` = `true`. For a given candidate `K`-value, `choose_order_nos` accumulates cost reciprocals in the reference variable `Interval_width`. In particular, note that the search is cut off as soon as `K * Cost_so_far` exceeds `Cost_limit`.

To simplify the usage of `choose_order_nos`, we define

```
fun choose(K,Evaluate,Dry_run) =
  choose_order_nos(No_of_REQs,nil,D,1.0,K,Evaluate,Dry_run)
```

```
fun choose_order_nos( No_of_REQs_left : int, So_far : (pos*exp) list,
      D_so_far : dec, Cost_so_far : real, K : real, Evaluate : bool,
      Dry_run : bool ) =
  if K*Cost_so_far > Cost_limit then
    ()
  else if No_of_REQs_left = 0 then
    if No_of_REQs>=2 andalso Evaluate andalso
       not(better_or_equal( pe_REQ D_so_far, pe_REQ_D ))
    then
      ()
    else if Dry_run then
      Interval_width := !Interval_width + 1.0/(K*Cost_so_far)
    else
    emit( D_so_far, map(fn(Pos,_) => REQ{top_pos=Pos},So_far),
          K*Cost_so_far * !Interval_width )
  else (
    map( fn(X,(Pos,E)) =>
      if exists( fn(Pos',_) =>
             pos_less(Pos,Pos') orelse is_prefix(Pos',Pos),
           So_far )
      then
        ()
      else
        choose_order_nos( No_of_REQs_left-1, (Pos,E)::So_far,
          if Dry_run andalso not Evaluate then D_so_far else
            pos_replace_dec(D_so_far,Pos,fn _ => E),
          Cost_so_far*(real X + A),K,Evaluate,Dry_run ),
      combine(fromto(1,N),Enumeration) ); () )
```

Figure 4.3: Finding the sum of cost reciprocals for a given `K`.

The sum of the cost reciprocals is computed by the following function.

```
fun sum K = (
  Interval_width := 0.0;
  choose(K,false,true);
  !Interval_width
  )
```

First, `N_REQ_trfs` employs a binary search to find a `K` such that

$$0.9 \leq \texttt{sum K} \leq 1.1.$$

We allow a 10% deviation from 1.0 to reduce the number of calls to `sum` that are needed to find `K`. Note that `K` was found with `Evaluate = false`. Since a REQ sequence may increase `pe_REQ` even though each individual REQ in the sequence does not increase `pe_REQ`, `N_REQ_trfs` executes the following code.

```
Interval_width := 0.0;
choose(K,true,true);
```

This sets `Interval_width` to the fraction of the sequences that do not increase the `pe_REQ` value. This fraction is then used for normalization together with `K` in the final call to `choose`, which is

```
choose(K,true,false)
```

This is the first call that has `Dry_run = false`, which means that the produced declarations are emitted from `N_REQ_trfs`.

4.2.3 The ABSTR Transformation Algorithm

This algorithm needs to choose

1. The arity n of the `let`-function `g` that is to be created.
2. `Top_pos` which is the position of $H(E_1, \ldots, E_n)$.
3. `Bottom_poses` which is the list of the positions of $E_1, \ldots, E_n$.

We will now discuss each of these three choices.

1. The arity n is chosen to 1 or 2. However, an EMB transformation may be applied immediately after an ABSTR transformation in order to increase the arity. Both the choice $n = 1$ and the choice $n = 2$ have the normalized cost 2.

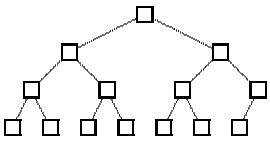


Figure 4.4: An expression tree.

2. `Top_pos` is chosen so that the size of $H(E_1, \ldots, E_n)$ is $2 + 2n$ or greater. The reason is that it, in principle, should be possible to choose the expressions $H, E_1, \ldots, E_n$ so that each expression contains at least two expression tree nodes that also occur in the program to be transformed. We assume that leaf subexpressions of this program are often too small to be worth "reusing". The $2+2n$ minimum size restriction means that there are fewer choices of `Top_pos` values.

   Example. Assume that `Top_pos` is to be chosen in an expression tree that is binary and complete, which means that the tree has 2^d nodes at each depth d in $\{0, 1, \ldots, d_{\max}\}$. Consider $n = 1$ only. Without any restriction, there are $2^{d_{\max}+1} - 1$ choices of `Top_pos`. With the $2 + 2n$ minimum size restriction, there are $2^{d_{\max}-1} - 1$ choices of `Top_pos`, which means that this restriction reduces the number of choices by about four times. □

   Each choice of `Top_pos` is assigned a normalized cost equal to the number of possible `Top_pos` choices.

3. Each position P in `Bottom_poses` must be such that `Top_pos` is a proper prefix of P. We also require that `Bottom_poses` does not contain any two positions P and P' such that P is a prefix of P'. We eliminate equivalent permutations of `Bottom_poses` by requiring that it is sorted according to the relation $<$ on positions. Yet another requirement on `Bottom_poses` is that the reused part of H must have size 2 or more. However, we allow E_i's of size 1 provided that the size constraints given above are satisfied. For a given choice of `Top_pos`, each possible choice of `Bottom_poses` is assigned a normalized cost equal to the number of possible choices.

Here is an example that shows the computation of the normalized cost of an ABSTR transformation.

Example. Assume that an ABSTR with arity 2 is to be applied to the expression tree in Figure 4.4. Since $n = 2$, the size of $H(E_1, E_2)$ is required to be 6 or more. This means that there are three choices of `Top_pos`, namely

1. `[]`
2. `[0]`
3. `[1]`

Assume that we choose `Top_pos` = `[1]`. Since the reused part of H is not allowed to have size 1, we cannot have

$$\texttt{Bottom\_poses} = \{\texttt{[1,0]}, \texttt{[1,1]}\}.$$

Therefore, `Bottom_poses` may only be chosen to one of the following alternatives.

1. {`[1,0]`, `[1,1,0]`}
2. {`[1,0,0]`, `[1,0,1]`}
3. {`[1,0,0]`, `[1,1]`}
4. {`[1,0,0]`, `[1,1,0]`}
5. {`[1,0,1]`, `[1,1]`}
6. {`[1,0,1]`, `[1,1,0]`}

The cost of the arity choice is 2. The `Top_pos` choice has cost 3. Each of the 6 choices of `Bottom_poses` has cost 6. Thus, the normalized cost of an ABSTR with arity 2 and `Top_pos` = `[1]` is $2 \cdot 3 \cdot 6 = 36$. □

In general, the cost of an ABSTR depends on the structure of the expression tree. Since this structure varies from program to program, it is difficult to do an exact analysis of the cost. However, we will now derive an upper limit on the cost. Let

$n_{\max} =$ The maximum allowed arity i.e., 2 in the current implementation.

$N_{\text{tot}} =$ The size of the expression tree.

$N_{\text{ABSTR}} =$ The size of $H(E_1, \ldots, E_n)$.

The cost of the choice of arity is of course $n_{\max}$. `Top_pos` may be chosen in no more than N_{tot} different ways. The constraint $N_{\text{ABSTR}} \geq 2 + 2n$ means that the number of choices is normally much less than N_{tot}, but it is difficult to provide a much tighter bound without further assumptions about the tree structure. If we ignore the prefix constraints on positions and the size constraint on H, there are

$$\left(\begin{array}{c} N_{\text{ABSTR}} \\ n \end{array} \right)$$

possible choices of $E_1, \ldots, E_n$. Therefore, the cost of an ABSTR is bounded by

$$n_{\max} \cdot N_{\text{tot}} \cdot \left(\begin{array}{c} N_{\text{ABSTR}} \\ n \end{array} \right) < n_{\max} \cdot N_{\text{tot}} \cdot \frac{N_{\text{ABSTR}}^n}{n!}.$$

For fixed $n_{\max}$, the interesting question is how this expression depends on n, N_{tot} and N_{ABSTR}. If n and N_{ABSTR} also are fixed, the cost bound is obviously proportional to N_{tot}. For varying n and N_{ABSTR}, there is a tug-of-war between the increasing factor N_{ABSTR}^n and the decreasing factor $1/(n!)$. Experimental results are more useful than the formula above. The reason is that it is important to consider the statistical properties, e.g. tree structure, of the programs and ABSTR transformations that arise in practice.

ABSTR transformations are implemented by the following function.

```
fun ABSTR_trfs( D : dec, Cost_limit : real, top_pos_ok : pos->bool,
    bottom_poses_ok : pos list -> bool,
    emit : dec*atomic_trf_record list*real->unit ) : unit = ...
```

This function is rather simple to implement. Therefore, we omit a more detailed discussion of the implementation. However, it is worth noting that no E_i may contain a symbol declared in H since this would lead to a violation of the scope rules of ML.

4.2.4 The CASE-DIST Transformation Algorithm

A CASE-DIST transformation consists of a sequence of *moves*. Each move corresponds to one of the following schemas in which h denotes a function symbol.

1. (a) $h(A_1, \ldots, A_{i-1},$ `case` E `of` $Match_1$ `=>` E_1 `|` $\ldots$ `|` $Match_n$ `=>` $E_n, A_{i+1}, \ldots, A_m$)
   $\longrightarrow$
   `case` E `of`
   $Match_1$ `=>` $h(A_1, \ldots, A_{i-1}, E_1, A_{i+1}, \ldots, A_m)$
   $\vdots$
   `|` $Match_n$ `=>` $h(A_1, \ldots, A_{i-1}, E_n, A_{i+1}, \ldots, A_m)$

   (b) `case` E `of`
   $Match_1$ `=>` $h(A_1, \ldots, A_{i-1}, E_1, A_{i+1}, \ldots, A_m)$
   $\vdots$
   `|` $Match_n$ `=>` $h(A_1, \ldots, A_{i-1}, E_n, A_{i+1}, \ldots, A_m)$
   $\longrightarrow$
   $h(A_1, \ldots, A_{i-1},$ `case` E `of` $Match_1$ `=>` E_1 `|` $\ldots$ `|` $Match_n$ `=>` $E_n, A_{i+1}, \ldots, A_m)$

2. (a) `case` E' `of`
   $Match'_1$ `=>` A_1
   $\vdots$
   `|` $Match'_i$ `=>` `(` `case` E `of` $Match_1$ `=>` E_1 `|` $\ldots$ `|` $Match_n$ `=>` E_n `)`
   $\vdots$
   `|` $Match'_m$ `=>` A_m

$\longrightarrow$

```
case E of
  Match_1 => (
    case E' of
      Match'_1 => A_1
    ⋮
    | Match'_i => E_1
    ⋮
    | Match'_m => A_m )
⋮
| Match_n => (
    case E' of
      Match'_1 => A_1
    ⋮
    | Match'_i => E_n
    ⋮
    | Match'_m => A_m )
```

(b)

```
case E of
  Match_1 => (
    case E' of
      Match'_1 => A_1
    ⋮
    | Match'_i => E_1
    ⋮
    | Match'_m => A_m )
⋮
| Match_n => (
    case E' of
      Match'_1 => A_1
    ⋮
    | Match'_i => E_n
    ⋮
    | Match'_m => A_m )
```

$\longrightarrow$

```
case E' of
  Match'_1 => A_1
⋮
```

```
  | Match'_i => ( case E of Match_1 => E_1 | ... | Match_n => E_n )
  ⋮
  | Match'_m => A_m
```

3. (a)
```
let
  Declarations
in
  case E of Match_1 => E_1 | ... | Match_n => E_n
end
⟶
case E of
  Match_1 => let Declarations in E_1 end
  ⋮
  | Match_n => let Declarations in E_n end
```

   (b)
```
case E of
  Match_1 => E_1
  ⋮
  | Match_i => let Declarations in E_i end
  ⋮
  | Match_n => E_n
⟶
let
  Declarations
in
  case E of
    Match_1 => E_1
    ⋮
    | Match_i => E_i
    ⋮
    | Match_n => E_n
end
```

Note that the schemas 1a, 1b and the schemas 2a, 2b are inverses of each other whereas the schemas 3a, 3b are not. Each a-schema moves the `case` outwards whereas each b-schema moves the `case` inwards.

A move according to schema 2a frequently leads to the introduction of dead code. Therefore, each such move is immediately followed by dead code elimination, which, for example, is capable of replacing an expression

```
case E' of
  Match'_1 => A_1
```

$\vdots$

| *Match'*$_i$ `=>` E_j

$\vdots$

| *Match'*$_m$ `=>` A_m

with some A_k if only the alternative *Match'*$_k$ is activated during execution for all sample inputs in the specification.

A move according to schema 3a frequently leads to the introduction of unnecessary declarations. These are removed by occurrence checking that does not require program execution. For example, the expression

```
let fun g Xs = f(Xs@Xs@Xs) in Ys@Xs end
```

is replaced by `Ys@Xs` since `g` does not occur in `Ys@Xs`.

Schema 2b is rarely used in practice. The experimental experience has shown that it is very difficult to anticipate which schemas that are needed and which are not. This means that it makes sense to also include schemas that seem to be theoretical artifice since they may be employed in unexpected and useful ways.

In order to explain how move sequences are generated, we need the concept of a move graph. Each node D in a move graph is a program. If there is an allowed move M that transforms D to D', there is a directed edge, labelled with M, from node D to node D'. The program to which a CASE-DIST transformation is to be applied corresponds to the start node.

A marking scheme is employed to ensure that the moves in a move sequence are "related". Marking will be explained with respect to the roots of the left and the right hand sides of the schemas. Note that the RHS root of each a-schema is a `case`. When an a-schema is applied, the RHS root and its children are marked. Similarly, when a b-schema is applied, the LHS root and its children are marked. The ML type of marked expressions is

```
(bool*'a,'b)e,
```

where the Boolean value is `true` if and only if the node is marked. The expression type constructor `e` was defined in Figure 2.2. The moves emanating from the start node may use any subexpression of the program that corresponds to the start node as the LHS root. Other moves are only allowed if the LHS root is marked or has at least one marked child or parent in the expression tree.

The move graph may contain very many nodes, each of which is a program that requires rather much space to store. Therefore, explicit construction of the move graph is too space consuming, which means that we should not use breadth-first search of the graph to find move sequences. Since we want to find move sequences in order of increasing length, we use iterative-deepening to search the graph. It is important to avoid visiting the same node more than once. This is achieved by storing the fingerprint of each visited node in a hash

table. Fingerprinting was discussed in Subsection 4.2.2. Of course, a fingerprint usually requires many orders of magnitude less storage space than a program e.g. 8 bytes versus 10^4 bytes.

Before presenting the ML implementation of the iterative-deepening search, we need a few auxiliary functions. The first auxiliary function is

```
fun find_children( E : (bool*'a,'b)e, Mark_enable : bool,
      emit : bool*(bool*'a,'b)e -> unit ) : unit = ...
```

The marked expression `E` is the right hand side of a program. The function `find_children` makes the call

```
emit(Dead_code_elim,New_E)
```

for each `New_E` such that there is an edge from the move graph node, that corresponds to `E`, to the node that corresponds to `New_E`. `Dead_code_elim` is `true` if and only if dead code elimination should be applied to `New_E`.

We use a hash table `Table`. Each entry in `Table` stores the fingerprint of an expression together with a pair of the form `( No_of_moves, Found )`. `No_of_moves` is the length of the shortest path, that has been found so far, from the start node to the node corresponding to the fingerprinted expression. `Found` is `true` if and only if the expression has been produced earlier during the current iteration.

We also need the auxiliary function `iterate` shown in Figure 4.5. The parameter `Move_count` is the length of the move graph path from the start node to the node corresponding to `D`. The parameter `Move_count_limit` is the maximum path length, which is deepened iteratively. The rest of the definition of `iterate` is self-explanatory. It is now easy to define a function `iteration` that performs one iteration. This function is called with `Move_count_limit`$= 1, 2, 3, \ldots$. Since the limit is increased by one from one iteration to the next, `Table` always contains the length of the shortest path from the start node to the node corresponding to a fingerprint. Therefore, the test

```
Move_count+1 < No_of_moves
```

in Figure 4.5 actually never becomes true, but it is still wise to include it in anticipation of future implementation changes. One such change would be to use iterative-deepening with extrapolation as in [Olsson 93], which would reduce the time wasted on re-expansion. The definition of `iteration` shown in Figure 4.6 is easy to understand without any further explanation.

