Program Optimization for Faster Genetic Programming[*]

Bradley J. Lucier
Purdue University
1395 Mathematical Sciences Building
W Lafayette, IN 47907
Email: lucier@math.purdue.edu.
Phone: (765) 494–1979.

Sudhakar Mamillapalli
Purdue University
Dept of Agricultural Engineering
W Lafayette, IN 47907
Email: sudhakar@ecn.purdue.edu
Phone: (765) 494–1196

Jens Palsberg
Purdue University
Dept of Computer Science
W Lafayette, IN 47907
Email: palsberg@cs.purdue.edu.
Phone: (765) 494–6012

ABSTRACT

We have used genetic programming to develop efficient image processing software. The ultimate goal of our work is to detect certain signs of breast cancer that cannot be detected with current segmentation and classification methods. Traditional techniques do a relatively good job of segmenting and classifying small-scale features of mammograms, such as micro-calcification clusters. Our strongly-typed genetic programs work on a multi-resolution representation of the mammogram, and they are aimed at handling features at medium and large scales, such as stellated lesions and architectural distortions. The main problem is efficiency. We employ program optimizations that speed up the evolution process by more than a factor of ten. In this paper we present our genetic programming system, and we describe our optimization techniques.

1 Introduction

Genetic programming [14] has been applied widely in image processing [20, 16, 6, 5]. For example, Harris and Buxton [11] applied genetic programming techniques to derive high performance edge detectors for one-dimensional signals. The resulting programs often compared favorably with hand-written edge detectors.

The ultimate goal of our work is to detect certain signs of breast cancer that cannot be detected with current methods. Traditional techniques do a relatively good job of segmenting and classifying small-scale features of mammograms, such as micro-calcification clusters [8]. We want to handle features at medium and large scales, such as stellated lesions and architectural distortions [19].

Poli [17] discussed genetic programming for image analysis, and he noted that without restrictions on which pixels can be accessed, the search space becomes huge. His paper presents a particular set of restrictions together with some experimental data. Still, his paper does not evaluate whether the chosen restrictions actually give an advantage over working in an unrestricted setting.

In this paper we study edge detection in 512×512 images with a multi-resolution representation [3]. Edge detection is a long-studied problem with many classical solutions [10]. We use strongly-typed genetic programs with (1) unbounded size, (2) unbounded integers, and (3) unrestricted access to all pixels. We show that a good edge detector can be evolved in a reasonable amount of time provided that aggressive program optimizations are employed.

In our setting, a genetic program maps a pixel index to a boolean value, indicating whether the pixel is an edge pixel or not. We use four well-known source-to-source optimizations [1] that together speed up the evolution process by at least a factor of ten. The optimizations are: 1) checks for out-of-boundary conditions; 2) variable bindings of the needed pixel values before evaluating the expressions; 3) use of fixnum (small integer) arithmetic wherever possible; and 4) flattening of and/or subtrees of expressions. The use of fixnum arithmetic seems to be the most effective optimization, while flattening and-or subtrees is the least effective. We represent the genetic programs as LISP-lists, and after the optimizations, we use off-the-shelf tools to compile the programs to C, and to compile and execute the generated C programs. Our experiments show that the optimizing C compiler alone is far from matching the speed-ups we get from the source-to-source optimizations. This confirms that powerful program optimizations are easier to do for high-level programs. We conclude that program optimization is a viable way to reduce the time needed for the massive computations for image-processing genetic programming.

In the following section we describe the language of genetic programs and the evolution process, in Section 3 we discuss our optimization techniques, and in Section 4 we present our experimental results.

[*]To appear in Proc. GP'98, Genetic Programming, Madison, Wisconsin, July 1998.

2 The Genetic Programs

Our genetic programs are represented as LISP-lists generated from the grammar:

Exp	::=	`point`
	|	*Constant*
	|	`(` *UnaryOp Exp* `)`
	|	`(` *BinaryOp Exp Exp* `)`
Constant	::=	$-4 \mid -3 \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid 3 \mid 4$
UnaryOp	::=	`abs` | `divide-by-2` | `not` | `value`
	|	`left` | `right` | `up` | `down`
	|	`deeper` | `shallower`
BinaryOp	::=	`+` | `-` | `*` | `<` | `and` | `or`

There are three types of data: indices, integers, and booleans. An index is defined by a triplet of integers, representing the row, column, and depth. Depth 0 corresponds to the pixels of the original image. A depth of 1 is obtained by aggregating 2×2 squares of pixels as showed in Figure 1. Assuming the figure represents a matrix, the value of the (2×2) pixel at row 1 and column 1 at depth 1 is obtained by averaging the values of the pixel at row 1, column 1; one-half the values of pixels adjacent to the four edges of the pixel at row 1, column 1; and one-fourth the values of the four pixels adjoining the corners of the pixel at row 1, column 1. Similarly a image at depth 2 is obtained by averaging the original image over 4×4 squares, each centered at one of the original pixels. Thus, one can view the original pixel data as being transformed into a pyramid.

