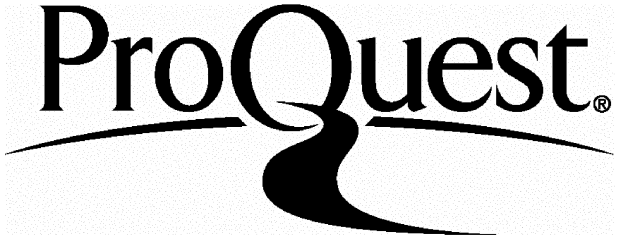
Lazy Image Processing

An Investigation into Applications of
Lazy Functional Languages to Image Processing

Yasuo Kozato

Thesis submitted for the degree of Doctor of Philosophy
University College London
October 1992
Revised, July 1994



Abstract

The suitability of lazy functional languages for image processing applications is investigated by writing several image processing algorithms. The evaluation is done from an application programmer's point of view and the criteria include ease of writing and reading, and efficiency.

Lazy functional languages are claimed to have the advantages that they are easy to write and read, as well as efficient. This is partly because these languages have mechanisms to improve modularity, such as higher-order functions. Also, they have the feature that no subexpression is evaluated until its value is required. Hence, unnecessary operations are automatically eliminated, and therefore programs can be executed efficiently. In image processing the amount of data handled is generally so large that much programming effort is typically spent in tasks such as managing memory and routine sequencing operations in order to improve efficiency. Therefore, lazy functional languages should be a good tool to write image processing applications. However, little practical or experimental evidence on this subject has been reported, since image processing has mostly been written in imperative languages.

The discussion starts from the implementation of simple algorithms such as pointwise and local operations. It is shown that a large number of algorithms can be composed from a small number of higher-order functions. Then geometric transformations are implemented, for which lazy functional languages are considered to be particularly suitable. As for representations of images, lists and hierarchical data structures including binary trees and quadtrees are implemented. Through the discussion, it is demonstrated that the laziness of the languages improves modularity and efficiency. In particular, no pixel calculation is involved unless the user explicitly requests pixels, and consecutive transformations are straightforward and involve no quantisation errors.

The other items discussed include: a method to combine pixel images and images expressed as continuous functions. Some benchmarks are also presented.

Table of Contents

List of Figures

List of Tables

Acknowledgements

This work was initiated through Canon's international technical traineeship scheme, which allowed me to spend two years from 1986 as a research student at University College London. After the setting-up of Canon Research Europe in Guildford, our management allowed and encouraged me to continue my research. Without their generosity my work would never have completed. Therefore, I would like to thank all the people at Canon, in particular Dr. H. Mitarai and Mr. H. Negishi for their continuous support.

A number of people have aided my work in many ways: First and foremost, I am indebted to my supervisor at UCL and now colleague at CRE, Paul Otto, for his patience, continuous encouragement and numerous relevant technical discussions and suggestions throughout; people who have a common research interest in the interdisciplinary area of functional programming and image processing: Ian Poole, for the excellent discussions and continuous communication; Michael Parsons, Thomas Breuel, Greg Michaelson and Mark Allsop for sending their theses, papers and relevant information; people from the Glasgow functional programming group, who, at the workshop in Ayr, broadened my knowledge of the state-of-the-art and future of lazy functional languages; Lennart Augustsson and Colin Runciman for providing tools and very useful suggestions for using them.

Finally, I would like thank Fukumi, my partner in my personal and professional life, who submitted her thesis 12 days before me. Both of us can now feel an intense sense of relief and drop those highly technical conversations over our meals.

Acknowledgement to the Revision

More than one and a half years have passed since the first submission of the thesis. During the revision work various things happened:

- The paper based on Chapter 8 of the first submitted version, "Benchmarking Real-Life Image Processing Programs in Lazy Functional Languages", was accepted and presented at the Conference on Functional Programming and Computer Architecture in June 1993. So I have attached the paper as an appendix and made major revisions in Chapter 8.

- A new Haskell compiler (GHC) from University of Glasgow became available, which is said to be most efficient at present. So I used GHC for the benchmarks in revised Chapter 8.

- My viva was held on 5th January 1993. The comments by the examiners were considered and reflected in the revision. I believe the thesis became much more polished and I would like to thank the examiners for their useful suggestions.

- I returned from U.K. to Japan, and my role changed from a researcher to a manager. These changes made the revision work much slower than I thought. But I am pleased that I have done the job.

Lastly, I would like to thank Paul Otto again who has encouraged myself and made useful suggestions and discussions all through the revision work, across 6000 miles!

Chapter 1: Introduction

1.1 Image Processing

1.1.1 What is image processing

Image processing [Gonzalez87a, Schalkoff89a, Pratt91a] is a large application area which includes literally everything which involves processing images. Although it is a diversified area, its common property is that it involves enormous amount of data. In order to deal with images on computers, we normally use *pixels*, measurements such as intensity at sampled points of an image, and an image is typically represented as an array of pixels. The number of pixels involved is varied but always large. For example, a television picture usually contains approximately 500×500 pixels, a high-definition TV image or a bit-map image on a current workstation roughly 1000×1000, an image for printing an A4 document about 3000×4000, and a panchromatic SPOT satellite image [Chevrel81a] 6000×6000. A pixel may contain one-bit (ON/OFF) information, a few bits or a real number to denote a value such as a gray level, a complex number for Fourier transformed images, a tuple such as RGB data for multi-spectral images, a label, or other types of information for various purposes. Images may be static, as well as dynamic; such as television pictures, or processing image sequences. In addition, images are not only 2D but also 3D or more; *voxels* sampled at 3D grid are used for representing 3D structures, e.g. MRI data [Woods91a] of human bodies.

As well as the sizes, the purpose of image processing varies extensively. It may be as simple as adjusting intensity of an image to improve the look of the image, or geometric transforms such as rotation and scaling. It may be transformation from one domain to the other domain, such as a Fourier transform which transforms an image from the spatial domain to the frequency domain; Hough transform to convert an image from the spatial domain to the Hough space; image compression such as JPEG [Wallace91a] and MPEG [Le Gall91a] for storing images or communicating through networks. Or, it may be for extracting necessary information out of an image such as description of edges, shapes, or labels. It may be a conversion from some description to reconstruct an image such as data obtained from computer tomography (CT) [Herman80a], MRI, or synthetic-aperture radar (SAR), or producing a picture from a picture description language such as PostScript [Adobe Systems Inc.90a]. In any case, image processing can be described as a certain process applied to input images or descriptions which yields output images or descriptions[1].

1.1.2 Programming image processing

Because a large amount of data is involved, programming image processing is rather tedious and cumbersome. We address the difficulties in programming image processing as follows:

- ***Programmers have to put extra effort in arranging sequences of operations, so that the amount of memory used is kept within reasonable bounds.***

  For example, even if the same operation is being applied, different code may be necessary for processing a television image and a satellite image, since the satellite image contains a lot more data than the television image, and so it may not be practical to have all the data in main memory. In such a case, the satellite image may be read, processed and written in a line by line or chunk by chunk manner, while the television image may be read and processed at once.

  As another example, in order to save memory, programmers normally overwrite an image which will not be used. If an operation is pointwise (See Section 2.2.2) it is OK to replace the input pixels with the output as the operation proceeds, but if it is a local neighbourhood operation (See Section 3.5), overwriting the input image does not yield the correct result because the resulting pixel value depends on the values of surrounding pixels. In such a case, either the output image should be kept separate, or careful arrangement of memory usage in order *not* to give the wrong result is necessary.

  In any case, programmers have to be very careful in arranging proper usage (and reusage) of memory and appropriate sequences of operations, which is not an easy task.

- ***Programmers have to write almost identical code for various operations on various images although there are a lot of common structures among them.***

  As described in the previous subsection, there are many different image types; binary, gray, colour, label, complex, or other kinds of images. But these images usually share a common structure, i.e. a 2D array of pixels, and the difference lies in the type of an element in an array. If a programming language does not have a facility to unify those image types, programmers will have to define those images separately and write almost identical code to process those images. For example, when a gray image and a colour image of the same

1. This definition is rather broader and more vague than the common definition, e.g. [Horn86a], where image processing is defined as image to image conversion. However, as will be shown, use of lazy functional languages allows unified treatment of these wider areas.

size are being defined, they may need separate definitions because the type of an element is a single value for the gray image, whereas it is a triple of RGB values for the colour image.

Not only the structures of image data, but also the structures of computation are shared among a number of image operations, such as pointwise operations, local neighbourhood operations, and so on. These computational structures are typically expressed by control structures in conventional languages, such as a double loop. If a programming language does not support passing various operations to a common control structure, programmers have to write almost identical code many times. For example, if they write programs to add two images and to subtract two images in a pixel by pixel manner, they normally end up with two separate but almost identical pieces of code in which only plus (+) and minus (-) symbols differ.

1.1.3 How do conventional languages cope?

This subsection reviews conventional languages and how they cope with the problems addressed above. No language can be a panacea, but certain facilities will work well to solve certain problems.

In *Fortran*[2], where only static arrays are used, arrays must be explicitly declared in source code, i.e., the size of images must be known at compile time. Also, it is very difficult to cope with changes of array size, and so image processing libraries written in Fortran tend to keep the size of images fixed through operations.

In *Pascal* and C, where memory can be dynamically allocated and deallocated via function calls, the size of images need not be known at compile time, but explicit allocation and deallocation must be described in source code.

C++ [Stroustrup86a] features *classes* which allow encapsulation of a number of data types and operations associated with them in one class, and the implementation details can be hidden from users. Thus, it is possible to hide explicit memory allocation/deallocation of complicated data structures from users, though it must be implemented somewhere by implementors. Also, *overloading* of functions and operators allows registration of different operations on different data types under the same name. So, for example, once various data types and operations have been packaged up as the class 'pixel', users do not have to be aware

2. up to Fortran 77, anyway.

of which types of pixels are being processed. This saves users from writing almost identical programs. However, these techniques are quite *ad-hoc* [Strachey67a] in the sense that the underlying operations work differently on different types despite that they bear the same name. If a new data type is to be added to a class, implementers must look into the implementation details.

A new feature of C++ called *templates* [Stroustrup91a] may be another solution; these allow various types and functions to be used as parameters in different instances of a common pattern of data, such as lists and arrays. But this is a kind of macro facility and its flexibility and applicability are rather limited.

Lisp automates memory management, so that normally users do not have to write code to allocate/deallocate storage. Also, it is an untyped language, so that a function argument can take various data types. However, it is not statically type-checked, which means that programmers have to work hard to produce correct programs, especially when complicated data structures are being used. Some dialects of Lisp, such as *Scheme* [Abelson85a] and *Common Lisp* [Steele Jr.90a], allow a functional style of programming in which a function can take other functions as its arguments. This is a convenient feature to save programmers from writing almost identical code, since various operations can be passed as arguments to the common control structure.

Table 1-1 Comparison between conventional languages

	Fortran	C, Pascal	C++	Common Lisp	Scheme
Automatic storage management			1	✔	✔
Static type checking	✔	✔	✔		
Classes and Overloading			✔, 2	3	
Functional-style programming			2	✔	✔

1. Memory management can be hidden from users of a library.
2. Also templates [Stroustrup91a] allow a limited form of polymorphism.
3. *CLOS* [Bobrow90a] is an object-oriented extension of Common Lisp which has classes.

Table 1-1 shows differences between conventional languages described above. Please

note that the table is based on the "standard" programming style. So, it is possible to program in a different style, though it could be unusual or inefficient. Also, languages are evolving, and various extensions are available to support particular styles of writing.

1.2 Functional Programming

This section gives a brief overview of *functional programming*. For more details, see various textbooks and surveys, e.g., [Henderson80a, Bird88a, Reade89a, Hudak89a].

1.2.1 Fundamental concepts

Functional programming falls in the category of *declarative programming*. The most important characteristic of declarative programming is that there are no implicit states, or no *side-effects*. This is because assignment is prohibited in declarative programming, which contrasts with *imperative programming*. Imperative programming has implicit states which may be modified or side-effected through assignment operations. Imperative languages include most conventional languages such as Fortran, Pascal, C, C++, Lisp, and so forth.

The basic computational model of functional programming is mathematical functions in contrast to *relations* in *logic programming* [Sterling86a] which is also categorised as declarative programming. Programs are constructed as a collection of function definitions and expressions are evaluated by the computer. The basic mechanism of evaluation is function application, i.e.,

output = function (inputs)

is the fundamental style of functional programming where *output* names the result of applying *function* to the argument *inputs* and is completely equivalent in its meaning. In other words, *referential transparency* is secured in functional programming, provided all functions are deterministic.

Because there are no side-effects, the output depends only on the input arguments, and how the evaluation is carried out is the system's role. Amongst various evaluation strategies, *eager* and *lazy* are the two which should be mentioned here. *Eager evaluation*, or *applicative order evaluation*, takes the strategy that a function is evaluated after all the input arguments have been evaluated. Whereas *lazy evaluation*, or *normal order evaluation*, evaluates an expression in a *demand-driven* manner, i.e., no argument is evaluated until its value is required. Lazy

evaluation is commonly used to implement languages with non-strict semantics (see Section 4.2).

One of the important features of functional languages is that a function is regarded as a first-class object, just like a number in conventional languages. A function which takes a function as argument, or delivers one as a result is called a *higher-order function*. The benefits of a higher-order function are that it can separate control structures from operations, and that arbitrarily complicated control structures can be relatively easily constructed from simple ones. Scheme and Common Lisp, as described in Section 1.1.2, have this facility although they have imperative features, as well.

Types are an important factor in programming languages to help producing reliable code. It is often desirable that program errors are detected as early as possible, and programs should not run if they contain type errors. Languages with this facility are said to be *strongly typed* and such a type checking mechanism is called *static type checking*. Most functional languages, as well as many imperative languages including Fortran, C, Pascal and C++, have this feature, but Common Lisp and Scheme do not, as these are *untyped*, or *dynamically typed languages*.

In addition, most functional languages allow *polymorphic functions*; that is, a function may take arguments of arbitrary type if the function does not depend on that type. For example, suppose there is an identity function which returns the same entity as its input argument. If the identity function takes a number it returns the number, and if the input argument is a string it returns the string. In this example, as long as the types of its input and output are the same, it can be of any type. Such a type is called a *polymorphic type* and a type system which allows polymorphic types is called a *polymorphic type system*.

1.2.2 A brief history

The most influential work which formed the theoretical basis of functional programming would be the *lambda calculus* invented by Church [Church41a] (see also [Barendregt84a]). As for the programming language side, the development of Lisp by McCarthy [McCarthy60a] was a big step forward. Lisp has been so popular in the artificial intelligence community in the U.S.A. that it has produced a lot of dialects and extensions. The most important developments amongst them were Scheme [Abelson85a] and Common Lisp [Steele Jr.90a] because they feature functions as first-class objects.

Other important work was done by Landin who proposed a language called *Iswim* [Landin66a]. He put a lot of syntactic and semantic ideas into the language, such as infix notations, introduction of *let* and *where*, and emphasised the importance of defining a syntactically rich language over a small but expressive core language.

The importance of Backus' Turing Award Lecture in 1977 [Backus78a] was that it drew much attention to functional programming, although *FP* itself has been a less common language.

The polymorphic type system, one of the principal features of modern functional languages, was invented by Hindley [Hindley69a] and Milner [Milner78a], hence called *Hindley-Milner type system*, and was first implemented in a programming language called *ML* [Gordon78a], later developed in *SML* [Milner84a]. SML has been one of the most widely-used functional languages.

A series of language designs carried out by Turner, started as *SASL* [Turner76a] then *KRC* [Turner82a], has led to the development of *Miranda*[3] [Turner85a] which we will use extensively in this thesis. The most notable feature of Miranda is lazy evaluation, as well as more common facilities such as polymorphic strong typing and higher-order functions. Currently, Miranda is the only functional language for which there is a commercially available implementation.

A more recent movement has been the development of *Haskell* [Hudak92a] by an international group of researchers. The aim of the project is to design a common language to avoid emergence of a number of similar functional languages and duplication of effort. Haskell is a general purpose, purely functional language with the features of higher-order functions, non-strict semantics, static polymorphic typing, and so forth. Besides these features which are in common with Miranda, it also incorporates novel features such as type classes and overloading integrated with the polymorphic type system and non-strict immutable arrays. As of now, a few implementations have appeared [The Yale Haskell Group91a, Augustsson92a, The AQUA Team93a]. Also, a subset of Haskell has been implemented by Jones as *Gofer* [Jones92a]. A good introduction to Haskell is given in [Hudak92b].

Another recent development worth mentioning is a language called *SISAL* [Böhm92a]. This language is in the category of *dataflow languages* designed for dataflow architectures

3. Miranda is a trademark of Research Software Ltd.

[Chambers84a]. On such machines operations are completely data dependent, and therefore sequencing of operations by a human programmer is not relevant. Hence, such languages are essentially functional. Other languages in this category include *Id* [Arvind89a]. The importance of SISAL is that its implementations on parallel machines are very fast. A recent report shows that it runs as fast as Fortran using various scientific applications including Fourier transforms and hydro-dynamics [Cann92a]. This result suggests that, although SISAL is not lazy, and is without polymorphic typing, functional languages could be implemented efficiently on parallel machines. Since image processing is a data intensive application area, this possibility may be a good sign for functional languages to be used for real image processing applications.

To conclude this subsection, Table 1-2 gives a comparison between principal functional languages in terms of their facilities. Of the languages reviewed so far, we are mainly using the lazy functional language Miranda. Section 1.2.4 describes the reasons for the selection and Section 1.3 describes the possible benefits of using lazy functional languages for image processing.

Table 1-2 Comparison of principal functional languages

	Miranda	Haskell	SML	SISAL	Scheme
Lazy evaluation	✔	✔	1, 2		2
Higher-order functions	✔	✔	✔	✔	✔
Polymorphic strong typing	✔	✔	✔		
Type classes and Overloading		✔			
Arrays		✔	3	✔	✔
Imperative features			✔		✔

1. *Lazy ML* [Augustsson84a] supports lazy evaluation.
2. It is possible, though limited (See [Paulson91a] for SML and [Abelson85a] for Scheme).
3. Some implementations have arrays [Paulson91a].

1.2.3 Implementations

Implementations of functional languages have often been described in terms of abstract machines, since programs are analysed and compiled into a sequence of instructions which an abstract machine can execute. Whether the machine is implemented in hardware or software appears to be a separate issue. However, it is generally considered that functional languages have potential to be implemented efficiently on parallel architectures since there are no side-effects and expressions can be evaluated in any order. In fact, the recent report on SISAL by Cann [Cann92a] can be regarded as a strong evidence to support this statement. Thus, although implementations of functional languages are still a research area, it should not be long before efficient parallel implementations appear.

The *SECD machine* invented by Landin [Landin64a] and later described by Henderson [Henderson80a] is one of the early-days implementations. It is a stack-based implementation and consists of four stacks, namely S: the *stack*, E: the *environment*, C: the *control list*, and D: the *dump*, hence the name. The SECD machine can be extended to support lazy evaluation.

Turner proposed a technique called *combinator*[4] *reduction* [Turner79a] by which high-level expressions can be compiled into expressions in a fixed set of low-level combinators named *SK combinators*. Hughes modified this method and developed *super-combinator compilers* [Hughes84b]. He showed that as the size of a program becomes larger the super-combinator method presents better performance than Turner's method. Both methods implement lazy evaluation and have formed the basis of modern implementations of lazy functional languages.

More recent implementations include *TIM (The Three Instruction Machine)* [Fairbairn87a], *the G-machine* [Augustsson87a, Johnsson87a], a parallel version of the G-machine [Kingdon91a], and so forth.

1.2.4 Miranda

Miranda has been chosen as a representative of lazy functional languages and the code that appears in this thesis is written mostly in Miranda. The reasons are:

- Miranda is a lazy functional language which has all the principal features that modern

4. A combinator is a lambda expression which contains no occurrences of a free variable [Barendregt84a].

functional languages have, such as higher-order functions and polymorphic strong typing.

- Miranda is small and easy-to-learn, and is capable enough to demonstrate the claims described below.
- A stable Miranda implementation was available when coding was being done.[5]
- Miranda works and integrates well on a UNIX workstation and its supporting environment such as on-line manuals is good.

For an introduction to programming in Miranda, see, e.g., [Turner86a] and [Bird88a]. In this thesis it is assumed that readers have basic knowledge of Miranda programming. Also, Miranda's built-in operators and functions defined in the *standard environment* are used without definition or explanation. If necessary, readers should refer to the Miranda's on-line manual [Research Software Limited89a].

1.3 Why Functional Programming Matters to Image Processing

Image processing programming in functional languages has been a less common approach until recently, partly because functional languages themselves are still under active research [Hudak89a] and not many real-world applications have been written [Hall92a]. Nevertheless, a few research results have shown that functional programming should be a good vehicle to describe image processing algorithms [Allsop91a, Breuel92a].

The overall theme of this thesis is *"are lazy functional languages 'good' for writing image processing programs?"*. The suitability of lazy functional languages for image processing is investigated by writing several image processing algorithms in lazy functional languages. The focus of the discussion is on how the features of these languages, laziness in particular, make image processing programming easier, or more difficult. Possible criteria are ease of writing and reading, and efficiency.

A particular issue related to ease of writing and reading is *modularity* of programs which describes the capability to keep code for separate operations separate and to require little work in putting them together. If programs can be written in a modular manner it will also improve

5. The version of Miranda used in this thesis is V2.014.

reusability, because different programs may comprise common operations or control structures.

It is assumed that time and space efficiency of the underlying implementation of lazy functional languages will be adequate for image processing applications. We are going to discuss efficiency from an application programmer's point of view, such as asymptotic behaviour, termination and the possibility of using better algorithms[6].

The following is a list of possible benefits of using lazy functional languages in image processing programming, by which the difficulties of image processing programming discussed in Section 1.1.2 may be overcome. These items will be discussed in detail with examples throughout the body of the thesis and revisited in the last chapter to discuss to what extent these benefits have been confirmed. The first four items do not concern laziness, but can be applied to any functional programming languages with higher-order functions and polymorphic typing:

1. The basic functional style, i.e. *output = function(inputs),* can naturally express image processing which is essentially described as an "input→process→output" process.

2. Higher-order functions allow different functions to be mapped onto a common structure, which enables separation of control and operations. This facility should improve modularity and reusability of code because images typically have a common structure, such as a 2D array of pixels.

3. Polymorphic typing allows *any* pixel types in a common image structure, which should provide a mechanism to handle various types of images, such as binary, gray, colour, or other kinds of images, in a unified manner.

4. An image may typically be represented as a 2D array of pixels, but it can also be viewed as a collection of rows and a row being a collection of pixels, i.e., an image and a row have the common structure. Using this nature, higher-order functions together with polymorphic typing should provide a simple mechanism to build up higher dimensional data and operations by combining lower dimensional ones which are usually easier to implement and debug.

---

6. However, the actual speed and memory usage using currently available languages are of interest. Chapter 8 gives some benchmarks.

And the following are three possible benefits which originate particularly from laziness:

5. Lazy evaluation ensures that an expression is not evaluated until its value is required. In image processing where each operation is usually very expensive, this facility to eliminate unnecessary operations may improve efficiency automatically. It may be more beneficial when not the whole image but only a small portion is required.

6. Functional languages embed memory management and routine sequencing operations in themselves. This, when combined with lazy evaluation, may become more beneficial in image processing. For instance, when consecutive operations are applied to an image, the execution sequence and the necessary intermediate memory can all be handled by the functional language implementation, so that programming will be easier.

7. One particular consequence of the embedded memory management and sequencing operations is that it may improve modularity, since different parts can be implemented as different functions and not much care is necessary in putting them together. For example, it may be possible to make a display function separate from the other image processing parts which can be used as a common programming part for display.

Of course there are drawbacks of using lazy functional languages. As a natural consequence of the fact that the languages do more work in order to be lazy[7], i.e., it is necessary to decide whether an operation is required or not, bigger overhead may be incurred and execution of programs may be rather slow compared with eagerly evaluated languages. This, however, should trade off with the above benefits, such as ease of programming and better efficiency when requiring a small portion. Also, as implementation techniques of lazy functional languages are advancing, programs will run faster when more efficient implementations appear.

1.4 Related Work

Application of functional languages to image processing is a little-researched area and only a few results have been reported. In this area, SML has been the most common language. SML features higher-order functions and polymorphic strong typing, but is not lazy and has imperative features such as references, imperative arrays, and referentially opaque I/O. So, it

7. For the overheads of laziness, see page 64.

is possible to program in an imperative style if a programmer chooses to do so.

Allsop [Allsop91a] implemented a basic image processing library called *FLIPT* (Functional Language Image Processing Toolkit) using SML and discussed basic concepts of image processing by comparing SML programs with Apply [Hamey89a] and C. Breuel [Breuel92a] implemented a library of functions for computer vision both in SML and C++ and discussed how elements of functional programming benefit the programming of various vision algorithms. They both concluded that functional programs are modular and easier to understand because the language has strong mechanisms, such as higher-order functions and polymorphism, to integrate from diverse sources. Breuel, in particular, stated that the language support for closures with unlimited extent enables one to implement lazy data structures which are particularly desirable when an algorithm does not require information at every pixel. However, SML is a strict language and so any laziness needs to be explicitly programmed (See Table 1-2). As a consequence, he did not demonstrate this in the programs. The overall programming style of SML, though it is functional, is quite different from lazy languages, and so separate work using lazy languages would be necessary to discuss effects of laziness.

Wallace *et al.* [Hopkins89a, Wallace92a] implemented the specific algorithms of scene labelling and parallel perspective inversion using SML. Although they mentioned the general advantages of functional programming such as modularity and ease of understanding, their emphasis is rather on how to identify and utilise the implicit parallelism of functional programs and how they can be efficiently executed on parallel machines. There is no mention of laziness.

Another common functional language is Miranda which we also use intensely in this thesis. Parsons [Parsons87a, Parsons89a] used Miranda to implement various data structures to represent graphical data and images, including lists of line segment, rasters, quadtrees, quadgraphs[8] [Parsons86a], run length code, and fractals. His focus is on the use of algebraic data types with *laws* [Thompson86a]. However, laws are difficult to use and have become an obsolete feature in the current Miranda release [Research Software Limited89a]. As for laziness, he concluded that it contributes to improving efficiency by non-strict transformations, displaying primitives at lower resolution, minimising relational tests, and in

8. A quadgraph is a data structure similar to a pointer-based quadtree, but it allows pointing to any node. So, it could be a cyclic or self-recursive data structure.

the higher-order version of *RasterOp* operations. His most important message is "we have been enlightened by the use of this language; it has suggested new and interesting definitions of old graphical ideas [Parsons87a]." However, he did not discuss how laziness affects not only the data structures, but also the way programs are written, in particular how it improves modularity of programs. Although he implemented interesting data structures, his emphasis was on graphics applications and he did not present commonly used image processing operations in a way that utilises laziness in a positive manner.

The other development using Miranda is by Poole [Poole92a]. His approach is fairly pragmatic. The execution speed of Miranda programs with the current implementation is rather slow for *real* image processing applications, so he uses Miranda as the front-end to the existing *Woolz* image processing server [Piper85a] written in C. Such systems may provide a good interim solution by combining the speed of a conventionally implemented image processing system with the flexibility and other advantages of a lazy functional language. He is currently working on a fully-functional version using Haskell, as efficient and reliable implementations are appearing.

Burton and Huntbach's work [Burton88a] does not use a functional language, but focuses on laziness in manipulating geometric objects. In particular, they advocated that lazy evaluation of geometric objects is useful whenever objects are represented by quadtrees and only a portion of the data structure is required. They implemented a lazy data structure of a quadtree in Pascal, in which an extra boolean member indicates whether the quadtree has been evaluated or not.

1.5 The Rest of the Thesis

In the remainder of the thesis, we will discuss whether lazy functional programming languages are suitable for writing image processing programs using a number of examples.

Chapter 2 serves as an introduction to image processing in Miranda to familiarise readers with this paradigm using very simple examples, such as unary and binary pointwise operations. Also an example Miranda session is provided in order for readers to understand how to communicate with the system. We call the version introduced in this chapter the *"naive"* version, because this is the image processing program which a first-time Miranda programmer would write.

In Chapter 3, an implementation called the *"with an origin"* version is presented. The categories of algorithms included are pointwise, image translation, and local neighbourhood operations such as convolution. This version introduces the concept of an origin into the data structure to represent an image, by which the image boundary problem commonly associated with image translation and local neighbourhood operations is solved elegantly. Also, a fairly large number of image processing algorithms based on these operations are implemented. It is remarkable that median filtering which is not usually considered to be convolution can be implemented using the function defined for convolution. The discussed items include: a polymorphic pixel type allows unified treatment of various categories of images; higher-order functions and polymorphic typing allow construction of 2D operations by combining 1D operations in a straightforward manner, and improve modularity.

Chapter 4 focuses on laziness in image processing. In the first part, laziness in the context of functional programming languages is described, where lazy evaluation is an implementation technique to allow non-strict semantics. Then, discussion moves on to the relationship between laziness and image processing. We discuss how laziness is a convenient feature to improve efficiency and how it improves modularity. The last part discusses the algorithms which are considered to be particularly suitable if expressed in lazy languages. We call these algorithms *inherently lazy* and discuss whether lazy languages can express these algorithms in a natural fashion.

Chapter 5 gives an implementation of inherently lazy geometric transformations using lists as the basic structure for images. Programs are written to utilise benefits of laziness of the language which include: (a) unless the user specifically asks an image to be displayed, no pixel operations are carried out, (b) and only the required parts are calculated, (c) program description is forward mapping in a straightforward manner, and (d) consecutive transformations are expressed as straightforward function composition and do not accumulate quantisation errors. It will be shown that pursuing these benefits of laziness leads to a novel way of image processing programming. Limitations and drawbacks of the algorithms are also discussed.

Chapter 6 presents the same algorithms as Chapter 5, but another implementation using hierarchical data structures. This chapter is divided into two parts: representing an image as a binary tree of binary trees, and as a quadtree. Using a tree of trees is an extension to the method which has been developed through the previous chapters, i.e. define 1D operations first, then

compose 2D operations using higher-order functions. We will demonstrate that this principle works for hierarchical data structures, as well, and that code is fairly easy to read and write. A quadtree is our first attempt to implement 2D data structures directly, since a simple calculation shows that a quadtree is supposed to have space benefit compared with a binary tree of binary trees. It will be shown that although the quadtree version seems less readable, it gives an easy mechanism of random access, i.e. look-up a value of an image at an arbitrary position.

Chapter 7 provides an implementation and some examples of combining pixel images and non-pixel images together, since this is another algorithm which lazy functional languages may be particularly good at. In the first section, issues related to image I/O are discussed because for the pixel image side this is a non-avoidable technique to be implemented. Then, a method to combine pixel and non-pixel images is described. Images are implemented as abstract data types which hide implementation details from application programmers and allow much higher level programming. Several examples are presented.

Chapter 8 gives some benchmarks comparing median filter operations written in Miranda, Haskell and C. The comparison may not be fair, but it still gives some idea of how fast/slow lazy functional languages run in real life using currently available implementations. We show that lazy functional versions run in nearly constant space even if the size of an image gets very large, which is not possible to achieve in C without extra programming effort.

Chapter 9 is the last chapter. It concludes the thesis by further discussions, summary and future developments. We will review how many of the benefits we listed earlier have been confirmed.

Chapter 2: Simple Image Processing in Miranda

2.1 Introduction

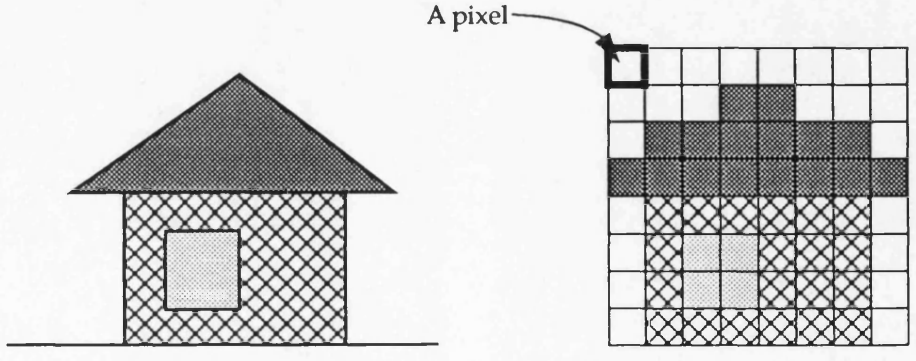
Functional programming is a relatively new paradigm and is considerably different in style from imperative programming in which image processing programs are usually written. Therefore, it is important to get familiar with the functional programming style first. This chapter is intended in this respect to give some simple image processing algorithms written in Miranda, which include pointwise operations both unary and binary.

In implementing these algorithms, we will try to be as naive as possible. Thus, we will call the program described in this chapter *the naive version*. In addition, an example Miranda session is presented because working procedures, i.e. from editing to running, are very different from ordinary compiled imperative languages such as Fortran and C.

2.2 The Naive Version

2.2.1 Raster images

An image is typically represented as a 2D and rectangular array of pixels, or *raster* (Figure 2-1). Although conceptually an image may not be rectangular or may be an abstract description of a scene, raster representation is most common because it is the easiest form for images to be manipulated on computers and I/O devices, such as frame buffers, scanners, and laser printers.



(a) A scene (b) Its raster image

Figure 2-1 A scene and its raster image

For representing an image using imperative languages, an array structure is typically used. In order to represent a 2D array of pixels in Miranda, use of a list of lists of pixels would be most straightforward, since arrays are not supported in Miranda. Also, the properties of a list describe the nature of a pixel image quite well, that is, it is an ordered collection of values and the elements of a list must have the same type. In Miranda, a 2D list is written as follows, assuming that pixels are numbers:

```
pixel == num
img   == [[pixel]]
```

'==' is a *type synonym* which allows users to introduce a new name for an already existing type. `[pixel]` defines a list whose elements are of the type `pixel`, and `[[pixel]]` defines a list of lists of pixels.

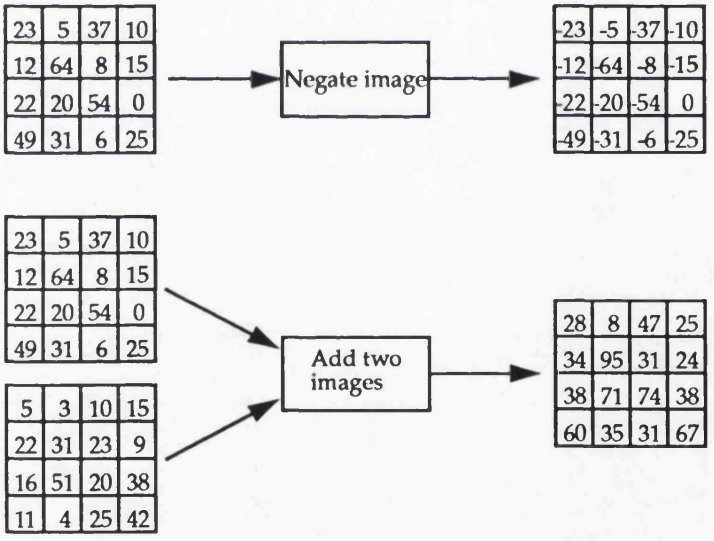


Figure 2-2 Pointwise operations

2.2.2 Pointwise operations

The term *pointwise operation* describes an operation which takes one or more images as its input and carries out a certain operation over the images in a pixel by pixel manner. The actual pixel operation may be as simple as negation or addition, or it may involve more complicated operations. The principles of the operation are: the result does not depend on any pixels of different position, and the same operation is carried out all through the images. In other

words, the operation only concerns the pixel itself in the case of an *unary pointwise operation*, or the pixel itself and the pixel of the same position in the other image in the case of a *binary pointwise operation*, but the same operation is carried out regardless of the position (Figure 2-2). Therefore, the pointwise operation provides a framework to define various image processing algorithms, rather than a specific algorithm.

2.2.3 Unary operations

We use the function `map` to implement unary pointwise operations. `map` is defined in the standard environment, and has the type:

```
map::(*->**)->[*]->[**]
```

Types can be parameterised using *type variables* such as * and ** in Miranda. A type variable can accept *any* type which is called a polymorphic type. Code to define the type is called a *type signature*. The above type signature can be read as *"for all types * and **, the function* `map` *takes two arguments, a function from * into ** and a list of elements of type *, and returns a list of elements of type **"*. Since the function takes a function as an argument, it is a higher-order function. What `map` does is to apply the function given as its first argument to each element of the list given as its second argument. So, for example, the increment `(+1)` and the logical negation `(~)` functions can be mapped on a list of numbers and booleans, respectively[1]:

`map (+1) [1,2,3]` reduces to `[2,3,4]`

`map (~) [True,False,False,True]` reduces to `[False,True,True,False]`

If the first argument is a function defined on a list, such as `(map (+1))`, then `map` makes a function on a 2D list. For example:

`map (map (+1)) [[1,2,3],[4,5,6]]` reduces to `[[2,3,4],[5,6,7]]`

This is what a unary pointwise operation should do. Thus, a unary pointwise operation can be defined as follows[2]:

```
unaryPointwise::(pixel->pixel)->image->image
unaryPointwise f im = map (map f) im
```

---

1. In Miranda, both operators and functions can be used interchangeably using *sections* [Bird88a].
2. The other style of expression will be discussed in Chapter 3.

It is not necessary to attach a type signature to every function definition, and even if a type signature is not given, Miranda's type checker infers types correctly if they are well-defined. In fact, it is often the case that the compiler infers more general types than those specified by programmers.

Miranda uses a technique known as *currying* [Bird88a] which assumes function application to be left associative. This is useful to reduce the number of brackets which has been a cause of unreadability of Lisp code, and to allow elimination of function arguments from the right. For example, it is possible to omit the right-most argument of the unary pointwise function, `im`, and redefine it as:

```
unaryPointwise f = map (map f)
```

Since `unaryPointwise` is defined as a higher-order function, it is representing rather a category of algorithms than an individual algorithm. Each operation can be defined by passing a certain function to the `unaryPointwise` function. For example:

```
negateImage    = unaryPointwise neg
absImage       = unaryPointwise abs
```

The first function negates each pixel value, and the second takes the absolute value of each pixel.

2.2.4 Binary operations

The binary pointwise operations can be implemented by using `map2` instead of `map`. The `map2` function is defined in the standard environment whose type signature is:

```
map2::(*->**->***)->[*]->[**]->[***]
```

How a binary pointwise function for 2D images is composed is exactly the same as the unary pointwise's case, which is defined as follows:

```
binaryPointwise f = map2 (map2 f)
```

Using this higher-order function, various individual operations can be defined. Following are functions to add two images by passing addition `(+)`, subtract an image from an image by passing `(-)`, take maximum of two pixels by `max2`, and take minimum by `min2`, respectively:

```
addImage = binaryPointwise (+)
subImage = binaryPointwise (-)

maxImage = binaryPointwise max2
minImage = binaryPointwise min2
```

2.3 An Example Miranda Session

A source program is called a *script* in Miranda which is a collection of definitions. The script for the naive version is given in Figure 2-3. This script contains test images such as `im1` and `im2` for simple debugging. For real usage, these test images should eventually be replaced with real images such as ones read from disks.

```
|| naive.m
|| The "Naive" Version

pixel == num
img   == [[pixel]]

|| pointwise operations

unaryPointwise::(pixel->pixel)->image->image
unaryPointwise f = map (map f)

binaryPointwise::(pixel->pixel->pixel)->image->image->image
binaryPointwise f = map2 (map2 f)

|| pointwise examples

negateImage = unaryPointwise neg
absImage = unaryPointwise abs

addImage = binaryPointwise (+)
subImage = binaryPointwise (-)

maxImage = binaryPointwise max2
minImage = binaryPointwise min2

im1 = [[49,31,6,25],[22,20,54,0],[12,64,8,15],[23,5,37,10]]
im2 = [[11,4,25,42],[16,51,20,38],[22,31,23,9],[5,3,10,15]]
```

Figure 2-3 The script, "naive.m"

When Miranda is started it compiles the script and checks types. If the script contains an error, Miranda reports it. Miranda is an interactive system, so that an expression which the user types in is interpreted and evaluated by the system, and the result is returned. If an expression contains a type error the system reports it. If an expression is incomplete as a

sufficient number of arguments is not specified, Miranda still evaluates the expression and returns that it is a function because it is a curried function. A sample session is shown in Figure 2-4, where 'Miranda' is a prompt and a line in bold-face following the prompt shows a user input.

```
                T h e   M i r a n d a   S y s t e m

              version 2.014 last revised 3 May 1990

               Copyright Research Software Ltd, 1989


(100000 cells)
compiling naive.m
checking types in naive.m
for help type /help
Miranda negateImage im1
[[-49,-31,-6,-25],[-22,-20,-54,0],[-12,-64,-8,-15],[-23,-5,-37,-10]]
Miranda addImage im1 im2
[[60,35,31,67],[38,71,74,38],[34,95,31,24],[28,8,47,25]]
Miranda maxImage im1 im2
[[49,31,25,42],[22,51,54,38],[22,64,23,15],[23,5,37,15]]
Miranda addImage im1 (negateImage (minImage im1 im2))
[[38,27,0,0],[6,0,34,0],[0,33,0,6],[18,2,27,0]]
Miranda ?unaryPointwise
unaryPointwise::(num->num)->[[num]]->[[num]] ||defined in "naive.m" line 11
Miranda ?addImage
addImage::[[num]]->[[num]]->[[num]] ||defined in "naive.m" line 21
Miranda addImage
<function>
Miranda addImage im1
<function>
Miranda addImage im1 im1
[[98,62,12,50],[44,40,108,0],[24,128,16,30],[46,10,74,20]]
Miranda negateImage [1,2,3]
type error in expression
cannot unify [num] with [[num]]
Miranda negateImage [['a','b'],['c','d']]
type error in expression
cannot unify [[char]] with [[num]]
Miranda
```

Figure 2-4 Miranda sample session

Chapter 3: From Pointwise to Local Neighbourhood Operations

3.1 Introduction

Although the algorithms are written in the lazy functional language Miranda, the focus in this chapter is not on laziness, but on higher-order functions, polymorphic typing and sophisticated data structures. In other words, the discussion in this chapter may be applied to any language if it has such facilities. The goal of this chapter is to implement algorithms which are still simple but are most commonly used in image processing, i.e. pointwise, image translation, and local neighbourhood operations, such as convolution.

Various techniques are introduced. Firstly, we will introduce the origin of an image into an image data structure, and use a pair of an origin and a list as a fundamental element. Hence, we call the implementation in this chapter the *"with an origin"* version. By introducing an origin, the image boundary problem commonly associated with image translation and local neighbourhood operations can be solved elegantly. Also, the type of a pixel is parameterised as a type variable, so that any type of images, such as gray, boolean, colour, or other types of images, can be handled in a unified manner. Another technique is to make the patterns of 1D and 2D structures identical, by which once operations have been defined on the 1D structure it is fairly straightforward to compose 2D operations by utilising those 1D operations. Since these techniques appear to be very convenient for implementing various image processing algorithms in a modular fashion, they are going to be used through the rest of the thesis.

After the polymorphic and higher-order local neighbourhood function is defined, it is demonstrated that a fairly large number of image processing algorithms can be implemented just by passing individual masks and functions as its arguments. These include the local averaging smoother, Laplacian edge detector and Sobel edge operator. Furthermore, using the same local neighbourhood function, the median filter can be implemented.

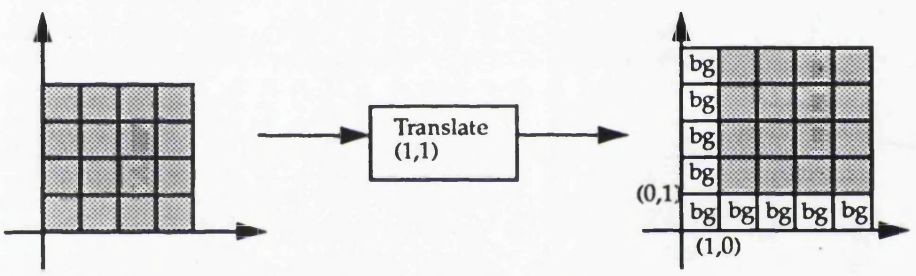
3.2 Image Representation

3.2.1 Why introduce an origin?

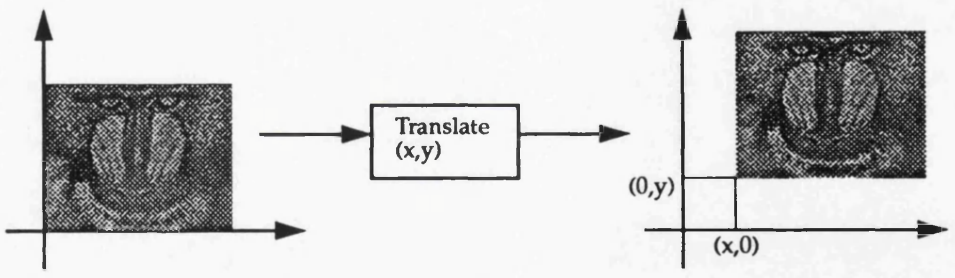
We are going to implement image translation which relates to positioning of an image and an origin is introduced to make this operation easier.

In the previous chapter, we represented an image as a 2D list of pixels (page 31), in which the position of an image was not explicit. In the case of binary pointwise operations, it

was assumed that the first pixel of the first row of each image is at the same position (Figure 2-2). However, using this representation an image may be translated as shown in Figure 3-1 (a) which would seem fairly restrictive; i.e. the amount of translation may be restricted to integer, a background value may have to be filled in if it is desirable that the first pixel retains a certain position, and the image size may increase if it is not desirable to lose the content of the image. Furthermore, translation of a negative amount would seem difficult. Conceptually, however, image translation should change only the position of an image and the content should not change (Figure 3-1 (b)). If the information of a position is kept separate from a pixel array, it will satisfy the above requirement. This is the reason why we introduce an origin as part of the image representation.



(a) Translation of a 2D array



(b) Translation by shifting an origin

Figure 3-1 Translation of images

As for the coordinate system, the conventional right-hand system is used in this thesis rather than the raster scan coordinate commonly used in image processing. The reason is to avoid confusion in geometric operations described in later chapters.

3.2.2 A polymorphic pixel type and interval representation

In the image representation in the naive version the type of a pixel was defined as a type synonym and was actually a number (Section 2.2.1). However, we would like to process not only images whose pixels are numbers, such as gray images, but also boolean images, colour images, complex images such as Fourier transformed ones, and so forth. A polymorphic type should be ideal to deal with these diversified images in a unified manner. Therefore, the type of a pixel is represented by a type variable.

In order to improve modularity, which is a key to ease of writing and reading, a data structure called *interval* is defined which represents the common structure for a row being a collection of pixels, and an image a collection of rows. This is important because once an operation is defined on an interval, then it may be applied to a row, as well as an image.

Based on the above consideration, the representation we take is *"a row as a pair of an x origin (the x coordinate where the row starts) and a list of pixels"* and *"an image as a pair of a y origin (the y coordinate where the image starts) and a list of rows"*. Image origins are expressed in the world coordinate system. The definition for a row and an image is given as follows:

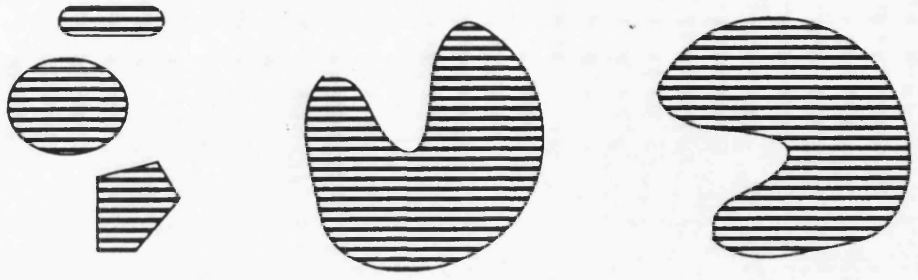
```
interval * == (num,[*])
row *      == interval *
image *    == interval (row *)
```

This representation implements three ideas: (i) the type of a pixel is parameterised as the type variable `*`, (ii) the patterns for a row and an image are identical using the common structure `interval`, and (iii) the information of the position of an interval is incorporated and kept separated from the list part. The length of an interval is made implicit as the length of the list part[1].

Although the interval representation can represent non-rectangular images, there are certain subtle restrictions on possible shapes. Since the representation assumes that the elements (either pixels or rows) in a list are equally spaced, an image cannot be represented if it does not satisfy this criterion. As shown in Figure 3-2 (a), an image composed of more than one discrete object cannot be represented. If an image is non-isotropic, it may or may not be represented depending on its orientation (Figure 3-2 (b) and (c)).

