Report on the Language $\mathbb{A}$

Nick Foti, Eric Kee, Ashwin Ramaswamy

May 27, 2008

Abstract

In this work we describe the language $\mathbb{A}$, which is designed to provide native use of one-dimensional arrays and also image operations.

1 Problem Statement

The X programming language currently only supports scalar data types and no data structures. For our compiler project, we chose to extend the xcom language to support 1-D arrays with semantics similar to vectors in Matlab. Our arrays are bounds-checked and extendable, shallow copied during assignment, and are pre-allocated for programmer convenience. We also provide an additional *image* type, for which we have implemented common image processing operations.

2 Review of Literature

The existing X language allows only scalar variables, which restricts the capabilities of the language. In our work we extend the X compiler to support a new language, $\mathbb{A}$, which allows the programmer to operate on array data structures of each original X data type and on images.

3 Requirements

In this section we list the requirements that we have chosen for the A language.

1. All elements of an array must be of the same type and all scalar types in X must be supported.

2. Array elements are accessed via bracket notation:

   ```
   x[4] := 1;
   ```

   where indexing starts at 0 like in C. The above line assigns the 5^{th} element of the array referenced by `x` to 4.

3. Arrays can be initialized with array constructors:

```
x := [0, 1, 2, 3];
```

4. Arrays elements can be assigned to scalar variables:

```
y := x[2];
```

5. Arrays can be indexed by any form of expression that evaluates to an integer within brackets:

```
z[1] := x[y*2];
x[z[1]] := x[4];
```

6. Multidimensional arrays are not supported. In other words, array indices must be scalar (and integer). The following lines are invalid:

```
x := [[1, 2], [3, 4]];
y[2, 4] := 2;
y[2.5] : = 7.8;
```

7. Arrays are automatically extended when indexed out of bounds on the left side of an assignment and a runtime exception is thrown when indexed out of bounds on the right.

8. Arrays of the same size support element-wise operations with syntax identical to scalar operations.

   - Integer arrays support the following operations: $+$ $-$ $*$ $/$ and $//$
   - Real arrays support the following operations: $+$ $-$ $*$ and $/$
   - Boolean arrays support the following operations: & and |

   Integer and real arrays support element-wise addition, subtraction, multiplication, and division. Integer arrays also support element-wise modulus. perations can be performed with syntax identical to scalar operations.

9. Images can be loaded using the `imgload` keyword. The user will be prompted prior to runtime to enter the desired image file name. For example:

```
x := imgload;
```

10. Images support pixel-wise addition ($+$), pixel-wise subtraction ($-$), unary negation ($-$), and convolution with an array ($*$).

4 Results

4.1 Design

In this section we describe how we designed our compiler to fulfill the requirements discussed in Section 3. To satisfy type consistency within arrays and allow arrays of all types (requirement 1), we create three new data types: an integer array type, real array type and a boolean array type.

We designed a covering grammar to satisfy our requirements on allowable syntax (see Requirements 2 through 5) and exclude syntax for multi-dimensional and nested arrays (Requirement 6). Our covering grammar accepts multi-dimensional and nested arrays; we catch these semantic errors in the parser and throw an error.

To satisfy the requirement that arrays are automatically extended when indexed out of bounds (Requirement 7), we designed a header data structure to store both the size of the array and a reference to the array data. We use the local variable in the frame to store a reference to the header. Figure 1 illustrates this scheme. The size field of the header allows for quick boundary checking to determine whether to extend the array or to throw a runtime error if applicable.

To satisfy the requirement that operations on array types should be performed using syntax identical to scalar operations (Requirement 8), we chose to handle the case of array assignment by performing a shallow copy. When array y is assigned to array x, the header and the data of x are destroyed, and x is made to reference the header of y. Figure 2 illustrates the procedure. Deep copy is performed by the programmer using a `do-od` loop to copy the array values. Additionally, we define arithmetic and comparison operations to be performed element-wise.

