Programming Language Support for Digitized Images or, The Monsters in the Closet

Daniel E. Stevenson*

Department of Computer Science
University of Iowa
Iowa City, IA 52242, USA

Margaret M. Fleck†

Department of Computer Science
University of Iowa
Iowa City, IA 52242, USA

Abstract

Computer vision (image understanding) algorithms are difficult to write, debug, maintain, and share. This complicates collaboration, teaching, and replication of research results. This paper shows how user-level code can be simplified by providing better programming language constructs, particularly a new abstract data type called a "sheet." These primitives have been implemented as an extension to Scheme.

Implementation of sheet operations is made challenging by the fact that images are extremely large, e.g. sometimes over 5 megabytes each. Therefore, operations that loop through images must be compiled from (a specialized subset of) Scheme into C. This paper discusses how the need for extreme efficiency affects the design of the user-level language, the run-time support, and the compiler.

1 Introduction

Within the past few years, computer imaging equipment has become dramatically cheaper and more reliable. Six years ago, a color camera and framegrabber cost $15,000 and was too heavy to lift. It can now be replaced by a $400 hand-held digital camera. As a result, cameras are becoming widely available. Both programmers and researchers are starting to incorporate images into computer science applications remote from the traditional home of images, computer vision (image understanding). The rapid spread of images is particularly obvious on the World-Wide Web.

Although many users require only image processing and graphics packages (e.g. xv, Photoshop), an increasing range of applications require basic image understanding. For example, researchers in the sciences and medicine use images to measure physical properties (e.g. blood vessel width) and screen for interesting events (e.g. unusual cell shapes). Companies, governments, non-profit organizations (e.g. museums), and private citizens are converting photo collections to digitized form. They require effective ways to locate images with specific content in large databases.

Computer vision lies on the border between computer science and electrical engineering. Traditionally, it has been isolated from the rest of computer science. In particular, it has seen little benefit from recent advances in the design and implementation of programming languages. Computer vision algorithms are still written primarily in C, occasionally now in C++.

As a result, computer vision code tends to be complex and repetitive. It is difficult to write, debug, maintain, and share. Inability to replicate results reported in research papers, which is the norm rather than the exception, is a major barrier to progress in this subfield. Collaboration with researchers in other areas of computer science is almost impossible.

More significantly, many computer vision algorithms are not much easier to program in high-level languages (in practice, always Common Lisp). Off-the-shelf compilers and interpreters do not provide the required efficiency. And the high-level language code tends to resemble a word-for-word translation of the C code. Existing abstract data types and con-

*Now at the University of Wisconsin–Eau Claire, Eau Claire, WI 54702

†Now at Harvey Mudd College, 301 E. Twelfth St., Claremont, CA 91711

trol constructs do not match the repetitive structure in this code and, therefore, cannot be used to simplify it.

This paper will present new abstract data types and associated operations. These primitives, implemented as an extension to Scheme [3], encapsulate much of the repetitive work common in computer vision code and can be compiled into efficient C code. We will discuss key issues involved in implementing the required compiler and run-time support, complementing previous work on compiling Scheme into C. And we will suggest how some of the features required in computer vision might find wider application in programming language design.

2 Images are huge!

The most distinctive feature of digitized images is their size. Digital cameras sold for the home PC market deliver 24-bit color images at sizes ranging from 320 by 240 up to 1600 by 1200. When the data is stored in packed binary arrays, this translates into between 0.23 and 5.76 **megabytes** per image. Images obtained from scanners or certain specialized cameras are even larger, as are video image sequences and 3D images from medical scanners (e.g. MRI). Although they can be compressed when stored in files, images must remain uncompressed during analysis.

Said another way, images are about 4-6 orders of magnitude larger than most objects traditionally found in a high-level language, and a single image may be comparable in size to the entire heap of a traditional Scheme session. Moreover, a typical vision function manipulates several such images (e.g. two inputs and one output). A typical interactive user doing experiments will use up all available memory in an attempt to manipulate (e.g. compare) as many images as possible simultaneously.

Because images are so large and many applications require fast response (e.g. tracking moving objects), computer vision programmers are obsessed with efficiency. Only the most stripped-down algorithms are fast when iterated over all one million locations in an image. Hand-coding critical functions in assembler has only recently become rare. This is a harsh environment in which to test programming language methods.

To avoid wasting scarce space, image values are typically stored as packed bytes, both in memory and in disk files, even though they are conceptually real numbers. Allocation and deallocation must be under user control, because images do not naturally become garbage (in the sense of becoming inaccessible to the user) quickly enough to prevent memory from filling up. They must be allocated outside the heap in a language such as Scheme, to prevent heap fragmentation and unnecessary copying of image data. And it must be possible for related images (e.g. an image and a subimage) to share storage.

Finally, algorithm designers must be extremely careful about how image data moves within the computer. Scanning through an image in the wrong order, on a machine with insufficient RAM, can cause dramatic swapping. Operations such as image rotation, which must access their input and output in different orders, sometimes require that very large images be divided into subblocks. Even if there is enough RAM, swapping could still occur inside internal memory caches.

When image processing or image-related graphics is done on an external board, transmitting image data between the board and main memory is often a significant fraction of total running time. Bus speed is one of the major obstacles in using PC-based cameras. On fine-grained parallel processors (e.g. the Connection Machine), transmission of the image from the camera to the processors may completely dominate running time.

