How to Invent Functions

J. Roland Olsson

Department of Computer Science
Chalmers University
412 96 Gothenburg
Sweden
(rolsson@cs.chalmers.se)

Abstract

The paper presents the abstraction transformation which is a fundamental method for creating functions in ADATE. The use of abstraction turns out to be similar to evolution by gene duplication which is emerging as the most important theory of "building blocks" in natural genomes. We discuss the relationship between abstraction and its natural counterparts, but also give novel technical details on automatic invention of functions. Basically, abstraction is the reverse of the inlining transformation performed by optimizing compilers.

1 Introduction

This paper describes a general and effective technique, called abstraction, for inventing new functions on-the-fly during automatic evolution of programs as in ADATE [8, 9]. In 1991, we invented and designed abstraction for use in ADATE, but expect that abstraction is useful also in other evolutionary computation paradigms such as GP [5] and possibly also EP [3].

Modern genetics research has shown that the genetic operators employed by nature are quite sophisticated and that a model only including for example point mutation and crossover is much too simple. Since natural evolution is the most successful form of evolutionary computation so far, it is natural to compare proposed genetic operators with their counterparts in nature. Therefore, we will below discuss if abstraction and substitution in ADATE have roles similar to gene families, repetitive DNA and RNA editing through substitution [4].

In order to explain abstraction in an easy way, we will actually start by describing how it would operate on DNA. The later sections of this paper give technical descriptions of abstraction as a program transformation and show that it is a fundamental and mathematically elegant operation.

As more and more genes are sequenced, it becomes increasingly clear that Susumo Ohno's theory of evolution by gene duplication [7] is correct. It is beyond the scope of this paper to delve into current genome research where new

gene families are discovered quite often. For example, Tadashi Imanishi [2] has a list of 425 families in the human genome, where the genes in each family are so homologous that they very likely share a common origin.

We will now introduce abstraction by studying the process where a sequence of DNA is copied to a new location in the genome and then diverges from the original through mutations.

For example, assume that we somewhere in the genome have the subsequence of base pairs

$$\texttt{GGATTCTGG},$$

which would be translated to the three amino acids glycine, phenylalanine and tryptophan. Assume that this sequence is copied and then mutated by changing the first `T` to `C` and that the complete genome becomes

$$\ldots\texttt{GGATTCTGG}\ldots\texttt{GGACTCTGG}\ldots,$$

where the right-most subsequence is a mutated copy of the left-most subsequence.

Such repetitive and almost equal building blocks can be described using a function definition where the parameters indicate the variation between the different copies of a block. The building block in the above genome, for instance, corresponds to the function definition

$$f(x) = \texttt{GGA}x\texttt{TCTGG},$$

which allows the complete genome to be written as

$$\ldots f(\texttt{T})\ldots f(\texttt{C})\ldots,$$

where the original subsequence is $f(\texttt{T})$ and the mutated copy is $f(\texttt{C})$.

Note that a human genome may contain a million calls to a function like f if f represents a so-called short interspersed element (SINE), some of which occur about a million times in the human genome! This function is then a small help function that is useful over and over again. However, the role of SINEs is still unclear.

Let us now repeat the above copy-and-mutate process using function invention from the beginning. To simplify this introductory presentation, assume that the function to be invented has exactly one parameter and that an actual argument is only one base pair i.e., `A`, `C`, `T` or `G`.

Given that the function definition is to be based on the subsequence `GGATTCTGG`, there is one possible function definition for each of the nine positions in the subsequence. These possibilities are

$$f(x) = x\texttt{GATTCTGG}$$
$$f(x) = \texttt{G}x\texttt{ATTCTGG}$$

$$f(x) = \texttt{GG}x\texttt{TTCTGG}$$
$$f(x) = \texttt{GGA}x\texttt{TCTGG}$$
$$f(x) = \texttt{GGAT}x\texttt{CTGG}$$
$$f(x) = \texttt{GGATT}x\texttt{TGG}$$
$$f(x) = \texttt{GGATTC}x\texttt{GG}$$
$$f(x) = \texttt{GGATTCT}x\texttt{G}$$
$$f(x) = \texttt{GGATTCTG}x$$

An abstraction is carried out by choosing one of these nine alternatives and then rewriting the "genome" accordingly. Using functional programming notation [12], we can for example obtain the "genome"

let
  $\texttt{fun}\ f(x) = \texttt{GGA}x\texttt{TCTGG}$
in
  $\ldots f(\texttt{T}) \ldots$
end

We say that GGAxTCTGG is the body of f whereas the original GGATTCTGG is the pre-body of f.