To satisfy the requirement that arrays can be initialized with array constructors (Requirement 3), we choose to allocate new memory to hold the elements of the array constructor. The variable on the left-hand side of the initialization is then referenced to the newly allocated space.

To satisfy the requirement that the programmer can load images (Requirement 9), we redesigned the grammar to include a new rule for `imgload`, added the image type to the symbol table. We chose to load the image data and create the image header in the frame before runtime. This decision simplifies our design because we would have to support strings in $\mathbb{A}$ to load images during runtime.

To satisfy the requirement that images can be manipulated with the operators defined in Requirements 9 and 10, we designed our system to perform shallow-copy on image assignment, similar to arrays. The addition, subtraction, and negation operations required no special design decisions; however, to perform convolution, we assumed that the convolution kernel is a square matrix represented as a 1D array. For example: `x:=y*[1, 1, 1, 1, 1, 1, 1, 1, 1]` convolves `y` with a 3×3 box filter and assigns the result to `x`. If the number of array elements is not a square number, we assume that the user has erroneously entered too many filter elements and we compute the `floor` of the square root

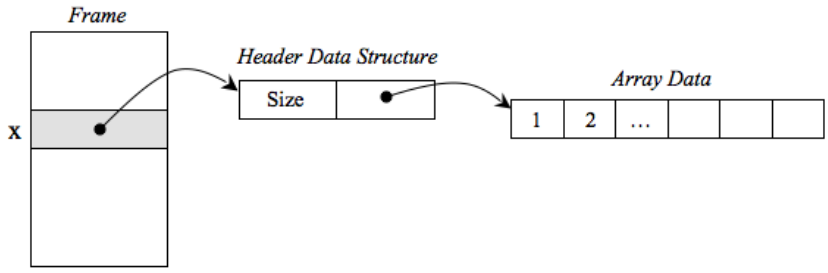

Figure 1: Array data structure design. The variable in the frame holds a reference to a header data structure that contains the size of the array and a reference to the data.

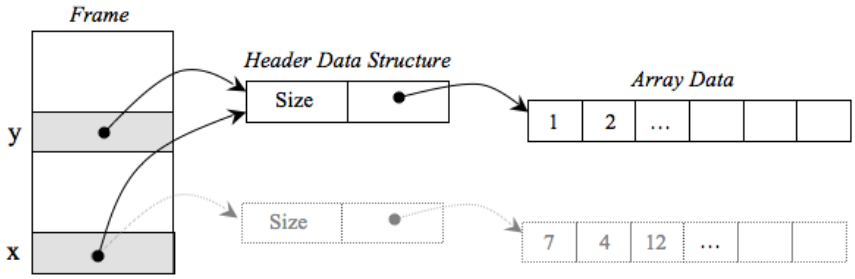

Figure 2: Arrays are shallow copied when performing array assignment. The header structure for x is destroyed and x is assigned to reference the same header structure as array y.

of the number of elements to determine the size of the intended filter.

4.2 Implementation

In this section we describe how we implemented each of the design decisions discussed in Section 4.1. To implement type consistency within arrays (Requirement 1) we modified the code that generates the symbol table to recognize array types and enforce this constraint. The examples below illustrate the type consistency cases for which we modified the symbol table generator:

1. If `x[5]:=2.1;` where `x` is an int array
2. If `x := [1, 2.3, 2];`
3. If `x[3] := 1;` where `x` is an int array. In this case the A language must understand that elements within an int array are of type int.

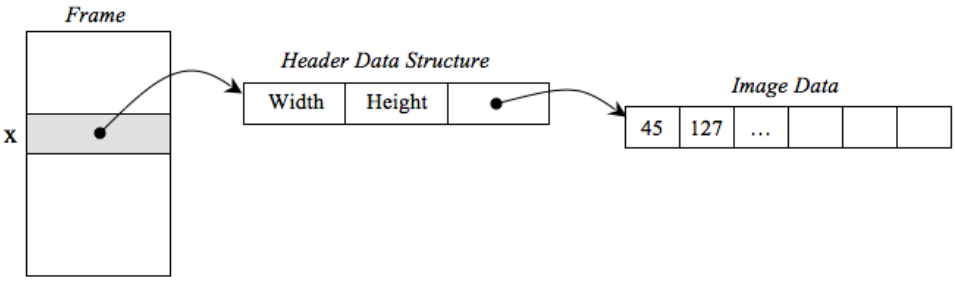

Figure 3: Image data structure design. The variable in the frame holds a reference to a header data structure that contains the width and the height of the image and a reference to the data.