3 Existing vision packages

A number of packages of standard image processing data structures and operations have been developed to aid research and teaching in computer vision. Some packages, such as the standard utilities xv and pbmplus, support only simple image processing and manipulation. Others, such as the HIPS Image Processing System [12], Khoros [10], LaboImage [6], and Matlab [19], cover a full range of low-level image processing utilities. Finally, some packages offer support for higher-level vision operations: the Image Understanding Environment (IUE) [11], KB-Vision [7], Vista [14], The Iowa Vision System [4], OBVIUS [5], and the Radius Common Development Environment [17]. Many large sites have at least one in-house package. And additional packages are used to support image understanding projects in scientific fields (e.g. astronomy, biology, medicine).

A typical computer vision package contains a library of C or C++ image operations and an interpreted front-end language. The front-end language is used to connect operations together, to communicate with the user, and to implement high-level ("smart") parts of algorithms such as automatic parameter setting and control of multi-stage or search algorithms.

3.1 Front-end language

Existing packages use a variety of front-end languages, whose main common feature is that they are always interpreted. Some use the Unix shell (HIPS) or interpreted C-like languages (Matlab). Others (Khoros, KBVision, IUE) use a graphical front-end language. Finally, one can use a high-level programming language, such as Common Lisp (Iowa Vision System, OBVIUS, Radius), Scheme, or ML. We strongly prefer to use a modern high-level language, because they are more powerful and simplify collaboration between computer vision and artificial intelligence.

The choice of front-end language is largely independent of which operations are included in the library. The Cantata dataflow interface has been used with at least three packages of operations (Khoros, KB-Vision, IUE). At least two recent languages, Tcl/Tk and the Elk [13] implementation of Scheme, were specifically designed to provide front-ends for a wide variety of applications.

3.2 Operations included in library

All reasonably-designed vision packages support a range of basic image manipulation operations such as display, cropping, rotation, altering greyscale or colormaps, and image statistics. Image processing packages (HIPS, Khoros, LaboImage, Matlab) also support linear filtering, some nonlinear filters, standard transforms (particularly the Fourier transform), and often simple edge detectors. However, these packages do not have good support for higher-level operations that manipulate edge-chains, features, and geometrical objects.

Packages designed for computer vision include data structures and pre-written code for high-level vision operations, either in the form of C++ classes and macros (IUE, KBVision, and Vista) or Common Lisp classes and methods (Iowa Vision System, OBVIUS, and Radius). However, the number of high-level operations in each package is quite limited. They typically include only one modern edge finder (typically Canny's), one camera calibration algorithm (invariably Tsai's), and a small selection of vision algorithms (edge segmentation and grouping, texture features, motion analysis, classification).

3.3 How have they fared?

Although these packages are all well-intentioned, and incorporate many interesting design ideas, there is no real prospect that any of them will become a standard tool used by most of the field. Computer vision algorithms are still the subject of active research and there are many recent, rival algorithms. However, each package includes only one or two algorithms for each task, often ones developed over ten years ago. For example, most packages do not include an edge finder more recent than Canny's.

Furthermore, different packages offer qualitatively different features. The best choice depends on how much money your site can spend, how much memory and disk space your machines have, what operating system you run, what applications you study (medical, satellite, etc.), what theoretical approach you favor, what other software (e.g. LISP) you have, and whether you prefer a graphic or a textual front-end.

In any attempt to address these problems, the most comprehensive packages (e.g. HIPS, IUE, KBVision) have grown so large that they are difficult to maintain and document. For example, the IUE contains over 575 classes, has over 800 pages of documentation, and consumes 500M of disk space [1]. A single package satisfying everyone's needs would be impossibly large.

4 A better approach

The difficulties in designing packages stem from the fact that designers are attempting to standardize at an inappropriate level of abstraction. Moreover, standardization at the correct level requires the full power of a compiler. Because there has been little cross-fertilization between computer vision and programming language design, package implementers have used only insufficiently powerful tools: library functions, classes, and macros.

4.1 The right level for standardization

Consider the case of the operators that generate texture features from a gray-scale image. Previous packages have attempted to provide a standard set of texture operators. However, the literature contains a wide range of texture operators, none of them have entirely satisfactory performance, respected researchers cannot agree about which ones perform best, and new operators are constantly being proposed. Since no consensus exists, standardization at this level is premature.

The correct place to standardize is at a level where there is a scientific consensus. This allows the programmer to have as much support as possible while making it easy to add new functionality and experiment with new variations in algorithms. Standardization at too low a level (C arrays) makes the programmer do most of the work by hand, while standardization at too high a level (image data structures, image processing functions) limits users to currently available techniques and discourages them from expanding the frontiers of science.

Therefore, we must provide standardization and support at the level of the basic data structures and control constructs used to write the library functions in computer vision packages. This would make the library functions simpler and easier to understand. Then, each programmer could create a package customized to their needs, by merging and modifying code from standard libraries.

4.2 Many drops of water make a river

Many previous researchers have approached this as a software engineering problem. Since each piece of repetitive work is conceptually simple, it should be easy for mature programmers to do it. So, it should be sufficient to standardize programming practice, so as to make everyone's code compatible. And, therefore, it should be sufficient to define a suitable set of macros, classes, and accessor functions (e.g. functions to extract value from different types of images).

This approach fails due to the sheer volume of pedestrian work required to properly write a low-level computer vision algorithm. Creative scientists, even junior ones and those doing applied industrial work, quickly get bored with repetitive coding. They will not take the time to make code sufficiently general or portable. And it is inappropriate to make them do so: repetitive work should be done by a computer.

4.3 The value of compilation

Our extension to Scheme, called Envision, uses a compiler to transform user-level Scheme code into efficient C code. Considerable research has been done on compilation of high-level languages and the newest Scheme-to-C compilers [9, 15, 16] perform quite well. However, these techniques have never been applied to generating computer vision code, partly due to lack of communication between programming language research and computer vision and partly due to the significant initial investment of time required to write or adapt a compiler.

