# Graphical Application and Visualization of Lazy Functional Computation

Sandra Periam Foubister

Submitted for the degree of Doctor of Philosophy

The University of York

Functional Programming Group,
The Department of Computer Science.

May 1995

# Abstract

Mere academic toys or the tools of the future? Lazy functional programming languages have undoubted attractive properties. This thesis explores their potential, from the programmer's point of view, for implementing interactive and graphical applications to which they do not seem immediately suited. The discussion is centred round two example applications.

One is a graphical design program based on an idea of the artist M. C. Escher. The thesis argues that the graphical user interface may be encapsulated in an "interpret" function that when applied by a mouse click to an interface of appropriate type yields the required behaviour.

The second example is a monitoring interpreter for a functional language. The idea is that if the mechanics of the reduction are presented at a suitable level of abstraction, this may be used to give insight into what is going on. On the basis of this the programmer might modify the code so that a program runs more efficiently in terms of speed and memory requirements.

Problems of displaying the reduction are addressed, and solutions proposed for overcoming these: displaying the graph as a spanning tree, to ensure planarity, with extra leaves replacing missing arcs; compacting the display into a quotient graph using equivalence classes for nodes; displaying only part of the graph and allowing the user to browse this; and checkpointing to reduce the number of reduction stages to show. A metalanguage for user definition of such visual filters is developed. This gives the programmer flexibility in attaining a meaningful view of the reduction process.

The conclusions are that, even using current implementations, lazy functional languages are not only capable, but well suited, to writing interactive graphical applications. However the problems inherent in laziness need to be tackled by allowing strictness annotations and by further development of monitoring facilities such as those proposed here.

# Contents

# List of Figures

# Acknowledgements

I am glad of this opportunity to thank some of the many people who have enabled me to complete this thesis: above all, of course, Colin Runciman my supervisor, for showing me the fun of functional programming and making it all possible; the wise polymath Alan Dix who cared for me when Colin was away; past and present members of the York functional programming group, especially Ian Toyn; many other people at York, notably all the support staff; Marc Thomas who helped me win a car; people in Edinburgh who stored the contents of my flat; Thomas Johnsson and Lennart Augustsson whose LML compiler marked the beginning of practical lazy functional programming; people in Glasgow especially Will "always helpful" Partain; Margaret Swain who is making good use of her first word processor in her 87th year and is a dear friend; Holly and Megan who keep me supplied with pictures for the walls; finally colleagues at Heriot-Watt who have encouraged my dual existence, especially the Vision Group whose machines have generously hosted several megabytes of Haskell compiler and window system files for me. The work was funded by a SERC grant.