There are six operations that map an index (i, j, k) to an index:

1. `left` returns the index of the location 2^k pixels left of the present position. For example if the current index has row 10, column 10 and depth 2, then `left` returns an index with row 10, column 6 and depth 2. If part of the $2^k \times 2^k$-pixel subsquare associated with this index lies outside the image, `left` returns an "out of bounds" condition.

2. `right`, `up`, and `down` return the index obtained by moving right, up, or down, respectively, with respect to the current location. Again, if this operation leads to moving out of the image then "out of bounds" is returned.

3. `shallower` increases the depth by one (in effect doubling the size of the data square), and `deeper` decreases it by one. If the operation results in moving out the the image, an "out of bounds" condition is returned. For example, `deeper` does not make sense at depth 0, since in our interpretation of the data, negative depth would imply sub-pixel resolution.

These six operations are potentially slow because of the checks for "out of bounds." In Section 3.1 we show how they, in many cases, can be executed efficiently.

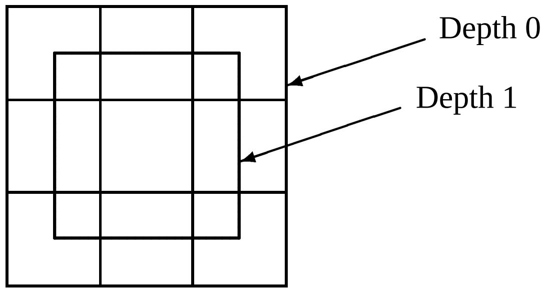

Figure 1: Depths of an Image

Our language of genetic programs is sufficient to generate any one of the family of biorthogonal wavelets [7, p.272]. These wavelets have arbitrarily high approximation power even though they are constructed using knowledge only of the averages of pixels on dyadic squares; thus, they can form a general framework for multi-resolution data representation. It is possible that an evolving genetic programming system will choose the order of representation that is best for a particular problem (e.g., first order for edge detection, second order for curvature analysis, etc.).

We only work with genetic programs that type check [2]. For example, the initialization process and the genetic operators generate only programs that type check. This approach to genetic programming was pioneered by Montana [15], see also [12]. Early genetic programming systems worked with the simplifying assumption, known as *closure*, that all variables, constants, arguments for functions, and values returned from these functions must be of the same data type.

A genetic program in our language takes a row and a column as input and return true or false indicating if that corresponding pixel in the image is an edge or not. The index of that pixel will look like $(i, j, 0)$, and it is denoted by the constant `point`. The function `value` returns for each index (i, j, k) the average value of the pixels on a subsquare of the image consisting of $2^k \times 2^k$ pixels centered at the pixel at row i and column j.

For each input image, we have created a companion image which contains just the edges. The goal of the genetic programming process is to evolve a program which for each image produces the corresponding edge image.

Each generation is composed of 2000 programs. Initially, programs are randomly generated (generation 0) and are applied to every pixel in the image. The thereby computed images are then compared to the "true" edge image. The number of discrepancies between the "true" and computed images are noted for both the set of edge pixels and the set of non-edge pixels of the true image; the penalty is the sum of the squares of the errors on these two sets. Based on this penalty the traditional genetic programming methods like cross over, mutation etc. are applied and the programs are successively refined in succeeding generations in order to achieve a target penalty. In this case, the target penalty is determined as follows. First a program is generated which detects no edges at all. The penalty of this program with respect to the true edge program is calculated and then the target penalty is set to $1/10^{\text{th}}$ of this value. After each generation the penalty is calculated for each of the programs. If in a particular generation a program achieves the target penalty then the evolution is terminated, otherwise the evolution stops after 100 generations.