---

1. The problem of not making the length explicit will be discussed in terms of efficiency in Chapter 8.



(a) More than one object (b) Non-isotropic (c) Non-isotropic but representable

Figure 3-2 Non-rectangular shapes

Representing an image as a collection of intervals is not a benefit of using functional languages, as it can be done in any languages which can handle data structures. This issue will be discussed in Section 3.6.5.

3.3 Pointwise operations

3.3.1 Definition of pointwise operations in Miranda

In the naive pointwise operations implemented in the previous chapter (Section 2.2), it is assumed that images are at the same position (though they can be of different size). But now we have the representation of images at different positions and pointwise operations between those images can be defined. Unary operations are straightforward as usually the operations will not change the size and position of an image, but binary operations need consideration. How to handle these operations is a matter of discussion but one possible scheme may be to define the result of a binary pointwise operation only where the both source images are defined. This may be reasonable because a binary operation without two operands usually does not have well-defined meaning [Kozato88a].

Pointwise operations for intervals are defined as follows:

```
unaryInterval::(*->**)->interval *->interval **
unaryInterval f (o,p) = (o, map f p)

binaryInterval::(*->**->***)->interval *->interval **->interval ***
binaryInterval f (o1,p1) (o2,p2)
   = (o, map2 f (drop (o2-o1) p1) p2) , if o1 < o2
   = (o, map2 f p1 (drop (o1-o2) p2)) , otherwise
     where o = max2 o1 o2
```

In order to take the common part in a binary operation, a new origin is calculated as the larger of the origins of two source intervals. `drop` takes a number, n, and a list, and returns the part of the list which remains after the first n elements have been removed. `map2` is similar to `map` but takes three arguments: a function of two arguments, f, and two lists. The result is a list in which each element is the result of applying the function f to the corresponding elements in the two lists. If the two lists are of different lengths, the excess portion of the longer list is ignored.

Note that these functions are defined as polymorphic higher-order functions. That is, as long as the second argument (and third one for a binary function) has the pattern of `interval`, the functions do not care about the content of the list part in the interval. It is the parameterised function, `f`, which cares about the content.

As the data structures of `interval` and `row` are identical, unary and binary pointwise operations for rows are:

```
unaryRow  = unaryInterval
binaryRow = binaryInterval
```

Composition of 2D operations using these 1D operations is exactly the same as the naive version (Section 2.2.3) and is equivalent to the following definitions using the *function composition operator* (.):

```
unaryPointwise  = unaryRow.unaryRow
binaryPointwise = binaryRow.binaryRow
```

These definitions are remarkably simple and easy to write and read because of their conciseness and modularity. Higher-order functions and polymorphic typing are the main contributors. The following subsections give a few example pointwise operations using the higher-order polymorphic pointwise functions defined here.

3.3.2 Simple pointwise examples

Picking up several operators and functions from the standard environment, example unary pointwise operations can be implemented:

```
negateImage  = unaryPointwise neg
absImage     = unaryPointwise abs
logImage     = unaryPointwise log
constImage x  = unaryPointwise (const x)
notImage      = unaryPointwise (~)
doubleImage   = unaryPointwise (*2)
```

Pixel operations used in the examples are as follows: `neg` does the unary minus. `abs` takes the absolute value of a number. `log` gives the natural logarithm. `const` returns a constant value and is defined as:

```
const x y = x
```

`(~)` is logical negation, and the last one `(*2)` multiplies by two.

And the following are a few example binary pointwise operations:

```
addImage = binaryPointwise (+)
maxImage = binaryPointwise max2
minImage = binaryPointwise min2
andImage = binaryPointwise (&)
orImage  = binaryPointwise (\/)
eqImage  = binaryPointwise (=)
```

`(&)` and `(\/)` are logical AND and OR, and `(=)` does comparison.

Type signatures can be omitted; Miranda infers the type of each function correctly. For example:

```
constImage::*->(num,[(num,[**])])->(num,[(num,[*])])
notImage::(num,[(num,[bool])])->(num,[(num,[bool])])
eqImage::(num,[(num,[*])])->(num,[(num,[*])])->(num,[(num,[bool])])
```

This shows that various types of images can be handled in a unified manner.

3.3.3 Look-up table functions

In image processing, it is often necessary to apply a fixed transformation to all pixels in an image. For example, gamma correction to compensate for the characteristics of a display

device, or to map pixel values to colour codes for pseudo-colour display. Such operations are commonly called look-up table operations, because the fixed transformation is normally represented as an array or look-up table. Using the `unaryPointwise` function defined above, a look-up table function of an integer image can be implemented by passing a list indexing function as an argument. For example, an inverse look-up table of an 8 bit integer image, whose pixel values range between 0 and 255, can be written as follows:

```
inverseLookUpTable = unaryPointwise ([255,254..0]!)
```

The operator `!` is *list indexing* to look up a specified element in a list. It should be pointed out that this look-up table works on an 8 bit integer image only, since an index must be an integer although the type of `!` is defined as `[*]->num->*` in Miranda[2]. It is rather awkward for certain applications that a language does not differentiate integers from real numbers because we often use the discrete nature of integers, e.g. as labels.

Since continuous functions can be passed to the higher-order pointwise function, it is easy to implement a continuous image look-up table. For example, a continuous version of an inverse look-up table can be defined as follows:

```
inverseLookUpTable2 range = unaryPointwise ((+range).neg)
```

Needless to say, more complicated look-up is possible, such as a table obtained by histogram equalisation.

3.3.4 Spatial conditional operations

The next example is a *spatial conditional operation*, or *parallel-if*, which is often used for parallel computers and does selection using boolean array objects [Hockney81a]. As illustrated in Figure 3-3, a spatial conditional operation returns pixels from either the second or third image depending on the boolean pixels in the first image.

---

2. `[1,2,3]!1` returns 2, but `[1,2,3]!1.0` is an error in Miranda.

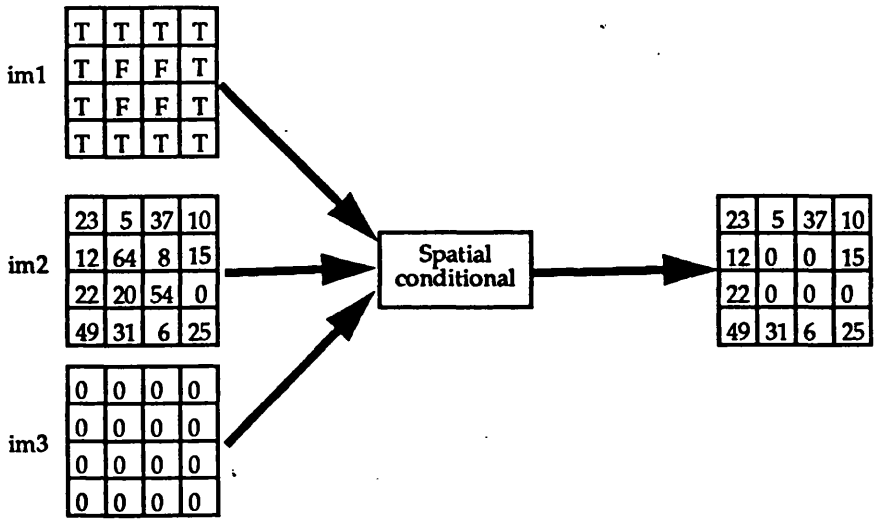


Figure 3-3 A spatial conditional operation

It can be defined in Miranda as:

```
condImage im1 im2 im3
   = binaryPointwise cond im1 (binaryPointwise pair im2 im3)
     where cond True (a,b) = a
           cond False (a,b) = b
           pair a b = (a,b)
```

This function takes three images: the first image is a boolean image and the second and third images are polymorphic images which have the same type. Depending on the contents of the boolean image, the function `condImage` returns pixels either in the second or the third image. Because the type of a pixel is defined as a polymorphic type, operations on mixed image types like this can be implemented without difficulty. Please also note that only the overlapping part of the three operand images is returned.

3.4 Image Translation

Using the image representation with an origin, image translation is very easy just by shifting the origin. There is no need to move pixel data, so it is very efficient. Also, as with the pointwise operations, a 2D operation can be easily constructed from a 1D operation.

```
translateInterval::num->interval *->interval *
translateInterval d (o,p) = (o+d,p)

translateRow = translateInterval
translateImage x y (o,p) = translateRow y (o,map (translateRow x) p)
```

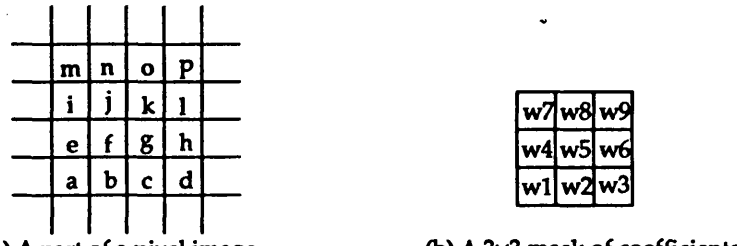
Because this function changes an image origin only, it is possible to express translation of an image by a non-integer amount. At the moment, this does not say much because the image representation uses lists and non-integer indices are not allowed. Even though an image is translated by a non-integer distance, an error may occur if the translated image is operated upon by the other functions, e.g. a binary pointwise operation. A mechanism to interpolate a list to allow non-integer indexing is necessary to reap the benefit. Implementations of interpolation are presented in Chapter 5.

3.5 Local Neighbourhood Operations

A *local neighbourhood operation* is so called because a pixel value is calculated as a result of operations involving the pixel itself and its surrounding pixels. It is also called a mask, template, window, or filter operation, because a mask covering neighbouring pixels is applied to each pixel in an image and certain operations are carried out within the mask. By replacing the size or contents of a mask, or operations to be performed, a fairly large collection of image processing algorithms can be implemented, e.g. image smoothing, image sharpening, edge detection, and so on [Gonzalez87a, Pratt91a]. In other words, a local neighbourhood operation, as with the pointwise operations, gives a certain framework to define various individual algorithms.

For example, it is intuitively understood that if every pixel in an image is calculated as the average value of itself and neighbouring pixels the image will be smoother (*neighbourhood averaging*). Figure 3-4 (a) shows a part of a pixel image in which pixel values are expressed as a, b, c, ... The following equation gives an average pixel value of the nine pixels at the position of the pixel 'f':

$$\text{average} = \frac{1}{9} \times (a+b+c+e+f+g+i+j+k) \qquad \text{(Eq. 3-1)}$$



(a) A part of a pixel image (b) A 3×3 mask of coefficients

Figure 3-4 A local neighbourhood mask

This averaging operation can be explained as applying a 3×3 mask of coefficients (Figure 3-4 (b)) to an image and performing a sum-of-products operation. The nine components of the mask are:

$$w1 = w2 = w3 = w4 = w5 = w6 = w7 = w8 = w9 = \frac{1}{9} \qquad \text{(Eq. 3-2)}$$

and the operation to be performed is:

$$value = a \times w1 + b \times w2 + c \times w3 + e \times w4 + f \times w5 + g \times w6 + i \times w7 + j \times w8 + k \times w9 \qquad \text{(Eq. 3-3)}$$

In order to calculate the whole image, the mask must be scanned across the whole image.

3.5.1 Convolution in Miranda

We consider a Miranda implementation of *convolution*, one of the most important local neighbourhood operations.

Discrete one-dimensional convolution of f(x) and g(x) is defined as

$$f(x)*g(x) = \sum_{m=-\infty}^{\infty} f(m)g(x-m) \qquad \text{(Eq. 3-4)}$$

where g(x) is called a *convolution mask* or *kernel*. And the definition of discrete two-dimensional convolution is:

$$f(x,y)*g(x,y) = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} f(m,n)g(x-m,y-n) \qquad \text{(Eq. 3-5)}$$

The neighbourhood averaging operation described in the previous subsection is a kind of convolution.

An alternative way of viewing the convolution is that it is a collection of pointwise operations between translated images. This method of calculation is commonly used on parallel machines [Clarke86a]. If the image described in Figure 3-4 (a) is translated by (1,1), then the pixel 'a' comes on top of the pixel 'f'. If the image is translated by (0,1), then the pixel 'b' comes to that position. If we repeat these operations eight times changing the directions of image translation, we then have eight shifted images and one non-shifted original. We can then multiply each image by predefined coefficients and add them together, which gives the same result. We call this approach the *parallel method*. In order to make direct use of the functions already defined, i.e. pointwise and translation, which have been designed to operate on the whole image, we will adopt the parallel method.

As in the implementation of pointwise and translation operations, the two-step approach is taken, i.e. construct a 1D operation first, then compose a 2D operation.

Figure 3-5 illustrates how 1D convolution is carried out by combining translation and pointwise operations. For 1D convolution, functions are parameterised as *multiplicative* and *additive* operations, as convolution is based on sum-of-products. The *multiplicative* parameter defines the operation between an element of a convolution mask and an element of a translated image (Figure 3-5 (a)). The amounts of translation are integers within half the size of a mask, i.e. if the size of a mask is 3 the amounts are -1, 0, and 1. Products are calculated by unary pointwise operations by passing the curried function, (*multiplicative a_mask_element*). The *additive* operation defines the operation between the results of the multiplicative operations (Figure 3-5 (b)). Accumulation of the products is carried out by binary pointwise operations by using the *additive* parameter.

In Miranda, 1D convolution can be expressed as follows:

```
conv1 mul add mask im
   = accum [prod (element mask p) (shiftInterval p) | p<-domain mask]
     where
     accum  = foldl1 add
     prod x = unaryInterval (mul x)
     shiftInterval p = translateInterval (p-((leng mask) div 2)) im

domain    = index.snd
leng      = (#).snd
element   = (!).snd
```

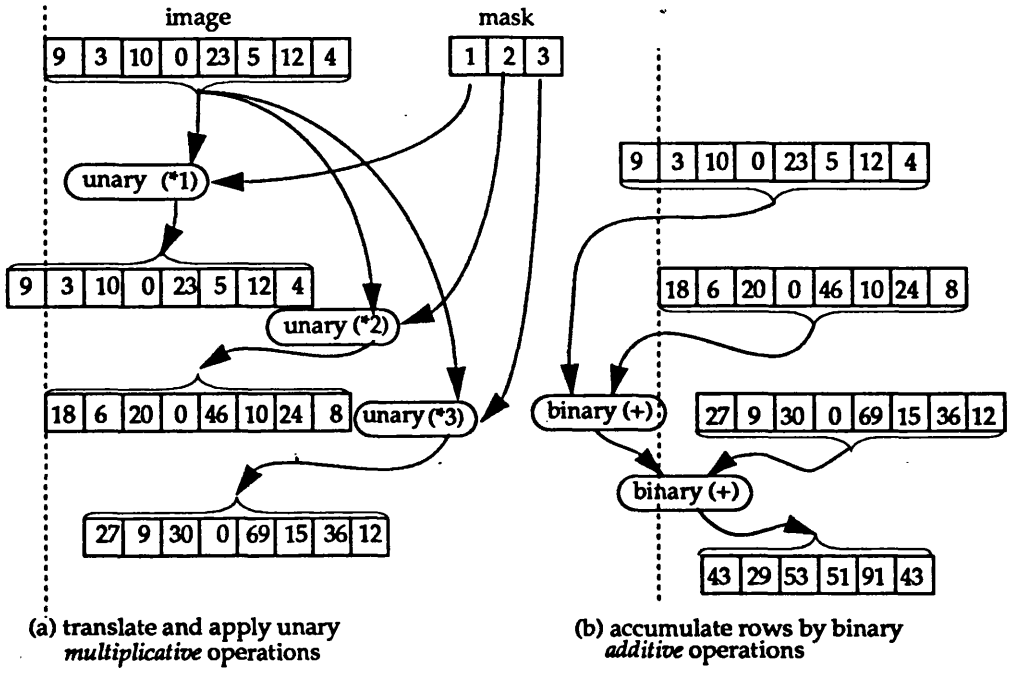


Figure 3-5 1D convolution by *multiplicative* and *additive* operations

The main definition of `convl` uses a *list comprehension* [Bird88a], in which the right-hand side of the vertical bar works as a generator, or a *qualifier*, and the list of the results from the expression of the left of the bar is returned. It employs a syntax adapted from conventional mathematics for describing sets. In the above code, `domain` returns a list or the indices of the list part of a mask by using `index` defined in the standard environment. `snd` is a function to take the second element in a pair, i.e. the list part. For example, if a mask is `(0,[1,2,3])` as in Figure 3-5 its domain will be `[0,1,2]`.

`prod` does the multiplicative operations described in Figure 3-5 (a) by passing the multiplicative function (`mul`) with a mask argument to the unary pointwise operation. Translation of an interval is expressed in `shiftInterval p` where `p` is an element of the domain of the mask. The interval is shifted by `p` to the right and a half of the size of the mask to the left, i.e. if the mask size is 3 the amounts of translation are -1, 0, and 1. The `#` operator used in `leng` calculates the length of a list and `div` does an integer division[3].

3. Note that, because of Miranda's laziness, an expression such as `leng mask` is only evaluated at most once for each evaluation of `convl` (in fact, it will be evaluated exactly once, since its value is always needed if convl's value is required). See Chapter 4.

`accum` is in charge of accumulating the results obtained by the `prod` operations. As shown in Figure 3-5 (b) a list of intervals is accumulated by the function argument `add`. `foldl1` is a higher-order function defined in the standard environment and is defined informally as follows:

foldl1 add [i1, i2, ..., in] = (add ... (add (add i1 i2) i3) ... in)

The *fold* operation family, like the map family, are also important higher-order functions defined on lists. There are a few variations of fold operations depending on the direction of an operation and whether it allows folding a non-empty list. `foldl1` is defined on a non-empty list and folds up the list from the left.

This `conv1` function is the actual "working part", which is defined polymorphically to accept either a row or an image. The top-level definition for convolution is just to give a suitable interface above it. Separate functions for convolution of rows and images can be defined simply as follows:

```
convolveRow   = conv1 (*) (binaryRow (+))
convolveImage = conv1 convolveRow (binaryPointwise (+))
```

`convolveRow` uses multiplication as the operation between an element of a mask and a pixel in a row, and a binary pointwise addition of rows which specifies the operation between the results of the multiplications. `convolveImage` takes `convolveRow` which defines the 1D convolution between a 1D image and a 1D mask, and a binary pointwise addition of images which specifies the operation between the results of the above convolutions. In this way convolution can be defined elegantly using Miranda.

3.5.2 Some examples of convolution - Smoothing, Laplacian and Sobel

The convolution function gives a framework and individual algorithms can be implemented by passing appropriate arguments to the function. For example, smoothing an image by neighbourhood averaging, as described on page 45, can be implemented as:

```
average = convolveImage (makeMask [[1/9,1/9,1/9],
                                   [1/9,1/9,1/9],
                                   [1/9,1/9,1/9]])

makeMask::[[*]]->image *
makeMask m = fn (map fn m) where fn x = (0,x)
```

Here, `makeMask` is a simple conversion from a 2D list to its image structure to be used as a parameter to the convolution function.

Likewise, *Laplacian edge detection filter* can be implemented by replacing the coefficients in the mask as follows:

```
laplacian = convolveImage (makeMask [[ 0, 1, 0],
                                     [ 1,-4, 1],
                                     [ 0, 1, 0]])
```

The same principle is seen in the following more elaborate example *Sobel edge operator*, where the gradient can be commonly approximated by adding absolute values of horizontal and vertical edge strengths [Gonzalez87a]:

```
sobel im
   = binaryPointwise (+) (absImage imh) (absImage imv)
     where imh   = convolveImage (makeMask hMask) im
           imv   = convolveImage (makeMask (transpose hMask)) im
           hMask = [[ 1, 2, 1],
                    [ 0, 0, 0],
                    [-1,-2,-1]]
```

The vertical mask can be obtained by transposing the horizontal mask using `transpose` which is defined in the standard environment.

3.5.3 Minimum and maximum filters

The examples described in this and next subsections are called *rank filters* which are local neighbourhood operations but are not conventionally regarded as convolution operations. We demonstrate the convenience of higher-order functions that those designed above for convolution can be used for rank filters without a change.

At each point in an image the *maximum filter* finds the maximum value in the local neighbourhood; correspondingly, the *minimum filter* finds the minimum value. Unlike the linear filters described in the previous subsection, these filters cannot be implemented only by changing the coefficients in a mask and it is necessary to change the function arguments to be passed to the `conv1` function. The implementation is:

```
maxFilterRow   = conv1 secondArg (binaryRow max2)
maxFilterImage = conv1 maxFilterRow maxImage

minFilterRow   = conv1 secondArg (binaryRow min2)
minFilterImage = conv1 minFilterRow minImage

secondArg a b = b
```

Note that the mask used in these filter functions specifies the size of a neighbourhood; the values of mask pixels are not significant.

3.5.4 Median filtering

The final example, the *median filter*, is also a rank filter. This filter takes the median within a neighbourhood, which is quite a common and effective method of noise removal. In the local neighbourhood examples described so far, the operations such as sum-of-products or finding a min/max value are separable in x and y directions, i.e. how the elements are grouped and in what order the operations are executed does not matter. So, "rows first, then an image" does not make any difference from any other calculation order. However, in the median operation, the median of the medians of rows may not be the same as the median of the area. A conventional method of median filtering is to produce a histogram of local pixels, then take the median of the histogram. Our approach is similar: first concatenate the neighbouring pixels into a list, which gives an image whose pixels are lists, or a *list image*, then take the median of each pixel list in a pointwise manner. Since production of a list of neighbourhood pixels can be done in any order, there is no need to modify the basic computation structure and hence the basic code `conv1` can be used unchanged. As a result, the implementation is surprisingly elegant:

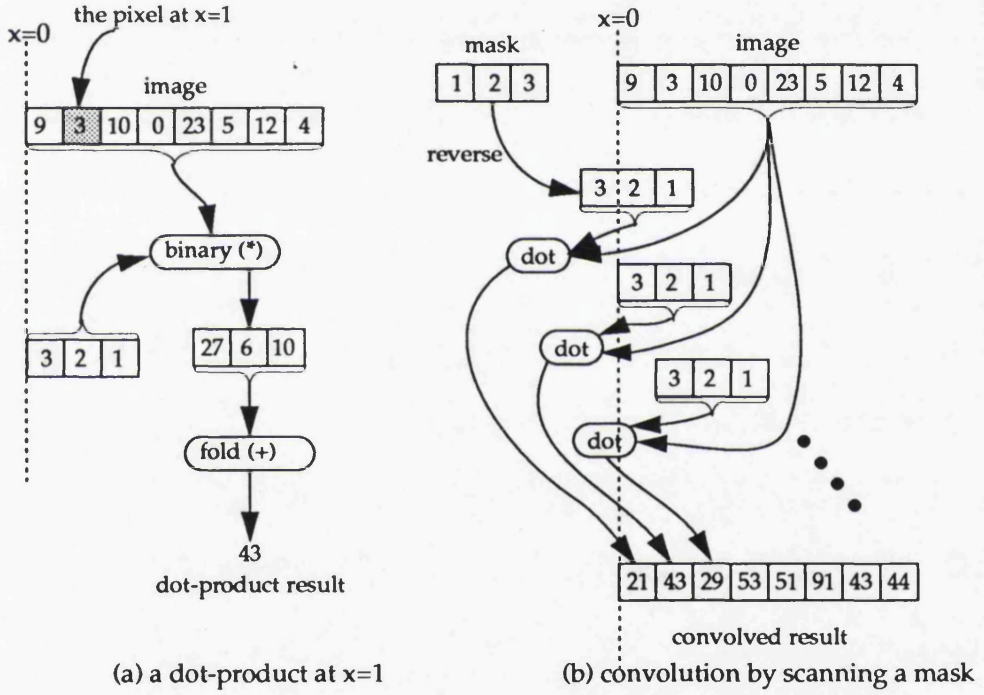
```
medianImage mask = (unaryPointwise median).(localHistImage mask)
median list = (sort list)!(#list div 2)

localHistImage = conv1 (localHistRow) (binaryPointwise (++))
localHistRow = conv1 f (binaryRow (++)) where f a b = [b]
```

Here, `sort` is a standard environment function for list sorting, and ++ is the operator to concatenate two lists.

Rank filters are not normally considered as examples of convolution, but the polymorphic higher order function designed for convolution is so general that it can be used to implement them successfully.

3.5.5 An alternative implementation of convolution

From a readability viewpoint, the above convolution function `conv1` may not be the best, since the modularisation is not the same as the most common definition of convolution (Eq. 3-4). Convolution is commonly defined as follows: a mask is turned over and scanned across an image, and a pixel is calculated as the dot product of the mask and the image at a certain position. If a mask is not turned over it is called *correlation* [Gonzalez87a]. Figure 3-6 illustrates how convolution faithful to the definition is carried out.



(a) a dot-product at x=1 (b) convolution by scanning a mask

Figure 3-6 Convolution – an alternative method

The following is a Miranda implementation of row convolution which attempts to trace the definition relatively faithfully:

```
conv2 mul add (mo,msk) im = correlation mul add (mo,reverse msk) im

correlation mul add mask im
   = (fst im, [dot mul add mask im x | x<-domain im])

dot mul add mask im x
   = accum (shiftMask $times im)
     where
     accum     = foldim add
     shiftMask = translateRow (x-((leng mask) div 2)) mask
     times     = binaryInterval mul

foldim op = (foldl1 op).snd
```

conv2 is defined as correlation; a mask is flipped by the `reverse` function which is defined in the standard environment.

In the definition of `correlation`, the qualifier in the list comprehension is the domain of the image. This means that the output image retains the domain of the original image, and hence the origin of the original image should be attached to the list of dot-products by the `fst` function which returns the first element in a pair. This is enabled by the property of the dot-product function which returns the dot-product of the overlapped area between an image and a mask. If the same result as `conv1` is desired, it is possible to do so by changing the domain and the origin appropriately.

The dot-product of a mask and an image at the position x is defined as `dot`, where the operations to produce a dot-product are parameterised as `mul` and `add`. Note that the type of the `add` function is different from the one previously used (page 47), since in the case of a dot-product of a row as in Figure 3-6 (a) the additive operation adds up (or accumulates) pixels, not rows. The sub-definition `accum` does this job. `shiftMask` aligns the mask and the image appropriately using the designated position x and the length of the mask. `times` defines the multiplication between the mask and the image by a binary pointwise operation. In the main definition of `dot`, this operation is made into a *do-it-yourself infix operator* by attaching the symbol `$` [Bird88a] attempting to make the code slightly more readable.

Using the new definition of convolution, a row and an image convolution can be expressed as follows:

```
convolveRow2   = conv2 (*) (+)
convolveImage2 = conv2 convolveRow2 (binaryRow (+))
```

It is very difficult to conclude which implementation is more readable. A feature of this implementation is that a convolved image retains the position and size of an original image. Also, programmers can alter the origin and size by changing the origin and the domain in the definition of `correlation`. Treatment of the position and edges of a convolved pixel image is a relatively difficult and generic problem which is discussed in the next subsection.

3.5.6 Convolution and image boundary problems

Our representation of an image with an origin give a solution to what is called *the image boundary problem*, or *border effect*, usually associated with local neighbourhood operations. Since a digital image has a finite size and outside the boundary can be considered as undefined, in the resulting image of a local neighbourhood operation, the pixel values adjacent to the image boundary, or more precisely, within a half the size of the mask, should be undefined. However, since there is often the restriction that an output image has the same size as an input image, which is to some extent related to representing an image as a 2D static array (Section 3.2.1), certain values are filled into the boundary pixels.

There are several other approaches to the image boundary problem. One approach is to use restricted local neighbourhoods at the boundaries, e.g. the average over only those pixels within the image where the local neighbourhood overlaps the boundary. Other approaches assume certain pixel values outside the image area, for example: the *centered, zero padded superposition* assumes outside the boundary of an image is filled with zero; the *centered, reflected boundary superposition* assumes that an input image is reflected at the border; and the *centered zero boundary superposition* fills zero in the pixels near the border of an output image [Pratt91a]. Each of these approaches works well in some circumstances and badly in others. The approach we used always gives correct answers; however it sometimes discards useful information at the boundary.

In retrospect the parallel method implemented in Section 3.5.1 uses the assumption that outside an image is undefined, so that the resulting image of a convolution shrinks, or is "peeled-off" since the edge pixel values are not well-defined. One of the advantages of our image representation with an origin is that it is easy to alter the position or size of an image. This is why we can easily maintain the domain of a convolved image where the result is well-defined.

The alternative method in Section 3.5.5 assumes that the area outside an image is filled

with zeros, i.e., the centred, zero padded superposition has been implemented. There is no difficulty in maintaining the domain of an original image by using the function `domain` to generate the original domain. In addition, the nature of the binary pointwise operation which returns only the overlapped area of its two operands works very well to calculate the dot-product of two rows.

Another solution to the image boundary problem may be to define an undefined pixel properly. For example, a pixel can be defined as either undefined or a value, expressed in Miranda using an *algebraic data type* as:

```
pixel * ::= Undefined | Value *
```

An algebraic data type is a simple mechanism for users to define a new type. `Undefined` and `Value` are called *constructors* which must be capitalised in Miranda.

However, although this algebraic data type for a pixel implements an undefined pixel, it would be necessary to define all the operations on this pixel type: even a small operation such as "adding two pixels" would need to be redefined. In addition, since Miranda does not support overloading, each operation must have a unique name, i.e. adding two pixels cannot be written using a plus symbol, and this would lead to quite unreadable code. Haskell, on the other hand, supports overloading which may be useful to handle the type of a pixel as a tagged union of a value and 'undefined'.

3.6 Discussions – What Are the Benefits?

3.6.1 Programming in functional style

As described in Section 1.1, image processing may be most commonly described as a certain function applied to input images or descriptions which yields output images or descriptions. Therefore, the very basic style of functional programming, i.e.

output = function (inputs)

should be suitable for describing image processing applications. Most conventional imperative languages including Fortran, Pascal, C and C++, allow this functional style notation, as opposed to the procedural style. A series of function applications can be written as:

output = function2 (function1 (inputs))

The use of functional notation can be understood as general convenience to avoid side-effects and improve readability, and has been discussed in various programming paradigms not necessarily in functional languages. This includes the work by Breuel [Breuel92a] and Sato *et al.* [Sato90a], who both used C++ to implement image processing functions and discussed the convenience of functional notation. However, in these conventional imperative languages, a function can take other functions as arguments but cannot return a function as its return value[4].

Whereas, functional languages[5] provide higher-order functions which can both take and return functions, and thus they allow composition of complicated functions using simple ones as their arguments. Also, many functional languages including Miranda have a mechanism to pass operators to a function, since an operator is just another form of a function. In addition, some syntactic facilities, namely the function composition operator and currying, add more convenience to functional languages.

For example, when a sequence of functions is applied to inputs, as in the above pseudo code, it may be described as[6]:

```
compositeFunction = function2 . function1
output = compositeFunction inputs
```

This style, i.e. applying a series of functions to inputs, is very typical in image processing, and the resulting code is fairly readable. For example, the median filter function (`medianImage`) is defined as a composite function of `localHistImage` and `unaryPointwise` (page 51).

Currying is also useful since it can eliminate unnecessary arguments in definitions, and allows the programmer to supply some but not all arguments in applications. For example, in the basic convolution function (`conv1` on page 47), the multiplicative operation `mul` is a binary operation, but it makes a unary operation as the curried function, `mul x`, which is

---

4. Some of these languages, e.g. C or C++, allow pointers to functions to be passed and returned. This provides some of the power but not all; for example, it does not support partial application, in which a function has some parameter values filled in in one function, and is then passed to another. An example of this in image processing occurs if a pixel gamma correction function (which has two arguments: the input pixel value, and the gamma value) is being passed to a lookup-table function, which expects an image and a function of one argument.
5. Including languages such as Scheme and Common Lisp, in this context.
6. In Miranda a sequence of functions should be read from right to left, which may seem unnatural for some.

passed to the unary pointwise function. Also, when various pointwise operations, such as `addImage` and `eqImage`, are defined (Section 3.3.2), it is obvious that `addImage` takes two images, and so does `binaryPointwise`. So, attaching two arguments for images in the definition is redundant.

3.6.2 Use of higher-order functions

As discussed in Section 1.1.2, there are many common control structures in image processing programs, and therefore higher-order functions should be particularly useful to handle various operations in a modular manner. We have successfully demonstrated the convenience this offers with a number of examples.

In the pointwise functions, for example, the actual operation on pixels is given as an argument, rather than hard-coded in the function. The local neighbourhood function also takes functions, *multiplicative* and *additive*, as its arguments. Despite these names, these operations are not necessarily multiplication of pixels and pointwise addition of intervals, but can be anything, as long as they are not ill-typed[7]. Using these functions a large number of operations have been implemented. There are three kinds of usage of the higher-order local neighbourhood function:

- Linear operations by replacing a mask, which include neighbourhood averaging, Laplacian, and Sobel edge operator. Most languages can cope with this category, as data can be taken as an argument to a function.

- Non-linear but separable operations, such as minimum and maximum filters. Because the higher-order local neighbourhood function works on "1D first then 2D" basis and functions such as min and max can be calculated in any order within a neighbourhood, these operations can be implemented by replacing the arguments for additive and multiplicative operations.

- Non-linear and non-separable operations, such as median filtering. In this case, the local neighbourhood function is used as an access method to neighbouring pixels. The neighbourhood pixels are concatenated to form a list image, and then the actual operation, such as taking a median, is mapped onto the list image in a pointwise manner.

---

7. However, if `mul` is not commutative, or `add` is not both associative and commutative, the result may not be equal to the value specified by the definition of convolution in Section 3.5.1.

Median filtering is not normally regarded as a form of convolution, but our higher-order function designed for convolution is so powerful that it has been implemented by simply passing appropriate arguments to the function.

The benefit of higher-order functions can also be described as separating control and operations which improves modularity of programs. The higher-order version of image operations can be regarded as describing a computational structure, and the actual operations are given later as the actual arguments. Those structures and operations are defined separately, and are reusable. Even if an image representation is changed and as a result the higher-order functions have to be modified, the actual operations may not need to be changed.

Although a number of people have stated this convenience for image processing applications (e.g., [Allsop91a, Breuel92a]), it is still surprising to see that such a large number of algorithms can be implemented based on just the pointwise, translation and local neighbourhood functions.

3.6.3 Construction of 2D operations from 1D

In defining the representation of an image, we took the basic strategy that an image is a collection of rows and a row is a collection of pixels. This representation improves modularity of programs considerably. The idea is to define a common data structure for rows and images using a polymorphic type: both images and rows are intervals of *something* (`"interval *"` in the code). The basic implementation style is to consider a 1D version first, which is usually much easier to implement and debug. This 1D operation should be defined as a higher-order polymorphic function so as to take itself as its argument. Then construct a 2D operation, which is fairly simple and easy to understand. This strategy is so important that it is going to be used even in more complicated operations described later.

A similar idea has been presented by Runciman [Runciman92b]. He implemented a Haskell version of a software package called TIP [Thimbleby87a] for use in interactive terminal-based programs. He noted the close similarity of the line (1D) and screen (2D) data structures and proposed *self-supporting delta structures*. Using this polymorphic type, he demonstrated that the two data structures can be defined as instances of the same polymorphic type.

3.6.4 Polymorphic typing and image categories

In Section 3.2.2 the type of a pixel was left as a parameter, or defined as a polymorphic type, and subsequent programming has been done in a polymorphic manner. This enables us to handle various categories of images, such as a gray, colour, boolean, or label image, in a unified manner. In other words, as far as programming is concerned, there is no need to worry about the category of images and no need to write separate code for each category. Miranda's type system infers the most general type as possible at compile time. In this way, polymorphic typing allows us to deal with various kinds of images in a unified manner.

As discussed in Section 1.1.3, there is a clear distinction between polymorphic strong typing and untyped systems, such as Lisp. Although untyped languages may also accept any type of pixels, they do not type-check at compile time, so that writing correct code may be rather difficult especially when complicated data structures are being used. On the other hand, polymorphic type systems allow any type and programs are type-checked at compile time, which may be considered to be a very convenient feature for image processing programming.

3.6.5 Using flexible data structures

We have introduced an origin to the image data structure, which gives various benefits to image processing programs described so far: the binary pointwise operation returns a resulting image only where both operand images are defined, so that it can cope with binary operations between images of different size and position (Section 3.3.1). Also, image translation is remarkably simple and efficient, requiring only shifting an origin. It also allows translation by a non-integer amount (Section 3.4)[8]. In addition, an origin solves the image boundary problem associated with local neighbourhood operations (Section 3.5.6).

Introducing an origin to an image data structure can be done in most programming languages by using, e.g., record in Pascal or struct in C. For instance, an image processing library known as Woolz [Piper85a] stores the domain and the value of an image as separate members in a data structure. A benefit of its implementation is that it is possible to write a function to calculate the area of the union of two images without involving their data part. Also, the data outside the overlapped area in a binary operation can be retrieved if it is desired. Whereas in our representation, the image position is explicit but not the size, and the treatment

8. How to handle images which are not aligned with an integer grid will be described later in Chapter 5 and Chapter 6.

of overlap is embedded in the basic list operation (`map2`). Therefore, redefinition of an image representation will be necessary if the above "area of union" function is to be implemented. However, in order to use these data structures in imperative languages, programmers usually have to allocate and deallocate memory explicitly. While in functional programming, use of those sophisticated data structures is very straightforward, as seen in the examples. This could be done using languages with automatic storage management, such as Lisp and Scheme, but these languages lack static type checking as described in Section 3.6.4, so that programming using these data structures may be more difficult.

3.7 Summary

In this chapter we have implemented the "with an origin" version which incorporates an origin in the image data structure. Using this representation a large variety of image processing algorithms have been implemented. Since the techniques developed in this chapter are not exploiting the laziness of Miranda, they may be applied to any programming languages which share the features we have used. In the next chapter we will focus on laziness, which is the main subject of this thesis, and discuss how it can be utilised for writing image processing programs.

Chapter 4: Laziness in Image Processing

4.1 Introduction

While the main theme of this thesis is laziness and how lazy languages can express image processing programs, the discussion so far has not particularly concerned lazy languages. That is, most functional languages, lazy or eager, and to some extent non-functional languages may have the same benefits if programmed with the same principle.

In this chapter the discussion is focused on laziness. First, laziness in the context of functional programming languages is described, where lazy evaluation is an implementation technique to allow non-strict semantics. Then, discussion moves on to the relationship between laziness and image processing. The meaning of the term 'laziness' is slightly different in the context of image processing, but it is a convenient feature to improve efficiency, and to improve modularity because programmers have to take little care in connecting functions together. The last part discusses algorithms which are considered to be particularly suitable if expressed in lazy languages. We call these algorithms *inherently lazy* and lazy languages can express these algorithms in a natural fashion.

4.2 Lazy Evaluation

Lazy evaluation is an implementation technique of functional languages which allows *non-strict semantics*. If a function is non-strict, even if it is applied to an ill-defined argument, such as an error, undefined, or non-terminating, it may not produce an ill-defined result. As an example of non-strict functions, let us consider an infinite list with some ill-defined elements and a function to select an element from the list using list indexing:

```
list = [1/0,undef,sum[0..],3]++[4..]
select = (!)
```

where `undef` is *'a completely undefined value'* defined in Miranda which has a polymorphic type. `sum` is applied to a list of numbers and returns their sum. `[0..]` is an infinite list starting at 0 and increasing by 1.

The following select operations succeed or fail depending on which element is required:

```
Miranda select list 0
program error: attempt to divide by zero
Miranda select list 1
program error: undefined
Miranda select list 2
<<not enough heap space -- task abandoned>>
Miranda select list 3
3
Miranda select list 1000
1000
```

An expression that does not denote a well-defined value in the normal mathematical sense is called *bottom*, written using the symbol ⊥. A function is said to be *strict* if it returns bottom whenever any of its arguments (or any part of its arguments) is equal to bottom. All other functions are non-strict. In the above example, the function `select` is applied to `list` which includes some elements which do not have a well-defined value, but `select` can still return a well-defined value. So it is non-strict.

4.2.1 Elements of lazy evaluation

This subsection briefly describes how laziness is achieved in the context of implementation of lazy functional languages. For more details, see [Peyton Jones87a] and [Peyton Jones92a]. The typical implementation of lazy evaluation includes the following three techniques:

1. **Normal order reduction**

   As most functional languages are based on the lambda calculus, the implementation is usually explained as reduction of a lambda expression to its normal form. *Normal order reduction*, or *outermost reduction*, describes the reduction strategy which reduces the leftmost *redex* (reducible expression) first if there is more than one redex. It ensures that an argument is required only when it is necessary because, in function application, the outermost redex is the function application itself. So, normal order reduction always evaluates the function before evaluating its arguments.

2. **Graph reduction**

   Let us look at an example first: given the function definition, `sqr x = x * x`, if we reduce the expression, `sqr(4+2)`, normal order reduction requires more reduction steps than its counterpart, called *applicative order*, or *innermost reduction* [Bird88a]. This can be solved by a technique called *graph reduction* which replaces a shared expression

with a pointer to avoid the same expression from being evaluated more than once. In this way, *normal order graph reduction* ensures that an argument is calculated only when it is required, and that it is evaluated at most once. The number of reduction steps is never more than applicative order reduction.

3. **Stop at weak head normal form**

   If, for example, an expression to be reduced consists of structured data, such as a list or a tree, it may not be necessary to reduce the term into the normal form because only some elements in the structure may be required. On the other hand, certain information will always be required. So reduction should stop at a stage called *weak head normal form*, — see, e.g. [Peyton Jones87a] for a definition. For example, an expression of the form *(expression1:expression2)* is in weak head normal form even if each expression in the list may not be normal form. The reduction does not proceed until which element is required is known.

4.2.2 Space efficiency of lazy evaluation

If we talk about efficiency, not only speed but space consumption is also an important factor. As Hughes discussed [Hughes84a], if a program involves a simple flow of data, such as counting the number of characters in a file, lazy evaluation provides a space efficient solution. Whereas, if there is branching or merging of streams of data which requires synchronisation between the streams due to the difference of data consumption speed, lazy evaluation may accumulate intermediate data structures. He proposed the use of language constructs to control the behaviour of programs. However, it has to be admitted that these constructs are rather difficult to use. Separately, Wadler tackled this problem and proposed the *listless transformer* to automate elimination of intermediate structures [Wadler84a]. However, the applicability of this technique is fairly limited. More recently, this problem is called *deforestation* [Wadler88a] to reduce intermediate tree structures and is still a difficult problem.

There is a lot of subtlety in space behaviour of lazy functional programs and it is difficult to predict or optimise the memory usage. For example, *space leaks* [Peyton Jones87a]. One possible suggestion from an application programmer's point of view is to use profiling tools [Runciman93a, Sansom93a] which provide information on space and time behaviour of a lazy functional program. Using this kind of tool, programmers may be able to analyse their programs and improve them (See Chapter 8).

4.2.3 Drawbacks of lazy evaluation

As shown above lazy evaluation ensures that any subexpression is not evaluated until it is required, and that it is evaluated at most once. Therefore, on the one hand, lazy evaluation is more efficient because it minimises the number of reduction steps. Also, if a program deals with only a simple flow of data stream, memory usage should be efficient. On the other hand, apparently, it involves a bigger overhead because each time reduction proceeds the system has to check whether the expression is in weak head normal form and decide whether or not to proceed. So, a drawback, but quite a major one, would be its execution speed. It should be said that, as far as speed is concerned, lazy evaluation pays off when the saving effect is greater than the overhead cost. However, it is not only efficiency but also modularity of programs that matters to image processing programming, as discussed in the following.

4.3 How Laziness Contributes to Image Processing

4.3.1 Laziness contributes to efficiency

In image processing applications, the size of data to be handled is usually large and operations are expensive. Therefore, by reducing unnecessary operations, the efficiency should be much improved [Lau-Kee91a, Kozato92a]. Poole [Poole92a] also addressed this issue and is currently working on a system which allows overloading of multiple image representations using Haskell. In his system, these internal representations are not visible to an application programmer and the system automatically converts an image between different representations. Since the system works lazily, conversion takes place only when it is necessary, which may save a large amount of computation.

For example, if we compare an image with a filtered image to check the effect of the filter we would define the following function:

```
diffImage filter im = binaryPointwise (-) im (filter im)
```

Let us consider the case where an image is derived as the result of an expensive operation, i.e. how the expression:

```
diffImage filter (expensiveOperation image)
```

is reduced. Using graph reduction, `expensiveOperation` is evaluated only once, although the argument `im` appears twice in the right-hand side of the definition of `diffImage`.

Also, in image processing, the whole image may be not always required, but only a partial image or an image only in reduced resolution may be required. For example, if a user of an image processing system is interested in a particular part of an image, such as a suspicious part in a medical diagnostic image, only that part should be needed. If a satellite image of the size 6000 × 6000 is displayed on a TV resolution screen of the size 500 × 500, pixels should be interleaved either by sub-sampling or averaging. In these cases, a lazy system is considered to be more efficient because it evaluates only the required parts of an image. We will use a display function to produce demand for pixel evaluation because in order to display an image every pixel in the display must have a value. Note that a display function is just an example of a function which requires pixel evaluation. This may be done in lazy functional languages by evaluating the following:

```
display parameters image
```

where *parameters* should contain the information such as size and resolution of the display. Lazy languages should be able to propagate the demand through to the input image, where unnecessary operations are eliminated. This is where lazy languages are regarded as particularly suitable, and will be implemented in Chapter 5 and Chapter 6.

4.3.2 Laziness contributes to modularity

Hughes argued that laziness improves programs' modularity because different parts of a problem can be implemented as different functions which can be put together using lazy evaluation as the "glue" [Hughes89a]. This may be true for image processing, too, if programmers choose to utilise laziness as the "glue" to put functions together.

For example, let us consider a very simple operation, *adding three images*, in an imperative language and a lazy functional language.

Assume that an image is represented as a 2D array in an imperative language. A common control structure to handle such an image is a double loop. In order to avoid generating intermediate results, programmers may like to put several operations in one double loop. Let us call a function to do this `add3`. In pseudo code with a C-like syntax, this `add3` function may be written as follows:

```
for (j=0; j<YSIZE; j++)
   for (i=0; i<XSIZE; i++)
      result[i][j]=im1[i][j]+im2[i][j]+im3[i][j];
```

It would also be possible to define a function to add two images. Let us call it add2 which can be defined as follows:

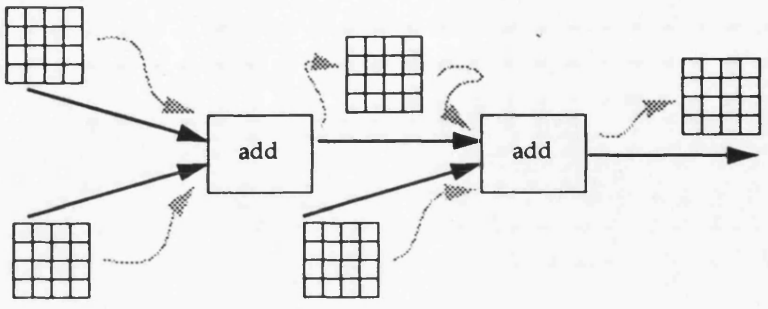
```
for (j=0; j<YSIZE; j++)
   for (i=0; i<XSIZE; i++)
      result[i][j]=im1[i][j]+im2[i][j];
```

Using this add2 function, it is possible to compose addition of three images by using functional notation, i.e., `add2(add2(im1,im2),im3)`. But this composite does not work in the same way as the above add3 because the usage of memory is different and hence the sequence of operations is different. This is considered to be a serious loss of modularity.

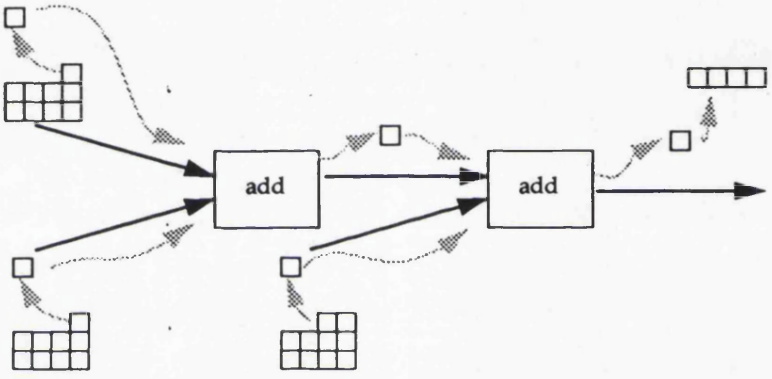
In lazy functional languages, on the other hand, if we pick a suitable representation for an image, there is not significant loss of performance even if we add three images by the style of `add2(add2(im1,im2),im3)`. Because, in this case, a piece of demand is to produce a pixel of the resulting image and the system automatically fetches necessary pixels from the input images.