In order to determine the normalized costs of CASE-DIST transformations, we first do a "dry" iterative-deepening search that does not emit any programs. This dry search is terminated when all allowed move sequences have been found or when a search time limit `Max_time` has been exceeded. The function `dry_search` shown in Figure 4.7 returns a pair of the form *(N,Cs)*, where

```
fun iterate( D as {func,pat,exp,dec_info} : (bool*'a,'b)d, Move_count,
      Move_count_limit, emit : (bool*'a,'b)e -> unit ) : unit =
  if Move_count = Move_count_limit then
    emit exp
  else
    let fun emit'(Dead_code_elim,New_exp) =
      if not(scope_check(New_exp,func::nil,vars_in_pat pat)) then
        ()
      else
      let
        val New_D = { func=func, pat=pat, exp=New_exp,
          dec_info=dec_info }
        val New_D as {exp=New_exp,...} =
          if Dead_code_elim then dead_code_elim' New_D else New_D
        val Fingerprint = exp_hash(rename(New_exp,true))
      in
      case H.peek (!Table) Fingerprint of
        NONE => (
          H.insert (!Table) (Fingerprint,(Move_count+1,true));
          iterate(New_D,Move_count+1,Move_count_limit,emit)
          )
      | SOME(No_of_moves,Found) =>
          if Move_count+1 < No_of_moves orelse
             (Move_count+1=No_of_moves andalso not Found) then (
            H.remove (!Table) Fingerprint;
            H.insert (!Table) (Fingerprint,(Move_count+1,true));
            iterate(New_D,Move_count+1,Move_count_limit,emit)
            )
          else
            ()
      end
    in
      find_children(exp,true,emit')
    end (* fun iterate *)
```

Figure 4.5: A help function for iterative-deepening.

```
fun iteration( D as {func,pat,exp,dec_info} : ('a,'b)d,
      Move_count_limit : int,
      emit :  ('a,'b)d * pos list -> unit ) : unit =
  let
    val All_case_poses = all_poses_filter(is_case_exp,#exp D)
    val exp = to_marked exp
    val pat = to_marked pat
  in
    Table := H.transform( fn(Move_count,_) => (Move_count,false))
              (!Table);
    map( fn Case_pos => iterate(
        {func=func,pat=pat,exp=mark_exp_at_pos(exp,Case_pos),
         dec_info=dec_info} :(bool*'a,'b)d,
        0,
        Move_count_limit,
        fn New_exp => emit( {func=func,pat=from_marked pat,
            exp=from_marked New_exp, dec_info=dec_info} : ('a,'b)d,
          marked_poses New_exp)),
      All_case_poses );
    ()
  end
```

Figure 4.6: Performing one iteration.

```
fun dry_search( D : ('a,'b)d, Max_time : real ) : int * array =
  let
    val Class_cardinalities = array(50,0)
    val Max_move_count_limit = ref 0
    val Emitted = ref true
    fun emit(New_D,_) = (
      update(Class_cardinalities,!Max_move_count_limit,
        sub(Class_cardinalities,!Max_move_count_limit)+1);
      Emitted := true
      )
    fun run() = if not(!Emitted) then () else (
      Emitted := false;
      inc Max_move_count_limit;
      iteration(D,!Max_move_count_limit,emit);
      run()
      )
  in
    init_hash_table(#exp D);
    timeLimit (real_to_time Max_time) run ()
    handle Time_out => ();
    (!Max_move_count_limit - 1, Class_cardinalities)
  end
```

Figure 4.7: Dry search that is needed to determine costs.

the dynamic array *Cs* contains the number of nodes at distance i from the start node for each i in $\{1, 2, \ldots, N\}$.

A so-called class consists of all nodes at the same distance from the start node. All programs in a given class are assigned the same cost, which increases with the distance from the start node. Let $c_1, \ldots, c_N$ be the elements in the dynamic array *Cs*. Let $\sigma_1, \ldots, \sigma_N$ be the sequence of cumulative sums of *Cs* i.e.,

$$\sigma_k = \sum_{i=1}^{k} c_i.$$

The cost of each program in class number k is chosen to be proportional to σ_k. In practice, it is common that c_k grows exponentially with k. If we assume $c_k = b^k$ for some branching factor b, we have

$$\sigma_k = \frac{b^{k+1} - 1}{b - 1} - 1,$$

which gives $\sigma_{k+1}/\sigma_k \approx b$. However, it may occasionally happen that c_k decreases with k. If we chose the program cost of class number k to be proportional to c_k, decreasing c_k would mean that programs produced with long move sequences would be cheaper than programs produced with short move sequences. Since this is undesirable, we use proportionality to σ_k instead of proportionality to c_k.

Assuming that K is the normalizing factor, we require

$$\sum_{i=1}^{m} \frac{c_i}{K\sigma_i} = 1,$$

where m is the greatest class index such that

$$K\sigma_m < \texttt{Cost\_limit}.$$

As for all other transformations, `Cost_limit` is the specified maximum cost of a CASE-DIST transformation. Cost computation is implemented by the following function.

```
fun cost_comp( Cost_limit : real, Max_move_count_limit : int,
      Class_cards : array ) : int * (int->real) = ...
```

This function returns the pair (m, φ), where the cost function φ is such that $\varphi(i) = K\sigma_i$ for each class index i. We now have all auxiliary functions that are needed to implement the `CASE_DIST_trfs` function, which is shown in Figure 4.8. The only parameter to `CASE_DIST_trfs` that has not been discussed is `CASE_DIST_cost_limit`, which says how much work that is to be expended on finding move sequences.

```
fun CASE_DIST_trfs( D : ('a,'b)d, Cost_limit : real,
     CASE_DIST_cost_limit : real,
      emit : ('a,'b)d * atomic_trf_record list * real -> unit )
      : unit =
  let
    val (Max_move_count_limit,Class_cards) =
      dry_search( D,
        0.4*CASE_DIST_cost_limit*synt_and_eval_time_per_exp() )
    val (Max_move_count_limit,Cost_comp) =
      cost_comp(Cost_limit,Max_move_count_limit,Class_cards)
  in
    init_hash_table(#exp D);
    map(fn Move_count_limit =>
      let fun emit'(New_D,Active_poses) =
        emit(New_D, CASE_DIST{activated_poses=Active_poses}::nil,
          Cost_comp Move_count_limit)
      in
        iteration(D,Move_count_limit,emit')
      end,
      fromto(1,Max_move_count_limit)
      );
    ()
  end
```

Figure 4.8: The implementation of the CASE-DIST transformation.

```
fun zeroth_order_ground_types(
        ty_con_exp( "->", Domain::Range::nil ) ) =
      zeroth_order_ground_types Domain @
      zeroth_order_ground_types Range
  | zeroth_order_ground_types( ty_con_exp( "tuple", Comp_types ) ) =
      flat_map( zeroth_order_ground_types, Comp_types )
  | zeroth_order_ground_types( T as ty_con_exp(_,Comp_types) ) =
      T :: flat_map( zeroth_order_ground_types, Comp_types )
  | zeroth_order_ground_types _ = nil
```

Figure 4.9: The ML definition of zeroth order ground types.

4.2.5 The EMB Transformation Algorithm

This algorithm needs to make the following choices.

1. The `let`-function that is to be embedded.
2. If the domain or the range is to be embedded.
3. Whether to use way 1 or way 2 of embedding tuple types.
4. If way 1 was chosen, the algorithm chooses `'a` to a so-called zeroth order ground type that occurs in the program to be transformed. This concept is defined below. If way 2 was chosen, the algorithm chooses the index i of the tuple type component T_i, the data type definition to be used and also a $T_{j,k}$ in the RHS of that definition.
5. One newly synthesized expression for each `?_emb` constant.

We will now discuss each of these choices.

1. If there are l `let`-functions that may be embedded, each `let`-function is assigned the normalized cost l.
2. Both choices are assigned the normalized cost 2.
3. Both choices are assigned the normalized cost 2.
4. **Way 1.** The concept "zeroth order ground type" is defined by the function shown in Figure 4.9, which extracts the zeroth order ground types that occur in a type expression. Recall that the representation of types was shown in Figure 4.1. The auxiliary `flat_map` function is defined as usual in functional programming i.e.,

   ```
   fun flat_map( f, Xs ) =
     case Xs of nil => nil | X1::Xs1 => f(X1)@flat_map(f,Xs1)
   ```

Assume that `RHS` is the right hand side of the program to be transformed. The set of allowed `'a` choices is given by

```
hash_make_set( exp_flat_map(
  zeroth_order_ground_types o type_of_exp, RHS ) )
```

The auxiliary `exp_flat_map` function is analogous to `flat_map` but defined on the type `('b,'c)e` instead of the type `'d list`. Each allowed choice of `'a` is assigned a normalized cost equal to the number of allowed choices.

Way 2. Given that the tuple type is $T_1 * \ldots * T_n$, the normalized cost of choosing T_i is n. Each choice of a data type definition, that may be employed to embed T_i, is assigned a normalized cost equal to the number of choices, which usually is only 1 or 2. Similarly, each $T_{j,k}$ in the RHS of the definition is assigned a normalized cost equal to the number of $T_{j,k}$'s in the RHS that match T_i.

5. This part of the implementation is somewhat more complicated than the four parts above. There are two issues, namely

   (a) How to find a list of candidate expressions for each `?_emb` constant. Assume that there are m `?_emb` constants i.e., that m lists are to be found.

   (b) How to combine the expressions in the m lists to expression sequences of length m and how to define the costs of these sequences.

   We first consider issue a and then issue b.

   Issue a is solved using the expression synthesizing `Replace.replace` function that also was used in the implementation of `R_trfs` in Figure 4.2. Figure 4.10 shows the implementation of a function `find_emb_expss` that returns a list containing the m lists that are to be found. The parameter `EMB_cost_limit`, that says how much work that is to be expended on expression synthesis, is uniformly distributed on the m `?_emb` positions in the parameter `Q_emb_poses`. A priority queue, implemented as a heap, is employed to ensure that each of the m lists are sorted in order of increasing syntactic complexity. In order to save memory space, the current implementation has an upper limit of 500 on the cardinality (`Max_heap_size`) of the heap.

   We now turn to issue b. Empirically, we have noted that the synthesized expressions that are needed usually are very small. For example, if the domain of `g` is embedded in way 1, each call `g`$(E_1, \ldots, E_n)$ is changed to `g`$(E_1, \ldots, E_n$`,?_emb)`, where the expression that is to replace the `?_emb` usually is very small. The synthesized expressions that are needed for R

```
exception Find_emb_expss
fun find_emb_expss( D : dec, EMB_cost_limit : real,
    Q_emb_poses : pos list ) : exp list list =
  if null Q_emb_poses then raise Find_emb_expss else
  let
    val Cost_limit = EMB_cost_limit / real(length Q_emb_poses)
    fun find_emb_exps( Q_emb_pos : pos ) : exp list =
      let
        val Es = ref( Heap.heap_nil )
        fun emit(New_D:dec,Cost,Not_activated_syms) =
          let
            val E = pos_to_sub(#exp New_D,Q_emb_pos)
            val Complexity = Evaluate.syntactic_complexity New_D
          in
            Es := Heap.heap_insert( (E,Complexity), !Es);
            if Heap.heap_size(!Es) > Max_heap_size then
              Es := (case Heap.heap_delete_min(!Es) of
                        SOME(_,New) => New)
            else
              ()
          end
      in
        Replace.replace( D, Q_emb_pos, nil,
          Cost_limit, nil, emit );
        rev( map(#1,Heap.heap_report( !Es )) )
      end
  in
    map( find_emb_exps, Q_emb_poses )
  end
```

Figure 4.10: Finding a list of lists of candidate expressions.

and REQ transformations are generally bigger. Recall that the unnormalized cost of a REQ sequence of length n was chosen to

$$\prod_{i=1}^{n}(x_i + 3),$$

where x_i is the order number of REQ number i in the sequence. The unnormalized cost of an EMB expression sequence of length m is chosen to

$$\prod_{i=1}^{m}(x_i + B)^2,$$

where B is a constant that is chosen to 7 in the current implementation. Thus, the EMB cost measure penalizes high order numbers more than the REQ cost measure. This is illustrated by the following example.

Example. Assume $n = m = 2$. Since $x_i \geq 1$, the lowest unnormalized REQ cost is 4^2 and the lowest unnormalized EMB cost is 8^4. Let us define

$$C_{\mathrm{REQ}}(x_1, x_2) = \frac{(x_1 + 3)(x_2 + 3)}{4^2}$$

and

$$C_{\mathrm{EMB}}(x_1, x_2) = \frac{((x_1 + 7)(x_2 + 7))^2}{8^4}.$$

The functions C_{REQ} and C_{EMB} give the ratio between the current cost and the lowest cost. Therefore, they are useful for illustrating the penalization of high order numbers. Note that C_{REQ} and C_{EMB} would not change if we used normalized costs since the normalization factor would be cancelled out by the division. For example, since

$$C_{\mathrm{REQ}}(10, 10) \approx 10.6$$

and

$$C_{\mathrm{EMB}}(10, 10) \approx 20.4,$$

we can say that the EMB cost measure penalizes the order numbers $x_1 = x_2 = 10$ about twice as much as the REQ cost measure.

Figures 4.11 and 4.12 show contour curves for C_{REQ} and C_{EMB} respectively. There are 15 units between two neighbouring curves in each figure. The bottommost curve in each figure is for C_{REQ} and C_{EMB} equal to 16. For example, it is easy to see that curve number 3 from the bottom in Figure 4.11 approximately corresponds to curve number 15 in Figure 4.12. This means that the order number values, that make $C_{\mathrm{REQ}} = 1+3\cdot15 = 46$, make $C_{\mathrm{EMB}} = 1 + 15 \cdot 15 = 226$. Also note that both figures show a small cost ratio if one order number is large and the other small. □

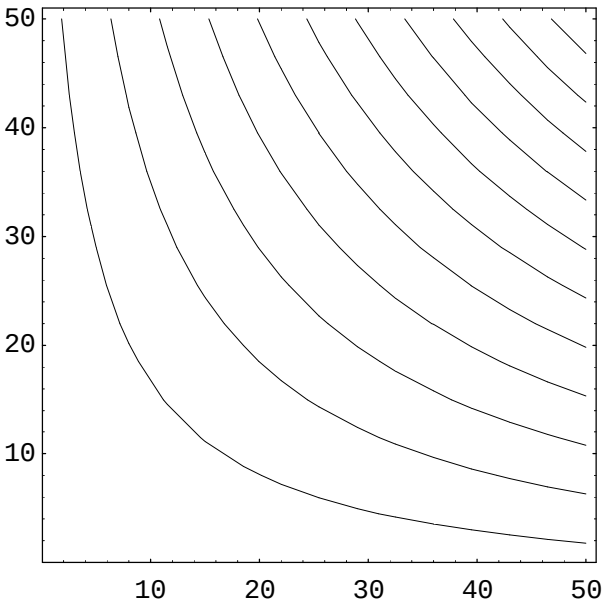


Figure 4.11: Contour curves for C_{REQ}

Given that `Exp_counts` is a list containing the lengths of the expression lists returned by `find_emb_expss`, the following function returns the normalizing factor.

```
fun compute_factor( EMB_cost_limit : real, Cost_limit : real,
      Exp_counts : int list ) : real = ...
```

The parameter `EMB_cost_limit` says how much work that should be spent on finding an accurate normalizing factor.

Figure 4.13 shows the definition of a function `replace_q_embs` that replaces the `?_embs` at the positions given by the parameter `Q_emb_poses` with newly synthesized expressions. The auxiliary `replace_q_embs'` function is defined in Figure 4.14.

EMB transformations are implemented by the function `EMB_trfs`, which has a straightforward but long-winded definition. The LHS of the definition is

```
fun EMB_trfs( D : dec, Cost_limit : real, EMB_cost_limit : real,
      top_pos_ok : pos -> bool,
      emit : dec * atomic_trf_record list * real -> unit )
      : unit = ...
```

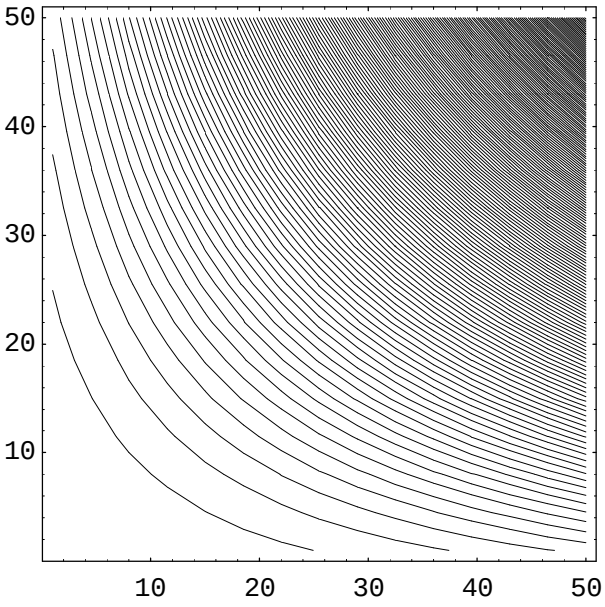


Figure 4.12: Contour curves for C_{EMB}

```
fun replace_q_embs( D : dec, Cost_limit : real, EMB_cost_limit : real,
      Q_emb_poses : pos list, emit : dec*real -> unit ) : unit =
  let
    val Ess = find_emb_expss( D, EMB_cost_limit*0.75,  Q_emb_poses )
    val Factor = compute_factor( EMB_cost_limit*0.25, Cost_limit,
      map(length,Ess) )
  in
    replace_q_embs'( D, Cost_limit, Q_emb_poses, Ess, Factor, emit )
  end
```

Figure 4.13: Replacing `?_embs`.

```
fun replace_q_embs'( D : dec, Cost_limit : real,
      Q_emb_poses : pos list, Ess : exp list list,
      Cost_so_far : real, emit : dec*real -> unit ) =
  if Cost_so_far *
     real_pow( order_no_cost 1, real(length Q_emb_poses) ) >
     Cost_limit
  then
    ()
  else
  case Q_emb_poses of
    nil => emit(D,Cost_so_far)
  | Pos::Poses =>
  case Ess of Es::Ess =>
    let
      val N = length Es
      val Cost_left = Cost_limit/Cost_so_far
      val Max_I =
        if null Poses then
          min2( op<, N, order_no_cost_inverse Cost_left )
        else
          N
    in
      map( fn(I,E) =>
        replace_q_embs'( pos_replace_dec(D,Pos, fn _ => E),
          Cost_limit, Poses, Ess, Cost_so_far*order_no_cost I,
          emit ),
      combine(fromto(1,Max_I),take(Max_I,Es)) );
      ()
    end
```

Figure 4.14: The auxiliary `replace_q_embs'` function.