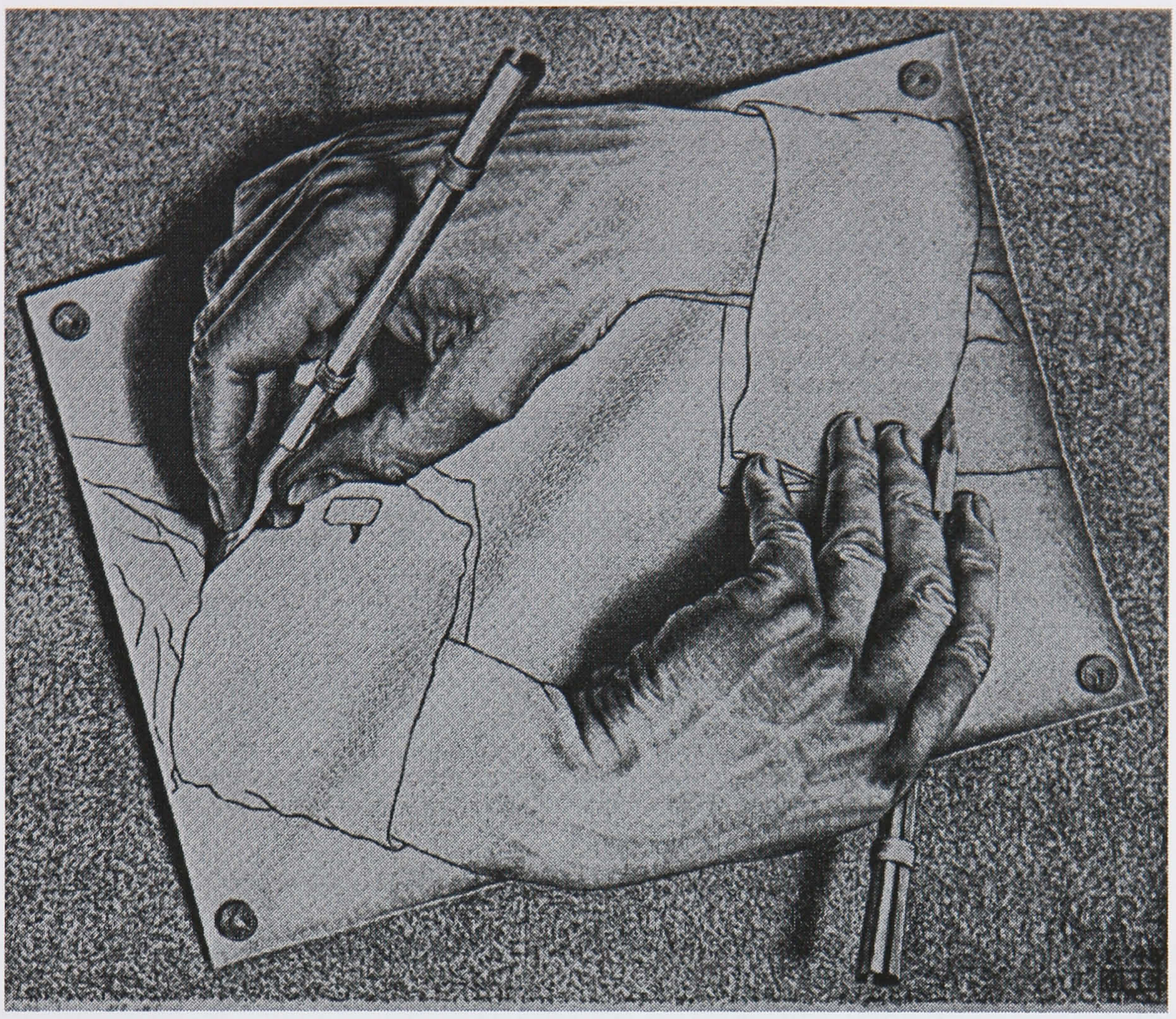
# Declaration

The design program discussed in Chapter 3, and much of the material in that chapter including the idea of a mouse click as a function application, I originally presented in a paper at the 3rd International Conference for Young Computer Scientists held in Beijing in 1991 [36]. A revised version appears as Chapter 4 in the book: Applications of Functional Programming [35].

I discussed the technique of displaying a graph as a tree with shared nodes indicated by display references at Graph Drawing '93 in Paris in a talk entitled: "The Display, Browsing and Filtering of Graph-trees".

The use of spatial and temporal filtering, here described in Chapter 5, and of the meta-language used to define compaction rules, are outlined in "Techniques for Simplifying the Visualization of Graph Reduction" [37], presented at the 1994 Glasgow Functional Programming Workshop in Ayr.

These papers are co-authored by Colin Runciman, my supervisor, and result from collaboration with him.



**Drawing hands by M. C. Escher**

This picture epitomises an outline of the thesis:
on the one hand lazy functional programming creates an interactive graphical program,
based on an idea of M. C. Escher;
on the other hand an interactive graphical program, itself a lazy functional program,
delineates a lazy functional computation.

# Chapter 1

# Introduction

## 1.1 Motivation

Lazy functional programming is like the curate's egg – good in parts. The virtues of the functional approach (see Section 1.1.1 below) are not in question, but the unpredictability of implementations in terms of the performance of programs sometimes outweighs these attractive features.

John Darlington writes [22]:

> "The late 1980s promise to be fascinating years for workers in declarative languages. This coming together of parallel machines, mature declarative languages and transformation based programming environments means that all the, mutually supporting, components are in place for a searching appraisal of the ultimate practicality of this approach."

The "practicality" of the approach will depend on its ability to deal with the space and time problems for which functional programming is infamous. The motivation for this thesis is to take part in that appraisal which is continuing into the 1990s. The focus is on the point of view of the programmer rather than the implementor: using current implementations can we provide evidence that this style of programming is viable? What information does the programmer need in order to write efficient programs?

### 1.1.1 The virtues of functional programming

Functional programming has several attractive properties which make research into its ultimate viability worthwhile:

**Directness** Once the programmer has abstracted the essence of a specification, the directness of the functional style allows it to be precisely reflected in the program text. The code produced is clear and readable, and therefore easy to modify.

**Freedom from side effects** Part of the directness results from freedom from side effects: a programmer may concentrate on a function definition without needing to consider possible consequences for other parts of the program. Functional expressions are referentially transparent, so functional programs are suited to equational reasoning. They may also be transformed manually or automatically to optimise performance.

**Lazy evaluation** The lack of side effects also results in the order of evaluation not being important. Lazy functional programming exploits this. It extends the scope of application for higher order functions, and, for example, allows termination conditions to be separated from loop bodies.

**Potential for parallelism** The flexibility in order of evaluation also gives functional languages apparent potential for parallel implementations. This aspect is a current area of research, and may well be of paramount importance for the the future use of functional programming.

**Higher order functions** The use of higher order functions, together with the possibility of exploitation of polymorphism, facilitates design abstraction, code reuse, conciseness of code, and reliability and ease of programming.

Such features "push back the conceptual limits on the way programs may be modularised" [50].

### 1.1.2 Aim of the thesis

The overall aim of the thesis is to demonstrate that the problems of space and time usage can be understood sufficiently for them to be controlled, so that these uncontroversial benefits of lazy functional languages can be reaped, for example, in the context of interactive graphical applications.

I am concentrating on interactive and graphical applications as these are likely to expose problematic subtleties arising from intricate program structures, and unpredictable evaluation order. For example the order of evaluation in a lazy language cannot be predicted in the absence of implementation details, but an interactive application requires precise sequencing; and the "state" of both program and display in a graphical application needs to be reconciled with the functional style. Such applications also afford possibilities for exploring laziness, for example in the use of "almost circular" definitions [4, 12]. Such applications are also likely to expose any "embarrassing pauses" or space leaks.

There are two complementary objectives, both fitting the heading of "See how they run":

1. to develop suitable programming techniques within a lazy functional programming system for interactive graphical applications, and
2. to develop an interactive functional programming environment (itself a purely functional program) in which program evaluation may be monitored and observed graphically. The aim here is to enable a programmer to write "better" programs, *i.e.* that use fewer resources, through better understanding of what is going on as they run.

Both objectives are explored in the context of particular applications, the first an interactive graphical design program based on an idea by the artist M. C. Escher, the other a minimal programming environment for a functional language. Both implementations are written in Haskell [34].

## 1.2 See how they run I — The Escher program

There are both potential advantages and disadvantages in writing pure declarative interactive graphical programs.

The implementation of a program architecture based on a functional description of the interface may lead to an enhanced clarity of programming; this clarity may be reflected in a declarative user interface. This suggests that the declarative style may be used to express directly, not only an executable prototype, but the implementation itself.