Compilation offers several advantages in this domain. It allows library operations to be written in the same language used in the package front-end. Type inference can expand a small number of user-level type declarations into type assignments for all variables. We can efficiently implement a new control form (scan) which eliminates much of the work of looping through all locations in a 2D image, without requiring function calls inside these loops.

Finally, our compiler can automatically perform a variety of optimizations (section 9). Many of these optimizations are common in hand-written vision code. However, human programmers do not apply them consistently, they apply them in unsafe ways, they use approximations with poor numerical behavior, and they do not adapt quickly to changes in the hardware, operating system, or C compiler. It is safer and more efficient to centralize such knowledge in the hands of the compiler writer.

4.4 The necessity of compilation

It is tempting to think that the same effect could be achieved without writing a new compiler, by using facilities such as classes, macros, and type definitions (e.g. in ML). Unfortunately, current languages and compilers seem to lack the power required to define and optimize our new data structures and operations.

First, translating our high-level code into a standard language requires rewrite rules which operate at compile-time (so the output code is efficient) but which are type-dependent. At compile-time, Lisp and Scheme support only type-independent rewriting (macros). The type system in ML[20] seems to lack a mechanism for parameterizing a type by one or more numbers. This gives us no clean way to write a rule which manipulates objects of varying geometrical dimension but which requires access to their dimensions.

Second, a central feature of Envision is that missing (unavailable) values are first-class objects. For example, a variable which normally contains a real value may occasionally be assigned a missing value. Handling missing values requires modifications to type inference rules, modifications to standard arithmetic operations, code analysis to determine which expressions can never return a missing values, and restructuring expressions to minimize the number of tests for whether a variable value is missing. Standard compilers do not contain such support.

5 Sheets

The repetitive parts of low-level vision code can be isolated and removed from user-level code using a new data structure called a "sheet." Sheets represent the large areas of packed storage found inside image representations. They provide substantial capabilities beyond those of arrays, but only capabilities on which there is considerable consensus in computer vision. Using sheets, it is easy to construct any of the wide variety of image representations currently in use.

A sheet represents a function between two spaces: a set of locations (the domain) and a set of values (the codomain). Each space is locally Euclidean: every small neighborhood looks like a piece of $\mathbb{R}^n$ or $\mathbb{Z}^n$. The function is represented to finite precision: values are only available at a finitely dense set of locations and are only known with limited precision. Each sheet contains homogeneous data: all values have the same type and precision. This allows packed storage and optimization of compiled code.

Sheets provide a layer of abstraction which insulates the programmer from the details of how image data is stored in arrays. The sheet appears to contain data of the type described in mathematical specifications of the algorithm. For example, a log-polar stereo map might appear to be tubular, to contain signed floating point values given to the nearest 20^{th} of a unit, and to have no values for certain locations (e.g. where a surface was occluded in one image). The user need not know the details of how this is implemented using a standard array of unsigned 16-bit integers.

5.1 Features provided by sheets

Specifically, the sheets provide support for multi-dimensional values, arbitrary ranges of locations and values, continuity, user-specified precision, circularity, partiality, shared storage, and restrictions.

Multi-dimensional values: The domain of a sheet may be of any dimension. This capability is already provided by arrays. However, in addition, the values stored in a sheet may be points of any dimension. The current implementation supports 1D and 2D domains as well as 1D, 2D, and 3D values. There are several ways to simulate a multi-dimensional codomain using arrays: the programmer need not remember which method is being used.

The use of multi-dimensional values allows the user to represent a wide variety of image data structure with sheets. For example, a motion vector field can be represented using a 2D sheet with 2D codomain. The outline of a 2D image region can be represented using a list which contains, among other things, a continuous 1D sheet with 2D values (the x and y coordinates of the curve).

Range: The locations in a sheet may be any rectangular subsection of 1D or 2D space. For many applications, the origin of the coordinate system should be placed at the projection center of the image. By contrast, arrays force the origin to lie in one corner of the image, requiring geometrical algorithms to constantly add and subtract offsets.

Similarly, the user can freely specify what range of values will be stored in the sheet. The user is not confined to the limited selection of ranges provided by arrays (e.g. unsigned integers, signed longs, floats) nor does the range have to start at zero.

Continuity: The domain and codomain of a sheet may be either continuous (locally like $\mathbb{R}^n$) or discrete (locally like $\mathbb{Z}^n$). Images have a continuous domain and codomain. Edge maps have a discrete domain and codomain. Color maps have a discrete (1D) domain, but a continuous (3D) codomain.

Sheets with discrete domain provide values only at grid locations, whereas sheets with continuous domain provide values at any location within the bounds of the sheet (by interpolation). Sheets with continuous codomain provide values as real numbers, whereas sheets with discrete codomain provide values as integers.

Arrays, by contrast, always have a discrete domain. For the codomain, computer vision programmers are forced to choose between two bad options: discrete integers with an appropriate precision or continuous reals with an inappropriately high precision (thus wasting memory).

Precision: Numbers used in computer vision have a known, finite precision, due to a combination of quantization and contamination with random noise. When a sheet is created, the user specifies the precision of the values to be stored in it. This, together with the user's range specification, automatically determines the number of bytes used to store the value at each pixel. Similarly, the user specifies the spacing between pixel locations.

When using arrays, the spacing between pixel locations is always one unit. For the integer arrays typically used in computer vision, this is also true of the output values. This forces pyramid-based algorithms, for example, to explicitly rescale values.