Chapter 5

Expression Synthesis

5.1 Description of the Expression Synthesis Problem and Its Complexity

The problem of synthesizing expressions may be defined as follows. Given a type T and a set of components, consisting of variables, functions and their types, we want to produce N expressions of type T. We require that the expressions are produced in order of increasing syntactic complexity and that they are typed in accordance with the components. Since it would be quite complicated to accomplish this using the syntactic complexity measure specified in Appendix A, which was discussed in Subsection 3.5.1, we use expression size, i.e., the number of nodes in the expression tree, as the syntactic complexity measure. Thus, we first generate all expressions of size 1, then all expressions of size 2 and so forth.

The set of all expressions of size less than or equal to some maximum size $s_{\max}$ is partitioned into equivalence classes such that all expressions in a class have the same semantics. The difference between the total number of expressions in this set and the number of classes may be enormous as illustrated by the following example.

Example. Assume that expressions of type `'a list` are to be produced using the components

```
Xs :  'a list
nil : 'a list
@ :  'a list * 'a list -> 'a list
```

The component `Xs` is a variable. The other two components are functions that are predefined in Standard ML. Let $T_t(s_{\max})$ be the total number of expressions of size $s_{\max}$ or less. Let $T_n(s_{\max})$ be the number of non-equivalent expressions of size $s_{\max}$ or less. Obviously, an expression tree of size s has $(s+1)/2$ leaves and $(s-1)/2$ internal nodes. Each leaf is either `nil` or `Xs` which gives $2^{(s+1)/2}$

possible choices of leaves. The number of binary trees with i internal nodes is a well-known combinatorial function, the so-called i'th Catalan number, which equals

$$\frac{1}{i+1}\left(\begin{array}{c}2i\\ i\end{array}\right) \approx \frac{4^i}{(i+1)\sqrt{\pi i}},$$

where we obtained the right hand side using Stirling's approximation of the factorial function. Summing for each number of internal nodes yields

$$T_t(s_{\max}) \approx \sum_{i=0}^{(s_{\max}-1)/2} \frac{2^{i+1}4^i}{(i+1)\sqrt{\pi i}} = \sum_{i=0}^{(s_{\max}-1)/2} \frac{2\cdot 8^i}{(i+1)\sqrt{\pi i}}.$$

It is easy to see that $T_t(s_{\max})/T_t(s_{\max}-2) \approx 8$ for large $s_{\max}$ i.e., exponential growth with a branching factor of $\sqrt{8} \approx 2.8$. However, all expressions that contain the same number of occurrences of `Xs` are equivalent. Therefore,

$$T_n(s_{\max}) = (s_{\max}+1)/2 + 1,$$

which is one more than the maximum number of occurrences of `Xs` in an expression tree of size $s_{\max}$.

In practice, we usually find it acceptable to try about 10^5 expressions. If we manage to synthesize one and only one expression per equivalence class, we would, in this example, be able to synthesize expressions of size up to about $2\cdot 10^5$. However, if we need to synthesize all expressions of size $s_{\max}$ or less, there is an upper size limit of about 15 as can be seen in Figure 5.1 which shows

$$\log_{10} T_t(s_{\max}) = \log_{10} \sum_{i=0}^{(s_{\max}-1)/2} 2^{i+1}\frac{1}{i+1}\left(\begin{array}{c}2i\\ i\end{array}\right)$$

□

In the example above, $T_t(s)$ is exponential in s whereas $T_n(s)$ is linear in s. It is much more common that both $T_t(s)$ and $T_n(s)$ grow exponentially with s.

5.2 Expression Synthesis in ADATE

This section discusses methods and heuristics for expression synthesis that have been implemented in ADATE. The next section presents alternatives and extensions to these methods.

5.2.1 The Interface to Expression Synthesis

We will first describe the top level interface of the expression synthesis implementation, which is the following function.

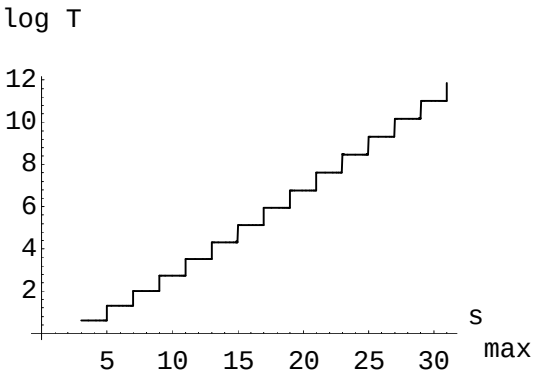


Figure 5.1: The logarithm of expression space cardinality as a function of size.

```
fun replace( D : dec, Top_pos : pos, Bottom_poses : pos list,
      Cost_limit : real, Min_once : symbol list list,
      emit : dec*real*symbol list->unit ) : unit = ...
```

This function was used under the name `Replace.replace` in the implementation of R transformations in Figure 4.2 and in the implementation of replacement of `?_emb` constants in Figure 4.10. It is also employed in the implementation of the `find_REQs` function mentioned in Subsection 4.2.2. The parameters of `replace` were described in Subsection 4.2.1. To begin with, we need to compute the components that may be used at position `Top_pos` in the program `D`. These components are computed by the call `comps_at_pos( D, Top_pos )`. The implementation of `comps_at_pos` shown in Figure 5.2 uses the following global variables and functions.

`comps_in_pat` Returns the components in a pattern.

`pos_fold` The implementation of this higher order function was given in Figure 2.3.

`Comps_to_use` Contains the components listed in the specification.

Recall that `Bottom_poses` specifies the positions of the E_i's that occur in the general replacement transformation schema

$$H(E_1, \ldots, E_n) \longrightarrow G(E_1, \ldots, E_n).$$

We require that each E_i occurs exactly once in $G(E_1, \ldots, E_n)$. The expressions $E_1, \ldots, E_n$ are represented by special components which we will denote by $\chi_1, \ldots, \chi_n$.

We will now present variable bindings made by the implementation of `replace`.

```
fun comps_at_pos( {func,pat,exp,dec_info=SOME Sch}, Pos : pos )
    : ty_env =
  let fun g _ = nil
    fun f(Comps,E,P::_) =
      case E of
        case_exp{rules,...} =>
          if P = 0 then
            Comps
          else
            comps_in_pat(#pat(nth(rules,P-1))) @ Comps
      | let_exp{dec_list,...} =>
          if P < length dec_list then
            let val {func,pat,exp,dec_info=SOME Sch} = nth(dec_list,P)
            in
              (func,Sch) :: comps_in_pat pat @ Comps
            end
          else
            Comps
      | _ => Comps
  in
    (func,Sch) :: comps_in_pat pat @
    pos_fold(f,g,Pos,exp) @ Comps_to_use
  end
```

Figure 5.2: Finding the components at a given position.

`Components` The list `comps_at_pos( D, Top_pos )` concatenated with the list of all pairs consisting of χ_i and its type schema.

`subst_fun` A substitution function of type `exp -> exp` that replaces each χ_i with the corresponding E_i.

`Max_once` Symbols that only are allowed to occur 0 or 1 times in each synthesized expression. Is bound to `[`χ_1`,...,`χ_n`]`.

`Min_once'` Symbols that must occur at least once. Is bound to `[[`χ_1`], ..., [`χ_n`]] @ Min_once`.

`emit_synted_exp` A function that takes a synthesized expression `G`, transforms `D` and calls `emit` with the resulting program.

With these bindings, the implementation of `replace` calls a more pure expression synthesis function `synt_n` as follows.

```
synt_n( type_of_exp(pos_to_sub(exp,Top_pos)), Components,
  subst_fun, D, Top_pos, Max_once, Min_once', emit_synted_exp,
  Cost_limit )
```

Note that the requirement that each χ_i occurs exactly once is implemented by putting χ_i in both `Max_once` and `Min_once'`. The definition of `synt_n` has the following form.

```
fun synt_n( Type : ty_exp, Components : ty_env, subst_fun : exp->exp,
    Current_prog : dec, Pos : pos, Max_once : symbol list,
    Min_once : symbol list list, emit : exp*real*symbol list->unit,
    N : real ) = ...
```

Expressions may be synthesized bottom-up or top-down. Bottom-up synthesis determines all the subtrees of a node before the node itself. Top-down synthesis determines the node before any of its subtrees. Of course, there are many possible bottom-up and top-down orders of the nodes in a tree. Postorder is an example of a bottom-up order. Preorder is an example of a top-down order.

The function `synt_n` synthesizes expressions top-down since this is easier to implement than bottom-up. The implementation of `synt_n` and its auxiliary functions consists of about 1100 lines of Standard ML code. Therefore, we will not present the complete implementation, but try to give a comprehensible overview. We start by giving a complete but highly simplified implementation of `synt_n`. Then, we describe additional techniques and heuristics that are employed in the real implementation.

```
fun exp_size( app_exp{args,...} ) =
  1+int_sum(map( exp_size, args ))

fun while_list( continue : unit -> bool, Xs, f ) : unit =
  case Xs of
    nil => ()
  | X1::Xs1 =>
      if continue() then
        ( f(X1); while_list(continue,Xs1,f) )
      else
        ()
```

Figure 5.3: Computing expression size and iterating over lists.

5.2.2 A Simplified Implementation of `synt_n`

In order to obtain a simplified implementation, assume that expressions of type T are to be synthesized and that all components have types of the form

$$T * T * \ldots * T\texttt{->}T,$$

where the left hand side contains 0, 1, 2 or more occurrences of the type T. This means that the only difference between component types is their arity. The type of a component of arity n is represented as a `unit list` of length n. The `synt_n` parameter `Components` is a list of (symbol, type) pairs. For example, the components `Xs`, `nil` and `@`, that were used in the first example in Section 5.1, correspond to

`Components` = `[ ("Xs",[]), ("nil",[]), ("@",[(),()]) ]`.

We also assume that the only parameters of `synt_n` are `Components`, an `emit` function and `N`, i.e., the number of expressions to be synthesized.

The implementation uses the auxiliary functions `exp_size` and `while_list` shown in Figure 5.3. The former returns the size of an expression. The latter is a list iteration "function" that makes the call `f X` for each element `X` in `Xs` as long as `continue()` is `true`.

Given that `S_max` is the maximum size of expressions to be synthesized, the auxiliary function `synt`, shown in Figure 5.4, makes the call `emit E` for each expression `E` such that `exp_size E <= S_max`.

The implementation of `synt_n` shown in Figure 5.5 calls `synt` with `S_max` = $1, 2, 3, \ldots$ until `N` expressions have been emitted. The total number of synthesized expressions is

$$T(1) + T(2) + T(3) + \ldots,$$

```
fun synt( S_max : int, Components : (symbol * unit list) list,
          emit : exp -> unit, continue : unit -> bool ) =
  if S_max <= 0 then
    ()
  else
    while_list(
      continue,
      Components,
      fn (F,Domain_type) =>
        synt_list( Domain_type, S_max-1, Components,
          fn Es => emit(app_exp{ func=F, args=Es, exp_info=NONE }),
          continue
          )
      )

and synt_list(Types,S_max,Components,emit,continue) =
  case Types of
    nil => emit nil
  | T1::Ts1 =>
      synt( S_max-length(Ts1), Components,
        fn E =>
          synt_list( Ts1, S_max-exp_size(E), Components,
            fn Es => emit(E::Es),
            continue
            ),
        continue
        )
```

Figure 5.4: Synthesizing all expressions of size `S_max` or less.

```
fun synt_n( Components : (symbol * unit list) list, emit : exp -> unit,
    N : int ) : unit =
  let
    val So_far = ref 0
    fun continue() = !So_far < N
  in
    while_list(
      continue,
      fromto(1,1000),
      fn S => synt( S, Components,
        fn E => if S = exp_size E then
                  (inc So_far; emit E )
                else
                  (),
        continue ))
  end
```

Figure 5.5: A highly simplified definition of `synt_n`.

where $T(s)$ is the number of expressions of size s or less. We assume that T grows exponentially with s, which implies that this total number is $O(\mathtt{N})$.

We will now present some aspects of the "real" implementation of `synt_n`.

5.2.3 Restrictions on the Synthesis of Recursive Calls

In the literature, the goal with recursion restrictions is often to guarantee termination. Our primary goal, on the other hand, is to reduce the number of expressions that need to be synthesized.

To say that a function is terminating or non-terminating is a vague and rough characterization of its time complexity. For most purposes, super-exponential time complexity is practically as bad as non-termination. Synthesized programs with such bad time complexity do occasionally arise during an inference. It would be quite difficult to do automatic syntactic time complexity analysis of synthesized programs. Therefore, we employ a call count limit as discussed in Subsection 3.5.2.

One possible restriction on recursive calls would be to only allow primitively recursive definitions. However, there are many algorithms, for example Quicksort, that are difficult to define in a primitively recursive way without major loss of efficiency. We view primitive recursion as being too restrictive to be compulsory, but it could be used as a heuristic guide.

The discussion of so-called terminating generator inductive (TGI) definitions

```
fun f(Xs:int list,Ys) =
  case Xs of
    nil => Xs
  | X1::Xs1 =>
  case Ys of
    nil => Ys
  | Y1::Ys1 =>
  case Y1<X1 of
    true => f(Xs1,Xs@Ys)
  | false => f(Xs@Xs,Ys1)
```

Figure 5.6: A partially non-terminating definition.

in [Dahl 92] inspired the following requirement on recursive calls, which is used in the "real" implementation of `synt_n`. Consider the synthesis of a recursive call $\mathtt{g}(A_1, A_2, \ldots, A_n)$ occurring in the declaration `fun g`$(V_1, V_2, \ldots, V_n)$ `=` $\ldots$. 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 `|` $\ldots$ `|` $Match_m$ `=>` RHS_m and

1. A_i is a proper subexpression of $Match_k$ or
2. $Match_k$ contains a variable W such that A_i is "smaller" than W.

Like primitive recursion, this requirement is too restrictive to allow efficient formulation of many interesting algorithms. Future versions of ADATE will use it only as a guide, for example by allowing up to 50% of the synthesized expressions to contain recursive calls that violate the requirement.

Note that the requirement does not guarantee termination since it only looks at one recursive call at a time. This is illustrated by the following example.

Example. Assume that `f : int list * int list -> int list` is defined as in Figure 5.6 The call `f( [2,2], [1,3] )` is non-terminating even though each recursive call in the definition of `f` satisfies the requirement. However, this is no problem since ADATE employs a call count limit. □

Also note that the requirement is purely syntactic. A less restrictive requirement would be to evaluate A_i and check if the evaluation result has a smaller size than the value of V_i. Of course, this check would be more time consuming than the purely syntactic check currently employed by ADATE.

5.2.4 Restrictions on the Synthesis of `case`-expressions

Assuming that no A or E contains any `case`, only the following three forms of expressions are synthesized.

1. E.

2. `case` A `of` $Match_1$ `=>` E_1 | ... | $Match_n$ `=>` 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, a synthesized expression contains 0, 1 or 2 `case`s such that each `case` occurrence either is the root or a child of the root of the expression. If N expressions are to be synthesized, $N/3$ expressions are chosen according to form 1, $N/3$ according to form 2 and $N/3$ according to form 3.

A `case`-analyzed expression A is sometimes such that the values of A cannot match one or more alternatives in the `datatype` definition of A. This means that one or more `case`-rules are redundant. The implementation of `synt_n` avoids such redundancy as follows.

Assume that `case` A `of` $Match_1$ `=>` $Unknown_1$ | ... | $Match_n$ `=>` $Unknown_n$ 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 $Sub = H(E_1, \ldots, E_m)$ 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` A `of` $Match_1$ `=>` $Unknown_1$ | ... | $Match_n$ `=>` $Unknown_n$; Sub). The resulting program is then executed for all sample inputs. Recall that 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 the special *Not_activated* constant.

This activation requirement is supplemented by requiring that the root of A is not a data type constructor e.g., `false`, `true`, `nil` or `::`.

5.3 Alternative Strategies for Expression Synthesis

5.3.1 Equivalence Checking

The problem of equivalence checking may be stated as follows. Given n synthesized expressions $E_1, \ldots, E_n$ and a newly synthesized expression E', is there any

E_i such that E' and E_i are equivalent in a given environment according to the semantics of Standard ML? In general, this question is undecidable. However, an equivalence checking algorithm may be quite useful even if it errs occasionally.

Equivalence checking is primarily useful if expressions are synthesized bottom-up and in order of increasing size. Let the order $<_{\text{synt}}$ be such that $E_i <_{\text{synt}} E_j$ holds for any two expressions E_i and E_j if and only if E_i is synthesized before E_j by the synthesis algorithm. Bottom-up is preferable to top-down since we can discard each partially synthesized subexpression that is equivalent to some other expression that precedes it in the $<_{\text{synt}}$-order. Thus, bottom-up synthesis allows earlier cut-off than top-down synthesis. This shallow backtracking may lead to substantial reductions of the effort spent on searching for non-equivalent expressions.

An algorithm that needs to ask the question "Is E' equivalent to E_i?" for each i in $\{1, 2, \ldots, n\}$ would take time $\Omega(n)$ to determine if there is any equivalent E_i. If we ran this algorithm for each newly synthesized expression, the total time required for equivalence checking would be $\Omega(n^2)$. Since n may be large, e.g. greater than 10^5, this quadratic time complexity is unacceptable. Therefore, methods such as inductive Knuth-Bendix completion [Kirkerud 92] seem to be fairly useless for equivalence checking.

Another possibility is to use a set of rewrite rules to try to obtain a canonical form for each newly synthesized expression. We could then use a hash table to determine if this form has been seen before, thus avoiding $\Omega(n)$ equivalence checks.

Alternatively, we could employ rewrite rules to try to simplify a newly synthesized expression. Assuming that simplification implies size reduction and that expressions are synthesized in order of increasing size, we can discard each expression that can be simplified.