We do not copy programs from one generation to the next. After the initial generation, all programs are created by cross over (thus, the cross over probability is 1.) For each training set, we do just one evolution. All mutation happens during cross over, with a probability of 5%. Intuitively, the cross over takes place with a 5% chance of a copy error. There are four different mutations which all insert some code: (1) “rotate 90 degrees,” (2) “reflect in the x-axis,” (3) “reflect in the y-axis,” and (4) “make shallower.” They happen with the probabilities (1) 1/3, (2) 1/6, (3) 1/6, and (4) 1/3 out of the 5% chance that a mutation will happen.

Our genetic programming system is implemented in Scheme [4] (a dialect of LISP) using Gambit-C [9], a version of the Gambit system that generates portable C code, and Meroon [18], which implements an object system on top of Scheme. Gambit-C is used to convert the generated programs into C. We use gcc with compiler optimization setting `-O1` to compile the generated C code. The setting `-O2` turns out be an inferior choice because the compile time increases more than the decrease in cumulative run time of the compiled code. Notice that although gcc attempts to do the same style of optimizations as we do at the source level, it does a much poorer job because much of the program structure is lost when reaching the C level.

3 Optimizations

3.1 Check for “Out of Bounds” Conditions

If an index-to-index operation moves an index out of the boundary of the image during the execution of a program with the argument `point`, then the program should immediately return ”don’t know” instead of ”true” or ”false” as to whether `point` is an edge pixel. A naive implementation performs a boundary check before each index-to-index operation. In contrast, our implementation performs a static analysis to determine a condition for when a given program never moves out of the boundary, and this condition is checked *before* each boolean clause of the program is calculated. If the check succeeds, then we can use index-to-index operations *without* boundary checks, and if it fails, then an error can immediately be reported. Being able to omit the boundary checks significantly speeds up execution. For example, if the current position has a column greater than four, then four `left` operations will never result in a position out of the image. Before execution we therefore check that the column is greater than four. In effect, we place a compact representation of all the boundary checks at the beginning of the execution instead at each individual index-to-index operation. Intuitively, this optimization transforms a program `exp` into

```
if condition then exp else error
```

where `exp` can be executed faster.

3.2 Variable Bindings

The `value` operation may be applied many times during a computation. A naive implementation performs `value` exactly when specified in the program text. In contrast, our implementation performs a static analysis to determine how the argument to `value` is offset from from the current index. Intuitively, we then replace the call `(value arg)` by `(value (make-index row column depth))` where there `row`, `column`, and `depth` have been determined from `arg`, and `make-index` is an auxiliary function which creates an index. For example, we can optimize the program

```
(< 1 (value (down (down (down point)))))
```

into

```
(< 1 (value (make-index (+ r 3) c d)))
```

where `r`, `c`, `d` are the current row, column, and depth. This optimization allows us to eliminate all index-to-index operations.

Sometimes the same value is needed in more than one place. Consider the following program:

```
(< (value point) (+ (value point) 1))
```

A naive implementation looks up twice the pixel at index `point`. In contrast, our implementation does common subexpression elimination, and transforms, intuitively, the program into:

```
(let ((g (value point))) (< g (+ g 1)))
```

Thus, the `value` operation is only executed once instead of twice.

Together, the two optimizations in this subsection transforms a program `exp` into

```
(let ((g1 (value ...))
      ...
      (gn (value ...)))
  compact-exp)
```

where `compact-exp` does not use index-to-index operations, or the `value operation`.

3.3 Fixnum Arithmetic

Our programs compute with unbounded integers. A naive implementation performs all integers operations on a representation which supports unbounded integers. In contrast, our implementation performs a static analysis to eliminate some expressions entirely, for example, a comparison of two constants, and otherwise to insert fixnum, or small integer, operations wherever possible. Particularly useful is the property that the `value` operation always returns an integer between 0 and 255. In essence, this optimization is a restricted form of partial evaluation [13]. Our static analysis is based on integer interval arithmetic. An alternative would be to use a more precise analysis based on Tarski's theorem, but this has a time complexity which is doubly exponential in the size of the program.

3.4 Flattening and/or

Consider the following expression:

```
(and (or A B)
     (and C D))
```

One can assume that the boundary checks and variable bindings for `A` must be computed before those for `C` and `D`, so we can use them in computing `C` and `D`. However, `B` may not be computed, so we cannot rely on `B`'s boundary checks or variable bindings in `C` and `D`. On the other hand, in:

```
(and (and A B)
     (and C D))
```

we *do* know that `B` must be computed before `C` and `D`. To make this clear to our optimizer, we rewrite subtrees consisting solely of `and` expressions or solely of `or` expressions as, for example:

```
(and A
     (and B
          (and C D)))
```

In this case `C` and `D` are evaluated in the environment of `B`, so some boundary checks and variable bindings can be avoided. We call this "flattening" of and/or trees. We apply this optimization before the others.