Figure 4-1 intuitively shows the above discussion, where the "program" is expressed as the boxes and black arrows, and the gray arcs denote image data flows. In an imperative language, it is the programmers' role to decide which method is taken. Because (b) is more space efficient, programmers may wish to process in pixel by pixel manner by putting several operations in one double loop, which can be considered to be loss of modularity. Whereas in lazy functional languages, with a suitable representation, the method (b) may automatically be taken. Programmers do not have to worry about the storage for intermediate results because it is handled by the system.

Figure 4-1 (b) handles an image as a lazy list of pixels, but of course there are other representations which implement different views of laziness in image processing. For example, if an image is represented by a hierarchical data structure such as a quadtree, it may be possible to produce a demand at a certain resolution. Hierarchical data structures are discussed in Chapter 6.



(a) Image by image manner



(b) Pixel by pixel manner

Figure 4-1 Adding three images in two ways

4.3.3 Degrees of laziness

There are various levels of laziness and 'full' laziness is representation dependent. For example, there are lazy image processing systems which allow laziness only at the image level. Poole's image analysis system [Poole92a] uses Miranda as the front-end of the system. However, images themselves are kept in the back-end image processing server. The principal reasons for this division are execution speed and the need to utilise existing image processing resources. Therefore, within Miranda, an image is not separable into pixels, but dealt with as one substance expressed as the type `image*` which is merely a string to store an image id. VPL1.0 [Otto92a] is a visual programming system for image processing whose underlying computational model is demand-driven functional programming. It uses a separate C++ library as the image processing back-end which does not allow pixel level operations. Thus, like Poole's system, although VPL itself works lazily, the degree of laziness is restricted to the image level.

As described in Section 4.2.1, in the context of functional languages, 'full' laziness is usually defined in terms of function applications. So, the maximum degree of laziness depends upon the level of the primitive data types. For example, if an integer is represented in binary form, as a stream of bits, least significant first, then evaluating `even n` will not require evaluating the more significant bits. Thus, it is possible that evaluating:

```
even (2 * expensive_function_known_to_return_an_integer)
```

would not need evaluate the expensive function. If, however, an atomic representation is used for integers, then primitive operations will automatically calculate all of the bits if any are needed, and so this laziness cannot occur. So, the degree of laziness depends on the representation and coding of data.

For images, this perhaps becomes clearer: an image represented by bitplanes could be lazy about not giving high precision; whereas an image represented by a tree of floating point numbers could be lazy about not evaluating unwanted areas of the image. In the implementation described so far, an image is basically represented as lists of pixels, and lists behave lazily in Miranda. Therefore, the degree of laziness is at the pixel level, i.e., only required pixels are calculated.

In order to get the benefit of this kind of lazy optimisation, the language must be implemented using knowledge of the data representation and it would be impossible to implement a language with every possible data coding method. Nevertheless, it may be a good idea to embed certain knowledge if the language is biased toward some specific applications.

4.3.4 Image processing and infinite data structures

The primary motivation of lazy evaluation is to allow non-strict functions, and one of its immediate benefits may be the use of infinite data structures. This allows very elegant solutions to various problems, e.g. finding prime numbers [Bird88a], Newton-Raphson square roots, and numerical differentiation [Hughes89a]. Also, for graphics applications, infinite data structures such as fractals [Parsons89a] and quadgraphs [Parsons86a] can be implemented elegantly. What about image processing applications?

An image is usually very large, but not infinite. So, unless a program is erroneous, it will terminate even if it takes a long time. In this sense, a language for image processing does not

need to be non-strict. Therefore, if a significant gain can be achieved by eliminating non-strictness, but retaining other features of laziness, e.g. the elimination of unnecessary operations, then it is worth considering. For example, Traub discussed the difference between non-strictness and laziness and proposed *lenient evaluation* [Traub91a]. This could be useful because it may reduce delay-related overhead.

4.4 Inherently Lazy Operations

There are some operations which could be called *inherently lazy* [Kozato92a]. That is, if these operations are programmed naively the program will be written in a demand-driven manner. We claim that lazy languages can describe these operations naturally. In this section, we will describe two examples, namely geometric transformations of pixel images and handling pixel and non-pixel images, of which geometric transformations will be described in detail in Chapter 5 and Chapter 6, and a method to combine pixel and non-pixel images will be implemented in Chapter 7.

4.4.1 Geometric transformations

Geometric transformations describe operations to map one coordinate to another coordinate which redefine the spatial relationship between points in an image. Geometric transformations include simple *affine transformations*, such as translation, scaling, rotation and shearing, and more elaborate ones, such as projective, bilinear and polynomial transformations. It is also termed *image warping* [Wolberg90a].

Applications of geometric transformations are wide-ranging, but geometric correction and distortion may be the two principal factors. For example, *image mosaicing* [Schalkoff89a] is a technique to compose a large image by "patching-up" a number of small images, such as producing an image of the earth from a number of satellite images. In this case, because the images are distorted by the varied orientation of the satellite, they need to be corrected properly. Geometric correction is also used for stereo matching algorithms [Horn86a], and many other applications. Geometric distortion is commonly used in various packages, for example, graphics, desk top publishing and image handling, where images are enlarged, shrunk, rotated, twisted, trimmed, and so forth.

As a simple case, affine transformations are described. An affine transformation preserves parallelism and can be usually expressed as a matrix multiplication on old

coordinates (x,y) to produce new coordinates (X,Y) (Figure 4-2). Using *homogeneous coordinates* [Foley90a] an affine transformation can be written as:

$$\begin{bmatrix} X & Y & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & 1 \end{bmatrix} \qquad \text{(Eq. 4-1)}$$

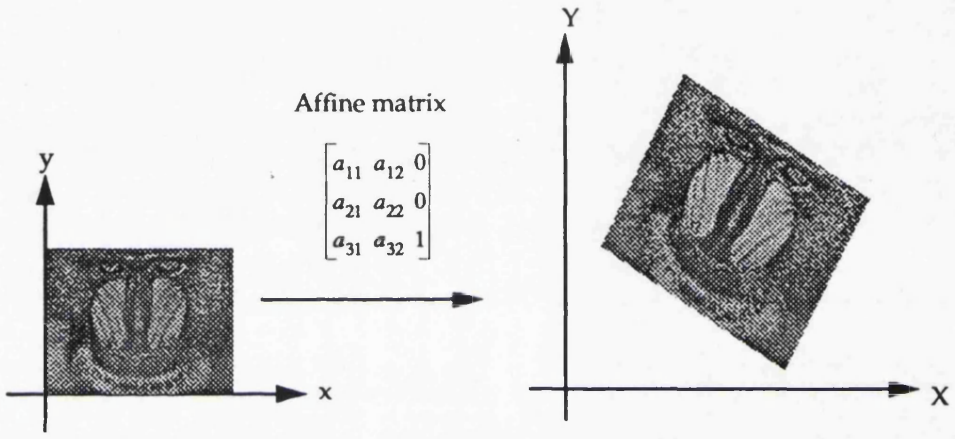


Figure 4-2 An affine transformation

Here, let us consider two problems in affine transformations of pixel images, namely holes & overlaps, and interpolation & quantisation errors:

a. **Forward vs backward mapping**

If the above matrix multiplication is directly applied to input pixels, the transformed image may include holes and overlaps. This is because the output image should be a pixel image, as well, although the transformed pixels may not fall onto the regular grid of the output image. This problem can be avoided if input pixels are considered as square patches that may be transformed into arbitrary quadrilaterals and output pixels are calculated by properly integrating these quadrilaterals. But this calculation is usually complicated and expensive [Wolberg90a].

For this reason, geometric transformations of pixel images are often programmed backwardly, i.e. in a demand-driven manner, using *inverse mapping*. In this respect, we regard geometric transformations as inherently lazy. What is normally done is to calculate an inverse transformation matrix, then scan the *output* image, calculate where

the corresponding pixel comes from in the input image, take the surrounding pixels, and finally calculate the pixel value using appropriate interpolation.

b. Consecutive transformations

When a series of transformations is applied to a pixel image and their intermediate results are stored as pixel images, two problems may arise: (i) each transformation involves computationally expensive interpolation, and (ii) quantisation errors produced in each transformation are accumulated through the consecutive operations. In order to avoid these drawbacks, it is common to calculate the composite matrix first, then process the image using inverse mapping. This, however, is purely for the sake of computing convenience, and also requires extra programming effort. Users would wish to process images as a straightforward sequence of transformations, e.g. rotate the image by 30 degrees, then translate it by (15, 23), then scale it by (1.2, 0.8), and so on.

Using a lazy functional language, the inverse mapping usually required for geometric transformations is provided by the evaluation strategy of the language and explicit programming of inverse mapping is not necessary. Programs can be written straightforwardly, so that the actual pixel calculation is not carried out unless users request the evaluation of pixels.

One example of an operation which forces evaluation of pixel is a display function. The output pixels in a rotated image are calculated only when `output` in the following definition is evaluated:

```
image1 = rotate angle inputImage
output = display parameters image1
```

Evaluation of `image1` may return a certain description of the rotated image, but not the rotated pixel image.

In addition, consecutive transformations are expressed explicitly and straightforwardly as function composition, rather than matrix composition, and involve no intermediate interpolation or quantisation errors. This is because lazy languages do not try to calculate results until the complete transformation is known. Further, consecutive transformations are expressed even when the data they will act upon is not yet available. Consecutive transformations may look like the following code:

```
consecutiveTransform = (rotate angle).(translate displacement)
output = display parameters (consecutiveTransform inputImage)
```

This way of programming can be regarded as more "natural" than imperative programming, because programmers do not have to rearrange the sequence of low-level operations and carefully consider memory usage in order to avoid holes & overlaps, intermediate interpolation and quantisation errors.

4.4.2 Combining pixel images and non-pixel images

We started the discussion with the common assumption that an image is an array of pixels which is usually large (Section 1.1). We are used to this representation very much and take pixels for granted. However, conceptually, an image should be a certain description of objects in the real world, which may be represented as a function of a continuous variable. Pixels are just one representation method in order for an image to be input through an array of sensors, stored in computer memory, or output to digital devices such as a frame buffer and a raster scan display. In other words, pixels are invented because they are convenient to handle images on digital computers, and they have nothing to do with abstract descriptions of objects in the real world.

With regard to describing the real world, lazy functional languages have potential because they are based on mathematical functions, so that there is little difficulty in handling continuous functions. Also, lazy functional languages have potential to be able to handle pixel images and images represented as continuous functions in a unified manner. In the previous subsection (Section 4.4.1), we discussed that lazy functional languages can display an image which has been arbitrarily transformed. To implement this technique, there should be a mechanism to get a display pixel value by interpolating arbitrarily placed pixels. This would also be seen as a look-up function of a pixel image using real number indexing. The type signature of such a function may look like:

```
lookUp::coord->image *->*
```

where the first argument is the type to designate a coordinate. Using this function, the display function will be implemented as a simple iteration.

With regard to a continuous function, this kind of look-up operation should be much easier because it is exactly the evaluation of the function itself. Let us call an image expressed

as a function a *function image* of the type `fImage`, which can be expressed as:

```
fImage::coord->*
```

And the function to look-up a pixel from a function image should have the type signature:

```
lookUpFImage::coord->fImage->*
```

Like the pixel images' case described above, the display function of a function image will be implemented as simple iteration of the `lookUpFImage` function. If we call it `displayFImage`, it can be used in the same way as the display function for a pixel image described in Section 4.4.1. For example, evaluating the expression:

```
displayFImage parameters a_function_image
```

will produce a list of lists of pixels.

In this way, a pixel image and a function image act in the same way. In non-lazy languages or imperative languages, if programmers wish to implement such a facility and program as such, it may be possible. But using lazy functional languages it should be easier, because, as already discussed, lazy languages carry out evaluation only when a result is required. Therefore, it is not the programmers' concern to prepare data in advance for possible and probably never occurring requests.

Before considering applications of combining pixel and non-pixel images, let us consider non-pixel images alone. Non-pixel images correspond to computer graphics in a broad sense, because objects are modelled not by pixels but by geometric data, e.g. a collection of polygons, and such geometries are transformed, lighting is calculated, a viewport is set on the geometric objects, and ultimately, they are scan-converted to pixels and displayed [Foley90a]. So, scan conversion is somewhat akin to the display function described so far, because the operation is driven by the demand to produce output pixels. A few approaches towards computer graphics using functional languages have been reported [Henderson82a, Parsons87a, Lakshminarasimhan89a, Checkland91a], but none discussed laziness of this nature although Parsons and Checkland used lazy functional languages. Parsons, in particular, implemented several scan conversion algorithms using Miranda, but his approach is that the function takes a graphical object and a raster, and updates the raster. Thus, it is not based on the kind of laziness which we have been discussing.

The first example of combining pixel and non-pixel images is image synthesis using non-pixel images. In Section 3.3.4, we implemented `condImage` to carry out the spatial conditional operation, where all the arguments had to be pixel images (See Figure 3-3). But now it is not a necessary constraint. For example, an alternative conditional operation can be implemented, where the first argument, i.e. a boolean image, is a function image, such as *"if a coordinate is inside the specified circle, then True, otherwise False"*. This will produce a synthesised image, e.g., as shown in Figure 4-3.

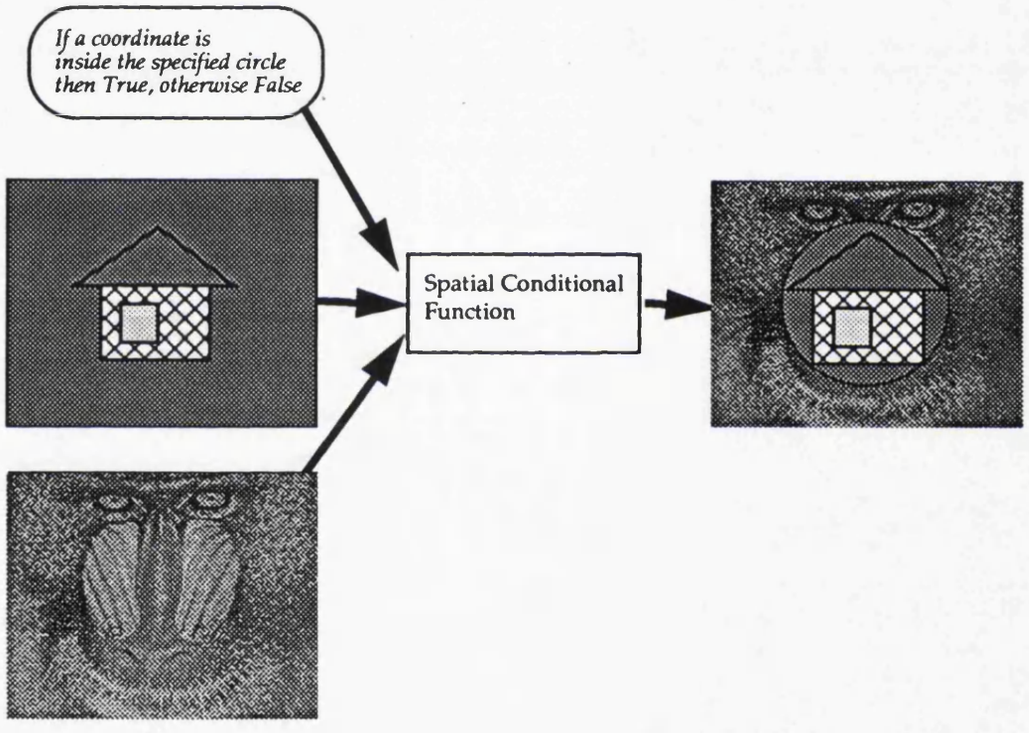


Figure 4-3 An image conditional function using a function image

Another application is object recognition by fitting models to photometry [Suetens92a]. The essence of the technique is to quantify similarities between models, or templates, and an image, and the simplest method is to use correlation. Models may be represented in pixels, or may be in some geometric description as in CAD models for machinery parts. Correlation is very similar to convolution except that a mask, or a template in this case, is not turned over (as was implemented in Section 3.5.5). The components of correlation are translation and pointwise operations, both of which can be implemented easily on function images, as well.

Yet another application is image filtering in the frequency domain. We have already

implemented various kinds of filtering in the spatial domain using local neighbourhood operations (Section 3.5), but filtering is also possible in the frequency domain. This can be done by transforming an image into the frequency domain, e.g. by Fourier transformation, applying a filter in a pointwise manner, then converting it back to the spatial domain. Filters in the frequency domain are typically given as equations, such as the ideal filter and Butterworth filter [Gonzalez87a]. Therefore, operations between a pixel image and a function image will be convenient.

As we have seen, there are various applications of combining pixel and non-pixel images. Some of the examples will be implemented in Chapter 7

4.5 Summary

In this chapter we have discussed laziness not only in the context of functional language implementations but also in terms of image processing applications. As image processing involves very expensive operations, lazy languages may provide efficiency by eliminating unnecessary operations. In addition, laziness may contribute modularity of programs. We have presented two particular areas which lazy functional languages would suit: geometric transformations and combining pixel and non-pixel images. These applications are implemented in the following three chapters.

Chapter 5: Affine Transformations

5.1 Introduction

As discussed in the previous chapter, geometric transformations are one class of operations which lazy functional languages particularly suit because they are inherently lazy. This chapter describes an implementation of affine transformations, i.e. any combination of translation, scaling and rotation, written in Miranda. Using lists as the basic representation of an image, the lesson we learned in Chapter 3 is again followed, i.e. define a 1D structure and operations first, then compose 2D operations using higher-order functions. Also, the data structures are designed to separate the list part and the other parameter part using tuples. In Chapter 3, we introduced only an origin as the other parameter since the purpose was to implement translations. Whereas here, additional parameters will be attached as we are implementing scaling and rotation.

Programs are written to utilise the laziness of the language. The following is a summary of the benefits of using lazy languages to express geometric transformations as discussed in Chapter 4:

- Unless the user specifically asks an image to be displayed, no pixel operations are carried out.
- And only the required parts are calculated.
- Program description is forward mapping in a straightforward manner.
- Consecutive transformations are expressed as straightforward function composition and do not accumulate quantisation errors.

It will be shown that pursuing these benefits of laziness leads to a novel way of image processing programming.

5.2 Design Overview

This section gives an overview of what is implemented and how it materialises the above benefits of laziness in concrete terms.

Let us assume a series of transformations illustrated in Figure 5-1, that is, *"we would like to translate an image by (-0.5, 1), rotate it by 45°, then scale it by (2, 1.5), and display it as the pixel*

image whose origin is (0,0) and size 5×5".

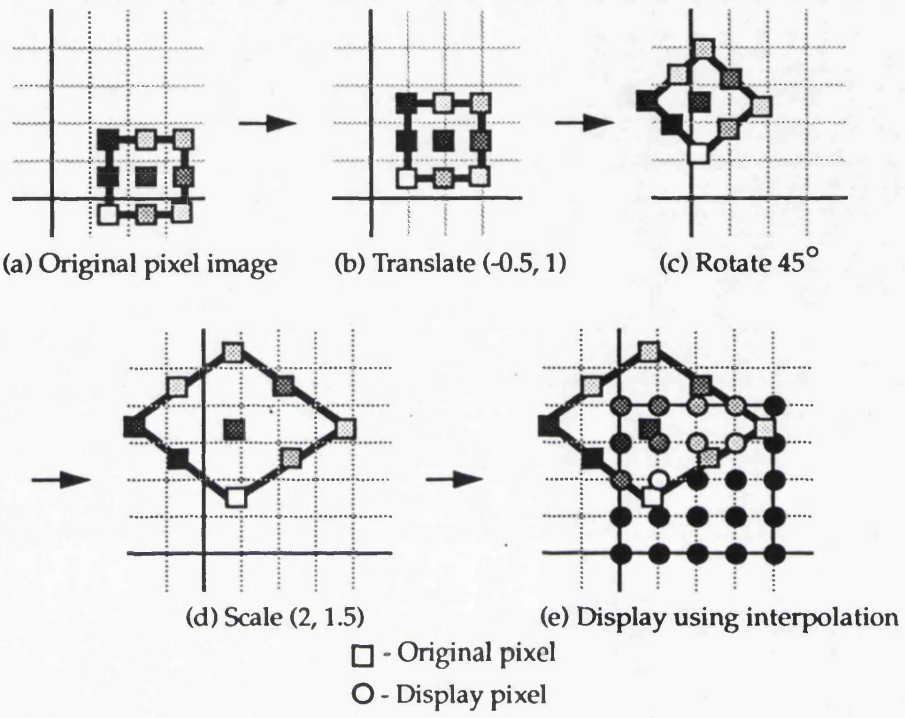


Figure 5-1 An example of affine transformations

If this series of transformations was expressed in an imperative language, the program would be far from the above description. That is, as already discussed in Section 4.4.1, in order to avoid holes & overlaps, intermediate interpolation, and accumulation of quantisation errors, the transformation matrix for the whole process should be composed from each transformation matrix first, then the composite matrix should be inverted, and lastly each pixel should be calculated using inverse mapping and an appropriate interpolation method. An imperative version of affine transformations is shown later (Figure 5-6) with a discussion of the difference between functional and imperative styles (Section 5.5.3).

In Miranda, this series of transformations can be written very directly as[1]:

```
transform = scale 2 1.5 . rotate (pi/4) . trans (-0.5) 1
```

This is just a function composition to express the transformations in a straightforward fashion,

1. `pi` is π in Miranda.

and no image calculation is involved. If this function is applied to an image, i.e. evaluating the expression:

```
transform image
```

produces the correct image data structure, but does not access or modify the pixel values themselves. This is due to the image representation adopted, in which the list part and the other parameters to describe the "shape" of the image are separated. (Please recall the image translation in Section 3.4 on page 44, where a translation did not affect the list part.)

A demand for calculating a display pixel is produced only by the user's desire to display the transformed image. The system automatically propagates the demand through to input and calculates the output pixel values. For example, given an appropriate definition of the function `display` which takes as arguments the origin and the size of the display (expressed as x and y coordinates), an interpolation function, and an image, evaluating the expression:

```
display 0 0 5 5 linear (transform image)
```

gives a 2D list of pixels to be displayed, shown as an array of O's in Figure 5-1 (e), which may be transferred to a display device or stored as a 2D image file. It should be noted that only required display pixels, 5×5=25 in this example, are calculated. As the transformed image only partially covers the display grid, a requested pixel that falls outside the transformed image is given the background value. In addition, the interpolate function, `linear` in this case to carry out bilinear interpolation, is parameterised, so that the other interpolation function can be slotted in.

In this way, the *"output = function(inputs)"* style of a functional language naturally leads to the same effect as the backward mapping required for geometric transformations, and avoids unnecessary operations and accumulation of quantisation errors.

5.3 Translation and Scaling

First, we will consider translation and scaling only. Because for these transformations operations can be completely separable for x and y directions, the principal method developed in Chapter 3 can be reused with some extension. In order to implement rotation, a little more thought is necessary which will be described in Section 5.4.

5.3.1 Image representation

The data structures of an image are defined with the same principle as, but a little extension to, the ones used in the "with an origin" version (Section 3.2.2).

```
interval1 *    == (num,num,[*]) || (origin,length,pixel_list)
row1 *         == interval1 *
image1 *       == interval1 (row1 *)
```

The type of a pixel is parameterised and the triple `interval1` is used both for a row and an image. An interval is a straight line segment on which list elements, i.e. either pixels or rows, are evenly distributed, and its members are an origin, length and a list. The first member refers to the starting position of a list, i.e. either the x coordinate of a pixel list in the case of a row or the y coordinate where a list of rows starts in the case of an image, both in world coordinates. The second member represents the length of the interval (not the length of the list). This length parameter specifies the scaling of the interval.

Using the interval representation, it is possible to represent non-rectangular images (See Figure 3-2). Also, if an operation is defined on an interval, it may be used for a row or an image without change, which improves modularity. Further, transformations do not affect the list part, but only change its origin and length which describe the "shape" of the interval. Thus, no pixel operations are involved and it is efficient.

5.3.2 Transformations

Because the origin and the length of an interval are separated from the list part, translation is just a shift of the origin, and scaling is a shift of the origin plus scaling of the length since scaling is done about the world origin. An implementation of translation and scaling is as follows:

```
transRow1,scaleRow1::num->row1 *->row1 *
transRow1 d (o,l,p) = (d+o, l, p)
scaleRow1 sc (o,l,p) = (sc*o, sc*l, p)

transImage1,scaleImage1::num->num->image1 *->image1 *
transImage1 dx dy (oy,ly,r) = transRow1 dy (oy,ly,map (transRow1 dx) r)
scaleImage1 sx sy (oy,ly,r) = scaleRow1 sy (oy,ly,map (scaleRow1 sx) r)
```

At this stage no pixel operation is carried out since translation and scaling only affect the parameters outside the pixels. It is also notable that 2D operations are constructed entirely

in terms of 1D operations.

5.3.3 Displaying an image

An image is displayed lazily, that is, a display function is designed so that demand for an output pixel drives evaluation. The whole process is somewhat akin to the *stream model* often used as a model of I/O to functional programs [Hudak88a]. In the stream model, a functional program takes a (possibly infinite) stream of data as input and delivers a (possibly infinite) stream of data as output, and this is done lazily, i.e. a request for output drives the functional program to read its input.

As constructing 2D operations can be done at a higher level and entirely in terms of 1D operations, a method to display a 1D image is described first.

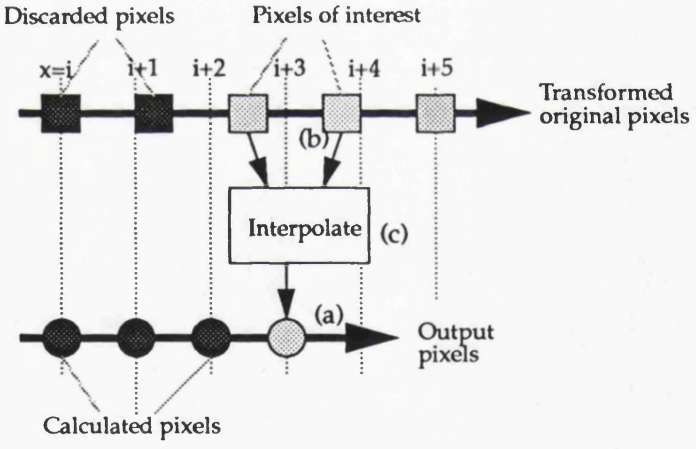


Figure 5-2 Displaying pixels through interpolation

We use the display function as the demander of pixel evaluation, i.e. until the user wishes to display an image, pixel operations are not carried out. The process for displaying can be broken down into the following three sub-processes and illustrated in Figure 5-2:

a. Scanning the output image along the display grid. An interval between two display positions is 1.

b. Pulling out necessary pixels from the original image in order to calculate an output pixel. Due to transformations, the original pixels may not be aligned with the integer grid.

c. Interpolation which calculates the value of the output pixel from the original pixels.

Because both input and output images are represented as lists, the overall process can be described as synchronisation of an output pixel list and an original pixel list through an interpolation function, and lazy evaluation does this automatically!

At first, a new intermediate data structure is introduced. The interval representation is convenient for transformations, but in order to get the actual pixel values, information about the position of each element will be needed. In the code this data structure is called `plist` which is *a list of pairs of (position, value)*, in which *value* is either a pixel or a row. Conversion from `interval` to `plist` is straightforward as follows:

```
plist * == [(num,*)]

intervalToPlist::interval1 *->plist *
intervalToPlist (o,1,[a]) = [(o,a)]
intervalToPlist (o,1,p)   = zip2 posList p
                            where posList = map ((+o).(*period)) [0..]
                                  period  = 1/(#p-1)
```

In lazy languages, only the necessary portions are converted to a plist, not the entire image. This means that transformed original pixels shown as □'s in Figure 5-2 are not calculated until those pixels are required.

The first function to consider is sub-process (a.), i.e. the driving force along the output display grid. The `disp` function defined below does this job:

```
disp::num->num->num->num->(num->**->plist *->**)->**->plist *->[**]
disp x n hi lo ifn bg pl
   = [] , if n<=0
   = bg:disp (x+1) (n-1) hi lo ifn bg pl , if ~(lo<=x<=hi)
   = ifn x bg pl:disp (x+1) (n-1) hi lo ifn bg (dropUpto (x+1) pl)
      , otherwise
```

The first and second arguments specify the display grid, i.e. the starting position and the number of pixels to be displayed. The former is incremented and the latter decremented through recursive calls. The arguments `hi` and `lo` define the range of the original image and stay throughout the calls. `ifn` is an interpolate function described later, into which currently nearest neighbour and linear interpolation can be slotted. Because the `disp` function is designed to be polymorphic, i.e. be able to display either a row or an image, it takes a background value, `bg`. This is because the background value for displaying a row is a pixel,

whereas the value for displaying an image is a row filled with the background pixels.

The next function to consider is sub-process (b.), i.e. the key function to synchronise the output and the original pixel lists. In order to interpolate pixels, an output pixel of interest must be between the two adjacent original pixels. The `dropUpto` function removes elements from the front until this criterion is satisfied. The arguments to the `dropUpto` function are a position and a plist:

```
dropUpto x [] = []
dropUpto x [x1] = [x1]
dropUpto x (x1:x2:xs)
   = (x1:x2:xs) , if isBetween x (positionOf x1) (positionOf x2)
   = dropUpto x (x2:xs) , otherwise

isBetween a b c = (b<=a<=c) \/ (b>=a>=c)
positionOf (p,v) = p
```

And lastly, interpolation, i.e. (c.), is defined. An implementation of nearest neighbour interpolation is given below. The `nearest` function takes a position of the display grid `x`, the background value `bg`, and a plist. It compares `x` with the positions of the first two elements in the plist and returns the value of the nearer.

```
nearest::num->*->plist *->*
nearest x bg [] = bg
nearest x bg [a] = valueOf a
nearest x bg (a1:a2:as)
   = valueOf a1 , if dist x (positionOf a1) <= dist x (positionOf a2)
   = valueOf a2 , otherwise

dist a b = abs (a-b)
valueOf (p,v) = v
```

The nearest neighbour interpolation does not involve the value elements in (position, value) pairs, so the function `nearest` can take a plist for either a row or an image.

As for linear interpolation, however, it requires the value elements. This means that the types of the functions for interpolating a row and an image differ, and separate functions are hence necessary. `li1` is the function to interpolate pixels, which takes the position and two surrounding elements of a plist and does ordinary linear interpolation. `li2` interpolates a plist of lists of pixels, which is implemented as a mapping of the function `li1` on each row. The function `linear0` is a higher-order function that makes the top-level type signatures

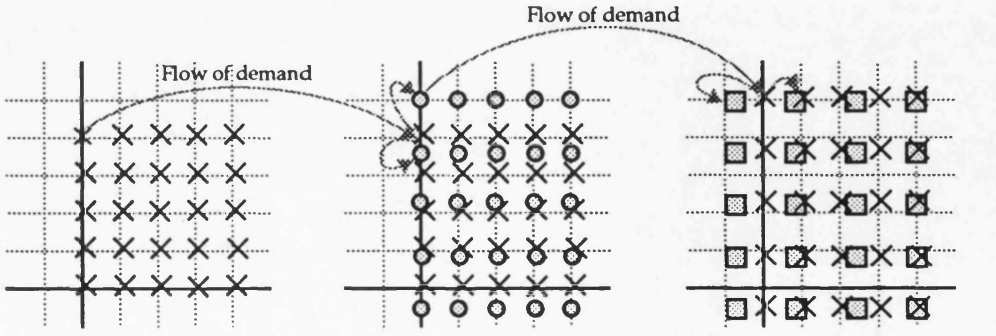
compatible. Hence:

```
linear1 = linear0 li1
linear2 = linear0 li2

linear0 fn x bg []          = bg
linear0 fn x bg [a]         = bg
linear0 fn x bg (a1:a2:as) = fn x a1 a2

li1 x (x1,v1) (x2,v2) = v1+(v2-v1)*(x-x1)/(x2-x1) , if x2 ~= x1
                      = bgp                        , otherwise

li2 y (y1,vs1) (y2,vs2) = map2 (li1 y) ys1 ys2
                          where ys1 = zip2 (repeat y1) vs1
                                ys2 = zip2 (repeat y2) vs2
bgp = 0
```

In this way all the necessary functions have been implemented.

Using the basic `disp` function, we can compose a 2D display function. In the following, `dispRow1` and `dispImage1` display a row and an image respectively:

```
dispRow1 x n ifn (o,l,p)
   = disp x n (o+l) o ifn bgp (intervalToPlist (o,l,p))

dispImage1 x y xn yn ifn1 ifn2 (o,l,p)
   = disp y yn (o+l) o ifn2 bg (intervalToPlist im1)
     where bg  = rep xn bgp
           im1 = (o,l,map (dispRow1 x xn ifn1) p)
```



(a) Pixels required at × (b) Pixels required at × and pixels aligned in x direction ○ (c) Pixels required at × and transformed pixels □

Figure 5-3 Interpolation in two steps

How an image is displayed is shown in Figure 5-3. Transformed pixels (c) are aligned in the x direction (b), and then y (a), using appropriate interpolate functions. But this is done in a demand-driven manner. So, a better explanation would be: in order to display a pixel, surrounding pixels are required for interpolation. In order to interpolate the pixel in the y direction, pixels are needed which have been interpolated in the x direction. A demand to produce a display pixel is propagated from (a)→(b)→(c), while a pixel is calculated from (c)→(b)→(a). In addition, the positions of the transformed pixels in (c) are not calculated unless they are required. In this way, laziness in Miranda naturally leads to the backward mapping required for geometric transformations without programming backwards.

5.3.4 Discussions

For translation and scaling, operations for x and y directions are completely separable. Therefore, the interval representation can be used for an image, as well as a row. For rotation, this is no longer true, as the transformations will interfere in each direction. Certain modifications are necessary, which will be described in the next section.

5.4 Rotation, Translation and Scaling

5.4.1 Image representation

A vector style representation for a row is introduced in order to implement rotation. A row is a straight line segment on which pixels are evenly distributed and is represented as a triple of an origin (or a start position), length and a pixel list, where the origin is a pair of x and y coordinates and the length is a pair of measurements in x and y directions. Since all the parameters to describe the "shape" are included in the definition of a row, an image is just a list of rows. The definitions are as follows:

```
row2 *    == ((num,num),(num,num),[*])
image2 * == [row2 *]
```

In addition, the plist structure defined in Section 5.3.3, i.e. a list of (position, value) pairs, is used. In the following definition, `rowToPlist` takes the x elements of a row and converts it to a plist, and `rowToPlistY` does the same operation for the y elements:

```
rowToPlist::row2 *->plist *
rowToPlist ((ox,oy),l,[a])      = [(ox,a)]
rowToPlist ((ox,oy),(lx,ly),p)
   = zip2 posList p
     where posList = map ((+ox).(*period)) [0..]
           period  = lx/(#p-1)

rowToPlistY ((ox,oy),(lx,ly),p) = rowToPlist ((oy,ox),(ly,lx),p)
```

Basically, there is not much difference between the data structures defined here and the ones previously used. If only the x or y elements of the origin and the length of `row2` are taken, it is identical to `interval1` defined on page 81. Thus, it can be seen that most functions we need to implement will be not very different from the ones defined so far.

5.4.2 Transformations

Transformations only affect the origin and length parts, but not the list part. As an image is just a list of rows, 2D operations can be implemented as ordinary function mapping:

```
transRow2 dx dy (o,l,p)    = (pair1 ((+dx),(+dy)) o,l,p)

scaleRow2 scx scy (o,l,p) = (scl o,scl l,p)
                            where scl = pair1 ((*scx),(*scy))

rotateRow2 th (o,l,p)      = (rot o,rot l,p)
                            where rot = pair2 (rotx th,roty th)
                                  rotx th x y = x*(cos th)-y*(sin th)
                                  roty th x y = x*(sin th)+y*(cos th)

pair1 (fn1,fn2) (x,y) = (fn1 x,fn2 y)
pair2 (fn1,fn2) (x,y) = (fn1 x y,fn2 x y)

transImage2 dx dy    = map (transRow2 dx dy)
scaleImage2 scx scy = map (scaleRow2 scx scy)
rotateImage2 th      = map (rotateRow2 th)
```

By combining these functions, an arbitrary affine transformation can be described as a function composition.

5.4.3 Displaying an image

As for the basic display function (`disp`) to produce a list of pixels from a plist, the one defined in Section 5.3.3 (page 83) can be used unchanged. All that is needed to display a rotated image

is to select appropriate elements and rearrange function arguments. The following is the `disp` function just as a reminder:

```
disp::num->num->num->num->(num->**->plist *->**)->**->plist *->[**]
disp x n hi lo ifn bg pl
   = [] , if n<=0
   = bg:disp (x+1) (n-1) hi lo ifn bg pl , if ~(lo<=x<=hi)
   = ifn x bg pl:disp (x+1) (n-1) hi lo ifn bg (dropUpto (x+1) pl)
      , otherwise
```

`dropUpto` used in this definition is identical to the previous definition (page 84).

The procedure to interpolate and display a transformed image is in two steps and illustrated in Figure 5-4. In (c), thick lines show transformed rows. In the first pass, the `disp` function is applied only to the x elements of rows, which yields pixels on the original rows but aligned with the grid of display in the x direction (⊙'s in Figure 5-4 (b)). If these elements are taken and viewed as rows in the y direction (thick gray lines in (b)), then the situation is exactly the same as the x direction. The `disp` function is usable without any changes, and this results in pixels aligned with the grid in both directions.

The actual execution, however, is carried out backwards. As shown by the arcs in Figure 5-4, a demand is produced by a request to display a pixel (Figure 5-4 (a)). The demand is then propagated and produces demands for two surrounding pixels in the y direction (see (b)). These demands are propagated to (c) and produces demands for surrounding pixels on transformed rows, which further produces requests for calculation of the transformed rows. Since rows are represented as lists, these operations are carried out not randomly, but sequentially along the display grid.

Figure 5-4 may look somewhat like a well-known technique called *two-pass transformations* [Catmull80a, Tanaka86a], in which a 2D transformation, such as a rotation, is decomposed into two 1D transformations. However, the two-pass algorithms divide a transformation into two steps with an intermediate pixel image being produced. Whereas in lazy languages, transformation and interpolation are carried out based on pixel-by-pixel requests. Therefore, no image is produced between the two interpolation steps.

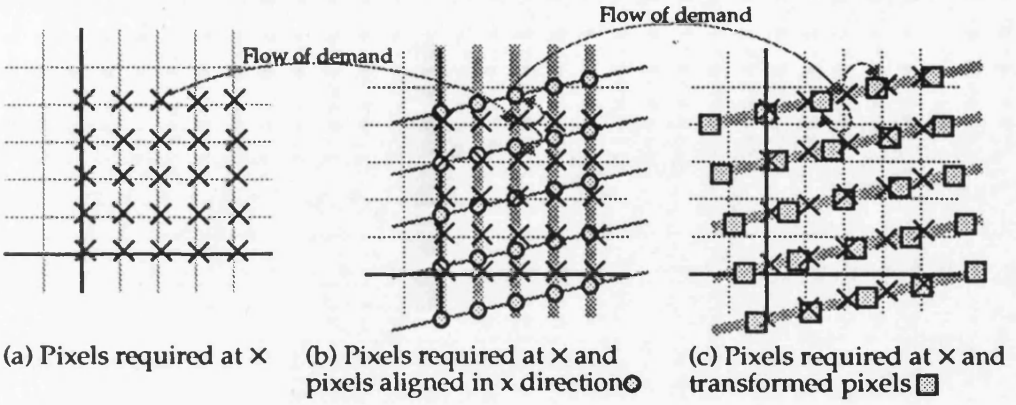


Figure 5-4 Displaying rotated pixels

From a programmer's point of view the above procedure is implemented automatically by the laziness of the language. Program description is direct and straightforward as follows:

```
dispPlist::num->num->(num->*->plist *->*)->*->plist *->[*]
dispPlist x n ifn bg pl = disp x n hi lo ifn bg pl
                          where hi = positionOf (last pl)
                                lo = positionOf (hd pl)

dispRow2::num->num->(num->*->plist *->*)->*->row2 *->row2 *
dispRow2 x n ifn bg ((ox,oy),(lx,ly),p)
   = ((x,ly/lx*(x-ox)+oy),(n-1,ly/lx*(n-1)),dsp)
     where
     dsp = disp x n (ox+lx) ox ifn bg (rowToPlist ((ox,oy),(lx,ly),p))

dispImage2::num->num->num->num->(num->*->plist *->*)
            ->*->image2 *->[[*]]
dispImage2 x y xn yn ifn bg im
   = transpose (map (dispPlist y yn ifn bg) (transpose im1))
     where im1 = map (rowToPlistY.dispRow2 x xn ifn bg) im
```

In `dispImage2`, the input image `im` is interpolated to align with the integer grid in the x direction by mapping `dispRow2` on rows, then these rows are converted into plists in the y direction using `rowToPlistY`. These plists are transposed for viewing as plists in the x direction and interpolated using `dispPlist`, then transposed back to the y direction. The `transpose` function defined in the standard environment does these transpose operations. All these high level functions are defined using the basic `disp` function.

With regard to the functions for interpolation, the nearest neighbour interpolation

(`nearest`) defined on page 84 and the linear interpolation (`linear1`) on page 85 can be used. There is no need to prepare separate interpolate functions for pixels and rows, as only one kind of interpolation of pixels is carried out in both directions.

5.5 Discussions

5.5.1 Limitation of the programs

We have so far used lists as the basic representation of an image. This means that the algorithms make the assumption that elements (pixels or rows) are aligned in ascending order. In addition, as shown in Figure 5-2, a display pixel list and an original pixel list must extend in the same direction. So, if an image is rotated more than ±90° the functions need to be modified to traverse the original list in the opposite direction. Also, there may be a precision problem if the rotation angle exceeds ±45° and approaches ±90°, since the slope of the original row becomes too steep to interpolate pixels properly using x elements.

As for other geometric transformations the code needs to be modified. For example, in order to implement perspective transformations, in which linearity, but not parallelism and spacing, of lines is preserved, transformation functions to rotate an image about x and y axes may need to be added. Also the function to convert a row to a plist should be modified to implement appropriate forward mapping. But after plists have been calculated, the functions for display can stay unchanged since the relation between the two lists (Figure 5-2) is the same even if the original pixels are not evenly distributed.

More complicated transformations in lazy functional languages are subject to further research, since pixels are not aligned on straight lines. However, the principle of separating parameters to describe the "shape" may be useful, though these parameters may represent complicated equations. Above all, laziness would become more useful when complicated transformations are implemented, because pixels are evaluated only when necessary and the complicated transformations may involve expensive operations.

5.5.2 Representation and efficiency

It has been discussed in Chapter 4 that lazy languages should be efficient because only required data is calculated. Using lists as the basic data structure means they have to be traversed sequentially from the top. If a small number of pixels near the beginning are

required (Figure 5-5 (a)), lazy languages should be efficient as the rest of the list is never visited or evaluated. However, if the pixels near the end are required (Figure 5-5 (b)), although the pixels on the original row are not evaluated unless requested (See the list selection example on page 62), at least the spine of the list will have to be traversed, i.e. elements will be visited from the head sequentially. If an image is large, hence the list is long, this traversal may be costly. Also, if pixels are required only sparsely (Figure 5-5 (c)), e.g. because the original image has been reduced, the skeleton of the list will be sequentially traversed although evaluation of the elements will take place only at required positions.

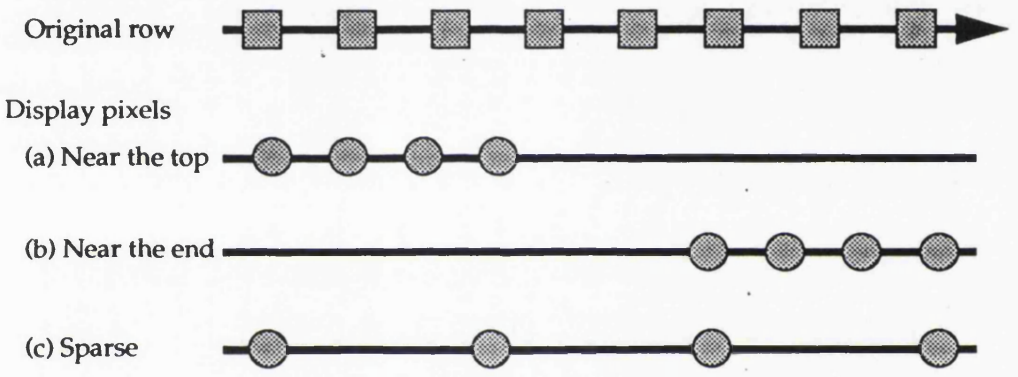


Figure 5-5 Spatial relationship between original and display pixels

In order to deal with these cases efficiently, hierarchical data structures such as trees and quadtrees may have higher potential since they are lazy about unwanted areas. These data structures are discussed in the next chapter.

5.5.3 Imperative vs lazy functional style

We have demonstrated how geometric transformations can be programmed straightforwardly in a lazy functional language. In contrast, in a conventional imperative language, programmers have to do considerably more work to provide composition of geometric transformations in a modular fashion. Figure 5-6 gives an example of an affine transformation written in a language with C-like syntax[2] to show how it would look. In order to avoid holes and overlaps, backward mapping is implemented by inverting a transformation matrix and scanning the output raster. When a series of transformations is required, this function will be

2. It should be noted that this is a toy-code to show the skeleton and much work is required for real use.

repeatedly applied. Each application will require interpolation and accumulate quantisation errors. So, a composite matrix should be calculated before this function is applied. From this observation, although the problem of accumulating errors due to the application of successive transformations has to be tackled in all programming languages, it can be said that the lazy functional languages provide a neater solution.

However, with regard to the style of programming, it is difficult to conclude that the lazy functional version is "more natural". The principal reason would be that we are used to viewing a pixel image as an array of pixels, and handling an image as lazy lists is quite an unusual way. On the other hand, if Miranda had had arrays it would have been very difficult to come up with the method we have developed in this chapter. Haskell does support arrays. Use of arrays in lazy functional languages to write image processing programs would be an interesting subject for further research in terms of efficiency and ease of reading and writing.

```
/* Affine using bi-linear interpolation */
float input_image[XS1][YS1],output_image[XS2][YS2];
int i, j, ui, vi;
float u, v, val1, val2, trans[3][3], inverse[3][3];

/* Calculate inverse transformation matrix */
invert(trans, inverse);

/* Double loop to scan the output image */
for (i=0; i<XS2; i++) {
  for (j=0; j<YS2; j++) {
     /* Calculate the corresponding position
     in the input image */
     u=i*inverse[0][0]+j*inverse[0][1]+inverse[0][2];
     v=i*inverse[1][0]+j*inverse[1][1]+inverse[1][2];
     /* If the pixel is out-of-bounds, assign the background value */
     if (u<0 || u>XS1 || v<0 || v>YS1)
        output_image[i][j]=BACKGROUND;
     else.{
        /* Bi-linear interpolation */
        ui=(int)u;
        vi=(int)v;
        val1=(1-u+ui)*input_image[ui][vi]
           +(u-ui)*input_image[ui+1][vi];
        val2=(1-u+ui)*input_image[ui][vi+1]
           +(u-ui)*input_image[ui+1][vi+1];
        output_image[i][j]=val1*(1-v+vi)+val2*(v-vi);
     }
  }
}
```

Figure 5-6 An affine transformation in C

5.6 Summary

In this chapter we have presented an implementation of affiňe transformations in Miranda. Validity of the discussion in Chapter 4 has been proved with the actual code. The main contributor is laziness embedded in the language, and it has been shown that the program description is straightforward and modular by positively utilising laziness in constructing the algorithms. Also, the code eliminates unnecessary calculations which are often very expensive in image processing. Some drawbacks have also been pointed out. The principal difficulty lies in the basic representation of an image, i.e. the use of lists, which must be traversed from the beginning. In the next chapter we will tackle this problem by implementing hierarchical data structures which are lazy about unwanted areas.