A major problem is to find suitable rewrite rules. Since synthesized expressions may contain occurrences of `let`-functions that have been defined by ADATE, the rewrite rules should be determined by ADATE based on the definitions of these `let`-functions, which may be rather arbitrary and general recursive functions. With the current state-of-the-art in rewrite systems research, this is unfeasible. Therefore, we do not consider employing rewrite rules or other purely deductive methods for equivalence checking.

We will now discuss a heuristic and more feasible equivalence checking method. Assume that the free variables, that may occur in a synthesized expression, are $X_1, \ldots, X_m$. Also assume that we have a set of sample values $\{J_1, \ldots, J_{\#J}\}$, where each J_k is an m-tuple that is to be substituted for $(X_1, \ldots, X_m)$. The newly synthesized expression E' is considered to be equivalent to a previously synthesized expression E_i if and only if $E'(J_k) = E_i(J_k)$ for all k in $\{1, \ldots, \#J\}$.

Example. Consider the expression synthesis problem in Section 5.1, where the components were the variable `Xs` and the functions `nil` and `@`. Obviously, $m = 1$ and $X_1 =$ `Xs`. Let $\#J = 1$ and $J_1 =$ `[10]`. It is easy to see that two

expressions E' and E_i are equivalent if and only if

$$E'(\texttt{[10]}) = E_i(\texttt{[10]}\ ).$$

For example, `(Xs@Xs)@Xs` is equivalent to `Xs@(Xs@Xs)` since both expressions evaluate to `[10,10,10]` with `Xs` = `[10]`. □

An implementation should compare the fingerprints of

$$\{E'(J_1), \ldots, E'(J_{\#J})\}$$

and

$$\{E_i(J_1), \ldots, E_i(J_{\#J})\}$$

instead of comparing the two sets directly since it would require too much space to store n expression sets for large n. By storing the n fingerprints in a hash table, it is easy to compare the fingerprint of the first set with the fingerprints of all the n sets of the second form in time $O(1)$.

There are two reasons why this sample value based equivalence checking may err.

1. The fingerprinting may err.
2. The set of sample values $\{J_1, \ldots, J_{\#J}\}$ may not contain any J_k such that $E'(J_k) \neq E_i(J_k)$ even though E' and E_i are non-equivalent.

In Subsection 4.2.2, we have seen that the first cause of failure is extremely unlikely. The probability of the second cause may be reduced by a careful choice of sample values.

Assume that each synthesized expression is to be used at position *Pos* in the program to be transformed. The variables $X_1, \ldots, X_m$, that may occur free in synthesized expressions, depend on *Pos*. We can use each value of the tuple $(X_1, \ldots, X_m)$, that arises during execution of the program for all sample inputs in the specification, as a J_k. Additionally, we may add a few random values to the set of sample values. Then, we need a probability distribution on the set of values given by the type of $(X_1, \ldots, X_m)$. An example of such a distribution is to say that each sample value size s not exceeding some maximum $s_{\max}$ is equally likely and that all values of size s also are equally likely.

However, the current implementation does not employ any of the equivalence checking methods discussed in this subsection. The reason is the following general problem with equivalence checking of synthesized expressions that are to appear in an "unfinished" program. Since the program is unfinished, the function f, which is the function to be inferred, and the `let`-functions defined in the program may have incomplete definitions. Since one or more of these functions usually may occur in a synthesized expression E, the values $E(J_1), \ldots, E(J_{\#J})$ may be quite different for the final program and the current, unfinished program. The most common case is that there is a k such that $E'(J_k) = E_i(J_k)$ for

```
fun v_count nil = 0
  | v_count(Sub::Subs) =
      if Sub is canonical then
        v_count Subs
      else
        1 + v_count( Subs with all occurrences of Sub replaced
              by the canonical form of Sub )
```

Figure 5.7: An operational definition of the number of violations.

the unfinished program whereas $E'(J_k) \neq E_i(J_k)$ for the final program. This is illustrated by the following example, where $f =$ `sort`.

Example. Consider the following unfinished program, which appeared in Subsection 3.5.3 where we showed why this program is better than the identity function.

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

No matter how we choose the values $J_1, \ldots, J_{\#J}$, a call of the form `sort` A would be considered to be equivalent to A for all expressions A. □

In spite of this problem, there are many situations where equivalence checking based on sample values would be useful, but it is difficult for ADATE to recognize these situations.

A rather straightforward heuristic is to allow a controlled number of violations of the non-equivalence requirement as follows. We say that an expression is canonical if and only if it is the first synthesized expression in its equivalence class, i.e., the least class element according to the total order $<_{\text{synt}}$. Assume that `Subs` is a bottom-up listing of the subexpressions of a synthesized expression E. The number of violations in E is `v_count Subs`, where `v_count` may be defined as shown in Figure 5.7. Note that the substitution in the last call to `v_count` also includes subexpressions of the subexpressions in `Subs`. The purpose of this substitution is to count a violation only once. Also note that it is easy to count the number of violations produced thus far during the bottom-up synthesis of an expression and to cut off when this number becomes too big. The number of expressions of size s that contain at most v violations often grows rapidly with v.

Example. Assume that expressions are synthesized using the components { `Xs, g1, g2, g3, g4` }, that the type of `Xs` is `int` and that the type of each gi is `int -> int`. Also assume `g1=g3` and `g2=g4`. The number of expressions of size s with 0 violations is 2^{s-1} whereas the number of expressions with at most 1 violation is $2^{s-1} + (s-1) \cdot 2^{s-1} = s \cdot 2^{s-1}$. The number of expressions with at most v violations is $2^{s-1} \cdot \sum_{i=0}^{v} \binom{s-1}{i}$. □

An empirical observation is that the best synthesized expressions normally contain no more than a few violations, which means that the number of expressions, that need to be synthesized and examined, can be greatly reduced by focussing on expressions with few violations. Recall that N is the number of expressions to be synthesized. For example, the synthesis algorithm could strive to produce $0.25N$ expressions with 0 violations, $0.15N$ expressions with exactly 1 violation and $0.1N$ expressions with exactly 2, 3, 4, 5, 6 or 7 violations.

Another problem with equivalence checking is that it requires too much time per synthesized expression. For example, we expect that the average synthesis time per expression would increase at least 10 times for the current implementation of ADATE if we employed sample value based equivalence checking. Since synthesized expressions typically are very small, the reduced branching factor would not compensate this increase, at least not for the inferences tried so far. However, we may choose to include such equivalence checking in future versions of ADATE, particularly if the ability of searching large expression spaces is to be improved.

5.3.2 Randomization

The goal with randomized expression synthesis is to pick only one or a few expressions in each equivalence class. For example, assume that N expressions are to be chosen from an expression space with total cardinality T. Let D be the cardinality of the equivalence class that contains desirable expressions. Randomized expression synthesis is primarily useful for big D. Assume that we choose expressions according to a uniform distribution on the expression space. For each random choice, the probability of choosing a desirable expression is D/T. If we make N choices, the probability that we choose at least one desirable expression is one minus the probability that we do not choose any desirable expression i.e.,

$$1 - (1 - \frac{D}{T})^N$$

. Let us define k so that $N = kT/D$ and assume that T/D is large, which means that the probability is

$$1 - (1 - \frac{D}{T})^{k\frac{T}{D}} = 1 - ((1 - \frac{1}{\frac{T}{D}})^{\frac{T}{D}})^k \approx 1 - e^{-k}.$$

For example, if we choose N so that $N = 5T/D$, we will try at least one desirable expression with a probability greater than 99.3%.

Assume that a finished program is built from n different synthesized expressions and that we need to make n sequential choices of synthesized expressions to obtain this program. Also assume that an appropriate choice of each one of these expressions is found by random expression synthesis with the probability $1 - e^{-k}$ and that the choices are so independent that the the probability of finding n appropriate choices is $(1 - e^{-k})^n$. If we want this overall probability to exceed a confidence limit μ, we have

$$e^{-k} < 1 - \mu^{\frac{1}{n}},$$

which gives

$$k > -\ln(1 - \mu^{\frac{1}{n}}).$$

Assume that n is large and that μ is close to 1, which means that $1 - \mu^{\frac{1}{n}}$ is close to 0. A first order Maclaurin series expansion gives

$$1 - \mu^{\frac{1}{n}} = -\frac{\ln \mu}{n} + O(\frac{1}{n^2}).$$

Ignoring $O(1/n^2)$ yields

$$k > -\ln(-\frac{\ln \mu}{n}) = \ln n - \ln(-\ln \mu).$$

For example, with $\mu = 0.95$, we have $k > \ln n + 2.97$. Even for very big programs i.e., large values of n, we can choose reasonably small k. For example, if the final program consists of one million synthesized expressions, we obtain 95% confidence with $k = 16.8$.

The ratio T/D indicates the hardness of an expression synthesis problem. In practice, this ratio varies widely.

Example. Let us add the `'a list` variable `Ys` to the components in the first example in Section 5.1 in order to obtain a more realistic expression synthesis problem. Thus, the components are `Xs`, `Ys`, `nil` and `@`. Let the expression space consist of all expressions of a size not exceeding $s_{\max}$. Assume that the function l is such that $l(E)$ is the preorder listing of the function and variable symbols in an expression E with all occurrences of `nil` and `@` removed e.g.

$$l(\texttt{Xs@(nil@Ys)}) = \texttt{[Xs,Ys]}.$$

The equivalence classes are given by the equivalence relation eq defined by

$$eq(E_1, E_2) = (l(E_1) = l(E_2)).$$

We will now examine the ratio

$$r(s_{\max}, L) = \frac{T(s_{\max})}{D(s_{\max}, L)},$$

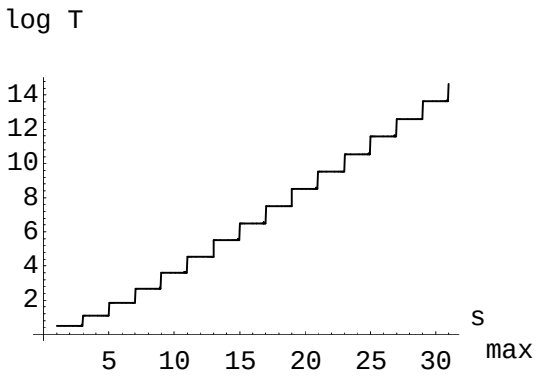


Figure 5.8: The total cardinality as a function of size.

where L is the length of the l-value of the class of desirable expressions.

It is easy to see that

$$T(s_{\max}) = \sum_{i=0}^{(s_{\max}-1)/2} \frac{3^{i+1}}{i+1} \left(\begin{array}{c} 2i \\ i \end{array} \right).$$

For a given expression size s such that the number of leaves, $(s+1)/2$, is greater than or equal to L, the number of combinations of the leaves of a desirable expression is $\left(\begin{array}{c} (s+1)/2 \\ L \end{array} \right)$. The number of desirable expressions with i internal nodes is

$$\left(\begin{array}{c} i+1 \\ L \end{array} \right) \frac{1}{i+1} \left(\begin{array}{c} 2i \\ i \end{array} \right),$$

which means that

$$D(s_{\max}, L) = \sum_{i=L-1}^{(s_{\max}-1)/2} \left(\begin{array}{c} i+1 \\ L \end{array} \right) \frac{1}{i+1} \left(\begin{array}{c} 2i \\ i \end{array} \right).$$

Figure 5.8 shows $\log_{10} T(s_{\max})$ for $1 \leq s_{\max} \leq 31$. Figure 5.9 shows $\log_{10} r(s_{\max}, L)$ for $1 \leq s_{\max} \leq 31$ with one curve for each L in $\{1, 2, \ldots, 15\}$. Thus, Figure 5.8 shows the number of expressions that need to be synthesized with exhaustive search whereas the curves in Figure 5.9 show the number of expressions generated by random search divided by a small constant. For example, for $s_{\max} = 30$, the exhaustive search would produce about 10^{14} expressions whereas we would

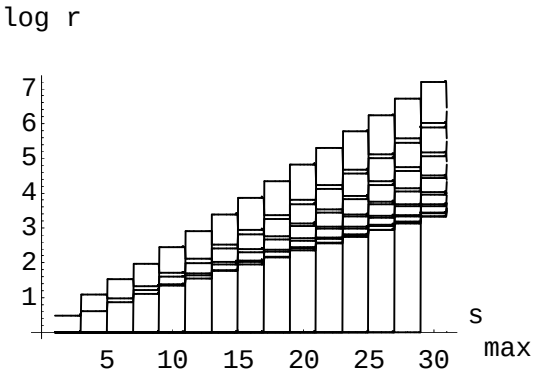


Figure 5.9: The hardness of random synthesis as a function of size.

expect the randomized search to produce between 10^4 and 10^8 expressions depending on L and k. Note that $D(s_{\max}, L)$ varies widely with L. □

A fundamental problem with randomized expression synthesis seems to be that we do not know D. However, using the simple strategy described below, this lack of knowledge causes the expected run time to be multiplied by no more than a small factor. We assume that both $r = T/D$ and T have approximately exponential growth i.e., $r \approx B_r^{s_{\max}}$ and $T \approx B_T^{s_{\max}}$ for constant branching factors B_r and B_T. Practical expression synthesis problems are so hard that we can assume $B_r \geq \sqrt{2}$ i.e., $\min(B_r) = \sqrt{2}$.

Our search strategy is as follows. Given N, i.e., the number of expressions to be synthesized, we choose $s_{\max}$ to

$$\lceil \log_{\min(B_r)} N \rceil = 2 \lceil \log_2 N \rceil.$$

We only synthesize expressions of sizes given by the set σ which is the set of all sizes s such that there exists expressions of size s and $1 \leq s \leq s_{\max}$. Let $\nu(s)$ be the number of expressions of size s that are to be synthesized. Obviously, we require $N = \sum_{s \in \sigma} \nu(s)$. Our strategy is to distribute N uniformly on σ i.e., choose $\nu(s)$ to $\lfloor N/\#\sigma \rfloor$ for each s in σ.

Example. If $N = 10^5$ and the components are `Xs`, `Ys`, `nil` and `C`, we obtain

$$\sigma = \{1, 3, 5, 7, \ldots, 2\lceil \log_2 10^5 \rceil - 1\} = \{1, 3, 5, 7, \ldots, 33\}$$

and

$$\nu(s) = \lfloor 10^5/17 \rfloor = 5882.$$

□

Even if we actually did know D and used this knowledge to determine $\nu(s)$, we would certainly have $\nu(s) \leq N$ for all s. Thus, no $\nu(s)$ value decreases by more than a factor $1/(2\lceil \log_2 N \rceil)$ just because we do not know D. Note that $2\lceil \log_2 N \rceil$ is an upper bound. The average performance of randomized search is better, but difficult to determine.

Recall that the confidence interval of randomized search is determined by the factor k, which we defined so that $N = kr$. The uniform distribution strategy gives a k-value that varies with size. We "over-sample" small sizes, which gives a high k-value and high confidence. The space of expressions of a size close to $s_{\max}$ is sampled more sparsely, which gives low k and low confidence. We let k be determined implicitly in this manner and do not try to compute k. Here is an example that illustrates "over-sampling".

Example. With $N = 10^5$ and the components `Xs`, `Ys`, `nil` and `0`, we obtained $\nu(1) = 5882$. Assume that exactly one of the three expressions of size 1 is desirable, which means that $k = 5882/3$ for size 1. The probability of finding a desirable expression of size 1 is

$$1 - (\frac{2}{3})^{5882} \approx 1 - 10^{-1036}.$$

□

Intuitively, over-sampling provides extra insurance against extremely bad luck. One may gain a factor of two or three by trying to avoid over-sampling. Since the gain is small, we do not discuss how to avoid over-sampling.

The algorithm for randomized expression synthesis chooses $\nu(s)$ random expressions according to a uniform distribution on the space of all expressions of a given size s. One way of choosing a random expression is as follows.

1. Let $t(s)$ be the cardinality of the space of expressions of size s, i.e., $t(s) = T(s) - T(s-1)$.
2. Choose a random number η according to a uniform distribution on $\{1, 2, \ldots, t(s)\}$.
3. Convert η to the corresponding expression according to some suitable enumeration of the space of expressions of size s.

The most difficult problem is how to enumerate the expression space. Here is an example of randomized synthesis with top-down enumeration of the expression space.

Example. We want to synthesize expressions of type `(int*int) list` using the following components.

```
X : int
Y : int
+ : int*int -> int
Xs : (int*int) list
nil : 'a list
```

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

Let $t(Ty, s)$ be the number of expressions of type Ty and size s. We now choose to define the size of an expression to be the number of occurrences of variables and functions, including all occurrences of implicit tuple constructors. Thus, the size of `Xs@Xs` is 4, not 3. The components above give the following definition of t.

$t(\texttt{int}, 1) = 2$.
$t(\texttt{int}, s) = t(\texttt{int*int}, s-1)$ when $s \geq 4$.
$t(\texttt{int*int}, s) = \sum_{i=1}^{s-2} t(\texttt{int}, i) \cdot t(\texttt{int}, s-1-i)$ when $s \geq 3$.
$t(\texttt{(int*int) list}, 1) = 2$.
$t(\texttt{(int*int) list}, s) = \sum_{i=1}^{s-3} t(\texttt{int*int}, i) \cdot t(\texttt{(int*int) list}, s-2-i) + \sum_{i=1}^{s-3} t(\texttt{(int*int) list}, i) \cdot t(\texttt{(int*int) list}, s-2-i)$ when $s \geq 4$.

If none of these equations apply, $t(Ty, s)$ is 0. Note that each summation corresponds to a component. For example, the first summation in the last equation corresponds to `::` whereas the last summation corresponds to `@`.