Note that abstraction is an equivalence-preserving operator which means that it does not change fitness. In the simple example above, the number of possible abstractions is linear in the length of the pre-body, but in practice, the number of possibilities is occasionally quadratic or even cubic in the size of the pre-body. Since ADATE sometimes may need to examine a substantial fraction of all possible abstractions, we require that an abstraction shows its fitness advantage as soon as possible so that irrelevant abstractions are eliminated quickly.

In ADATE, this quick demonstration of usefulness is accomplished by one or more REQ and R transformations [9] that immediately follow an abstraction.

Very roughly speaking, this would correspond to the insertion of $f(\texttt{A})$, $f(\texttt{C})$, $f(\texttt{T})$ or $f(\texttt{G})$ somewhere in the above "genome", for example yielding

let
  $\texttt{fun}\ f(x) = \texttt{GGA}x\texttt{TCTGG}$
in
  $\ldots f(\texttt{T}) \ldots f(\texttt{C}) \ldots$
end

Note that this genome is equivalent to the one obtained through the above copy-and-mutate process that did not use abstraction, but the copied sequence

is now easer to identify and reuse since it is the body of a function. Since functions produced by ADATE often are recursive, a copy-and-mutate operator could not replace abstraction.

In ADATE, abstraction is a fundamental and highly useful program transformation that has proven its value over and over again since we introduced it in 1991. Most of the examples on the ADATE web site [10] would not work without it. During large scale program evolution, abstraction is essential for at least the following two reasons.

1. The user of an automatic programming system should not be required to define all needed help functions. Instead, the user should define a small number of primitives whereas the system automatically constructs a possibly large number of help functions.

2. The system can construct a help function exactly where it is needed and avoid having a too large scope for the function.

Scope restriction is important when human beings write programs but much more so in automatic programming based on combinatorial search. If many irrelevant functions are available at a given position in a program, combinatorial search for program improvements at that position becomes harder.

A form of scope restriction actually seems to exist in DNA since repeats often occur in localized regions. One of several examples in [4] is the clustering of multiple copies of genes encoding ribosomal RNA in humans. One can speculate that the probability that a copy will become located at a given position is inversely proportional to the distance between that position and the position of the original. Such probabilistic scoping could also be useful in ADATE, GAs and GP. Note that ADATE already has a variant of such a scoping scheme, but its usefulness has not yet been experimentally verified since synthesized programs are too small.

2 Abstraction Applied to Functional Program "Genomes"

Abstraction is in principle applicable to practically all kinds of "genomes", including binary strings as in GAs, machine code as in Peter Nordin's work [6] and functional programs as in ADATE. We will now use a small and pedagogical example to show how abstraction operates on functional programs.

Assume that we want ADATE to synthesize a function `max3` that finds the greatest of three integers, which is a simple program indeed. The specification contains a number of input-output pairs where each input is a tuple with three integers and the corresponding output is one of these three. The only primitive in the specification is `<`.

Using ADATEs a-little-bigger-a-little-better population management [11], the following program will typically become a permanent member of the population

```
fun max3( X, Y, Z ) = if X < Y then Y else X
```

ADATE would actually use case-analysis instead of an if-expression, but we use `if` here to improve readability.

Assume that the entire body of `max3` is chosen as the pre-body of an abstraction of arity (number of parameters) one, two, three or four, which gives many alternatives, for example

```
fun max3( X, Y, Z ) =
  let
    fun g( V ) = if V < Y then Y else X
  in
    g( X )
  end
```

This abstraction and most others are not useful, but the following is.

```
fun max3( X, Y, Z ) =
  let
    fun g( V1, V2 ) = if V1 < V2 then V2 else V1
  in
    g( X, Y )
  end
```

ADATE has substituted `V1` for `X` and and `V2` for `Y`. An almost trivial R transformation [9] then inserts `g( _, Z )` and yields the following 100% desirable program

```
fun max3( X, Y, Z ) =
  let
    fun g( V1, V2 ) = if V1 < V2 then V2 else V1
  in
    g( g( X, Y ), Z )
  end
```

Thus, it was easy for this abstraction to very quickly prove its usefulness.