But the abstraction involved in using the declarative style means that the programmer no longer has control over storage management, so implementations of functional languages may make less efficient use of conventional machine resources than other languages [60]. Unless the programmer has access to monitoring facilities, the time and space properties of programs are often unpredictable: the programmer may unwittingly create a program that requires an unexpectedly large, or even increasing, amount of space in which to store shared structures and suspended computations; this may then slow the program down because of time given over to memory management, and the program may crash if the memory requirements become too great.

Another possible problem is that interfacing with an imperative window system could result in a lack of referential transparency. There is already evidence that such problems may be overcome [27], and there is a current spate of active research explicitly aimed at defining a suitable graphical interface (*e.g.* [18, 96]). But at present it remains an open problem.

The purpose of Part I of the thesis is to investigate the practical limits of the pure lazy functional paradigm by implementing an interactive graphical application in Haskell, explicitly reflecting the specification in the program code, and observing the program's run time behaviour. The Escher program discussed in Chapter 3 provides an early example of a simple declarative graphical interface, and it is argued in Section 2.1.2, in Chapter 2, that the apparent problem of different displays resulting from the same input is artificial. The aim here is to build on the work of Andrew Dwelly [27] which suggests that the potential problems can indeed be overcome, and that the expressiveness of a lazy functional language may indeed be exploited in this context. He writes, in connection with his *dialogue combinators* (see page 13):

> "The techniques presented here, allow the construction of modern graphical user interfaces with a lazy functional language. Such interfaces have the advantages of being both compactly and understandably described, as well as being efficiently executable."

The Escher program confirms the expectations engendered by Dwelly's work in a more substantial application. It also develops the concept of the interface as a structure to which an interpreting function may be applied, by means of a mouse click, yielding the required interface behaviour. Dwelly goes on to say:

> "It is interesting to note that one area of computer science that has still to benefit from graphic user interface design, is that of software environments for functional languages ..."

And this leads to the second aspect of the thesis: the development and use of a monitoring interpreter for a quintessential non-strict functional programming language.

## 1.3 See how they run II — The monitoring interpreter

In approaching the space and time problems mentioned above the functional programmer has only recently begun to have access to tools akin to those available to the imperative programmer for analysing program behaviour. In 1989 Augustsson and Johnsson [9] were writing:

> "There is ... a lack of tools for analysing program behaviour; the usual UNIX tools for profiling programs, like "prof", do not work so well in a lazy evaluation context, or with higher order functions. When programming in a style making much use of the predefined higher order functions like `map`, `reduce`, etc. the profiler may well say that most of the time is spent in `map` or `reduce` — hardly a big help when trying to pinpoint the bottlenecks in one's program."

Although the situation is currently being remedied, as discussed in Chapter 4, there is still a need for tools which give the user details of the reduction process in a digestible and meaningful form. Statistics about a computation may be revealing, but it may be that some form of *visualization* of the reduction process is needed to expose the nature of a problem: relevant structural properties of the program being run may not be exposed by a statistical account of the composition of the heap.

The discussion in Part II of the thesis is based around the design, implementation and use of a monitoring interpreter. It is unusual in that it is a graphical functional programming environment written in a purely functional style. This enables further observations to be made regarding the suitability of a lazy functional language for such an application.

### 1.3.1 Rationale for the interpreter

People are unable to predict the behaviour of a lazy functional program because, although the order of reduction is deterministic in a given sequential implementation, it is not intuitively obvious. Even with statistics, or diagrammatic summaries, about the memory usage as provided by cost centre or heap profiling, discussed in Chapter 4, the exact causes cannot be shown, and it may be therefore that the programmer does not gain understanding of what is going on in sufficient detail to be able to control it.

One solution would be to make all details of the reduction open to inspection. Two problems arise: the level at which to do this and, whatever level is used, the overwhelming amount of information that would be provided. There is a need to be able to relate the data to the source code. To portray the reduction in terms of the combinators to which it gets translated is inadequate and potentially confusing.

Simple graph reduction/template instantiation fulfils the needs to relate the observation of the process to the source code while being sufficiently close to reduction using supercombinators to be likely to throw light on the performance resulting from a real implementation. This is discussed further in Chapter 5. Having chosen this level of presentation we are left with the other problem — of too much to show. The program graph could be displayed in its entirety on the screen – but even using labeling with source names the overall view is complex even in simple examples. So the problem is to get a handle on the graph so that it may be understood. One of the sources of complexity is the crossing of arcs in a display, another is its potential size. There are various possible solutions to these such as only showing part of the graph and (somehow) ensuring as much planarity as possible – one that completely solves the arc crossing problem, but at the expense of potentially making the size problem worse, is to use a *graph tree*, a spanning tree of the graph with missing arcs displayed as extra leaves (see Chapter 5).

In order to compact such a structure, or, indeed, the original graph, without losing the meaning and structure of the graph, the proposed solution is to display a quotient graph where each vertex is a subgraph of the orginal graph. The partitioning of the graph is according to equivalence rules which state whether or not any adjacent pair of graph nodes belong to the same subgraph, *i.e.* whether the arc between them should be collapsed. In order that the viewer may control the display the equivalence rules need to be flexibly definable by the user on the basis of accessible primitive conditions on the relevant nodes.

Similarly, as the reduction proceeds, the viewer needs to focus on specific sections of computation: this time it is conditions on complete graphs that need to be used to determine which sections of the reductions may, at least temporarily, be skipped over.

A metalanguage is devised to enable the user to define his/her own filters over a display and/or over a sequence of reduction steps, and a highly interactive interface proposed so that such filters may be flexibly applied to create useful views of the computation.