Chapter 6: Using Hierarchical Data Structures

6.1 Introduction

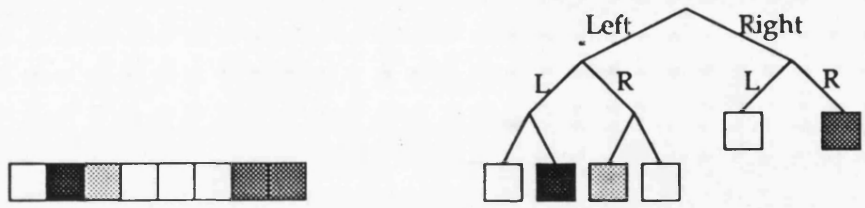
In the previous chapter we implemented affine transformations and display using lists as the basic image representation. In this chapter we present the same algorithms employing hierarchical data structures instead. Basically, two implementations are presented: the first one uses a binary tree to represent a row and composes a 2D image as a binary tree of binary trees, which is an extension to the method we have used so far, i.e. define 1D operations first, then compose 2D operations using higher-order functions. We will demonstrate that this principle works for hierarchical data structures, as well, and that programming is fairly simple and easy. The second implementation uses a quadtree as the fundamental data structure to represent an image. A quadtree has an advantage in terms of space efficiency over a binary tree of binary trees, but it is a 2D structure and functions should be implemented directly for 2D. Using these data structures, it will be shown that lazy evaluation efficiently implements the display functionality where an image is displayed only at required region and resolution.

6.2 An Overview of Hierarchical Data Structures

This section gives a brief introduction to hierarchical data structures. For more detailed descriptions and overall surveys, readers may refer to [Samet84a, Samet88a, Samet90a].

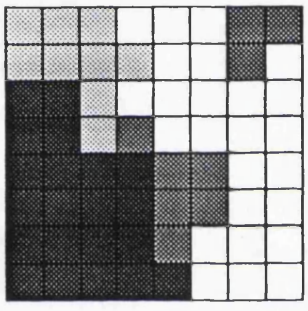
Using hierarchical data structures including binary trees, quadtrees and octrees to represent 1D, 2D and 3D data respectively is an important technique in image processing and computer graphics. The fundamental concept is to recursively subdivide the space, regardless of the dimension, until all the subspaces have a "simple" description such as a uniform data value. Figure 6-1 shows an example of a row and its corresponding binary tree, and an image and its quadtree.

In the tree representations the *root node* corresponds to the entire data, from which subdivision starts. Each child of a node represents an interval, quadrant, or octant in a binary tree, quadtree, or octree, respectively. *Tree condensation* describes an operation to replace a node with children of a uniform value, called a *redundant node* [Hunter79a], with a *leaf node* of that value for which no subdivision is necessary.

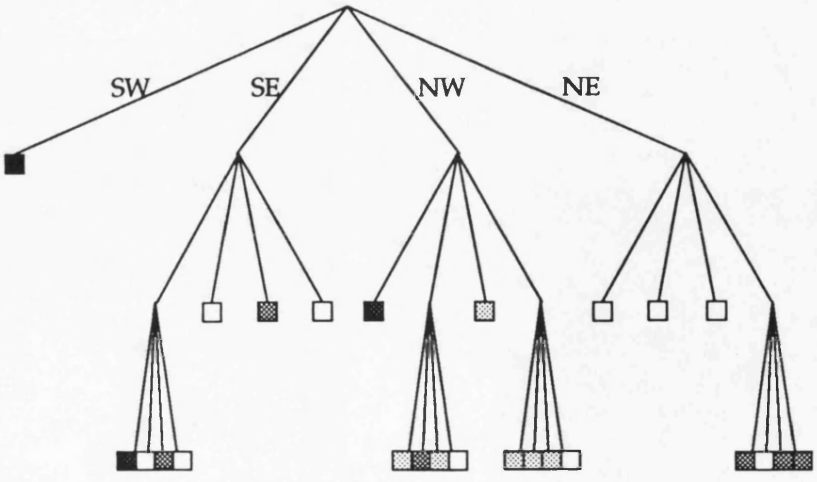


(a) A row

(b) Its binary tree



(c) An image



(d) Its quadtree

Figure 6-1 Example binary tree and quadtree

One of the primary advantages of using such hierarchical data structures is the data compression effect. By tree condensation, if an image contains large homogeneous areas the number of nodes will be reduced considerably. Another advantage is the variable resolution aspect, i.e., as you go down the hierarchy the resolution becomes finer and finer. This is particularly useful when the scene is more complex than the required resolution because it is not necessary to traverse the whole data. It is also useful when transmitting an image with progressive encoding, where a rough sketch of an image is sent first, followed by more detailed descriptions to make the image more and more precise [Knowlton80a, Wallace91a].

With regard to the implementations of hierarchical data structures, the most obvious one is to use pointers. Also, various pointerless representations have been proposed, such as a *linear quadtree* [Gargantini82a] which carries leaf nodes encoded as location codes, and *DF-expression* [Kawaguchi80a] which is based on preordered traversal of nodes. The primary purpose of using these pointerless representations is further reduction of memory usage.

6.3 Hierarchical Data Structures and Lazy Evaluation

The design goal of this chapter is exactly the same as the one described in the previous chapter (See Section 5.2). That is, affine transformations and display of an image through interpolation; where transformations are implemented as separate functions, a series of transformations is expressed as a function composition and does not involve costly interpolation or quantisation errors, and without an explicit request for displaying a transformed image no pixel calculations are carried out.

Linear transformations of images represented by hierarchical data structures have been proposed by Hunter and Steiglitz [Hunter79b] for quadtrees and by Meagher [Meagher82a] for octrees. Here the transformations are performed on a hierarchical data structure and produce a hierarchical data structure as a result. Our transformation does not change the data structure itself, rather it only changes the parameters which describe the "shape" of the tree. When a display function is applied to an image represented by a hierarchical data structure, a 2D list of display pixels is produced. This principle is very similar to the implementation presented in the previous chapter.

In the previous implementation using lists, some drawbacks were pointed out (Section 5.5.1 and Section 5.5.2). That is, although lazy evaluation eliminates unnecessary and

expensive operations, such as evaluating unnecessary list elements, the list must still be traversed sequentially from the top in order to search for required pixels. This could be expensive if the list is long and required pixels are near to the end of the list, or pixels are required only sparsely. Also, there is a restriction that a display pixel list and a transformed original pixel list must extend in the same direction.

The aim of using hierarchical data structures is to overcome these drawbacks and can be summarised as follows:

- The implementation should be more efficient when pixels are required only at a reduced resolution, because it is not necessary to traverse trees down to the pixel level.
- Hierarchical data structures generally allow more efficient random access than linear lists and access speed is normally independent of the position in an image. Thus, (i) there will be less restriction about the range of geometric transformations, and (ii) even when pixels are required from the rear, it will not slow down its access speed.

Related to the second point, coding may be easier compared with the one using lazy lists (implemented in Section 5.3.3) because, using hierarchical data structures, random access allows an image to be treated more like arrays, with which most image processing programmers are familiar.

6.4 A Binary Tree of Binary Trees

6.4.1 Relation with other data structures

The first attempt is to use a binary tree of binary trees of pixels, or simply a tree of trees, as a basic representation of an image. The motivation for not implementing quadtrees straight away is to investigate whether the method we have developed, i.e. composing higher-dimensional operations from lower ones, can be applied or not. In general, 1D data structures and operations are easier to implement and debug and higher-order functions should make construction of higher-order operations straightforward.

As far as space efficiency is concerned, a tree of trees requires more storage than a quadtree. A simple calculation shows that, for representing NxN pixels using pointer-based representations, a tree of tree requires at most N^2-1 nodes, each has two pointers. Whereas a quadtree requires at most $(N^2-1)/3$ nodes, each has four pointers. This could, however, be

traded off against ease of programming, especially when 3D or higher dimensional data are considered. As our interest is in programming aspects, the approach to compose higher-order operations from lower ones may be worth investigating.

With regard to using binary trees for representing images, Knowlton [Knowlton80a] and later Ouksel *et al.* [Ouksel92a] reported the *bintree* representation of an image. A bintree is produced by subdividing an image into halves, instead of quarters for a quadtree (Figure 6-2 (b)). Whereas our approach is to subdivide an image into a tree of rows, each of which is further subdivided into a tree of pixels (Figure 6-2 (c)). Thus, although they both use binary trees as the fundamental data structure, they are different. It should be noted that, in terms of compression effect, a bintree is better than a tree of trees, and is better still than a quadtree since the minimum unit of tree condensation is two elements.

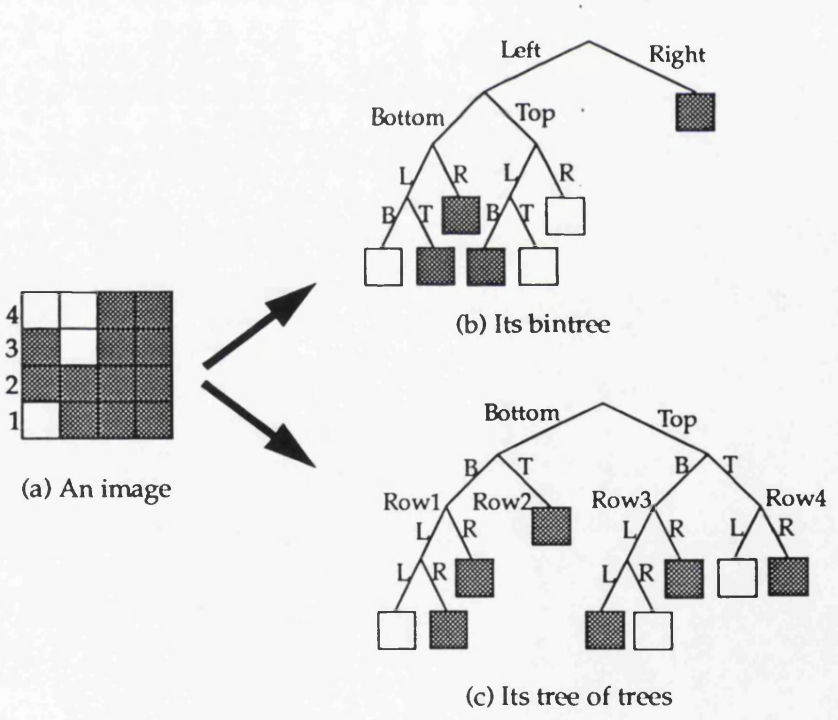


Figure 6-2 A bintree and a tree of trees

6.4.2 Image representation

A tree can be introduced by an algebraic data type definition. The type of a leaf node is parameterised by using the type variable * which is the type of a pixel in the same way as we

parameterised it in previous representations (See Section 3.2.2 for example), and a tree is defined as a recursive data type as follows:

```
tree * ::= Nil | Leaf * | Node (tree *) (tree *)
```

This can be read as *"a tree of * comprises either Nil, Leaf of *, or Node which consists of two further trees of *"*.

Like the definition in Section 5.3.1, the actual pixel data part and the parameters to describe the "shape" should be separated. Those parameters should be defined so as to make handling the data part easy. With the tree definition above, we use the position of the root, the length of the first step to go down the tree one level, and the tree itself. The reason is that these parameters are conveniently used when a tree is traversed from the root. The data structure `intervalT` is defined as a triple of these two parameters and a tree using a parameterised type for a pixel, as follows:

```
intervalT * == (num,num,tree *)
rowT *      == intervalT *
imageT *    == intervalT (rowT *)
```

For example, in Figure 6-3, as the row extends from x=0 to 8, its root position is x=4 and the first step is 2. So, the row is expressed as:

```
(4,2,Node (Node (Node (Leaf 3) (Leaf 42)) (Leaf 36)) (Node (Node (Leaf 10)
(Leaf 23)) (Leaf 23)))
```

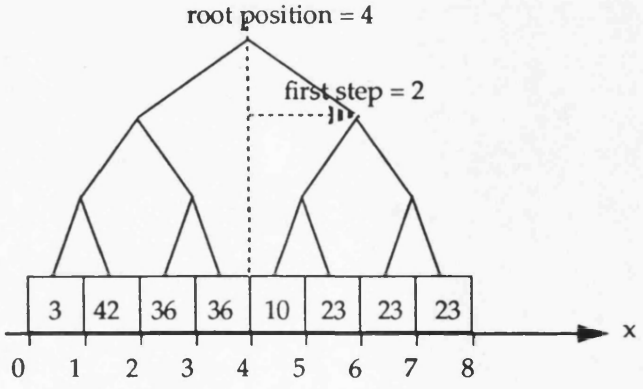


Figure 6-3 A pixel tree, root position and first step

It should be noted that the definitions for a row and an image are identical to the ones defined in Section 5.3.1. Another point to note is that we introduced `Nil` in the tree definition, so that it is possible to represent non-rectangular shapes including ones with holes. If the size of a tree is not a power of two, the tree is still treated as having potential to represent the size up to a minimum power of two which is larger than the size. A tree is strictly balanced in the sense that the shape of the tree represents exactly the shape of the interval, even if the tree itself may not be a balanced tree. For example, Figure 6-4 contains a hole and the size is not equal to the 2's power, but it can be treated as the tree whose potential maximum size is 8. This can be represented as:

```
(4,2,Node (Node (Node (Leaf 3) (Leaf 42)) (Node (Leaf 36) Nil)) (Node (Node
(Leaf 10) (Leaf 23)) Nil))
```

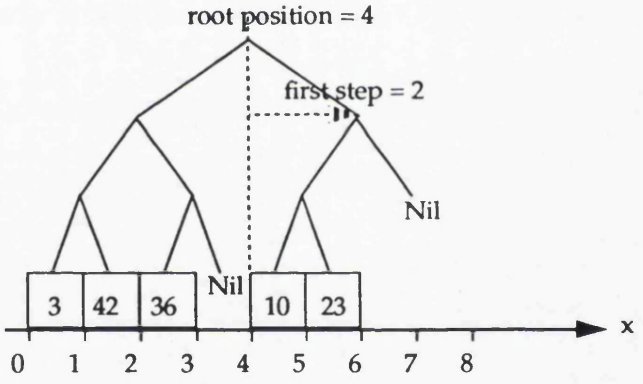


Figure 6-4 A row with Nil's

6.4.3 Generating an image represented by a tree of trees

An image is typically represented by an array of pixels, or a list in Miranda when, for example, input from a file. So the first thing to be done is to convert an image of the list representation into the tree representation. As usual, a function in 1D is considered first, then a 2D operation will be implemented in a higher level.

It is assumed that pixels in a row are left packed and rows in an image are bottom packed, i.e. the origin of an original image is (0,0) and the elements are aligned with the integer grid. The basic `makeTree` function is defined as follows:

```
makeTree::[*]->num->tree *
makeTree [] s    = Nil
makeTree [x] 1   = Leaf x
makeTree list s = Node (makeTree left half) (makeTree right half)
                  where left  = take half list
                        right = drop half list
                        half  = s div 2
```

The second argument of the type num is used as the size of the tree to be made out of the list, since the possible size of a tree is a power of 2 though the number of pixels in a list may not be equal to it. It would be possible to calculate the size using the length of a list, but taking the length each time may be expensive[1]. Thus, the size argument is carried which is halved at each recursive call.

Using this `makeTree` function, the other related functions can be defined: `listToTree` converts a 1D list to a tree, `listToInterval` a 1D list to an interval, and `listToImage` a 2D list to an image. The definitions for these functions follow:

```
listToImage::[[*]]->imageT *
listToImage l = listToInterval (map listToInterval l)

listToInterval::[*]->intervalT *
listToInterval l = (s/2, s/4, condenseTree (listToTree l))
                   where s = maxSize (#l)

listToTree::[*]->tree *
listToTree l = makeTree l (maxSize (#l))

maxSize 0 = 0
maxSize 1 = 1
maxSize n = 2^(entier(log (n-1)/log 2)+1)
```

In the `listToInterval` function, a tree is condensed by the function `condenseTree` which will be described in the next subsection.

6.4.4 Tree condensation

A higher-order function to rearrange a tree can be defined as follows, which takes a function to reshape a tree and applies the function recursively to its children:

1. Related discussion is presented in Chapter 8.

```
reformTree::(tree *->tree *)->tree *->tree *
reformTree f (Leaf x)        = f (Leaf x)
reformTree f Nil             = f Nil
reformTree f (Node t1 t2)  = f (Node (reformTree f t1) (reformTree f t2))
```

The tree condensation function is defined just by passing a function to convert a Node with two Leaf's of the same value into a Leaf of that value, or two Nil's to one:

```
condenseTree::tree *->tree *
condenseTree = reformTree fun
               where fun (Node (Leaf x) (Leaf x)) = Leaf x
                     fun (Node Nil Nil)           = Nil
                     fun t                        = t
```

6.4.5 Translation and scaling

In order to define geometric transformation functions, it is necessary to define a higher-order function to map a function over each element in a tree. The effect of the function is the same as the map function on a list. The code for this function can be defined as follows:

```
mapTree::(*->**)->tree *->tree **
mapTree f Nil             = Nil
mapTree f (Leaf x)        = Leaf (f x)
mapTree f (Node x1 x2)    = Node (mapTree f x1) (mapTree f x2)
```

Using this function, and considering that transformations do not affect the tree part, geometric transformations can be defined in an identical way to those defined for the list representation (See Section 5.3.2):

```
transRowT d (r,s,t)  = (d+r,s,t)
scaleRowT sc (r,s,t) = (sc*r,sc*s,t)

transImageT dx dy (ry,sy,t)
   = transRowT dy (ry,sy,mapTree (transRowT dx) t)
scaleImageT scx scy (ry,sy,t)
   = scaleRowT scy (ry,sy,mapTree (scaleRowT scx) t)
```

6.4.6 Displaying an image

Displaying an image represented by a hierarchical data structure is carried out lazily in terms of resolution. When resolution becomes higher than required, traversal stops at that level and

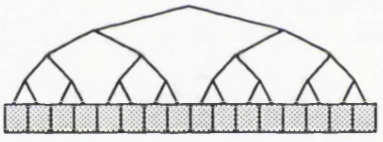
the average value of the pixels underneath is calculated. The process for displaying can be modularised into the following three sub-processes which can be implemented as separate functions. Figure 6-5 illustrates how transformed pixels are displayed at required resolution:

a. Scanning the output image along the display grid. An interval between two display positions is 1 (the rightward arrow in Figure 6-5 (b)).

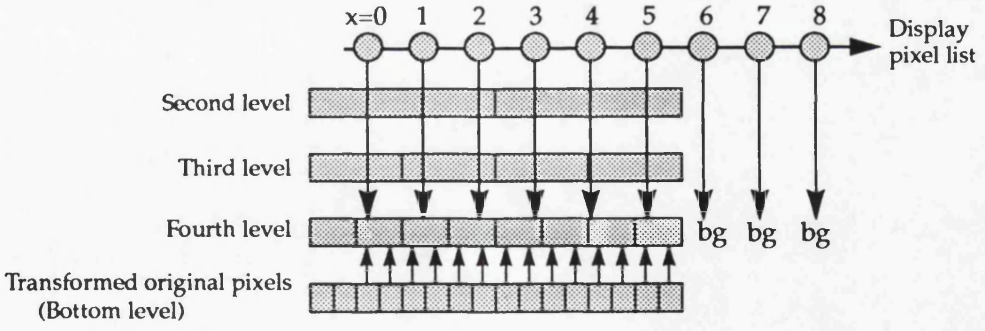
b. Traversing down the tree of transformed pixels until an interval between two elements becomes smaller than 1, i.e. the first step in the interval data structure is less than 0.5 (the downward arrows in Figure 6-5 (b)).

c. Averaging pixels underneath which gives the value of the output pixel (the upward arrows in Figure 6-5 (b)).

If a pixel outside the range of the original is required, a background value is returned (the display pixels at x=6, 7, 8, in Figure 6-5).



(a) A pixel tree



(b) Displaying at appropriate resolution

Figure 6-5 Displaying pixels with averaging at a required resolution

The following `dispT` function corresponds to the sub-process (a.). Assuming that a function to look up the value from the pixels underneath is properly defined (`lookUpT`), the `dispT` function can be defined recursively as follows:

```
dispT::num->num->(*->*->*)->*->intervalT *->[*]
dispT x n fn bg i
   = [] , if n=0
   = lookUpT x fn bg i:dispT (x+1) (n-1) fn bg i , otherwise
```

The first and second arguments specify the starting position and the number of pixels to be displayed. The third argument is a function for averaging which is polymorphic to average either pixels or rows. This is similar to the disp function defined for lists (page 83) which takes a polymorphic interpolate function. It is also necessary to pass a background value as an argument for the same reason as described in Section 5.3.3.

The second function, sub-process (b.), traverses a tree until the required resolution is reached, and looks up a value from the subtrees underneath. A background value is returned if the position sought is outside the range or the value of the tree is Nil. If the tree is a Leaf, then the value of the Leaf is returned. Otherwise, either an average value is returned or the tree is further traversed depending on the required resolution. The definition follows:

```
lookUpT::intervalT *->num->(*->*->*)->*->*
lookUpT (r,s,t) x fn bg          = bg , if (x<r-2*s)\/(r+2*s<=x)
lookUpT (r,s,Nil) x fn bg        = bg
lookUpT (r,s,Leaf a) x fn bg     = a
lookUpT (r,s,Node t1 t2) x fn bg
   = avrTree (Node t1 t2) fn bg  , if s<0.5
   = lookUpT (r-s,s/2,t1) x fn bg, if x<r
   = lookUpT (r+s,s/2,t2) x fn bg, otherwise
```

The function `avrTree` calculates the average value of the subtrees underneath, i.e. the sub-process (c.). Depending on whether the elements of a tree are numbers or lists of numbers, the methods for averaging differ. So, `avrTree` is defined as a higher-order function which takes a function to average either numbers or lists of numbers:

```
avrTree::tree *->(*->*->*)->*->*
avrTree Nil fn bg         = bg
avrTree (Leaf a) fn bg    = a
avrTree (Node t1 t2) fn bg= fn (avrTree t1 fn bg) (avrTree t2 fn bg)

avrList::[num]->[num]->[num]
avrList l1 l2 = [(a+b)/2 | (a,b)<-zip2 l1 l2]

avrNum::num->num->num
avrNum a b = (a+b)/2
```

In this way, the necessary functions to implement the sub-processes (a) - (c) have been

prepared. Using these functions, the function to display an image represented by a tree of trees can be defined. The `dispT` function is mapped over each row (by `mapTree`) to output a list of pixels of a required resolution in the x direction. And the resulting tree of lists is operated by the `dispT` function to output a list of lists of pixels. The definition follows:

```
displayT::num->num->num->num->imageT num->[[num]]
displayT x y xn yn (r,s,t)
 = dispT y yn avrList (rep xn bgp) (r,s,mapTree (dispT x xn avrNum bgp) t)
```

6.4.7 Discussions

For the same reason as we discussed in Section 5.3.4, i.e. keeping x and y operations separate, rotation is not implemented using the current data structure of a tree of trees. However, the reason for taking this representation is to test whether the technique we have developed (i.e. an identical pattern for x and y data structures makes programming easier) can be applied to data structures other than lists. It has now been confirmed that this technique can be applied to a binary tree of binary trees.

Rotation may be implemented fairly easily by using existing techniques. For example, we can modify the definition of a row to be a triple of a root position (x, y), a first step length (x, y), and a binary pixel tree. The whole image may be represented by a list of rows. This representation is very similar to the one we defined in Section 5.4.1 except that a tree instead of a `list` is used to represent a row. By mapping the one dimensional `dispT` function defined on page 105 onto the x elements of each row, a list of lists of pixels aligned in the x direction is obtained. A function for aligning these pixels in the y direction has previously been implemented (See Section 5.4.3).

In the next section, we introduce a quadtree to represent an image. Affine transformations and display at required resolution are implemented using quadtrees.

6.5 Quadtrees

6.5.1 The 2D vector abstract data type

Before implementing an image represented by a quadtree, we will introduce an *abstract data type* to represent and manipulate 2D vectors. The image data structures used so far are defined in the x direction first, then composed to represent 2D images. However, the quadtree

implemented in this section is a 2D data structure, so that various operations to manipulate 2D geometry will be necessary. We define an abstract data type called *vector* for this purpose which is essentially a pair of x and y elements. By packaging up a vector as an abstract data type, the readability of code will be improved since it hides the implementation detail from programmers' view. A part of the implementation is described in this subsection, but for the full implementation, see Appendix A.9.

An abstract data type can be defined in Miranda using `abstype` and the type signatures for its operations can be introduced by using `with`. For example:

```
abstype vector
with vMake::(num,num)->vector
     vXelement::vector->num
     vYelement::vector->num
     vAdd::vector->vector->vector
```

where `vMake` makes a vector from a pair of numbers, `vXelement` and `vYelement` return the x and y element of a vector respectively, and `vAdd` adds two vectors. An example implementation follows:

```
vector   == (num,num)

vMake      (x,y) = (x,y)
vXelement  (x,y) = x
vYelement  (x,y) = y

vFun2 fn (x1,y1) (x2,y2) = (fn x1 x2, fn y1 y2)
vAdd  = vFun2 (+)
```

In this way, a vector can be manipulated as one piece of data and access to the data is only through the operations defined for the vector data type.

6.5.2 Image representation

As briefly introduced in Section 6.2, the term 'quadtree' is used to describe a class of hierarchical data structures whose common property is that they are based on the principle of recursive decomposition of space [Samet84a]. The particular kind of quadtree which is studied in this section is called *a region quadtree* which is based on the successive subdivision of an image into four quadrants of equal size [Klinger76a].

With regard to quadtrees in functional languages, Parsons implemented them using

Miranda [Parsons87a]. He presented a straightforward implementation of a region quadtree that did not allow non-square images or images with holes. Since his primary application was graphics, he implemented a number of functions to convert graphics primitives such as points and lines, not derived from pixel images, into a quadtree. He also implemented geometric operations on a quadtree but they are rather limited: rotating a quadtree by multiples of 90 degrees, and reflecting a quadtree by multiples of 45 degrees. These functions are applied on a quadtree and produce a quadtree as a result.

Our implementation allows non-square images which may have holes. This can be done by introducing *Nil quadtrees* as an extension to the binary tree definition (Section 6.4.2). Like a binary tree being able to represent an interval of the size up to a power of two, a quadtree can represent a square of the size up to a power of two and the shape of a quadtree precisely represents the shape of the image. Also, an average value is attached to each node since an average value to represent subtrees underneath a node will be required when displaying the node. The definition is as follows[2]:

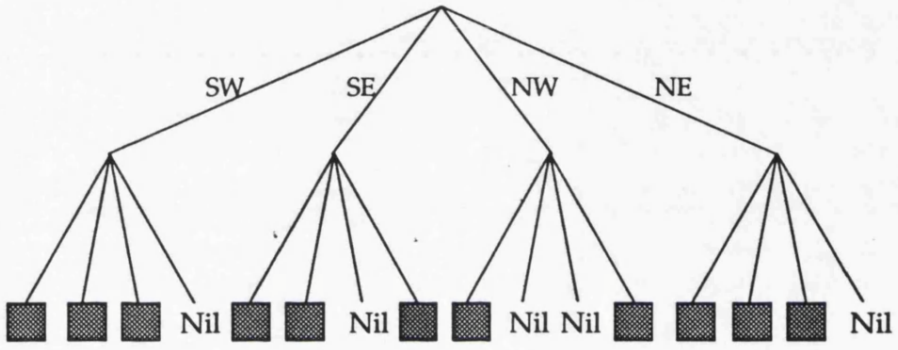
```
qtree *
   ::= QNil | QLeaf * | QNode * (qtree *) (qtree *) (qtree *) (qtree *)
```

The type of a pixel is parameterised using the type variable *. A quadtree of pixels is either a Nil, Leaf of a pixel, or Node with the average value of the subtrees and its four child subtrees. In the code, the constructor names `QNil`, `QLeaf` and `QNode` are used to avoid confusion with the binary tree definition. Figure 6-6 shows an example of a non-square image with a hole.

We use a different ordering of the four child subtrees from the common ordering used in most image processing applications using quadtrees, i.e. NW, NE, SW, SE, based on the common raster order. Instead, we use SW, SE, NW, NE based on the conventional x y coordinate system because it will make handling of various geometric operations easier.

2. As an extension to the binary tree definition on page 100, we treat four subtrees separately. But there are many alternatives, such as representing subtrees by a 4-list.

(a) A non-square image with a hole
(▩ denotes a defined pixel)



(b) Its quadtree

Figure 6-6 Representing a non-square image with a hole

As in the various image representations used so far, a few parameters are attached to the pixel image itself in order to represent the "shape" of a quadtree and its position in world coordinate. By analogy with the image definition using binary trees (Section 6.4.2), the root coordinate and the first step length will be necessary. Because the geometric operations which will be defined on images are affine, a quadtree may become a parallelogram. So, two vectors to represent the first steps are necessary. We use vectors for first steps in SW and SE directions. Hence, the definition of an image:

```
coord      == vector
imageQT *  == (coord,vector,vector,qtree *)
```

where the first element is the root coordinate, the second the SW first step vector, and the third the SE first step vector. An (x,y) pair can be converted to a vector (or a coord) by `vMake(x,y)`.

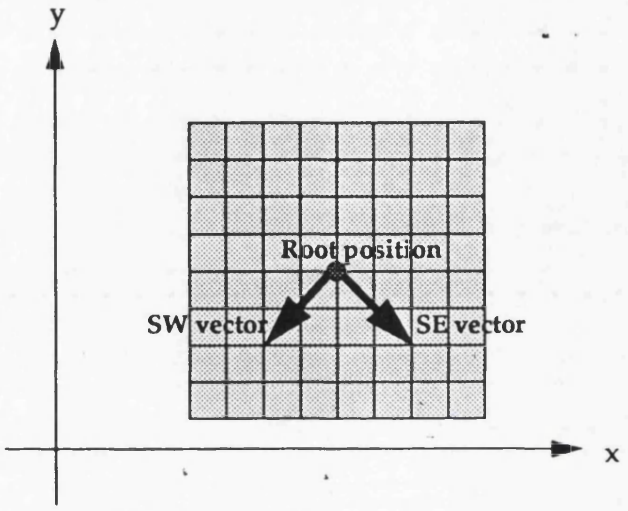


Figure 6-7 Root position and two vectors of a quadtree

6.5.3 Converting a 2D list to the image data structure

The first function to consider is to convert a 2D rectangular list of pixels, the most common representation of an image, to the image data structure using a quadtree, i.e. a quadtree version of `makeTree`.

As a quadtree is based on subdividing space into four, a function to do this subdivision needs to be defined. This function can be regarded as a 2D version of `take` and `drop`. The following `quarter` function returns either the SW, SE, NW, or NE square part of a list of lists:

```
quarter::num->num->[[*]]->[[*]]
quarter i n
   = divide eastWest northSouth
     where divide f g l = g (map f l)
           eastWest     = [take n, drop n]!(i mod 2)
           northSouth   = [take n, drop n]!(i div 2)
```

The arguments of this function are an index to specify which portion to be taken (0:SW, 1:SE, 2:NW, 3:NE), and the number of elements to be taken. For example:

`quarter 0 2 [[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]]` returns
`[[1,2],[5,6]]`

`quarter 1 2 [[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]]` returns
`[[3,4],[7,8]]`

Since an average value has been included as a part of the quadtree definition, a quadtree is produced from a 2D list by calculating average values. The following `avrQTree` function calculates the average value of a quadtree using `valQTree` which returns the value of a quadtree. In order to make these functions polymorphic, a background value and an averaging function should be taken as arguments:

```
avrQTree::*->(*->*->*->*->*)->qtree *->qtree *->qtree *->qtree *->*
avrQTree bg f t0 t1 t2 t3
 = f (valQTree bg t0) (valQTree bg t1) (valQTree bg t2) (valQTree bg t3)

valQTree::*->qtree *->*
valQTree bg QNil          = bg
valQTree bg (QLeaf v)     = v
valQTree bg (QNode v t0 t1 t2 t3) = v
```

Using all these functions, the function to convert a 2D list into a quadtree can be defined as follows:

```
makeQTree::*->(*->*->*->*->*)->[[*]]->num->qtree *
makeQTree bg fn [[]] s     = QNil
makeQTree bg fn []   s     = QNil
makeQTree bg fn [[a]] 1    = QLeaf a
makeQTree bg fn lists s
   = makeQNode bg fn (subt 0) (subt 1) (subt 2) (subt 3)
     where half   = s div 2
           subt i = makeQTree bg fn (quarter i half lists) half

makeQNode bg fn t0 t1 t2 t3
   = QNode (avrQTree bg fn t0 t1 t2 t3) t0 t1 t2 t3
```

The `makeQTree` function takes a background value, an averaging function, a 2D list, and the size of a quadtree to be built. The role of the last argument is the same as the one introduced in the 1D case (See `makeTree` on page 102). The first two cases of `makeQTree` are for an empty list. The next case is that a pixel becomes a leaf node, and the last case is for subdividing the area further. The `makeQNode` function takes a background value, an averaging function, and four child quadtrees, and returns a `QNode`.

Using this `makeQTree` function, functions to convert a 2D list of numbers into a quadtree and into an image structure can be defined as follows:

```
listsToQTree::[[num]]->qtree num
listsToQTree lists
   = QNil , if lists=[]
   = condenseQTree (makeQTree bgp average4 lists s) , otherwise
     where s = maxSize (max2 (#(hd lists)) (#lists))

listsToImageQT::[[num]]->imageQT num
listsToImageQT l
   = (vMake(s/2,s/2),vMake(-s/4,-s/4),vMake(s/4,-s/4),
      condenseQTree (makeQTree bgp average4 l s))
     where s = maxSize (max2 (#(hd l)) (#l))

average4 a b c d = (a+b+c+d)/4
```

where condenseQTree is a quadtree condensation function defined in the next subsection.

6.5.4 Quadtree condensation

The structure of a quadtree condensation function is the same as the one defined for a binary tree, i.e., a higher-order function to rearrange the shape of a quadtree and the actual condensation function. A quadtree version can be defined as follows:

```
reformQTree::(qtree *->qtree *)->qtree *->qtree *
reformQTree f (QLeaf x) = f (QLeaf x)
reformQTree f QNil      = f QNil
reformQTree f (QNode v qt0 qt1 qt2 qt3)
   = f (QNode v (reformQTree f qt0) (reformQTree f qt1)
                (reformQTree f qt2) (reformQTree f qt3))

condenseQTree::qtree *->qtree *
condenseQTree
   = reformQTree fun
     where
     fun (QNode v (QLeaf x) (QLeaf x) (QLeaf x) (QLeaf x)) = QLeaf x
     fun (QNode v QNil QNil QNil QNil)                     = QNil
     fun t                                                 = t
```

6.5.5 Transformations

Translation, scaling and rotation of an image can be expressed by using some operations for 2D vectors as follows:

```
transQTree dx dy (r,sw,se,qt) = (vAdd (vMake (dx,dy)) r,sw,se,qt)
scaleQTree scx scy (r,sw,se,qt) = (scl r,scl sw,scl se,qt)
                                  where scl = vMul (vMake (scx,scy))

rotateQTree th (r,sw,se,qt) = (rot r,rot sw,rot se,qt)
                              where
                              rot = vMap2 (rotx th, roty th)
                              rotx th x y = x*(cos th)-y*(sin th)
                              roty th x y = x*(sin th)+y*(cos th)
```

Note that, unlike the image representations used so far where the "1D first, then 2D" policy was adopted, 2D geometric operations are directly implemented

6.5.6 Displaying an image

Since average values to represent the subtrees underneath have been incorporated in the definition of a quadtree, the process to display pixels can be modularised into the following two steps:

a. Scanning the output image along the display grid. An interval between two display positions is 1.

b. Traversing down the quadtree of transformed pixels until an interval between two elements becomes smaller than 1, i.e. the lengths of both SW and SE first step vectors are less than $1/\sqrt{2}$. At that point the value is fetched using the previously defined `valQTree` function.

The sub-process (a.) is a straightforward recursive definition. The following `dispQT` scans in 1D and displayQT does it in 2D:

```
displayQT::num->num->num->num->*->imageQT *->[[*]]
displayQT xs ys xn yn bg im
   = [] , if yn=0
   = dispQT xs xn ys bg im:displayQT xs (ys+1) xn (yn-1) bg im , otherwise

dispQT::num->num->num->*->imageQT *->[*]
dispQT xs xn y bg im
   = [] , if xn=0
   = lookUpQTree (vMake (xs,y)) bg im:
     dispQT (xs+1) (xn-1) y bg im , otherwise
```

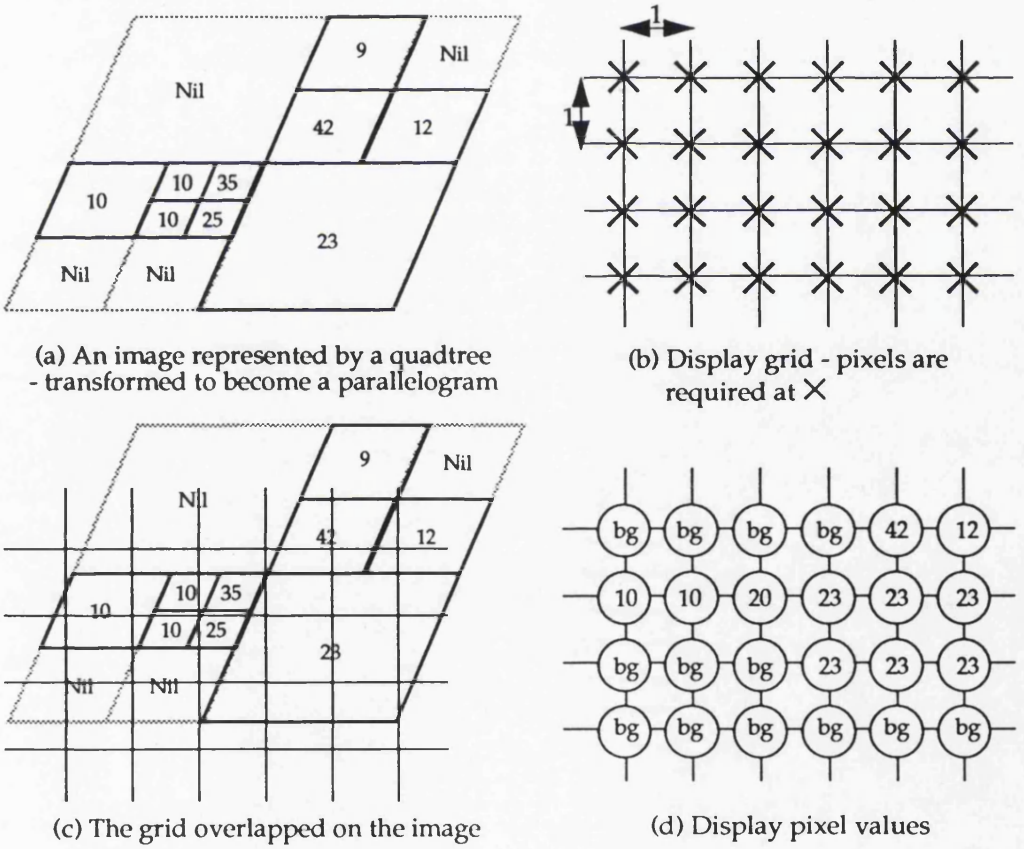
In the sub-process (b.), there are three cases to consider:

1. If the point is outside the transformed quadtree or the quadtree is Nil, then the background value is returned.

2. If the quadtree to look at is a Leaf or the quadtree is smaller than required, i.e. both the SW and SE first steps are less than $1/\sqrt{2}$, then return the value of the quadtree. This may be either the Leaf value or the average value of the Node.

3. Otherwise, traverse the quadtree further down.

Figure 6-8 illustrates an example of displaying a quadtree. Figure 6-8 (a) is an image represented by a quadtree. The image has been affine transformed to become a parallelogram and contains Nil elements. (b) is a display grid, in which the interval between two display positions is 1. (c) shows the display grid overlapped on the image. (d) is an array of display pixels.



(a) An image represented by a quadtree - transformed to become a parallelogram

(b) Display grid - pixels are required at ×

(c) The grid overlapped on the image

(d) Display pixel values

Figure 6-8 Displaying a quadtree at a required resolution

The following `lookUpQTree` function is an example implementation which takes a point coordinate, a background value and an image as its arguments. The three cases in the

code correspond to the three cases described above.

```
lookUpQTree::coord->*->imageQT *->*
lookUpQTree p bg (root,sw,se,qt)
   = bg              , if ~(vIsInside p vertexList)\/qt=QNil
   = valQTree bg qt , if isQLeaf qt\/(vLength sw<len)&(vLength se<len)
   = lookUpQTree p bg (nextRoot,vHalf sw,vHalf se,subTree qt), otherwise
     where
     len          = 1/sqrt 2
     vertexList   = map ((vAdd root).vDouble) [sw,se,vNeg sw,vNeg se]
     s            = sw $vAdd se
     w            = sw $vSub se
     isWest       = vDirection (root $vSub p) (root $vAdd s $vSub p) < 0
     isSouth      = vDirection (root $vSub p) (root $vAdd w $vSub p) > 0
     (nextRoot, i) = (root $vAdd. sw, 0) , if isWest & isSouth
                   = (root $vAdd se, 1) , if ~isWest & isSouth
                   = (root $vSub se, 2) , if isWest & ~isSouth
                   = (root $vSub sw, 3) , if ~isWest & ~isSouth
     subTree (QNode v a b c d) = [a,b,c,d]!i

isQLeaf (QLeaf a) = True
isQLeaf qt        = False
```

Although the number of cases is only three, the code looks quite complicated as a number of judgements are involved to decide which case to take and which subtree to traverse. Note also that in this implementation there are many repeated computations in displaying an image, since it does not utilise a previous pixel computation in calculating a next pixel and a quadtree is always traversed from the root. It would be possible to improve the code but more investigation is needed.

6.6 Discussions

6.6.1 Comparing trees with quadtrees

Comparing the two implementations, the quadtree version appears less readable than the 'tree of trees' version; the number of cases to be considered increases as the dimension of data goes up (compare `makeTree` with `makeQTree`, and `lookUpT` with `lookUpQTree`). Given the corresponding increase in programming difficulty between tree of trees and quadtrees, it is likely that the implementation of octrees would be quite complex and difficult.

The principal reason for the programming difficulty of quadtrees lies in the fact that quadtrees cannot be constructed from 1D structures. In the tree of trees implementation, 1D

operations are defined first and there is little difficulty in composing 2D operations using higher-order functions. The method of composition is identical to what we have developed for the list of lists implementation. In the quadtree version, since the data structure itself is two-dimensional, there appears to be no way to program its operations in a similarly modular way. On the other hand, as discussed in Section 6.4.1, a quadtree requires a smaller number of nodes than a tree of trees to represent an image, so if an image gets larger and larger, the tree of trees representation will be less memory efficient compared with the quadtree representation.

It can therefore be concluded that one of the advantages of functional programming is the ease of programming it offers by allowing complicated data structures and functions to be composed using existing simpler ones. However, it is not clear whether programs composed in this way can always be as efficient.

6.6.2 Higher-dimensional data and overloading

In the quadtree version, the basic operations to handle higher dimensional data, such as `vector`, are required to manipulate that data directly, and abstract data types are an effective way to encapsulate such data and operations. However, there is still some difficulty in using such data types. The operations on vectors can be written using user-defined infix operators, such as `$vAdd` and `$vSub`, but these operators are not easy to read without a good knowledge of their meaning. The code would be much clearer and easier to read if the operators such as + and – were overloaded to express vector operations, as well as scalars. In this respect the type classes and the overloading mechanism employed in Haskell seem attractive.

6.6.3 Comparing hierarchical data structures with lists

This subsection discusses some issues regarding the representation of images raised by comparing the representations described in this chapter and the list representation employed in the previous chapter.

One of the reasons for using hierarchical data structures is that they allow more efficient random access than linear lists (Section 6.3). This simplifies the coding of functions to display a transformed image since programmers do not have to express a function to synchronise output and input, such as `dropUpto` defined on page 84. In the `dispT` function implemented in Section 6.4.6, the original image stays unchanged through recursive calls. Programming with random access and simple iteration is a common style in image processing, e.g. using an

array and a loop. Thus, if ease of programming is viewed as the closeness to the currently familiar style, this style may be advantageous for many programmers. However, there is a trade-off with efficiency, as pointed out in Section 5.5. That is, if input and output images are relatively similar in orientation and resolution, the list representation may present better efficiency since it uses the nature of lists, i.e. elements are sequentially ordered, which are treated as lazy streams.

Another important point to add in terms of representations is the data compression aspect of hierarchical data structures. Since our interest has been how hierarchical data structures can be programmed in a lazy functional language, we did not specify the kinds of images to be processed. In general, the fewer the number of leaf values and the larger the homogeneous areas in an image are, the more compression effect can be expected. The space efficiency in this context is data dependent and the overall efficiency should be discussed in terms of applications to real images.

The final aspect in this discussion of representations is the use of arrays in lazy functional languages. Since Haskell has arrays, it may be of interest to implement the same algorithms using arrays and compare the issues of ease of writing/reading and efficiency. As far as random accessibility is concerned, arrays ensure O(1) time access, but the structure must be allocated beforehand. Although Haskell's arrays are non-strict, i.e. unless an element of an array is required the element is not evaluated, the entire memory block for an array is allocated as soon as any part of it is needed. In image processing this can be expensive, since the size of an array is usually very large. Use of arrays has not been considered here, and has been left for future development.

6.7 Summary

We have presented two implementations of geometric transformations and image display using hierarchical data structures as the fundamental representation. The first part of this chapter described a binary tree of binary pixel trees, in which the technique, "1D first, then 2D using higher-order functions", has been successfully applied to hierarchical data structures. The second part described the quadtree representation which should be more space efficient, but less easy to program because of the less modular style of programming. The actual efficiency is data dependent and left for future investigation.

Chapter 7: Combining Pixel and Non-Pixel Images

7.1 Introduction

As discussed in Chapter 4 lazy functional languages should allow operations involving pixel images and non-pixel images, such as an image expressed in a continuous function, to be treated uniformly. This is enabled by the following two characteristics of lazy functional languages:

- The languages are *functional* and can handle functions and data without discrimination, so that even an image expressed as a function can be processed using function composition.
- The languages are *lazy* and do not try to produce output pixels unless it is absolutely necessary regardless of whether the original image is represented in pixels or functions. Even when a pixel image and a function image are combined, output pixels are only calculated if they are needed.

In this chapter, an implementation of combining pixel and non-pixel images is presented. Both types of images are implemented as abstract data types including file I/O to allow processing real images.

7.2 File I/O in Miranda

So far we have deliberately ignored discussing how to process real images such as the ones from image input devices or stored as image files. But communication with outside world is essential if the algorithms are applied to real images.

7.2.1 Rasterfile read/write functions

As an example of image file format, Sun's *rasterfile format* [Sun Microsystems Inc.87a] is taken. Although there are various image file formats around, they are not much different in essence, i.e. the header information containing attributes such as size, depth, etc., and the image itself. Rasterfile format consists of three parts: a header containing 8 integers, a (possibly empty) set of colour map values, and the pixel image stored line by line in increasing y order. The integers contained in the header are encoded as *eight four-byte integers*, and a *four-byte integer* is encoded as a *thirty-two-bit binary number* in which the bits are ordered from MSB to LSB. Although this is quite a common scheme used in the UNIX/C environment on many machines, data

encoding including byte order is machine dependent and the following code will need to be modified if it runs on machines with different schemes.

Miranda treats a file as a list of characters. The type signature of the Miranda function to read a file is defined as a conversion from a file name to a list of characters:

```
read :: [char]->[char]
```

Since there is no compatibility between Miranda's type num and the encoding scheme for a number described above, it is necessary to implement a conversion from a series of four characters to a series of numbers. The following `fourCharToNum` converts a list of four characters to a number and `charToNumList` converts a list of characters to a list of numbers:

```
charToNumList::[char]->[num]
charToNumList [] = []
charToNumList cs = fourCharToNum (take 4 cs):charToNumList (drop 4 cs)

fourCharToNum::[char]->num
fourCharToNum cs = foldl1 (+) (map2 (*) (map code cs)
                   (map (256^) [3,2..0]))
```

where code is a function to convert a `char` to a num.

Conversely, if a number in Miranda is written as a four-byte integer a conversion from a list of numbers to a list of four-byte integers must be explicitly written. The following two functions are example to do this job:

```
numToCharList::[num]->[char]
numToCharList [] = []
numToCharList (n:ns) = numToFourChar n++numToCharList ns

numToFourChar::num->[char]
numToFourChar n = map (decode.(mod 256).(n div)) (map (256^) [3,2..0])
```

where decode is a function to convert a num to a `char`.