This example shows that abstractions can be substitutive which means that a number of equal subexpressions are replaced with one and the same parameter. These equal subexpressions can be arbitrarily big and need not consist of a single variable as above.

Since substitution is a very fundamental operation in mathematics and computer science, we were not surprised to find that it also occurs in natural genetics. Simply speaking, DNA is transcribed to messenger RNA (mRNA) which is translated into proteins. However, RNA is edited in complex and still mysterious ways [4] before it is translated to proteins. For example, editing can use so-called guide RNA (gRNA) templates and seems capable of performing operations similar to systematic substitution and possibly more advanced operations that remain to be discovered.

From a computer science point of view, abstraction is related to inlining [1] which is a common transformation in compilers that expand a function call by replacing it with a properly instantiated copy of the body. Abstraction tries to find the function definition by reversing this process.

3 The Algorithm for the Abstraction Transformation

A naive algorithm for abstraction is quite simple and would just produce all alternative abstractions of a given arity and consider them all to be equally important. For a given pre-body represented as an expression tree consisting of s nodes, there is never more than $\binom{s}{a}$ non-substitutive alternative abstractions if a function of arity a is to be invented. Even if a typically is very small and the number of alternatives due to various constraints is considerably less than $\binom{s}{a}$, the number of alternatives grows sufficiently quickly with s to motivate heuristics that give more weight to abstractions that are more likely to be useful.

3.1 Weighted Search

ADATE iteratively deepens [8] a so-called cost limit l that says how many children programs that are to be produced from a given parent program. An abstraction is assigned a cost c which indicates that l/c programs are to be based on the abstraction. For example, if l =5000 and an abstraction has cost 50, this abstraction will be allowed 100 children to prove its usefulness. Thus, an abstraction that heuristically seems to be more likely to lead to a fitness improvement should be assigned a lower cost.

There are many choices that need to be made before an abstraction is fully specified e.g., the arity, the pre-body and the subexpressions to parameterize. The cost of an abstraction depends on the sequence of choices that the abstraction algorithm had to make before producing it. This sequence represents a root-to-leaf path in an implicit decision tree with the original program in the root and programs containing invented functions in the leaves.

Each branch in the decision tree is labeled with a weight that is multiplied with the cost limit of the parent node to obtain the cost limit of the child node. Like a probability, a weight is a number between zero and one. The sum of all weights on the out-going branches of a node is one.

If $w_1, w_2, \ldots, w_{\#w}$ are all the weights along a root-to-leaf path, the cost of the abstraction corresponding to the leaf is $1/w_1/w_2/\ldots/w_{\#w}$.

Example. Assume for a moment that there only is a choice between arity one and arity two and that arity one has weight 0.7 whereas arity two has weight 0.3. If the initial cost limit is 10000, ADATE would produce a total of 7000 programs based on abstractions of arity one and a total of 3000 programs based on abstractions of arity two. □

An important and difficult question is how to determine the weights. One approach is to have a large set of different examples, at least a few hundred, and

repeatedly run ADATE on the set with different weights to see how the total run time for the set changes. It would be possible to automatically optimize the weights in this way, but enormous CPU resources would be required since even a single run on a single example may take a few days on a single processor.

Our approach was to study the relative frequency with which a given branch seems to result in a fitness improvement, but the weights we obtained should be considered as ad hoc values that are unlikely to be optimal. Since we used only 12 examples , the statistical sample was so small that we "equalized" or "smoothed" the weights which means that for example arity eight is assigned a small weight even though it does not appear in any of the experiments.

3.2 Positions and Substitutions

In order to identify a node in an expression tree, ADATE uses a so-called position which is a list of natural numbers `[` i_1, i_2, ..., $i_{\#i}$ `]` specifying how to walk to the node if you start at the root. Each i_k specifies that the next node to walk to from the current node is child number i_k. The left-most child has number 0. For example, the position of `Y` in `g1( X, g2( Y, Z ) )` is `[ 1, 0 ]`.

We say that the position of a pre-body is a *top* position whereas the position of a subexpression to be parameterized is a *bottom* position. For example, the first abstraction in section 2 uses the top position `[]` and the bottom position `[ 0, 0 ]`.