## 1.4 Outline of thesis

**Chapter 2** considers the problems of sequencing and referential transparency in relation to interactive graphical programs. It goes on to review the principal approaches to writing interactive functional programs. There is then a review of evidence that the functional style is particularly appropriate to manipulating graphics, provided by existing examples of interactive graphical lazy functional programs. The possibility of a convenient declarative definition of the graphical user interface is explored. Techniques for interfacing between a functional program and a window system are outlined. Finally the "Escher program" to be discussed in Chapter 3 is introduced.

**Chapter 3** describes the implementation of an interactive graphical program in a lazy functional language. It investigates:

1. advantages and disadvantages of using a lazy functional programming language for such an application;
2. whether the performance of the program is satisfactory — *i.e.* the first aspect of "See how they run";
3. a declarative implementation of the user interface, including:
   - the representation of a mouse click as a function application;
   - the incorporation of principles of user interface design;
   - the viability of a generic functional model of interaction.

There is first an account of the application from the user's point of view; then the implementation is discussed, ending with an account of the interface; the program is reviewed according to each of the points above; finally a "Future work" section proposes possible extensions to the program, and work deriving from its implementation.

**Chapter 4** reviews monitoring and profiling tools for functional languages. Existing systems are discussed under the headings:

- Routine collection of statistics
- Side effecting tracing
- Debugging without side effects
- Purpose built environments
- Profiling graph reduction

The chapter closes with a discussion in which the requirements for the proposed monitoring interpreter are established.

**Chapter 5** discusses the design of a programming environment to incorporate the monitoring interpreter — the second aspect of "See how they run". The nature of the language to be interpreted is described and justified. An account is given of the reduction process. Problems involved in displaying graph reduction are identified, and solutions involving filters are proposed. A metalanguage is described for defining functions to compact the display, and to determine which reduction steps to show. Finally an overview of the prototype system is given.

**Chapter 6** presents the implementation of the programming environment. The reduction needs to proceed through identifiable steps, and to permit the gathering of information both at a global level, such as the *number* of the current step, and at the level of individual program nodes, such as the name of the function the application of which created them. The display needs to incorporate the elements proposed in Chapter 5, such as the presentation of the graph as a browsable tree, and the compaction of the display according to user defined rules.

There is first an account of the implementation of the reduction. Then a technique for transforming a program graph into a structure that may be displayed without crossing of arcs is delineated. The implementation of the checkpointing and of the compaction of the display is described. The final section discusses the appearance and functionality of the user interface.

**Chapter 7** illustrates the potential of the system by showing examples of its use. There are specimen screen dumps to show how the system may be used for teaching and for locating errors. Then there is a demonstration of the effect of browsing, and of how a spatial filter may be tailored to the compaction of a particular display. This is followed by an account of the problems of labeling a compacted graph. Finally there is discussion of the limitations inherent in the approach taken.

**Chapter 8** concludes by tying together the various strands of the thesis, assessing what has been achieved, and proposing future work.

# Chapter 2

# Graphics and interaction

## 2.1 Introduction

Functional programming is beginning to yield programs that run at a viable speed, suggesting that this concise and clear way of writing programs may be exploited in interactive graphical applications. Interactive functional programs were being written in SASL as early as 1979 [94]; and the seminal work on functional graphics, Henderson's Functional Geometry [45] was published in 1982. But until implementations supported acceptably fast processing of functional programs, perhaps with the advent of Lazy ML [9], and Ponder [103], there was no incentive to write functional programs that were both interactive and graphical.

Moreover, even with the possibility of programs running at an acceptable speed, there remains the problem of referential transparency. We take it as axiomatic that referential transparency is required, so that the concomitant benefits of functional programming[1] may be exploited. However, as we are working with *non-strict* languages, in which the order of evaluation may not be directly inferred from the program text, there are potential problems with the *sequencing* needed in an interactive program. Referential transparency might also appear to have been violated when the *same input* to a graphical program may result in *different displays*, depending on the state of the window system.

### Outline of chapter

This chapter considers the problems of sequencing and referential transparency. It goes on to review the principal approaches to writing interactive functional programs. There is then a review of evidence that the functional style is particularly appropriate to manipulating graph-

[1] expressiveness, ease of transformation and potential for parallelism

ics, provided by existing examples of interactive graphical lazy functional programs. The possibility of a convenient declarative definition of the graphical user interface is explored. Techniques for interfacing between a functional program and a window system are outlined. Finally the "Escher program" to be discussed in the next chapter is introduced.

### 2.1.1 Sequencing

In an interactive program the order of output events, and the timing of output events with respect to the program input, has to be predictable, given a particular input. The programmer has to ensure that, whatever order of reduction is chosen by the implementation, the program will progress as required at run time. For example Wray [103] points out that a prompt should be output before the evaluation of any expression referring to the input.

To obviate the problem of sequencing, the programmer has either to craft the program very carefully, or to make use of programming schemes that pre-package the sequencing, for example: continuations [49], transaction combinators [86, 26], dialogues [63], and the monadic style [70]. These are described below.

To some extent the techniques employed to control sequencing will depend on features of the language used. For example David Turner's languages from SASL [94] to Miranda [93] have included user input as a primitive lazy list. Such languages can, therefore, use all the techniques available for manipulating lazy lists.

### 2.1.2 Referential transparency

The apparent problem of different displays resulting from the same input is artificial. The representation of the result of evaluating an expression is not part of the result, whether directly displayed on the screen, *via* the operating system, or indirectly *via* a window manager. However, the result of an expression may, in its representation, change the display environment which is an aspect of the state of the window manager. For example, in a monochrome graphical context it may change the drawing mode from black on white to inverse video. Changing the graphical display is updating it, so a program that does this appears to be manipulating an external variable. It may also affect the representation of future results. Yet there is no violation of referential transparency. The possibility of the *representation* of the result being a change in the environment (that may affect the representation of future results) is not of direct concern to the program that is producing these results. The intermediate results of the program can be regarded as side-effecting actions, which, themselves, are precisely determined. Recent work in Glasgow [70] by Phil Wadler and Simon Peyton Jones has captured this within the Haskell type system: I/O procedures become part of the intermediate values that are computed. This *monadic* style is described in Section 2.2.3.