In order to deal with various rasterfile parameters, a program needs to be able to interpret the eight numbers in the header as a series of parameters. If the code was written in C, this would be done by just `#include <rasterfile.h>`, but since Miranda does not have a mechanism to include C's header files this should be coded somewhere. In the following example, the rasterfile parameters are implemented as a list of functions from a header (represented as a list of numbers) to a number:

```
[ras_magic, ras_width,ras_height, ras_depth,
 ras_length,ras_type, ras_maptype,ras_maplength] = [(!i)|i<-[0..7]]

headerLength  = 32
ras_magic_num = 1504078485
rmt_equal_rgb = 1
```

In the current implementation, only gray images whose pixels are 8 bit integer can be handled. To impose this restriction, the header parameters should be checked to see whether the file satisfies it. The following `readHeader` takes a filename and returns a header:

```
readHeader::[char]->[num]
readHeader name
   = error "getHeader:not a rasterfile."              , if ~isRaster
   = error "getHeader:not a byte image."              , if ~is8bit
   = error "getHeader:unexpected colour map type"     , if ~isEqualRgb
   = header , otherwise
      where
      isRaster   = (ras_magic header=ras_magic_num)
      is8bit     = (ras_depth header=8)
      isEqualRgb = (ras_maptype header=rmt_equal_rgb)
      header     = charToNumList (take headerLength (read name))
```

Using the above defined "low-level" functions, the following is an example implementation of reading a rasterfile and converting it to an image structure using the function `listsToImage` defined in Section 6.5.3:

```
readImage::[char]->image num
readImage name
   = (listsToImage.reverse.splitList width.map code.drop skip)
      (read name)
      where    skip     = headerLength+ras_maplength header
               width    = ras_width header
               header   = readHeader name

splitList::num->[*]->[[*]]
splitList n []    = []
splitList n l     = [take n l]++(splitList n (drop n l))
```

The code may seem fairly readable because `(read name)` is the producer of a list of characters and a chain of functions to process the string is described as a composite function. The long string read from a file is `drop`'d to leave only the pixel part; the function `code` is `map`'d to convert `char`'s to `num`'s; the long list is split into a 2D list; it is `reverse`'d to convert from the common raster order to the conventional x-y coordinate order; then it is converted to

an image structure. However, in terms of memory efficiency, this is rather inefficient since it involves `reverse` which usually requires the whole image to be stored in memory. This can be avoided if raster order image representation is used.[1]

Writing an image to a file is basically a reverse process of the above. Using a function to convert an image structure to a 2D list of pixels (named `display` in the code), an example is given in the following. Note that the functions to manipulate 2D vectors defined in Appendix A.9 are used:

```
writeImage::coord->vector->image num->[char]->[Sys_message]
writeImage origin size im name
   = [Tofile name (header++cmap++data), Closefile name]
      where
      header   = numToCharList [ras_magic_num,w,h,8,1,1,1,768]
      w        = vXelement size
      h        = vYelement size
      l        = w*h
      cmap     = map decode ([0..255]++[0..255]++[0..255])
      data     = (map decode.concat.reverse) (display origin size im)
```

A header represented as a list of eight numbers that is converted to a list of characters, a colour map for a straightforward gray image by giving a linear table for each colour, and a list of pixels are concatenated and written to a file. The type `[Sys_message]` is a special type in Miranda to communicate with outside world [Research Software Limited89a].

7.2.2 Discussions on I/O

Input/output operations are essential in image processing, but judging from the above example, it has to be admitted that Miranda would not particularly good at expressing I/O. Two issues may be pointed out.

In order to communicate with outside world, encoding/decoding of foreign data needs to be carried out, such as converting a four-byte integer to a number, or a one-byte character to a number. This problem would not arise as long as a file is written and read by one language in a persistent way. For example, Miranda provides the `readvals` function which can read a file directly if the file has been written by Miranda. Also in Haskell, if a file has been written by a `WriteBinFile` request, the file can be read in by a `ReadBinFile` request. But this

1. It would be straight forward to avoid this cost by storing the image in memory in raster order. However, for ease of explanation we have not done this. See Section 3.2 .

facility is implementation dependent and data persistency is not maintained even between different Haskell compilers [Trinder92a]. Although this kind of inconvenience would be inevitable in I/O in any case, it may not be wise to get such low level operations done by a high level language such as Miranda because of its execution speed. One possible solution to this data persistency problem would be for a language to incorporate code to import foreign data written in another language. For example, the new Glasgow Haskell compiler has a facility which allows arbitrary C functions to be called from the functional program without losing referential transparency [Hall92a].

The other issue of note is related to polymorphic typing. The above functions, `readImage` and `writeImage`, handle images of numbers only. In order to handle other kinds of images such as boolean or character images, a separate function with a unique name for each type of images will be necessary. A polymorphic function accepts any type as long as the function behaves exactly in the same way for any type, but in I/O this is not the case, i.e. there is no way to convert polymorphic types to a list of characters. This means that, although in the heart of functional programming polymorphic typing works quite nicely (Section 3.6.4), once I/O is involved, it becomes necessary for programmers to be aware that which concrete types are being processed.

The problem of polymorphic typing and communication with the outside world can be found in Miranda's special function `show` [Research Software Limited89a]. The function has been defined to cope with the need to convert an arbitrary value to its printable representation without requiring an infinite number of functions. However, the `show` function is rather difficult to use because it seems like a polymorphic function but actually it is not. The type of `show` must be determined monomorphically. For example, if the following function is defined without its type signature:

```
myShow x = "hello"++show x++"world\n"
```

the compiler will infer the type of `myShow` to be `*->[char]` and the type of `x` to be `*`. Since this usage of `show` is polymorphic the compiler reports an error. Hence, a type signature is essential in such a case.

The above discussion would be a typical example of why overloading is desirable. Haskell's facility of function overloading combined with polymorphic typing [Wadler89a] may become a solution to the problem, since it allows switching operations depending on the

type of data to be processed. Once separate I/O functions under the same name have been defined for separate instances in a type class, application programmers do not have to pay attention to which concrete types are being processed.

7.3 The Abstract Data Type: *function images*

An abstract data type called *function image* or `fImage` implements a non-pixel image. It is basically a function from a coordinate to a pixel. A coordinate is identical to the abstract data type *vector* described in Section 6.5.1, which is essentially a pair of x and y. Since a function image is the function itself, various processes can be written in functional languages using function compositions. In this section, geometric transformations of a function image are implemented as example.

7.3.1 Type signatures

The principal reason for using abstract data types would be to encapsulate data and operations together and hide detailed implementation from application programmers. For them it is sufficient to understand the type signatures for the abstract data type and its operations.

The following is the definition of the abstract data type for a function image or `fImage`:

```
filename == [char]

abstype fImage *
with makeFImage::(coord->*)->fImage *
     writeFImage::coord->vector->fImage num->filename->[sys_message]
     displayFImage::coord->vector->fImage *->[[*]]
     lookUpFImage::coord->fImage *->*
     translateFImage::vector->fImage *->fImage *
     scaleFImage::vector->fImage *->fImage *
     rotateFImage::num->fImage *->fImage *
```

As in the image data structures introduced so far, the type of a pixel is parameterised as the type variable `*`. `makeFImage` converts a function from a coordinate to a pixel into a function image.

Here we address again the problem with the polymorphic data types and I/O discussed in Section 7.2.2, Although a function image can be an image of any type, when an image is written to a file it is necessary to know exactly what the type of its pixel is. In the current

implementation, only an image of numbers can be handled (see the type signature of `writeFImage`). If the other types of images are to be dealt with, a separate function for each type must be defined under a unique name.

7.3.2 Implementation

The full implementation is given in Appendix A.6.

A function image is simply a function from a coordinate to a pixel. So, `makeFImage` is the function itself:

```
fImage * == coord->*
makeFImage fn = fn
```

`writeFImage` is almost identical to the function `writeImage` on page 122, so its implementation is not described here.

The next function is `displayFImage` which, like the display functions defined for images represented as lists or hierarchical data structures, converts a function image to a 2D list of pixels. The implementation is in two steps; `dispFImage` to produce a 1D pixel list and `displayFImage` to produce a 2D pixel list:

```
displayFImage position size fim
   = [] , if vYelement size=0
   = dispFImage position size fim
      :displayFImage (vYup position) (vYdown size) fim , otherwise

dispFImage position size fim
  = [] , if vXelement size=0
  = fim position:dispFImage (vXup position)(vXdown size) fim , otherwise
```

`lookUpFImage` is extremely simple, since a look-up function for a function image is the application of the function image itself:

```
lookUpFImage p fim = fim p
```

Lastly, three functions to transform a function image are defined. These are extremely simple, too, since these functions can be defined as function composition of 2D vector transformation and the function image itself:

```
translateFImage v fim = fim.((vAdd.vNeg) v)
scaleFImage v fim     = fim.((vMul.vRecip) v)
rotateFImage th fim   = fim.(vMap2 (rotx (-th), roty (-th)))
                        where rotx th x y = x*(cos th)-y*(sin th)
                              roty th x y = x*(sin th)+y*(cos th)
```

As we have seen, image processing of a function image can be expressed very simply as function compositions.

7.3.3 Examples

This subsection covers a few examples of function images. The procedure to produce a function image is to define a function which takes a coordinate, i.e. a vector, and returns a pixel, i.e. a number, then to produce a function image by using `makeFImage`. If it is desirable to write an image file, use `writeFImage` specifying an area. Note that currently we can write an 8bit integer gray image only (Section 7.2.1).

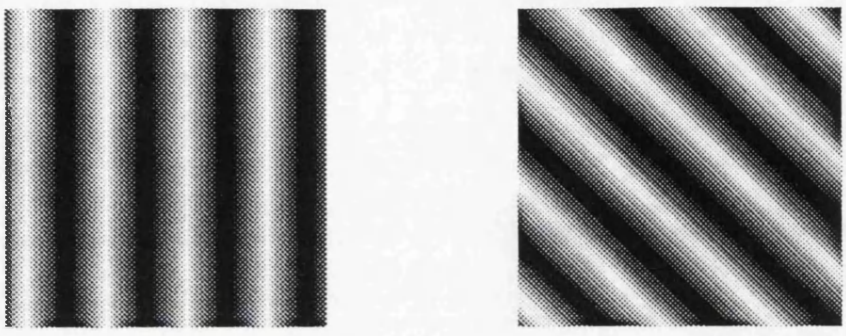
The first example is a simple sinusoidal image vibrating only in x direction. The code to express this function and to write this function image to a file as a 256×256 image is:

```
org = vMake (0,0)
siz = vMake (256,256)

fun1 c = 255*(1+sin (vXelement c*pi/32))/2
fim1 = writeFImage org siz (makeFImage fun1) "fim1.ras"
```

A rotation of this image by 45 degrees can be expressed as follows:

```
fim1R = writeFImage org siz im "fim1R.ras"
        where im = rotateFImage (pi/4) (makeFImage fun1)
```

The following Figure 7-1 shows these resulting function images.



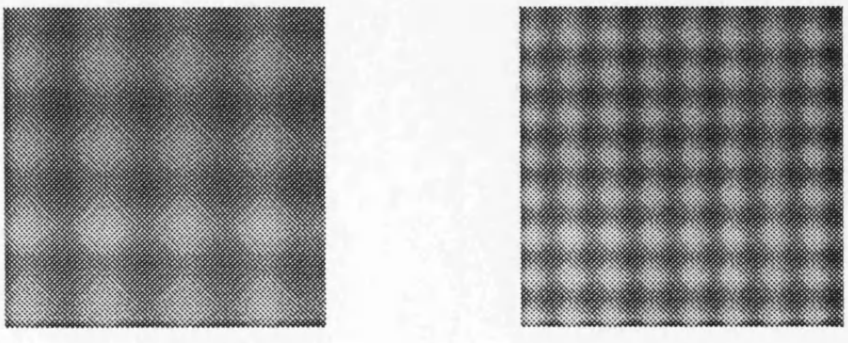
(a) fim1.ras (b) fim1R.ras

Figure 7-1 A sinusoidal image and its rotation

The next example is also sinusoidal but vibrating both in x and y directions, and involves scaling by (1/2,1/2). Example code is as follows:

```
fun2 c = 255*(2+1/2*(sin(x*pi/32)+sin(y*pi/32)))/4
         where x = vXelement c
               y = vYelement c

fim2  = writeFImage org siz (makeFImage fun2) "fim2.ras"
fim2S = writeFImage org siz im "fim2S.ras"
        where im = scaleFImage (vMake(1/2,1/2)) (makeFImage fun2)
```

And the resulting images are shown in Figure 7-2.



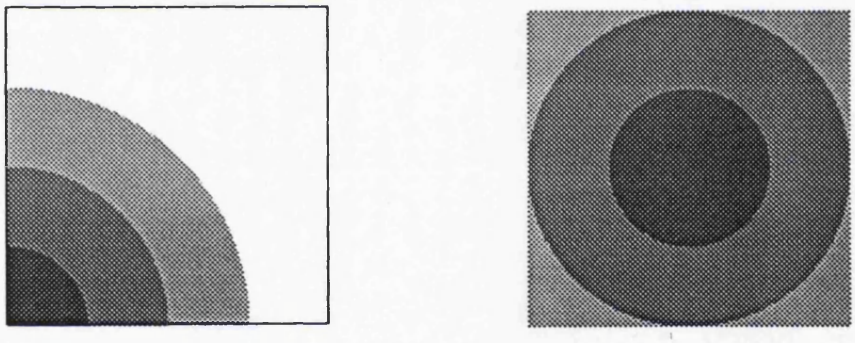
(a) fim2.ras (b) fim2S.ras

Figure 7-2 A 2D sinusoidal image and its scaling

The last example is not a continuous function as above, but concentric circles with discrete values. Also its translation is shown:

```
fun3 c = 63  , if radius<64
       = 127 , if radius<128
       = 191 , if radius<192
       = 255 , otherwise
         where radius = sqrt(vXelement c^2+vYelement c^2)
fim3  = writeFImage org siz (makeFImage fun3) "fim3.ras"
fim3T = writeFImage org siz im "fim3T.ras"
        where im = translateFImage (vMake(128,128)) (makeFImage fun3)
```

The resulting function images are shown in Figure 7-3.



(a) fim3.ras (b) fim3T.ras

Figure 7-3 A concentric circle image and its translation

7.4 The Abstract Data Type: *pixel images*

All the functions to manipulate pixel images have already been defined both for the list representation (Chapter 5) and the hierarchical representation (Chapter 6), and they have the same semantics. In other words, both implementations represent the same abstract data type. This section gives the type signature for that abstract data type. An example implementation using the list representation is given in Appendix A.7, and the quadtree representation in Appendix A.8.

7.4.1 Type signatures

The type signatures for the abstract data type `pImage` can be defined as follows:

```
abstype pImage
with readPImage::filename->pImage
     makePImage::[[pixel]]->pImage
     writePImage::coord->vector->pImage->filename->[sys_message]
     displayPImage::coord->vector->pImage->[[pixel]]
     translatePImage::vector->pImage->pImage
     scalePImage::vector->pImage->pImage
     rotatePImage::num->pImage->pImage
```

The type signatures are very similar to the ones for `fImage`. A major difference is that a pixel image is generated by reading from a file (`readPImage`) or converting from a 2D list (`makePImage`) instead of being made from a function.

Also, the type of pixels is restricted to `num` for the same reason as described in Section 7.2.2.

7.4.2 Examples

This subsection presents a few examples of the usage of pixel and non-pixel images. The examples may give some ideas of the level at which application programmers might work.

The first example reads an image file (`mandrill.im8`) whose original size is 250wx200h pixels, scales it by (1/2, 2/3), and writes it to a file (`pim1S.ras`) as the origin (0, 0) and the size (128, 128).

```
siz2  = vMake (128,128)
pim1S = writePImage org siz2 (sc im) "pim1S.ras"
        where
        im = readPImage "testimages/mandrill.im8"
        sc = scalePImage (vMake(1/2,2/3))
```

The next example reads the same file, rotates it by 45 degrees, translates it by (64, -24), and writes it to a file (`pim1SRT.ras`).

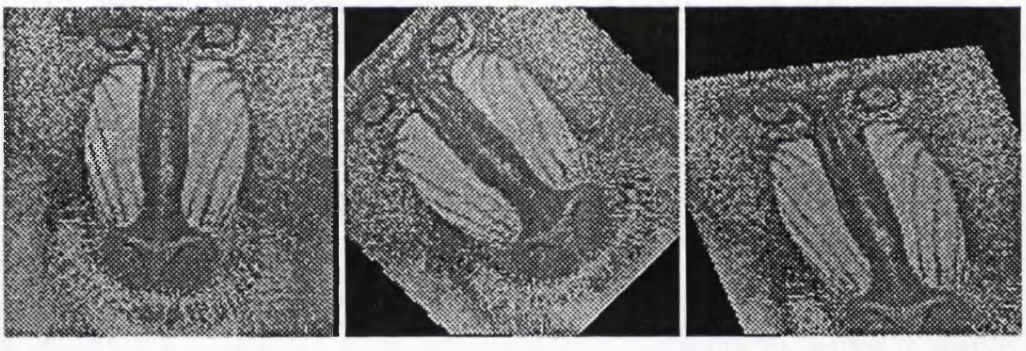
```
pim1SRT = writePImage org siz2 (tr im) "pim1SRT.ras"
          where
          im = readPImage "testimages/mandrill.im8"
          tr = trans.rotate.scale
          rotate = rotatePImage (pi/4)
          scale  = scalePImage (vMake(1/2,2/3))
          trans  = translatePImage (vMake(64,-24))
```

Another example operates a series of 6 transformations to demonstrate how

straightforwardly transformations can be expressed. In addition, the image quality is not degraded because there is no accumulation of quantisation errors, and no parts of the image have been lost because no intermediate images are produced. The code is as follows:

```
pim1SRTSRT = writePImage org siz2 (tr im) "pim1SRTSRT.ras"
             where
             im = readPImage "testimages/mandrill.im8"
             tr = trans2.rotate2.scale2.trans1.rotate1.scale1
             scale1  = scalePImage (vMake(1/2,2/3))
             rotate1 = rotatePImage (pi/4)
             trans1  = translatePImage (vMake(64,-24))
             scale2  = scalePImage (vMake(1.2,0.9))
             rotate2 = rotatePImage (-pi/6)
             trans2  = translatePImage (vMake(-20,25))
```

The resulting images are shown in Figure 7-4.



(a) pim1S.ras (b) pim1SRT.ras (c) pim1SRTSRT.ras

Figure 7-4 Transformed pixel images

The last example combines pixel and non-pixel images. It demonstrates what we described in Chapter 4, i.e. a spatial conditional operation using non-pixel images (Figure 4-3). This can be defined fairly simply as follows:

```
condIm im1 im2 im3
   = bPointwise cond im1 (bPointwise pair im2 im3)
     where cond True  (a,b) = a
           cond False (a,b) = b
           pair a b         = (a,b)
           bPointwise       = map2.map2

fun4 c = True  , if inside
       = False , otherwise
         where inside = (x-64)^2 + (y-64)^2 - 60^2 < 0
               x = vXelement c
               y = vYelement c

cim = condIm im1 im2 im3
      where
      im1 = displayFImage org siz2 (makeFImage fun4)
      im2 = displayPImage org siz2 (readPImage "pim1S.ras")
      im3 = displayFImage org siz2 (makeFImage fun2)

cim1 = writePImage org siz2 (makePImage cim) "cim1.ras"
```

condIm is just a redefinition of the spatial conditional operation since the previous definition on page 44 is for the representation with an origin and we have not defined the operation for a plain list of lists. fun4 defines the conditional part as a function image. Please note that although fun4 does not return a number but a boolean value it is still possible to make a function image as long as the image is not written to a file. cim provides the contents to the arguments to the condIm function, and evaluating cim1 produces an image file of the synthesised image. The result of this operation is given in Figure 7-5[2].

2. Note that the mandrill image in the output slightly differs from that in the input. This is because the output colour map is created as in cmap on page 122, while the input colour map is slightly different from that. How to handle colour maps from different sources would be another major problem.

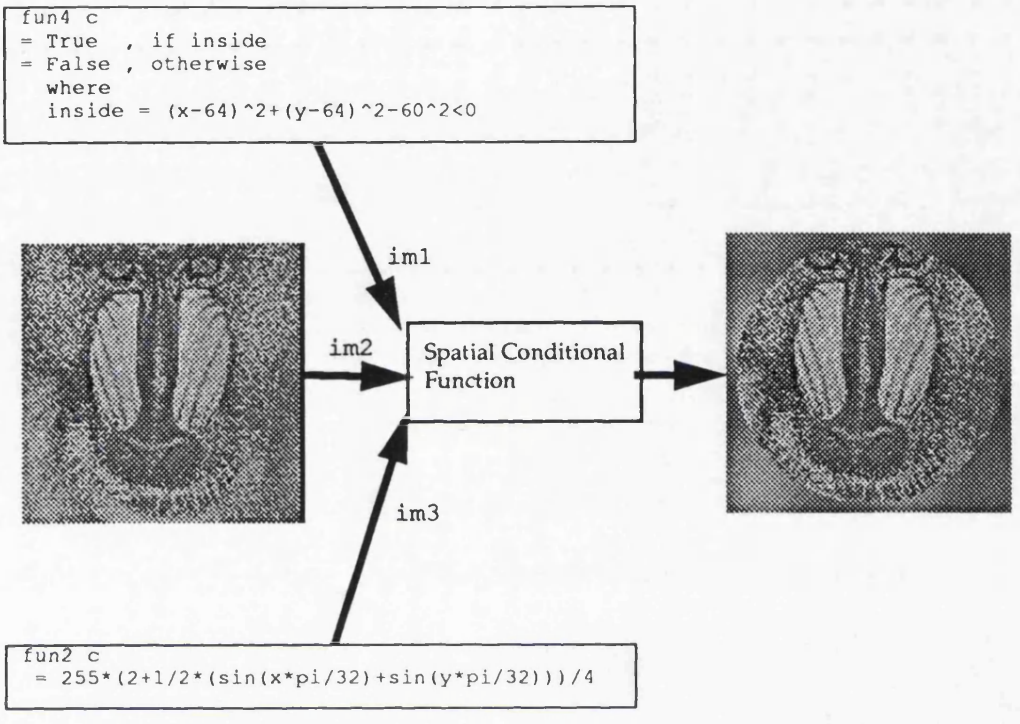


Figure 7-5 An example spatial conditional

7.5 Discussions

7.5.1 Applications of non-pixel images

In this chapter, we have demonstrated a method to combine pixel and non-pixel images as a potential of the utility of lazy functional languages. Because functional languages handle functions as first-class objects, an image expressed as a function can be handled without difficulty and processing such images can be written as function compositions. See the implementation of geometric transformations of `fImage` (Section 7.3.2). In addition, because of the laziness of the languages pixels are produced from a function image only when they are required. This property is of most benefit when a complicated series of functions are applied to images since no intermediate pixel images are produced.

However, how the technique of handling non-pixel images is useful is left as a subject for further research; the examples shown in this chapter are very primitive and just to show the possibility. One promising area would be computer graphics as discussed in Section 7.5.3,

because lazy functional languages can manipulate mathematical representations directly and eliminate unnecessary operations in rendering by conducting calculations at only required resolution. In particular, *texture mapping* [Heckbert86a] combines geometrically transformed pixel images and geometric objects; a task for which the method we have proposed could be applicable.

As found in a number of image processing textbooks, it is a general practice that many image processing algorithms are defined in the continuous domain using mathematical functions, then they are converted to the discrete domain and implemented to process pixel images. Lazy functional languages may potentially be useful to bridge this gap.

7.5.2 Integration of fImage and pImage

In the implementation described in this chapter `fImage` and `pImage` are separate, e.g. see the sub-definitions `im1`, `im2` and `im3` in `cim` on page 131. But ideally, these images should be integrated into one data type and individual image types should be hidden from application programmers. In Miranda this may be done by defining an algebraic data type for a general image which is a tagged union of either `fImage *` or `pImage`. In Haskell, this may be done by defining a type class for a general image whose instances are `fImage *` and `pImage`.

7.5.3 How to integrate multiple representations

Through the thesis, we have developed quite a few representations of an image. We started with a list of lists (Section 2.2.1) in the naive version. Then, based on the list of lists representation we attached various parameters, i.e. a list of lists with an origin (Section 3.2.2), with an origin and length (Section 5.3.1), with an origin and length represented as pairs of x and y elements (Section 5.4.1). We then adopted hierarchical data structures as the basic representation and implemented a binary tree of binary trees with a root position and first step length (Section 6.4.2), and a quadtree with a root position and first step length in 2D (Section 6.5.2).

What we have not discussed is how later versions can cope with algorithms defined for previous versions. The way we developed the representations is fairly ad-hoc, i.e. *to implement a new operation a new parameter is necessary, so attach it.* Therefore, for example, convolution cannot be applied to an image represented for rotation. In order to apply convolution defined in Section 3.5.1 to an image represented in Section 5.4.1, it will be necessary to check whether

the rows in the image are horizontal and whether the intervals of pixels and rows are equal to 1. That is, in the row data structure:

((x_origin,y_origin), (x_length,y_length), list_of_pixels)

it will be necessary to check whether *y_length* is 0 and *x_length* is equal to the length of *list_of_pixels.*, and also to check if the *y_origins* in the image form an arithmetic series with 1 as the increment. If they are, then conversion to a convolvable form would be straightforward. If not, an operation akin to 'display' will be required.

We have used 'display' as the demander of pixel evaluation since the original motivation came from laziness in displaying a transformed pixel image. But now it has turned out that 'display' is to convert an image representation to a form suitable for an operation, such as display. In fact, when an operation involving both pixel and non-pixel images is designed, these images are "displayed" to form 2D lists before the operation is carried out. In this case, a 2D list happens to be the most *naive* way (See Chapter 2) to combine the two kinds of images, and the display functions happen to be available to convert each representation to the common representation.

When an image processing library is to be designed using lazy functional languages, how to choose a good representation is an important issue. This is left as further work. One possible idea would be to use *multiple representations* or *isomorphic representations* [Poole92b] which carry a number of representations of an image as a form of a tuple, an algebraic data type, or a type class (e.g. in Haskell). A processing algorithm may be defined as a function composition of a conversion of representations and the algorithm itself. Because the system works lazily, until a processing path is specified by the user, no conversions take place. Once a processing path has been defined and input data has been made available, the process proceeds, where conversions from one representation to another take place only when necessary. In this way, it would be possible to integrate various representations.

A similar idea has been presented in [Parsons87a] for graphics applications, e.g. specification of coordinates in cartesian and polar systems and colours specified in any of the equivalent colour models.

Chapter 8: Real-Life Efficiency — Some Benchmarks and Possible Improvement

8.1 Introduction

We started the investigation by assuming that the efficiency of lazy functional languages is adequate for writing image processing applications (Section 1.3). The discussions have been carried out by relying on the fact that lazy functional languages are implemented so as to do what they promised to do. This kind of assumption is appropriate for most of the investigations in this thesis. However, people may ask, *"By the way, in reality, how fast are they, or how many (mega) bytes of memory do they use?"* This chapter answers this question.

Benchmarking is not a simple business. The purpose of comparison in this chapter is to give a rough idea of the performances of code written in three languages, namely Miranda, Haskell, and C. In this respect, this test may be quite different from comparing machine architectures or language implementations using the same source code, such as *SPEC* [SPEC92a] or *nofib* benchmarking [Partain93a]. There are a few approaches to this kind of benchmarking:

- Use the code which a typical programmer[1] would write in each language. This may result in different algorithms between languages.
- Carry out optimisation to have the best possible code in each language. This may also lead to different algorithms.
- Take a reasonable algorithm, e.g. simple, fast, etc., and implement the algorithm in each language.

We will take the first approach, i.e. what a typical programmer would write. This will also highlight how programmers would be likely to write inefficient code unconsciously, and how such code can be improved. All in all, the comparison will not at all be fair, but may be sufficient to give a rough order of efficiency in the three languages. The other interesting philosophical and technical arguments about benchmarking can be found both in image processing, e.g. [Uhr86a], and in lazy functional languages, e.g. [Partain93a].

We use the median filter as an example algorithm since it is very well-known and widely used in image processing. We compare execution speed and memory usage of the median filter written in Miranda, Haskell and C. Fortunately, by the time of writing, a few

1. Note that a programmer who is typical of one group (e.g. professional numerical analysts) may not be typical of another group (e.g. computer science graduates).

Haskell implementations have become available[2]. We use *GHC* [The AQUA Team93a] developed at University of Glasgow, since it is the latest implementation and currently said to be most efficient. After we compare the three versions, the cause of inefficiency is analysed and possible improvement of the C and Haskell versions is tried out.

8.2 An Unfair Competition

8.2.1 Contender No.1 — *Miranda*

The Miranda code for the median filter (Figure 8-1) is based on the one written in Section 3.5.4. This code has been discovered as a by-product of the general local neighbourhood function which was initially designed for convolution. Therefore, no efficiency issues have been considered. In fact, at a glance, the code looks rather inefficient since an entire list image, i.e. an image whose pixel is a list of pixels, is produced although there are a lot of common elements between adjacent pixel lists. In addition, as discussed in Section 7.2.2, very low-level conversion related to I/O between Miranda numbers and 32 bit integers has to be carried out in Miranda.

Differences from the code in Section 3.5 are listed as follows:

- The first two lines are the magic words to make Miranda code available as a command at the UNIX shell level.
- The second line is the *main* expression to be evaluated. The function `main` has been implemented to describe the "input→process→output" sequence. The header and the colour map are copied from the input to the output file. The pixel data part is converted to a list of numbers, then to the image data structure with an origin, processed by the function passed as an argument, converted back to a list of numbers and then to a list of characters for file output.
- The size of the input image is maintained. For the boundary pixels, a background value, 0 in this case, is filled in, i.e. centered zero boundary superposition (see Section 3.5.6).

2. For benchmarks using another implementation called hbc [Augustsson92a], see the paper [Kozato93a] attached in Appendix B.

```
#! /usr/local/bin/mira -exp
main "testimages/gantei.ras" (medianImage mask) "out.ras"

|| YK 28/09/92
|| median1.m
|| median filter for benchmarking (Chapter 8)
|| Identical to the definition in Chapter3

|| The "with an Origin" Version

interval * == (num,[*])
row *      == interval *
image *    == interval (row *)

|| rasterfile I/O

main::[char]->(image num->image num)->[char]->[sys_message]
main infile proc outfile
   = [Tofile outfile (header++cmap++data), Closefile outfile]
     where
     input          = read infile
     header         = take headerLength input
     cmap           = take cmapLength (drop headerLength input)
     cmapLength     = ras_maplength hdr
     hdr            = charToNumList header
     data           = (im2l.proc.l2im)
                      (drop (headerLength+cmapLength) input)
     l2im           = l12im.(splitList width).(map code)
     width          = ras_width hdr
     height         = ras_height hdr
     l12im l1       = fn (map fn l1) where fn x = (0,x)
     bgr            = rep width bgp
     bgp            = 0
     im2l (o,l)     = map decode ((concat (rep o bgr)++
                      concat (map flatten l)++
                      concat (rep (height-o-#l) bgr)))
     flatten (o,l) = (rep o bgp)++l++(rep (width-o-#l) bgp)

splitList::num->[*]->[[*]]
splitList n [] = []
splitList n l  = [take n l]++(splitList n (drop n l))

[ras_magic, ras_width,ras_height, ras_depth,
 ras_length,ras_type, ras_maptype,ras_maplength] = [(!i)|i<-[0..7]]

headerLength  = 32
ras_magic_num = 1504078485
rmt_equal_rgb = 1

charToNumList::[char]->[num]
charToNumList [] = []
charToNumList cs = fourCharToNum (take 4 cs):charToNumList (drop 4 cs)

fourCharToNum::[char]->num
fourCharToNum cs
   = foldl1 (+) (map2 (*) (map code cs) (map (256^) [3,2..0]))

|| pointwise operations

unaryInterval::(*->**)->interval *->interval **
unaryInterval f (o,p) = (o, map f p)

binaryInterval::(*->**->***)->interval *->interval **->interval ***
binaryInterval f (o1,p1) (o2,p2)
   = (o, map2 f (drop (o2-o1) p1) p2) , if o1 < o2
   = (o, map2 f p1 (drop (o1-o2) p2)) , otherwise
     where o = max2 o1 o2

unaryRow  = unaryInterval
binaryRow = binaryInterval

unaryPointwise::(*->**)->image *->image **
unaryPointwise f  = unaryRow (unaryRow f)

binaryPointwise::(*->**->***)->image *->image **->image ***
binaryPointwise f = binaryRow (binaryRow f)

|| image translation

translateInterval::num->interval *->interval *
translateInterval d (o,p) = (o+d,p)

translateRow = translateInterval

|| convolution

conv1 mul add mask im
   = accum [prod (element mask p) (shiftInterval p) | p<-domain mask]
     where
     accum  = foldl1 add
     prod x = unaryInterval (mul x)
     shiftInterval p
            = translateInterval (p-((length mask) div 2)) im

domain    = index.snd
length    = (#).snd
element   = (!).snd

|| median filter

mask = makeMask 3

makeMask::num->image num
makeMask n = fn (map fn (rep n (rep n 0))) where fn x = (0,x)

medianImage mask = (unaryPointwise median).(localHistImage mask)
median list = (sort list)!(#list div 2)

localHistImage = conv1 (localHistRow) (binaryPointwise (++))
localHistRow = conv1 f (binaryRow (++)) where f a b = [b]
```

Figure 8-1 Miranda code for median filtering

8.2.2 Contender No.2 — *Haskell* (Version 1)

The Haskell code (Figure 8-2) has been produced by filtering the Miranda code of Figure 8-1 through the shell script *mira2hs* written by Howe [Howe92a], then modified by hand to implement the main routine. This means that this Haskell code has not been written to utilise the facilities provided by Haskell. For example, arrays in Haskell could have been used instead of lists, which might have improved the efficiency. In addition, the low-level conversion between 32 bit integers and `Int`s is done in Haskell. If the facility such as I/O monads [Hall92a] which allow C functions to deal with low-level jobs without violating referential transparency had been used, the efficiency might have been improved.

Changes made are:

- The `main` function has been added, which is of the type `Dialogue` defined as:

```
type Dialogue = [Response] -> [Request]
```

- The `ip` function forms the main part of the job, which takes an input character list, an image processing function, and returns an output list of characters. This is akin to the `main` function defined in the Miranda version.

- The `splitList` function has been changed to utilise the library functions, `chopList` and `splitAt`, in the hope that library functions could be more efficient than user-defined functions. For `chopList`, see [Augustsson92b]. For this purpose the module `ListUtil` is imported. Also, we use quick sort defined in the library to sort a list of pixels. The module `QSort` is imported for this purpose.

- Two functions (`rep` and `subscripts`) defined in *mira2hs* are used, since these are defined in Miranda but not in Haskell. `subscripts` is equivalent to `index` in Miranda.

Otherwise, algorithms are identical to the Miranda version. It is noticeable how much Haskell inherits Miranda's facilities. The code is compiled by the ghc compiler version 0.19 using the "`-syslib hbc -O2`" flag.

```
-- YK 28/09/92
-- Median1.hs
module Main (main) where

import ListUtil
import QSort

main :: Dialogue
main resps
   = [ReadFile "testimages/gantei.ras",
      WriteFile "out1.ras"
        (case resps!!0 of
         Str contents -> ip contents (medianImage mask)
         Failure ioe  -> "Error")]

-- The "with an Origin" Version
type Interval a = (Int,[a])
type Row a      = Interval a
type Image a    = Interval (Row a)

-- rasterfile I/O
ip::[Char]->(Image Int->Image Int)->[Char]
ip input proc
   = header++cmap++pixl
     where header = take headerLength input
           cmap   = take cmleng (drop headerLength input)
           cmleng = ras_maplength hdr
           hdr    = charToNumList header
           pixl   = (im2l.proc.l2im) (drop (headerLength+cmleng) input)
           l2im   = l12im.(splitList width).(map ord)
           width  = ras_width hdr
           height = ras_height hdr
           l12im l1 = fn (map fn l1) where fn x = (0,x)
           bgr    = rep width bgp
           bgp    = 0
           im2l (o,l) = (map chr . concat) (rep o bgr ++
                                map flatten l ++ rep (height-o- length l) bgr)
           flatten (o,l) = (rep o bgp)++l++(rep (width-o- length l) bgp)

splitList::Int->[a]->[[a]]
splitList n l = chopList (splitAt n) l

[ras_magic, ras_width,ras_height, ras_depth,
 ras_length,ras_type, ras_maptype,ras_maplength] = [(!!i)|i<-[0..7]]

headerLength  = 32
ras_magic_num = 1504078485
rmt_equal_rgb = 1

charToNumList::[Char]->[Int]
charToNumList [] = []
charToNumList cs = fourCharToNum (take 4 cs):charToNumList (drop 4 cs)

fourCharToNum::[Char]->Int
fourCharToNum cs
   = foldl1 (+) (zipWith (*) (map ord cs) (map (256^) [3,2..0]))

numToCharList::[Int]->[Char]
numToCharList []     = []
numToCharList (n:ns) = numToFourChar n++numToCharList ns

numToFourChar::Int->[Char]
numToFourChar n = map (chr.(`mod` 256).(n `div`)) (map (256^) [3,2..0])

-- pointwise operations
unaryInterval::(a->aa)->Interval a->Interval aa
unaryInterval f (o,p) = (o, map f p)

binaryInterval::(a->aa->aaa)->Interval a->Interval aa->Interval aaa
binaryInterval f (o1,p1) (o2,p2)
   | (o1 < o2) = (o, zipWith f (drop (o2-o1) p1) p2)
   | otherwise = (o, zipWith f p1 (drop (o1-o2) p2))
     where o = max o1 o2

unaryRow = unaryInterval
binaryRow = binaryInterval

unaryPointwise::(a->aa)->Image a->Image aa
unaryPointwise f  = unaryRow (unaryRow f)
binaryPointwise::(a->aa->aaa)->Image a->Image aa->Image aaa
binaryPointwise f = binaryRow (binaryRow f)

-- image translation
translateInterval::Int->Interval a->Interval a
translateInterval d (o,p) = (o+d,p)
translateRow = translateInterval

-- convolution
conv1 mul add mask im
   = accum [prod (element mask p) (shiftInterval p) | p<-domain mask]
     where
     accum  = foldl1 add
     prod x = unaryInterval (mul x)
     shiftInterval p = translateInterval (p-((leng mask) `div` 2)) im
domain    = subscripts.snd
leng      = length.snd
element   = (!!).snd

-- median filter
mask = makeMask 3
makeMask::Int->Image Int
makeMask n = fn (map fn (rep n (rep n 0))) where fn x = (0,x)

medianImage mask = (unaryPointwise median).(localHistImage mask)
median list = (sort list)!!((length list) `div` 2)

localHistImage = conv1 (localHistRow) (binaryPointwise (++))
localHistRow = conv1 f (binaryRow (++)) where f a b = [b]

rep :: Int -> b -> [b]
rep n x = take n (repeat x)

subscripts :: [a] -> [Int] -- Miranda index
subscripts xs = f xs 0
                where f []     n = []
                      f (_:xs) n = n : f xs (n+1)
```

Figure 8-2 Haskell code for median filtering (Version 1)

8.2.3 Contender No.3 — *C* (median-by-qsort)

The C code for median filtering shown in Figure 8-3 has been freshly written for this benchmarking purpose. The first two and a half columns in the source list are the main routine and the code for file I/O and various house-keeping. The median algorithm is implemented in the function `med` on page 143. The fundamental algorithm is to generate a local histogram within `radius`, sort the local histogram using quick sort defined in the library (hence, we call this version *median-by-qsort*), and get the `mid`'th element from the histogram. So, the basic median algorithm is relatively similar to the Miranda and Haskell versions. The size of input image is maintained and 0's are filled in the image boundary[3]. The program is compiled with the "`-O4`" option to optimise at the maximum level.

3. This is done by the `memset` function!

```
/*
 * median-by-qsort.c
 * Median filter using quick sort in the library
 * - for benchmark in Chapter 8
 */

#include <stdio.h>
#include <fcntl.h>
#include <rasterfile.h>
#include <malloc.h>

#define BGP 0

char *progname;

main(argc, argv)
     int argc;
     char *argv[];
{
  struct rasterfile ras;
  void usage(), writeheader(), writecmap(), writedata();
  int rasteropen(), outopen(), checkheader();
  char *inputfilename();
  struct rasterfile readheader();
  unsigned char *readcmap(), *readdata(), *med();
  int ifd, ofd, masksize;
  unsigned char *data, *cmap, *outdata;

  progname = argv[0];

  ifd = rasteropen("testimages/gantei.ras");
  ofd = outopen("out.ras");
  masksize = 3;

  ras = readheader(ifd);
  writeheader(ofd, ras);
  if (ras.ras_maptype != RMT_NONE)
    {
      cmap = readcmap(ifd, ras);
      writecmap(ofd, cmap, ras);
    }
  data = readdata(ifd, ras);

  outdata = med(data, ras.ras_width, ras.ras_height, masksize);
  writedata(ofd, outdata, ras);
  close(ofd);
  exit(0);
}

void
fatal(msg)
char *msg;
{
  fprintf(stderr, "%s: %s\n", progname, msg);
  exit(1);
}

int
rasteropen(filename)
char *filename;
{
  int fd;

  if ((fd = open(filename, O_RDONLY)) == -1) fatal("cannot open
rasterfile");
  return fd;
}

int
outopen(filename)
char *filename;
{
  int fd;

  if ((fd = open(filename, O_CREAT|O_RDWR,0644))==-1)
    fatal("cannot open output file");
  return fd;
}

struct rasterfile
readheader(fd)
int fd;
{
  struct rasterfile ras;

  if (read(fd, &ras, sizeof(ras)) != sizeof(ras))
    fatal("cannot read header");
  return ras;
}

void
writeheader(fd, ras)
int fd;
struct rasterfile ras;
{
  if (write(fd, ras, sizeof(ras)) != sizeof(ras))
    fatal("cannot write header");
}

unsigned char *
readcmap(fd, ras)
int fd;
struct rasterfile *ras;
{
  unsigned char *cmap;

  cmap = (unsigned char *)malloc(ras->ras_maplength);
  read(fd, (char *)cmap, ras->ras_maplength);
  return cmap;
}

void
writecmap(fd, cmap, ras)
int fd;
unsigned char *cmap;
struct rasterfile *ras;
```

Figure 8-3 C code for median filtering

```
{
  if (write(fd, cmap, ras->ras_maplength) != ras->ras_maplength)
    fatal("cannot write colour map");
}

unsigned char *
readdata(fd, ras)
int fd;
struct rasterfile *ras;
{
  unsigned char *data;

  data = (unsigned char *)malloc(ras->ras_length);
  read(fd, (char *)data, ras->ras_length);
  return data;
}

void
writedata(fd, data, ras)
int fd;
unsigned char *data;
struct rasterfile *ras;
{
  if (write(fd, data, ras->ras_length) != ras->ras_length)
    fatal("cannot write data");
}

/*****************************************************************************/
#include <memory.h>

static int
bytecompare(i,j)
unsigned char *i, *j;
{
  return(*i - *j);
}

unsigned char *
med(in, xsize, ysize, msize)
unsigned char *in;
int xsize, ysize, msize;
{
  unsigned char *out = (unsigned char *) malloc(xsize * ysize);

  /* fill in edge pixels with background value */
  {
    memset((char*)out, BGP, xsize*ysize); /* what the heck - do the lot! */
  }

  /* now do the medians */
  {
    int nbh_area = msize*msize;
    /* buffer to ease sorting */
    unsigned char *buf = (unsigned char *)malloc(nbh_area);
    int radius = msize / 2;
    int x, y;
    unsigned char *median_location = buf + nbh_area/2;
    for (y = radius; y < ysize-radius; y++){
      for (x = radius; x < xsize-radius; x++){
        int offset = x + xsize*y;
        int i, j;
        for (i = -radius; i <= radius; i++)
          for (j = -radius; j <= radius; j++)
            *buf++ = in[offset+i+j*xsize];
        buf -= nbh_area;
        qsort((char*)buf,nbh_area,sizeof(unsigned char),bytecompare);
        out[offset] = *median_location;
      }
    }
    free(buf);
  }
  return out;
}
```

Figure 8-3 C code for median filtering - continued

8.2.4 Result

The three versions are made into commands at the UNIX shell level, and to measure the speed and memory use the c-shell built-in command "`time`" is used. The three programs ran on a Sun Microsystems' SPARCstation2 with 32MB of main memory, running SunOS 4.1.3, and all files on local disk. Table 8-1 shows a result of the test using an 8 bit gray level image of the size 256×256 with the median mask of the size 3×3. Slow-down ratios indicate how many times Miranda and Haskell are slower assuming the speed of C to be 1. We use elapsed time to calculate the slow-down ratios, since we are interested in the total time including CPU time and other overhead, e.g. time spent waiting for I/O or paging. Since elapsed time also includes time consumed by other processes, it is important to measure these timings on a machine which is otherwise idle. Note also that times vary between runs, but we used the result of runs that fall within the typical cases.

Table 8-1 A benchmark result

Language	Time in seconds[a]			Slow-down ratio	Maximum real
	elapsed[b]	user	system	using elapsed time	memory used in kB[c]
Miranda[d]	382	376.9	1.6	127	9192
Haskell	44	43.3	0.5	15	4608
C	3	3.4	0.1	1	424

a. The time command gives elapsed time, CPU times devoted to the user's process and consumed by the kernel by providing the tags "%E %U %S" via the "time" shell variable (see the manual pages for csh on the SunOS system).
b. Note that user and system time are rounded to the nearest 0.1 s by the time command, and elapsed time to the nearest second. This can lead to apparent anomalies, such as the elapsed time being reported as less than the user + system times.
c. The maximum real memory is given in units of system pages (8kB) by providing the tag "%M" to the time command.
d. Pre-compilation of the Miranda program improves the performance slightly. It gives: elapsed - 369, user - 363.4, system - 1.7, and memory - 9144.

As mentioned, the competition may not be regarded as fair for a number of reasons: the Miranda code was not intended to be a median filter but only a by-product of convolution without consideration of efficiency; the Haskell code was converted from Miranda with very little knowledge of Haskell programming; each version uses a different sort function defined in each library and its implementation is not known, and so forth. However, despite all the unfairness, the result gives an idea of the speed and memory use that average programmers could expect. The Miranda version is more than a hundred times slower than C and is too slow

for image processing applications. In contrast, Haskell is only 15 times slower than C. The Miranda implementation is very stable and reliable, but is now at least a few years old, while the development of Haskell is currently an on-going project and various ideas for improvement are being tested [Hall92b, Peyton Jones92b]. Therefore, we can expect more speed-up in the Haskell version. More discussions on performance are given in Section 8.6

In the following two sections, we try a few improvements on the C and Haskell versions to get better performance.

8.3 Improvement of the C Code

8.3.1 Replacing quick sort with insertion sort (median-by-sort)

We used the library function `qsort` to sort local histograms, but this involves the overhead of a function call for each comparison during the sort. When comparing bytes, this overhead is likely to be significant, so we replaced qsort with an insertion sort for bytes — see Figure 8-4. (Insertion sort is simple and one of the most efficient algorithms for sorting small lists like this, i.e. lists of 9 elements or so. See, for example, [Knuth73a]). We call this revised algorithm *median-by-sort*. The code for median filtering is otherwise identical to the one given in Figure 8-3.

```
/* median-by-sort */
#include <memory.h>

/* qsort is too slow ... */
void
sort(v,n)
unsigned char v[];
int n;
{
  /* insertion sort */
  int i;
  for (i = 1; i < n; i++){
      unsigned char x = v[i];
      int j;
      for (j = i-1; (j >= 0) && (v[j] > x); j--)
          v[j+1] = v[j];
      v[j+1] = x;
  }
}
```

Figure 8-4 Median by sort in C

8.3.2 Incremental updating of local histograms (median-incremental)

The above two algorithms (median-by-qsort and median-by-sort) update an entire local histogram for each position. However, there are overlapped pixel values between

neighbouring local histograms, e.g., for a 3×3 median mask 6 pixels are common between two adjacent positions. So, it is possible not to replace the entire histogram, but to update the histogram incrementally in scanning the input image. In addition, in updating the histogram the median for the previous pixel can be kept and used for searching a new median pixel. We call this new algorithm *median-incremental* coded as shown in Figure 8-5.

```
/* median-incremental */
#include <memory.h>

unsigned char *
med(in, xsize, ysize, msize)
unsigned char *in;
int xsize, ysize, msize;
{
  unsigned char *out = (unsigned char *) malloc(xsize * ysize);
  int y;
  int radius = msize / 2;

  /* fill in edge pixels with background value */
  {
  memset((char*)out, BGP, xsize*ysize);  /* what the heck - do the lot! */
  }

  /* now do the medians */
  for (y = radius; y < ysize-radius; y++){
  int hist[256];
  int m;
  int s;        /* sum of hist[0..m] inclusive */
  int offset = xsize*y; /* position of centre of mask */
  int mid = (msize*msize+1)/2;
  int i,j;
  /* first build the histogram etc */
  for (i = 0; i < 256; i++)
      hist[i] = 0;
  for (j = -radius; j <= radius; j++)
      for (i = -radius; i <= radius; i++)
          hist[in[offset+i+j*xsize]]++;
  /* find the median */
  s = 0;
  m = -1;
  do {
      m++;
      s += hist[m];
  } while (s < mid);
  /* now m = median */
  out[offset] = m;

  /* now scan along the row, incrementally updating */
  for (j = radius+1; j < xsize-radius; j++){
      /* update histogram */
      offset -= radius;
      for (i = -radius; i <= radius; i++){
          int x = in[offset+i*xsize];
          hist[x]--;
          if (x <= m)
              s--;
      }
      offset += 2*radius + 1;
      for (i = -radius; i <= radius; i++){
          int x = in[offset+i*xsize];
          hist[x]++;
          if (x <= m)
              s++;
      }
      offset -= radius;
      /* adjust m to be median again */
      while (s-hist[m] >= mid){
          s -= hist[m];
          m--;
      }
      while (s < mid){
          m++;
          s += hist[m];
      }
      /* now m = median */
      out[offset] = m;
  }
  }
  return out;
}
```

Figure 8-5 Median by incremental histogram updating

8.3.3 Improved result

A result obtained by these improvements is shown in Table 8-1. We achieved execution time of 1 second by median-by-sort, and less than 1 second by median-incremental. Although these are quite remarkable achievements, it should be noted that in order to improve the performance of programs better algorithms should be used, and this optimisation should normally be done by hand. This requires extra programming effort.