Before starting random synthesis, we tabulate t in order to allow quick computation of $t(Ty, s)$. The table for $s \leq 22$ is shown in Figure 5.10. Using t, it is easy to convert an order number η to the corresponding expression. We will now discuss the definition of a function `nat_to_exp` such that the value of

$$\texttt{nat\_to\_exp}(Ty, s, \eta)$$

is the expression of type Ty and size s with order number η. In order to simplify the definition, we let 0 be the first order number. Assume that Ty is `(int*int) list`. If $s = 1$, we must have $\eta = 0$ or $\eta = 1$, where we can say that $\eta = 0$ corresponds to `Xs` and that $\eta = 1$ corresponds to `nil`. If $s \geq 4$, the root is `::` if

$$\eta < \sum_{i=1}^{s-3} t(\texttt{int*int}, i) \cdot t(\texttt{(int*int) list}, s-2-i)$$

and `@` otherwise. Assume that the root is `@`. Let η' be

$$\eta - \sum_{i=1}^{s-3} t(\texttt{int*int}, i) \cdot t(\texttt{(int*int) list}, s-2-i).$$

Then, we compute the greatest size s' such that

$$\eta' - \sum_{i=1}^{s'-1} t(\texttt{(int*int) list}, i) \cdot t(\texttt{(int*int) list}, s-2-i) \geq 0$$

s	$t(\texttt{int}, s)$	$t(\texttt{int*int}, s)$	$t(\texttt{(int*int) list}, s)$
1	2	0	2
2	0	0	0
3	0	4	0
4	4	0	4
5	0	0	0
6	0	16	8
7	16	0	16
8	0	0	0
9	0	80	80
10	80	0	80
11	0	0	32
12	0	448	672
13	448	0	448
14	0	0	640
15	0	2688	5376
16	2688	0	2816
17	0	0	8704
18	0	16896	42240
19	16896	0	20992
20	0	0	100352
21	0	109824	329984
22	109824	0	190208

Figure 5.10: Expression space cardinality as a function of size and type.

and set η'' to this difference. Let η_l and η_r be the order numbers of the left and the right subtree respectively. If we write η'' as

$$\eta_l \cdot t(\texttt{(int*int) list}, s-2-s') + \eta_r\,,$$

we get

$$\eta_l = \lfloor \eta''/t(\texttt{(int*int) list}, s-2-s') \rfloor$$

and

$$\eta_r = \eta'' \bmod t(\texttt{(int*int) list}, s-2-s').$$

The left subtree is

$$\texttt{nat\_to\_exp}(\texttt{(int*int) list}, s', \eta_l).$$

The right subtree is

$$\texttt{nat\_to\_exp}(\texttt{(int*int) list}, s-2-s', \eta_r).$$

□

We have only discussed the definition of `nat_to_exp` for the components in the example above. It is somewhat tedious, but not too difficult, to give an implementation of `nat_to_exp` that is parameterized by the components. The current version of ADATE does not employ random synthesis since the enumerative heuristic search presented in Section 5.2 suffice for the small expression synthesis problems that we have encountered in practice.

Chapter 6

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.

6.1 Compound Transformation Forms

The choice of an atomic transformation t_i, $i \geq 2$, 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. Consider the last compound transformation in the inference of `sort` presented in Subsection 4.1.2. 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 $t_{\#t}$ = R and $t_i \neq$ R for each $i < \#t$. Each transformation t_i, $i \geq 2$, 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 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 the 22 forms shown in Figure 6.1. 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 8 coupling rules above were 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. Assume that each of the last three transformations needs to choose exactly one subexpression. Without coupling, there would be about $N^3/2$ such choices whereas there are about $N^3_{RHS}/2$ choices with coupling, which means that coupling is particularly important for small N_{RHS}/N ratios. The denominator 2 is used since the first

R
EMB R
REQ ABSTR REQ R
REQ ABSTR REQ REQ R
REQ ABSTR EMB REQ R
REQ ABSTR EMB REQ REQ R
REQ ABSTR EMB R
REQ ABSTR R
REQ R
CASE-DIST ABSTR REQ R
CASE-DIST ABSTR REQ REQ R
CASE-DIST ABSTR EMB REQ R
CASE-DIST ABSTR EMB REQ REQ R
CASE-DIST ABSTR EMB R
CASE-DIST ABSTR R
CASE-DIST R
ABSTR REQ R
ABSTR REQ REQ R
ABSTR EMB REQ R
ABSTR EMB REQ REQ R
ABSTR EMB R
ABSTR R

Figure 6.1: All forms.

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

6.2 Syntactic Checking and Pruning of Programs

In addition to the checks performed during the synthesis of an atomic transformation, there are heuristic checks that depend on the preceding atomic transformations. We have chosen to apply these checks during the synthesis of compound transformations. The current implementation of ADATE employs the following two checks.

1. Static `case` checking.
2. Pattern occurrence checking.

We will now discuss these two checks.

6.2.1 Static `case` Checking

Consider a function defintion of the form

$$\texttt{fun g}(V_1, \ldots, V_n)\texttt{=}RHS.$$

A subexpression of RHS that does not depend on $V_1, \ldots, V_n$ is said to be static. The static `case` check does not allow static `case`-analyzed expressions since the outcome of a static `case`-analysis is the same for each recursive call to `g`.

Example. Consider the expression `X2<X1` that is `case`-analyzed in both of the following two programs.

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

fun sort Xs =
  case Xs of nil => Xs
  | X1::Xs1 =>
  let fun g V1 =
```

```
fun static_case_check(D as {func,pat,exp,...} : dec ) :  bool =
  scc(exp,vars_in_pat pat)
and scc(E,Vars) =
  case E of
    app_exp{args,...} =>  forall(fn Arg => scc(Arg,Vars), args)
  | case_exp{exp,rules,...} =>
      not(null(intersection(Vars,zero_arity_apps exp))) andalso
      scc(exp,Vars) andalso
      forall( fn{pat,exp} => scc(exp,vars_in_pat pat @ Vars), rules )
  | let_exp{dec_list,exp,...} =>
      scc(exp,Vars) andalso
      forall(static_case_check,dec_list)
```

Figure 6.2: The implementation of static `case` checking.

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

The occurrence of `X2<X1` in the first program is static whereas the occurrence in the last program is not static. □

The static `case` check is quite simple to implement as shown in Figure 6.2. The auxiliary function `zero_arity_apps` returns the leaves in an expression.

To avoid too much pruning, the static `case` check is only used when the coupling rule ABSTR → R is applied. Just before the R, ADATE requires that the current version `D` of the `let`-definition introduced by the ABSTR is such that

`static_case_check D = true`.

6.2.2 Pattern Occurrence Checking

The goal with pattern occurrence checking is to eliminate some futile REQs. A REQ, that preserves semantics no matter what context it appears in, is rather meaningless. For example, since addition of integers is commutative, it is rather futile to replace `X+Y` with `Y+X`. Subsection 4.1.2 showed some REQs that are not futile. For example, the REQ that replaces an occurrence of the `int list` variable `Xs` with `X1::nil`, where `X1 = hd Xs`, makes sense since the values of the occurrence of `Xs` might not remain singletons when the program is further transformed.

```
fun remove_as Pat = exp_map(
  fn as_exp{var,pat,exp_info} =>
       app_exp{func=var,args=nil,exp_info=exp_info}
   | Sub => Sub,
  Pat )

fun prohibited_exps Pat =
  let val As_subs =
    exp_filter(fn as_exp{...} => true | _ => false, Pat)
  in
    map(fn as_exp{pat,...} => remove_as pat, As_subs)
  end
```

Figure 6.3: Two auxiliary functions for pattern occurrence checking.

Consider a `case`-expression of the following form.

$$\texttt{case } A \texttt{ of } Match_1 \texttt{ => } E_1 \texttt{ | } \ldots \texttt{ | } Match_n \texttt{ => } E_n.$$

Inside E_i, it is obvious that A and $Match_i$ are equivalent. Pattern occurrence checking only allows the smallest of the expressions A and $Match_i$ to occur in an expression introduced by a REQ inside E_i. Note that it is necessary to convert $Match_i$ to an ordinary expression by removing `as`-patterns from $Match_i$.

In order to implement pattern occurrence checking, we need the two help functions in Figure 6.3. Recall that `exp_map` and `exp_filter` are analogous to the functions `map` and `filter` on lists. It is easy to see that `remove_as Pat` is `Pat` with all subpatterns of the form V `as` Sub replaced by V. Since the subpattern Sub has a greater size than the variable V in an `as`-pattern V `as` Sub, we say that Sub is prohibited. The function `prohibited_exps` finds all such prohibited subpatterns in a pattern.

Assume that `fun f` Pat `=` RHS is the program being transformed and that `Top_pos` is the position of a REQ in RHS. Pattern occurrence checking is performed with the call

`pattern_occurrence_check(` RHS`, prohibited_exps` Pat`, Top_pos )`

for each REQ position `Top_pos`. Figure 6.4 shows the definition of `pattern_occurrence_check`. Like the static `case` check, the pattern occurrence check is applied just before the R in the coupling rule ABSTR → R.

```
fun pattern_occurrence_check( E : exp, Prohibited : exp list,
      Pos : pos ) : bool =
  let
    fun g _ = nil
    fun f(Pats,E_sub,P::_) =
      case E_sub of
        case_exp{exp,rules,...} =>
          if P=0 then
            Pats
          else
            let val Pat = #pat(nth(rules,P-1))
                val Stripped_pat = remove_as Pat
            in
              (if exp_size exp < exp_size Stripped_pat then
                 Stripped_pat
               else
                 exp) ::
              prohibited_exps Pat @ Pats
            end
      | _ => Pats
    val Pats = pos_fold(f,g,Pos,E)
    val E_sub = pos_to_sub(E,Pos)
  in
    forall( fn Pat => null(exp_filter(fn Sub => Sub=Pat, E_sub)),
      Prohibited@Pats )
  end
```

Figure 6.4: The implementation of pattern occurrence checking.

6.3 Using the Forms to Produce Programs

Given a current program P and a form $t_1 \ldots t_{\#t}$, P is the input of atomic transformation t_1. The output program from atomic transformation t_i is the input of atomic transformation t_{i+1} for each i in $\{1, \ldots, \#t - 1\}$. The output program from $t_{\#t}$ is sent to the "population control" algorithm described in Chapter 7.

6.3.1 Cost Limit Computation for Forms

Assuming that the cost limit before each t_i is $\texttt{Cost\_limit}_i$ and that each t_i has cost C_i, we have $\texttt{Cost\_limit}_{i+1} = \texttt{Cost\_limit}_i / C_i$. If there was no pruning due to static `case` and pattern occurrence checking, it would be possible to use the same $\texttt{Cost\_limit}_1$ for all forms. Let W_{tot} be the work goal for all forms taken together. Let N_{forms} be the number of forms i.e., 22. Since pruning is employed, we have chosen a work goal of $W_{\text{tot}}/N_{\text{forms}}$ programs for each form. Thus, all forms are supposed to produce equally many programs.

The form cost limit $\texttt{Cost\_limit}_1$ is deepened iteratively with a branching factor β. The choice of branching factor is discussed in Chapter 7. The first iteration has $\texttt{Cost\_limit}_1 = 100$. This means that iteration number i has $\texttt{Cost\_limit}_1 = 100 \cdot \beta^i$, where it is assumed that the first iteration has number 0. The first iteration is run for all forms. When more than $W_{\text{tot}}/N_{\text{forms}}$ programs have been produced using a specific form during some subsequent iteration, the form is not used any more to produce children of the current program P.

Recall that $t_{\#t}$ always is R and that there is no $i < \#t$ such that t_i is R. Therefore, the expression synthesis algorithm does not need to normalize the costs of synthesized expressions, which means that it is reasonable to choose the actual cost of synthesized expression number i to i. As discussed in Section 4.2, atomic transformation algorithms other than the R algorithm, that employ expression synthesis, contain their own normalization methods and do not need normalized costs of synthesized expressions.

Since we choose the cost of synthesized expression number i to i, a form cost limit $\texttt{Cost\_limit}_1$ would normally lead to the production of $\texttt{Cost\_limit}_1$ children if no pruning is used. With static `case` and pattern occurrence pruning, there may be a production of only $0.1\texttt{Cost\_limit}_1$ children. However, there are combinations of current programs and forms such that the production is even lower. For example, the form EMB R cannot be used to produce any children at all if the current program P does not contain any `let`-function. Therefore, the maximum $\texttt{Cost\_limit}_1$ to be used during the iterative-deepening is chosen to $20 W_{\text{tot}}/N_{\text{forms}}$.

6.3.2 Computation of REQ, EMB and CASE-DIST Cost Limits

We only discuss the computation of REQ cost limits below, but EMB and CASE-DIST cost limits are computed in exactly the same way. Therefore, all occurrences of REQ in the following discussion may be replaced with EMB or CASE-DIST as appropriate.

Recall that a `REQ_cost_limit` determines how much work that should be spent on finding REQ transformations. Since the cost of synthesized expression number i is i, the amount of work that will be spent on finding REQs equals the `REQ_cost_limit`. If the `REQ_cost_limit` is too small, we run the risk of missing REQs with good pe_{REQ} values. If the `REQ_cost_limit` is too large, REQ transformations will require a too large fraction of the overall execution time.

Given a form $t_1 \ldots t_i \ldots t_{\#t}$, where t_i is REQ, let $w(\texttt{Cost\_limit}_i)$ be the expected work, excluding the work spent on finding REQs, that will be done using the remaining part $t_i \ldots t_{\#t}$ of the form and cost limit $\texttt{Cost\_limit}_i$. If $t_1 \ldots t_{\#t}$ contains only one REQ sequence, we choose `REQ_cost_limit` for t_i to $w(\texttt{Cost\_limit}_i)$. Otherwise, we choose it to $0.7w(\texttt{Cost\_limit}_i)$. The reason that the latter `REQ_cost_limit` is slightly lower is that we do not want the total work spent on finding REQs for a given form to be too much greater than the total other work for the form.

The remaining question is how to compute the expected work $w(\texttt{Cost\_limit}_i)$. Let σ_i be the sum of the $\texttt{Cost\_limit}_i$ cost limits that have been used during previous employments of the form part $t_i \ldots t_{\#t}$. Let σ'_i be the corresponding sum of the non-REQ work that actually was done. If $\sigma'_i < 100$, we assume that it is too small to be used for statistical forecasting and choose $w(\texttt{Cost\_limit}_i)$ to a default value of $0.3\texttt{Cost\_limit}_i$. Otherwise, we choose

$$w(\texttt{Cost\_limit}_i) = \max(\frac{\sigma'_i}{\sigma_i}, \frac{1}{20})\texttt{Cost\_limit}_i,$$

where the last argument of the max function ensures that a reasonable amount of work will be spent on finding REQs even if σ'_i/σ_i is quite small i.e., if pruning has been hefty. Separate σ'_i and σ_i sums are maintained for each t_i that is a REQ in each form. When the compound transformation algorithm is restarted with a new current program P, all such sums are initialized to 0 in order to adapt them individually to each program P. Also note that the iterative-deepening of the form cost limit $\texttt{Cost\_limit}_1$ contributes to good estimation of $w(\texttt{Cost\_limit}_i)$.

6.3.3 Match Error Handling

Assume that program P_{i+1} is an output from an atomic transformation t_i in some form $t_1 \ldots t_{\#t}$, where $t_{\#t}$ always is R. The expression that is inserted into $P_{\#t}$ by $t_{\#t}$ may contain special *Not_activated* constants as described in Subsection 5.2.4. These constants, however, may become activated in $P_{\#t+1}$.

Example. Consider the synthesis of the list concatenation function `@` using the single sample input

```
( [1,2,3,4,5], [6,7,8,9] ).
```

Assume that $\#t = 1$, $t_1 = \mathrm{R}$ and that P_1 is `fun @(Xs,Ys) = ?`. The best synthesized expression produced by `synt_n` is

```
case Xs of nil => Not_activated | X1::Xs1 => X1::@(Xs1,Ys).
```

This expression is produced from the following unfinished `case`-expression.

