Style characterization of machine printed texts
Bagdanov, A.D.

Citation for published version (APA):
Bagdanov, A. D. (2004). Style characterization of machine printed texts

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 6

A functional approach to software design in image processing research environments

"Programming languages are rigorous but incomplete approximations of the language of mathematics."

-Didier Rémy, *Using, Understanding, and Unraveling the OCaml Language*

6.1 Introduction

In this chapter we will depart from the sort of document analysis problems that have concerned us for the first four chapters, and turn to a topic that permeates all of them. This chapter is about marshaling all of the disparate fragments of functionality necessary to perform experimental image processing and computer vision research.

Each previous chapter was concerned, in various degrees, with image processing. In chapter 2 we assumed perfect segmentations of document images were available. Chapters 3 and 4 required morphological and recursive filter operators. In the preceding chapter we relied heavily on Gaussian filtering and neighborhood operations to implement iterative solutions to non-linear diffusion partial differential equations. The tools proposed in this chapter would apply equally well to many other application areas.

The image processing software requirements in research environments are very much a moving target. Specific needs can be ephemeral, existing only for the duration of a single paper, thesis, project, or whim. As such, a single researcher can spend a great deal of energy putting together all of the software components necessary for just a single experiment. In particular, a large amount of time can be spent in translating relevant mathematical abstractions of theory into the software engineering abstractions of practice.

In addition to the changing needs of researchers, another consideration in the design of a flexible software environment is the varying levels of interaction a researcher will require from it when developing ideas. When prototyping, for example, interaction and responsiveness are key, as well as the ability to rapidly express ideas in the form
of running code. When running large scale experiments, however, interaction becomes less important, and efficiency is paramount. Much experimental software is written to satisfy specific and immediate needs, and then it is thrown away. While we could endlessly debate whether or not this is the correct software model for researchers, the fact remains that this is how it is, and it is unlikely to change overnight. We choose, rather, to focus on how to optimize such a model so that it can be more effective.

In the next section we expand on these observations with a discussion of the motivations and specific implementation decisions made. Section 6.5 begins the development with descriptions and examples of the primitive image processing operations supported by our system. Section 6.6 continues with a discussion of how native, efficient, back-end implementations of the the basic functionality can be rolled in, supporting optimized, native-code efficiency for applications requiring it. In section 6.7 we show how these primitive operations provide the building blocks for the construction of very high level abstractions that are meaningful to researchers. All developments are illustrated through the use of actual code examples and case studies.

### 6.2 A critique of pure reason

As almost all software development strategy is driven by personal experience, we summarize our own recent experiences. We assume that our experience is indicative of other, comparable complex software engineering projects.

We have been involved in a many man-year software development project building a large image processing library, Horus, based on identifiable generic abstractions underlying much of the core of image processing practice [99]. This effort has been successful in capturing the core essence of image processing functionality in a compact and maintainable base of code.

The library is implemented in C++, and runs under a variety of modern operating systems. The core abstractions provided by Horus are:

- **Pixel domains**
  Sixteen types of pixels are supported by Horus. They are divided into scalar and vector types, and all commonly encountered image formats are covered by the supported Horus types.

- **Primitive pixel operations**
  Each implementation of a Horus pixel datatype must conform to an interface that specifies the exact arithmetic operations it must support. The supported pixel operations in Horus is an exhaustive list of unary, binary, and relational functions defined over the supported pixel domains.

- **Images**
  Images are instantiated over pixel domains using the C++ template mechanism. Images in Horus are containers for image data and the essential information needed by operations to interpret them. Image types are represented by signatures which encode their underlying pixel domain and dimensionality. Horus provides support for one-, two-, and three-dimensional images.
6.2. A critique of pure reason

- **Generic image operations**
  The design of the core functional abstractions of Horus is based on a number of patterns common to image processing operations. Central to our development in this chapter are the Unary Pixel Operation (UPO), Binary Pixel Operation (BPO), Reduce Operation (RedOp), and Generalized Convolution (GenConv). Each class of operations has a unique data access and arithmetic application pattern to it. Operations are instantiated using the C++ template mechanism. A complete instantiation of a generic operation requires a type for all images and a specification of all arithmetic operations.

This is a description of the abstract functional core of Horus. Additional layers of functionality are also provided to allow easier manipulation of images. A large set of functions is also included in Horus that call patterns pre-instantiated over the most common image types and operations. This minimizes the effort a user must expend instantiating individual operations. A CORBA layer is integrated into the Horus system allowing inter-operability between remote machines and added flexibility through language bindings, as well as a Java graphical user interface for interactive experimentation. Language bindings are supplied, via the CORBA layer, for Java, Perl, and indirectly to the Matlab scientific computing platform.

While Horus has been successful in satisfying the original intent and desires of its designers and software engineers, end users – both individual and institutional – have been slower to recognize its advantages. A number of complex, inter-related factors have caused this, but the reasons can be distilled down to a number of specific observations made by actual users:

1. **Horus is big.**
   As is typical of an advanced C++ library, the current Linux binary distribution of Horus consists of approximately 50Mb of shared libraries. Along with this are about three megabytes of header files, of which there are over one thousand. Considering all of this, the footprint of Horus is certainly not dainty by any measure. Aggravating this observation is the fact that most image processing problems will only require a tiny fraction of the functionality offered by the entire distribution.

2. **Horus is bafflingly complex.**
   The accuracy of the observation is certainly a matter of perspective. To the seasoned C++ programmer, Horus is not overly complex. Image processing, and computer vision in particular, is one of those fields that sits at the nexus of a great many disciplines. Engineers, mathematicians, physicists, and researchers from many other backgrounds have made important contributions. The complexity of Horus is, at least in part, a byproduct of its sound design. It was designed to be generic and maintainable. Its core abstractions were not designed to be accessible to people from such a sweeping range of backgrounds.

3. **Horus takes forever to compile.**
   Again a matter of perspective, but in part true due to items one and two above. A complete, optimized build of Horus can take hours. While this is not often necessary in practice, excessive compile times are still a concern particularly in
the prototyping stage of a project. At this stage it is important to be able to rapidly turn ideas into running code. Waiting for a compiler – or worse, trying to decipher a subtle typing error within a many-times-nested C++ template expansion – can be frustrating to say the least.

4. *Horus doesn't do what I want.*

This observation could perhaps be more accurately phrased "I don't know how to make Horus do what I want." This is perhaps the inevitable bottom line, and the natural result of the previous observations. While the library offers a broad selection of functionality over many different image types, the need inevitably arises to extend the core functionality in some way. Such extensions require researchers to deal with the size and complexity of Horus, and also tests their patience with compile times. Part of the reason for this is that the “language” of Horus is very far distanced from the natural language of mathematics. This factor becomes acutely frustrating during prototyping phases, where developed code may be immediately thrown away.

The accuracy and relevance of all the above observations can be debated endlessly. It cannot be doubted, however, that the observations are a reflection of end user perceptions of Horus as a big and complex image processing system that is difficult to use. In a sense, Horus has been successful in managing the complexity of implementing and maintaining a large image processing system, but has not adequately addressed the problems of managing the complexity of using such a system.

### 6.2.1 Analysis

Horus is representative of a modern trend toward genericity in the design of sustainable image processing and computer vision software. Libraries such as Khorus [62] and SCIL-image [59] from the previous generation of image processing software surpassed their limits of maintainability with the increasing demand for new image representations and the operators required on them. Modular design resulted in a combinatorial expansion in the codebase that had to be maintained when integrating new image types and operators. Advances in software engineering suggested that object oriented and generic design might be the solution.

The Image Understanding Environment (IUE) is an example of object oriented design of image processing software [49]. It defines a class hierarchy that abstracts the concepts of image processing data and functionality. The library makes use of the design patterns of object oriented software engineering to achieve maximal sharing of code between operation defined over varying datatypes.

The VIGRA computer vision library uses STL-style genericity to provide implementations of image processing functionality as C++ template instantiations [64]. It is comparable to Horus in this respect, and our motivations for re-abstraction away from expressing image processing and computer vision theory in C++ syntax are equally valid with VIGRA as a model.

In our approach, we have decided to take some of the design successes of modern image processing environments, and concentrate on building tools to support image processing and computer vision research that are more suited to each stage of the experimental research process.
In some sense, the core of image processing software environments based on generic design is not composed of image processing functions, but rather generic recipes for instantiating image processing functions. In many libraries, instantiated recipes of many functions for every type of image representation are pre-defined so users did not have to cope with the complexity of instantiating their own functions. Figure 6.1 provides an example of what is required to instantiate an image processing operating in Horus [58]. It is provided only as an example of the complexity involved. Instantiating similar operations can be accomplished by applying the cut-and-paste paradigm and making simple modifications. The process is, however, burdensome and syntactically tedious.

Even researchers who are experienced C++ programmers do not usually reason
in terms of objects, iterators, and template instantiations when initially developing a theory. The language of C++ is very far from the mathematical languages of partial differential equations and mathematical morphology. This distance causes frustration and can lead to subtle errors introduced in the process of translating theory into executable code.

The needs of a researcher change and evolve during the course of a project. In the initial stages, when he is usually interested in playing with an idea on only a handful of images, interaction and responsiveness are very important. He wants to see as quickly as possible whether an idea has merit and is worthy of deeper investigation. We have therefore concentrated on developing interactive systems that can be dynamically reconfigured at runtime. When an idea has evolved to the point where a researcher wants to test it on several thousand images, such interactive abstractions become less meaningful.