Syntax Implementation. To implement the syntax of the 𝔸 language, we create the covering grammar that is included in Appendix A. Because this grammar allows multi-dimensional and nested arrays, we implement special syntax checking in the parser to generate errors in these cases. We modified the functions for the *var* and $factor$ rules to catch these syntax errors. Our syntax checking code generates an error if more than one opening bracket is encountered before seeing a closing bracket.

To implement the image syntax, we updated the symbol table generator to associate a new type with the terminal *imgload*. We also created a new usage specifier to allow images to be both inputs and outputs. To illustrate the need for a new usage specifier, consider: `y:=imgload;` Because `y` only appears on the left of an assignment, X considers `y` to be an output variable. In 𝔸, we want `y` to be both an input and an output variable and the new usage specifier makes this possible. We modified the symbol table generator rules to assign the new usage specifier and we modified both `xInputs`, and `xOutputs` to load and display the image.

Low-Level Array Implementation. To implement 𝔸 arrays in correspondence with our design decisions we decided to pre-allocate array headers and data as a prolog to the program. The array prolog runs after the original prolog and emits code to allocate a header data structure. The header consists of a 4-byte integer to store the current size of the array as well as a pointer to the data memory. Both fields in the header are initialized to null. See Figure 1.

To implement array constructors, we added a function to emit assembly instructions that move each expression in the array constructor to a temporary variable. We then create an array header and allocate space for the elements in the array constructor. Each temporary variable is then copied into the new array and the new array is assigned to the variable on the left-hand side of the expression.

We also implemented two new functions, `subVarL` and `subVarR` to emit assembly instructions that return the address of a single element within an array on the left and right sides of an assignment, respectively. Function `subVarL` performs bounds checking on the array index by comparing the index to the size field in the array header. If the array index exceeds the size of the array we call `realloc` to expand the data array. We then compute and return the address of the indexed array element. Function `subVarR` works similarly except we generate a runtime error if the index exceeds the size of the array.

We modified the `store` function to handle array types on the left and right of assignments. There are two cases, either a subscripted variable or an array occurs. When a subscripted variable occurs on the left or right side of an assignment, `store` must emit code to indirectly access the memory address referenced by the array operand. When an array occurs code is emitted that first checks if we are assigning the array to itself; if so, we take no action. If an array, y, is being assigned to another array, x, then we must first destroy x's array data structure (data and header) and then point x to y's header. This last case is what we call shallow copying.

We implemented element-wise array operations to extend scalar operation syntax to arrays of the same size. We implemented these operations using C routines in the `getCfun` function provided. The operations that we support are as follows:

1. Integer Arrays: $+ \quad - \quad * \quad / \quad // \quad < \quad <= \quad = \quad \sim= \quad >= \quad >$
2. Real Arrays: $+ \quad - \quad * \quad / \quad < \quad <= \quad >= \quad >$
3. Boolean Arrays: & |

These operations behave as expected with array types and (mathematically valid) composition of them may be used anywhere an `expr` may be used. Currently, $\sim$ is not supported in $\mathbb{A}$. (We felt that implementing another operation had little educational value because the task had become routine and we weren't learning anything new.)

To view the contents of arrays when they are output symbols (only used on the left) in an A program we wrote our own mex functions, rather than using `getCfun`. These functions access the array data structures directly to print the contents of the array. Afterwards, the array data structures are free, so that when our A program terminates we have not created any memory leaks. We implemented one mex function to print integer and boolean arrays and one to print real arrays.