4 Experimental Results

It takes time to carry out an optimization. We are interested in optimizations for which the time to carry them out is smaller than the reduction in run-time of the programs. Our experiments show how much time each optimization gains or loses. Our approach is to add one optimization at a time.

We used three images, of a bank, Lenna, and an F16, see Figures 2.1, 3.1 and 4.1. The true edge pictures for each of these are given in Figures 2.2, 3.2 and 4.2. Genetic programming was applied to derive programs which could determine the edges successfully. The target penalties used were 1/10th the penalty of the program which could not determine any of the edges for Lenna and Bank, while it was chosen to be 1/12th for the F16 image since using 1/10th the value did not give satisfactory results for this image. Figures 2.3, 3.3 and 4.3 give the results obtained; by comparison with the actual edge figures the programs seem to be successful in determining most of the edges. Table 1 summarizes the penalties and the number of generations that were necessary to obtain the results for each of these images. The Lenna image, being the most complicated, took 52 generations to attain the target penalty while F16 took only 20 generations. In order to test the robustness of the generated program, the edge detection program of Lenna was applied to both Bank and F16 and the images are given in Figures 5.1 and 5.2. Both are close to the actual edge images, which gives us confidence on the robustness of the generated program. The penalties obtained with application of the program generated for Lenna on F16 was 0.9×10^8, which is less than 1.0×10^8, the minimum penalty obtained for F16 at generation 20 (Table 1). However, the penalty obtained when applied to the Bank image is 2.2×10^8, slightly higher than 1.8×10^8 that was obtained when the program generated using the Bank image itself was used.

The time taken with no optimization, and with each of the four optimizations incrementally applied is given in Table 1. The time taken is the sum of user and system times necessary to obtain the target penalty. This time includes the source-to-source optimizations (where applied), the Scheme$\rightarrow$C translation, the compilation of the C code, and running the resulting programs.

Table 1 shows that the optimizations speed up the evolution process by more than a factor of ten. The optimizations are not equally effective. Flattening and/or's do not help much. The savings is hardly one hour but still the runtime decreased (and the optimization takes only about 10 lines of code to implement.) Use of fixnum arithmetic resulted in big savings. For example, for the Bank image with fixnum arithmetic, the runtime was 19.8 hours while without it was 136 hours. Also adding variable bindings and boundary checks caused a significant decrease in runtime. Boundary checks and variable bindings together decreased further the runtime by 2.5 times (from 350 hours to 136 hours for the Bank image) with the boundary checks responsible for most of it (boundary checks cut down the run time by more than a factor of two.) The same trend can be observed in all the images and the results indicate that using fixnum arithmetic caused the maximum difference while out-of-line boundary checks were the next most effective optimization strategy. Variable bindings also provided substantial savings while flattening and/or hardly made much of a difference.

In order to determine further the savings obtained due to optimization, penalties and run times at the end of 0, 9, 19 and 31 generations are given in Table 1 for both optimized and unoptimized cases for the bank image. The run time per generation increases for later generations since the programs are much more complex than in initial generations. For the same reason, the effect of optimization is more profound during these later generations. For example at generation 0, the runtime for the unoptimized version was about 8 times the runtime for the optimized version, while at the end of 31 generations the runtime for the unoptimized case was more than 17 times of the runtime of the optimized case, indicating great savings during later generations.

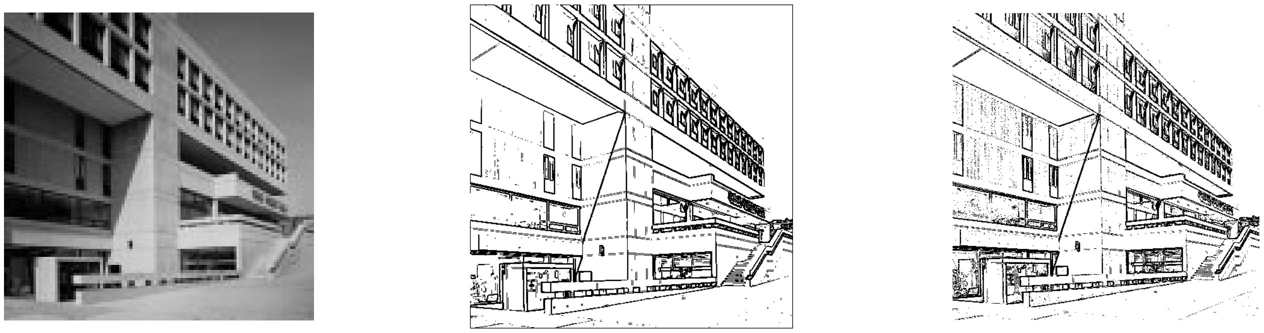
Figure 2: 1) A bank, 2) the edges, 3) the edges determined by the program generated based on the bank.