Referential transparency is also at stake in the case of programs that interact in other ways with the outside world. For example, functions that take a filename as argument should return the same result, given the same file name. Yet, over time, the "contents" of the file with that name may change. Various solutions have been proposed to this — for example the program may only be allowed to read a file once, then to keep whatever the file holds as the referent of that filename for the whole of the computation no matter what happens to the "real" file meantime. Another solution is to use the monadic scheme mentioned above.

## 2.2 Interaction

The functional approach to programming has developed from a theoretical background which has threads of mathematics, lambda calculus and denotational semantics. This theorising was not geared towards the writing of useful programs, and, in particular, the pragmatics of writing *interactive* programs was not of immediate concern to the early pioneers. Even now some strict functional languages, those that do not apply a function until all its arguments are fully evaluated, regard I/O as being beyond the domain of the pure functional language. For example in SML [59] there is an `input` "command": `input(std_in,10)` refers to the next 10 characters typed in at the keyboard. This treatment of I/O has its own problems of suitable packaging to ensure correct sequencing. There is, however, validity in the view that interactive functional programs have two elements — one is pure; the other, concerned with I/O, is side effecting. This contrasts with the view of an interactive functional program as having a potentially infinite stream of input which is processed into a potentially infinite stream of output. In a strict system such an input list would be treated like any other, so a list-processing function would not be able to provide the basis for an interactive application as all the input would need to be present before the program could be executed.

This section presents the solution that was found for this, the use of *continuations*, then an alternative control system, *transaction combinators*, that can be used in non-strict languages. It goes on to outline various systems that have been proposed for dealing with I/O more generally in functional languages, and concludes with a look at a recent development, the use of *state monads*, which appears to offer a neat answer to the problem.

### 2.2.1 Continuations

The first interactive functional programs were written with the use of *continuations*. Initially this was in the broadest sense using so-called *Landin streams* (see below), then, beginning with HOPE, the technique was used with lazy lists.

#### Landin's streams

Landin [57] proposes a solution to the problem of a language not being able to handle a potentially infinite list directly. He introduces a special function that he calls a *stream*. In the kind of strict language that he is discussing, a function is applied to a list of arguments. A stream is a nullary[2] function: applied to an empty list of arguments it returns a pair of which the first component is the head of the stream, and the second component is another stream, representing the tail. Burge [15] (p 136) notes that such streams are "...most useful for implementing functions which process character streams from input".

In order to structure an interactive program with such a representation of the input, *continuations* may be used. A continuation style version of a function takes an extra, functional, parameter called "the continuation". The result of the normal application of the original function is given to this continuation function as an argument, so that the continuation represents "the rest of the program". The use of continuation functions is not peculiar to interactive programs.

The continuation style of interaction was proposed for HOPE [17], a strict language, but with *one* lazy feature, a lazy `cons`. In this proposal, a function `input` takes an argument of type `device`, and returns a lazy list, where items are read from the device when needed. Similarly a function `output` evaluates the elements of a list and directs output to an indicated device.

#### Lazy lists

The use of lazy lists allows other control structures in addition to continuations. Lazy lists have been used to represent input to functional programs since SASL [94]. Confusingly, these lazy lists are also referred to as *streams*, though in the context of modern lazy functional programming languages there is little danger of ambiguity in the use of the term.

The "stream style" of interaction refers to a program mapping a lazy stream of input to a lazy stream of output. Hudak and Sundaresh [49] demonstrate that this is equivalent in

[2]Landin calls it *none-adic*

expressiveness to the continuation style.

### 2.2.2 Transaction combinators

An alternative to the use of continuations, which exploits the laziness of streams, yet allows them to be used in a controlled way, was proposed in 1986 by Simon Thompson [86]. A similar scheme was put forward by Andrew Dwelly in 1988 [26]. This is the *transaction combinator* style. Pieter Koopman's editor [56] uses specialised transaction combinators in its implementation. The idea was first mooted by John O'Donnell [63], whose `dialogue` function is a combinator that he defines in order to describe and implement components of an applicative programming environment.

#### Thompson combinators

Thompson, using Miranda notation, defines a function type: `interact`, which epitomises an individual interaction:

```
interact * ** = (input, *) → (input, **, output)
```

The type is parametrised on the program states before and after the interaction. A function of type `interact` takes as argument some input and a state, and returns the unused input, a new state, possibly of different type to the original one, and some output.

He goes on to propose combining forms, *combinators*, for such interactions. These are examples of control structures that help build composite interactions. They also have the benefit of making implicit the recursion required by the interactive program.

Transaction combinators are often of type `interact * *`, where the type of the program state remains constant to allow cyclic interaction. However, where the exact number of transactions is explicit in the combinator, the type of the state may change. For example Figure 2.1 shows how the combinator `seq` combines two interactions performed one after the other. This also, incidentally, illustrates a benefit of lazy evaluation: the function `make_output`, which pushes a string on to the output stream, allows the output of `out1` before the invocation of `inter2`.

Thompson defines a whole library of transaction combinators and associated functions, including combinators for iteration, selection between interactions, and sequencing.

```
seq :: interact * ** -> interact ** *** -> interact * ***
seq inter1 inter2 x
    = make_output out (inter2 (rest,st))
      where (rest, st, out) = inter1 x

make_output :: output -> (input, *, output) -> (input, *, output)
make_output piece (in, st, out) = (in, st, piece++out)
```

Figure 2.1: Sequencing transaction combinators.