Low-Level Image Implementation. To implement images in $\mathbb{A}$, we implemented *mex* functions that load images into our system from a matlab array and return them to malab after the program has run. Our existing array functions in the emitter were sufficient to implement image operations with minor modification. We also added C routines to `getCfun` that implement image addition, subtraction, negation, and convolution. These additional functions were straightforward because we wrote them in C.

Miscellaneous Modifications. We also implemented a few fixes to bugs that we found in the existing xcom code. For instance the `spillAllRegs` function was only spilling one register because of an erroneous `return` statement. Another example is that the `getReg(exclude)` function was not excluding the requested register. We also needed to construct a workaround for the fact that the `fstAp` instruction did not work. Our workaround involved moving the desired address into the ESI register and using the `fstMp` instruction to offset from that address where we would use the offset 0.

4.3 Validation

In this section we discuss how we validated that our implementation is functional and fulfilled the stated requirements. We also report the results of our testing. We performed some regression testing to ensure that existing functionality of X was not broken. Our validation consisted of numerous cases testing both correct and incorrect programs. See Appendices B and C for a full listing of our test cases. We found that all correct programs executed as expected, and all incorrect programs produced the expected error. Test cases that involved images were visually evaluated for correct functionality: addition of two images should produce a transparency effect, negation produces an obvious intensity inversion, and convolution results will produce edge-detected images when the correct convolution kernel is used.

5 Conclusions and Future Work

We have successfully integrated both one-dimensional arrays and an image type into X. Our new language, $\mathbb{A}$, performs bounds checking, automatic array expansion, and element-wise array operations that have identical syntax to the original X types. $\mathbb{A}$ also provides support for array constructors, which can be used in the same way as array variables. We do not support `i2f`, `f2i`, and $\sim$ because we decided that implementing these operators had little educational value after we had implemented so many others.

We have successfully added an image type $\mathbb{A}$ that allows the programmer to perform image manipulations using common arithmetic operators rather than complex loops. The results of image computations are displayed to the user as outputs after the program completes. Future work on $\mathbb{A}$ should resolve a bug with array constructors (if multiple array constructors are combined on the right-hand side of an assignment, the array constructors are not freed). The image type can also be extended to handle more image operations like image-image convolution, and scalar-image multiplication.

APPENDIX

A Grammar

```
program
  stmts eof
stmts
  stmt
  stmts ; stmt
stmt

  selection
  iteration
  assignment
selection
  if alts fi
iteration
  do alts od
alts
  alt
  alts :: alt
alt
  guard ? stmts
guard
  expr
assignment
  vars := exprs
  vars := subprogram := exprs
       := subprogram := exprs
  vars := subprogram :=
       := subprogram :=
vars
  var
  vars , var
var
  id
  var [ expr ]
exprs
  expr
  exprs , expr
subprogram
  id
expr
  disjunction
disjunction
```

```
  conjunction
  disjunction | conjunction
conjunction
  negation
  conjunction & negation
negation
  relation
  ~ relation
relation
  sum
  sum <  sum
  sum <= sum
  sum =  sum
  sum ~= sum
  sum >= sum
  sum >  sum
sum
  term
  - term
  sum + term
  sum - term
term
  factor
  term * factor
  term / factor
  term // factor
factor
  imgload
  true
  false
  integer
  real
  var
  [ exprs ]
  ( expr )
  b2i factor
  i2r factor
  r2i factor
  rand
```