```
case Xs of nil => Unknown1 | X1::Xs1 => Unknown2.
```

The program used for activation checking is

```
fun @(Xs,Ys) =
  ( case Xs of nil => Unknown1 | X1::Xs1 => Unknown2; ?  )
```

$Unknown_1$ will be replaced by *Not_activated* whereas $Unknown_2$ will be replaced by synthesized expressions, for example `X1::@(Xs1,Ys)`. The resulting program is

```
fun @(Xs,Ys) =
  case Xs of nil => Not_activated | X1::Xs1 => X1::@(Xs1,Ys)
```

Due to recursive calls, the *Not_activated* constant will be activated during the execution of this program. This is called a match error. □.

When a match error has been detected during the execution of a program, ADATE tries to fix the error by replacing the occurrence of *Not_activated*, that caused the error, with synthesized expressions. Frequently, these expressions are supposed to handle a recursive "base case". The current implementation is not able to handle more than one match error at a time.

Assume that M match errors and W programs have been produced so far. The cost limit employed when replacing a *Not_activated* constant is chosen to $0.1W/(M+100)$ in order to avoid spending too much time on such replacements. For example, if we on average have one match error per five hundred programs, the cost limit will be about fifty for large W and M. Both W and M are initialized to 0 when the compound transformation algorithm is restarted with a new current program P.

Since the cost limit depends only on the match error ratio $W/(M + 100)$, this scheme for handling match errors may not be sufficiently general. It was primarily designed to handle quite small recursive base cases. However, there is no general rule saying that recursive base cases have to be small, but they do tend to be small in practice. A more general scheme would be to make the cost limit directly dependent on $\texttt{Cost\_limit}_{\#t}$ and the order number of the synthesized expression inserted by the atomic transformation $t_{\#t}$, which always is R.

Chapter 7

The Overall Search for Programs

The algorithm for overall search maintains a population of programs. It repeatedly selects a program from the population, sends it to the algorithm for synthesis of compound transformations and receives transformed children programs from this algorithm. We say that such a selected program is expanded.

7.1 Population Structure

Initially, the population consists of a single copy of the initial program i.e., a program of the form `fun f` *Pat* `= ?`, where the tuple pattern *Pat* is automatically constructed by ADATE using the domain type of `f`.

The population is partitioned into classes such that all programs in a class contain the same number of `case`-expressions. Each class is partitioned into subclasses such that all programs in a subclass contain the same number of `let`-expressions. The purpose of this partitioning is to maintain diversity by ensuring that programs with low `case` or `let` counts are not "killed" by superior programs with higher `case` or `let` counts.

Each subclass contains three programs. Program number i in subclass number l of class number c is the best program found so far according to program evaluation function $\texttt{pe}_i$ that contains exactly c `case`-expressions and l `let`-expressions.

7.2 Selection and Insertion of Programs

7.2.1 Selection

Let (c_P, l_P) be the `case` and `let` counts of a program P in the population. Let $\Pi(P)$ be the set of all programs in the population with (c, l) values such that

$$c < c_P \vee (c = c_P \wedge l < l_P).$$

The program to be expanded next is chosen to a program P with a minimum (c_P, l_P) value such that P is better than all programs in $\Pi(P)$ according to at least one program evaluation function pe_i. Of course, P is expanded only once using the same W_{TOT} work goal.

Assume that c_{best_i} is the `case` count of the best program found so far as judged by $\texttt{pe}_i$. 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 $\texttt{pe}_2$ 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.

7.2.2 Insertion

Let Q be a program that is a candidate for insertion into the population i.e., that has been received from the compound transformation algorithm. First, we apply a quick rejection test to Q. This test is meant to quickly determine if Q is good enough to be worth further and more time consuming processing. Q fails the quick rejection test if and only if it is worse than all programs in the subclass (c_Q, l_Q) according to the program evaluation function pe_{REQ}.

If Q passes the test, it is subjected to dead code elimination and elimination of redundant definitions of `let`-functions, which yields a program Q'. A `let`-function `g` is considered to be redundant if and only if it is non-recursive and if unfolding of all calls to `g` and removal of the definition of `g` does not increase the syntactic complexity of the program.

We choose to discard Q' if it is not better than all its proper ancestors according to at least one program evaluation function pe_i. The concept proper ancestor is defined as follows. If Q is produced from P using a compound transformation, P is the parent of Q. A proper ancestor of Q is either P or one of P's proper ancestors.

The next question is if any of the three programs in the subclass $(c_{Q'}, l_{Q'})$ should be replaced by Q'. If there is any i such that Q' is better than program number i in the subclass according to program evaluation function pe_i, ADATE replaces program number i with Q'.

7.3 Iterative-Deepening Search

The work goal W_{tot} is deepened iteratively using a branching factor α. Iteration number 0 has $W_{\text{tot}} = 10000$. Iteration number i has $W_{\text{tot}} = 10000\alpha^i$. During an iteration, programs are selected from the population and inserted into the population as described above. An iteration terminates when no program in the population is eligible for expansion.

Recall that the algorithm for synthesis of compound transformations uses iterative-deepening of the initial cost limit $\texttt{Cost\_limit}_1$. An "overall" iteration with a given work goal W_{tot} will be called a primary iteration whereas an iteration made by the algorithm for synthesis of compound transformations will be called a secondary iteration. Note that many secondary iterations are made during one primary iteration.

We will now discuss the choice of the primary branching factor α and the secondary branching factor β. We assume that there is a minimum cost C such that a desirable program certainly will be found if W_{tot} is so large that secondary iterations with $\texttt{Cost\_limit}_1 \geq C$ are run completely. Remember that a secondary iteration using a given form is terminated i.e., not run completely, when the number of programs that have been produced with the form exceeds $W_{\text{tot}}/N_{\text{forms}}$. Assume that there is a critical parent program P such that sufficiently good children can be found only if $\texttt{Cost\_limit}_1 \geq C$. Intuitively, P is the bottle-neck of a genealogical path that leads to a desirable program. The critical secondary iterations are the ones that expand P.

Let n be the number of the last critical secondary iteration, which is the one that produces a sufficiently good child. To simplify the following discussion, we assume that both the first primary iteration and the first secondary iteration are run with an initial limit of 1 instead of 10000 and 100 respectively. This means that secondary iteration number i is assumed to have $\texttt{Cost\_limit}_1 = \beta^i$. Obviously, n is the smallest value such that $\beta^n \geq C$, which implies $n = \lceil \log_\beta C \rceil$.

The degree of pruning made by the algorithm for synthesis of compound transformations depends on the current program, the form and the $\texttt{Cost\_limit}_1$ value of the current secondary iteration. For each form number i, we assume that the ratio between the number of programs synthesized using form i and $\texttt{Cost\_limit}_1$ rather rapidly approaches some limit γ_i when $\texttt{Cost\_limit}_1$ grows i.e., that a constant fraction $1 - \gamma_i$ of the programs are pruned for large $\texttt{Cost\_limit}_1$ values. One could relax this assumption and for example study the effects of oscillating ratios. However, based on limited experiments, we feel that such effects are negligible and do not study them here.

Since the last iterations produce many more programs than the first, we can assume that the number of programs produced by the n first secondary iterations using form i is

$$\gamma_i\beta^0 + \gamma_i\beta^1 + \ldots + \gamma_i\beta^{n-1} = \gamma_i\frac{\beta^n - 1}{\beta - 1}.$$

Let us now for a moment assume that no pruning is used and that the first secondary iteration is run with cost limit γ_i. It is easy to see that this situation is equivalent to using pruning and limit 1 for the first secondary iteration. Since the expected overall number of produced programs is essentially the same even if the initial cost limit for a given form is any reasonable number, say between 10^{-3} and 10^3, we can ignore pruning altogether. In the following discussion, we therefore assume $\gamma_i = 1$ for all i.

Let m be the number of the first primary iteration that may be good enough i.e., with

$$\alpha^m > \frac{\beta^n - 1}{\beta - 1}.$$

Let m' be the number of the first primary iteration that is certain to be good enough i.e., with

$$\alpha^{m'} \geq \frac{\beta^{n+1} - 1}{\beta - 1}.$$

Primary iteration number m is such that the cost interval $]\beta^{n-1}, \beta^n]$ may be only partly covered whereas this interval will be fully covered during primary iteration number m'. It is easy to see that

$$m \approx \lceil \log_\alpha \frac{\beta^n - 1}{\beta - 1} \rceil$$

and

$$m' \approx \lceil \log_\alpha \frac{\beta^{n+1} - 1}{\beta - 1} \rceil.$$

Let p_i be the probability that primary iteration number i is the first one that produces a sufficiently good child of P. The total work spent on expanding P during iterations number $0, 1, \ldots, i$ is

$$\frac{\alpha^{i+1} - 1}{\alpha - 1}.$$

The expected total work $W_P(C, \alpha, \beta)$ is

$$\sum_{i=m}^{m'} p_i \frac{\alpha^{i+1} - 1}{\alpha - 1}.$$

The children programs produced by primary iteration number j are a superset of the children programs produced by primary iteration number $j-1$. This means that the probability q_{i-1} that a desirable program is not found during primary iterations number $0, 1, \ldots, i-1$ equals the probability that a desirable program is not found during primary iteration number $i-1$. Assume that primary iteration number i is about to be started. The probability q_{i-1} depends on how large a

part of the interval $]\beta^{n-1}, \beta^n]$ that has been covered so far. Assuming that each program in the interval is equally likely to be covered, we have

$$q_{i-1} = 1 - \frac{\text{Cardinality of part covered so far}}{\text{Cardinality of the entire interval}},$$

which equals

$$1 - \frac{\max(0, \min(\beta^n, \alpha^{i-1} - \frac{\beta^n - 1}{\beta - 1}))\frac{\beta - 1}{\beta}}{\beta^n - \beta^{n-1}} = 1 - \frac{\max(0, \min(\beta^n, \alpha^{i-1} - \frac{\beta^n - 1}{\beta - 1}))}{\beta^n}.$$

Let r_i be the probability that iteration number i discovers a desirable program even though iterations number $0, 1, \ldots, i-1$ did not discover any desirable program. Note that $p_i = q_{i-1} r_i$, where we have

$$r_i = \frac{\text{Cardinality of part only covered by iteration number } i}{\text{Cardinality of part not covered by iteration number } i - 1},$$

which is

$$\frac{\max(0, \min(\beta^n, \alpha^i - \frac{\beta^n - 1}{\beta - 1}))\frac{\beta - 1}{\beta} - \max(0, \min(\beta^n, \alpha^{i-1} - \frac{\beta^n - 1}{\beta - 1}))\frac{\beta - 1}{\beta}}{\beta^n - \beta^{n-1} - \max(0, \min(\beta^n, \alpha^{i-1} - \frac{\beta^n - 1}{\beta - 1}))\frac{\beta - 1}{\beta}} =$$

$$\frac{\max(0, \min(\beta^n, \alpha^i - \frac{\beta^n - 1}{\beta - 1})) - \max(0, \min(\beta^n, \alpha^{i-1} - \frac{\beta^n - 1}{\beta - 1}))}{\beta^n - \max(0, \min(\beta^n, \alpha^{i-1} - \frac{\beta^n - 1}{\beta - 1}))}.$$

We want to determine α and β so that the expected ratio $W_P(C, \alpha, \beta)/C$ is minimized. Therefore, we need to know the distribution of the random variable C i.e., the minimum "bottle-neck" cost. We assume that computers are so slow that is unreasonable to have $C > 10^8$ and that they are so fast that there is no need to worry about $C < 10^5$. These computing speed assumptions motivated choosing

$$\Pr(C = X) = \frac{1}{K(X + 10^5)},$$

where K is a normalizing constant that equals

$$\sum_{X=1}^{10^8} \frac{1}{X + 10^5} \approx 6.90875.$$

Given α and β, the expected $W_P(C, \alpha, \beta)/C$ ratio is

$$E(\alpha, \beta) = \sum_{X=1}^{10^8} \Pr(C = X) \frac{W_P(X, \alpha, \beta)}{X}.$$

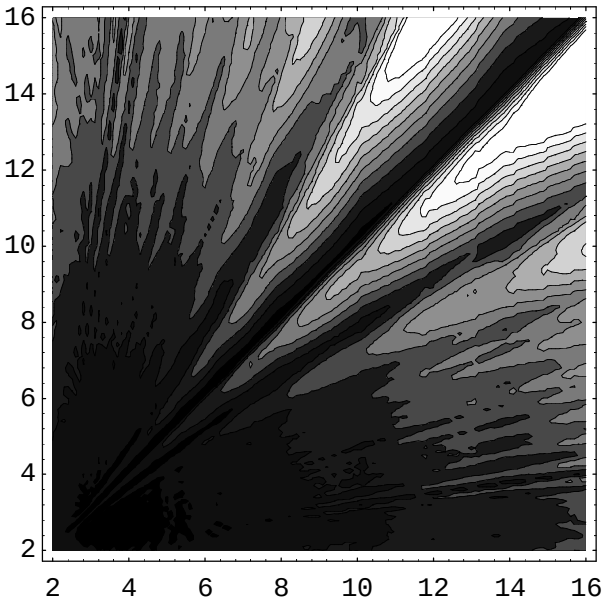


Figure 7.1: A coarse map of $E(\alpha, \beta)$.

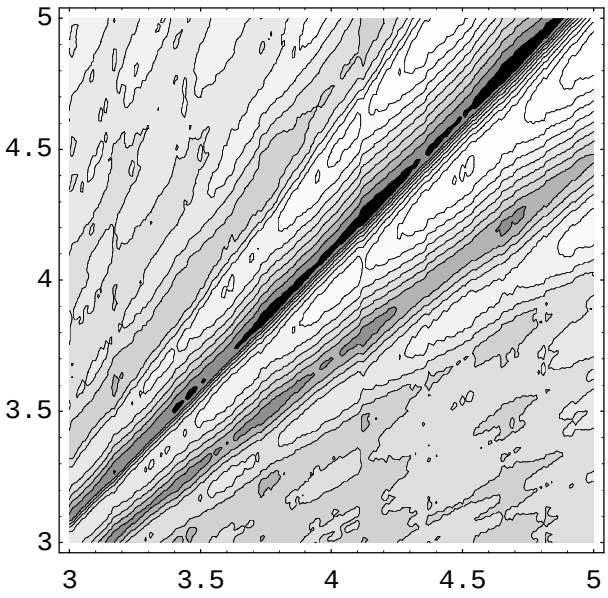


Figure 7.2: A fine map of $E(\alpha, \beta)$.

```
local

fun best_list_prune1(Min_call_count,Xs:best_type list) =
  case Xs of
    nil => nil
  | X1::Xs1 =>
      if #call_count  X1 < Min_call_count then
        X1::best_list_prune1(#call_count X1,Xs1)
      else
        best_list_prune1(Min_call_count,Xs1)

in

fun best_list_prune(X1::Xs1) =
  X1::best_list_prune1(#call_count X1,Xs1)

end (* local *)
```

Figure 7.3: The ML function for pruning `Best_list`.

Figure 7.1 shows a contour plot of $E(\alpha, \beta)$ for $2 \leq \alpha \leq 16$ and $2 \leq \beta \leq 16$ with α on the vertical axis and β on the horizontal axis. Black represents the smallest $E(\alpha, \beta)$ values whereas white represents the greatest values. By studying the numerical data, we found that the region $3 \leq \alpha \leq 5$ and $3 \leq \beta \leq 5$ contained the best $E(\alpha, \beta)$ values. This region is shown in Figure 7.2, where the best $E(\alpha, \beta)$ values are approximately located along the line $\alpha = \beta$. Since smaller α and β values give less variance for the ratio $W_P(C, \alpha, \beta)/C$, we chose a good (α, β) value not too far from $(0, 0)$, namely $\alpha = 3.88$ and $\beta = 3.77$. Since $E(3.88, 3.77) \approx 3.93$, the average execution time of ADATE increases about 3.93 times because we cannot employ a fixed, minimum $\texttt{Cost\_limit}_1$ value.

7.4 Which are the Best Synthesized Programs?

Let $pe_{\text{REQ}-min}$ be the best pe_{REQ} value of any program that has been found so far. During the entire execution, ADATE maintains a list `Best_list` that only contains programs with $pe_{\text{REQ}} = pe_{\text{REQ}-min}$. Each synthesized program is considered for insertion into `Best_list`. Let $(S_1, T_1), (S_2, T_2), \ldots, (S_n, T_n)$ be the syntactic complexity and call count values of the programs in `Best_list`, which is sorted so that $(S_1, T_1) \leq (S_2, T_2) \leq \ldots \leq (S_n, T_n)$. When a new program with $pe_{\text{REQ}} = pe_{\text{REQ}-min}$ has been found, it is inserted into the appropriate position in `Best_list`, which is then pruned so that the $T_1, T_2, \ldots, T_n$ values are strictly decreasing. The pruning function is shown in Figure 7.3. The purpose

of this pruning is to ensure that a greater syntactic complexity always is compensated by a smaller call count. When ADATE synthesizes a program with a pe_{REQ} value that is better than any other pe_{REQ} value found so far, `Best_list` is set to the singleton list that only contains this program.

The execution of ADATE is viewed as a perpetual process i.e., the more execution time the better. It is up to the user to decide when execution is to be terminated. The output of ADATE is the contents of `Best_list`.

Chapter 8

Sample Specifications, Inferred Programs and Run Times

Polynomial simplification. This problem was discussed in Section 3.4. The specification consisted of

1. The type `int` and the type declaration `datatype 'a list = nil | :: of 'a * 'a list`.
2. The primitive `= : int * int -> bool`.
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.

   I_1 = `[ ]`

   I_2 = `[ (3,2) ]`

   I_3 = `[ (3,2), (5,2), (12,2), (11,2) ]`

   I_4 = `[ (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 Figure 3.1.

Note that these 4 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.

```
fun simplify Xs =
  case Xs of nil => Xs
  | (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

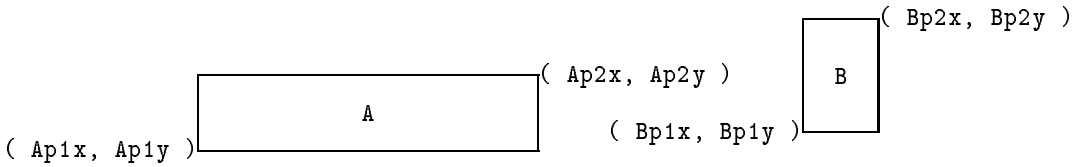


Figure 8.1: Two non-intersecting rectangles and their coordinates.

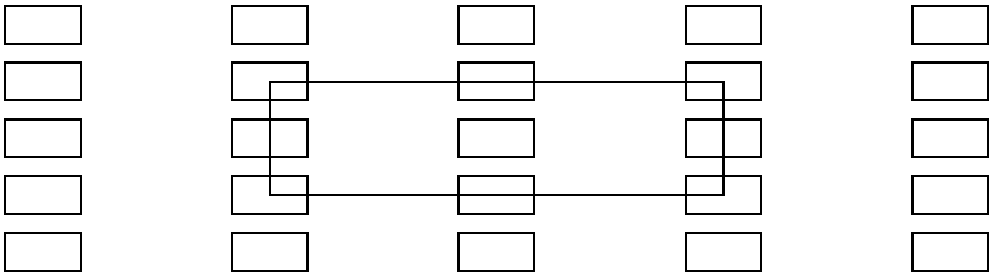


Figure 8.2: The set of input rectangles.

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. Figure 8.1 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 Figure 8.2 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`. 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 Subsection 4.1.4.

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 one auxiliary function.