Circularity: A sheet's domain and/or codomain may be circular. The current implementation supports the following *topological types*: linear (flat) domain and/or codomain, circular domain (e.g. a closed 2D region boundary), tubular 2D domain (e.g. a log-polar image), toroidal 2D domain (both dimensions are circular), and circular codomain (e.g. an image whose values are angles).

The topological type determines what happens if a program attempts to access locations outside the domain range, or store values outside the codomain range. For example, circular codomain values are reduced to the right range using modular arithmetic, whereas linear codomain values are approximated with the closest value in the range. Interpolation of circular values also uses modified algorithms.

Partiality: Values may be unavailable for some locations in a sheet. This may happen in the middle of the sheet (e.g. occluded regions in stereo disparity maps) or adjacent to its edges (e.g. an image which has been rotated or corrected for lens distortion). References to such a location, or to a location outside the sheet's storage range, return a special "missing" value.

In array-based code, missing values can be indicated by storing a special reserved value in the array. Unfortunately, programs don't all use the same reserved value. Different array types (e.g. unsigned short, signed long, float) require different reserved values. Most programs (notably edge finders) do not handle missing values at all.

Shared storage: The packed storage area of a sheet is separated from header information, such as scaling. Two sheets with different headers can share the same packed storage. As a result, rescaling or shifting a sheet does not require extensive memory allocation or copying, just creation of a new modified header.

Restrictions: The header information for each sheet includes a focus area, used to limit display and scanning operations (section 6.4). Thus, a subsection can be extracted from a sheet by combining a new header with the same packed storage.

5.2 How sheets improve programming

Certain packages provide support for some of these features, but this support is partial and erratic. As a result, most programmers build their own versions of features by hand. These implementations are special-purpose and incompatible with one another. They often use substandard methods, such as bilinear interpolation.

Standardized support allows simple and portable user-level code. It ensures that good methods are used for standard operations such as scaling and interpolation, that these operations are fully debugged, and that they are implemented using a standard portable language (e.g. ANSI C). It helps users write algorithms which handle the full range of images.

Figures 1 and 2 show C and Envision code for a typical operation. The Envision code is shorter than the C code, despite its longer (Lisp-style) function names. The C code is restricted to 8-bit unsigned values, whereas the Envision code handles sheets with any range of values. The Envision code uses second-order, rather than bilinear, interpolation and leaves smaller holes around missing values. It requires fewer input arguments. And, it rotates about the image center, a meaningful user-level location, rather than about the upper-left corner of the storage array.

```
void rewindow
 (double startx, double starty, double angle,
  *char array1, *char array2, int xsize1,
  int ysize1, int badval1, int xsize2, int ysize2,
  int badval2, int minval2, int maxval2)
{
  int newx, newy, lowx, highx, lowy, highy, intout;
  double realx, realy, errorx, errory, v1, v2, v3,
      v4, sinangle, cosangle, interpolated_value;
  sinangle = sin(angle); cosangle = cos(angle);
  for (newx = 0; newx < xsize2; newx++) {
   for (newy = 0; newy < ysize2; newy++) {
     realx = startx + newx*cosangle + newy*sinangle;
     lowx = floor(realx); highx = ceil(realx);
     errorx = (highx - realx);
     realy = starty + newy*cosangle - newx*sinangle;
     lowy = floor(realy); highy = ceil(realy);
     errory = (highy - realy);
     if (lowx < 0 || lowy < 0
          || highx >= xsize1 || highy >= ysize1)
         array2[(newx * ysize2) + newy] = badval2;
     else {
        v1 = array1[(lowx * ysize1) + lowy];
        v2 = array1[(lowx * ysize1) + highy];
        v3 = array1[(highx * ysize1) + lowy];
        v4 = array1[(highx * ysize1) + highy];
        if (v1 == badval1 || v2 == badval1
            || v3 == badval1 || v4 == badval1)
           array2[(newx * ysize2) + newy]
              = badval2;
        else {
           interpolated_value =
                errorx * errory * v1 +
                errorx * (1.0 - errory) * v2 +
                (1.0 - errorx) * errory * v3 +
                (1.0 - errorx) * (1.0 - errory) * v4
                + 0.5;
           intout = interpolated_value;
           if (intout < minval2) intout = minval2;
           else if (intout > maxval2)
              intout = maxval2;
           array2[(newx * ysize2) + newy]
              = intout;}}}}}
```

Figure 1: C code to rotate and shift an image array.

```
(bulk-define
    rewindow ; name
   ((manifold 2 1) ; input types
        (manifold 2 1) real real real)
    unspecified ; output types
  (lambda (insheet outsheet xoffset yoffset angle)
     (let ((sinangle (sin angle))
           (cosangle (cos angle)))
       (scan (location outsheet)
          (let*
     ((point (sample->point location))
             (xcoord (point-coordinate point 0))
             (ycoord (point-coordinate point 1)))
           (sample-set! location
             (sheet-ref insheet
                (+ xoffset
                   (* xcoord cosangle)
                   (* ycoord sinangle))
                (- (+ yoffset (* ycoord cosangle))
                   (* xcoord sinangle)))))))))
```

Figure 2: Envision code for the same operation.

6 Other new language features

To make full use of sheets, Envision provides a range of other new language features.

6.1 Points

A new data type, the "point," is introduced to represent locations in the domain of a sheet and values stored in its codomain. Following the standard conventions of pure mathematics, a 1D point is simply a number. Higher dimensional points, such as 2D and 3D points, are implemented as structures. Basic arithmetic operations are extended to work transparently on higher dimensional points.