B Test Cases of Correct Programs

Below are test cases of correct programs using arrays. All test cases resulted in correct output. Note that column numbers reported for errors below may be incorrect because the programs were entered in a single line of text. Here we

have formatted the test programs for clarity.

```
x[5]:=4;
y:=x[5];
x[y]:=2;
z:=x[y];
z:=x[x[5]];

y:=x[9]; ‘arrays have an initial size of 10

i:=0;
do i<5 ? a[i]:=i*2;
          b[i]:=a[i]+3;
          i:=i+1;
od;
ta:=a;
tb:=b;
aPb:=a+b;
aMb:=a-b;
aTb:=a*b;
aDb:=a/b;
aDDb:=a//b;
aLb:=a<b;
aLEb:=a<=b;
aEb:= a=b;
aNEb:=a~=b;
aGb:=a>b;
aGEb:=a>=b;

i:=0;
j:=1.6;
do i < 5 ? af[i]:=j;
             bf[i]:=af[i]+3.14159;
             i:=i+1;
             j:=j+2.34;
od;
taf:=af;
tbf:=bf;
aPbf:=af+bf;
aMbf:=af-bf;
aTbf:=af*bf;
aDbf:=af/bf;
aLbf:=a<b;
aLEbf:=a<=b;
aEbf:= a=b;
aNEbf:=a~=b;
```

```
aGbf:=a>b;
aGEbf:=a>=b;

i:=0;
do i < 5 ?
    if i//2=0 ?
        ab[i]:=true; bb[i]:=false;
    :: i//2=1 ?
        ab[i]:=false; bb[i]:=true;
    fi;
    i:=i+1;
od;
bb[4]:=true;
aAb:=ab&bb;
aOb:=ab|bb;


‘Image Test Cases
‘This loads two images, smooths the first with a 7x7 box filter,
‘applies a 5x5 laplacian edge detector to the second image, and then adds
‘the two images together to get a result
x:=imgload;
y:=imgload;
f:=[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
    1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
    1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
    1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
    1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
    1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
    1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0];

g:=[-1.0, -1.0, -1.0, -1.0, -1.0,
    -1.0, -1.0, -1.0, -1.0, -1.0,
    -1.0, -1.0, 24.0, -1.0, -1.0,
    -1.0, -1.0, -1.0, -1.0, -1.0,
    -1.0, -1.0, -1.0, -1.0, -1.0];
z:=x*f;
p:=y*g;
q:=p+z;’

‘ TEST #1
‘This loads two images, smooths the first with a 10x10 box filter,
‘applies a 5x5 laplacian edge detector to the second image, and then adds
‘the two images together to get a result
x:=imgload;
y:=imgload;
```

```
f:=[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
    1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
    1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
    1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
    1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
    1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
    1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
    1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
    1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
    1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0];

g:=[-1.0, -1.0, -1.0, -1.0, -1.0,
    -1.0, -1.0, -1.0, -1.0, -1.0,
    -1.0, -1.0, 24.0, -1.0, -1.0,
    -1.0, -1.0, -1.0, -1.0, -1.0,
    -1.0, -1.0, -1.0, -1.0, -1.0];

 z:=x*f;
 p:=y*g;
 q:=p+z;'

'This loads two images, smooths the first with a 5x5 box filter,
'applies a 5x5 laplacian edge detector to the second image and displays the
'intermediate results
x:=imgload;
y:=imgload;
f:=[1.0, 1.0, 1.0, 1.0, 1.0,
    1.0, 1.0, 1.0, 1.0, 1.0,
    1.0, 1.0, 1.0, 1.0, 1.0,
    1.0, 1.0, 1.0, 1.0, 1.0,
    1.0, 1.0, 1.0, 1.0, 1.0];

g:=[-1.0, -1.0, -1.0, -1.0, -1.0,
    -1.0, -1.0, -1.0, -1.0, -1.0,
    -1.0, -1.0, 24.0, -1.0, -1.0,
    -1.0, -1.0, -1.0, -1.0, -1.0,
    -1.0, -1.0, -1.0, -1.0, -1.0];

z:=x*f;
p:=y*g;'

'This loads one image and smooths it with a very large gaussian.
x:=imgload;
```

```
f:=[   1.0,   12.0,   55.0,   90.0,   55.0,   12.0,    1.0,
      12.0,  148.0,  665.0, 1097.0,  665.0,  148.0,   12.0,
      55.0,  665.0, 2981.0, 4915.0, 2981.0,  665.0,   55.0,
      90.0, 1097.0, 4915.0, 8103.0, 4915.0, 1097.0,   90.0,
      55.0,  665.0, 2981.0, 4915.0, 2981.0,  665.0,   55.0,
      12.0,  148.0,  665.0, 1097.0,  665.0,  148.0,   12.0,
       1.0,   12.0,   55.0,   90.0,   55.0,   12.0,    1.0];

z:=x*f;'

‘ Load an image and assign another variable to it
x:=imgload;
y:=x

‘ Load two images and add them together, creates compositing effect
x:=imgload;
y:=imgload; z:=x+y;

‘ Load two images and subtract them
x:=imgload;
y:=imgload;
z:=x-y;

‘ Load an image and negate it, reverses colors of pixels
x:=imgload;
y:= -x

‘Test #2
bush:=imgload;
hat:=imgload;
matte:=imgload;
tmp1:=bush- (-matte);
tmp2:=hat - matte;
res:=tmp1+tmp2;
```