Recall that a single parameter may replace several equal subexpressions in the pre-body through substitution. For example, in the second abstraction in Section 2, the bottom positions for `V2` are `[ 0,1 ]` and `[ 1 ]` . The list of positions associated with a given parameter is said to be an *alternative*. Note that the number of alternatives to be chosen equals the arity.

If two alternatives `[` $P_1, \ldots, P_{\#P}$ `]` and `[` $Q_1, \ldots, Q_{\#P}$ `]` are chosen, no P_i is allowed to be a prefix or a suffix of any Q_k since this would lead to an attempt to parameterize the same subexpression occurrence more than once.

Assuming that the pre-body contains n equal subexpressions $E_1, \ldots, E_n$, the number of alternatives only containing positions of E_i's is 2^n, which means that trying all possible substitutions may cause combinatorial explosions. However, we have only observed substitutions that replace all E_i's with a common ancestor and never so far seen any use for substitutions restricted to arbitrary subsets of E_i's.

Therefore, ADATE only considers substitutions that replace all occurrences in a given subtree. With this heuristic, it can be shown that the maximum number of extra alternatives due to substitution is $s - \log_2(s+1)$, where s is the size of the pre-body i.e., the number of nodes in the pre-body.

Example. Figure 2 shows all 18 alternatives produced by ADATE for the pre-body in Figure 1. The four last alternatives are substitutive i.e., replace two or more equal subexpressions. The last alternative, for example, replaces the two last occurrences of `As`. □

The choice of arity, the choice of top position and the choice of alternatives are discussed in the following three subsections.

```
case As of nil => nil | cons( A1, As1 ) =>
case As1 of nil => As | cons( A2, As2 ) =>
case Bs of nil => As | cons( B1, Bs1 ) =>
case g1( As1, Bs1 ) of nil => nil | cons( C1, Cs1 ) => Cs1
```

Figure 1: A sample pre-body.

```
[ [0] ]              [ [1] ]                  [ [2] ]
[ [2,0] ]            [ [2,1] ]                [ [2,2] ]
[ [2,2,0] ]          [ [2,2,1] ]              [ [2,2,2] ]
[ [2,2,2,0] ]        [ [2,2,2,0,0] ]          [ [2,2,2,0,1] ]
[ [2,2,2,1] ]        [ [2,2,2,2] ]            [ [0], [2,1], [2,2,1] ]
[ [1], [2,2,2,1] ]   [ [2,0], [2,2,2,0,0] ]   [ [2,1], [2,2,1] ]
```

Figure 2: The alternatives for the pre-body in Figure 1.

3.3 The Choice of Arity

In the current version of ADATE, the weights of arities are chosen according to the following list of (arity, weight) pairs.

```
val Arity_weights =
  [ ( 8, 0.02 ), ( 7, 0.02 ), ( 6, 0.02 ), ( 5, 0.02 ), ( 4, 0.04 ),
    ( 3, 0.08 ), ( 0, 0.20 ), ( 2, 0.28 ), ( 1, 0.32 ) ]
```

For example, arity zero is assigned the weight 0.20. An arity zero abstraction is a special case where the invented function actually becomes a local constant with the same value as the pre-body.

Please note that ADATE does not need to introduce all parameters immediately and can increase arity as needed using the embedding transformation [8, 9]. The newest version of ADATE also uses abstraction to increase the number of parameters of an existing function. Combinatorially, this is much cheaper than the embedding transformation in [8, 9]. However, we do not have space here for a discussion of embedding.

3.4 The Choice of Top Position

Recall that a cost limit l says how many programs that are to be produced. If there are many choices of top positions and many alternatives for each choice, the cost of an abstraction may become too high and exceed l. Therefore, it is often necessary to only try a subset of all possible top positions. Like most ADATE transformation algorithms, the abstraction algorithm has a built-in "small-is-beautiful" preference, meaning that a smaller pre-body typically is preferred to a bigger. For a given arity, we will now describe what subset of top positions that ADATE chooses.

An abstraction is sometimes coupled to a previous transformation [8, 9] which means that not all positions in the program are to be allowed as top positions. The abstraction algorithm is given a `top_pos_ok` predicate as argument. This predicate says if a given position is allowed and is the first constraint on the positions to be chosen.