6.2 Missing values

Missing values (unspecified, bottom, ...) are widely used in programming language design. As far as we know, however, Envision is the first language in which they are first-class objects. That is, they can be assigned to variables, stored in data structures including sheets, and so forth. Basic arithmetic operations have been extended to handle the possibility that some of their inputs may be missing (typically returning a missing value) and to return missing values in other appropriate cases (e.g. division by zero).

We believe that first-class missing values are an extremely useful feature, which could be used elsewhere in high-level languages. For example, they could serve as the values of symbols which have not yet been bound, or as the value of assoc for items not in the list.

Missing values also represent a different philosophy for error handling. Standard languages trigger an error by default, and optionally let the user install error handlers. By default Envision triggers error breaks much more seldom. The user must force additional error breaks by explicitly testing whether some condition is satisfied (e.g. some value is nonmissing). This "mellow" convention is essential for image processing, in which the failed computation is typically only one of a million similar computations, most of which probably succeeded.

6.3 Samples

Any location in a continuous sheet can be accessed, but only certain specific locations, namely those on the storage grid, can be set. Envision includes direct pointers to grid locations, called "samples." Samples allow the programmer to bypass scaling and interpolation of locations in a sheet's domain, necessary for optimizing certain low-level vision algorithms.

To do this safely, programmers have no direct access to the raw array coordinates stored inside each sample. Rather, high-level operations allow them to find the sample nearest a floating-point location, retrieve the samples at the two extreme corners of a sheet's focus area, find the scaled coordinates of a sample, move by a specified displacement on the sample grid, and so forth. Because of the restricted direct access, Envision samples are similar to Java's "safe pointers."

6.4 Scanners

Applying a low-level vision operation to a sheet typically requires enumerating the samples in its focus area. In traditional computer vision programming, the user must extract the array bounds and compose a double loop. In Envision, most enumeration is done with the high-level primitive SCAN. Explicit loops are reserved for algorithms with unusual structure.

Scan has the following syntactic form, in which the test and the scanner are optional:

```
(scan (variable sample-or-sheet
       test scanner)
 expr1 expr2 ....)
```

Scan enumerates the samples in the focus area, binding the variable to each one in turn and evaluating the expressions inside the form. The scan starts at the input sample, or at the first location in the input sheet. When a sample passes the test or the end of the sheet is reached, it halts, returning the current sample and a boolean indicating whether it ran out of samples. The returned sample allows the scan to be restarted from where it left off.

The scanner input determines which samples are enumerated (e.g. every sample? every other sample?) and in which order. Envision includes a selection of standard scanners, including a default one used if the scanner input is omitted. Programmers can easily add new ones.

Scan differs from the standard Scheme map operator in two ways. It enumerates locations (samples), not values. This is essential in image processing, where an output value typically depends not only on the corresponding input value but also on values near it. Second, scan gives the programmer flexible control of the enumeration order. Such control is more important in computer vision than in traditional Scheme applications, because a 2D image can be ordered in more useful ways than a 1D list can.

6.5 Geometrical objects

Graphical display is largely handled by a single primitive DRAW. Input to DRAW includes a geometrical object, a window, a location in the window, and a list of options (e.g. color, filled, size). The type of the geometrical object determines what sort of figure will be drawn.

Geometrical objects include 2D sheets (mapped onto the window), 1D sheets (drawn as curves), points, line segments, polygons, ellipses, and text. The parameters in line segments, polygons, and ellipses may be points or 1D sheets. In the latter case, the object represents a set of objects if the sheets are discrete. If they are continuous, the object represents a swept strip or volume. These geometrical objects can also be used in implementing high-level vision algorithms.

6.6 File storage

Scheme includes only ASCII file I/O, unsuitable for storage of objects as large as sheets. Envision provides a second type of structured file I/O, in which packed sheet data is written in binary and other objects are written in a tagged ASCII representation. Operations are also provided to read bytes and sequences of bytes, for reading other image file formats (e.g. PPM, JPEG) and communicating with devices (e.g. cameras).

6.7 Storage areas, open files, windows

As described above (section 2), sheets cannot be garbage collected automatically. To help the user manage sheet space, Envision maintains a list of *storage groups*, i.e. sets of sheets sharing a common packed storage area. For each storage area, it can provide one representative sheet, which the user can examine, e.g. when deciding whether to deallocate it.

Since we are forced to provide storage-management tools for one badly-behaved type of data, there is no conceptual problem extending such tools to other places where they would be useful. The ease with which users can lose pointers to open files and windows is a long-standing problem in high-level language design. Envision allows the user to list such open connections.

7 Our implementation

We have implemented Envision as an extension to the Scheme48 implementation of Scheme [9]. The overall structure is determined by three main ideas: separation of sheet data from the high-level front-end, variable compilation, and separation of sheet type handling into run-time and compile-time components.

7.1 Two-processor architecture

We have implemented Envision using two processes. The front-end is Scheme, augmented with a variety of functions implementing Envision's user front-end and its compiler. Sheet data, window graphics, and binary file I/O, however, are handled by a separate C program. This "coprocessor" is connected to Scheme via a Unix socket.

Crucially, packed data from sheet storage areas is never passed down the socket. To the user, sheet data appears to reside in Scheme. In fact, the packed storage areas for sheets reside in the coprocessor. Scheme has only the header data for each sheet, plus a unique identifier that allows it to identify the storage area when communicating with the coprocessor. Therefore, the socket connection does not have to be fast. In our experience, the only interfaces fast enough for image transmission use shared memory and these tend to be fragile.