C Test Cases of Incorrect Programs

Below are test cases of incorrect programs that use arrays. In each case, the program reports the appropriate error.

```
x[1][2]:=2;
```

```
‘ ??? Multi-dimensional arrays not supported, got '[', at line 1 col 5

y[[10]]:=30;
‘ ??? Nested arrays not supported, got '[', at line 1 col 3

z[3]:=[1,2,3,4];
??? Symbols: Nested arrays not supported at line 1, col 1

x[1,3]:=2;
‘ ??? expected ] on right, got ',', at line 1 col 4

‘ This fragment fails because we should not use a real as an array index
x[10]:=1.0;
y[x[0]]:=3.0;
‘ Symbols: expression type mismatch at line 1, col 15

‘ This fails because we should not use a boolean as an array index
x[10]:=true;
y[x[0]]:=3;
‘ Symbols: expression type mismatch at line 1, col 16

x:=[1,2,3,4]+3;
‘ ??? Symbols: integer type not allowed by context at line 1, col 14

x[5]:=4.3;
y[5]:=x[9];
z:=x//y
‘ ??? Symbols: real type not allowed by context at line 1, col 7

x[5]:=10;
y[5]:=10.1;
z:=x[5]+y[5];
‘ ??? Symbols: type mismatch at line 1, col 26

‘ We do get a runtime error for the following code, but graceful termination
‘ of the program is not yet implemented.
x[5]:=6; y:=x[13];
‘ ??? Attempted to access st(38); index out of bounds because numel(st)=16.


‘ NEGATIVE TEST CASES
‘ Check for imgload as a reserved word
imgload:=3;
??? expected eof, got 'imgload', at line 1 col 1

‘ No image-image convolution
```

```
x:=imgload;
y:=imgload;
z:=x*y;
??? Symbols: real array necessary at line 1, col 28

‘ No integer array in convolution
x:=imgload;
y:=imgload;
z:=x*[1,2,3];
??? Symbols: integer type not allowed by context at line 1, col 35

‘ Try spurious integers in image addition
x:=imgload;
y:=x+3
??? Symbols: integer type not allowed by context at line 1, col 18

‘ No array of images
x[1]:=imgload;
??? Symbols: imgload returns type image at line 1, col 7

x:=imgload;
y[1]:=x
??? Symbols: imgload returns type image at line 1, col 4

x:=imgload;
y:=[x]
??? Symbols: imgload returns type image at line 1, col 4

‘ convolution has to be img*real, not real*img
x:=imgload;
z:=[1.0,2.0,3.0]*x;
??? Symbols: real type not allowed by context at line 1, col 25
```

D Image Results

Below are image test results.