For a given top position P, assume that $n_a(P)$ is the number of legal combinations of a alternatives where a is the arity. The cost of an abstraction is chosen to $n_c \cdot n_a(P)$, where n_c is the total number of chosen top positions. Basically, n_c is increased as long as $n_c \cdot n_a(P) \leq l$ for all chosen P. The specific algorithm is as follows.

All positions that satisfy the `top_pos_ok` predicate are classified into three groups, namely chosen positions, candidate positions and other positions. A position is said to be chosen iff ADATE has decided to use it as a top position. A candidate position is such that all its children have been chosen. Initially, there are no chosen positions and only the leaves are candidates.

As soon as a position P becomes a candidate, $n_a(P)$ is computed and stored in a priority queue organized in order of increasing ($n_a(P)$, $s(P)$) where $s(P)$ is the pre-body size.

Assume that n_c is the current number of chosen positions. The first position P in the queue is chosen repeatedly until $n_c \cdot n_a(P) > l$. The reason that the priority queue uses $s(P)$ as a secondary key is primarily that $n_a(P)$ is independent of P when the arity is zero.

3.5 The Choice of Alternatives

The choice of alternatives uses fairly simple heuristics and weighting but still requires a carefully designed algorithm to avoid bad combinatorial properties.

A simple heuristic rule is that the size of the body except alternatives must be at least two. Another rule is that a bottom position never is allowed to specify a so-called dont-know constant [9].

However, the most important heuristics is weighting based on the usage of parameters. The main principle is to discriminate against bodies containing if-tests that do not depend on any parameter. Since ADATE uses `case` instead of `if`, we will formulate this principle for case-expressions.

A case-expression in a body is considered to be *static* if it has at least two activated rules and the expression it analyzes does not depend on any parameter of the function being invented. Let C be the set of all case-expressions in a body that do not occur in a function definition local to the body. The body and the abstraction that introduced it are classified as static if C is empty or at least one case-expression in C is static. An abstraction that is not static is said to be *dynamic*.

The weight 0.7 is allocated to dynamic abstractions whereas 0.3 is allocated to static abstractions. If there are no abstractions of one kind (static or dynamic), the weight 1.0 is allocated to the other kind.

For a given top position, dynamic abstractions are synthesized by "covering" a dependency graph that models the relationship between case-expressions. The

graph is directed, acyclic and has exactly one node for each case-analyzed expression such that the corresponding case-expression has at least two activated rules.

Static abstractions could be produced by ensuring that the graph is not covered, but this is unnecessary since static abstractions in general are substantially easier to produce than dynamic abstractions. Therefore, it is sufficient with a straightforward combinatorial search that produces all abstractions and eliminates the dynamic ones.

Example. Consider the pre-body in Figure 1. The four nodes in the dependency DAG correspond to the case-analyzed expressions `As`, `As1`, `Bs` and `g1( As1, Bs1 )`. For example, there is an edge from `Bs` to `g1( As1, Bs1 )` since the latter expression directly depends on the former. The edges in the DAG are (`As`, `As1`), (`As`, `g1( As1, Bs1 )`) and (`Bs`, `g1( As1, Bs1 )`). □

An abstraction is dynamic iff it *covers* all roots of the DAG without eliminating all case-expressions from the body. We say that a bottom position covers a root iff it is an ancestor or a descendant of the root.

ADATE does not actually produce the DAG. Instead, it finds the roots directly for a given top position using a preorder traversal, remembering the case-rule variables introduced so far. A case-analyzed expression is a root iff it doesn't contain any of these variables. The preorder traversal does not enter *Decs* in a sub-expression of the form

`let` *Decs* `in` *E* `end`

3.5.1 How to Choose Bottom Positions for Dynamic Abstractions

A *root alternative* is such that it covers at least one root i.e., contains at least one position which is an ancestor or a descendant of at least one root.

Example. The pre-body in Figure 1 has two roots, namely `As` at position `[0]` and `Bs` at position `[2,2,0]`. In Figure 2, the five root alternatives are

```
[ [0] ]   [ [2] ]   [ [2,2] ]   [ [2,2,0] ]
[ [0], [2,1], [2,2,1] ]
```

Assuming that the arity a is three, the following table shows some combinations of root alternatives and other alternatives. Note that each combination of root alternatives eliminates all roots from the body.