This architecture offers three advantages. First, since our primary field is not programming languages and we wanted to produce a usable prototype quickly, we simplified our work by using an existing Scheme implementation and not modifying its internals. Second, individual users create new coprocessor binaries whenever they link in C code generated for them by the compiler. It is convenient that they need only rebuild the coprocessor binary, because the Scheme binary is 30 times larger and more complicated to rebuild. Finally, we wanted to test how well algorithms could be divided into sheet-processing and high-level parts. A clean algorithmic separation would simplify taking advantage of specialized image processing boards or a second processor, without placing undue strain on the bus or network connecting these to the main processor.

7.2 Variable compilation

A typical computer vision algorithm contains both functions that manipulate the values within sheets and high-level functions that operate on more traditional data structures. Because they manipulate objects of grossly different size, these two types of functions require very different types of compilation. (It would be premature to decide whether this is a binary distinction or two samples from a continuous variation.)

High-level scheme functions are compiled into Scheme48's byte code by Scheme48's compiler. They can use all the features of Scheme and Envision. However, Scheme functions which use sophisticated features cannot be compiled into code efficient enough to scan operations across large sheets, on current hardware. [1]

Functions which scan across sheets must be compiled for maximum efficiency. The types of all variables must be determined at compile time. The user must declare the types of input and output values, because it is frequently impossible to infer whether numbers and sheets are real/continuous or integer/discrete. An error is triggered if the compiler cannot determine the type of any internal variable.

Furthermore, such functions cannot allocate or dellocate non-stack space, nor can they call graphical display functions. Variables are restricted to points, booleans, sheets, and samples. Only points can be

[1] They would run faster if compiled into C, e.g. using Bigloo [15, 16], but not fast enough.

missing. Only basic control constructs are allowed. These restrictions were inspired, in part, by those of Prescheme [9].

Previous optimizing Lisp and Scheme compilers, such as Bigloo or Franz Common Lisp, regard the input code as fixed and take as their goal optimizing it as much as possible. The harsh environment of image processing forces us to take a different attitude for sheet-handling code: the code must run with sufficient speed and therefore the user must be helped to write such code, by a combination of language restrictions and compiler errors. Computer vision researchers find it frustrating to guess what modifications to their code will convince a general-purpose compiler to make it run fast enough.

Envision's compiler produces two types of output for sheet-handling functions. For debugging code on small images, they can be compiled into code that runs on the coprocessor's stack machine (section 8.2). For final use, they are compiled into C code, which can be linked directly into the coprocessor.

7.3 Sheet typing

Type features for sheets (and, by extension, samples) are divided into two groups. The dimensionality of the domain and codomain, and whether each is continuous or discrete, have profound effects on algorithm design. Only the most trivial functions (e.g. copy) are polymorphic across these distinctions. Furthermore, they drastically affect C code generation, e.g. whether C variables are declared as floats or ints, whether 1D or 2D code is substituted when expanding a SCAN form, and whether the inputs to addition are numbers or vectors. Therefore, these type distinctions are resolved at compile-time.

By contrast, the design of most computer vision algorithms does not depend on the other type features: range, precision, circularity, or whether missing values might be present. Previous systems typically forced users to make some of these distinctions at compile-time, resulting in annoyingly non-general code (e.g. edge finders that would run only on signed 8-bit images). The increase in generality obtained by resolving these distinctions at run-time is worth the small penalty in running time.

8 The Envision coprocessor

The coprocessor provides four capabilities: allocation and dellocation of packed storage for sheets, the run-time component of sheet support, a stack machine that can run user-defined code, and a graphics interface with a built-in event loop.

8.1 Implementation of sheets

Sheets are implemented as arrays of bytes. Depending on the range and precision requested by the user, between one and three bytes are allocated per pixel. Missing values are marked by storing a special reserved value into the array. Sheets with 2D domain are implemented using a 1D array of pointers to the first element in each row, which results in faster access than standard address arithmetic. Sheets with 2D or 3D codomain are implemented using two or three arrays.

With each sheet, the coprocessor stores three accessor functions. These accessors are given raw array coordinates: scaling is handled by Scheme and the compiler. Two accessors retrieve and set the value at a particular integer array location. The third accessor, present only for sheets with continous domain, retrieves an interpolated value for a location specified by floating point coordinates. All three accessors handle circularity, missing values, and range restrictions. However, they return raw integer values, to which the compiler must apply the appropriate scaling and offset.

Interpolation is implemented using a nine-point second-order polynomial interpolate. When all 9 values are available, this is similar to the six-point interpolate described in [2]. However, our interpolation algorithm handles any pattern of missing values, handling simple cases quickly and decaying gracefully to a bilinear, linear, or nearest-neighbor interpolate as required. This capability is required for correct, fully general implementation of operations such as image rotation. However, it would be very difficult for most users to implement on their own and no previous vision package provides it.

8.2 Running functions

From the Scheme interpreter, the user can call user-defined functions installed in the coprocessor. These include fully compiled functions linked directly into the coprocessor and also functions compiled, for debugging, into instructions for the coprocessor's stack interpreter. The interpreter includes simple assembler instructions, a function calling mechanism, special handlers for basic sheet operations, and a wide range of mathematical functions directly available in the standard C library (e.g. trignometric functions).

When some inputs to a function are sheets, the sheet header information is passed from Scheme to the coprocessor. Sheet headers are large compared to the other types allowed (216 bytes), and it is very inefficient to copy them around C's stack or the coprocessor's stack. Therefore, sheet headers passed from Scheme are stored in a special array and referred to by number. The contents of this array are static during the function call, because user-defined coprocessor functions cannot create, delete, or modify sheets.