### Dwelly combinators

Dwelly [26] proposes a similar set of combinators. There are two minor differences. One is that the type of the program state is assumed to be constant: he parametrises his `Dialogue` type, which is otherwise equivalent to Thompson's `interact` type, on only *one* state type. The second difference is that the state and input are regarded as separate arguments, rather than as a pair. A further option would be to regard the input as part of the program state, in which case an interaction function would return a new state and some output, without explicit reference to the rest of the input.

As Dwelly applies transaction combinators to the manipulation of the graphical user interface, his work is particularly relevant here and is further discussed later in this chapter (Section 2.3.4) [27].

### Koopman combinators

Koopman [56] uses specialised transaction combinators, with arguments specific to his application, in his functional definition of an editor. For example Figure 2.2 shows the function that he calls `commandinterpreter` that selects the combinator to apply next, represented by `editoperation`, as well as controlling the overall interaction.

```
commandinterpreter text commands
  = response: prompt : commandinterpreter newtext nextcommands
    WHERE
    commandline: rest                   = commands
    editoperation                       = parse commandline
    response: newtext: nextcommands = editoperation text rest
```

Figure 2.2: Koopman's `commandinterpreter`.

This is an early demonstration of the suitability of functional programming languages for elegantly implementing interactive programs. He notes, for example, that his program is an order of magnitude smaller than than a comparable program in an imperative language, that it was quickly written, and easily extended. He points out that such a program could be incorporated into an integrated functional programming environment, which is indeed something that O'Donnell [63] was doing at round about the same time.

**O'Donnell combinators**

O'Donnell's dialogues [63] predate, yet in some ways extend, the Thompson/Dwelly model. A dialogue is an abstraction of the interaction between two processes. It can be used to describe, not only a human using a computer, but also two communicating processes. It is an interactive session between two participants, each of which has a state that contains information about the history of the interaction. Each also has a transition function: `stp_fcn`, that defines its actions.

This `stp_fcn` is similar to a Thompson/Dwelly combinator, but the indication that the dialogue is to end is determined in the transition function, rather than in the overall controlling function. It returns the stream of unused inputs, a list of outputs to be sent to the other participant, a new state, and a Boolean value to indicate whether that participant wishes to terminate the dialogue. The `dialogue` function repeatedly applies `stp_fcn` to the current values of `inputs` and `state` in order to find the new `inputs'` and `state'`. The inputs that the `stp_fcn` did not consume are used in the next step of the dialogue unless the dialogue terminates.

One of the participants begins the dialogue by starting the other. From then on each computes a new state and a new output from its previous state and the last input it received. Such functions can be used to implement a programming environment, which the user can extend by creating new components.

### 2.2.3 Strategies for marrying I/O with referential transparency

The discussion so far has concentrated on the concerns of style of interaction and control of sequencing. There are other questions that need to be addressed. There is a need to ensure that I/O is implemented in such a way that the functional program is referentially transparent, and that facilities are offered for all flavours of I/O that a program might require — not just user interaction, but communication with all sorts of devices and processes. Even Haskell [34], the Esperanto of functional languages, does not fully come to grips with the problem (see below).

This section presents various strategies for coping with the conflicting demands of "pure" functional I/O and the messy real world of asynchronicity, non-deterministic merging and parallelism:

- Henderson's use of tags, and an `interleave` function;
- Stoye's message passing;
- HOPE+C's result continuations;
- Concurrent Clean's event I/O;
- Haskell's approach to I/O;
- the monadic approach.

### Henderson's operating system

Henderson [46] defines a multi-user operating system in 250 lines of functional code that has a database application and an editor. It also has a facility to run programs. The text of these programs is put into the database by means of the editor. He introduces *tagging* to allow separate users to see on their monitor only the responses associated with their particular requests. Additional tagging could also be used to allow the user to access different databases, with the user explicitly tagging requests at the keyboard[3].

Henderson implements an `interleave` "function" that behaves in a demand driven way: "because of demand for its result, it constantly demands its arguments". In order to implement `interleave` as a *real* function, he considers time-stamping items to enable `interleave` to choose between its arguments. This is effectively adding a "fair merge", an idea that was later explored by Abramsky and Sykes [1].

### Stoye's message passing

William Stoye [83] proposes a system which also uses a non-deterministic *merge* operator. As the non-determinism is only used at the "bottom level" of a program, he regards this as an improvement on Henderson's proposed functional operating system, which is not referentially transparent.

Stoye doesn't attempt to make his `merge` a function. Part of the run-time system, referred to as "the sorting office", does the merging of the output streams from active processes. It sorts them and merges them into input streams according to their tags. He con-

[3] Henderson's use of the terms *response* and *request* is from the point of view of the user, rather than the program. This kind of usage may be the basis of the confusion that the Haskell `Response` and `Request` types can cause, as these are from the point of view of the Haskell program.

siders that such isolation of non-determinism from the functional processes is a convenient way of maintaining their referential transparency.

### Result continuations

Nigel Perry [65, 66] champions another technique for maintaining the separation of the pure functional aspect of a program from the side-effecting parts. The technique uses so-called *result continuations*. These are implemented in HOPE+C, a research language specially designed to demonstrate the result continuation system.

Under this scheme, a `program` is a function of type: $\alpha \rightarrow$ `Result` where $\alpha$ is the type of the initial state, and `Result` is a pair of an *operation request* and a (continuation) function of type $\beta \rightarrow$ `Result`, where $\beta$ is the type of the value returned by the operation request.

This scheme is attractive, in that HOPE+C allows isolation of the parts of the program that are referentially transparent. But it forces the continuation style which may have an unattractive imperative feel, and HOPE+C does not capture the spirit of declarative I/O. What is needed is a language, or a method of writing interactive programs, in which the programmer could write without needing to worry about the problem of referential transparency, knowing that the system being used would guarantee this.