### 6.3 Design considerations

#### 6.3.1 Goals

From the analysis in the previous section we established several goals for our image processing environment:

- **Functionality on demand**
  We concentrate on providing *only* the functionality required at any specific time, and the tools to provide rapid (re-)configuration as needed.

- **Relevant and meaningful abstractions**
  Emphasis is placed on building tools capable of providing abstractions that are more meaningful to a researcher developing a new idea. At the very least, these abstractions should allow a researcher to play with ideas before committing them to a more laborious implementation in a native image processing environment.

- **Interaction, flexibility, and scalability**
  What we are proposing is an interactive system for experimental image processing. This interactivity should be flexible in expression, and also in scalability. When interaction is no longer important, efficient implementations must be possible.

- **Minimal effort**
  The implementation of the goals above should be transparent to the user. We strive for seamless integration of functionality and different levels of interaction, requiring minimal effort from the user.

#### 6.3.2 Choice of language

A prime motivation for our approach is the observation that most theories in computer vision and image processing treat images as *functions* rather than numerical arrays. Theories are derived in functional form and then separately discretized for implementation. If researchers wish to treat images as functions, it makes sense to defer
discretization decisions as much as possible, preferably until after a working prototype of the theory has been developed. Satisfying this need requires the use of languages that do not discriminate against functions, or otherwise designate them to some special category. In other words, a programming language that treats functions as first class objects that can be created, modified, and returned as values from procedures is needed.

While the concept of functions as first class objects is central to the design of all functional programming languages, another important consideration for our requirements is the proper handling of types. Experience has shown that in practical, working situations, a programmer can expend a great deal of effort and experience much frustration trying to get the types correct. A single image processing application can require the handling of many different types of numerical image representations, e.g. byte, integer, floating point, etc., and assembling the appropriate functions for the desired types can be difficult even for simple programs. Most imperative programming languages, and indeed most functional languages, do not provide the sort of flexibility and safety needed to ensure correct program executing through strict static type checking.

Much research in the programming language community has concentrated on the correct, static typing of programs during compilation. These efforts culminated in the early eighties with the definition of the programming language ML (standing for Meta-Language) [75]. ML was one of the first languages implementing a system for polymorphic type inference, in which the types of higher order objects such as functions need not be explicitly specified, but can be inferred from the constraints on its constituent elements. Modern descendants in the ML family include SML/NJ [85, 111, 41], Haskell [50], and OCaml [21, 23].

All of the modern dialects of ML posses the same core features of ML, but OCaml is distinguished in many ways, and was selected for our application for the following reasons:

- **OCaml is an impure functional programming language**
  OCaml supports a broad range of imperative features in addition to its functional core. Mandating conformance to a pure functional programming paradigm is too restrictive for image processing applications, and OCaml's looping constructs, arrays, and mutable data structures offer the comfort and flexibility needed. Great effort has been expended by the language designers to ensure the type soundness of these imperative features, and their use does not affect the ability of the OCaml compiler to ensure type safety in the resulting executables.

- **OCaml has an efficient, native code compiler**
  The standard OCaml distribution includes a native code compiler capable of generating very efficient code for a variety of architectures and operating systems [10]. In particular, OCaml benchmarks favorably in comparison to C and C++ for array and matrix manipulation.

- **OCaml interfaces well with the outside world**
  OCaml will never be the host language for large-scale experimental or production vision systems. Despite the impressive benchmarks of its native code compiler, OCaml's place in our framework is for prototyping operations quickly. OCaml provides means for seamlessly interfacing with foreign functions and data. We
will exploit this feature to maintain relationships with the foreign type system of Horus, so that OCaml prototypes can be transparently transformed into efficient, C++ implementations.

- **OCaml allows for dynamic, compile-tile syntax extension**

OCaml is no more the language of image processing and computer vision then C++. In OCaml, however, syntax is a malleable concept. Through the use of the camlp4 pre-processor it is possible to dynamically extend the core syntax of the language. It operates on the abstract syntax trees of the OCaml language during parsing, and as such should be distinguished from the usual conception of pre-processor in C-like languages. With it we can provide concrete syntactical constructions that are more meaningful to users.

To summarize, we found OCaml to posses the right combination of features for our purpose. It is a modern functional programming language, allowing programs to manipulate functions just as they would any other type. Its polymorphic type system enables to the compiler to statically ensure type safety of programs, eliminating many programming errors that would otherwise only be detected disastrously at runtime. The rest of this chapter assumes that the reader is familiar with the syntax and concepts of OCaml. For those readers who are not, a good starting point is the OCaml website [81]. A book on application development in OCaml is also available [21].

### 6.3.3 Previous work

There have also been a number of functional approaches to the design of image processing tools. The Envision system is an extension to the Scheme programming language (a modern dialect of LISP) specifically designed to support image processing [107]. The intentions of the Envision designers resonate very well with our own in that they see the shift toward C++-style genericity as a stopgap measure in the struggle to achieve truly generic, maintainable, and portable image processing functionality. They too recognize that changing requirements create race conditions between users and designers of image processing tools. Users demand specific functionality, and implementors race to implement it, hoping that the requirement will last at least as long as development. The Envision system satisfies our goal for interaction and flexibility, but provides no explicit means for scaling applications. We focus on modeling the functionality of native-code software environments to support this.

The LISP Universal Shell (Lush) is another functional approach to image processing [71]. Its design strategy is closest to our own. The Lush interpreter supports inline C compilation in Lisp source code. Because of this, C and Lisp code can manipulate the same data, allowing for the same time of interaction and scalability as we are aiming for. In fact, the first prototype of the DjVu web-centric document distribution system was implemented in Lush [39]. The system is semi-monolithic. It includes thousands of image processing and machine learning functions. Our approach differs from the designers of Lush in two important ways. We deliberately hide the backend implementation of optimized functions, rather than allowing the user to directly manipulating data with native code. Our design is also decidedly non-monolithic. One of our primary goals is to achieve the most functionality with the smallest amount of code.
Figure 6.2: The proposed system architecture. Users interact with the system through the OCaml toplevel. The core abstractions of the OCaml implementation are models of Horus core abstractions.

6.4 Architecture

The architecture of our proposed system is illustrated in figure 6.2. Each of the core abstractions of Horus are modeled in the OCaml implementation, allowing fully functional image processing operations to be implemented entirely in OCaml. This is a crucial requirement of our system. In order to achieve the level of interaction required, image processing operations must be definable and manipulable in OCaml.

The modeled core abstractions and the code generator interact to create instantiated Horus implementations of image processing functions. Functions in the configured Horus backend are transparently substituted into the OCaml system at runtime. Image processing operations have a dual existence. They are prototyped interactively in OCaml, code is generated and compiled, and the foreign implementation is imported back into the OCaml imaging system in place of the prototype. The interface to functionality does not change, only the efficiency.

In this architecture users are isolated from the details of backend library creation. The foreign functions of the Horus backend conform to the same interface as the OCaml ones, but the details of code generation, compilation, and dynamic linking are completely hidden from the user. By this, the system is used as a configuration tool as well as an experimental image processing environment.

6.5 Primitive types and operations

In this section we describe all of the basic types and operations in our system. We begin with a discussion of the OCaml types used to represent the physical and algebraic structure of images. We only include representative examples of primitive operations for brevity. Details of external image representation and I/O are omitted for clarity.
6.5.1 Types and typing

The discussion begins, quite naturally, with the abstract, polymorphic datatype used to encapsulate information about images:

```ocaml
type ('a, 'b) cap = {
  width : int;
  height : int;
  data : ('a, 'b, Bigarray.c_layout) Bigarray.Array2.t
}
```

This data structure is parametrized by the two polymorphic type parameters ('a, 'b), which are placeholder types for the internal OCaml representation of image data and the external native data representation, respectively. This type of parameterization allows specific image types (discussed below) to define their own internal and external data representation formats. Note that this data type declaration doesn’t do or define anything except for constraints on structures of type cap (short for *capsule*). For now the only constraints imposed are that a cap contain an integer width and height, an OCaml two-dimensional BigArray in c_layout format. Only when the type parameters ('a, 'b) are supplied in concrete cap implementation will this type become concrete.

Now we will define the core abstract image types. This is done in three stages. First, we define the Base module type which uses the cap structure above and supplies essential functions for creating, reading, and writing images:

```ocaml
module type Base =
  sig
    type dom
    type ext
    type cbpo_t = (dom —> dom —> dom)
    type cupo_t = (dom —> dom)
    val read : string —> (dom, ext) cap
    val write : (dom, ext) cap —> string —> unit
    val create : int —> int —> (dom, ext) cap
  end
```

This module defines the two types dom and ext, which provide the internal and external data representation needed to fully instantiate the cap type for an image type. Note how the read, write, and create functions use the (concrete) type (dom, ext) cap for the images they take and return. The other two types defined in the Base module provide our first glimpse of the algebraic structure with which we will equip our images. The type cbpo_t, short for *Concrete Binary Pixel Operation (type)*, is the type signature to which some algebraic pixel operations must conform. Similarly, the cupo_t type (*Concrete Unary Pixel Operation*) is the signature for unary operations.