List sorting. The specification contained the `datatype`-definition for lists and the relation `<` on integers. The sample inputs were given in Subsection 4.1.1. Subsection 4.1.2 contains the inferred program. Appendix B shows the complete output of ADATE for this sample inference.

List splitting. The specification contained the `datatype`-definition for lists. The output evaluation function was described in Section 3.3.

The run times shown in Table 8.1 were obtained using the Standard ML of New Jersey compiler, version 0.93, and IBM RS6000-580 and RS6000-590 workstations running only ADATE and AIX processes. Most of the experiments were run on the 590 but some were run on the 580. For the latter experiments, the table shows the equivalent run times on the 590. We found that the ratio between 580 and 590 run times is 1.5. The equivalent times on SUN SparcStation 10's or DECStation 5000's are two to three times longer.

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.

Table 8.1: Run times.

Problem	*Run time in hours:minutes*
Polynomial simplification	22:56
Rectangle intersection	4:35
BST deletion	70:51
BST insertion	16:23
List reversal	0:4
List intersection	5:10
List delete min	12:7
Permutation generation	22:35
List sorting	0:27
List splitting	0:3

Chapter 9

Related Work

We will discuss the following four categories of work that is related to ADATE.

1. Program synthesis using computation traces.
2. Inductive logic programming.
3. Genetic programming.
4. Program transformation.

Category 1 is older and less interesting than categories 2, 3 and 4. Therefore, we will only discuss it briefly. The most interesting work in category 1 is the inference of LISP programs from input-output pairs as surveyed by D.R. Smith [Smith 82]. Smith writes that the methods in his survey stem from Summer's [Summers 77] insight 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. 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.

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

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.

Categories 2, 3 and 4 are rarely discussed together in the literature even though they all study automatic inference of programs. One reason for this separation is that categories 2 and 3 are only a few years old. We will use the following criteria to evaluate categories 2, 3, 4 and ADATE.

Specification form. What is the ratio between the difficulty of writing a specification and the difficulty of writing a program that satisfies it?

Degree of automation. How much interaction between the system and the user is required?

Creativity. Can the system create good programs that are novel, non-trivial and unexpected?

Inventivity. Can the system invent new functions and data types?

Program constraints. What are the forms of inferred programs? For example, can the system deal with recursion and real-valued constants?

Effectiveness in various domains. For which types of problems is the system suitable? Which class of algorithms can be inferred in each domain?

Efficiency. Time and space complexity for

1. the inference system and
2. inferred programs.

Next, we present categories 2, 3, 4 and discuss them with respect to these criteria. We will use the criteria to evaluate ADATE in Chapter 10. These three categories and ADATE have disjoint capabilities and somewhat different goals, which means that it is difficult to compare them directly using a common set of criteria. Each category is interesting and unique from several points of view. It is desirable to keep this in mind when reading the following critical presentations.

9.1 Inductive Logic Programming

A system in this category uses specifications consisting of

1. Background knowledge K, which in the most general case is a set of user supplied predicate definitions.
2. Positive examples ε^+, which are ground atoms.
3. Negative examples ε^-, which also are ground atoms.

The system tries to find a set of clauses H such that ε^+ can be inferred from H and K using SLD-resolution with a depth-first search strategy and such that ε^- cannot be inferred. The following example illustrates ILP specification.

Example. Consider the problem of finding a definition of a predicate `sort` such that `sort(Xs,Ys)` holds if and only if the list of integers `Ys` is a sorted permutation of the list `Xs`. Here is a typical ILP specification that facilitates the inference of insertion sort.

1. Background knowledge.

```
insert(X,[],[X]).
insert(X,[Y|Ys],[X,Y|Ys]) :- X<=Y.
insert(X,[Y|Ys],[Y|Zs]) :- Y<X, insert(X,Ys,Zs).
```

2. Positive examples.

```
sort([],[]).
sort([1],[1]).
sort[2],[2]).
...
sort([3,1,2],[1,2,3]).
...
```

3. Negative examples.

```
sort([],[1]).
sort([],[2]).
...
sort([3,1,2],[3,1,2]).
...
```

Using this specification, an ILP system would hopefully infer the following definition.

```
sort([],[]).
sort([X|Xs],Ys) :- sort(Xs,Zs), insert(X,Zs,Ys).
```

The negative examples are not needed if the closed world assumption is used. This assumption means that each example that is not in ε^+ is assumed to be in ε^-. This is controversial since ε^+ usually contains only a small fraction of all positive examples. Normally, there are infinitely many possible positive examples whereas ε^+ is finite and explicitly listed in the specification. Here is a critical evaluation of ILP with respect to the criteria listed above.

Specification form. We have discovered four fundamental problems with ILP specifications. Here are the four problems together with illustrative examples.

Problem 1. Typically, an ILP specification requires

1. extremely many examples or
2. very well chosen examples.

In the above specification, for example, ε^+ must contain two examples e_1 and e_2 such that `sort(Xs,Zs)`$(mgu($`e1`$,$`sort([X|Xs],Ys)`$))=e_2$ e.g. e_1 =`sort([3,2,1],[1,2,3])` and e_2 =`sort([2,1],[1,2])`. Thus, e_2 is the recursive call that `sort` makes when working on e_1. Two alternative ways of ensuring that ε^+ contains e_1 and e_2 are

1. to list all possible examples up to some maximum size or
2. to have a clever user who can anticipate the forms of recursive calls and choose examples accordingly.

Alternative (1) was chosen in the specification of `sort` above, which has $\varepsilon^+ \cup \varepsilon^- = \{\texttt{sort}(\mathrm{As}_1, \mathrm{As}_2) : \mid \mathrm{As}_i \mid \leq 3 \text{ and } \mathrm{As}_i \subseteq \{1,2,3\}\}$, which gives $\mid \varepsilon^+ \cup \varepsilon^- \mid = (1+3+9+27)^2 = 1600$.

Both alternatives are often unfeasible.

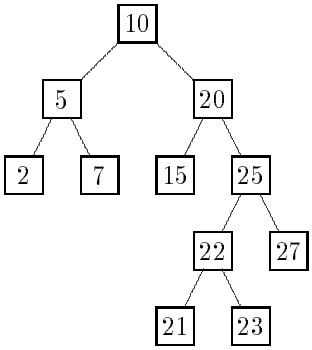
Alternative (1) is unfeasible since $\mid \varepsilon^+ \cup \varepsilon^- \mid$ often grows super-exponentially with the maximum size $S_{\max}$ and since $S_{\max}$ in general only can be kept small for "toy examples". In the specification above, $\mid \varepsilon^+ \cup \varepsilon^- \mid = (\sum_{S=0}^{S_{\max}} S_{\max}^S)^2$ which is $\Omega(S_{\max}^{2S_{\max}})$. With the closed world assumption, $\mid \varepsilon^+ \cup \varepsilon^- \mid = \mid \varepsilon^+ \mid = \sum_{S=0}^{S_{\max}} S_{\max}^S$ which is $\Omega(S_{\max}^{S_{\max}})$.

Alternative (2) is undesirable since a user who can anticipate all forms of recursive calls most likely also is able to write the program to be inferred and thus does not need an inference system. This "call anticipation problem" is not well studied by the ILP community even though it seems to impose a fundamental limit on ILP systems when it comes to inferring recursive programs. Unfortunately, the call anticipation problem appears over and over again.

Example. A predicate `bst_del` is such that `bst_del(X,Xs,Ys)` holds if and only if deletion of the integer `X` from the binary search tree (BST) `Xs` yields the BST `Ys`. The binary tree constructors are `bt_nil` and `bt_cons`. A `bt_cons` term has the form `bt_cons( Root, Left_sub_tree, Right_sub_tree )`. For example, assume that the following definition of `bst_del` is to be inferred.

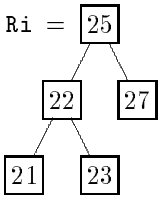
```
bst_min( bt_cons(Ro,bt_nil,_), Ro ).
bst_min( bt_cons(_,Le,_), M ) :- bst_min(Le,M).

bst_del( X, bt_nil, bt_nil ).
bst_del( X, bt_cons(Ro,Le,Ri), bt_cons(Ro,Le',Ri) ) :-
  X<Ro, bst_del(X,Le,Le').
bst_del( X, bt_cons(Ro,Le,Ri), bt_cons(Ro,Le,Ri') ) :-
  Ro<X, bst_del(X,Ri,Ri').
bst_del( X, bt_cons(X,bt_nil,Ri), Ri ).
bst_del( X, bt_cons(X,Le,bt_nil), Le ).
bst_del( X, bt_cons(X,Le,Ri), bt_cons(M,Le,Ri') ) :-
  bst_min(Ri,M), bst_del(M,Ri,Ri').
```

Also assume that 20 is to be deleted from the BST

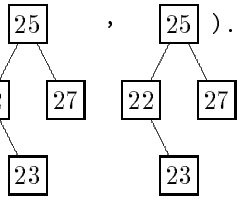


The recursive call `bst_del(M,Ri,Ri')` is then made with `M` = 21 and

`Ri` =



This means that ε^+ would need to contain the example

`bst_del( 21,`  `,` `).`

A user of an ILP system unfortunately needs to know the `bst_del` program *before* it is inferred in order to anticipate the need for this example.

Problem 2. Many specifications should not contain any outputs at all since the outputs reflect the user's knowledge of a particular algorithm. This is also illustrated by the BST deletion specification above. Thus, it is better to specify a requirement that the output must satisfy e.g.

```
bst_del(X,Xs,Ys) :-
  inorder(Xs,Xs_nodes), inorder(Ys,Ys_nodes),
  del_one(X,Xs_nodes,Ys_nodes).
```

The call `inorder(Zs,Nodes)` puts the inorder listing of the tree `Zs` in the list `Nodes`. The call `del_one(Z,Zs,Zs')` deletes one occurrence of `Z` from the list `Zs` yielding `Zs'`. Note that the predicates `inorder`

and `del_one` may be easily defined by the user without knowledge of any good BST deletion algorithm.

Problem 3. A serious limitation of ILP specifications is that they do not use continuous grading of output quality and program time and space complexity. For example, the `bst_del` program should produce an output BST of small "average" depth in time $O(\log n)$, where n is the number of nodes in the input BST, which is assumed to have depth $O(\log n)$. A powerful inference system would perhaps be able to utilize the depth and time grades to infer a BST deletion algorithm that uses techniques similar to Tarjan's [Tarjan 83] splay heuristic. However, this algorithm would be much more complicated than the one above.

Problem 4. Most ILP systems need to have the background knowledge in the form of ground facts. There are two ways of achieving this, namely

1. To require that the user provides ground facts instead of clauses. This method is used by FOIL [Cameron-Jones and Quinlan 94].
2. To convert clauses to ground facts as in GOLEM [Muggleton and Feng 92]. This method is problematic since the number of ground facts needs to be limited by only allowing a maximum of h binary resolutions when producing a fact. The number of ground facts grows exponentially with h even for very simple and common examples.

   Example. Consider a predicate `is_bt` which holds if and only if its argument is a binary tree. For simplicity, assume that no information is stored in the tree nodes, which gives the following definition of `is_bt`.

   ```
   is_bt(bt_nil).
   is_bt(bt_cons(Le,Ri)) :- is_bt(Le), is_bt(Ri).
   ```

   It is obvious that the number of binary trees with h nodes grows exponentially with h.

Thus, we have identified the following four drawbacks of ILP specification.

1. The need for either extremely many examples or call anticipation.
2. The need for outputs that mirror a particular algorithm.
3. Inability to utilize continuous grades.
4. Conversion of background knowledge to ground facts.

An ILP system, that suffers from one or more of these specification problems, will never become an effective tool for general purpose logic programming. All systems in the literature suffer from at least three of the problems.

Degree of automation. Some ILP systems, such as MIS [Shapiro 83], CIGOL [Muggleton and Buntine 88] and SIERES [Wirth and O'Rorke 92], ask the user questions during the inference process. An example of such a question is "Give me a `Ys` such that `insert(4,[1,5,8],Ys)` holds". This so-called oracle requirement is used to circumvent the call anticipation problem. However, the oracle requirement is in general unreasonable since

1. the user, as discussed above, should not be required to provide outputs and
2. very many questions may need to be asked.

Creativity. The literature does not contain any novel recursive algorithm developed by an ILP system. The inference processes are on the contrary strongly guided towards a specific, pre-conceived algorithm.

Inventivity. A particularly interesting development in ILP is the invention of new predicates, which is reviewed by Irene Stahl [Stahl et. al. 93]. 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 Subsection 4.1.2, 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 [Muggleton and Buntine 88] and SIERES [Wirth and O'Rorke 92], 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 literature does not describe any ILP system that can generalize argument types e.g. change the type "list of integers" to the type "list of lists of integers".

Program constraints. In general, ILP systems impose few constraints on the form of inferred programs. GOLEM employs the so-called ij-determinate restriction, which leads to an enormous reduction of the search space when using relative least general generalization to construct clauses. Even though this restriction rules out certain non-deterministic programs, for example the standard n-queens program, it still seems to be a mild restriction that does not preclude most interesting programs.

FOIL only infers function free clauses, but this does not make FOIL less general since programs easily can be rewritten to function free form by introduction of extra predicates. FOIL also restricts recursive calls so that non-termination always is avoided.

Effectiveness in various domains. ILP is a promising method for automatic "concept" learning in domains with vast amounts of real-world data. The logic programs that describe "concepts" are typically non-recursive and have much of the same flavour as the decision trees inferred by Quinlan's ID3 algorithm [Quinlan 86].

Given a well chosen specification and a "helpful oracle", ILP systems are also quite good at inferring recursive programs.

ILP is normally not applied to numerical, continuous optimization problems that require the inference of one or more floating point numbers.

Efficiency. ILP systems perform a rather constrained search and therefore usually infer programs very rapidly.

There are very few useful theoretical results on the time complexity of ILP. One recent result [Muggleton and Feng 92], for example, is that the length of clauses produced with relative least general generalization and the ij-determinate restriction is $O((mft)^{(j^i)})$, where m is the number of predicates in the specification, f is the maximum arity of such a predicate and t is the number of terms in the least general generalization of the examples. The usefulness of this upper bound strongly depends on i and j, which are maximum values of two syntactic measures of clause complexity. Roughly speaking, i is the maximum depth of the variable dependency DAG for any clause. See [Muggleton and Feng 92] for details. Muggleton and Feng found that $i = j = 2$ suffice for a number of recursive programs such as the ones for list reversal and Quicksort. Note that this empirical result is necessary to motivate the usefulness of the theoretical result.

Due to the lack of useful, purely theoretical time complexity results, there is a strong need for experimental assessment of ILP systems. To check the extremely short run times given in the ILP literature, we decided to try version 6 of FOIL on the insertion sort specification given above. The experiment was carried out using a SUN SparcStation 10 and the GNU C compiler. The closed world assumption was employed. The background knowledge clauses for `insert` were converted to all positive examples involving numbers chosen from $\{1, 2, 3\}$ and lists of length less than or equal to 3.

FOIL inferred the following insertion sort program in 2.1 seconds.

```
sort([],[]) :-
sort(A,B) :- component(A,C,D), sort(D,E), insert(C,E,B)
```

This efficiency is really splendid.

However, the efficiency of an inferred program is rather arbitrary since FOIL only considers termination and does not use other time complexity measures.

9.2 Genetic Programming

The primary difference between genetic programming (GP) and genetic algorithms (GAs) is that the former encodes a solution as a LISP program whereas the latter normally uses bit string encoding. Since GP is an offspring from GA research, much of the discussion below holds for GAs as well. The originator of GP is John Koza [Koza 92].

A GP specification contains the following.

1. Background knowledge consisting of a set of constants and functions that are to be used in inferred LISP programs.
2. A fitness function which takes an inferred program as argument and returns a floating point number. The probability of "survival" of the program is proportional to this number.
3. Sample inputs that are used to compute the fitness function.
4. Search control parameters. There are 19 parameters, but only a few of them normally needs to be adjusted when tackling a new inference problem. Some of the most important parameters are below given with default values in parentheses.
   (a) Population size (500).
   (b) Maximum number of generations (51).
   (c) Probability of crossover (0.9).
   (d) Probability of reproduction (0.1).
   (e) Probability of mutation (0).

As indicated by the choice of default parameter values, the main program transformation is crossover, i.e., random exchange of subexpressions between two programs. Crossover, consisting of an exchange of substrings, is also the most important transformation in GAs.

Crossover is only effective if the schema theorem [Holland 76] is applicable. We have identified the following basic problem with crossover. When inferring a large expression E, the schema theorem requires that E primarily is 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 $j \neq i$.

Each E_i may be viewed as a "schema". Unfortunately, practically all recursive programs consist of coupled E_i's.

Example. 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 [Koza 92].

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

The right hand side can be written as $E_1 E_2$ with

$$E_1 = \texttt{fn As => if null Xs then Ys else As}$$

and

$$E_2 = \texttt{hd Xs :: @(tl Xs,Ys)}.$$

The fitness advantage of E_2 cannot 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 only the inference of one single "recursive" program is presented in Koza's book. This program, which computes the Fibonacci numbers, does not contain any explicit recursive calls. Instead, it uses a problem-specific, user-defined operator `srf` which provides memoization. The operator is defined so that `(srf K D)` returns the value `(fib K)` if `K` is smaller than `J` which is the argument of the first call `(fib J)`. Otherwise, `(srf K D)` returns the default value `D`. Thus, this sample inference is rather tricky and dependent on the specialized `srf` operator.

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

Here is an evaluation of GP with respect to the criteria listed above.

Specification form. The form of specifications is similar to the one in our ADATE system. This form was presented in Section 3.3. The specifications in Koza's book are very good at supporting evolutionary program development and are well worth studying independently of the rest of the book.