### Histories and event I/O

An alternative to maintaining a separation between the functional program and the environment is to pass the environment around within the functional program.

Backus' FL [10] has an *implicit* history parameter as additional argument to every function, and as part of every result, though it is unchanged except for occasions where I/O takes place. The history component models the state of I/O devices and the file system.

Another more recent proposal comes from the University of Nijmegen [2], regarding the language Concurrent Clean. Several mechanisms are involved in their treatment of I/O. Firstly there is *explicit environment passing where needed*: rather than passing the environment to *all* functions, or to *none*, it is passed only to functions with side-effects. Secondly, single threaded environments can be created by the use of an extension to the type system of a *unique* type predicate: UNQ. Type rules and type definitions can contain UNQ predicates. Figure 2.3 shows the definition of a *unique* file type using the UNQ notation.

This defines a type `UFILE` which is equivalent to `FILE`, but instances of its type will be used linearly. Thus the UNQ type predicate can be used to force programs to use objects in a sin-

```
TYPE
:: UFILE -> UNQ FILE
```

Figure 2.3: Example of the UNQ annotation.

gle threaded way, and offers possibilities for generating efficient code, for example in the implementation of arrays. However, they point out that a functional model for I/O should be multi-threaded, and should specify the least possible amount of reduction order; and neither file nor stream based models are well suited for describing such behaviour. Concurrent Clean, therefore, uses *event* I/O, which is an explicit environment passing method.

The environment is modeled as an `IOsystem` of `IOstates`, each of which is a UNQ abstract object. Each `IOstate` is associated with a `Device`, an object that encapsulates a single thread of I/O. The program can only perform I/O through an `IOstate`. In order that the `Devices` may cooperate, each `Device` function operates, not only on its current `IOstate`, but also on a `Programstate`. As the interaction proceeds, input events to the program are in turn dispatched to the appropriate device, like the procedure in Stoye's sorting office.

### Standard Haskell's I/O system

Haskell's I/O system regards a program as communicating with the outside world *via* synchronised streams (lazy lists) of messages. A program issues a stream of requests to the operating system, for example: `WriteFile String String` or `ReadFile String`. These are of type `Request`. In reply the program receives a stream of responses of type `Response`, for example: `Success` or `Str String`.

A Haskell program has the type:

`Dialogue :: [Response] → [Request].`

Both textual and binary forms of `Request` and `Response` are provided for.

As a continuation based version of I/O may be defined in terms of a stream based one, such as Haskell's, a consistent set of primitive transactions for continuation based I/O is also provided. For example, corresponding to the file system `Request`:

`AppendFile String String`

there is a continuation transaction using which the programmer may express directly "what to do with" the associated `Response`:

`appendFile :: String → String → FailCont → SuccCont → Dialogue`

The type `SuccCont` is a synonym for `Dialogue`, and `FailCont` a synonym for `IOError` → `Dialogue`.

This is adequate for simple I/O, but does not cater for non-determinism, asynchronicity, nor parallelism. There is surely a case for explicit acknowledgement of *time* as an independent parameter — in addition, that is, to the relative time implied by sequencing. The LML `hiaton` is available to the Chalmers' Haskell B. compiler, and goes some way towards alleviating the problem, but is not a standard component of Haskell. In the case of the Glasgow compiler, the `ccall` used to implement monadic I/O (see below) is made available to the programmer, but the need to use such a non-functional extension appears to expose a limitation on the current language definition.

### The monadic approach

Wadler [100] proposes the use of monads, a concept taken from category theory, as a convenient structuring mechanism for certain kinds of programs written in a functional language — particularly those that require a program "state" to be passed round throughout the program. The use of state monads not only enables single-threading of the state to be guaranteed, but also allows the *type* of the state to be changed with minimal alteration to the text of the program.

The Glasgow Haskell compiler makes heavy use of the monadic style in its implementation. Of particular relevance here is the use of monads in conjunction with a non-functional `ccall` to permit referentially transparent interactive programs to be written in a quasi imperative style [70]. This is similar to the use of result continuations in HOPE+C, described above. The `ccall` is a non-standard extension to Haskell. It can call any "function" written in `C`. Used indirectly, and safely packaged in a monadic type, the `ccall` enables referentially transparent `ccalls` to be made, but it is also made *directly* available to the programmer so is a potential source of unsoundness as well as power.

In [70] the `IO a` type is presented as a way of reconciling *being* with *doing*. The type `IO a` represents actions which, when performed, may do some I/O and then return a value of type `a`. For example:

```
getcIO :: IO Char
putcIO :: Char → IO ()
```

`getcIO` is an action which reads in a character from the standard input and returns that character; and `putcIO a` is an action which writes the character `a` to standard output (and returns nothing of interest, hence the `()`).

Such primitive `IO` operations may be combined to provide the basis for interactive programs. For example:

```
bindIO ::  IO a → (a → IO b) → IO b
```

> "If `m :: IO a` and `k :: a → IO b` then ``m `bindIO` k`` behaves as follows: first perform action `m`, yielding a value `x` of type `a`, then perform action `k x`, yielding a value `y` of type `b`, and then return value `y`."

The Glasgow Haskell I/O system, apart from the `ccall` itself, is implemented in Haskell. The type `IO a` is defined as a function which takes the state of the world as argument, and returns the new state of the world and a value of type `a`. As the `IO` type is implemented as a monad, the world state is used in a single threaded way, and I/O operations are applied to the real world immediately they are computed. The "world" value manipulated by the program is a dummy, as the real world is updated in place, but it is kept as a token to ensure the correct sequencing of the interaction. The type can then be regarded as being that described above. An I/O monad has also been incorporated into the Yale Haskell system.