We now define the interface signature for the ConcreteImage type, which contains all of the Base types and functions, and specifies exactly the algebraic pixel operations...
which must be supplied by any image type implementation:

```ocaml
module type ConcreteImage =
  sig
    include Base
    val add : cbpo_t
    val sub : cbpo_t
    val mul : cbpo_t
    val div : cbpo_t
    val neg : cupo_t
  end
```

Only a few example algebraic pixel operations are included above. The supported pixel operations need not be limited to the above examples, but to include more here would be distracting. Indeed, any function conforming to the cbpo\.t or cupo\.t signatures can be included as needed.

Now we examine a concrete implementation of a basic image type. Figure 6.3 shows the implementation of images of floats (called scalar float images to distinguish them, for example, from images of float vectors that might be used to implement color images). Structurally, there is not much difference between the implementation and the signature for ConcreteImage defined above. Most importantly, however, is the inclusion of concrete types for dom and ext, concretizing cbpo\.t and cupo\.t in the process. Implementations of all of the arithmetic pixel operations are also supplied.

This formulation of the basic types and operations is satisfactory from an operational viewpoint, but from a design perspective it is desirable to hide the implementation of all arithmetic operations defined over pixel domains. At the same time it is necessary to not hide the actual type of the underlying domain (so that constant pixel values can be easily created, for example). We will accomplish this goal by hiding the types of the arithmetic operations in a separate module. Two new polymorphic pixel operation types are introduced in the visible interface:

```ocaml
type 'a bpo\.t
type 'a upo\.t
```

whose (hidden) implementation is provided as:

```ocaml
type 'a bpo\.t = 'a -> 'a -> 'a
type 'a upo\.t = 'a -> 'a
```

Since the actual implementation is hidden in the interface, the types of unary and binary pixel operators are abstract and cannot be manipulated outside of the module implementing them.

A new image type is now defined, mirroring the ConcreteImage signature, but
File: pixalgebra.ml

(* The concrete implementation of images of floats *)
module ScalarfloatConc =
  struct
    (* Internally OCaml floats, externally 32-bit floats *)
    type dom = float
    type ext = float32_elt

    (* Primitive pixel op types *)
    type cbpo_t = (dom -> dom -> dom)
    type cupo_t = (dom -> dom)

    (* I/O: by default, images are normalize to 0-1, i.e. /255 *)
    let read s = ...
    let write c s = ...

    let create w h =
      { width = w;
        height = h;
        data = Array2.create float32 c_layout h w }

    (* The arithmetic ops in person *)
    let add = (+ .)
    let sub = (- .)
    let mul = (* .)
    let div = (/ .)
    let neg a = -.a
  end

Figure 6.3: A structure implementing images of scalar floats. The structure supplies concrete types for all abstraction. The arithmetic operations are implemented using the OCaml operators for floats.

using the new operator types:

module type AbstractImage =
sig
  include Base
  val add : dom bpo_t
  val sub : dom bpo_t
  val mul : dom bpo_t
  val div : dom bpo_t
  val neg : dom upo_t
end

And lastly, a functor is used to lift a ConcreteImage to an AbstractImage. The functor
is just a fancy identity operator the copies all of the functions and types from the ConcreteImage, constraining the types we want to hide through the AbstractImage signature:

```ocaml
module PrimeImage = functor (I : ConcreteImage) ->
  struct
    type dom = I.dom
    type ext = I.ext
    let read = I.read
    let write = I.write
    let create = I.create
    let add = I.add
    let sub = I.sub
    let mul = I.mul
    let div = I.div
    let neg = I.neg
  end

The interface to the ScalarFloat pixel algebra is exposed as:

module Scalarfloat : AbstractImage with type dom = float
module ScalarfloatConc : ConcreteImage with type dom = float

and the implementation is provided by the appropriate functor call:

module Scalarfloat = PrimeImage(ScalarfloatConc)

This hiding of the actual pixel operator types is a bit too restrictive; the very fact that they are functions are hidden from the user. It is sometimes necessary to create images directly from OCaml functions - for convenient construction of convolution kernels, for example. We can relax these constraints by providing application functions that take binary and unary pixel operation from the image structure and apply them, as well as an of_fun function to create an image from a function given a support range. Their interface is defined as:

```ocaml
val bapply : ('a bpo_t) -> 'a -> 'a -> 'a
val uapply : ('a upo_t) -> 'a -> 'a
val of_fun : (int -> int -> 'a) -> int -> int -> ('a, 'b) cap
```

and are implemented as:

```ocaml
let bapply f a b = f a b
let uapply f a = f a
let of_fun f w h =
  let r = I.create w h in
  for y = -(h / 2) to (h / 2) do
    for x = -(w / 2) to (w / 2) do
      r.data.{y + (h / 2), x + (w / 2)} <- f x y
    done;
  done;
```
Chapter 6. A functional approach to software design in image processing research environments

Note that by using polymorphic types in the AbstractImage definition and for application functions, they can be used for any image type, and need not be duplicated for all image implementations.

We now have all of the basic elements for pixel domains. All I/O operations have been included to allow creating and saving of images, and we have defined an algebraic structure that must be provided in all pixel implementations. Further, we have ensured the safety of our implementation by hiding it behind an abstract module. Next, the processes may be discussed by which we lift these primitive pixel operations to images. By abstracting the pixel domains underlying images we are able to define and constrain the set of valid algebraic operations on pixels.

6.5.2 Primitive image operations

Pixel domains were defined with two fundamental pixel operation types: unary and binary. Unary image operations over images take a single input image, apply a unary pixel operation to every pixel, and return the result. Binary image operations take as input two images, apply a binary pixel operation to the corresponding pixels in the two inputs, and return the result. It is natural that these primal types are the first to consider. As they form the basic operation in our algebra of pixels, so shall they in the algebra of images we will now define.

Binary pixel operations

We start by defining the module type used to encapsulate the type information about our binary pixel operations:

```ocaml
module type Bpo =
  sig
  type dom
  type ext
  val op : (dom, ext) cap -> (dom, ext) cap -> (dom, ext) cap
  val op_val : dom -> (dom, ext) cap -> (dom, ext) cap
end
```

This Bpo type contains all of the type information about the underlying pixel domain, and also requires two functions: op which takes two images and returns another, and op_val which takes a pixel value from the underlying domain, an image over the same domain, and returns an image.

All that is required to instantiate a binary pixel operation over images is an AbstractImage and a binary pixel operation from the same AbstractImage. This is done again through an OCaml functor, shown in figure 6.4. Note that, as required by the signature, the functor provides implementations of op, which takes two images and returns the result image, and op_val which allows us to replace one of the images with a constant value. Instantiation and use of these operations is shown in the example program in figure 6.5.
File: pixalgebra.ml

module MakeBpo =
  functor ( I : AbstractImage ) ->
    functor ( O : sig val op : I.dom bpo.t end ) ->
  struct
    type dom = I.dom
    type ext = I.ext
    let op im1 im2 =
      let result = I.create im1.width im1.height in
      let (dl, d2, r) = (im1.data, im2.data, result.data) in
      for y = 0 to im1.height - 1 do
        for x = 0 to im1.width - 1 do
          r.{y,x} <- (O.op dl.{y,x} d2.{y,x})
        done; done; result
    end

Figure 6.4: The OCaml type functor used to lift a binary pixel operation to binary image operations.

File: util.ml

open Pixalgebra
open Scalarfloat

(* Instantiate the operations we'll need *)
module BpoAdd = MakeBpo(Scalarfloat)(struct let op = Scalarfloat.add end)
module BpoMul = MakeBpo(Scalarfloat)(struct let op = Scalarfloat.mul end)

(* Load the two images, halve them *)
let trui = BpoMul.op_val 0.5 (Scalarfloat.read "trui.tif")
let sche = BpoMul.op_val 0.5 (Scalarfloat.read "schema.tif")

(* Add the images, write out the result *)
let _ = Scalarfloat.write (BpoAdd.op trui sche) "result.tif"

Figure 6.5: A simple example program illustrating the instantiation and use of binary pixel operations on images.
Note also that this satisfies one of our original requirements. The program shown in figure 6.5 only instantiates the image operations that are absolutely necessary, and this is accomplished with a minimum of syntactic fuss as compared to the C++ example in figure 6.1.

**Unary pixel operations**

Unary pixel operations are lifted to operate over images in the same way binary pixel operations were in the previous section. We begin with a type signature for Upo modules:

```plaintext
module type Upo =
  sig
    type dom
    type ext
    val op : (dom, ext) cap -> (dom, ext) cap
  end
```

and a functor to make a Upo from an AbstractImage and a unary pixel operation:

```plaintext
module MakeUpo =
  functor ( I : AbstractImage ) ->
    functor ( O : sig val op : I.dom upo.t end ) ->
      struct
        type dom = I.dom
        type ext = I.ext
        let op im =
          let result = I.create im.width im.height in
          let di = im1.data in
          let r = result.data in
          for y = 0 to im.height - 1 do
            for x = 0 to im.width - 1 do
              r.{y,x} <- O.op di.{y,x}
            done;
            done;
            result
          end
```

Unary operations on images are then instantiated in the same way as binary ones:

```plaintext
module UpoNeg = MakeUpo(Scalarfloat)(struct let op = Scalarfloat.neg end)
```

Note that unlike the Bpo interface, unary pixel operations do not require an op_val operation.

**Reduce operations**

The previous section shows how primitive binary and unary pixel operations can be lifted to operate on images, instantiating operators which take and return images. These operations were completely specified by a primitive pixel operation and a generic
module MakeRedOp =
  functor (I : AbstractImage) ->
    functor (O : sig val op : I.dom bpo_t val neut : I.dom end) ->

  struct
    type dom = I.dom
    type ext = I.ext
    let neut = O.neut
    let op im =
      let w = im.width in
      let h = im.height in
      let d = reshape_1 (genarray_of_array2 im.data) (w * h) in
      let rec my_fold c v =
        if c == (w * h) then v else my_fold (c + 1) (O.op v d.{c}) in
      my_fold 0 neut
  end

Figure 6.6: The OCaml functor used to instantiate a reduce operation from a pixel domain, and binary pixel operation, and a neutral element from the domain.

recipe for lifting it to the correct operation over images. All of the remaining image operations are meta-operations, in that they will lift already instantiated operations to form higher-level ones. The next important class of image operations are reduction operators which take an image and return a pixel value. A reduce operation can be instantiated from any binary pixel operator.

module type RedOp =
  sig
    type dom
    type ext
    val neut : dom
    val op : (dom, ext) cap -> dom
  end

Like all of the previous operations, the RedOp type maintains information about the underlying pixel domain, and in addition keeps a neutral element neut used to initialize the operation. This neutral element will depend on the binary operation used to instantiate the RedOp.

Reduce operations correspond naturally to the fold operation on lists in functional programming languages. The implementation of the RedOp functor is shown in figure 6.6. In our implementation the image is converted into a one dimensional array and apply a tail recursive fold operation, which allows the OCaml compiler to aggressively optimize the operation. The neutral element neut is used, of course, as the initial value for the fold.
106  Chapter 6. A functional approach to software design in image processing research environments

**File: util.ml**

```ml
open Pixalgebra
open Scalarfloat

module Bpo = MakeBpo(Scalarfloat)
module Red = MakeRedOp(Scalarfloat);
module BpoAdd = Bpo(struct let op = Scalarfloat.add end)
module BpoMul = Bpo(struct let op = Scalarfloat.mul end)

module RedMin =
    MakeRedOp(struct let op = Scalarfloat.min let neut = 256.0 end)
module RedMax =
    MakeRedOp(struct let op = Scalarfloat.max let neut = 0.0 end)

let sche = Scalarfloat.read "house1.tif"

let s = RedMin.op sche
let b = RedMax.op sche

let result = BpoMul.op_val (255.0 /. (b -. s)) (BpoAdd.op_val (.s) sche)
let _ = Scalarfloat.write result "result.tif"
```

Figure 6.7: An example of instantiation and use of binary and reduce operations. This is the first example program, which reads in an image and stretches the grey values to achieve a minimum and maximum of zero and one.

---

**Generalized convolution operations**

The convolution operation can well be called the workhorse of image processing and computer vision. The standard convolution of function \( f \) with function \( g \) is defined
discretely as:

\[ (f \ast g)(x, y) = \sum_{u=x-\delta_x}^{x+\delta_x} \sum_{v=y-\delta_y}^{y+\delta_y} f(x, y)g(x-u, y-v), \]

where the sum is taken over a local rectangular neighborhood around \((x, y)\) parameterized by its width \(\delta_x\) and height \(\delta_y\).

This convolution functional can be generalized by observing that it is naturally decomposed into two arithmetic operations. In the convolution operation defined above, we can replace the multiplication of \(f\) and \(g\) with an arbitrary binary pixel operation, and the summation with an arbitrary reduce operation. Given an instantiated binary pixel operation and reduce operation, we define a parameterized \text{GenConv} signature as:

```ocaml
module type GenConv =
  sig
    type dom
    type ext
    val op : (dom, ext) cap \rightarrow (dom, ext) cap \rightarrow (dom, ext) cap
  end
```

Note that, but for the exclusion of the \text{op\_val} function, the signature for generalized convolutions is identical to that of binary pixel operation.

As with all image operations, \text{GenConv} implementations are created using OCaml type functors. Figure 6.8 shows the functor implementing generalized convolutions. The generalized convolution pattern performs many array accesses, and all internal scratch arrays are implemented using native OCaml arrays for efficiency. All temporary scratch arrays are created using the \text{create} function of the underlying pixel domain. At its core, the generalized convolution operation used the instantiated binary and reduce image operations to compute the result for each pixel. Figure 6.9 presents an example of a generalized convolution used to perform simple image sharpening.

These are the generic recipes for lifting operations from the pixel domain to images. The patterns are common to many image processing operations. Abstraction of the underlying domain allows genericity in that all operations can be instantiated over structures conforming to the \text{AbstractImage} signature.

### 6.6 Backend substitution

It is now time to reap the benefits of our efforts to maintain, through abstraction, the relationship between the native OCaml implementation and the foreign Horus implementation. This was done so that efficient, native-code implementations of image processing operations can be immediately generated from instantiated OCaml operations. In this way, the OCaml implementation can be thought of as a specification of desired functionality. In our sample implementation we have carefully chosen pre-existing Horus patterns and algebraic pixel operations, and inflexibly required patterns to be instantiated with pixel operations defined in an \text{AbstractImage} type which hides the implementation details of primitive pixel operators.
The substitution begins by extending the ConcreteImage type with extra information about how operations over a specific pixel type should be implemented in the Horus backend:

```ml
module ScalarfloatConc =
  struct
    ...
    let hxtype = "HxScalarDouble"
    let hxdata = "float"
    let hxptr = "HxDataPtr2dFloat"
    let hxbigarray = "BIGARRAY_FLOAT32"
    ...
  end
```

In brief, these additional elements tell the system that the Horus arithmetic type (C++ class) that implements pixels of this particular OCaml type is HxScalarDouble, that
File: sharpen.ml

```ocaml
open Pixalgebra
open Scalarfloat

module A = Scalarfloat

module BpoMul = MakeBpo(A)(struct let op = A.mul end)
module RedAdd = MakeRedop(A)(struct let op = A.add let neut = 0.0 end)
module GCMulAdd = MakeGenConv(A)(RedAdd)(BpoMul)

let sharpen_k k =
  let f x y = match x, y with
    | (0, 0) -> 8.0 *. k +. 1.0
    | (_, _) -> (-.k)
  in
    A.of_fun f 3 3

let sharpen_im k = GCMulAdd.op im (sharpen_k k)
```

Figure 6.9: An example instantiation and use of a generalized convolution. A sharpening kernel and function to perform sharpening are defined.

pixels are represented by the native C++ type float, that the Horus data pointer type is HxDataPtr2dFloat, and that the interface between OCaml and Horus is made using an OCaml Bigarray of type BIGARRAY_FLOAT32.

This information establishes the link between our native OCaml types and the types underlying the Horus image processing system. To complete the link between OCaml and Horus, we annotate each primitive pixel operation in ConcreteImage with the corresponding Horus operations. The new types for concrete primitive binary and
Unary operations is:

```ocaml
module type Base = sig
  ...
  type cbpo_t = (dom -> dom -> dom) * string * string
  type cupo_t = (dom -> dom) * string
  ...
end
```

where each operator is now represented as a tuple containing the actual OCaml operator and one or more additional strings annotating the operation with the corresponding operator name from the Horus pixel implementation.

As a specific example, these are the concrete operators from the ScalarfloatConc type:

```ocaml
module ScalarfloatConc = struct
  ...
  let add = ( .+ ), "operator+", "operator+="
  let sub = ( -. ), "operator-", "operator-="
  let mul = ( *. ), "operator*", "operator**"
  let div = ( /. ), "operator/", "operator/="
  let min = min, "min", "minAssign"
  let max = max, "max", "maxAssign"
  let neg a = -.a, "operator--"
  ...
end
```

Note that each binary operation is equipped with a string representing the Horus operator used for generating a binary image operator and an image reduce operation. For this reason operations of type cbpo_t are annotated with composite assignment operators as well as the corresponding arithmetic infix operator. The PrimeImage functor remains unchanged, as all it does is lift the ConcreteImage to an AbstractImage, hiding the actual representation of operations in the process.
Next, each functor defining image operations is equipped with a new function that generates Horus-compatible C++ code using skeletons similar to the code shown in figure 6.1. This is illustrated with the MakeBpo functor:

```ocaml
module MakeBpo = functor (I : AbstractImage) ->
  functor (O : sig val op : I.dom bpo_t end) ->
struct
  ...
let gen f op_name =
  let h xtype, hx data, (_, op_func, _) = I.h xtype, I.hx data, O.op in
let neut = 0 in
let fp = open_out ("hxbuild/" ^ f) in
output_string fp interpolate file "bpo_skel.txt";
  close_out fp;
  name := "Hx" ^ op_name ^ "_caml"
end
```

This function generates the instantiation for the Horus operation corresponding to this Bpo using the information now provided in AbstractImage. The C++ code for the instantiated Horus operation is placed in a special hxbuild directory, which serves as a repository for generated code in our system.

The system maintains and imports a shared library of foreign functions (in pixalgebra.ml):

```ocaml
(* A reference to the shared library supplying foreign functions *)
let chorus_lib =
  try
    ref [(Dl.dl_open "dllChorus.so")]
  with Sys_failure e -> print_endline e; ref []

(* This function does a build of all generated code *)
let build_lib = fun () ->
match !chorus_lib with
  | (lib :: []) ->
    Dl.dl_close lib;
    Unix.system "cd hxbuild; hxmake.sh";
    chorus_lib := [(Dl.dl_open "dllChorus.so")]
  | [] ->
    Unix.system "cd hxbuild; hxmake.sh";
    chorus_lib := [(Dl.dl_open "dllChorus.so")]
```

When the imaging system starts up, the shared library dllChorus.so is loaded. The function build_lib can be called after generation of all operations, i.e. after calls to the gen function of an instantiated Bpo, forcing a build of all new generated code. Calls to build_lib also force a re-load of the dllChorus.so library, refreshing any pre-existing symbols that may have been loaded.

Finally, once an operation has been instantiated and generated, and the library has been built, the following OCaml type functor is used to replace the existing OCaml implementation with the native Horus code (only the code specific to the implementation
of the binary image operation is included, again from pixalgebra.ml):