Table 8-2 A benchmark result of the improved C versions

Algorithms	Time in seconds			Maximum real memory in kB
	elapsed[a]	user	system	
median-by-qsort	3	3.4	0.1	424
median-by-sort	1	1.3	0.1	424
median-incremental	0	0.6	0.0	424

a. As described in Table 8-1, elapsed time is rounded to the nearest second and may be less than the sum of user and system times.

8.4 Improvement of the Haskell Code

In this section, we conduct careful examination of the Haskell code and try possible improvement to get better performance. For the detailed description of the analysis and improvement using the heap profiling tool [Runciman92a, Runciman93a] that comes with hbc [Augustsson92a], see Appendix B.

8.4.1 Attaching 'length' to the interval structure (Version 2)

In the Haskell code shown in Figure 8-2 (Version 1), there is the `ip` function which is responsible for the whole sequence from file input, image processing, to file output, and the following code is in the subdefinitions of `ip`:

```
im2l (o,l) = (map chr . concat) (rep o bgr ++ map flatten l ++
                                 rep (height-o- length l) bgr)
flatten (o,l) = (rep o bgp)++l++(rep (width-o- length l) bgp)
```

Input and output streams are long lists of characters, but image processing is defined on a 2D interval structure, a pair of an origin and a list of elements (Section 3.2.2). `im2l` is designed to convert a 2D interval structure to a plain list, and `flatten` converts a 1D interval to a plain

list. The need for these functions comes from the intention to have equal sizes of input and output, so that the results of the three versions can be made identical. Thus, in the original implementation in Chapter 3, this issue was not considered at all. In the `im2l` function, there is the code fragment "`length l`". Usually `length` itself consumes a list but does not keep the whole list. But in the above code, because the list '`l`' is shared between `flatten` and `length`, and the `length` function is not evaluated until the end, the content of the whole list is kept until `length` is evaluated. And here, this '`l`' is *the whole image!*

As discussed in Chapter 23 of [Peyton Jones87a] and in [Hughes84a], there is a lot of subtlety in behaviour of lazy functional programs. Our code may be a good example of *space leaks* caused by the scheduling problem. Because the Haskell code handles an image as a list of lists of pixels and lists are treated lazily through input, process and output, and the process is a simple local neighbourhood operation that does not rely on data at distant positions, it should not cause any space leaks. However, as shown in Appendix B., our code had a space leak for the above reason.

The code has been modified to overcome the problem of accumulating the whole image. The interval structure has been modified to become a triple of (origin, length, list). The length was considered to be redundant in the original code in Chapter 3, but now it seems a good idea to add the new member to the data structure. The length of a row and the length of an image are provided as `width` and `height` respectively in the header information of a rasterfile. So, there is almost no overhead in obtaining this information. The new interval structure in Haskell is:

```
type Interval a = (Int,Int,[a])
```

The subdefinitions for the improved `ip` function are shown as follows:

```
ll2im ll = (0,height,map fn ll) where fn x = (0,width,x)
im2l (o,ln,l) = (map chr . concat) (rep o bgr ++ map flatten l ++
                                        rep (height-o- ln) bgr)
flatten (o,ln,l) = (rep o bgp)++l++(rep (width-o- ln) bgp)
```

The other functions which need modification are the unary and binary pointwise operations, and the translation of intervals:

```
unaryInterval f (o,ln,p) = (o,ln, map f p)
binaryInterval f (o1,ln1,p1) (o2,ln2,p2)
   | (o1 < o2) = (o, ln, zipWith f (drop (o2-o1) p1) p2)
   | otherwise = (o, ln, zipWith f p1 (drop (o1-o2) p2))
     where o = max o1 o2
           ln = max 0 ((min (o1+ln1) (o2+ln2)) - o)
translateInterval d (o,ln,p) = (o+d,ln,p)
```

Also modifications of `conv1` and `makeMask` are necessary, as well as related functions such as `domain`, `element`, etc. A new definition of these functions is as follows:

```
conv1 mul add mask im
   = accum [prod (element mask p) (shiftInterval p) | p<-domain mask]
     where
     accum  = foldl1 add
     prod x = unaryInterval (mul x)
     shiftInterval p = translateInterval (p-((second mask) `div` 2)) im

domain         = subscripts.third
element        = (!!).third
second (a,b,c) = b
third (a,b,c)  = c
makeMask n     = fn (map fn (rep n (rep n 0))) where fn x = (0,n,x)
```

In any of the above functions, it can be said that the modifications are relatively trivial. There is no change in the fundamental algorithm, but just addition of an extra parameter to save calculations.

8.4.2 Eliminate identical operations (Version 3)

The basic algorithm of our median filtering is to generate a list image, in which the length of each pixel list is equal to the square of the mask size. Then, the median of each pixel (list) is taken as a unary pointwise operation. Therefore, the function to take a median is the same all through the list image, but a compiler does not spot this fact. Lazy evaluation shares the same expressions only when these occur in the same function call. In our original code (Version 1) the index of a median is calculated every time, e.g. 65536 times for a 256×256 image. The following is the original code to take a median:

```
median list = (sort list)!!((length list) `div` 2)
```

where the length does not change, e.g. if a median mask is 3×3 the index, i.e. "`(length list) `div` 2`", is 4 and is constant all through the operations. Utilising this knowledge, it is

possible to modify the code, so that the index is calculated only once before the median function is called. The new definition follows:

```
medianImage n
   = (unaryPointwise (rankFilter m)).(localHistImage (makeMask n))
     where m = n*n `div` 2
rankFilter m list = (sort list)!!m
```

where n is the mask size. Since the new function takes a rank order as a parameter, it works as a general rank filter rather than only a median. Hence, the name has been changed.

8.4.3 Whether to fold up a list from left or right? (Version 4)

We defined the higher-order convolution function (`conv1`) to take multiplicative and additive operations as parameters and accumulation is defined within the function as "`foldl add`". As discussed in Chapter 6 of [Bird88a], fold operations behave very subtly; for functions, such as (+) or (*), that are strict in both arguments and can be computed in constant time and space, `foldl` is more efficient. Whereas for functions, such as (&) or (++), that are non-strict in some argument, `foldr` is often more efficient. Therefore, it may be a mistake to hard-code the direction of accumulation within the function definition of convolution.

Based on the above consideration, the new definition of `conv1` below takes an accumulation instead of an additive operation. Since an append operator (++) is passed as an argument in order to produce a list image, accumulate from the right should be more efficient. The modified code is the following:

```
conv1 mul accum mask im
   = accum [prod (element mask p) (shiftInterval p) | p<-domain mask]
     where
     prod x = unaryInterval (mul x)
     shiftInterval p = translateInterval (p-((second mask) `div` 2)) im

localHistImage = conv1 f accum
                 where f     = localHistRow
                       accum = foldr1 (binaryPointwise (++))

localHistRow   = conv1 f accum
                 where f a b = [b]
                       accum = foldr1 (binaryRow (++))
```

8.4.4 Use of cons instead of append (Version 5)

It is generally quicker to use cons (:) instead of append (++) to attach an element to a list, because in order to append two lists, the one in front should be traversed. In the median filtering it is possible to use cons in the 1D local histogramming operation. The modified function is:

```
localHistRow
   = conv3 f accum
     where f a b        = b
           accum (r:rs) = foldr (binaryRow (:)) (unaryRow (:[]) r) rs
```

8.4.5 Insertion sort instead of quick sort (Version 6)

As we discussed in optimising C code (Section 8.3.1), insertion sort is normally regarded as one of the best choices for sorting small lists like this. We implemented insertion sort in Haskell as follows and replaced the library quick sort with this insertion sort:

```
-- insertion sort
sort                :: Ord a => [a] -> [a]
sort []              = []
sort (x:ys)          = insert x (sort ys)

insert              :: Ord a => a -> [a] -> [a]
insert x []          = [x]
insert x ys@(y:ys')
        | x <= y     = x:ys
        | otherwise  = y:insert x ys'
```

The entire code of Version 6 is given in Figure 8-6.

```
module Main (main) where
import ListUtil

-- insertion sort
sort                :: Ord a => [a] -> [a]
sort []              = []
sort (x:ys)          = insert x (sort ys)
insert              :: Ord a => a -> [a] -> [a]
insert x []          = [x]
insert x ys@(y:ys')
       | x <= y      = x:ys
       | otherwise   = y:insert x ys'

main :: Dialogue
main resps
   = [ReadFile "gantei.ras",
      WriteFile "out2.ras"
        (case resps!!0 of
         Str contents -> ip (medianImage 3) contents
         Failure ioe  -> "Error")]

-- The "with an Origin" Version
type Interval a = (Int,Int,[a])
type Row a      = Interval a
type Image a    = Interval (Row a)

-- rasterfile I/O
ip::(Image Int->Image Int)->[Char]->[Char]
ip proc input
   = header++cmap++((im2l.proc.l2im) (drop (headerLength+cmleng) input))
     where
     header = take headerLength input
     cmap   = take cmleng (drop headerLength input)
     cmleng = ras_maplength hdr
     hdr    = charToNumList header
     l2im   = l12im.(splitList width).(map ord)
     width  = ras_width hdr
     height = ras_height hdr
     l12im l1 = (0,height,map fn l1) where fn x = (0,width,x)
     bgr    = rep width bgp
     bgp    = 0
     im2l (o,ln,l) = (map chr . concat) (rep o bgr ++ map flatten l ++
                                         rep (height-o- ln) bgr)
     flatten (o,ln,l) = (rep o bgp)++l++(rep (width-o- ln) bgp)

splitList::Int->[a]->[[a]]
splitList n l = chopList (splitAt n) l

[ras_magic, ras_width,ras_height, ras_depth,
 ras_length,ras_type, ras_maptype,ras_maplength] = [(!!i)|i<-[0..7]]
headerLength  = 32
ras_magic_num = 1504078485
rmt_equal_rgb = 1
charToNumList::[Char]->[Int]
charToNumList [] = []
charToNumList cs = fourCharToNum (take 4 cs):charToNumList (drop 4 cs)
fourCharToNum::[Char]->Int
fourCharToNum cs=foldl1 (+) (zipWith (*) (map ord cs)(map (256^) [3,2..0]))

numToCharList::[Int]->[Char]
numToCharList []     = []
numToCharList (n:ns) = numToFourChar n++numToCharList ns
numToFourChar::Int->[Char]
numToFourChar n = map (chr.(`mod` 256).(n `div`)) (map (256^) [3,2..0])

-- pointwise operations
unaryInterval::(a->aa)->Interval a->Interval aa
unaryInterval f (o,ln,p) = (o,ln, map f p)
binaryInterval::(a->aa->aaa)->Interval a->Interval aa->Interval aaa
binaryInterval f (o1,ln1,p1) (o2,ln2,p2)
   | (o1 < o2) = (o, ln, zipWith f (drop (o2-o1) p1) p2)
   | otherwise = (o, ln, zipWith f p1 (drop (o1-o2) p2))
     where o = max o1 o2
           ln = max 0 ((min (o1+ln1) (o2+ln2)) - o)
unaryRow = unaryInterval
binaryRow = binaryInterval
unaryPointwise::(a->aa)->Image a->Image aa
unaryPointwise f  = unaryRow (unaryRow f)
binaryPointwise::(a->aa->aaa)->Image a->Image aa->Image aaa
binaryPointwise f = binaryRow (binaryRow f)

-- image translation
translateInterval::Int->Interval a->Interval a
translateInterval d (o,ln,p) = (o+d,ln,p)
translateRow = translateInterval

-- convolution
conv3 mul accum mask im
   = accum [prod (element mask p) (shiftInterval p) | p<-domain mask]
     where
     prod x = unaryInterval (mul x)
     shiftInterval p = translateInterval (p-((second mask) `div` 2)) im
domain    = subscripts.third
element   = (!!).third
second (a,b,c) = b
third (a,b,c)  = c

-- median filter
makeMask::Int->Image Int
makeMask n = fn (map fn (rep n (rep n 0))) where fn x = (0,n,x)
medianImage n = (unaryPointwise (median m)).(localHistImage (makeMask n))
                where m = n*n `div` 2
median m list = (sort list)!!m
localHistImage = conv3 f accum
                 where f     = localHistRow
                       accum = foldr1 (binaryPointwise (++))
localHistRow = conv3 f accum
               where
               f a b       = b
               accum (r:rs) = foldr (binaryRow (:)) (unaryRow (:[]) r) rs

rep :: Int -> b -> [b]
rep n x = take n (repeat x)
subscripts :: [a] -> [Int]      -- Miranda index
subscripts xs = f xs 0
                where f []     n = []
                      f (_:xs) n = n : f xs (n+1)
```

Figure 8-6 Haskell code for median filtering (Version 6)

8.4.6 Summary of the improvement

The following Table 8-3 summarises the result of the improvement. The condition of the runs is the same as the one described in Section 8.2.4. Through the various improvement of the Haskell code, Version 6 achieved 32 seconds in executing median filter operation on an image of 256×256 and this is a 27% speed up from Version 1. As for memory usage, these modifications did not improve the performance. The biggest contribution to execution time was from changing sorting algorithm to use insertion sort. (The cause of the speed-up was similar to the reason that the C version speeded up when the library qsort was replaced by an insertion sort. In both cases, the library functions could not inline the comparisons. Optimising compilers with access to the libraries could remove this overhead in both cases.) The other tweaks did not improve the performance very much. In particular, using cons instead of append (Version 5) even worsened the execution time, which was contradictory to our expectation. By using hbc, however, this modification made a slight improvement as described in Appendix B., and so there may be unknown overhead in using cons in GHC. More investigation is needed.

Table 8-3 Improved results of Haskell versions

Versions	Time in seconds			Maximum real memory in kB
	elapsed	user	system	
Version 1 (original)	44	43.3	0.5	4608
Version 2 (eliminate calculation of length)	41	40.1	0.6	4608
Version 3 (eliminate index calculation)	40	39.8	0.5	4608
Version 4 (accumulate from the right)	39	38.5	0.5	4608
Version 5 (Use (:) instead of (++))	40	39.0	0.6	4608
Version 6 (insertion sort instead of quick sort)	32	31.6	0.7	4608

8.5 Further Benchmarks — I/O and Larger Images

To analyse the performance further, we conducted a few more tests; I/O-only versions in Haskell to test how much time is spent for I/O, and the memory and timing to process larger images both in Haskell and C.

8.5.1 How much time is consumed by I/O?

As a source of unfairness of the benchmark tests, we referred to I/O in the functional versions (Sections 8.2.1 and 8.2.2). However, we have not tested how much of the execution time is spent for I/O which includes low level conversions between 32 bit integers and numbers used in the functional programs.

Based on the Haskell's Version 1 and Version 6 (though versions from 2 to 6 will give the same result since the I/O-only versions are identical for these), we replaced the image processing function with an identity function to see how much time is used for I/O, and the following is the result of a set of runs:

Table 8-4 I/O overheads of the Haskell versions

Versions	Time in seconds			Maximum real memory in kB
	elapsed	user	system	
Version 1	44	43.3	0.5	4608
Version 1 (I/O only)	5	5.1	0.6	4592
Version 6	32	31.6	0.7	4608
Version 6 (I/O only)	5	4.3	0.6	4600

The result shows that about 5 seconds (≅ 11% for Version1, and ≅ 16% for Version 6) is spent for I/O which could be improved if a better I/O scheme was taken.

8.5.2 Space and time behaviour for larger images

Because we eliminated the source of space leaks in Section 8.4.1, the improved Haskell version should run in constant space even if the size of an image gets very large. Whereas in the C version, it is obvious from the code that the amount of memory increases as the size of an

image increases, since we only allocates two buffers for input and output. The purpose of this test is to investigate how space and time behaviour changes as the image size increases.

We use the C's median-incremental and the Haskell's versions 5 and 6 for this test. We use 8 bit integer images whose sizes are 256×256, 512×512, 1024×1024, 2048×2048 and 4096×4096. The test condition is the same as Section 8.2.4, except that image files are not on the local disk due to disk space shortage. Following Table 8-5 shows a result, in which "Time" indicates elapsed time and "Memory" indicates maximum real memory used during execution of the process, both from c-shell's built-in `time` command.

Table 8-5 Results for larger images

		256×256	512×512	1024×1024	2048×2048	4096×4096
C median-incremental	Time (sec)	1[a]	4	20	81	347
	Memory (kB)	416	696	2232	8376	26968
Haskell (Version 5)	Time (sec)	40	159	660	2738	11513
	Memory (kB)	4624	4608	4616	4632	4696
Haskell (Version 6)	Time (sec)	34	132	541	2269	9622
	Memory (kB)	4624	4608	4616	4632	4672

a. The elapsed time of C's median-incremental algorithm differs from that shown in Table 8-2. This is probably due to a combination of rounding of the time command as described in Table 8-1, and I/O effects. In particular, the timing in Table 8-2 was measured using local disk only, while this was done using remote disk for the data files.

Following Figure 8-7 and Figure 8-8 show time and space behaviour respectively of C and Haskell based on the above result. Both in C and Haskell, execution time increases in proportion to the number of pixels and the ratio between them does not vary very much, i.e. Haskell is approximately 30 times slower. As for space, C consumes larger space as the image size increases. While in the Haskell versions, space consumption is more or less constant even if the image size gets very large. Up to a certain size (somewhere between 1024×1024 and 2048×2048) C is more space efficient than Haskell, but above that Haskell becomes more space efficient. Moreover, the C version crashes due to out of swap space when the image size exceeds a certain limit[4]. To make the C version run, modification of the program will be

4. In fact, this happened when we processed an image of 8192×8192.

inevitable. In contrast, the Haskell version will continue to work although it is slow.

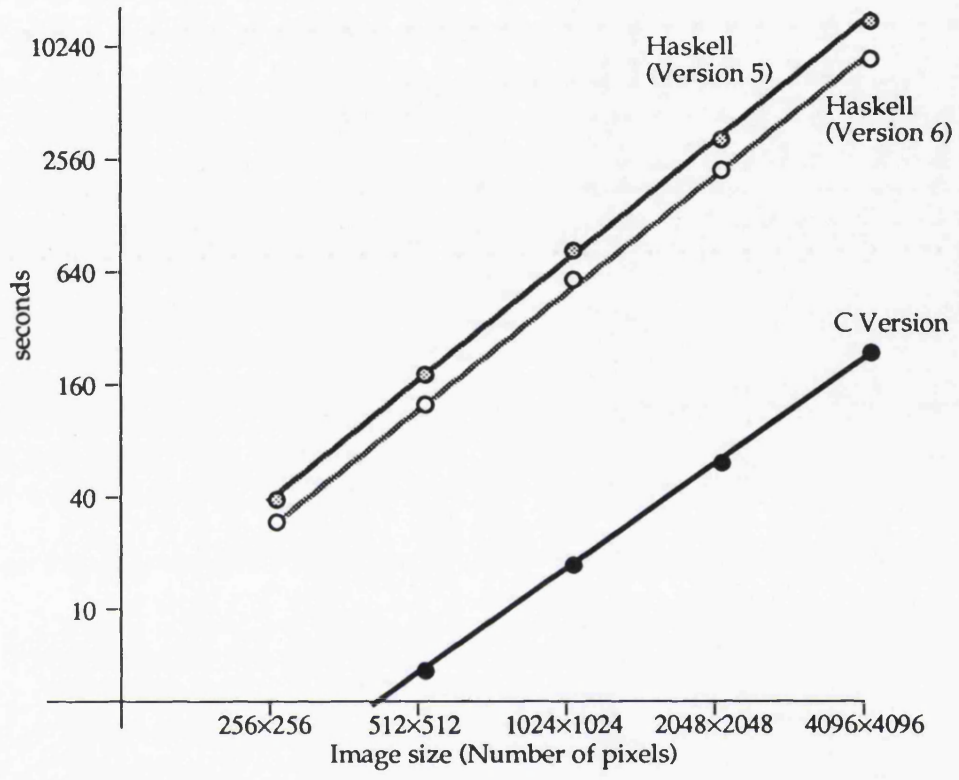


Figure 8-7 Graphic plot of time behaviour for larger images

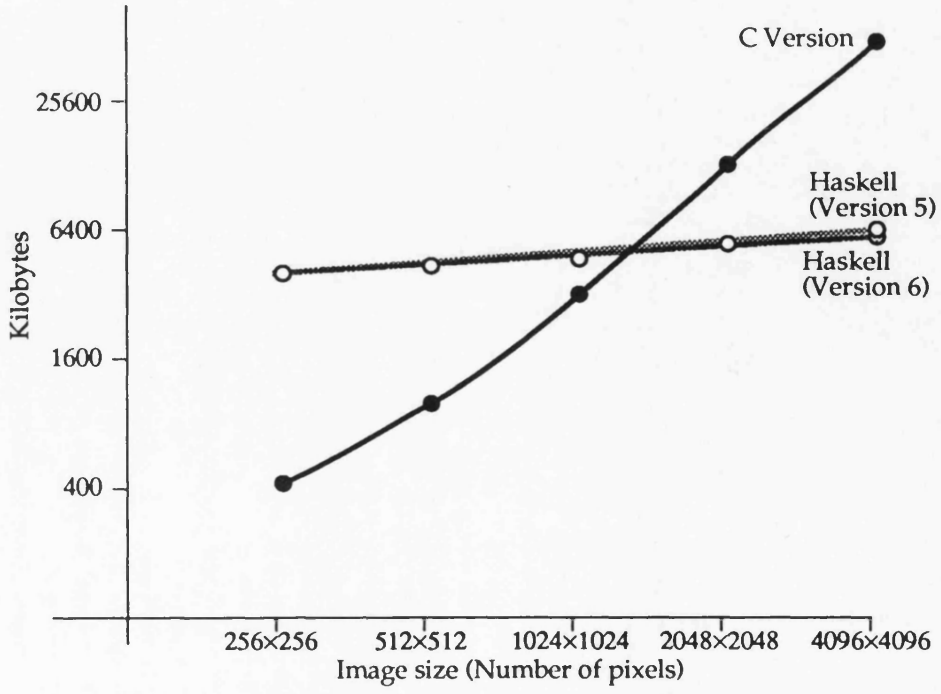


Figure 8-8 Graphic plot of space behaviour for larger images

8.6 Discussions

8.6.1 When lazy functional languages beat imperative languages

Table 8-1 shows that a recent compiled implementation of a lazy functional language outperforms an older interpretive implementation of a similar language by almost a factor of ten for the median filtering program. This result is obtained without making any alterations to the program to exploit the characteristics of the newer language and compiler. The same table also shows that a compiled C program for the same task outperforms the compiled functional program by more than a further factor of ten. However, the compilation of lazy functional languages is an active field of research in which major advances are still being made: there may yet be significant further improvements in the performance of functional programs on stock hardware.

Optimising code is not an easy task regardless of the languages. In C and Haskell, we achieved remarkable improvement in performance by using better algorithms in both cases, but it required extra programming effort. See the complete rework of the code to produce median-incremental (Figure 8-5). If a language is designed to be easy to implement better algorithms, better performance can be achieved more easily by using the language. There are a number of advantages of using lazy functional languages described all through the thesis, including ease of reading and writing. So, for rapid prototyping or executable specifications, lazy functional languages are well worth considering.

Furthermore, as shown in Section 8.5.2, when an image to be processed becomes very large, lazy functional languages beat imperative languages in terms of space efficiency. As described in Section 4.3, one of the advantages in using lazy functional languages for image processing is better memory management, i.e., data is always produced and consumed according to the needs and garbage is continuously collected. Whereas, the C version allocates memory for both input and output images, so that the increase of memory is proportional to the number of pixels in the images. If we process a very large image in the C version, it is impossible to process the image by the current code. In order to avoid this, programmers will need to do extra work to get the C code working well on large images, probably by rewriting the code to process an image line-by-line or chunk-by-chunk.

8.6.2 Use of other data structures

- Arrays

  The modification we made in Version 2 is to attach the known length of a list as an extra member of the data structure instead of calculating it afterwards, which has avoided accumulating the whole list and has fixed the space leak problem. This suggests that use of arrays may be plausible, because the fact that the size is known beforehand indicates there is no problem in allocating the area. In fact, an array implemented in Haskell has the range of its indices in its data structure, so that there is no overhead in obtaining the size of an array. It is not yet known whether arrays are easy to use for image processing in functional languages. More investigation is needed.

- Hierarchical data structures

  We did not test the versions which use hierarchical image representations, since we used ordinary rasterfiles as input data which need to be converted to such data structures. This can be an expensive operation, and there seems no benefit in using hierarchical data structures as long as we use the rasterfile format and the median filtering application. Whether use of hierarchical data structures is beneficial or not will depend on applications, and we need more investigation to demonstrate hierarchical data structures are more beneficial in real-life applications.

8.6.3 Caveats of lazy functional languages

It may be generally true that lazy functional languages are adequately efficient even with a naive way of programming. However, as we have experienced, if more efficiency is to be sought, optimising code may not be an easy job. There is a lot of subtlety in the behaviour of lazy functional programs and it is not too obvious from the code itself, whether to use foldl or foldr, for example.

In order to write more efficient code, programmers still have to work out the sources of inefficiency. For example, to avoid repeated calculations, they will have to add an extra parameter to a function if its value is known before the function calls. Compilers will spot redundancy within a function but not across different functions calls. Also, it may be a good idea to avoid possibly expensive operations which may involve a large amount of data, such as *length*. These precautions may be very general regardless of the kinds of languages.

Chapter 9: Conclusions

9.1 Overview

We have investigated the suitability of lazy functional languages for expressing image processing applications by implementing a variety of algorithms. Though Miranda has been used as the main implementation language, most of the discussions are independent of languages and may be applied to any other language as long as it has the same features. Some of the issues may be quite novel not only because there are only a few real applications written in lazy functional languages, but also we have tried to utilise the features of the languages, laziness in particular, in a positive manner. We should now come back to the original question, *"are lazy functional languages 'good' for writing image processing programs?"* and revisit the list of possible benefits given in Section 1.3.

9.2 Benefits of Functional Image Processing

9.2.1 Image processing in functional style

Image processing can be described as a certain function applied to input images or descriptions which yields output images or descriptions, i.e. input→process→output. Therefore, the basic style of functional programming:

output = function (inputs)

naturally describes image processing operations; for example, describing a series of operations, such as image file I/O functions to convert data from/to a long list of characters (Section 7.2.1) and a series of geometric transformations (Section 7.4.2)[1].

However, the real benefit of functional programming is not only its syntactic style, but its strong capability to construct complex operations from simple ones in a modular fashion, as described in the following.

9.2.2 Higher-order functions and image processing

Images usually have a regular and common structure, such as a 2D array of pixels, and higher-order functions are particularly useful to handle various operations on the same structure. This statement may be quite obvious and has previously been mentioned, e.g. [Breuel92a,

1. Many functions are used in a curried form including the collection of examples presented to show the capability of pointwise and convolution functions (Sections 3.3 and 3.5).

Parsons87a]. However, what we have achieved by the local neighbourhood operations implemented in Section 3.5 seems quite remarkable and has practical value.

We have demonstrated that a large collection of local neighbourhood operations can be constructed from only three basic functions: pointwise, translation and convolution. There are three kinds of usage of the convolution function:

- Linear operations, such as neighbourhood averaging, Laplacian, and the Sobel edge operator. These operations can be implemented by replacing a mask. Most languages can cope with this category, as data can be given as an argument to a function.

- Non-linear but separable operations, such as minimum and maximum filters. Because the convolution works on a "1D first then 2D" basis and functions such as min and max can be calculated in any order within a local neighbourhood, those operations can be implemented by replacing the arguments for additive and multiplicative operations.

- Non-linear and non-separable operations, such as median filtering. In this case, the convolution is used as an access method to neighbourhood pixels. The neighbourhood pixels are concatenated to form a list of pixels at the position, which is a separable operation that our convolution can deal with. Then the actual operation, such as taking a median, is mapped onto the list image in a pointwise manner.

Using the same technique, other local neighbourhood operations such as morphological operations may be implemented without much difficulty. When an image processing package is to be designed using a functional language, the convolution proposed in this thesis will be a strong candidate as a member, which allows users to implement various local neighbourhood operations without writing low-level code.

The general methodology would be not to implement an individual algorithm directly, but to grasp the common structure of operations and define these structures as higher-order polymorphic functions. Individual algorithms can be implemented later at a much higher level. This principle has been termed *algorithm categorisation* in [Parsons87a].

9.2.3 Polymorphic pixel types

In defining a representation of images, we have used the technique to leave the type of a pixel as a polymorphic type using a type variable. This may be a general method to allow processing

various categories of images in a unified manner, and has previously been used in [Allsop91a]. Using this technique, programmers do not have to write almost identical code for each different image category, and the concrete type can be inferred when the arguments (data and functions) to a function have been specified. The spatial conditional operation (Section 3.3.4) would be a good example of combining a boolean image and polymorphic images. In short, from the programmers' point of view, polymorphic typing allows a modular way of programming by reusing polymorphic functions as program parts.

However, a serious drawback has been pointed out when polymorphic data are used to communicate with the outside world (Section 7.2.2), as I/O is an essential operation in image processing. In such a case, programmers have to be aware of which concrete types are being processed, and they have to apply an explicit type conversion for each type. It has been suggested that, to overcome this problem, overloaded functions should be defined for I/O which call an appropriate conversion depending on a member in the type class. In addition, if a language has a facility to incorporate such low level functions written in an appropriate language such as assembler or C, it may overcome the I/O efficiency problem. Haskell may be able to cope with both suggestions.

9.2.4 Composing complex data and operations

An image is typically represented as a regular N dimensional array and functional languages allow construction of higher-dimensional data and operations using lower ones. This would be a general benefit in expressing image processing algorithms in functional languages and was mentioned in [Breuel92a]. However, our contribution is that we have presented a very clear methodology to realise this benefit which would be practically useful for implementing many data structures and operations for image processing.

The principle is to define a common data structure for *any* dimension using polymorphic types, then to implement an operation on the 1D version making it as polymorphic and higher-order as possible, and lastly to apply the 1D operation to itself to compose higher-dimensional operations. If the data structure is a plain list, composing the operations for a list of lists is straightforward as in [Allsop91a]. But using our method, more elaborate structures can be made common for all dimensions. We have defined *interval* as the common structure which is a tuple of parameters to describe the "shape" of the interval and the data in the interval, and regarded a row as an interval of pixels and an image as an interval

of rows. As concrete examples, we have implemented `interval *` (Chapter 3), `intervall *` (Chapter 5) and `intervalT *` (Chapter 6), and shown that the methods to compose higher-dimensional operations are very straightforward regardless of the basic data representations being lists or trees. We have not implemented 3D or higher dimensions, but they may be constructed using the same principle. This is another good example of modularity gained from the features of functional languages.

However, ease of programming may not always bring efficiency, since data structures and operations are not optimised for each dimension. As discussed in Section 6.6.1, the binary tree version is much easier to program than the quadtree version, but in terms of space usage the quadtree version should be more efficient. We have not tested the practical effect of this trade-off in real image processing applications, which is left as future work.

9.3 Benefits of Lazy Image Processing

9.3.1 Why lazy image processing?

Lazy evaluation delays the evaluation of an argument until its value is required, which is particularly suitable for image processing since each image operation is usually expensive, and by reducing unnecessary operations, the efficiency will be much improved. We often see statements of this kind, e.g. [Breuel92a, Lau-Kee91a], but in practice we do not fully understand what this really implies. This has been the principal motivation of the investigation in this thesis.

We have presented an extensive discussion regarding laziness and image processing in Chapter 4. The conclusions drawn from the discussion would simply be the following (see Section 4.3):

- ***Laziness contributes to efficiency of image processing programs:***

  There are three cases: (i) when a non-strict operation, such as a conditional operation, is applied to images, some of the images need not be evaluated, (ii) when the same argument appears more than once in an expression, it will be evaluated at most once, and (iii) when only a part of an image or an image in reduced resolution is required, not all the pixels are evaluated.

- ***Laziness contributes to modularity of image processing programs:***

Modularity, the capability to keep code for separate operations separate, may be a key issue of ease of reading and writing. Using lazy languages, programs can be written in a modular manner without extra programming effort or much loss of efficiency. For example, when an image operation comprises a series of sub-operations, programmers can describe the composition of the series straightforwardly as:

operation = *sub_op_n* . .... . *sub_op_3* . *sub_op_2* . *sub_op_1*

As evaluation is driven by demand, even if each sub-operation processes large data structures, intermediate storage between sub-operations is automatically handled by the system. In addition, even if each sub-operation consists of a number of lower-level operations, rescheduling of individual operations is automated by the system. And above all, as described above, this work can be done efficiently.

To give supporting evidence for these two statements, we have implemented geometric transformations, in particular affine transformations, in a lazy functional language. The algorithms have been chosen because they are inherently lazy and lazy languages can express them nicely. The following three subsections describe direct benefits from writing image processing programs in lazy functional languages to support the above two statements.

9.3.2 Eliminating unnecessary operations

In order to produce a partial demand for an image, we have implemented the display functions which enable users to specify a desired area. "Display" is merely an example function since it is easy to understand that pixels are required in order to display an image, but it can be any function which issues requests for part of an image; for example, a function to calculate a histogram of a certain part of an image. In order to produce a display pixel, interpolation is usually required, since the original image may be transformed and the pixels may not be aligned with the display grid. Since interpolation can be an expensive operation, the saving which results from the elimination of unnecessary operations will be large. In this way, we have successfully implemented functions which only carry out operations that are directly required.

It is also important to choose suitable representations of an image to get more benefit from lazy evaluation, because traversal of a data structure is needed to get access to required pixels even though its elements may not be evaluated. From this consideration, we have

implemented basically two types of representations: linear data structures, i.e. lists (Chapter 5), and hierarchical data structures, i.e. trees and quadtrees (Chapter 6). The former utilises the nature of ordered collection of elements, so that it is suitable when the order of demands is relatively close to the order of pixels (i.e. a display pixel list and an original pixel list are close to each other), or the required position is close to the top of a list, whereas the latter is suitable when pixels are only sparsely or relatively randomly required.

However, we did not compare the actual efficiency between those representations. This is because there are the other benefits of using particular data structures for particular applications. For example, hierarchical data structures are beneficial when large data compression effect can be expected, such as an image with large uniform areas. And unless these benefits are taken into account, it would not seem worthwhile comparing the efficiency. Currently, all the original images can only be handled as lists and conversions from lists to trees/quadtrees are required, which can be very expensive. This consideration of practical efficiency is left as future work.

9.3.3 Laziness and house-keeping operations

It would be possible using *non*-lazy languages to implement a system in which only required calculation is performed. But in order to do so, extra programming effort will be needed to manage memory and related routine sequencing operations. One of the benefits of lazy functional languages is that those house-keeping operations are embedded and no extra programming is involved. This becomes clearer when consecutive operations are applied to a pixel image: a series of operations is composed to become a composite function and its execution is driven by a demand to produce an output pixel. Therefore, there is no need to arrange memory for intermediate results. The particular benefit in geometric transformations is that consecutive transformations do not accumulate quantisation errors, because intermediate pixel images will never be produced.

In short, programming in lazy functional languages is easier because they release the programmer from cumbersome house-keeping tasks.

9.3.4 Laziness as the "glue" of functions

The direct benefit from the above discussion, i.e. lazy languages embed house-keeping operations, would be that programs can be made modular. As Hughes discussed this matter

using non-image processing examples [Hughes89a], different parts of a problem can be implemented as different functions and they can be put together using lazy evaluation as the "glue". We have demonstrated that this principle is also applicable to image processing programs.

For example, the 1D display function implemented in Section 5.3.3 consists of three basic parts: (i) scanning output raster, (ii) fetching necessary pixels from the original pixel list, and (iii) interpolation. These program parts can also be used for other programs, e.g. in the display function for a different representation (See Section 5.4.3). Likewise, the 1D display function in Section 6.4.6 using binary trees as the basic representation can also be modularised into three sub-processes: (i) scanning output raster, (ii) tree traversing until required resolution is satisfied, and (iii) calculation of the average of the sub-trees underneath. These parts can be "plugged in" without any consideration of synchronising operations or arrangement of memory between operations, i.e. those operations are "glued" together using lazy evaluation.

9.4 Utility of Lazy Image Processing

The utility of lazy image processing is not only to carry out conventional image processing operations efficiently as described above, but also it may create a new paradigm. By utilising both the functional and lazy nature of languages, it would be possible to handle pixel images and non-pixel images, such as images expressed as continuous functions, in a unified manner. Unless there is a demand to produce pixels, those function images can stay as functions and image processing may be carried out as function composition. When those two types of images are to be combined, necessary pixels will be produced from the function images.

We have demonstrated the possibility of this paradigm in Chapter 7. Although the examples presented are very primitive pointwise operations only, they at least show that it can be done. If we come up with suitable applications of this paradigm, it might develop a new frontier since an image should be a certain abstract description of objects in the real world.

However, how to best encapsulate various image representations is left as further work. Haskell's type class facilities look promising, but more investigation is needed.

9.5 Are Lazy Functional Languages Inefficient?

Even though we show off a variety of benefits of lazy functional languages, people would not use them unless they are reasonably efficient for their applications. In Chapter 8, we have presented some benchmarks to compare the actual speed and memory use of currently available lazy functional languages with a conventional imperative language. The comparison is not necessarily fair because different algorithms are used for each language. Using the median filtering example programmed relatively naively, it has been shown that the Haskell program runs 15 times slower than C (Table 8-1). The importance of this speed penalty depends on the application, but we can conclude that, taking all the benefits of lazy functional languages into account, lazy functional languages could become usable for some image processing uses, such as prototyping and executable specifications. The results in the thesis should help to dispel the myth, "Functional programming languages are toys" [Hudak89a].

Furthermore, we demonstrated in Chapter 8 with practical evidence that the Haskell version runs in nearly constant space even if an image gets very large. In contrast, the space usage of the C version is proportional to the number of pixels to be processed and the program does not run for an image larger than a certain size. If we want the C version to cope with much larger images, modification of the code is inevitable, which requires extra programming effort. What we discussed as the benefits of lazy functional languages for image processing applications has been confirmed by the benchmarking tests.

With regard to applying a series of image operations, a facility called *element chaining* is being discussed and proposed by the ISO committee for image processing and interchange [ISO92a]. The aims of element chaining are to minimise data storage, to eliminate unnecessary data flow, and so on. This shows that, although the scope of the technology may be quite different from lazy functional languages, the idea of memory management and instruction rescheduling as performed by implementations of lazy languages is highly desirable in image processing applications.

We also tried some improvement of the median filter code in Haskell and found that, although lazy functional languages are adequately efficient even with naive coding, there is still considerable room for improvement, in both the speed of the code generated by functional compilers and the tools to help functional language programmers write more efficient code. Better tools and compilers are becoming available, see for example [Runciman92a, Sansom93a,

The AQUA Team93a]. We expect emergence of these tools to encourage wider use of lazy functional languages.

9.6 What Has Been Achieved?

In comparison with the very diverse applications of image processing in the real world, the variety of algorithms we have presented in this thesis is small, and covers only a small portion of image processing. However, we have shown how a lazy functional language can be used to succinctly express a range of image processing algorithms, and how it can express some operations (e.g. geometric transformations) in a more modular and direct way than languages with strict semantics. Furthermore, we have shown how unnecessary image processing operations can be eliminated simply by using the laziness of the language.

However, one possibly large obstacle is the unfamiliar style, since writing image processing programs in lazy functional languages is quite new and most image processing programmers have only programmed in imperative languages. In order to overcome this difficulty, we need to accumulate more experience and motivate them by demonstrating convenience and adequate efficiency in real applications. Of course, a library of basic image processing modules and routines would also be a great help; if the efficiency was adequate, then the language facilities like polymorphism and higher-order functions would allow programming at a much higher level. In this light, the main contribution of this thesis is to present some of such experience, which should help start programming in the lazy functional style.

Supported by the achievements above, we can conclude the thesis by saying, *"Lazy functional image processing is nearly there."*

Appendix A. Function Definitions

A.1 Pointwise, Translation and Convolution "with an Origin"

```
|| YK 18/09/92
|| origin.m
|| Miranda code described in Chapter 3


|| The "with an Origin" Version

interval * == (num,[*])
row *      == interval *
image *    == interval (row *)


|| pointwise operations (1D)

unaryInterval::(*->**)->interval *->interval **
unaryInterval f (o,p) = (o, map f p)

binaryInterval::(*->**->***)->interval *->interval **->interval ***
binaryInterval f (o1,p1) (o2,p2)
   = (o, map2 f (drop (o2-o1) p1) p2) , if o1 < o2
   = (o, map2 f p1 (drop (o1-o2) p2)) , otherwise
     where o = max2 o1 o2

unaryRow  = unaryInterval
binaryRow = binaryInterval


|| pointwise operations (2D)

unaryPointwise  = unaryRow.unaryRow
binaryPointwise = binaryRow.binaryRow


|| example pointwise operations

negateImage  = unaryPointwise neg
absImage     = unaryPointwise abs
logImage     = unaryPointwise log
constImage x = unaryPointwise (const x)
notImage     = unaryPointwise (~)
doubleImage  = unaryPointwise (*2)

addImage = binaryPointwise (+)
subImage = binaryPointwise (-)
maxImage = binaryPointwise max2
minImage = binaryPointwise min2

andImage = binaryPointwise (&)
orImage  = binaryPointwise (\/)
eqImage  = binaryPointwise (=)


|| example - inverse look up table

inverseLookUpTable = unaryPointwise ([255,254..0]!)

inverseLookUpTable2 range = unaryPointwise ((+range).neg)


|| example - spatial conditional

condImage im1 im2 im3
   = binaryPointwise cond im1 (binaryPointwise pair im2 im3)
     where cond True  (a,b) = a
           cond False (a,b) = b
           pair a b = (a,b)


|| image translation

translateInterval::num->interval *->interval *
translateInterval d (o,p) = (o+d,p)

translateRow = translateInterval
translateImage x y (o,p) = translateRow y (o,map (translateRow x) p)


|| convolution

convolveRow   = conv1 (*) (binaryRow (+))
convolveImage = conv1 convolveRow (binaryPointwise (+))

conv1 mul add mask im
```

```
   = accum [prod (element mask p) (shiftInterval p) | p<-domain mask]
     where
     accum  = foldl1 add
     prod x = unaryInterval (mul x)
     shiftInterval p = translateInterval (p-((leng mask) div 2)) im

domain    = index.snd
leng      = (#).snd
element   = (!).snd
foldim op = (foldl1 op).snd

|| example - average, laplacian

average = convolveImage (makeMask [[1/9,1/9,1/9],
                                   [1/9,1/9,1/9],
                                   [1/9,1/9,1/9]])

laplacian = convolveImage (makeMask [[ 0, 1, 0],
                                     [ 1,-4, 1],
                                     [ 0, 1, 0]])

makeMask::[[*]]->image *
makeMask m = fn (map fn m) where fn x = (0,x)


|| example - sobel

sobel im
   = binaryPointwise (+) (absImage imh) (absImage imv)
     where imh   = convolveImage (makeMask hMask) im
           imv   = convolveImage (makeMask (transpose hMask)) im
           hMask = [[ 1, 2, 1],
                    [ 0, 0, 0],
                    [-1,-2,-1]]


|| example - maximum and minimum filter

maxFilterRow   = conv1 secondArg (binaryRow max2)
maxFilterImage = conv1 maxFilterRow maxImage

minFilterRow   = conv1 secondArg (binaryRow min2)
minFilterImage = conv1 minFilterRow minImage

secondArg a b = b


|| example - median filter

medianImage mask = (unaryPointwise median).(localHistImage mask)
median list      = (sort list)!(#list div 2)

localHistImage = conv1 (localHistRow) (binaryPointwise (++))
localHistRow   = conv1 f (binaryRow (++)) where f a b = [b]


|| another convolution - faithful to the definition

convolveRow2   = conv2 (*) (+)
convolveImage2 = conv2 convolveRow2 (binaryRow (+))

conv2 mul add (mo,msk) im = correlation mul add (mo,reverse msk) im

correlation mul add mask im
   = (fst im, [dot mul add mask im x | x<-domain im])

dot mul add mask im x
   = accum (shiftMask $times im)
     where
     accum     = foldim add
     shiftMask = translateRow (x-((leng mask) div 2)) mask
     times     = binaryInterval mul
```

A.2 Translation and Scaling Using List Representation

```
|| YK 07/09/92
|| geometric1.m
|| Miranda code described in Chapter 5

|| translation and scaling only

|| type synonyms

interval1 * == (num,num,[*])  || (origin,length,pixel_list)
row1 *      == interval1 *
image1 *    == interval1 (row1 *)

plist * == [(num,*)]
```

```
|| translation and scaling for rows and images

transRow1,scaleRow1::num->row1 *->row1 *
transRow1 d (o,l,p)  = (d+o, l, p)
scaleRow1 sc (o,l,p) = (sc*o, sc*l, p)

transImage1,scaleImage1::num->num->image1 *->image1 *
transImage1 dx dy (oy,ly,r) = transRow1 dy (oy,ly,map (transRow1 dx) r)
scaleImage1 sx sy (oy,ly,r) = scaleRow1 sy (oy,ly,map (scaleRow1 sx) r)


||  Conversion from an interval1 to a plist

intervalToPlist::interval1 *->plist *
intervalToPlist (o,l,[a]) = [(o,a)]
intervalToPlist (o,l,p)   = zip2 posList p
                            where posList = map ((+o).(*period)) [0..]
                                  period  = l/(#p-1)


|| basic display function

disp::num->num->num->num->(num->**->plist *->**)->**->plist *->[**]
disp x n hi lo ifn bg pl
   = [] , if n<=0
   = bg:disp (x+1) (n-1) hi lo ifn bg pl , if ~(lo<=x<=hi)
   = ifn x bg pl:disp (x+1) (n-1) hi lo ifn bg (dropUpto (x+1) pl) , otherwise


|| Various functions for display

dispRow1 x n ifn (o,l,p) = disp x n (o+1) o ifn bgp (intervalToPlist (o,l,p))

dispImage1 x y xn yn ifn1 ifn2 (o,l,p)
   = disp y yn (o+1) o ifn2 bg (intervalToPlist im1)
     where bg  = rep xn bgp
           im1 = (o,l,map (dispRow1 x xn ifn1) p)

dispImageNN1,dispImageLI1::num->num->num->num->image1 num->[[num]]
dispImageNN1 x y xn yn = dispImage1 x y xn yn nearest nearest
dispImageLI1 x y xn yn  = dispImage1 x y xn yn linear1 linear2


|| specialised "tail" function

dropUpto x []         = []
dropUpto x [x1]       = [x1]
dropUpto x (x1:x2:xs)
   = (x1:x2:xs) , if isBetween x (positionOf x1) (positionOf x2)
   = dropUpto x (x2:xs) , otherwise

isBetween a b c = (b<=a<=c) \/ (b>=a>=c)


|| Nearest neighbour interpolate function

nearest::num->*->plist *->*
nearest x bg []         = bg
nearest x bg [a]        = valueOf a
nearest x bg (a1:a2:as) = valueOf a1
                          , if dist x (positionOf a1) <= dist x (positionOf a2)
                        = valueOf a2 , otherwise

dist a b = abs(a-b)


|| Linear interpolate functions

linear1 = linear0 li1
linear2 = linear0 li2

linear0 fn x bg []         = bg
linear0 fn x bg [a]        = bg
linear0 fn x bg (a1:a2:as) = fn x a1 a2

li1 x (x1,v1) (x2,v2) = v1+(v2-v1)*(x-x1)/(x2-x1) , if x2 ~= x1
                      = bgp                       , otherwise

li2 y (y1,vs1) (y2,vs2) = map2 (li1 y) ys1 ys2
                          where ys1 = zip2 (repeat y1) vs1
                                ys2 = zip2 (repeat y2) vs2

bgp = 0
positionOf (p,v) = p
valueOf (p,v) = v
```