Root choices		Other choices
`[ [0] ]`	`[ [2,2] ]`	`[ [1] ]`
`[ [0] ]`	`[ [2,2] ]`	`[ [2,0] ]`
`[ [0] ]`	`[ [2,2] ]`	`[ [2,1] ]`
`[ [0] ]`	`[ [2,2,0] ]`	`[ [1] ]`
`[ [0] ]`	`[ [2,2,0] ]`	`[ [2,0] ]`
`[ [0] ]`	`[ [2,2,0] ]`	`[ [2,1] ]`
`[ [0] ]`	`[ [2,2,0] ]`	`[ [2,2,1] ]`

□

ADATE first chooses root alternatives so that each root is covered and then chooses other alternatives. Note that one or more of the root alternatives are allowed to be redundant in the respect that they are not needed to cover all roots.

Deciding if a cardinality a subset of the alternatives suffices to cover all roots is an instance of the NP-complete MINIMUM COVER problem. Therefore, it is difficult to find an algorithm with a run time that is polynomial in a in the worst case and that decides if it is possible to cover all roots using an arity-a abstraction on a given pre-body.

If there are n root alternatives and ancestor-descendant relationships are ignored, the number of possible choices of root alternatives is $\binom{n}{a}$. If both the pre-body and the number of roots are big and the roots are not localized to a small part of the pre-body, there might not be any covering choice of roots.

If it is not possible to finish, the algorithm for making root choices tries to detect this early. If the arity left is a', the algorithm checks the number of roots taken out by each remaining alternative and computes the sum for the a' alternatives that take out most roots. If the number of uncovered roots exceeds this sum, it is impossible to finish.

4 Conclusions

The experimental data available through [10] show that function invention through abstraction is both indispensable and effective in ADATE. Therefore, it was fascinating to discover that nature already uses similar tricks and that these are in the process of being unraveled through ongoing gene sequencing efforts. In particular, sequencing has uncovered vast amounts of repetitive DNA and the prevalence of homologous genes sharing a common ancestor gene, which shows the importance of gene duplication in natural evolution.

The weighted iterative-deepening search presented above is unlikely to be optimally adapted to the statistical properties of functional programming e.g., the initial arity of a function when the function is first conceived. However, we believe that the average number of abstractions produced before finding one that is useful is within a small factor, say three, of the optimum on a large set of examples.

Since our first publications on ADATE [8, 9], the implementation of abstraction has been refined with substitution and built-in production of dynamic abstractions, which is particularly important for large pre-bodies. We have also expanded the range of initial arities and successfully employed abstraction mechanisms to perform embedding.

Abstraction is fundamental in ADATE and very likely in most forms of large scale evolution, including the development of life on Earth.

References

[1] A. W. Appel, *Modern compiler implementation in ML*, Cambridge University Press, 1998.

[2] T. Endo, T. Imanishi, T. Gojobori and H. Inoko, Evolutionary significance of intra-genome duplications on human chromosomes, *Cell*, number 205, December 31, 1997, pp. 19–27.

[3] D. B. Fogel, *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*, IEEE Press, 1996.

[4] W. S. Klug and M. R. Cummings, *Essentials of genetics*, Prentice-Hall, 1999.

[5] J. R. Koza, *Genetic programming: on the programming of computers by means of natural selection*, MIT Press, 1992.

[6] J. P. Nordin, *Evolutionary program induction of binary machine code and its applications*, Krehl Verlag, Münster, 1997.

[7] S. Ohno, *Evolution by gene duplication*, Springer-Verlag, 1970.

[8] J. R. Olsson, Inductive functional programming using incremental program transformation and Execution of logic programs by iterative-deepening A* SLD-tree search, Research report 189, Dr scient thesis, ISBN 82-7368-099-1, University of Oslo, 1994.

[9] J. R. Olsson, Inductive functional programming using incremental program transformation, *Artificial Intelligence*, volume 74, number 1, March 1995, pp. 55–83.

[10] J. R. Olsson, Web page for Automatic Design of Algorithms through Evolution, http://www-ia.hiof.no/~rolando/adate_intro.html (current Nov. 23, 1998).

[11] J. R. Olsson, Population management for automatic design of algorithms through evolution, *International Conference on Evolutionary Computation*, 1998.

[12] Å. Wikström, *Functional Programming Using Standard ML*, Prentice Hall International, 1987.