8.3 Graphics support

Finally, the coprocessor manages the user's interaction with the window system. It includes operations for creating and destoying windows, and primitives for displaying graphical objects on them. Events such as window resizing, motion, and exposure are handled automatically. Command-type mouse events (moving and clicking inside windows) will eventually be handled by an X-based user-interface program. The current implementation uses the X window system but the user-level language is generic and should be compatible with many window systems.

9 The compiler

The Envision compiler transforms the user-level code into an intermediate language, with operations and control structure similar to C, but with Lisp-like syntax. This transformation is carried out by Scheme-to-Scheme transformation rules inspired by (but rather different from) those in [8]. From the intermediate language, it is easy to generate both C code and code for the coprocessor's stack machine.

With the exception of certain straightforward basic components, our compiler handles problems largely disjoint from those treated by previous compilers (e.g. [9, 15, 16]). On the one hand, we excluded control constructs which threatened to create difficulties and which did not seem useful in writing sheet-handling functions (which tend to have a restricted structure). On the other hand, the Envision compiler spends considerable effort optimizing the handling of higher-dimensional objects and missing values.

9.1 Type inference

Envision's type inference engine allows types to be parameterized by dimension. Points have a single dimension parameter (1D, 2D, or 3D). Sheets and samples have two dimensions: one for the domain and one for the codomain. This allows type inference rules to enforce appropriate dimension relations among the inputs to a function, and between the inputs and outputs. It also gives compiler functions access to the dimensions as numbers. For example, the function which extracts the kth coordinate of a vector must test whether k is in the correct range.

Type inference handles missing points by assigning them a type with a special (wildcard) value in place of the dimension. These types are fused with normal types in the obvious way, e.g. when a missing and a non-missing value are generated by the two branches of an if statement.

9.2 Scanner substitution

To avoid function calls inside loops at run-time, the enumeration code from the specified scanner is substituted into each scan form at compile time. This substitution process must examine whether the sheet input is a sheet or sample, and whether it is 1D or 2D. For example, the default scanner FORWARD contains two versions of the looping code, one for 1D sheets and one for 2D sheets. While not conceptually complex, type-dependent substitution cannot be accomplished with standard Scheme macros and must therefore be built into the compiler.

9.3 Real numbers

It is incredibly inefficient to rebuild real-number handling, including utilities such as trigonometric functions and random-number generation, on top of integer arithmetic. C provides well-optimized (if not ideally general) real number handling, which we used.

9.4 Missing values

To a first approximation, a missing value might appear anywhere that a point is expected. A naive implementation might incorporate a test for missing values into the implementation of each sensitive operation, e.g. all the numerical functions. However, this results in massively more testing than is performed in hand-written code.

The compiler performs a "purity analysis" to determine which variables and expressions can never return a missing result. A value may never be missing because it came (directly or indirectly) from a constant input. It might be the result of a strict operator [2] applied to never-missing values. Certain operations for retrieving sheet parameters (e.g. the offset) cannot return missing values. These appear in the expansion of extremely common functions, notably those for setting and accessing values in sheets.

Finally, a value can never be missing if it is protected by an explicit test for whether it is missing. Some tests occur in the user-level code. Others are generated automatically when the compiler expands sensitive operations.

The compiler also recognizes connected groups of strict functions and collects all required tests for missing values at the start of the group. Prior to this step, the compiler restructures the code so that expressions do not contain any forms which might generate side-effects, so that a left-to-right order of evaluation is preserved when sub-expressions are extracted.

9.5 Vector operations

Expansion of vector operations ensures that temporary variables are allocated as needed, but not if the input is a variable or a constant. Moreover, if an input is explicitly headed by a 2D or 3D point constructor (or a sequence ending in such a form), the input is decomposed into its constituents at compile-time rather than at run-time. For example, a sequence of 3D vector + and * operations will become three 1D numerical expressions followed by only one point constructor.

This complex analysis of the input, and deconstruction of some inputs, is beyond the power of restricted macro facilities such as Scheme's define-syntax. Moreover, the proper expansion depends on the dimensionalities of the inputs and, thus, it must follow type inference.

Expansion of vector operations must follow generation of tests for missing inputs. The vector expansions destroy groups of strict operations, because a single user-level operation may become a complex form which sets and uses temporary variables. (A particularly bad example is the cross-product operation.) Therefore, even if a general-purpose compiler supported missing values, it would still be difficult to define vector operations using user-level features such as macros or classes.

9.6 Sheet references

Operations which set or access sheet values must also be expanded with some care. To a first approximation, they simply add scaling and offset to the low-level access function. However, like vector operations, they must allocate temporary variables exactly when needed. Furthermore, if the domain and/or codomain is discrete, scaling must be done using exact integer operations which check that the divisions involved have no remainder. Therefore, certain parts of this expansion must be delayed until after type inference has been done.

9.7 Space allocation

Dynamic storage allocation is too slow to be allowed inside sheet-handling functions. All the non-stack storage required must be allocated before the function is called, and passed to it. Therefore, sheet-handling functions cannot use lists to return several objects and so the compiler must support multiple return values.

Points, including numbers and missing points, are represented by C structs. When such a value is returned by a form inside a SCAN loop, it is inefficient to create the struct anew in each iteration of the loop (even on C's stack). Instead, the compiler reserves stack space for all such internal point structs once, at the start of the sheet-handling function.

When, in addition, an internal point value is known never to be missing, the compiler can pre-set the missing? field of the point struct. Thus, inside a scan loop, the only fields of the struct which must be examined or set are the actual numerical values.

[2] I.e. an operator which never returns a missing value when given non-missing inputs.

This optimization eliminates essentially all overhead on 1D arithmetic involving non-missing numbers.