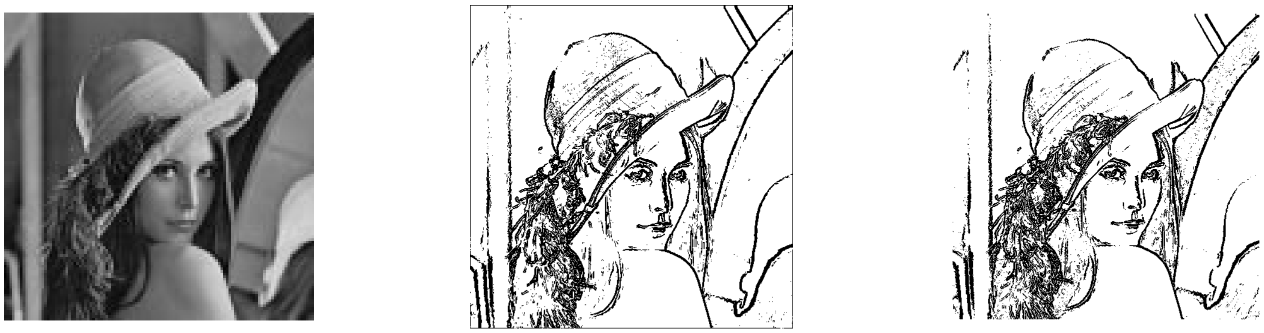
Figure 3: 1) Lenna, 2) the edges, 3) the edges determined by the program generated based on Lenna.

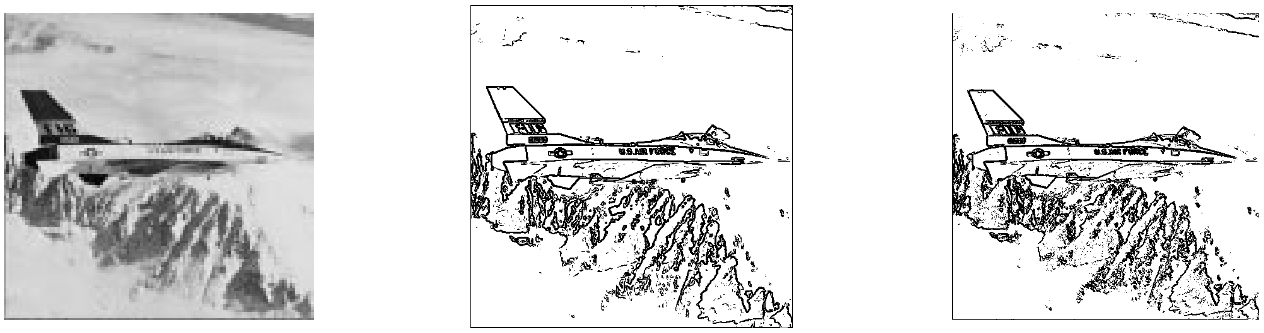
Figure 4: 1) An F16, 2) the edges, 3) the edges determined by the program generated based on the F16.

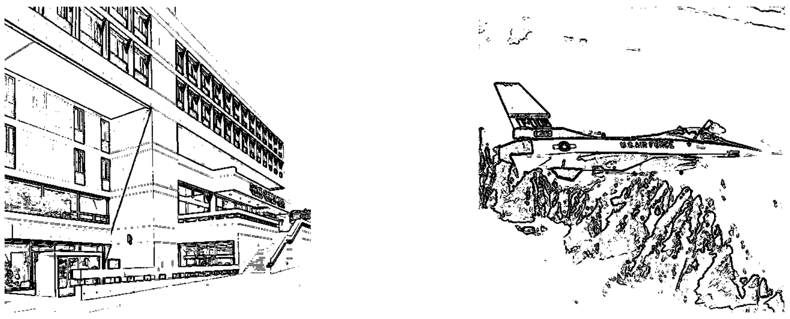
Figure 5: 1) The bank and F16 edges determined by the program generated based on Lenna.