A.3 Affine Transformations Using List Representation

```
|| YK 07/09/92
|| geometric2.m
|| Miranda code described in Chapter 5

|| Affine - translation, scaling, rotation

|| image representation

row2 *   == ((num,num),(num,num),[*])
image2 * == [row2 *]

plist * == [(num,*)]


|| translation, scaling and rotation for 1D and 2D images

transRow2 dx dy (o,l,p)    = (pairl ((+dx),(+dy)) o,l,p)

scaleRow2 scx scy (o,l,p) = (scl o,scl l,p)
                            where scl = pairl ((*scx),(*scy))

rotateRow2 th (o,l,p)      = (rot o,rot l,p)
                             where rot = pair2 (rotx th,roty th)
                                   rotx th x y = x*(cos th)-y*(sin th)
                                   roty th x y = x*(sin th)+y*(cos th)

pairl (fn1,fn2) (x,y) = (fn1 x,fn2 y)
pair2 (fn1,fn2) (x,y) = (fn1 x y,fn2 x y)

transImage2 dx dy   = map (transRow2 dx dy)
scaleImage2 scx scy = map (scaleRow2 scx scy)
rotateImage2 th     = map (rotateRow2 th)


||  Conversions from a row to a plist

rowToPlist::row2 *->plist *
rowToPlist ((ox,oy),l,[a]) = [(ox,a)]
rowToPlist ((ox,oy),(lx,ly),p) = zip2 posList p
                                 where posList = map ((+ox).(*period)) [0..]
                                       period  = lx/(#p-1)

rowToPlistY ((ox,oy),(lx,ly),p) = rowToPlist ((oy,ox),(ly,lx),p)


|| The basic display function

disp::num->num->num->num->(num->**->plist *->**)->**->plist *->[**]
disp x n hi lo ifn bg pl
   = []                                  , if n<=0
   = bg:disp (x+1) (n-1) hi lo ifn bg pl , if ~(lo<=x<=hi)
   = ifn x bg pl:disp (x+1) (n-1) hi lo ifn bg (dropUpto (x+1) pl), otherwise


|| Various functions for display

dispPlist::num->num->(num->*->plist *->*)->*->plist *->[*]
dispPlist x n ifn bg pl = disp x n hi lo ifn bg pl
                          where hi = positionOf (last pl)
                                lo = positionOf (hd pl)

dispRow2::num->num->(num->*->plist *->*)->*->row2 *->row2 *
dispRow2 x n ifn bg ((ox,oy),(lx,ly),p)
   = ((x,ly/lx*(x-ox)+oy),(n-1,ly/lx*(n-1)),dsp)
     where dsp = disp x n (ox+lx) ox ifn bg (rowToPlist ((ox,oy),(lx,ly),p))

dispImage2::num->num->num->num->(num->*->plist *->*)->*->image2 *->[[*]]
dispImage2 x y xn yn ifn bg im
   = transpose (map (dispPlist y yn ifn bg) (transpose iml))
     where iml = map (rowToPlistY.dispRow2 x xn ifn bg) im

dispImageNN2,dispImageLI2::num->num->num->num->image2 num->[[num]]
dispImageNN2 x y xn yn = dispImage2 x y xn yn nearest bgp
dispImageLI2 x y xn yn = dispImage2 x y xn yn linear bgp

bgp = 0

|| specialised "tail" function

dropUpto x []         = []
dropUpto x [x1]       = [x1]
dropUpto x (x1:x2:xs)
   = (x1:x2:xs) , if isBetween x (positionOf x1) (positionOf x2)
   = dropUpto x (x2:xs) , otherwise

isBetween a b c = (b<=a<=c) \/ (b>=a>=c)
positionOf (p,v) = p


|| Nearest neighbour interpolate function
```

```
nearest::num->*->plist *->*
nearest x bg []          = bg
nearest x bg [a]         = valueOf a
nearest x bg (a1:a2:as) = valueOf a1
                          , if dist x (positionOf a1) <= dist x (positionOf a2)
                        = valueOf a2 , otherwise

dist a b = abs(a-b)
valueOf (p,v) = v


|| Linear interpolate functions

linear::num->num->plist num->num
linear x bg []                    = bg
linear x bg [a]                   = bg
linear x bg ((x1,v1):(x2,v2):as) = v1+(v2-v1)*(x-x1)/(x2-x1), if x2~=x1
                                  = bg                      , otherwise
```

A.4 Translation and Scaling Using Tree of Trees Representation

```
|| YK 07/09/92
|| treeOfTrees1.m
|| Miranda code described in Chapter 6

|| translation and scaling using tree of trees representation

|| type definitions

tree * ::= Nil | Leaf * | Node (tree *) (tree *)

intervalT * == (num,num,tree *)
rowT *      == intervalT *
imageT *    == intervalT (rowT *)

|| Conversion from a pixel list to a pixel tree (left-packed)

makeTree::[*]->num->tree *
makeTree [] s   = Nil
makeTree [x] 1  = Leaf x
makeTree list s = Node (makeTree left half) (makeTree right half)
                  where left  = take half list
                        right = drop half list
                        half  = s div 2

|| Conversion from a pixel list to a row

listToImage::[[*]]->imageT *
listToImage l = listToInterval (map listToInterval l)

listToInterval::[*]->intervalT *
listToInterval l = (s/2, s/4, condenseTree (listToTree l))
                   where s = maxSize (#l)

listToTree::[*]->tree *
listToTree l = makeTree l (maxSize (#l))

maxSize 0 = 0
maxSize 1 = 1
maxSize n = 2^(entier(log (n-1)/log 2)+1)

|| The background value!

bgp = 0

|| Tree condensation

reformTree::(tree *->tree *)->tree *->tree *
reformTree f (Leaf x)     = f (Leaf x)
reformTree f Nil   = f Nil
reformTree f (Node t1 t2) = f (Node (reformTree f t1) (reformTree f t2))

condenseTree::tree *->tree *
condenseTree = reformTree fun
               where fun (Node (Leaf x) (Leaf x)) = Leaf x
                     fun (Node Nil Nil)           = Nil
                     fun t                        = t

|| translation and scaling for rows and images

transRowT d (r,s,t)  = (d+r,s,t)
scaleRowT sc (r,s,t) = (sc*r,sc*s,t)

transImageT dx dy (ry,sy,t) = transRowT dy (ry,sy,mapTree (transRowT dx) t)
scaleImageT scx scy (ry,sy,t)
   = scaleRowT scy (ry,sy,mapTree (scaleRowT scx) t)
```

```
mapTree::(*->**)->tree *->tree **
mapTree f Nil          = Nil
mapTree f (Leaf x)     = Leaf (f x)
mapTree f (Node x1 x2) = Node (mapTree f x1) (mapTree f x2)

|| display functions

displayT::num->num->num->num->imageT num->[[num]]
displayT x y xn yn (r,s,t)
   = dispT y yn avrList (rep xn bgp) (r,s,mapTree (dispT x xn avrNum bgp) t)

dispT::num->num->(*->*->*)->*->intervalT *->[*]
dispT x n fn bg i = [] , if n=0
                  = lookUpT x fn bg i:dispT (x+1) (n-1) fn bg i , otherwise

lookUpT::intervalT *->num->(*->*->*)->*->*
lookUpT (r,s,t) x fn bg          = bg , if (x<r-2*s)\/(r+2*s<=x)
lookUpT (r,s,Nil) x fn bg        = bg
lookUpT (r,s,Leaf a) x fn bg     = a
lookUpT (r,s,Node t1 t2) x fn bg
   = avrTree (Node t1 t2) fn bg  , if s<0.5
   = lookUpT (r-s,s/2,t1) x fn bg, if x<r
   = lookUpT (r+s,s/2,t2) x fn bg, otherwise


|| Averaging functions

avrTree::tree *->(*->*->*)->*->*
avrTree Nil fn bg          = bg
avrTree (Leaf a) fn bg     = a
avrTree (Node t1 t2) fn bg= fn (avrTree t1 fn bg) (avrTree t2 fn bg)

avrList::[num]->[num]->[num]
avrList l1 l2 = [(a+b)/2 | (a,b)<-zip2 l1 l2]

avrNum::num->num->num
avrNum a b = (a+b)/2
```

A.5 Affine Transformations Using Quadtree Representation

```
|| YK 07/09/92
|| quadtree.m
|| Miranda code described in Chapter 6

|| Affine transformation of a quadtree

%include "vector"

|| type definitions - ordering of subtrees is SW, SE, NW, NE

qtree * ::= QNil | QLeaf * | QNode * (qtree *) (qtree *) (qtree *) (qtree *)

coord     == vector
imageQT * == (coord,vector,vector,qtree *)

|| geometric transformations

transQTree dx dy (r,sw,se,qt) = (vAdd (vMake (dx,dy)) r,sw,se,qt)

scaleQTree scx scy (r,sw,se,qt) = (scl r,scl sw,scl se,qt)
                                  where scl = vMul (vMake (scx,scy))

rotateQTree th (r,sw,se,qt) = (rot r,rot sw,rot se,qt)
                              where
                              rot = vMap2 (rotx th, roty th)
                              rotx th x y = x*(cos th)-y*(sin th)
                              roty th x y = x*(sin th)+y*(cos th)

|| functions for averaging a quadtree

avrQTree::*->(*->*->*->*->*)->qtree *->qtree *->qtree *->qtree *->*
avrQTree bg f t0 t1 t2 t3
   = f (valQTree bg t0) (valQTree bg t1) (valQTree bg t2) (valQTree bg t3)

valQTree::*->qtree *->*
valQTree bg QNil                  = bg
valQTree bg (QLeaf v)             = v
valQTree bg (QNode v t0 t1 t2 t3) = v

|| conversion from a list of pixel lists to a quadtree. Bottom-left
|| packed rectangular is assumed.

makeQTree::*->(*->*->*->*->*)->[[*]]->num->qtree *
makeQTree bg fn [[]] s      = QNil
makeQTree bg fn []   s      = QNil
makeQTree bg fn [[a]] 1     = QLeaf a
makeQTree bg fn lists s
   = makeQNode bg fn (subt 0) (subt 1) (subt 2) (subt 3)
```

```
     where
     half   = s div 2
     subt i = makeQTree bg fn (quarter i half lists) half

makeQNode bg fn t0 t1 t2 t3 = QNode (avrQTree bg fn t0 t1 t2 t3) t0 t1 t2 t3

quarter::num->num->[[*]]->[[*]]
quarter i n
   = divide eastWest northSouth
     where divide f g l = g (map f l)
           eastWest     = [take n, drop n]!(i mod 2)
           northSouth   = [take n, drop n]!(i div 2)

|| For images whose pixels are nums

listsToQTree::[[num]]->qtree num
listsToQTree lists
   = QNil , if lists=[]
   = condenseQTree (makeQTree bgp average4 lists s) , otherwise
     where s = maxSize (max2 (#(hd lists)) (#lists))

listsToImageQT::[[num]]->imageQT num
listsToImageQT l
   = (vMake(s/2,s/2),vMake(-s/4,-s/4),vMake(s/4,-s/4),
      condenseQTree (makeQTree bgp average4 l s))
     where s = maxSize (max2 (#(hd l)) (#l))

maxSize 0 = 0
maxSize 1 = 1
maxSize n = 2^(entier(log (n-1)/log 2)+1)

average4 a b c d = (a+b+c+d)/4

|| Quadtree condensation

reformQTree::(qtree *->qtree *)->qtree *->qtree *
reformQTree f (QLeaf x) = f (QLeaf x)
reformQTree f QNil      = f QNil
reformQTree f (QNode v qt0 qt1 qt2 qt3)
   = f (QNode v (reformQTree f qt0) (reformQTree f qt1)
                (reformQTree f qt2) (reformQTree f qt3))

condenseQTree::qtree *->qtree *
condenseQTree
   = reformQTree fun
     where
     fun (QNode v (QLeaf x) (QLeaf x) (QLeaf x) (QLeaf x)) = QLeaf x
     fun (QNode v QNil QNil QNil QNil)                     = QNil
     fun t                                                 = t

|| display functions

displayQT::num->num->num->num->*->imageQT *->[[*]]
displayQT xs ys xn yn bg im
   = [] , if yn=0
   = dispQT xs xn ys bg im:displayQT xs (ys+1) xn (yn-1) bg im , otherwise

dispQT::num->num->num->*->imageQT *->[*]
dispQT xs xn y bg im
   = [] , if xn=0
   = lookUpQTree (vMake (xs,y)) bg im:dispQT (xs+1) (xn-1) y bg im , otherwise

lookUpQTree::coord->*->imageQT *->*
lookUpQTree p bg (root,sw,se,qt)
   = bg             , if ~(vIsInside p vertexList)\/qt=QNil
   = valQTree bg qt , if isQLeaf qt\/(vLength sw<len)&(vLength se<len)
   = lookUpQTree p bg (nextRoot,vHalf sw,vHalf se,subTree qt), otherwise
     where
     len            = 1/sqrt 2
     vertexList     = map ((vAdd root).vDouble) [sw,se,vNeg sw,vNeg se]
     s              = sw $vAdd se
     w              = sw $vSub se
     isWest         = vDirection (root $vSub p) (root $vAdd s $vSub p) < 0
     isSouth        = vDirection (root $vSub p) (root $vAdd w $vSub p) > 0
     (nextRoot, i) = (root $vAdd sw, 0) , if isWest & isSouth
                   = (root $vAdd se, 1) , if ~isWest & isSouth
                   = (root $vSub se, 2) , if isWest & ~isSouth
                   = (root $vSub sw, 3) , if ~isWest & ~isSouth
     subTree (QNode v a b c d) = [a,b,c,d]!i

isQLeaf (QLeaf a) = True
isQLeaf qt = False

bgp = 0
```

A.6 Abstract Data Type: fImage

```
||  YK 16/09/92
||  functionimage.m
```

```
||  Abstract data type for image as function

%include "vector"
%export fImage makeFImage writeFImage displayFImage lookUpFImage
        translateFImage scaleFImage rotateFImage

filename == [char]
coord    == vector


abstype fImage *
with makeFImage::(coord->*)->fImage *
     writeFImage::coord->vector->fImage num->filename->[sys_message]
     displayFImage::coord->vector->fImage *->[[*]]
     lookUpFImage::coord->fImage *->*
     translateFImage::vector->fImage *->fImage *
     scaleFImage::vector->fImage *->fImage *
     rotateFImage::num->fImage *->fImage *

fImage * == coord->*

makeFImage fn = fn

displayFImage position size fim
   = [] , if vYelement size=0
   = dispFImage position size fim
     :displayFImage (vYup position) (vYdown size) fim , otherwise

dispFImage position size fim
   = [] , if vXelement size=0
   = fim position:dispFImage (vXup position) (vXdown size) fim , otherwise

writeFImage origin size fim name
  = [Tofile name (header++cmap++data), Closefile name]
     where
     header = numToCharList [ras_magic_num,w,h,8,1,1,1,768]
     w      = vXelement size
     h      = vYelement size
     l      = w*h
     cmap   = map decode ([0..255]++[0..255]++[0..255])
     data   = map (decode.clip)
              ((concat.reverse) (displayFImage origin size fim))

clip x = 0        , if x<0
       = 255      , if x>255
       = entier x , otherwise

ras_magic_num = 1504078485

numToCharList::[num]->[char]
numToCharList []     = []
numToCharList (n:ns) = numToFourChar n++numToCharList ns

numToFourChar::num->[char]
numToFourChar n = map (decode.(mod 256).(n div)) (map (256^) [3,2..0])

lookUpFImage p fim = fim p

translateFImage v fim = fim.((vAdd.vNeg) v) .

scaleFImage v fim = fim.((vMul.vRecip) v)

rotateFImage th fim = fim.(vMap2 (rotx (-th), roty (-th)))
                      where rotx th x y = x*(cos th)-y*(sin th)
                            roty th x y = x*(sin th)+y*(cos th)
```

A.7 Abstract Data Type: pImage - The List Implementation

```
|| YK 16/09/92
|| pixelimageL.m

|| Miranda code described in Chapter 7
|| Abstract data type pImage (implemented as lists)

%include "vector"
%export pImage readPImage makePImage writePImage displayPImage
        translatePImage scalePImage rotatePImage

filename == [char]
coord    == vector
pixel    == num

abstype pImage
with readPImage::filename->pImage
     makePImage::[[pixel]]->pImage
     writePImage::coord->vector->pImage->filename->[sys_message]
     displayPImage::coord->vector->pImage->[[pixel]]
     translatePImage::vector->pImage->pImage
```

```
     scalePImage::vector->pImage->pImage
     rotatePImage::num->pImage->pImage

|| image representation

row2   == ((num,num),(num,num),[num])
pImage == [row2]

plist * == [(num,*)]

|| functions related to rasterfile I/O

readPImage name
   = (listsToPImage.reverse.splitList width.map code.drop skip) (read name)
     where skip   = headerLength+ras_maplength header
           width  = ras_width header
           header = readHeader name

makePImage = listsToPImage

writePImage origin size im name
   = [Tofile name (header++cmap++data), Closefile name]
     where header = numToCharList [ras_magic_num,w,h,8,1,1,1,768]
           w      = vXelement size
           h      = vYelement size
           l      = w*h
           cmap   = map decode ([0..255]++[0..255]++[0..255])
           data   = (map (decode.entier).concat.reverse)
                    (displayPImage origin size im)

splitList::num->[*]->[[*]]
splitList n [] = []
splitList n l  = [take n l]++(splitList n (drop n l))

[ras_magic, ras_width,ras_height, ras_depth,
 ras_length,ras_type, ras_maptype,ras_maplength] = [(!i)|i<-[0..7]]

headerLength  = 32
ras_magic_num = 1504078485
rmt_equal_rgb = 1

readHeader::filename->[num]
readHeader name
   = error "getHeader:not a rasterfile."          , if ~isRaster
   = error "getHeader:not a byte image."          , if ~is8bit
   = error "getHeader:unexpected colour map type", if ~isEqualRgb
   = header                                       , otherwise
     where
     isRaster   = (ras_magic header=ras_magic_num)
     is8bit     = (ras_depth header=8)
     isEqualRgb = (ras_maptype header=rmt_equal_rgb)
     header     = charToNumList (take headerLength (read name))

charToNumList::[char]->[num]
charToNumList [] = []
charToNumList cs = fourCharToNum (take 4 cs):charToNumList (drop 4 cs)

fourCharToNum::[char]->num
fourCharToNum cs = foldl1 (+) (map2 (*) (map code cs) (map (256^) [3,2..0]))

numToCharList::[num]->[char]
numToCharList []     = []
numToCharList (n:ns) = numToFourChar n++numToCharList ns

numToFourChar::num->[char]
numToFourChar n = map (decode.(mod 256).(n div)) (map (256^) [3,2..0])

|| translation, scaling and rotation for 1D and 2D images

transRow2 dx dy (o,l,p)    = (pairl ((+dx),(+dy)) o,l,p)

scaleRow2 scx scy (o,l,p) = (scl o,scl l,p)
                            where scl = pairl ((*scx),(*scy))

rotateRow2 th (o,l,p)      = (rot o,rot l,p)
                             where rot = pair2 (rotx th,roty th)
                                   rotx th x y = x*(cos th)-y*(sin th)
                                   roty th x y = x*(sin th)+y*(cos th)

pairl (fn1,fn2) (x,y) = (fn1 x,fn2 y)
pair2 (fn1,fn2) (x,y) = (fn1 x y,fn2 x y)

translatePImage d = map (transRow2 (vXelement d) (vYelement d))
scalePImage s     = map (scaleRow2 (vXelement s) (vYelement s))
rotatePImage th   = map (rotateRow2 th)

||  Producing a pImage from a 2D list

listsToPImage l = zip3 (zip2 [0,0..] [0..]) (zip2 [w,w..] [0,0..]) l
                  where w = #(hd l)-1

||  Conversions from a row to a plist

rowToPlist ((ox,oy),l,[a])     = [(ox,a)]
rowToPlist ((ox,oy),(lx,ly),p) = zip2 posList p
```

```
                                where posList = map ((+ox).(*period)) [0..]
                                      period  = lx/(#p-1)

rowToPlistY ((ox,oy),(lx,ly),p) = rowToPlist ((oy,ox),(ly,lx),p)


|| The basic display function

disp::num->num->num->num->(num->**->plist *->**)->**->plist *->[**]
disp x n hi lo ifn bg pl
   = []                                    , if n<=0
   = bg:disp (x+1) (n-1) hi lo ifn bg pl , if ~(lo<=x<=hi)
   = ifn x bg pl:disp (x+1) (n-1) hi lo ifn bg (dropUpto (x+1) pl), otherwise

|| Various functions for display

dispPlist::num->num->(num->*->plist *->*)->*->plist *->[*]
dispPlist x n ifn bg pl = disp x n hi lo ifn bg pl
                          where hi = positionOf (last pl)
                                lo = positionOf (hd pl)

dispRow2 x n ifn bg ((ox,oy),(lx,ly),p)
   = ((x,ly/lx*(x-ox)+oy),(n-1,ly/lx*(n-1)),dsp)
     where dsp = disp x n (ox+lx) ox ifn bg (rowToPlist ((ox,oy),(lx,ly),p))

dispImage2 x y xn yn ifn bg im
   = transpose (map (dispPlist y yn ifn bg) (transpose im1))
     where im1 = map (rowToPlistY.dispRow2 x xn ifn bg) im

displayPImage o s
   = dispImageNN2 (vXelement o) (vYelement o) (vXelement s) (vYelement s)

dispImageNN2 x y xn yn = dispImage2 x y xn yn nearest bgp

bgp = 0

|| specialised "tail" function

dropUpto x []         = []
dropUpto x [x1]       = [x1]
dropUpto x (x1:x2:xs)
   = (x1:x2:xs) , if isBetween x (positionOf x1) (positionOf x2)
   = dropUpto x (x2:xs) , otherwise

isBetween a b c = (b<=a<=c) \/ (b>=a>=c)
positionOf (p,v) = p

|| Nearest neighbour interpolation

nearest::num->*->plist *->*
nearest x bg []         = bg
nearest x bg [a]        = valueOf a
nearest x bg (a1:a2:as) = valueOf a1
                          , if dist x (positionOf a1) <= dist x (positionOf a2)
         = valueOf a2 , otherwise

dist a b = abs(a-b)
valueOf (p,v) = v
```

A.8 Abstract Data Type: pImage - The Quadtree Implementation

```
|| YK 16/09/92
|| pixelimageQ.m

|| Miranda code described in Chapter 7
|| Abstract data type pImage (implemented as quadtrees)

%include "vector"
%export pImage readPImage makePImage writePImage displayPImage
        translatePImage scalePImage rotatePImage

filename == [char]
coord    == vector
pixel    == num

abstype pImage
with readPImage::filename->pImage
     makePImage::[[pixel]]->pImage
     writePImage::coord->vector->pImage->filename->[sys_message]
     displayPImage::coord->vector->pImage->[[pixel]]
     translatePImage::vector->pImage->pImage
     scalePImage::vector->pImage->pImage
     rotatePImage::num->pImage->pImage
     showPImage::pImage->[char]

|| type definitions - ordering of subtrees: SW, SE, NW, NE
```

```
qtree * ::= QNil | QLeaf * | QNode * (qtree *) (qtree *) (qtree *) (qtree *)

pImage == (coord,vector,vector,qtree pixel) || (root,SW,SE,pixels)

|| functions related to rasterfile I/O

readPImage name
   = (listsToPImage.reverse.splitList width.map code.drop skip) (read name)
     where skip   = headerLength+ras_maplength header
           width  = ras_width header
           header = readHeader name

makePImage = listsToPImage

writePImage origin size im name
   = [Tofile name (header++cmap++data), Closefile name]
     where header = numToCharList [ras_magic_num,w,h,8,1,1,1,768]
           w      = vXelement size
           h      = vYelement size
           l      = w*h
           cmap   = map decode ([0..255]++[0..255]++[0..255])
           data   = (map decode.concat.reverse) (displayPImage origin size im)

splitList::num->[*]->[[*]]
splitList n [] = []
splitList n l  = [take n l]++(splitList n (drop n l))

[ras_magic, ras_width,ras_height, ras_depth,
 ras_length,ras_type, ras_maptype,ras_maplength] = [(!i)|i<-[0..7]]

headerLength  = 32
ras_magic_num = 1504078485
rmt_equal_rgb = 1

readHeader::filename->[num]
readHeader name
   = error "getHeader:not a rasterfile."          , if ~isRaster
   = error "getHeader:not a byte image."          , if ~is8bit
   = error "getHeader:unexpected colour map type", if ~isEqualRgb
   = header                                       , otherwise
     where
     isRaster   = (ras_magic header=ras_magic_num)
     is8bit     = (ras_depth header=8)
     isEqualRgb = (ras_maptype header=rmt_equal_rgb)
     header     = charToNumList (take headerLength (read name))

charToNumList::[char]->[num]
charToNumList [] = []
charToNumList cs = fourCharToNum (take 4 cs):charToNumList (drop 4 cs)

fourCharToNum::[char]->num
fourCharToNum cs = foldl1 (+) (map2 (*) (map code cs) (map (256^) [3,2..0]))

numToCharList::[num]->[char]
numToCharList []     = []
numToCharList (n:ns) = numToFourChar n++numToCharList ns

numToFourChar::num->[char]
numToFourChar n = map (decode.(mod 256).(n div)) (map (256^) [3,2..0])

|| functions related to quadtrees

listsToPImage l = (root, sw, se, condenseQTree (makeQTree l s))
                  where s    = maxSize (max2 (#(hd l)) (#l))
                        root = vMake ( s/2, s/2)
                        sw   = vMake (-s/4,-s/4)
                        se   = vMake ( s/4,-s/4)

maxSize 0 = 0
maxSize 1 = 1
maxSize n = 2^(entier(log (n-1)/log 2)+1)

|| conversion from a list of pixel lists to a quadtree

makeQTree::[[pixel]]->num->qtree pixel
makeQTree [[]] tsize  = QNil
makeQTree []   tsize  = QNil
makeQTree [[a]] 1     = QLeaf a
makeQTree lists tsize = makeQNode (subt 0) (subt 1) (subt 2) (subt 3)
                        where
                        half   = tsize div 2
                        subt i = makeQTree (quarter i half lists) half

makeQNode::qtree pixel->qtree pixel->qtree pixel->qtree pixel->qtree pixel
makeQNode t0 t1 t2 t3 = QNode (avrQTree t0 t1 t2 t3) t0 t1 t2 t3

quarter::num->num->[[*]]->[[*]]
quarter i n = divide westEast southNorth
              where divide f g l = g (map f l)
                    westEast     = [take n, drop n]!(i mod 2)
                    southNorth   = [take n, drop n]!(i div 2)

|| functions for averaging a quadtree
```

```
average4::num->num->num->num->num
average4 a b c d = (a+b+c+d)/4

avrQTree::qtree num->qtree num->qtree num->qtree num->num
avrQTree t0 t1 t2 t3
   = average4 (valQTree t0) (valQTree t1) (valQTree t2) (valQTree t3)

valQTree::qtree num->num
valQTree QNil                = bgp
valQTree (QLeaf v)           = v
valQTree (QNode v t0 t1 t2 t3) = v

bgp = 0

|| Quadtree condensation

transformQTree::(qtree *->qtree *)->qtree *->qtree *
transformQTree f (QLeaf x) = f (QLeaf x)
transformQTree f QNil    = f QNil
transformQTree f (QNode v qt0 qt1 qt2 qt3)
   = f (QNode v (transformQTree f qt0) (transformQTree f qt1)
              (transformQTree f qt2) (transformQTree f qt3))

condenseQTree::qtree *->qtree *
condenseQTree = transformQTree fun
                where
                fun (QNode v (QLeaf x) (QLeaf x) (QLeaf x) (QLeaf x)) = QLeaf x
                fun (QNode v QNil QNil QNil QNil)           = QNil
                fun t          = t

|| geometric transformations

translatePImage d (root,sw,se,qt) = (vAdd d root, sw, se, qt)

scalePImage sc (root,sw,se,qt) = (scl root, scl sw, scl se, qt)
                                 where scl = vMul sc

rotatePImage th (root,sw,se,qt) = (rot root, rot sw, rot se, qt)
                                  where
                                  rot = vMap2 (rotx th, roty th)
                                  rotx th x y = x*(cos th)-y*(sin th)
                                  roty th x y = x*(sin th)+y*(cos th)

|| display functions

displayPImage origin size im
   = [] , if vYelement size=0
   = disp origin size im:
     displayPImage (vYup origin)(vYdown size) im , otherwise

disp origin size im
   = [] , if vXelement size=0
   = lookUpPImage origin im:disp (vXup origin) (vXdown size) im , otherwise

lookUpPImage p (root,sw,se,qt)
   = bgp          , if ~(vIsInside p vertexList)\/qt=QNil
   = valQTree qt , if isQLeaf qt\/(vLength sw<0.5)&(vLength se<0.5)
   = lookUpPImage p (nextRoot, vHalf sw, vHalf se, subTree qt), otherwise
     where
     vertexList = map ((vAdd root).vDouble) [sw,se,vNeg sw,vNeg se]
     s = sw $vAdd se
     w = sw $vSub se
     isWest  = vDirection (root $vSub p) (root $vAdd s $vSub p) < 0
     isSouth = vDirection (root $vSub p) (root $vAdd w $vSub p) > 0
     (nextRoot, i) = (root $vAdd sw, 0) , if isWest & isSouth
                   = (root $vAdd se, 1) , if ~isWest & isSouth
                   = (root $vSub se, 2) , if isWest & ~isSouth
                   = (root $vSub sw, 3) , if ~isWest & ~isSouth
     subTree (QNode v a b c d) = [a,b,c,d]!i

isQLeaf (QLeaf a) = True
isQLeaf qt = False

isQNode (QNode v a b c d) = True
isQNode qt = False

showPImage (r,sw,se,QNil)     = " QNil"
showPImage (r,sw,se,QLeaf a) = " (QLeaf "++show a++")"
showPImage (r,sw,se,QNode v a b c d)
   ="\n(QNode "++show v++showPImage (r,sw,se,a)++showPImage (r,sw,se,b)
    ++showPImage (r,sw,se,c)++showPImage (r,sw,se,d)++")"
```

A.9 Abstract Data Type:*vector*

```
||  YK 17/03/92
||  vector.m
||  Abstract data type for 2D vector handling

func2  == num->num->num
```

```
abstype vector
with vMake::(num,num)->vector
     vXelement::vector->num
     vYelement::vector->num
     vMap2::(func2,func2)->vector->vector
     vAdd::vector->vector->vector
     vSub::vector->vector->vector
     vMul::vector->vector->vector
     vDiv::vector->vector->vector
     vHalf::vector->vector
     vDouble::vector->vector
     vNeg::vector->vector
     vRecip::vector->vector
     vLength::vector->num
     vXup::vector->vector
     vXdown::vector->vector
     vYup::vector->vector
     vYdown::vector->vector
     vIsInside::vector->[vector]->bool
     vDirection::vector->vector->num
     showvector::vector->[char]

vector   == (num,num)

vMake     (x,y) = (x,y)
vXelement (x,y) = x
vYelement (x,y) = y

vFun1 fn (x,y) = (fn x, fn y)
vFun2 fn (x1,y1) (x2,y2) = (fn x1 x2, fn y1 y2)

vMap1 (fn1,fn2) (x,y) = (fn1 x, fn2 y)
vMap2 (fn1,fn2) (x,y) = (fn1 x y, fn2 x y)
vMap3 (fn1,fn2) (x1,y1) (x2,y2) = (fn1 x1 x2, fn2 y1 y2)
vMap4 (fn1,fn2) (x1,y1) (x2,y2) = (fn1 x1 y1, fn2 x2 y2)

vAdd  = vFun2 (+)
vSub  = vFun2 (-)
vMul  = vFun2 (*)
vDiv  = vFun2 (/)
vHalf = vFun1 (/2)
vDouble = vFun1 (*2)
vNeg  = vFun1 (neg)
vRecip = vFun1 (1/)

vLength (x,y) = sqrt (x^2+y^2)

vXup   (x,y) = (x+1,y)
vXdown (x,y) = (x-1,y)
vYup   (x,y) = (x,y+1)
vYdown (x,y) = (x,y-1)

vIsInside p vl
   = and [vDirection v1 v2 >= 0 |
          (v1,v2)<-zip2 (map (vSub p) vl) (map (vSub p) vlr)]
     where vlr = (tl vl)++[hd vl]

vDirection (x1,y1) (x2,y2) = x1*y2-x2*y1

showvector (x,y) = "("++show x++","++show y++")"
```

Appendix B. Paper Presented at Conference on Functional Programming Languages and Computer Architecture '93

Benchmarking Real-Life Image Processing Programs in Lazy Functional Languages

Y. Kozato
Canon Inc., 890-12 Kashimada,
Saiwai-ku, Kawasaki-shi, Kanagawa 211, Japan
e-mail: kozato@cis.canon.co.jp

G. P. Otto
Canon Research Centre Europe Ltd., 17/20 Frederick Sanger Road
Surrey Research Park, Guildford, Surrey GU2 5YD, U.K.
e-mail: otto@canon.co.uk

Abstract

This paper presents our practical experience gained from writing image processing programs in lazy functional languages. We give some benchmarking results comparing median filter operations written in C, Miranda[1] and Haskell. (The median filter is a common method for removing noise from images.) Also, using a profiling tool, we achieve remarkable improvement of the Haskell code. In particular, we show that the Haskell program runs in constant space, which is difficult to achieve in C without extra programming effort. Although the performance of lazy functional language is beginning to make them feasible for specific purposes such as rapid prototyping, better compilers and tools are still needed to encourage wider use in image processing applications.

1 Introduction

With the recent development of implementing lazy functional languages, applications of these languages to data intensive areas such as computer graphics [13, 4] and image processing [11, 12] are appearing. In these applications, the fact that lazy languages delay the evaluation of arguments until their values are required seems particularly suitable, because these applications typically involve a large amount of data and the efficiency can be much improved by reducing unnecessary computation.

In addition, programming these applications is rather tedious and cumbersome, also because a large amount of data should be dealt with. In lazy functional languages, however, since house-keeping tasks such as memory management and routine sequencing operations are embedded in the languages, programmers can be released from writing code for these tasks and concentrate on designing algorithms.

On the other hand, as a natural consequence of the fact that the languages do more work in order to be lazy, bigger overheads may be incurred and execution of programs may be rather slow compared with conventional languages, in which programs in these application fields are usually written. Unless some indication of execution speed and memory usage is presented, lazy functional languages will not be used for real applications because time and space efficiency is crucial for those data intensive areas. However, few experimental or practical results have been reported so far.

Based on the above consideration this paper gives some benchmarking results of real-life image processing programs by comparing median filtering [6] programs written in C, Miranda [24] and Haskell [9]. Median filtering has been selected because it is very well-known and widely used in image processing for removing noise in a gray image.

By the time of writing, a few Haskell compilers have become available. One particular compiler called hbc [1] comes with a heap profiling tool which enables us to analyse the cause of inefficiency of programs and to improve the code [19]. After we compare the versions in Miranda, Haskell, and C, the cause of inefficiency is analysed by using the heap profiling tool, and possible improvement of the Haskell version is tried out.

1.1 Benchmarking

Benchmarking is not a simple business. The purpose of the comparison is to give an idea of the performance of code written in the three languages, Miranda, Haskell and C. In this respect, this test may be quite different from comparing machine architectures or language implementations using the same source code, such as SPEC [21] or `nofib` benchmarking [14]. There are a few approaches to this kind of benchmarking:

- Use the code which a typical programmer[2] would write in each language. This may result in different algorithms between languages.
- Carry out optimisation to have the best possible code in each language. This may also lead to different algorithms.

[1] Miranda is a trademark of research Software Ltd.

[2] Note that a programmer who is typical of one group (e.g. professional numerical analysts) may not be typical of another group (e.g. computer science graduates).

ACM-FPCA'93-6/93/Copenhagen, DK
© 1993 ACM 0-89791-595-X/93/0006/0018...$1.50

- Take a reasonable algorithm, e.g. simple, fast, etc., and implement the algorithm in each language.

We will take the first approach, i.e. what a typical programmer would write. This will also highlight how programmers would be likely to write inefficient code unconsciously, and how such code can be improved. All in all, the comparison may not be fair, but will be sufficient to give a rough order of efficiency in the three languages.

2 The Algorithm - Median Filtering

Median filtering is in the category of local neighbourhood operations, in which the gray level of a pixel is replaced by the median of the gray levels in a neighbourhood of that pixel. For example in Figure 1 the value of the gray-hatched pixel (valued 64) is replaced by 22, since the pixel values of its 3×3 neighbourhood, i.e. within the 3×3 median mask, are 23, 5, 37, 12, 64, 8, 22, 20, and 54, and their median is 22.

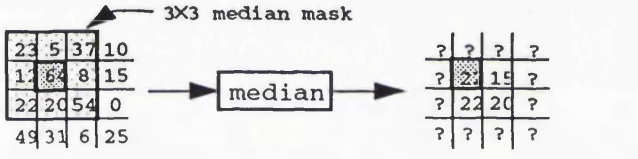
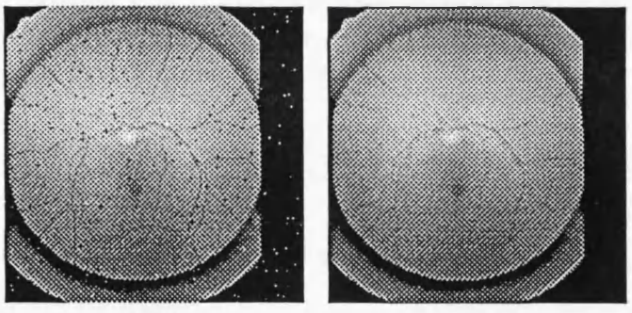


Figure 1: Pixel values of 3×3 median filtering

Figure 2 shows an example result of median filtering of a retina image. The image contains 256×256 pixels and the size of the median mask is 3×3. Noise in the original image is removed and the filtered image looks slightly smoother than the original.



(a) original image (b) filtered image

Figure 2: Example images

3 Median Filtering in Miranda

We implement median filtering using a higher-order function which is designed for general local neighbourhood operations. (It was originally designed for convolution, but it turned out to be more general.) Since various image processing operations have common computational structures, such as pointwise and local neighbourhood operations, use of higher-order functions together with polymorphism is particularly suitable for describing these operations. Just by defining the higher-order local neighbourhood function, a large number of individual operations can be implemented [12].

The algorithm can be divided into two parts: first produce an image whose pixel is a list of neighbouring pixels within a mask. We call this a *list image*. In the above example (Figure 1), the pixel at the position of the pixel value of 64 will become [23, 5, 37, 12, 64, 8, 22, 20, 54]. Then, the median of each pixel list is taken at each point.

In the following, we use Miranda's notation, but as you will see, Haskell code will not be much different.

3.1 Image representation

To begin with, we introduce a data structure called `interval` which represents the common structure for a row being a collection of pixels, and an image a collection of rows. This is important because once an operation is defined on an interval, then it can operate on a row, as well as an image. This gives us a very simple and modular way of constructing higher-dimensional operations from lower ones. The actual representation we take is *"a row as a pair of an x origin (the x coordinate where the row starts) and a list of pixels"* and *"an image as a pair of a y origin (the y coordinate where the image starts) and a list of rows"*. Image origins are expressed in the world coordinate system. The definition for a row and an image is given as follows:

```
interval * == (num,[*])
row *      == interval *
image *    == interval (row *)
```

3.2 Pointwise operations

The first part is to produce an image whose pixel is a list of neighbouring pixels. In order to implement a general local neighbourhood function, we need pointwise and translate operations.

Pointwise operations can be defined on intervals as follows:

```
unaryInterval::(*->**)->interval *->interval **
unaryInterval f (o,p) = (o, map f p)
```

An unary pointwise operation takes a function to be applied to a pixel, and an interval as its arguments. Obviously, it does not change the origin of an interval and the function is applied to each element of the list by function mapping.

```
binaryInterval::(*->**->***)->interval *
                  ->interval **->interval ***
binaryInterval f (o1,p1) (o2,p2)
  =(o, map2 f (drop (o2-o1) p1) p2), if o1 < o2
  =(o, map2 f p1 (drop (o1-o2) p2)), otherwise
     where o = max2 o1 o2
```

A binary pointwise operation takes a function and two intervals. The alignment of the two intervals is done by calculating a new origin by taking the maximum of the two origins, and only the overlapped part is calculated, which is implemented in the function `map2`.

Composing 2D image operations from the above operations defined on intervals is very straightforward. Since a row is an interval itself, pointwise operations for rows are identical to those for intervals. Pointwise operations for images can be composed by using function composition operators. These definitions follow:

```
unaryRow  = unaryInterval
binaryRow = binaryInterval

unaryPointwise  = unaryRow.unaryRow
binaryPointwise = binaryRow.binaryRow
```

3.3 Image translation

The next function is for shifting an image. This function is used to have access to neighbouring pixels. Since the origin is separated from the pixel data part in the interval definition, translation of an image is very simple as follows:

```
translateInterval d (o, p) = (o+d, p)

translateRow = translateInterval
translateImage x y (o, p)
  = translateRow y (o, map (translateRow x) p)
```

3.4 The higher-order local neighbourhood function

Using the above defined functions, we give a definition of a higher-order function for general local neighbourhood operations. The function is primarily designed for convolution which is the most common local neighbourhood operations and which can be defined as a sum-of- products operation. Therefore, the function takes two function arguments for the "sum" and "product" operations.

The following function `local` takes the two functions `mul` and add. `mul` defines the operation between a pixel in a shifted image and an element in a mask, and add defines the operation between the results of `mul`s. The subdefinition `accum` represents accumulation of the products. The data arguments, `mask` and `im` are assumed to be of the type `image *` defined above.

```
local mul add mask im
 = accum [prod (element mask p)
   (shiftInterval p) | p<-domain mask]
   where
   accum   = foldl1 add
   prod x  = unaryInterval (mul x)
   shiftInterval p
           = translateInterval
              (p-((leng mask) div 2)) im

domain  = index.snd
leng    = (#).snd
element = (!).snd
```

This local function is a higher-order function for general local neighbourhood operations. Therefore, individual operations can be implemented by passing operations and mask data to the function. For example, genuine convolution can be defined as:

```
convolveRow = local (*) (binaryRow (+))
convolveImage
= local convolveRow (binaryPointwise (+))
```

3.5 A median filter function

Using the above local neighbourhood function, median filtering is defined in two steps: the first step is to produce a list image, i.e. an image whose pixels are lists of neighbouring pixels, and the second is to take the median of each pixel list in a pointwise manner.

```
medianImage mask
  = (unaryPointwise median).(localHist mask)
median list
  = (sort list)!(#list div 2)

localHist
  = local (localHistRow) (binaryPointwise (++))
localHistRow = local f (binaryRow (++))
               where f a b = [b]
```

The function `localHist` produces a list image by passing the append function `(++)` to the higher-order local neighbourhood function. The function `median` is a pointwise function which takes a list image and returns an image whose pixels are the medians of the list image. `medianImage` is the top-level function to be used by the user.

3.6 Image I/O

I/O functions are essential for processing real images, such as the ones stored in a file or input from input devices. We use Sun's rasterfile [22] format as an example image file format. Rasterfile format consists of three parts: a header containing 8 integers, a (possibly empty) set of colour map values, and the pixel image stored line by line in increasing y order. See appendices for the actual code for I/O. Please note that there is no compatibility between 32 bit integers commonly used in the UNIX/C environment and `num` in Miranda or `Int` in Haskell. So some conversion code between these is necessary.

4 The Competition

4.1 Contender No.1 — *Miranda*

The full Miranda code for the median filter is given in Appendix A. (Originally 107 lines, but reformatted slightly to fit in this paper.) As we have seen in the above section, this code is based on the higher-order function for general local neighbourhood operations which is designed not solely for median filtering. Therefore, no efficiency issues have been considered.

The following is a list of features of the code:

- The first two lines are the magic words to make Miranda code available as a command at the UNIX shell level.
- The second line is the `main` expression to be evaluated. The function `main` has been implemented to describe the "input → process → output" sequence. The header and the colour map are copied from the input to the output file. The pixel data part is converted to a list of numbers, then to the image data structure with an origin, processed by the function passed as an argument, converted back to a list of numbers and then to a list of characters for file output.
- The size of the input image is maintained. For the boundary pixels, a background value, 0 in this case, is filled in.

4.2 Contender No.2 — *Haskell*

The Haskell code (Appendix B) was based on the Miranda code. This means that this Haskell code has not been written to utilise the facilities provided by Haskell. For example, arrays in Haskell could have been used instead of lists, which might have improved the efficiency. In addition, the low-level conversion between 32 bit integers and Ints is done in Haskell. If the facility such as I/O monads [7] which allow C functions to deal with low-level jobs without violating referential transparency had been used, the efficiency might have been improved slightly.[3]

The changes made are:

- The `main` function has been added, which is of the type `Dialogue` defined as:

  ```
  type Dialogue = [Response] -> [Request]
  ```

- The `ip` function forms the main part of the job, which takes an input character list, an image processing function, and returns an output list of characters. This is akin to the `main` function defined in the Miranda version.

- The `splitList` function has been changed to utilise the library functions, `chopList` and `splitAt`, in the hope that library functions could be more efficient than user-defined functions. For `chopList`, see [2]. For this purpose the module `ListUtil` is imported.

- Two functions (`rep` and `subscripts`) are defined as these are defined in Miranda but not in Haskell. `subscripts` is equivalent to `index` in Miranda.

Otherwise, the algorithms is identical to the Miranda version. It is noticeable how much Haskell inherits Miranda's facilities. The code is compiled by the `hbc` compiler (using the `-O` flag) and an executable file of `a.out` format is generated.

4.3 Contender No.3 — *C*

The C code for median filtering shown in Appendix C has been freshly written for this benchmarking purpose. Therefore, the C version is the most efficiency conscious of the three contenders. Appendix C shows the median function only. The complete program (including file I/O, various house-keeping and comments) has 215 lines of code. The algorithm starts by generating a local histogram within `radius` and get the `mid`'th element from the histogram. Then the input image is scanned, incrementally updating the local histogram and median. The size of input image is maintained and 0s are filled in the image boundary. The program is compiled with the `-O4` option to optimise at the maximum level.

4.4 A result

The three versions are made into commands at the UNIX shell level, and the UNIX facility `/bin/time` is used to measure the speed. The three programs ran on a Sun Microsystems' SPARCstation IPX with main memory of 16 MB. Table 1 shows a result of the test using an 8 bit gray level image of the size 256 × 256 with the median mask of the size 3 × 3.[4] The slow-down ratio indicates how many times Miranda and Haskell are slower assuming the speed of C to be 1.

Table 1: A benchmark result

Language	Time in seconds	Slow-down ratio
Miranda	409.8	273
Haskell	54.3	36
C	1.5	1

As mentioned, the competition may not be regarded as fair. This is because the Miranda code was not intended to be a median filter but uses the general local neighbourhood function without consideration of efficiency; the Haskell code was converted from Miranda and does not use Haskell's specific facilities; while the C code was written with efficiency in mind. However, despite all the unfairness, the result gives a certain idea of speed which average programmers could expect. The Miranda implementation is very stable and reliable, but is now at least a few years old, while the development of Haskell is currently an on-going project and various ideas for improvement are being tested [8, 16]. Therefore, we can expect more speed up in Haskell implementations, but they may never be as fast as C.

5 Heap Profiling and Improvement of the Haskell Code

5.1 The heap profile of the Haskell version (Version 1)

The lack of debugging and profiling tools has been one of the obstacles for wide use of lazy functional languages. Compared with, for example, C which has a number of convenient facilities to debug, analyse and improve code, lazy functional languages have had almost nothing to allow programmers to do this kind of job. In Miranda, for instance, there is a facility to report some statistics such as the number of reductions, cells, and garbage collections, and the CPU time [17], but this does not show causes of inefficiency or bugs. A graphical tool to exhibit runtime behaviour of Miranda code has been developed by Taylor [23], but it is a test implementation and not available yet.

The recent development of profiling tools for Haskell by Runciman and Wakeling [18] and by Sansom and Peyton Jones [20] seems interesting, because they provide information on the dynamic space and time behaviour of programs. Using these tools, it is possible to look at which parts are costly which may suggest possible improvements to the programmer. In fact, Runciman et al. reported that the tools can be used to bring about a dramatic reduction in the space consumption using a few examples [18, 19]. In the following, we use the heap profiling tool by Runciman et al.