Attempting to avoid overhead on simple arithmetic is common in compilers for high-level languages. However, it is difficult to do in the context of a form which is evaluated only once (or a few times) each time the function is called. Our optimization is easy and often applicable, because it exploits the fact that scan loops contain many strict arithmetic operations and forms inside scan loops are evaluated many times.

9.8 Linear operations

Hand-written C code often takes advantage of the fact that image processing operations often add or subtract values from different locations in the same image. We intend, eventually, to detect forms making multiple references to the same sheet and factor out the scaling and offset applied when extract values from the sheet.

10 Implementation results

We are in the late stages of debugging Envision and writing appropriate example code and new-user documentation. Draft documentation can be found on our web site [18] and we anticipate releasing the code in early fall. It requires only standard Unix components (ANSI C, sockets, and Xlib) and currently runs on three operating systems (Linux, HP-UX, and SGI Irix). Both Scheme48 and Envision are extremely small systems: compressed source for both will fit on two floppy diskettes.

The current system is fast enough for many types of image processing research. Rotation of a 538 by 364 image (by an arbitrary angle) takes 6 seconds on a 120MHz Pentium PC and under 3 seconds on an SGI Indigo 2. Functions run on the coprocessor's stack machine take about 7 times as long: still sufficiently fast for debugging using small examples. We believe that a combination of further optimizations and rapidly increasing processor speed will eventually make the output suitable for a wide variety of image processing applications. The most demanding real-time applications may, however, require a more sophisticated compiler which can take advantage of image processing boards.

11 Conclusions

This paper has shown how techniques of modern programming language design can be successfully applied to an unusual domain: computer vision. We have seen that that image handling requires interesting new programming language constructs, including new data types (sheets, samples), a new mapping-type operator (scan), and first-class support for missing values. We have also seen that, although the huge size of images demands extreme efficiency, this can be achieved by compiling a high-level language using current methods, appropriately adapted to the specific application.

Acknowledgments

This research was supported by NSF grants IRI-9501493, IRI-9420716, and IRI-9209728. We would also like to thank Richard Kelsey, Jonathan Rees, Manuel Serrano, and Val Tannen.

References

[1] Amerinex Applied Imaging (1996) "Image Understanding Environment Program: Overview," http:// www.aai.com/ AAI/ IUE/ IUE.html, 9 September 1996.

[2] Bracewell, Ronald N. (1995) *Two-Dimensional Imaging*, Prentice-Hall, Englewood Cliffs NJ.

[3] Clinger, William, Jonathan Rees, et al. (1991) "Revised^4 report on the algorithmic language scheme," *ACM Lisp Pointers* 4/3, 1–55.

[4] Fleck, Margaret and Daniel Stevenson, "The Iowa Vision System Project," http://www. cs.uiowa.edu/ ~mfleck/ vision-html/ vision-system.html, 13 September 1996.

[5] Heeger, David and Eero Simoncelli, "OBVIUS," http:// white.stanford.edu/ ~heeger/ obvius.html, 1 November 1996.

[6] Jacot-Descombes, Alain, Marianne Rupp, and Thierry Pun, "LaboImage: a portable window-based environment for research in image processing and analysis," *SPIE Proc. Vol 1659: Image Processing and Interchange: Implementation and Systems* (1992), pp. 331–340.

[7] "KBVision," Amerinex Applied Imaging Inc., Amherst, MA, http:// www.aai.com/ AAI/ KBV/ KBV.html, 1 November 1996.

[8] Kelsey, Richard and Paul Hudak (1989) "Realistic Compilation by Program Transformation," *Proc. 16th ACM Symp. on Principles of Programming Languages*, pp. 281–292.

[9] Kelsey, Richard and Jonathan Rees (1995) "A Tractable Scheme Implementation", *Lisp and Symbolic Computation 7(4)*.

[10] "Khoros," Khoral Research Inc., Albuquerque, NM, http:// www.khoros.unm.edu/ khoros/, 1 November 1996.

[11] Kohl, Charles and Joe Mundy (1994) "The Development of the Image Understanding Environment," *CVPR 94*, pp. 443–447.

[12] Landy, Michael S., Yoav Cohen, and George Sperling, "HIPS: A Unix-Based Image Processing System," *CVGIP 25* (1984), pp. 331–347.

[13] Laumann, Oliver and Carsten Bormann (1994) "Elk: the Extension Language Kit," *USENIX Computing Systems* 7/4, pp. 419-449.

[14] Pope, Arthur R. and David G. Lowe, (1994) "Vista: A Software Environment for Computer Vision Research," *CVPR 94*, pp. 768–772.

[15] Serrano, M. and Weis, P. (1995) "Bigloo: a portable and optimizing compiler for strict functional languages," SAS 95, pp. 366–381.

[16] Serrano, M. (1994) *Vers une compilation portable et performante des langages fonctionnels*, Thèse de doctorat d'université, Université Pierre et Marie Curie (Paris VI), Paris, France.

[17] SRI International, "RCDC Home Page," http:// www.ai.sri.com/ perception/ software/ rcde.html, 1 November 1996.

[18] Stevenson, Daniel and Margaret Fleck, "Envision: Scheme with Pictures," http://www. cs.uiowa.edu/ ˜mfleck/ envision-docs/ envision.html, 7 June 1997.

[19] Thompson, Clay M. and Loren Shure (1993-95) "Image Processing Toolbox for use with MATLAB," MathWorks Inc., Natick, MA.

[20] Ullman, Jeffrey D. (1994) Elements of ML Programming, Prentice-Hall, Englewood Cliffs NJ.