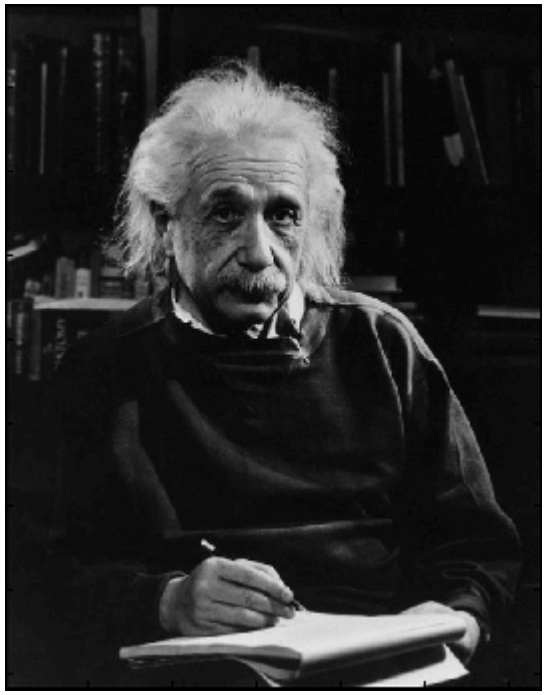

Figure 4: This image is the original image used in Test #1.

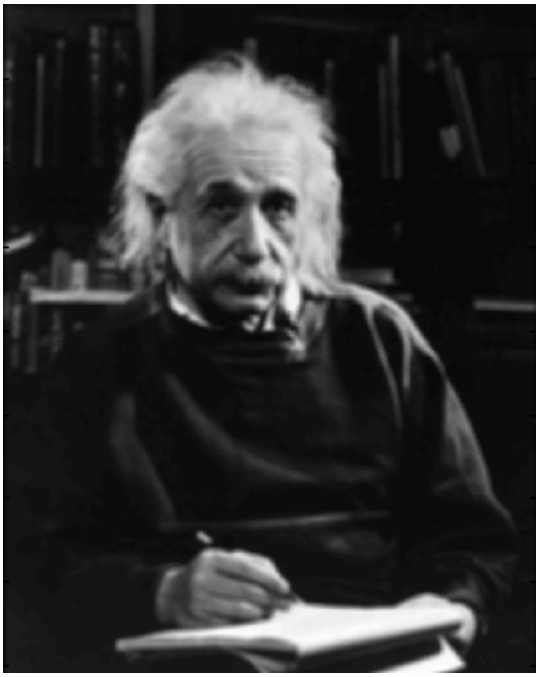

Figure 5: This image is the result of convolving a 10x10 box filter with the image in Figure 4.

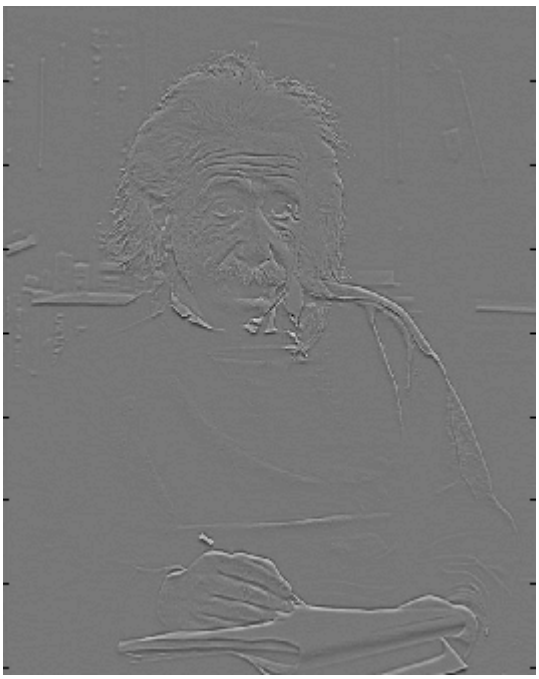

Figure 6: This image is the result of convolving a 5x5 Laplacian edge-detection filter with the original image in Figure 4.