```ocaml
module MakeFastBpo =
  functor ( R : Bpo ) ->
  struct
  ...
  let op =
    let bpo_wrap f =
      fun (im1 : (dom, ext) cap) (im2 : (dom, ext) cap) ->
      { width = im1.width;
       height = im1.height;
       data = f im1.data im2.data } in
    let lib = List.hd !chorus_lib in
    let sym = Dl.dl_sym lib !name in
    bpo_wrap (Dl.call2 sym)
  ...
  end
```

This functor replaces the OCaml operation from an instantiated Bpo with the corresponding operation from the already loaded dllChorus.so library. The most important feature of this functor, and indeed of this entire section, can be seen by looking at the signature of this functor (from pixalgebra.mli):

```ocaml
module MakeFastBpo :
  functor ( R : Bpo ) ->
  Bpo with type dom = R.dom and type ext = R.ext
```

The functor takes an instantiated Bpo and also returns a Bpo. That is, the fast functions conform to the same interface as the original Bpo. Because of this, and because we have maintained the relationship between OCaml and Horus types, any pre-existing code using the OCaml Bpo will now work exactly as before, but with the optimized Horus backend.

As an example, figure 6.10 shows how the sharpening convolution of figure 6.9 can be made more efficient by generating optimized Horus implementations of the required operation.

The developments in this section built directly on the abstraction of primitive pixel and image operations, whose definition was tightly constrained to conform to a model of the core abstractions of Horus. By constraining the abstraction of pixel domains and generic patterns to model their analogues in a foreign, native-code image processing library we are able to automatically generate efficient, optimized code to replace the OCaml implementations of low-level image processing functions. The interface to the substituted foreign functions is identical to the OCaml implementation.

### 6.7 Case studies

In this section usage patterns for the system are examined.
File: sharpen.ml

```ml
open Pixalgebra
open Scalarfloat

module A = Scalarfloat

module BpoMul = MakeBpo(A)(struct let op = A.mul end)
module RedAdd = MakeRedOp(A)(struct let op = A.add let neut = 0.0 end)
module OcGCMulAdd = MakeGenConv(A)(RedAdd)(BpoMul)

(* NEW: generate code for convolution, build library, and make fast *)
let _ = OcGCMulAdd.gen "GCMulAdd.c" "GCMulAdd"
let _ = build_lib ()
module GCMulAdd = MakeFastGenConv(OcGCMulAdd)

let sharpen k k =
  let f x y = match x, y with
   | (0, 0) -> 8.0 *. k +. 1.0
   | (_, _) -> (-.k)
  in
  A.of_fun f 3 3

let sharpen im k = GCMulAdd.op im (sharpen k k)
```

Figure 6.10: The sharpening function revisited. The generalized convolution is replaced by an optimized, native-code version.

## 6.7.1 Linear scalespace

Linear scalespace theory is quite popular in the image processing and computer vision research fields [61]. The basic idea is to take an input image $I$ and embed it in a one parameter family of smoothed images by convolving it with a Gaussian convolution kernel:

$$u_f(x;\sigma) = \int_{y \in \mathbb{R}^2} I(y)g(x - y;\sigma)dy$$

where $g(x;\sigma)$ is the isotropic two dimensional Gaussian:

$$g(x;\sigma) = \frac{1}{2\pi\sigma^2}e^{-\frac{|x|^2}{2\sigma^2}}.$$  

The scale parameter $\sigma$ controls the amount of smoothing applied to the image. Increasing $\sigma$ results in simpler images with details below a certain spatial scale reduced and eventually removed.

One nice property of such representations is that derivatives of the image can be taken simply by taking the derivative of the Gaussian kernel and performing the con-
Chapter 6. A functional approach to software design in image processing research environments

File: scalespace.ml

open Pixalgebra

module A = Scalarfloat.Scalarfloat

module BpoMul = MakeBpo(A)(struct let op = A.mul end)
module RedAdd = MakeRedOp(A)(struct let op = A.add let neut = 0.0 end)
module GCMulAdd = MakeGenConv(A)(RedAdd)(BpoMul)

let rec h n x = match n with
  | 0 -> 1.
  | 1 -> 2. *. x
  | _ -> 2. *. x *. (h (n - 1) x) -.
  | 2. *. (float_of_int (n - 1)) *. (h (n - 2) x)

let gld s x =  
  let pi = 3.14159265358 in
  let norm = 1.0 /. (s *. (sqrt (2.0 *. pi))) in
  norm *. (exp (-. (x ** 2.0) /. (2.0 *. s ** 2.0)))

let gld_h s ox = fun xi _ ->
  let x = (float_of_int xi) in
  (-.1.0 /. (s *. (sqrt 2.0))) ** (float_of_int ox) *. (h ox (-. x /. (s *. (sqrt 2.0)))) *. (gld s x)

let gld_v s ox = fun x y -> gld_h s ox y x

let do_sep_gaus i s ox oy =
  let sz = 2 * (int_of_float (3. *. s +. 0.5)) + 1 in
  let h = A.of_fun (gld_h s ox) sz 1 and v = A.of_fun (gld_v s oy) 1 sz in
  GCMulAdd.op (GCMulAdd.op i h) v

Figure 6.11: Gaussian scalespace in 30 lines of OCaml code. This collection of functions implements Gaussian convolution at arbitrary scale and order of differentiation.

such Gaussian derivative operators are widely used for edge detection, image smoothing, and sharpening.

A complete implementation of Gaussian scalespace functionality is given in figure 6.11. The code in figure 6.11 provides a function capable of computing Gaussian derivatives of any order and any scale, hence it represents the basic building blocks of scalespace. By way of explanation, the implementation of Gaussian scalespace is developed in the interactive OCaml toplevel.

First, all operations will be defined to operate in the Scalarfloat domain, which is natural since the Gaussian kernels are defined on the reals. Development begins by
opening modules and creating an alias for the Scalarfloat image domain:

```pascal
# open Pixalgebra;
# open Scalarfloat;
# module A = Scalarfloat;;
module A :
  sig
  ...
end
```

Note that we omit the lengthy signature definitions returned by many functions, and replace them with ellipses in the source fragments. The aliasing of Scalarfloat allows more generality and brevity in what follows.

Next, to perform convolutions we must instantiate all of the operations necessary. We need a Bpo implementing the multiplication operation, a RedOp to sum the pixel values in an image (or neighborhood, actually), and a GenConv instantiated using these operations:

```pascal
# module BpoMul = MakeBpo(A)(struct let op = A.mul end);;
module BpoMul :
  sig
  ...
end
# module RedAdd = MakeRedOp(A)(struct let op = A.add let neut = 0.0 end);;
module RedAdd :
  sig
  ...
end
# module OcGCMulAdd = MakeGenConv(A)(RedAdd)(BpoMul);;
module OcGCMulAdd :
  sig
  ...
end
```

The backend implementation is finalized by generating optimized Horus code for the generalized convolution:

```pascal
# OcGCMulAdd.gen "GCMulAdd.c" "GCMulAdd";;
- : unit = ()
# build_lib ();;
- : unit = ()
# module GCMulAdd = MakeFastGenConv(OcGCMulAdd);;
module GCMulAdd :
  sig
  ...
end
```

This is not strictly necessary, but will make our interactive development of the Gaussian convolution code more responsive and is a good illustration of the type of progressive prototyping and implementation supported by the system.
Now that we have a configured, compiled, and optimized backend providing no more than the basic functionality needed, the conceptual heart of the task at hand is the generation and application of Gaussian convolution kernels. We will use the fact that the two dimensional Gaussian kernel is separable into horizontal and vertical components to make our implementation efficient and elegant. That is:

\[ g(x; \sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{|x|^2}{2\sigma^2}} = g(x; \sigma)g(y; \sigma) , \]

where \( g(x; \sigma) \) and \( g(y; \sigma) \) are one dimensional Gaussians.

To enable the generation of Gaussian derivative kernels of any order of differentiation, we will use the fact that Gaussian derivative functions of any order may be generated using the Hermite polynomials, which are defined recursively as follows:

\[
H_n(x) = \begin{cases} 
1 & \text{if } n = 0 \\
2x & \text{if } n = 1 \\
2xH_{n-1}(x) - 2(n-1)H_{n-2}(x) & \text{otherwise}
\end{cases}
\]

The univariate Gaussian derivative of order \( n \) can then be written as:

\[
\frac{d^n}{dx^n} g(x; \sigma) = \left( -\frac{1}{\sigma\sqrt{2}} \right)^n H_n \left( -\frac{x}{\sigma\sqrt{2}} \right) g(x; \sigma)
\]

We now turn these definitions into working code. We first need the zero order univariate Gaussian:

```plaintext
# let glid s x =
let pi = 3.14159265358 in
let norm = 1.0 /. (s * (sqrt (2.0 * pi))) in
norm * (exp (-. (x ** 2.0) /. (2.0 * s ** 2.0)));;
val glid : float -> float = <fun>
```

and a function to compute the Hermite polynomial of any order:

```plaintext
# let rec h n x = match n with
| 0 -> 1.
| 1 -> 2. *. x
| _ -> 2. *. x *. ((h (n - 1) x) -. ((float_of_int (n - 1)) *. (h (n - 2) x)));
val h : int -> float = <fun>
```

Combining these two functions we now define a function that computes any desired Gaussian derivative in the \( x \) direction:

```plaintext
# let gdlid_h s ox = fun xi _ ->
let x = (float_of_int xi) in
(-.1.0 /. (s *. (sqrt 2.0))) ** (float_of_int ox) *.
(h ox (-. x /. (s *. (sqrt 2.0)))) *. (glid s x);
val gdlid_h : float -> int -> int -> 'a -> float = <fun>
```
Note that our "univariate" `gdld_h` function has an unexpected arity of two (after partial application to `s` and `ox` values). We define it in this way so that it can be passed directly to the `of_fun` function (defined way back in section 6.5, thus directly discretizing convolution kernels from this function). For univariate derivatives in the `y` direction we use a higher order OCaml function which simply swaps the `x` and `y` parameters:

```ocaml
# let gdld_v s ox = fun x y -> gdld_h s ox y x;
val gdld_v : float -> int -> 'a -> int -> float = <fun>
```

And finally we put it all together into a function that does the separated convolution:

```ocaml
# let do_sep_gauss i s ox oy =
  let sz = 2 * (int_of_float (3. *. s +. 0.5)) + 1 in
  let h = A.of_fun (gdld_h s ox) sz 1 and v = A.of_fun (gdld_v s oy) 1 sz in
  GCMulAdd.op (GCMulAdd.op h i) v;;
val do_sep_gauss :
  (GCMulAdd.dom, GCMulAdd.ext) Pixalgebra.cap ->
  float -> int -> int -> (GCMulAdd.dom, GCMulAdd.ext) Pixalgebra.cap = <fun>
```

This function takes an image parameter, a `float` parameter (the scale `s`), two `int` parameters (the order of differentiation in the `x` and `y` directions), and returns the convolved image.

A common validation technique for convolution-type operations is to test it on an image containing a single white pixel at its center. Convolving such an image will result in an image containing the convolution kernel itself, which can be inspected for accuracy. As a good illustration of the flexibility and expressiveness of our system, a dot function is defined:

```ocaml
# let dot x y = match x, y with 0, 0 -> 1.0 | _, _ -> 0.0;;
val dot : int -> int -> float = <fun>
```

(i.e. a function that takes the value 1 at the origin and 0 everywhere else). Convolving our `dot` with the Gaussians is then simply a matter of:

```ocaml
# let d2x3y = do_sep_gauss (A.of_fun dot 21 21) 3.0 2 3;;
val d2x3y : (GCMulAdd.dom, GCMulAdd.ext) Pixalgebra.cap =
  {width = 21; height = 21; data = <abstr>}
```

Figure 6.12 shows the result of convolving an image with different Gaussian derivatives and the corresponding kernel resulting from convolving our the function with a Gaussian derivative.

### 6.7.2 Complete lattice morphology

Mathematical morphology, like linear scalespace, is also popular in the image processing community. It is a non-linear theory, which can be described in terms of the primitive operations of *erosion* and *dilation*. The standard geometric interpretations of the erosion and dilation operators on binary images do not easily generalize to images
of arbitrary types. Mathematical morphology is most naturally and completely characterized within a complete lattice framework, equipped with the notion of adjunctions.

The complete lattice formulation of erosions and dilations over pixel lattices is taken from Heijmans [44]. Given a nonempty set \( \mathcal{L} \) and a binary relation \( \leq \) on \( \mathcal{L} \), the pair \( (\mathcal{L}, \leq) \) is called a partially ordered set, or poset, if:

- \((O1)\) \( X \leq X \) \hspace{1cm} \text{(reflexivity)}
- \((O2)\) \( X \leq Y \) and \( Y \leq X \implies X = Z \) \hspace{1cm} \text{(anti-symmetry)}
- \((O3)\) \( X \leq Y \) and \( Y \leq Z \implies X = Z \) \hspace{1cm} \text{(transitivity)}

for all \( X, Y, Z \in \mathcal{L} \). Further, a poset \( (\mathcal{L}, \leq) \) is said to be totally ordered if:

- \((O4)\) \( X \leq Y \) or \( Y \leq X \) for every \( X, Y \in \mathcal{L} \)

A totally ordered poset is called a chain. Finally, a poset \( (\mathcal{L}, \leq) \) is a lattice if for any finite subset \( X \) of \( \mathcal{L} \), the supremum and infimum of \( X \) (with respect to the partial order \( \leq \) ) exists. If the supremum and infimum exists for any subset of \( \mathcal{L} \), the poset is a complete lattice. The infimum and supremum of a set are denoted as \( \bigwedge X \) and \( \bigvee X \), respectively.

Given a complete lattice \( \mathcal{T} \) the power lattice \( \mathcal{L} = \mathcal{T}^\mathcal{T} \) is the space of all functions
mapping $\mathcal{T}$ into itself. The partial ordering on $\mathcal{L}$ given by:

$$F \leq G \text{ if } F(x) \leq G(x) \forall x \in \mathcal{L}$$

defines a partial ordering on $\mathcal{L}$ induced by the partial ordering on our original domain $\mathcal{T}$. Further, it can be shown that $\mathcal{L}$ is a complete lattice with infimum and supremum defined as:

$$(\bigwedge_{i \in I} F_i)(x) = \bigwedge_{i \in I} (F_i(x)), \quad x \in \mathcal{T}$$

$$(\bigvee_{i \in I} F_i)(x) = \bigvee_{i \in I} (F_i(x)), \quad x \in \mathcal{T}$$

Note that the order on the functional space $(\mathcal{L}, \leq)$ is determined completely by the order over $(\mathcal{T}, \leq)$. For our purposes $(\mathcal{T}, \leq)$ will correspond to a pixel domain, and the function space $(\mathcal{L}, \leq)$ to images.

One more concept from the complete lattice theory of morphology is needed. Let $\varepsilon$ and $\delta$ be operators on a complete lattice $(\mathcal{L}, \leq)$. The pair $(\varepsilon, \delta)$ is called an adjunction if

$$\delta(Y) \leq X \iff Y \leq \varepsilon(X)$$

holds for all $X, Y \in \mathcal{L}$. It can be shown that if $(\varepsilon, \delta)$ is an adjunction, then they satisfy the following distributivity properties:

$$\varepsilon(\bigwedge_{i \in I} X_i) = \bigwedge_{i \in I} \varepsilon(X_i)$$

$$\delta(\bigvee_{i \in I} X_i) = \bigvee_{i \in I} \delta(X_i)$$

for every family $X_i \in \mathcal{L}$, $i \in I$. When the operators satisfy these properties, $\varepsilon$ is called an erosion and $\delta$ a dilation, and for any $\varepsilon$ that distributes over infima there is a unique $\delta$ for which $(\varepsilon, \delta)$ is an adjunction (and dually for any $\delta$ that distributes over suprema). Lastly, recalling how we lifted the ordering from the pixel domain $(\mathcal{T}, \leq)$ to the functional domain $(\mathcal{L}, \leq)$, it can be shown that $(\delta, \varepsilon)$ is an adjunction on $(\mathcal{L}, \leq)$ if and only if there exists an adjunction $(\varepsilon_{y,x}, \delta_{x,y})$ on the pixel lattice $(\mathcal{T}, \leq)$ such that:

$$(\varepsilon F)(x) = \bigwedge_{y \in \mathcal{T}} \varepsilon_{y,x}(F(y))$$

$$(\delta F)(x) = \bigvee_{y \in \mathcal{T}} \delta_{x,y}(F(y))$$

This pair of low-level operations $(\varepsilon_{y,x}, \delta_{x,y})$ is a pixel lattice adjunction. Note that in the general case, pixel lattice adjunction operators are spatially variant in their definition. All of the operators we will consider are spatially invariant, however.

We now begin translating the complete lattice theory of mathematical morphology into running code. It will be shown how new morphologies of completely different character can be created simply by defining the basic pixel lattice adjunction. All of
the example pixel domains are totally ordered and the min and max operators required
induce a complete lattice structure over them.

The first type needed is for a pixel lattice adjunction:

```
module type PLA =
sig
type dom
type ext
module Be : Bpo with type dom = dom and type ext = ext
module Bd : Bpo with type dom = dom and type ext = ext
val e : int -> int -> dom
val d : int -> int -> dom
end
```

This will, in the end, be the only structure needed to instantiate a working morphology
over image types, as all higher level structures will be parameterized by a pixel lattice
adjunction. Internal to a PLA structure are the binary pixel operations Be and Bd which
will be used to combine the results of a pixel lattice erosion or dilation with the input
image.

Next we need a type to represent an adjunction that has been lifted to the domain
of images:

```
module type Adjunction =
sig
module I : AbstractImage
val erode : (I.dom, I.ext) cap -> int -> int -> (I.dom, I.ext) cap
val dilate : (I.dom, I.ext) cap -> int -> int -> (I.dom, I.ext) cap
end
```

The Adjunction type records the image domain it is defined in, and holds the erode
and dilate functions that have been lifted to operate on images.

The lifting of a PLA and Adjunction is accomplished with the MakeAdjunction type
functor shown in figure 6.13. There are a few items worthy of comment in this figure.
First of all, we are finally using a truly generalized instantiation of the GenConv pattern.
The Adjunction instantiates a generalized convolution using both the min and max
reduce operations from the AbstractImage over which it is defined. The generalized
convolution then uses the Be and Bd binary operations defined in the underlying pixel
lattice adjunction as the Bpo for the respective convolutions. The erode and dilate
functions take two integer parameters, specifying the width and height of the support
for the operation. The operations discretize the structuring functions for the erosion or
dilation using the of_fun function from the underlying pixel domain. Because of this,
all erosions and dilations are constructed from structuring functions centered about
the origin.

The Adjunction type above could be used to perform erosions and dilations, but
we will go one step further and define a Morphology type which encapsulates the
adjunction and provides some higher-level morphological operations. This is done
because it makes no sense to perform an erosion from one adjunction, followed by a
dilation from another adjunction, and call it an opening. The Morphology type is
File: morphology.ml

module MakeAdjunction =
  functor (I : AbstractImage) ->
    functor (P : PLA with type dom = I.dom and type ext = I.ext) ->
    struct
      module I = I
      module RedMin =
        MakeRedOp(I)(struct let op = I.min let neut = I.big_val end)
      module RedMax =
        MakeRedOp(I)(struct let op = I.max let neut = I.small_val end)
      module Erode = MakeGenConv(I)(RedMin)(P.Be)
      module Dilate = MakeGenConv(I)(RedMax)(P.Bd)

      let erode im w h =
        let sf = I.of_fun P.e w h in
        Erode.op im sf

      let dilate im w h =
        let sf = I.of_fun P.d w h in
        Dilate.op im sf
    end

Figure 6.13: An OCaml type functor to lift a pixel lattice adjunction to the image lattice. The adjunction module maintains its own instantiated image operations used to perform the erosion and dilation that make up the adjunction.

defined as:

module type Morphology =
  sig
    type dom
    type ext
    val erode : (dom, ext) cap -> int -> int -> (dom, ext) cap
    val dilate : (dom, ext) cap -> int -> int -> (dom, ext) cap
    val mopen : (dom, ext) cap -> int -> int -> (dom, ext) cap
    val mclose : (dom, ext) cap -> int -> int -> (dom, ext) cap
    val dyr : (dom, ext) cap -> int -> int -> (dom, ext) cap
  end

As with the other image operators defined previously, we copy the type information dom and ext from the underlying domain. In addition to the erode and dilate functions, taken directly from the adjunction giving rise to this morphology, functions for performing openings and closings are provided, and well as a function dyr which performs a morphological gradient operation. Figure 6.14 shows the generic implementation of the MakeMorph type functor which construct a Morphology from an Adjunction. Note how, in addition to ensuring that openings and closings are computed within the same
module MakeMorph = functor (A : Adjunction) ->
  struct
    type dom = A.I.dom
    type ext = A.I.ext
  module BpoSub = MakeBpo(A.I)(struct let op = A.I.sub end)
  let dilate = A.dilate
  let erode = A.erode
  let mopen im v h = dilate (erode im v h) w h
  let mclose im v h = erode (dilate im v h) w h
  let dyrm im v h =
    let e = erode im w h in
    let d = dilate im w h in
    BpoSub.op d e
end

Figure 6.14: The OCaml type functor used to build a morphology structure from an adjunction. The type encapsulates everything needed to compute erosions, dilations, and other higher level morphological operators.

adjunction, the MakeMorph functor also ensures that the size of support is the same for an erosion and dilation used to construct openings or closings.

Figure 6.15 gives the definition for an adjunction consisting of flat, rectangular structuring functions. The adjunction is quite naturally defined as such, with the pixel lattice erosion and dilation simply defined as the constant function taking zero everywhere.

To illustrate how different adjunctions can be defined, giving rise to morphological operators having completely different characteristics, see figure 6.16 for the definition and instantiation of parabolic morphology. Parabolic morphology is the morphological analogue of Gaussian scalepace discussed in section 6.7.1 [112]. The pixel lattice adjunction is defined using the equations of a parabola, with the direction of the parabola reversed for the dilation to satisfy the adjunction property.

6.7.3 An algebraic expression compiler

Using the abilities of the camlp4 OCaml preprocessor to extend the syntax of OCaml it is possible to construct algebraic expression compilers, allowing users to use simple algebraic expressions to implement image processing functions, rather than host language constructs. Algebraic expressions will be evaluated within lazy evaluation contexts, allowing for arbitrary bindings of images and constant pixel values to variable names.

From the definition of lazy evaluation contexts variable names are bound to values. By convention, uppercase identifiers will designate images, and lowercase variables constants. An evaluation context is represented abstractly by the following polymorphic
File: flat.ml

```ocaml
open Pixalgebra
open Scalarfloat
open Morphology

module A = Scalarfloat
module FlatAdj = MakeAdjunction(A)
(
  struct
    module Be = MakeBpo(A)(struct let op = A.add end)
    module Bd = Be
    type dom = A.dom and ext = A.ext
    let e = fun _ -> 0.0
    let d = e
  end
)

module FlatMorph = MakeMorph(FlatAdj)
```

![Images of morphological operators: erosion, dilation, opening](image)

Figure 6.15: Code and examples of flat morphological operators. The adjunction is defined by a function that simply takes the value zero everywhere. The corresponding erosion is the local minimum operator, and the dilation is the local maximum.

lazy list structure:

```ocaml
type 'a stream =
    Nil
  | Cons of 'a Lazy.t * 'a stream Lazy.t
```

and concretely as a stream of type sym:

```ocaml
type sym =
    Image_var of (A.dom, A.ext) cap
  | Const_var of A.dom
```

where the module A is the AbstractImage over which the algebraic expressions will be
File: parabolic.ml

open Pixalgebra
open Scalarfloat
open Morphology

module A = Scalarfloat

module Parabolic =
  struct
    type dom = A.dom
    type ext = A.ext
  module Be = MakeBpo(A)(struct let op = A.add end)
  module Bd = Be
  let e x y = (float_of_int (x * x + y * y)) /. 255.
  let d x y = (float_of_int(-(x * x + y * y))) /. 255.
end

module ParaAdj = MakeAdjunction(A)(Parabolic)
module ParaMorph = MakeMorph(ParaAdj)

(parabolic) erosion  (parabolic) dilation  (parabolic) gradient

Figure 6.16: Parabolic morphology with examples. The pixel lattice adjunction is defined, quite naturally, with parabolas. Integer support coordinates must first be converted to floats before computing the parabola.

defined. A few utility functions will make it easier to access and manipulate bindings from an evaluation context. First, a function to retrieve a binding from a lazy evaluation context:

let rec lassoc l s =
  match l with
  | Cons (hd, tl) ->
    let (n, v) = Lazy.force hd in
    if n = s then Lazy.force v else lassoc (Lazy.force tl) s
  | Nil -> raise Not_found
and a function to concatenate two lazy lists:

```ocaml
let rec lconcat li l2 =
  match li with
  | Cons (hd, tl) -> Cons (hd, lazy (lconcat (Lazy.force tl) l2))
  | Nil -> Lazy.force l2
```

Now for our first actual extension to the core syntax. This extension uses the camlp4 preprocessor to add a syntactical construct allowing for the easy creation of evaluation contexts:

```
EXTEND
  expr :: LEVEL "expr1"
  [[ "Env"; "="; r = LIST1 expr SEP "and"; ""]" ->
    let rec consify l =
      match l with
      | hd :: tl -> <:expr< Cons (lazy $hd$, lazy ($consify tl$)) >>
      | [] -> <:expr< Nil >> in
    let foo = consify r in
    <:expr< $anti:foo$ >> ]];

expr :: LEVEL "expr1"
  [[ x = LIDENT; "="; y = expr ->
    <:expr< ($str:x$, lazy (Const_var $y$)) >>
  | x = UIDENT; "="; y = expr ->
    <:expr< ($str:x$, lazy (Image_var $y$)) >>
  ]];
END
```

This extension adds an Env keyword to OCaml, allowing lazy contexts to be created like so (in the OCaml toplevel):

```
# let env = Env [T = A.read "trui.tif" and C = A.read "cermet.tif)];
val env : (string * Algebraic.sym lazy_t) Algebraic.stream =
  Cons (<lazy>, <lazy>)
```

Note that, due to laziness, no image I/O is done in declaring this evaluation context. By definition, the image files are only read when a symbol is retrieved from the context:

```
# lassoc env "T";;
- : Algebraic.sym =
  Image_var
  {Pixalgebra.width = 256; Pixalgebra.height = 256; Pixalgebra.data = <abstr>}
```

This type of lazy construction will allow us to create bindings between variables and images that (potentially) require a lot of time and effort to compute, while the declaration of the actual evaluation context is instantaneous. Also, once the evaluation of A.read has been forced by accessing a variable in the environment it need not be evaluated again for subsequent references.
We can now define the syntax of our algebraic expressions. In keeping with the other examples in this chapter, we only include a few example algebraic operations in our parser. The grammar for expressions is:

```plaintext
EXTEND
  alge:
  [ "add" LEFTA
    [ x = alge; "+"; y = alge -> infix loc x y <:expr< A.add >>
    | x = alge; "-"; y = alge -> infix loc x y <:expr< A.sub >> ]
  ]
  | "mul" RIGHTA
  [ x = alge; "+"; y = alge -> infix loc x y <:expr< A.mul >>
  | x = alge; "/"; y = alge -> infix loc x y <:expr< A.div >> ]
  | "expr" NONA
  [ x = FLOAT ->
    <:expr< fun env -> let f = my_fos $str:x$ in fun i j -> f >>
  | x = UIDENT ->
    <:expr< fun env -> let f = deref (lassoc env $str:x$) in fun i j -> f i j >>
  | x = LIDENT ->
    <:expr< fun env -> let f = deref (lassoc env $str:x$) in f >>
  | "("; e = alge; ")" -> e ]
];
END
```

The grammar uses a utility infix function that inserts a new infix operator into the grammar being defined. There are a few notable items in this code. First, constant pixel values are implemented, in the finest tradition of algebraic image processing, and constant functions. Thus, constants are not handled as special cases. Also note how each rule returns the abstract syntax tree of a function defined over integer coordinates.

The syntax extension is defined as a camlp4 quotation:

```plaintext
let bind_exp s = Grammar.Entry.parse alge (Stream.of_string s)
let bind_pat s = failwith "not implemented term_pat"
let _ = Quotation.add "alge" (Quotation.ExAst (bind_exp, bind_pat))
let _ = Quotation.default := "alge"
```

and we can now write expressions like so (using the evaluation environment defined above):

```plaintext
# <<< 0.5 * (T + C) >> env;;
- : int -> int -> Algebraic.A.dom = <fun>
#
```

Compare this with the code in figure 6.5 which performs the same image computation using instantiated image operations in the host language.

In addition to allowing for more elegant and compact representations of simple operations like the one above, this type of syntactic extension also allows us to build more meaningful abstractions that are faithful to the algebraic language underlying image
processing theories. Take the Gaussian scalespace example discussed in section 6.7.1. It is most natural for researchers to work with the various Gaussian derivatives using notation like $L_{xxyy}$ and $L_{xzx}$ rather than statements in a programming language [35]. Using lazy evaluation contexts and this simple algebraic expression compiler, we can easily provide this functionality. Given an input image and a scale of differentiation, we can define the Njet [61] of the image as:

```plaintext
let njet im s =
  let ell nx ny =
    let rec repeat s n = if n == 0 then "" else s · (repeat s (n - 1)) in
    "L" · (repeat "x" nx) · (repeat "y" ny)
  in
  let rec order n s im =
    let rec zip x y =
      if x == -1 then Nil else
      let term = lazy (ell x y, lazy (Image_var (do_sep_gauss im s x y))) in
      Cons (term, lazy (zip (x - 1) (y + 1)))
    in
    zip n 0
  in
  let rec senv n im s =
    lconcat (order n s im) (lazy (senv (n + 1) im s)) in
  lconcat (Env [I = im and s = s and s2 = s*s]) (lazy (senv 0 im s))
```

This function returns a lazy evaluation environment, initially containing the original image, the scale, and the square of the scale, capable of generating arbitrary Gaussian derivatives upon demand.

For example, the familiar sharpening procedure done by subtracting the Laplacian of the image from the smoothed original can be computed as:

```plaintext
# let trui = njet (A.read "trui.tif") 1.5;;
val trui : (string * Algebraic.sym lazy_t) Algebraic.stream =
Cons (<lazy>, <lazy>)
# let sharp = <L - 0.5 * s2 * (Lxx + Lyy) >> trui;;
val sharp : int -> int -> Algebraic.A.dom = <fun>
```

Note how the sharp function is now defined over lazy Njet environments, corresponding to the desire in scalespace to define operations defined over all scales. It is also important to note that the semantics of all symbols are supplied entirely by the lazy evaluation context. Our implementation of Njets does not require any modification of the parser for algebraic expressions. Indeed, the parser need not know anything about Gaussian derivatives or how to compute them.
As a final example, we can also define an evaluation context which lazily reads in all images from a directory:

```ocaml
let lazy_dir s =
  let h = Unix.opendir s in
  let rec doit han =
    try
      let f = Unix.readdir han in
      if (Filename.check_suffix f "tif") then
        let name = String.capitalize (Filename.chop_extension f) in
        let hd =
          lazy (name, lazy (Image.var (A.read (s ~ f))))
        in
        Cons (hd, lazy (doit han))
      else doit han
    with End_of_file -> Unix.closdir h; Nil
  in
  doit h
```

As before, all image I/O is lazily evaluated so that images are not read until they are accessed, and once they are loaded they need not be read again. Lazy directories like this can be used to provide working environments to users so that they do not have to worry about image I/O at all. This type of convenience environment is also useful for evaluating functions on a number of test images.

These case studies illustrate the expressiveness of the OCaml image processing system we have developed. The implementation of linear spaces is more or less a direct translation of the functional theory. While exhibiting less affinity with the underlying theory, the complete lattice implementation of mathematical morphology allows erosions and dilations to be completely specified functionally in terms of low-level pixel lattice adjunctions. The algebraic expression compiler illustrated the flexibility of OCaml by allowing core syntactic extensions to the language to be added which are more expressive and appropriate to the image processing domain.

### 6.8 Discussion

In this chapter we have described a functional approach to the design of experimental image processing and computer vision research software systems. By abstracting the pixel domains underlying images we were able to define the set of valid algebraic operations on pixels. Generic recipes for lifting these operations from the pixel domain to images were then defined for a number of patterns common to many image processing operations. By constraining the abstraction of pixel domains and generic patterns to model their analogues in a foreign, native-code image processing library we were able to automatically generate efficient, optimized code to replace the OCaml implementations of low-level image processing functions. Case studies illustrated theoretical abstractions can be modeled in meaningful ways, and that core language extensions can be incorporated in support of this.
We retrospectively revisit the goals established in section 6.3.1:

- **Functionality on demand**
  The ability of our system to provide this can be seen in the examples in figure 6.5 where only the primitive image operations needed for the task at hand are instantiated. This results in smaller executables, and digestible packages of functionality.

- **Relevant and meaningful abstractions**
  Our example implementation of Gaussian scalespace in section 6.7.1 is more or less a direct translation of the equations of the theory, with few extraneous syntactic distractions in the implementation interfering. The example of complete lattice morphology given in section 6.7.2 shows how higher abstractions can be built on top of the provided primitives, but exhibits less affinity with the algebraic equations underlying the theory.

- **Interaction, flexibility, and scalability**
  The example development of Gaussian scalespace in the OCaml toplevel is a good example of the level of interaction supported by our system. This has the effect of minimizing the effort a researcher must expend in committing ideas to runnable code. This approach provides flexibility at the expense of performance, but by preserving the relationship between OCaml and foreign abstractions allows interactively prototyped code to scale effectively.

- **Minimal effort**
  Substitution of optimized native-code implementations of functions is transparent, but must still be done explicitly by the user. There is still a shift in thinking that occurs when switching between an OCaml and native code backend.

Our implementation has been directly influenced by the design and algorithmic patterns of Horus. In many ways Horus is representative of the modern trend toward genericity in image processing and computer vision software design. The push toward genericity ameliorates the problems of code maintenance that plague the previous generation of software environments for experimental image processing. They accomplish this through abstraction of the functional essence of image processing operations into generic patterns. This increase in level of abstraction puts pressure on designers and researchers to cope with it, and tests the limits of efficient compilation. The approach we suggest helps manage this complexity through re-abstraction of the core functionality of generic image processing libraries. By modeling the genericity of foreign libraries we are able to minimize user exposure to the complexity involved with working directly with the underlying patterns.

The object oriented approach to image processing software design, best exemplified by IUE [49], has not been addressed at all. IUE models the fundamental concepts of image processing with classes and class hierarchies. We focused instead on the functional core of image processing. Object oriented design is a valid strategy for creating sustainable software environments in image processing, but objects do not capture the functional essence of image processing operations in ways that are as flexible as the generic approach.
Elegance and expressiveness are subjective qualities, to say the very least. We believe that the OCaml approach offers greater flexibility in implementing image processing theories, and that the expressiveness of OCaml allows for implementations that are some ways more faithful to the original functional, algebraic character of the underlying theories. Functional programming languages have always suffered, however, from the stigma of esoterism, and it is unlikely that the image processing and computer vision research communities at large will be amenable to adopting the functional programming paradigm anytime soon. In informal conversations with the chief software architect of the Horus image processing system, it has been remarked that all we are really doing is trading one type of obfuscation for another. The tools and approaches described in this chapter, however, are also useful for configuring libraries of functionality for end-users. In such a roll as a tool for expert users, the functional approach could be used to great effect in simplifying the tasks of creating, defining, and maintaining high level tools for users.

We have only scratched the surface of what is possible with a functional approach to modeling image processing theories. An important limitation of our current system is that images are “closed” under all of the algebraic operations we have defined. That is, there is no way to define an image operation that takes a floating point image and returns an integer image, or an operation that adds a double image to an integer one. Such operations are important (the inner product of two color images should probably be a scalar), but they also introduce ambiguities that do not harmonize well with the rigid type inference implicit in our OCaml implementation. The answer to the question: *What should the result of convolving an integer image with a floating point kernel be?* is an unqualified *it depends*. Even so, the tools we have described are complete enough to already be useful despite their limitations.

While neither perfect nor complete, we believe that this approach to image processing provides a tantalizing balance between utility and meaningfulness. Functional theories can remain functional all the way from prototyping to implementation, and a non-trivial amount of expressiveness can be preserved between the notebook and the machine. And although symbolic, and *not* numerical, programming has traditionally been the bread-and-butter of functional programming languages, we have shown in this chapter that they can be applied effectively in the domain of image processing.