The heap profiling result of the median filter code (Appendix B) is shown in Figure 3. In the graph the diamond marks on the time axis indicate garbage collections. It shows monotonically linear increase of space until the last moment and more than a megabyte is eventually used. This looks extremely memory-hungry.

[3] For the Haskell programs presented in this paper, 10–15% of the runtime was taken by I/O (measured by replacing the median filter with the identity operation).

[4] We use Miranda V2.014 and hbc release 0.999.3.

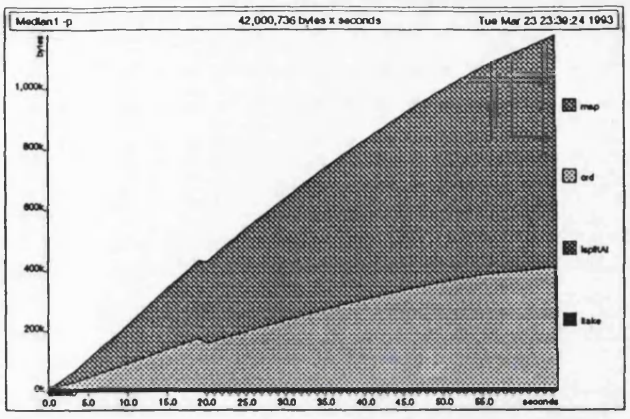


Figure 3: A heap profile graph of median filtering

5.2 A culprit spotted

Because the code handles an image as a list of lists of pixels and lists are treated lazily through input, process and output, and the process is a simple local neighbourhood operation which does not rely on data at distant positions, the profile should have been flatter. Figure 3 suggests that there is some code which accumulates the whole image at once.

Careful examination of the code in Appendix B has been conducted and the culprit has been found in the ip function which is responsible for the whole sequence from file input, image processing, to file output. The following code is in the subdefinitions of ip:

```
im2l (o,l)
 = (map chr.concat)(rep o bgr ++ map flatten l ++
    rep (height - o - length l) bgr)
flatten (o,l)
 = (rep o bgp)++l++(rep (width - o - length l)bgp)
```

Input and output streams are long lists of characters, but image processing is defined on the 2D interval structure, which is a pair of an origin and a list of elements. `im2l` is designed to convert a 2D interval structure to a plain list, and `flatten` converts a 1D interval to a plain list. The need for these functions comes from the intention to have equal sizes of input and output, so that the results of the three versions can be made identical. Thus, in the original implementation of median filtering itself in section 3.5, this issue was not considered at all.

As discussed in Chapter 23 of [15] and in [10], there is a lot of subtlety in behaviour of lazy functional programs. Our code may be a good example of the scheduling problem addressed by Hughes. In the `im2l` function, there is the code fragment "`length l`". Usually `length` itself consumes a list but does not keep the whole list. But in the above code, because the list 'l' is shared between `flatten` and `length`, and the `length` function is not evaluated until the end, the content of the whole list is kept until `length` is evaluated. And here, this "`l`" is *the whole image!*

5.3 Modification of the code (Version 2)

The code has been modified to overcome the problem of accumulating the whole image. The interval structure has been modified to become a triple of (origin, length, list). In this particular code, the length of a row and the length of an image are provided as `width` and `height` respectively in the header information of a rasterfile. So, there is almost no overhead in obtaining this information. The new interval structure in Haskell is:

```
type Interval a = (Int,Int,[a])
```

The subdefinitions for the improved ip function are:

```
l12im l1
  = (0,height,map fn l1) where fn x = (0,width,x)
im2l (o,ln,l)
  = (map chr.concat)(rep o bgr ++ map flatten l ++
     rep (height - o - ln) bgr)
flatten (o,ln,l)
  = (rep o bgp) ++ l ++ (rep (width - o - ln) bgp)
```

The unary and binary pointwise operations, and interval translation, also need modification:

```
unaryInterval f (o,ln,p) = (o,ln, map f p)
binaryInterval f (o1,ln1,p1) (o2,ln2,p2)
  |(o1<o2) = (o,ln,zipWith f (drop(o2-o1)p1)p2)
  |otherwise
           =(o,ln,zipWith f p1(drop(o1-o2)p2))
           where
           o = max o1 o2
           ln = max 0 ((min(o1+ln1)(o2+ln2))-o)

translateInterval d (o,ln,p) = (o+d,ln,p)
```

Also modifications to `local` and `makeMask` are necessary, as well as related functions such as `domain`, `element`, etc. A new definition of these functions is as follows:

```
local mul add mask im
  = accum [prod (element mask p)
           (shiftInterval p) | p<-domain mask]
    where accum  = foldl1 add
          prod x = unaryInterval (mul x)
          shiftInterval p
                 = translateInterval
                    (p-((second mask) `div` 2))im

domain         = subscripts.third
element        = (!!).third
second (a,b,c) = b
third (a,b,c)  = c

makeMask n
  = fn (map fn (rep n (rep n 0)))
    where fn x = (0,n,x)
```

In any of the above functions, it can be said that the modifications are relatively trivial. There is no change in the fundamental algorithm, but just addition of an extra parameter to save calculations.

5.4 Improved results

With the above relatively minor modifications, a new heap profile is shown in Figure 4. This graph has been drawn in the same scale as Figure 3 to highlight the difference. It is remarkable to see that the program runs in a constant space as garbage is constantly collected, and how little memory the

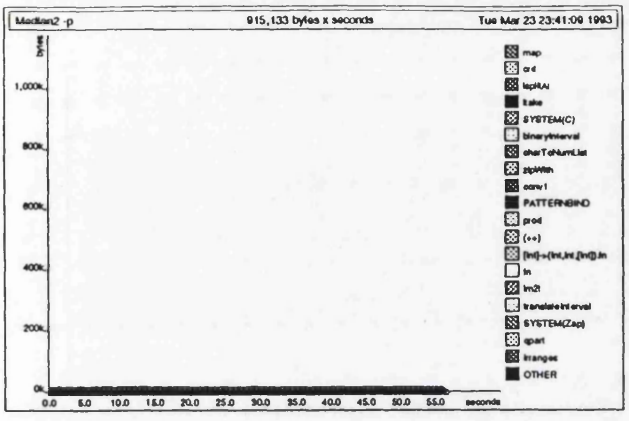


Figure 4: An improved heap profile (Version 2)

program consumes just by those modifications. Without the profiling tool, it would have been much harder to notice the inefficient memory usage of the previous version.

With regard to the execution time, the improved version took 48.3 sec (cf. the original version 54.3 sec).

6 Further improvement

We have further tried to improve the Haskell code for median filtering to get the best possible speed, as follows:

6.1 Eliminate identical operations (Version 3)

The basic algorithm of our median filtering is to generate a list image. The length of each pixel list is equal to the square of the mask size. Then, the median of each pixel (list) is taken as an unary pointwise operation. Therefore, the function to take a median is the same all through the list image, but a compiler does not spot this fact. Lazy evaluation shares the same expressions only when these occur in the same function call. In our original code the index of a median is calculated every time, e.g. 65536 times for a 256×256 image. The following is the original code to take a median:

```
median list = (sort list)!!((length list) `div` 2)
```

where the length does not change, e.g. if a median mask is 3×3 the index, i.e. "`(length list) `div` 2`", is 4 and is constant all through the operations. Utilising this knowledge, it is possible to modify the code, so that the index is calculated only once before the median function is called. The new definition follows:

```
medianImage n
  = (unaryPointwise (rankFilter m)) .
                (localHistImage (makeMask n))
    where m = n*n `div` 2
          rankFilter m list = (sort list)!!m
```

where n is the mask size. Since the new function takes a rank order as a parameter, it works as a general rank filter rather than only a median. Hence, the name has been changed.

This reduces the time slightly, to 46.2 sec.

6.2 Whether to fold up a list from left or right? (Version 4)

We defined the higher-order local neighbourhood function (`local`) to take two functions (`mul` and `add`) as parameters and accumulation is defined within the function to be `foldl add`. As discussed in Chapter 6 of [3], fold operations behave very subtly; for functions, such as `(+)` or `(*)`, that are strict in both arguments and can be computed in constant time and space, `foldl` is more efficient. Whereas for functions, such as `(&)` or `(++)`, that are non- strict in some argument, `foldr` is often more efficient. Therefore, it may be a mistake to hard-code the direction of accumulation within the function definition of convolution.

Based on the above consideration, the new definition of `local` below takes an accumulation instead of an add operation. Since an append operator `(++)` is passed as an argument in order to produce a list image, accumulate from the right should be more efficient. The modified code is the following:

```
local mul accum mask im
  = accum [prod (element mask p)
           (shiftInterval p) | p<-domain mask]
    where
    prod x = unaryInterval (mul x)
    shiftInterval p
        = translateInterval
           (p-((second mask) `div` 2)) im

localHistImage
  = local f accum
    where f     = localHistRow
          accum = foldr1 (binaryPointwise (++))

localHistRow
  = local f accum
    where f a b = [b]
          accum = foldr1 (binaryRow (++))
```

This reduces the execution time slightly, to 44.6 sec.

6.3 Use of cons instead of append (Version 5)

It is generally quicker to use cons `(:)` instead of append `(++)` to attach an element to a list, because in order to append two lists, the one in front should be traversed. In the median filtering it is possible to use cons in the 1D local histogramming operation. The modified function is:

```
localHistRow
  = conv3 f accum
    where
    f a b = b
    accum (r:rs)
     =foldr (binaryRow (:))(unaryRow (:[]) r) rs
```

The execution time after this modification is 44.4 sec.

6.4 Summary of the improvement

Table 2 summarises the result of the improvement.

Table 2: Comparison of execution times

Versions	Time in seconds	Slow-down ratio
Version 1 (original)	54.3	36
Version 2 (eliminate calculation of length)	48.3	32
Version 3 (eliminate index calculation)	46.2	31
Version 4 (accumulate from the right)	44.6	30
Version 5 (Use (:) instead of (++))	44.4	30
C Version	1.5	1

7 Discussion

7.1 Are lazy languages inefficient?

The benchmark result given in Table 2 shows that lazy functional languages are much slower than C. In particular Miranda runs a few hundred times slower, which may be regarded as unrealistic for image processing applications. However, as shown in Table 2, with a little effort to improve the Haskell code, we have achieved a slow-down ratio of about 30. This speed can be considered to be not unrealistic for some purposes such as rapid prototyping.

We have to consider two factors here:

1. There are a number of advantages of using lazy functional languages such as facilities to improve modularity [12]. So, if we consider lazy functional languages as a rapid prototyping or specification tool they are well worth considering.

2. Implementations of lazy functional languages are under active research and a large amount of improvement can be expected. Thus, based on the benchmarking results we may be able to say that lazy functional languages are only a few 10s times slower than C.

However, although lazy functional languages are beginning to be feasible, they are still rather slow for wide use in the image processing community where the demand for efficiency is high. We have not been able to investigate the remaining causes of the slowness yet. One possibility would be concerning type classes and overloading in Haskell. We might be unconsciously writing code which uses unnecessarily excessive overloading, which may reduce efficiency. We need further investigation and, to do so, better analysis tools are desirable.

7.2 When lazy functional languages can beat imperative languages

Using the current code, if an image to be processed becomes very large lazy functional languages can beat imperative languages. As shown in the improved heap profile (Figure 4), the Haskell version runs in constant space because data is always produced and consumed and garbage is continuously collected. Whereas, the C version allocates memory for both input and output images, so that the increase of memory is proportional to the number of pixels in the images. If we were to process a very large image in the C version, it would become slower above a certain image size due to paging and various house-keeping operations carried out by the system. When an image becomes even larger, it will be impossible to process the image by the current C code. For example, a panchromatic SPOT satellite image [5] contains 6000×6000, i.e. 36 million pixels; a 400 dpi colour scan of an A4 page contains about 45 Mb. In order to avoid this problem, programmers need to do extra work to get the C code working well on large images, typically by rewriting the code to process an image line-by-line or chunk-by- chunk.

7.3 Use of arrays

Miranda does not have arrays, but Haskell does. However, we did not use them here because, at the time of writing, arrays were not supported well in the available compilers.

7.4 Caveats of lazy functional languages

It may be generally true that lazy functional languages are adequately efficient even with a naive way of programming. However, as we have experienced, if more efficiency is to be sought, optimising code may not be an easy job. There is a lot of subtlety in the behaviour of lazy functional programs and the behaviour is not obvious from the code itself.

In order to write more efficient code, programmers still have to work out the sources of inefficiency. For example, to avoid repeated calculations, they will have to add an extra parameter to a function if its value is known before the function calls. Compilers will spot redundancy within a function but not across different function calls. Also, even simple operations (such as length) can be very expensive if they cause a large amount of data to be retained in memory.

When analysing code, we have found profiling tools very convenient. Though profiling lazy functional programs is not a simple task [20], judging from the current status, i.e. very few of them, any tools may be useful. We would expect emergence of more convenient tools to encourage wider use of lazy functional languages.

8 Conclusions

We have presented some benchmarking results of median filtering written in Miranda, Haskell and C. Although we have presented only a small set of benchmarks with a small set of languages, the result shows that lazy functional languages are feasible for specific purposes such as rapid prototyping.

Using the heap profiling tool, we have shown that the lazy functional version runs in constant space, which is difficult to achieve using imperative languages without extra programming effort.

However, the performance is not yet sufficient for wide use of the languages for real image processing applications. In order to encourage wider use, we need better tools to help identify ways in which the programs can be written more efficiently. Also, we need better compilers to generate efficient machine code from what we have written.

References

[1] Augustsson, L., *Haskell B. user's manual (draft)*, July 1992.

[2] Augustsson, L. and T. Johnsson, *Lazy ML user's manual*, Chalmers University, August 1992.

[3] Bird, R. and P. Wadler, *Introduction to Functional Programming*, Prentice-Hall, 1988.

[4] Checkland, I. and C. Runciman, *Development of a Prototype Geometric Modelling System using a Functional Language*, Technical Report, Department of Computer Science, University of York, York, 1991.

[5] Chevrel, M., M. Courtis, and G. Weill, "The SPOT satellite remote sensing mission," *Photogrammetric Engineering and Remote Sensing*, vol. 47, no. 8, pp. 1163 – 1171, 1981.

[6] Gonzalez, R. C. and P. Wintz, *Digital Image Processing (Second Edition)*, Addison-Wesley, 1987.

[7] Hall, C., K. Hammond, W. Partain, S. L. Peyton Jones, and P. Wadler, "The Glasgow Haskell Compiler and Grip: A Retrospective," in *Draft Proceedings of Fifth Annual Glasgow Workshop on Functional Programming*, Ayr, Scotland, July 1992.

[8] Hall, C., K. Hammond, W. Partain, S. L. Peyton Jones, and P. Wadler, *Abstracts of GRIP/GRASP-related papers and reports 1990 and after*, University of Glasgow, Glasgow, February 1992.

[9] Hudak, P. et al., "Report on the Programming Language Haskell — A Non-strict Purely Functional Language (Version 1.2)," *SIGPLAN NOTICES*, vol. 27, no. 5, ACM, May 1992.

[10] Hughes, R. J. M., *Parallel Functional Languages Use Less Space*, 1984.

[11] Kozato, Y. and G. P. Otto, "Geometric Transformations in a Lazy Functional Language," in *Proceedings of 11th International Conference on Pattern Recognition*, vol. IV, pp. 128 – 132, The Hague, August 1992.

[12] Kozato, Y., *Lazy Image Processing: An Investigation into Applications of Lazy Functional Languages to Image Processing*, PhD thesis, University College London, October 1992.

[13] Parsons, M. S., "Image Representations Using Miranda Laws," *Computer Graphics Forum*, vol. 8, pp. 99 – 106, North-Holland, 1989.

[14] Partain, W., "The nofib Benchmark Suite of Haskell programs (pre-release Ayr draft)," in *Draft Proceedings of Fifth Annual Glasgow Workshop on Functional Programming*, pp. 207 – 215, Ayr, Scotland, July 1992.

[15] Peyton Jones, S. L., *The Implementation of Functional Programming Languages*, Prentice-Hall International, 1987.

[16] Peyton Jones, S. L., *Functional programming at Glasgow University*, Dept of Computing Science, Glasgow University, April 1992.

[17] Research Software Limited, *Miranda System Manual (V2.014)*, 1989.

[18] Runciman, C. and D. Wakeling, "Profiling a Compiler (Early Draft)," in *Draft Proceedings of Fifth Annual Glasgow Workshop on Functional Programming*, pp. 216 – 224, Ayr, Scotland, July 1992.

[19] Runciman, C. and D. Wakeling, *Heap Profiling of Lazy Functional Programs*, Technical Report, Department of Computer Science, University of York, York, 1992.

[20] Sansom, P. and S. L. Peyton Jones, "Profiling Lazy Functional Languages (Working Paper)," in *Draft Proceedings of Fifth Annual Glasgow Workshop on Functional Programming*, pp. 238 – 251, Ayr, Scotland, July 1992.

[21] SPEC, *The January 1992 SPEC/CINT92/CFP92 announcement*, Standard Performance Evaluation Corporation, 1992.

[22] Sun Microsystems Inc., *SunOS Reference Manual*, 1987.

[23] Taylor, J., "A System For Representing The Evaluation Of Lazy Functions," Technical Report, no. 522, Dept. of Computer Science, Queen Mary and Westfield College, London, February 1991.

[24] Turner, D. A., "Miranda: A non-strict functional language with polymorphic types," in *Lecture Notes in Computer Science*, vol. 201, Springer, 1985.

A Median Filtering Code in Miranda

```
#! /usr/local/bin/mira -exp
main "gantei.ras" (medianImage mask) "out.ras"

|| The "with an Origin" Version
interval * == (num,[*])
row *      == interval *
image *    == interval (row *)

|| rasterfile I/O
main::[char]->(image num->image num)->[char]->
                                        [sys_message]
main infile proc outfile
 =[Tofile outfile (header++cmap++data),
    Closefile outfile]
   where
   input      = read infile
   header     = take headerLen input
   cmap       = take cmapLen
                    (drop headerLen input)
   cmapLen = ras_maplength hdr
   hdr        = charToNumList header
   data       = (im2l.proc.l2im)
                (drop (headerLen+cmapLen) input)
   l2im       = ll2im.(splitList width).(map code)
   width      = ras_width hdr
   height     = ras_height hdr
   ll2im ll   = fn (map fn ll) where fn x = (0,x)
   bgr        = rep width bgp
   bgp        = 0
   im2l (o,l) = map decode ((concat (rep o bgr)++
                    concat (map flatten l)++
                    concat (rep (height-o-#l) bgr)))
   flatten (o,l)= (rep o bgp)++l++
```

```
                              (rep (width-o-#l) bgp)
splitList::num->[*]->[[*]]
splitList n [] = []
splitList n l  = [take n l]++
                              (splitList n (drop n l))
[ras_magic, ras_width,ras_height, ras_depth,
 ras_length,ras_type, ras_maptype,ras_maplength]
   = [(!i)|i<-[0..7]]

headerLen     = 32
ras_magic_num = 1504078485
rmt_equal_rgb = 1

charToNumList::[char]->[num]
charToNumList [] = []
charToNumList cs
 = fourCharToNum (take 4 cs):charToNumList
                                             (drop 4 cs)
fourCharToNum::[char]->num
fourCharToNum cs
 = foldl1 (+) (map2 (*) (map code cs)
   (map (256^) [3,2..0]))

|| pointwise operations
unaryInterval::(*->**)->interval *->interval **
unaryInterval f (o,p) = (o, map f p)

binaryInterval::(*->**->***)->interval *
                  ->interval **->interval ***
binaryInterval f (o1,p1) (o2,p2)
 = (o, map2 f (drop (o2-o1) p1) p2) , if o1 < o2
 = (o, map2 f p1 (drop (o1-o2) p2)) , otherwise
   where o = max2 o1 o2

unaryRow  = unaryInterval
binaryRow = binaryInterval

unaryPointwise::(*->**)->image *->image **
unaryPointwise f  = unaryRow (unaryRow f)

binaryPointwise::(*->**->***)->image *->image **
                                          ->image ***
binaryPointwise f = binaryRow (binaryRow f)

|| image translation
translateInterval::num->interval *->interval *
translateInterval d (o,p) = (o+d,p)
translateRow = translateInterval

|| convolution
local mul add mask im
 = accum [prod (element mask p) (shiftInterval p)
                                  | p<-domain mask]
   where
   accum  = foldl1 add
   prod x = unaryInterval (mul x)
   shiftInterval p
      = translateInterval (p-((length mask) div 2))
                                                 im
domain    = index.snd
length    = (#).snd
element   = (!).snd

|| median filter
mask = makeMask 3

makeMask::num->image num
makeMask n
 = fn (map fn (rep n (rep n 0))) where fn x = (0,x)

medianImage mask
 = (unaryPointwise median).(localHistImage mask)
median list = (sort list)!(#list div 2)

localHistImage
 = local (localHistRow) (binaryPointwise (++))
localHistRow
 = local f (binaryRow (++)) where f a b = [b]
```

B Median Filtering Code in Haskell

```
module Main (main) where
import ListUtil
import QSort

main :: Dialogue
main resps
 = [ReadFile "gantei.ras",
    WriteFile "out1.ras"
     (case resps!!0 of
     Str contents -> ip contents (medianImage mask)
     Failure ioe  -> "Error")]

-- The "with an Origin" Version
type Interval a = (Int,[a])
type Row a      = Interval a
type Image a    = Interval (Row a)

-- rasterfile I/O
ip::[Char]->(Image Int->Image Int)->[Char]
ip input proc
 = header++cmap++pixl       where
      header = take headerLen input
      cmap   = take cmleng (drop headerLen input)
      cmleng = ras_maplength hdr
      hdr    = charToNumList header
      pixl   = (im2l.proc.l2im)
               (drop (headerLen+cmleng) input)
      l2im   = l12im.(splitList width).(map ord)
      width  = ras_width hdr
      height = ras_height hdr
      l12im l1 = fn (map fn l1) where fn x = (0,x)
      bgr    = rep width bgp
      bgp    = 0
      im2l (o,l) = (map chr . concat)
                   (rep o bgr++map flatten l++
                    rep (height-o- length l) bgr)
      flatten (o,l) = (rep o bgp)++l++
                      (rep (width-o- length l) bgp)

splitList::Int->[a]->[[a]]
splitList n l = chopList (splitAt n) l

-- Rasterfile I/O omitted; similar to Miranda's

-- pointwise operations
unaryInterval::(a->aa)->Interval a->Interval aa
unaryInterval f (o,p) = (o, map f p)

binaryInterval::(a->aa->aaa)->Interval a
```

```
              ->Interval aa->Interval aaa
binaryInterval f (o1,p1) (o2,p2)
 | (o1 < o2) = (o, zipWith f (drop (o2-o1) p1) p2)
 | otherwise = (o, zipWith f p1 (drop (o1-o2) p2))
   where o = max o1 o2

unaryRow = unaryInterval
binaryRow = binaryInterval

unaryPointwise::(a->aa)->Image a->Image aa
unaryPointwise f  = unaryRow (unaryRow f)
binaryPointwise::(a->aa->aaa)->Image a->Image aa
                                    ->Image aaa
binaryPointwise f = binaryRow (binaryRow f)

-- image translation
translateInterval::Int->Interval a->Interval a
translateInterval d (o,p) = (o+d,p)
translateRow = translateInterval

-- convolution
local mul add mask im
 = accum [prod (element mask p) (shiftInterval p)
                                  | p<-domain mask]
  where
  accum  = foldl1 add
  prod x = unaryInterval (mul x)
  shiftInterval p
   = translateInterval (p-((leng mask) `div` 2)) im
domain    = subscripts.snd
leng      = length.snd
element   = (!!).snd

-- median filter
mask = makeMask 3
makeMask::Int->Image Int
makeMask n
 = fn (map fn (rep n (rep n 0))) where fn x = (0,x)

medianImage mask
 = (unaryPointwise median).(localHistImage mask)
median list = (sort list)!!((length list) `div` 2)

localHistImage
 = local (localHistRow) (binaryPointwise (++))
localHistRow = local f (binaryRow (++))
                                  where f a b = [b]
rep :: Int -> b -> [b]
rep n x = take n (repeat x)

subscripts :: [a] -> [Int]       -- Miranda index
subscripts xs = f xs 0
                where f []     n = []
                      f (_:xs) n = n : f xs (n+1)
```

C Median Filtering Code in C

```
unsigned char *
med(in, xsize, ysize, msize)
unsigned char *in;
int xsize, ysize, msize;
{
  unsigned char *out
    = (unsigned char *) malloc(xsize * ysize);
  int y;
  int radius = msize / 2;

  /* fill in edge pixels with background value */
    memset((char*)out, BGP, xsize*ysize);

  /* now do the medians */
  for (y = radius; y < ysize-radius; y++){
    int hist[256];
    int m;
    int s;        /* sum of hist[0..m] inclusive */
    int offset = xsize*y;
                  /* position of centre of mask */
    int mid = (msize*msize+1)/2;
    int i,j;
    /* first build the histogram etc */
    for (i = 0; i < 256; i++)
      hist[i] = 0;
    for (j = -radius; j <= radius; j++)
      for (i = -radius; i <= radius; i++)
        hist[in[offset+i+j*xsize]]++;
    /* find the median */
    s = 0;
    m = -1;
    do {
      m++;
      s += hist[m];
    } while (s < mid);
    /* now m = median */
    out[offset] = m;

    /* now scan along the row, incrementally
                                          updating */
    for (j = radius+1; j < xsize-radius; j++){
      /* update histogram */
      offset -= radius;
      for (i = -radius; i <= radius; i++){
        int x = in[offset+i*xsize];
        hist[x]--;
        if (x <= m)
          s--;
      }
      offset += 2*radius + 1;
      for (i = -radius; i <= radius; i++){
        int x = in[offset+i*xsize];
        hist[x]++;
        if (x <= m)
          s++;
      }
      offset -= radius;
      /* adjust m to be median again */
      while (s-hist[m] >= mid){
        s -= hist[m];
        m--;
      }
      while (s < mid){
        m++;
        s += hist[m];
      }
      /* now m = median */
      out[offset] = m;
    }
  }
  return out;
}
```

References

Abelson85a

Abelson, H. and G. J. Sussman, *Structure and Interpretation of Computer Programs*, The MIT Press, 1985.

Adobe Systems Inc.90a

Adobe Systems Inc., *Postscript Language Reference Manual*, Addison Wesley, 1990.

Allsop91a

Allsop, M., *Applications of Functional Methods to Image Processing*, Macquarie University, Sydney, Australia, July 1991.

The AQUA Team93a

The AQUA Team, *The Glorious Haskell Compilation System Version 0.19 User's Guide*, Department of Computing Science, University of Glasgow, December 1993.

Arvind89a

Arvind, R. S. Nikhil, and K. K. Pingali, "I-Structures: Data Structures for Parallel Computing," *ACM Trans. on Programming Languages and Systems*, vol. 11, no. 4, pp. 598 - 632, ACM, October 1989.

Augustsson84a

Augustsson, L., "A Compiler for Lazy ML," in *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, pp. 218 - 227, ACM, 1984.

Augustsson87a

Augustsson, L., *Compiling Lazy Functional Languages, Part II*, PhD Thesis, Chalmers University of Technology, 1987.

Augustsson92a

Augustsson, L., *Haskell B. user's manual (draft)*, July 1992.

Augustsson92b

Augustsson, L. and T. Johnsson, *Lazy ML user's manual*, Chalmers University, August 1992.

Backus78a

Backus, J., "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *Communications of the ACM*, vol. 21, no. 8, pp. 613 - 641, ACM, 1978.

Barendregt84a

Barendregt, H. P., *The Lambda Calculus - Its Syntax and Semantics (Revised Edition)*, North-Holland, 1984.

Bird88a

Bird, R. and P. Wadler, *Introduction to Functional Programming*, Prentice-Hall, 1988.

Bobrow90a

Bobrow, D. G., L. G. Demichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon, "Common Lisp Object System," in *CommonLisp - The Language (Second Edition)*, Steele Jr., G. L., pp. 770 - 864, Digital Press, 1990.

Böhm92a

Böhm, A. P. W., R. R. Oldehoeft, D. C. Cann, and J. T. Feo, *SISAL Reference Manual - Language Version 2.0*, 1992.

Breuel92a

Breuel, T. M., "Functional Programming for Computer Vision," in *SPIE Conference on Image Processing*, San Jose, February 1992.

Burton88a

Burton, F. W. and M. M. Huntbach, "Lazy Evaluation of Geometric Objects," *IEEE CG&A*, vol. 4, no. 1, pp. 28 - 33, IEEE, January 1988.

Cann92a

Cann, D., "Retire Fortran? A Debate Rekindled," *Communications of ACM*, vol. 35, no. 8, pp. 81 - 89, August 1992.

Catmull80a

Catmull, E. and A. R. Smith, "3-D Transformation of Images in Scanline Order," in *SIGGRAPH 80*, pp. 279 - 285, 1980.

Chambers84a

Chambers, F. B., D. A. Luce, and G. P. Jones, *Distributed Computing*, Academic Press, London, 1984.

Checkland91a

Checkland, I. and C. Runciman, *Development of a Prototype Geometric Modelling System using a Functional Language*, Technical Report, Department of Computer Science, University of York, York, 1991.

Chevrel81a

Chevrel, M., M. Courtis, and G. Weill, "The SPOT satellite remote sensing mission," *Photogrammetric Engineering and Remote Sensing*, vol. 47, no. 8, pp. 1163 - 1171, 1981.

Church41a

Church, A., *The Calculi of Lambda Conversion*, Princeton University Press, 1941.

Clarke86a

Clarke, K. A., "Computer Tomography," in *Cellular Logic Image Processing*, ed. M. J. B. Duff and T. J. Fountain, pp. 141 - 172, Academic Press, 1986.

Fairbairn87a

Fairbairn, J. and S. Wray, "TIM - a simple lazy abstract machine to execute supercombinators," in *Functional Programming Languages and Computer Architecture*, LNCS 274, Springer Verlag, 1987.

Foley90a

Foley, J. D. and A. van-Dam, and Steven K. Feiner, and John F. Hughes, *Computer Graphics - Principles and Practice*, Addison-Wesley, 1990.

Gargantini82a

Gargantini, I., "An Effective Way to Represent Quadtrees," *Communications of ACM*, vol. 25, no. 12, pp. 905 - 910, ACM, December 1982.

Gonzalez87a

Gonzalez, R. C. and P. Wintz, *Digital Image Processing (Second Edition)*, Addison-Wesley, 1987.

Gordon78a

Gordon, M., R. Milner, L. Morris, M. Newey, and C. Wadsworth, "A metalanguage for interactive proof in LCF," in *Conference Record of the 5th Annual ACM Symposium on Principles of Programming Languages*, pp. 119 - 130, ACM, 1978.

Hall92a

Hall, C., K. Hammond, W. Partain, S. L. Peyton Jones, and P. Wadler, *The GRASP project*, University of Glasgow, Glasgow, February 1992.

Hall92b

Hall, C., K. Hammond, W. Partain, S. L. Peyton Jones, and P. Wadler, *Abstracts of GRIP/ GRASP-related papers and reports 1990 and after*, University of Glasgow, Glasgow, February 1992.

Hall93a

Hall, C., K. Hammond, W. Partain, S. L. Peyton Jones, and P. Wadler, "The Glasgow Haskell Compiler: A Retrospective," in *Functional Programming, Glasgow 1992 (Proceedings of the 1992 Glasgow Workshop on Functional Programming, Ayr, Scotland, 6-8 July 1992*, pp. 62 - 71, Springer-Verlag, 1993.

Hamey89a

Hamey, L. G. C., J. A. Webb, and I. C. Wu, "An Architecture Independent Programming Language for Low-Level Vision," *Computer Vision, Graphics, and Image Processing*, vol. 48, no. 2, pp. 246 - 264, Academic Press, November 1989.

Heckbert86a

Heckbert, P. S., "Survey of Texture Mapping," *IEEE Computer Graphics and Applications*, vol. 6, no. 11, pp. 56 - 67, November 1986.

Henderson80a

Henderson, P., *Functional Programming : Application and Implementation*, Prentice-Hall, London, 1980.

Henderson82a

Henderson, P., "Functional Geometry," in *LISP and Functional Programming Symposium Papers*, pp. 179 - 187, ACM, 1982.

Herman80a

Herman, G. T., *Image Reconstruction from Projections*, Academic Press, New York, 1980.

Hindley69a

Hindley, R., "The Principle Type Scheme of an Object in Combinatory Logic," *Transactions of American Mathematics Society*, vol. 146, pp. 29 - 60, 1969.

Hockney81a

Hockney, R. W. and C. R. Jesshope, *Parallel Computers*, Adam Hilger Ltd., Bristol, 1981.

Hopkins89a

Hopkins, S., G. J. Michaelson, and A. M. Wallace, "Parallel Imperative and Functional Approaches to Visual Scene Labelling," *Image and Vision Computing*, vol. 7, no. 3, pp. 178 - 193, Butterworth, London, August 1989.

Horn86a

Horn, B. K. P., *Robot Vision*, The MIT Press, Cambridge, Massachusetts, 1986.

Howe92a

Howe, D., *mira2hs (in the contribution to hbc distribution)*, Department of Computing, Imperial College, London, 1992.

Hudak88a

Hudak, P. and R. Sundaresh, "On the expressiveness of purely functional I/O systems," *Technical Report*, no. YALEU/DCS/RR-665, Dept. of Computer Science, Yale University, 1988.

Hudak89a

Hudak, P., "Conception, Evolution, and Application of Functional Programming Languages," *ACM Computing Surveys*, vol. 21, no. 3, pp. 359 - 411, ACM Press, September 1989.

Hudak92a

Hudak, P., S. L. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzman, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson, "Report on the Programming Language Haskell - A Non-strict Purely Functional Language (Version 1.2)," *SIGPLAN NOTICES*, vol. 27, no. 5, ACM, May 1992.

Hudak92b

Hudak, P. and J. H. Fasel, "A Gentle Introduction to Haskell," *SIGPLAN NOTICES*, vol. 27, no. 5, ACM, May 1992.

Hughes84a

Hughes, R. J. M., *Parallel Functional Languages Use Less Space*, 1984.

Hughes84b

Hughes, R. J. M., *The Design and Implementation of Programming Languages* , PhD thesis, Oxford University, Oxford, 1984.

Hughes89a

Hughes, J., "Why Functional Programming Matters," *The Computer Journal*, vol. 32, pp. 98 - 107, Cambridge University Press, Cambridge, April 1989.

Hunter79a

Hunter, G. M. and K. Steiglitz, "Operations on Images Using Quad Trees," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 1, no. 2, pp. 145 - 153, IEEE, 1979.

Hunter79b

Hunter, G. M. and K. Steiglitz, "Linear Transformation of Pictures Represented by Quad Trees," *Computer Graphics and Image Processing*, pp. 289 - 296, July 1979.

ISO92a

ISO, *Image Processing and Interchange - Functional Specification Part2: The Programmer's Imaging Kernel System Application Program Interface*, ISO/IEC Committee Draft (CD) 12087-2, ISO/IEC JTC1 SC24, April 1992.

Johnsson87a

Johnsson, T., *Compiling Lazy Functional Languages*, PhD Thesis, Chalmers University of Technology, 1987.

Jones92a

Jones, M. P., *An Introduction to Gofer (draft version)*, 1992.

Kawaguchi80a

Kawaguchi, E. and T. Endo, "On a Method of Binary-Picture Representation and Its Application to Data Compression," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 2, no. 1, pp. 27 - 35, IEEE, 1980.

Kingdon91a

Kingdon, H., D. Lester, and G. L. Burn, "The HDG-Machine: A Highly Distributed Graph Reducer for a Transputer Network," *The Computer Journal*, vol. 34, no. 4, pp. 290 - 302, 1991.

Klinger76a

Klinger, A. and C. R. Dyer, "Experiments in picture representation using regular decomposition," *Computer Graphics and Image Processing*, vol. 5, no. 1, pp. 68 - 105, 1976.

Knuth73a

Knuth, D. E., The Art of Computer Programming, vol. 3, Addison-Wesley, 1973.

Knowlton80a

Knowlton, K., "Progressive transmission of gray-scale and binary pictures by simple, efficient, and lossless encoding schemes," *Proc. IEEE*, vol. 68, pp. 885 - 896, IEEE, 1980.

Kozato88a

Kozato, Y., "Design Considerations for "General-Purpose" Image Processing Systems," *End-of-2nd-year report*, Department of Computer Science, University College London, September 1988.

Kozato92a

Kozato, Y. and G. P. Otto, "Geometric Transformations in a Lazy Functional Language," in *Proceedings of 11th International Conference on Pattern Recognition*, vol. IV, pp. 128 - 132, The Hague, August 1992.

Kozato93a

Kozato, Y. and G. P. Otto, "Benchmarking Real-Life Image Processing Programs in Lazy Functional Languages", in *Proceedings of FPCA '93*, Copenhagen, June 1993.

Lakshminarasimhan89a

Lakshminarasimhan, A. L. and M. Srivas, "A Framework for Functional Specification and Transformation of Hidden Surface Elimination Algorithms," *Computer Graphics Forum*, vol. 8, no. 2, pp. 75 - 98, June 1989.

Landin64a

Landin, P. J., "The Mechanical Evaluation of Expressions," *Computer Journal*, vol. 6, no. 4, pp. 308 - 320, January 1964.

Landin66a

Landin, P. J., "The Next 700 Programming Languages," Communications of the ACM, vol. 9, no. 3, pp. 157 - 166, ACM, 1966.

Lau-Kee91a

Lau-Kee, D., A. Billyard, R. J. Faichney, Y. Kozato, G. P. Otto, M. Smith, and I. Wilkinson, "VPL: An Active, Declarative Visual Programming System," in *Proceedings of IEEE Workshop on Visual Languages*, Kobe, Japan, October 1991.

Le Gall91a

Le Gall, D., "MPEG: A Video Compression Standard for Multimedia Applications," *Communications of the ACM*, vol. 34, no. 4, pp. 46 - 58, April 1991.

McCarthy60a

McCarthy, J., "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I," *Communications of the ACM*, vol. 3, no. 4, pp. 184 - 195, ACM, 1960.

Meagher82a

Meagher, D., "Geometric Modeling Using Octree Encoding," *Computer Graphics and Image Processing*, pp. 129 - 147, June 1982.

Milner84a

Milner, R., "A Proposal for Standard ML," in *1984 ACM Symposium on Lisp and Functional Programming*, pp. 184 - 197, Austin, Texas, August 1984.

Milner78a

Milner, R., "A theory of type polymorphism in programming," *Journal of Computer and System Science*, vol. 17, pp. 348 - 375, 1978.

Otto92a

Otto, G. P., D. Lau-Kee, and Y. Kozato, "Design and implementation issues in VPL: visual language for image processing," in *Proceedings of SPIE/IS&T conference on image processing*, vol. 1659, pp. 240 - 253, SPIE, San Jose, February 1992.

Ouksel92a

Ouksel, M. A. and A. Yaagoub, "The Interpolation-Based Bintree and Encoding of Binary Images," *CVGIP: Graphical Models and Image Processing*, vol. 54, no. 1, pp. 75 - 81, Academic Press, January 1992.

Parsons86a

Parsons, M. S., "Generating Lines Using Quadgraph Patterns," *Computer Graphics Forum*, vol. 5, pp. 33 - 39, North-Holland, 1986.

Parsons87a

Parsons, M. S., *Applicative Languages and Graphical Data Structures*, PhD Thesis, Computing Laboratory, University of Kent, September 1987.

Parsons89a

Parsons, M. S., "Image Representations Using Miranda Laws," *Computer Graphics Forum*, vol. 8, pp. 99 - 106, North- Holland, 1989.

Partain93a

Partain, W., "The nofib Benchmark Suite of Haskell programs (pre-release Ayr draft)," in *Functional Programming, Glasgow 1992 (Proceedings of the 1992 Glasgow Workshop on Functional Programming, Ayr, Scotland, 6-8 July 1992*, pp. 195 - 202, Springer-Verlag, 1993.

Paulson91a

Paulson, L. C., *ML for the Working Programmer*, Cambridge University Press, 1991.

Peyton Jones87a

Peyton Jones, S. L., *The Implementation of Functional Programming Languages*, Prentice-Hall International, 1987.

Peyton Jones92a

Peyton Jones, S. L. and D. R. Lester, *Implementing Functional Languages*, Prentice Hall, 1992.

Peyton Jones92b

Peyton Jones, S. L., *Functional programming at Glasgow University*, Dept of Computing Science, Glasgow University, April 1992.

Piper85a

Piper, J. and D. Rutovitz, "Data structures for image processing in a C language and Unix environment," *Pattern Recognition Letters*, vol. 3, pp. 119-129, North-Holland, March 1985.

Poole92a

Poole, I., "A Functional Programming Environment for Image Analysis," in *Proceedings of 11th International Conference on Pattern Recognition*, vol. IV, pp. 124 - 127, The Hague, 1992.

Poole92b

Poole, I., *Private communication*, April, August, 1992.

Pratt91a

Pratt, W. K., *Digital Image Processing (Second Edition)*, Wiley, 1991.

Reade89a

Reade, C., *Elements of Functional Programming*, Addison-Wesley, 1989.

Research Software Limited89a

Research Software Limited, *Miranda System Manual (V2.014)*, 1989.

Runciman92a

Runciman, C. and D. Wakeling, *Heap Profiling of Lazy Functional Programs*, Technical Report, Department of Computer Science, University of York, York, 1992.

Runciman92b

Runciman, C., "TIP in Haskell — another exercise in functional programming," in *Proceedings of 1991 Glasgow Workshop on Functional Programming*, ed. R. Heldal, C. K. Holst and P. Wadler, Springer-Verlag, 1992.

Runciman93a

Runciman, C. and D. Wakeling, "Heap Profiling of a Lazy Functional Compiler," in *Functional Programming, Glasgow 1992 (Proceedings of the 1992 Glasgow Workshop on Functional Programming, Ayr, Scotland, 6-8 July 1992*, pp. 203 - 214, Springer-Verlag, 1993.

Samet84a

Samet, H., "The quadtree and related hierarchical data structures," *ACM Computing Surveys*, vol. 16, no. 2, pp. 817 - 860, ACM, 1984.

Samet88a

Samet, H. and R. E. Webber, "Hierarchical Data Structures and Algorithms for Computer Graphics Part I: Fundamentals," *IEEE Computer Graphics and Applications*, vol. 8, no. 3, pp. 48 - 68, May 1988.

Samet90a

Samet, H., *The Design and Analysis of Spatial Data Structures*, Addison-Wesley, 1990.

Sansom93a

Sansom, P. and S. L. Peyton Jones, "Profiling Lazy Functional Languages (Working Paper)," in *Functional Programming, Glasgow 1992 (Proceedings of the 1992 Glasgow Workshop on Functional Programming, Ayr, Scotland, 6-8 July 1992*, pp. 227 - 239, Springer-Verlag, 1993.

Sato90a

Sato, H., H. Okazaki, T. Kawai, H. Yamamoto, and H. Tamura, "The VIEW-Station Environment: Tools and Architecture for a Platform-Independent Image Processing Workstation," in *Proceedings of the 10th International Conference on Pattern Recognition*, Atlantic City, NJ, June 1990.

Schalkoff89a

Schalkoff, R. J., *Digital Image Processing And Computer Vision*, Wiley, 1989.

SPEC92a

SPEC, *The January 1992 SPEC/CINT92/CFP92 announcement*, Standard Performance Evaluation Corporation, 1992.

Spivey90a

Spivey, M., "A Functional Theory of Exceptions", *Science of Computer Programming*, vol. 14, pp. 25 - 42, North-Holland, 1990.

Steele Jr.90a

Steele Jr., G. L., *CommonLisp - The Language (Second Edition)*, Digital Press, 1990.

Sterling86a

Sterling, L. and E. Shapiro, *The Art of Prolog*, MIT Press, 1986.

Stoy81a

Stoy, J. E., *Denotational Semantics*, MIT Press, 1981.

Strachey67a

Strachey, C., "Fundamental Concepts in Programming Languages", in *Lecture Notes for International Summer School in Computer Programming*, Copenhagen, August 1967.

Stroustrup86a

Stroustrup, B., *The C++ Programming Language*, Addison-Wesley, 1986.

Stroustrup91a

Stroustrup, B., *The C++ Programming Language (Second Edition)*, Addison-Wesley, 1991.

Suetens92a

Suetens, P., P. Fua, and A. J. Hanson, "Computational Strategies for Object Recognition," *ACM Computing Surveys*, vol. 24, no. 1, pp. 5 - 61, ACM, March 1992.

Sun Microsystems Inc.87a

Sun Microsystems Inc., *SunOS Reference Manual*, 1987.

Tanaka86a

Tanaka, A., M. Kameyama, S. Kazama, and O. Watanabe, "A Rotation Method for Raster Images Using Skew Transformation," in *Proc. IEEE Conference on Computer Vision and Pattern Recognition*, pp. 272 - 277, June 1986.

Taylor91a

Taylor, J., "A System For Representing The Evaluation Of Lazy Functions," *Technical Report*, no. 522, Dept. of Computer Science, Queen Mary and Westfield College, London, February 1991.

Thimbleby87a

Thimbleby, H. W., "The Design of a Terminal independent Package," *Software — Practice and Experience*, vol. 17, no. 5, pp. 351 - 367, May 1987.

Thompson86a

Thompson, S., "Laws in Miranda," in *Proc. of the 1986 ACM Conference on Lisp and Functional Programming*, pp. 1 - 12, ACM, Cambridge, Massachusetts, August 1986.

Traub91a

Traub, K. R., *Implementation of Non-Strict Functional Programming Languages*, The MIT Press, Cambridge, Massachusetts, 1991.

Trinder92a

Trinder, P. and K. Hammond, and D. McNally, "Functional Languages can't Manipulate Persistent Data (DRAFT)," in *Draft Proceedings of Fifth Annual Glasgow Workshop on Functional Programming*, pp. 282 - 290, Ayr, Scotland, July 1992.

Turner76a

Turner, D. A., "SASL Language Manual," *University of St. Andrews Technical Report*, 1976.

Turner79a

Turner, D. A., "A new implementation technique for applicative languages," *Software - Practice and Experience*, vol. 9, pp. 31 - 49, 1979.

Turner82a

Turner, D. A., "Recursion Equations as a Programming Language," in *Functional Programming and its Applications*, ed. J. Darlington, P. Henderson, and D. A. Turner, pp. 1 - 28, Cambridge University Press, 1982.

Turner85a

Turner, D. A., "Miranda: A non-strict functional language with polymorphic types," in *Lecture Notes in Computer Science*, vol. 201, Springer, 1985.

Turner86a

Turner, D. A., "An Overview of Miranda," in *SIGPLAN Notices*, December 1986.

Uhr86a

Uhr, L., K. Preston, Jr., S. Levialdi, and M. J. B. Duff (eds), *Evaluation of Multicomputers for Image Processing*, Academic Press, 1986.

Wadler84a

Wadler, P., "Listlessness is Better than Laziness - Lazy evaluation and garbage collection at compile-time," in *Proceedings of the 1984 ACM Conference on LISP and Functional Programming*, pp. 45 - 52, ACM, August 1984.

Wadler88a

Wadler, P., "Deforestation: Transforming Programs To Eliminate Trees," in *European Symposium on Programming*, Lecture Notes in Computer Science 300, Springer, 1988.

Wadler89a

Wadler, P. and S. Blott, "How to make ad-hoc polymorphism less ad hoc," in *Proceedings of 16th ACM Symposium on Principles of Programming Language*, pp. 60 - 76, ACM Press, January 1989.

Wallace91a

Wallace, G. K., "The JPEG Still Picture Compression Standard," *Communications of the ACM*, vol. 34, no. 4, pp. 30 - 44, ACM, April 1991.

Wallace92a

Wallace, A. M., G. J. Michaelson, P. McAndrew, K.G. Waugh, and W. J. Austin, "Dynamic Control and Prototyping of Parallel Algorithms for Intermediate and High-Level Vision," *IEEE Computer*, vol. 25, no. 2, pp. 43 - 53, IEEE, February 1992.

Woods91a

Woods, J. W. (ed), *Subband Image Coding*, Kluwer academic publishers group, 1991.

Wolberg90a

Wolberg, G., *Digital Image Warping*, IEEE Computer Society Press, Wachington DC, 1990.

The Yale Haskell Group91a

The Yale Haskell Group,, *The Yale Haskell Users Manual*, Yale University, New Haven, May 1991.

Subject Index

Code Index