The main difference between ADATE and GP specifications is that the latter very rarely use more than one fitness measure, whereas the former use at least four measures. In particular, the syntactic complexity measure

is not used by GP, which means that inferred programs normally are much more complicated than they need to be. In order to alleviate this problem, Koza allows the specification to contain domain-specific rewrite rules that are used to simplify programs during the inference process. An example of such a rule is `(append Xs nil)` $\longrightarrow$ `Xs`. Considering the state-of-the-art in rewrite system research, this approach is far from generally applicable.

Degree of automation. GP is fully automatic i.e., does not rely on user interaction.

Creativity. The programs developed by GP are very different from normal programs The primary reasons are that GP programs

1. often are extremely complicated in comparison with normal programs and
2. contain mathematical equations that are "almost correct", whereas equations derived by mathematicians more often either are completely correct or wrong.

Koza's notion of "almost correct" is quite interesting and greatly facilitates evolutionary inference.

Inventivity. Koza uses a program transformation that he calls "automatic function definition". This transformation is similar to the abstraction transformation discussed in Subsection 4.1.2. No invented recursive functions are presented in Koza's book. Neither is there any transformation similar to the embedding transformation that was introduced in Subsection 4.1.4.

Program constraints. Recursive calls are not allowed in any inference presented in Koza's book. With this exception, there are few constraints on inferred programs. Since LISP has an unusually poor type system, GP does normally not even use type constraints. In many cases, the lack of typing unfortunately leads to an enormous increase in search space cardinality.

Effectiveness in various domains. Given specifications that facilitate evolutionary inference, GP is amazingly good at inferring formulas. Most of the inferred "programs" in Koza's book are in fact mathematical formulas that do not use common programming language constructs such as iteration, recursion and case-tests. As explained above, iterative or recursive programs cannot be effectively produced by GP.

Efficiency. Koza does not provide any run times, but it is obvious that GP is very computationally demanding. Chapter 8 in his book presents experimental results concerning the total number of programs produced during

an inference. This number is in the neighbourhood of 10^5 for simple problems and in excess of 10^6 for the more difficult problems.

Since the run time of inferred programs is not used as a fitness measure, they are unlikely to be particularly efficient.

9.3 Program Transformation

The deductive inference of programs from formal specifications, which usually are expressed in predicate logic or similar formalisms, is an old and well studied area of research. We will here only review the inference of executable programs from specifications that are unfeasible to execute. The reason for this unfeasibility is either that the specification is non-constructive or that it requires at least exponential time to execute. Thus, this review will not cover optimizing compiler transformations such as common subexpression elimination, loop unrolling etc. The deductive inference of programs from non-constructive specifications is a research area that aims for a much lower degree of automation than ADATE, which means that it is more weakly related to ADATE than ILP and GP. Therefore, we will give a less detailed presentation of this area.

A formal specification of a function f often contains the following.

1. Background knowledge consisting of types, function definitions and possibly also specialized inference information such as function-specific rewrite rules.
2. The type $D \rightarrow R$ of f.
3. An input condition $i : D \rightarrow$`bool` such that $i(I)$ must hold for each legal input I.
4. An output condition $o : D \times R \rightarrow$`bool`. A program is correct if and only if $i(I)$ implies $o(I, f(I))$ for all I in D.

Below is an example of a Horn clause specification of a sorting function f. The specification can be directly executed using SLD-resolution but the time required to sort n integers is $\Omega(n!)$.

1. The types `int` and `int list` and the $<$-relation on integers are given as background knowledge, which also contains the definitions

```
del_one(X,[X|Xs],Xs).
del_one(X,[Y|Xs],[Y|Ys]) :- del_one(X,Xs,Ys).

is_perm([],[]).
is_perm([X|Xs],Ys) :- del_one(X,Ys,Zs), is_perm(Xs,Zs).
```

```
sorted([]).
sorted([X]).
sorted([X1,X2|Xs]) :- X1<=X2, sorted([X2|Xs]).
```

2. The type of f is `int list` $\rightarrow$ `int list`.
3. The input condition is always true.
4. The output condition is

```
o(Xs,Ys) :- is_perm(Xs,Ys), sorted(Ys).
```

A program transformation system such as the one in [Komorowski 93] can assist in the gradual transformation of the above specification to a reasonably efficient sorting program.

Here is a characterization of program transformation using the same criteria as for ILP and GP.

Specification form. It is quite clear that a formal specification can be very much simpler than the programs that satisfy it. Another advantage of a formal specification is of course that it practically always is sufficient.

Degree of automation. Practically all program transformation systems, for instance KIDS [Smith 90] and PROSPECTRA [Krieg-Brückner et. al. 91], are semi-automatic and totally dependent on system-user interaction during an inference. This dependence is so pervasive that program transformation is not to be regarded as an area of machine learning whereas ILP, GP and ADATE all are machine learning methods. The goal of program transformation research is machine-aided programming rather than fully automatic programming.

Since program transformation systems need deductive inference and theorem proving, fully automatic and general program transformation seems to require fully automatic and general theorem proving, which yet is to be achieved.

Creativity. Since a program transformation system is strongly dependent on the user, it is not creative.

Inventivity. Inventivity also depends on the user.

Program constraints. There are practically no constraints. The programming language does not even need to be applicative.

Effectiveness in various domains. Given a sufficiently competent user, it is hard to think of any algorithm in any domain that cannot be developed using a program transformation system.

Efficiency. Program transformation systems are usually very efficient since the search for deductive inferences is highly constrained and user controllable. Inferred programs can also be very efficient.

Chapter 10

Conclusions and Future Work

We start this chapter by evaluating ADATE with respect to the criteria that were used for evaluating inductive logic programming, genetic programming and program transformation in Chapter 9.

Specification form. The combination of sample inputs and output evaluation function must support evolutionary program transformation. Otherwise, there are few constraints on the form of a specification, which may be much easier to write than any desirable program. The requirement that the specification must support evolutionary transformation resembles the requirement that text-books must be pedagogical in order to support gradual and progressive learning. A minimum requirement is that there exists a genealogical path $P_1, P_2, \ldots, P_{n-1}, P_n$, where P_1 is the initial program, P_n is a desirable program and the compound transformation costs C_i required to transform P_i to P_{i+1} are so low that no C_i corresponds to more than a few hours of CPU time. The writing of pedagogical text-books relies more on common sense than on fixed rules. This also characterizes the art of specification writing.

Degree of automation. ADATE is fully automatic. The only run time "interaction" is that the user should decide when to terminate an inference i.e., when sufficiently good program evaluation function values have been achieved.

Creativity. ADATE can automatically synthesize novel and non-trivial recursive programs. Even though the programs written by ADATE may be quite difficult to discover for human programmers, there is still an enormous gap between human creativity and the "creativity" of ADATE.

Inventivity. In comparison with the program synthesis systems reviewed in Chapter 9, ADATE has a superior ability to automatically invent new functions and data types.

Program constraints. One constraint in the current version of ADATE is that synthesized recursive calls are required to contain at least one "decreasing" argument as discussed in Subsection 5.2.3. However, this constraint is only employed to reduce the number of synthesized expressions and would be easy to remove if one is willing to accept a two- or three-fold increase of the average inference time. There are few other constraints.

Effectiveness in various domains. The current version of ADATE is especially suitable for inferring small recursive programs such that a few auxiliary functions are missing in the specification. Of course, it can infer non-recursive programs as well. There are no mechanisms for optimization of numerical constants that occur in inferred programs.

Efficiency. The main disadvantage of ADATE is the long inference times. The systems for induction of logic programs reviewed in Chapter 9 are much faster. However, they do need to acquire much more knowledge from the users.

The program evaluation function pe_2 was designed to contribute to the inference of programs with good time complexity, but it seems to be difficult to always achieve the best possible time complexity using "natural" specifications. For example, the ADATE user may have to be satisfied with $O(n^2)$ instead of $O(n \log n)$ for sorting.

The current version of ADATE does not consider space complexity.

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 $\texttt{pe}_i$. 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.

Bibliography

[Angluin 78] D. Angluin, On the complexity of minimum inference of regular sets, Information and Control 39 (1978) 337–350.

[Angluin and Smith 83] D. Angluin and C.H. Smith, Inductive inference: theory and methods, Computing Surveys 16 (1983) 239–269.

[Biermann and Krishnaswamy 76] A.W. Biermann and R. Krishnaswamy, Constructing programs from example computations, IEEE Transactions on Software Engineering 2 (1976) 141–153.

[Biermann 78] A.W. Biermann, The inference of regular LISP programs from examples, IEEE Transactions on Systems, Man and Cybernetics 8 (1978) 585–600.

[Blumer et. al. 86] A. Blumer, A. Ehrenfeucht, D. Haussler and M. Warmuth, Classifying learnable geometric concepts with the Vapnik-Chervonenkis dimension, Proceedings of the 18th ACM Symposium on Theory of Computing (1986) 273–282.

[Boehm 76] B.W. Boehm, Software engineering, IEEE Transactions on Computers 25 (1976) 1226–1241.

[Boehm 88] B.W. Boehm, A spiral model of software development and enhancement, IEEE Computer 21 (1988) 61–72.

[Cameron 88] R.D. Cameron, Source encoding using syntactic information source models, IEEE Transactions on Information Theory 34 (1988) 843–850.

[Cameron-Jones and Quinlan 94] R.M. Cameron-Jones and J.R. Quinlan, Efficient top-down induction of logic programs, SIGART Bulletin 5 (1994) 33–42.

[Dahl 92] O.J. Dahl, Verifiable Programming (Prentice Hall International, 1992).

[Garey and Johnson 79] M.R. Garey and D.S. Johnson, Computers and Intractability (W.H. Freeman, San Fransisco, 1979).

[Glover 89] Tabu search, Part I, ORSA Journal of Computing 1 (1989) 190–206.

[Gold 78] E.M. Gold, Complexity of automaton identification from given data, Information and Control 37 (1978) 302–320.

[Holland 76] J.H. Holland, Adaptation in Natural and Artificial Systems (University of Michigan Press, 1976).

[Johnson 90] D.S. Johnson, Local optimization and the traveling salesman problem, Proceedings of the 17th Colloquium on Automata, Languages and Programming, (Springer-Verlag, 1990) 446–461.

[Johnson 94] D.S. Johnson, Data structures for traveling salesmen, unpublished manuscript, (personal communication, 1994).

[Jounnaud and Kodratoff 80] J.P. Jounnaud and Y. Kodratoff, An automatic construction of LISP programs by transformations of functions synthesized from their input-output behavior, International Journal of Policy Analysis and Information Systems 4 (1980) 331–258.

[Kijsirikul et. al. 92] B. Kijsirikul, M. Numao, and M. Shimura, Discrimination based constructive induction of logic programs, Proceedings of the Tenth National Conference on Artificial Intelligence (MIT Press, 1992) 44–49.

[Kirkerud 92] B. Kirkerud, Lecture Notes for in307, Department of Informatics, University of Oslo (1992).

[Kirkpatrick et. al. 83] S. Kirkpatrick, C.D. Gellat and M.P. Vecchi, Optimization by simulated annealing, Science 200 (1983) 671–680.

[Komorowski 93] J. Komorowski, Special issue on partial deduction, Journal of Logic Programming 16 (1993) Guest editor.

[Korf 85] R.E. Korf, Depth-first iterative-deepening: an optimal admissible tree search, Artificial Intelligence 27 (1985) 97–109.

[Koza 92] J.R. Koza, Genetic Programming (MIT Press, Cambridge, Massachusetts, 1992).

[Krieg-Brückner et. al. 91] B. Krieg-Brückner, E.W. Karlsen, J. Liu, and O. Traynor, The PROSPECTRA methodology and system: Uniform transformational (Meta-) development, Proceedings of the VDM'91 Symposium (Springer-Verlag, 1991).

[Li and Vitányi 93] M. Li and P.M.B. Vitányi An Introduction to Kolmogorov Complexity and Its Applications (Springer-Verlag, 1993).

[Muggleton and Buntine 88] S.H. Muggleton and W. Buntine, Machine invention of first-order predicates by inverting resolution, in: Proceedings of the Fifth International Conference on Machine Learning (Morgan-Kaufmann, 1988) 339–352.

[Muggleton 92] S.H. Muggleton, Inductive logic programming, in: S. Muggleton, ed., Inductive Logic Programming (Academic Press, London, 1992) 4–21.

[Muggleton and Feng 92] S.H. Muggleton and C. Feng, Efficient induction of logic programs, in: S.H. Muggleton, ed., Inductive Logic Programming (Academic Press, London, 1992) 281–298.

[Olsson 93] R. Olsson, Execution of logic programs by iterative-deepening A* SLD-tree search, BIT 33 (1993) 214–231.

[Partsch 90] H.A. Partsch, Specification and Transformation of Programs: A Formal Approach to Software Development (Springer-Verlag, 1990).

[Peyton Jones 87] S.L. Peyton Jones, The Implementation of Functional Programming Languages (Prentice-Hall, 1987).

[Pitt and Warmuth 89] L. Pitt and M. Warmuth, The minimum consistent DFA problem cannot be approximated within any polynomial, Proceedings of the 21st ACM Symposium on Theory of Computing (ACM Press, 1989) 421–432.

[Quinlan 86] J.R. Quinlan, Induction of decision trees, Machine Learning 1 (1986) 81–106.

[Rissanen 82] J. Rissanen, A universal prior for integers and estimation by minimum description lengths, Annals of Statistics 11 (1982) 416–431.

[Shapiro 83] E.Y. Shapiro, Algorithmic Program Debugging, (MIT Press, 1983).

[Smith 82] 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.

[Smith 90] D.R. Smith, KIDS: A semiautomatic program development system, IEEE Transactions on Software Engineering 16 (1990) 1024–1043.

[Stahl et. al. 93] I. Stahl, B. Tausend and R. Wirth, Predicate invention in ILP – an overview, Proceedings of the European Conference on Machine Learning (Springer-Verlag, 1993) 41–55.

[Summers 77] P.D. Summers, A methodology for LISP program construction from examples, Journal of the ACM 24 (1977) 161–175.

[Tarjan 83] R.E. Tarjan, Data Structures and Network Algorithms, (SIAM Press, 1983).

[Valiant 84] L.G. Valiant, A theory of the learnable, Communications of the ACM 27 (1984) 1134–1142.

[Wikström 87] Å. Wikström, Functional Programming Using Standard ML (Prentice Hall International, 1987).

[Wirth and O'Rorke 92] R. Wirth and P. O'Rorke, Constraints for predicate invention in: S. Muggleton, ed., Inductive Logic Programming (Academic Press, London, 1992) 299–318.

Appendix A

The ML Definition of Syntactic Complexity

```
val Normal_lengths =
  ( ~(ln 0.025), ~(ln 0.15), ~(ln 0.325), ~(ln 0.5) )
val Analyzed_lengths = ( ~(ln 2.5E~3), ~(ln 1.5E~2),
  ~(ln 0.387045454545455), ~(ln 0.595454545454546) )

fun syntactic_complexity( D : ('a,'b)d ) : real =
let
fun sc_of_exp( N_internals, N_leaves,
      Lengths as (Let,Case,Internal,Leaf), E : ('a,'b)e ) : real =
  case E of
    app_exp{func,args,...} =>
      if func="?" then
        if is_not_activated_exp E then
          0.0
        else
         Leaf + ln(real N_leaves)
      else if null args then
        Leaf + ln(real N_leaves)
      else
        Internal + ln(real N_internals) + real_sum(
        map(fn A => sc_of_exp(N_internals,N_leaves,Lengths,A),
          args)) +
        (if null(tl args) orelse func="tuple" then
           0.0
         else
           Internal + ln(real N_internals)
```

```
          (* Accounts for the "implicit" tuple constructor. *)
        )
  | case_exp{exp,rules,...} =>
      Case +
      sc_of_exp( N_internals, N_leaves,
        case rules of _::nil => Lengths | _ => Analyzed_lengths,
        exp) +
      real_sum(map( fn{pat,exp} =>
            sc_of_exp( N_internals, N_leaves+length(vars_in_pat pat),
              Lengths, exp ),
            rules))
  | let_exp{dec_list,exp,...} =>
      Let+
      real_sum(map( fn D => sc_of_dec( N_internals+length(dec_list),
                                       N_leaves, Lengths,
                                       D),
               dec_list )) +
      sc_of_exp( N_internals+length(dec_list), N_leaves, Lengths,
        exp )

and sc_of_dec( N_internals, N_leaves, Lengths, {pat,exp,...}
    : ('a,'b)d ) =
  sc_of_exp( N_internals, N_leaves+length(vars_in_pat pat),
    Lengths, exp )

val Arity_zero_funs = filter( fn F =>
  case assoc(F,Predefined.ty_env) of
    { ty_exp=ty_con_exp("->",_),... } => false
  | _ => true,
  Spec.Funs_to_use
  )

in
  sc_of_dec( 2+length(Spec.Funs_to_use)-length(Arity_zero_funs),
                  length(Arity_zero_funs)+1, Normal_lengths,
                  D ) / ln 2.0
end (* fun syntactic_complexity *)
```

Appendix B

The Raw Log File for List Sorting

The log file is basically self-explanatory. The sample inputs in the specification were presented in Subsection 4.1.1. ADATE regards a program as correct if it can sort all five of these input lists. Note that the first such program is the one with identification number `(4,1,828)`, which is found after 1617.52 seconds (about 27 minutes) of execution time. At end of the log file, after the text

```
THE BEST INDIVIDUALS FOUND SO FAR ARE
```

are the best programs found thus far when the inference process was interrupted. The smallest of these programs is `(3,1,1420)`, which is less complex than `(4,1,828)`.