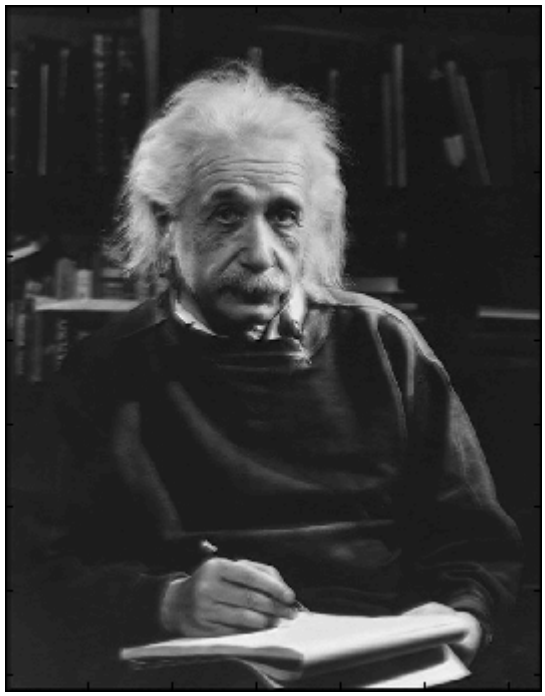

Figure 7: This image is the result of adding the image in Figure 5 to the image in Figure 6. Notice that edge details are preserved while smooth regions have been blurred.

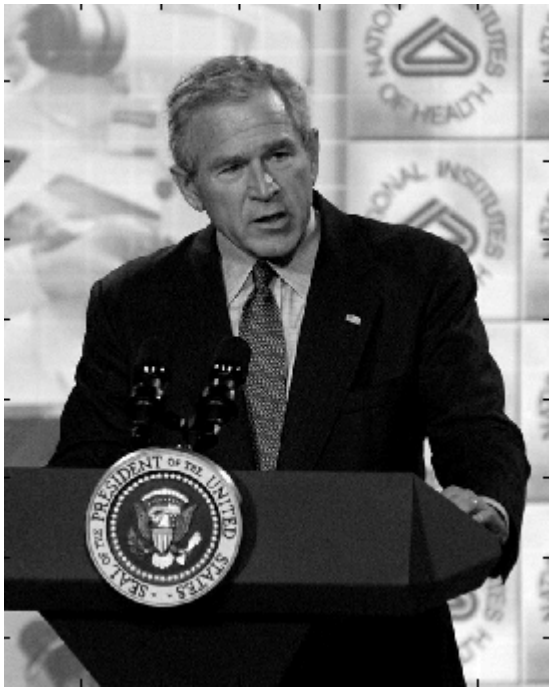

Figure 8: This image was the original image used in Test #2.

Figure 9: This image was the second original image used in Test #2.

Figure 10: This image is a matte image that was used to combine the images in Figure 8 with Figure 9

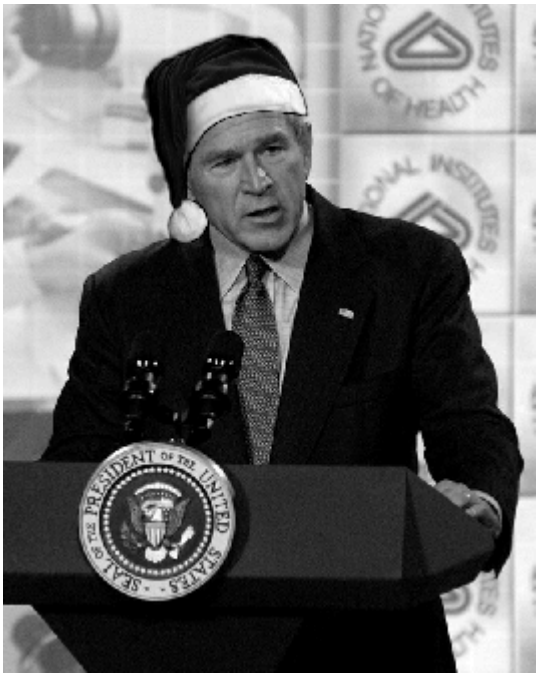

Figure 11: This image is the result of adding the image in Figure 8 to the image in Figure 9 after apply the matte image in Figure 10 to both. Notice that the Santa hat now appears on top of the man's head.