Image	Number of Generations	Final Penalty	Optimizations				
			None	1	1–2	1–3	1–4
Bank	31	1.8×10^8	350	150	136	19.8	19.8
Lenna	52	1.9×10^8	583	250	228	36.0	35.2
F16	20	1.0×10^8	230	99	86	10.7	10.6

Bank Image, Generation	Penalty	Optimizations	
		None	1–4
0	6.8×10^8	3.9	0.5
9	4.5×10^8	40.8	4.4
19	2.3×10^8	139.0	10.3
31	1.8×10^8	350.0	19.8

Table 1: Total runtime in hours; optimizations: (1) boundary checks, (2) variable bindings, (3) fixnum, (4) flatten and/or.

References

[1] Alfred V. Aho, Ravi I. Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, second edition, 1986.

[2] Luca Cardelli. Type systems. In *CRC Handbook of Computer Science and Engineering*, chapter 103. CRC Press, 1997.

[3] A. Chambole, R. A. DeVore, N.-Y Lee, and B. J. Lucier. Nonlinear wavelet image processing: Variational problems, compression, and noise removal through wavelet shrinkage. *IEEE Trans. Image Processing*, 7:319–335, 1998.

[4] William Clinger and Jonanthan Rees (editors). Revised^4 report on the algortihmic language Scheme. November 1991.

[5] Jason M. Daida, Tommaso F. Bersano-Begey, Steven J. Ross, and John F. Vesecky. Computer-assisted design of image classification algorithms: Dynamic and static fitness evaluations in a scaffolded genetic programming environment. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 279–284, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[6] Jason M. Daida, Jonathan D. Hommes, Tommaso F. Bersano-Begey, Steven J. Ross, and John F. Vesecky. Algorithm discovery using the genetic programming paradigm: Extracting low-contrast curvilinear features from SAR images of arctic ice. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 21, pages 417–442. MIT Press, Cambridge, MA, USA, 1996.

[7] I. Daubechies. Ten lectures on wavelets. In *CBMS-NSF Regional Conference Series in Applied Mathematics*, volume 91. SIAM, 1992.

[8] R. A. DeVore, B. J. Lucier, and Z. Yang. Feature extraction in digital mammography. In A. Aldroubi and M. Unser, editors, *Wavelets in Medicine and Biology*, pages 145–161. CRC Press, Boca Raton, 1996.

[9] Marc Feeley. *Gambit-C, version 2.6*. University of Montreal, Canada, 2.6 edition, June 1997.

[10] R. C. Gonzalez and R. E. Woods. *Digital Image Processing*. Addison-Wesley, 1992.

[11] Christopher Harris and Bernard Buxton. Evolving edge detectors. Research Note RN/96/3, UCL, Gower Street, London, WC1E 6BT, UK, January 1996.

[12] T. Haynes, R. Wainwright, S. Sen, and D. Schoenfeld. Strongly typed genetic programming in evolving cooperation strategies. In *Proc. Sixth Int. Conference on Genetic Algorithms*, 1995.

[13] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, 1993.

[14] John R. Koza. *Genetic Programming : On the Programming of Computers by Natural Selection*. MIT Press, Cambridge Massachusetts, 1992.

[15] David J. Montana. Strongly typed genetic programming. BBN Technical Report #7866, Bolt Beranek and Newman, Inc., 10 Moulton Street, Cambridge, MA 02138, USA, March 1994.

[16] Thang Nguyen and Thomas Huang. Evolvable 3D modeling for model-based object recognition systems. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 22, pages 459–475. MIT Press, 1994.

[17] Riccardo Poli. Genetic programming for image analysis. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Proc. Genetic Programming 1996*, pages 363–368. MIT Press, 1996.

[18] Christian Queinnec. *Meroon V3, A Small, Efficient and Enhanced Object System.* Ecole Polytechnique and INRIA-Rocquencourt, 91128 Palaiseau Cedex, France, 1993.

[19] L. Tabar and P. B. Dean. *Teaching Atlas of Mammography*. Thieme, Inc., 1985.

[20] Walter Alden Tackett. Genetic programming for feature discovery and image discrimination. In Stephanie Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 303–309, University of Illinois at Urbana-Champaign, 17-21 July 1993. Morgan Kaufmann.