## 2.3 Graphics

The previous section shows how the potential problems for interactive functional programs, involving sequencing and referential transparency, may be overcome. This section reviews pioneering work on functional programming and *graphics* that demonstrates that the functional style is more than suitable for programs that incorporate the manipulation of graphics. Conceptually a function may take a picture as argument and return a picture as result. For example a function could be defined to invert a picture along the horizontal axis (see Figure 2.4).

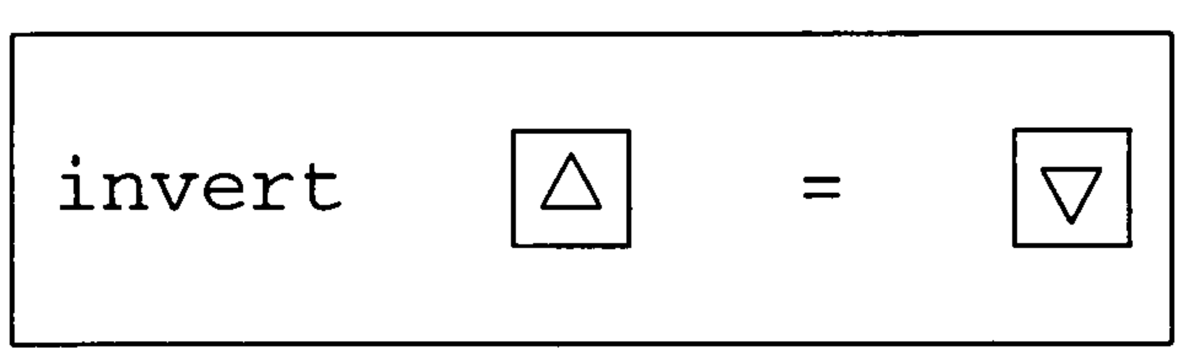



Figure 2.4: A function from `picture` to `picture`.

The first work on functional graphics concentrated on the *representation* of a picture such that a function applied to it may return another, modified, picture. Pictures can then be combined in various ways to create other pictures. Four papers that embody this idea, and apply it in novel ways, are outlined next. They are Henderson's "Functional Geometry" [45], for its seminal status, Arya's "Processes in a Functional Animation System" [8], which describes the creation of *functional movies*, and two early accounts of interactive graphical applications: Wray's spreadsheet [103], and Dwelly's graphical application of transaction combinators [27].

### 2.3.1 Functional Geometry

This is the classic work on graphics and functional programming — all subsequent work in the area refers to it, yet the article itself only references a book about the artist Maurits Escher [28].

Henderson introduces a method of describing pictures. He then uses this to simulate the structure of one of Escher's woodcuts: *Square Limit*. The particular functions that Henderson defines for creating pictures from other pictures are, accordingly, strongly geared towards his Escher example.

#### Pictures

In Henderson's scheme, a `picture` is a set of line segments *defined with reference to a grid.* A function, `grid`, is used to build pictures (Figure 2.5).

```
grid : integer X integer X List (linesegment) -> picture
```

Figure 2.5: The type of Henderson's picture building function.

A line segment is represented by the four integers that make up the coordinates of its two end points. A picture need not be as high nor as wide as the grid, but the size of the grid will affect the display of the picture in relation to a *bounding box* which provides its *display area.* For example a picture defined in a bounding box 10 units high, with a maximum `y` coordinate of 7, will always have a maximum `y` coordinate that is $\frac{7}{10}$ the height of any rectangular bounding box in relation to which it is displayed.

The bounding box is defined by three vectors which describe the position of the lower left corner of the box, in relation to the origin in question, and the length and orientation of its sides. The bounding box may be a rectangle or other parallelogram. In order for a picture to be displayed, its grid is fitted into the bounding box and its line segments drawn to and from the appropriate coordinates.

### Building pictures from pictures

Pictures may be built from other pictures. For example the function `flip` reflects a picture on a vertical axis exactly bisecting the picture's grid, and the function `beside` puts two pictures next to each other such that `beside (m,n,p,q)` is the picture obtained by juxtaposing `p` to the left of `q` with rescaling along the `x` axis resulting in the ratio of their widths being `m` to `n`. The types of these functions are given in Figure 2.6.

```
flip : picture -> picture

beside : integer X integer X picture X picture -> picture
```

Figure 2.6: Types of `flip` and `beside`.

Similarly, `above (m,n,p,q)` is the picture obtained by juxtaposing `p` *above* `q` with rescaling on the `y` axis resulting in the ratio of their heights being `m` to `n`.

Using `nil` as the picture with no line segments in it, `above` and `beside` can be used to define pictures that are "distortions" of the original.

Another function, `rot`, performs 90 degree anticlockwise rotation of the picture. The bounding box, however, does not rotate, so the rotated picture will not have the same shape as the original unless the bounding box is a square.

### Escher's Square Limit

Finally Henderson presents functions that he uses to create a convincing diagram resembling Escher's *Square Limit* from four elements similar to those on which the actual print is based. As he uses *square* bounding boxes, the elements are not themselves distorted, but the juxtaposition of fullsize squares with smaller ones results in an overall, controlled, distortion.

## 2.3.2 Functional Movies

In his Ph.D thesis [7] and subsequent FPCA article [8] Kavi Arya describes a functional programming system for producing graphical animations. He uses Miranda as his functional programming language and SunView[4] as his window system. The motivation was the problem of rapidly prototyping animation sequences, in particular where this involves interac-

[4] Sun Visual/Integrated Environment for Workstations

tion between components of the animated sequence. He uses a functional language, so the program is more easily changed than if he had been using an imperative language, and a prototype can be relatively quickly developed.

The work is carried out using a 2D key frame animation system, where successive frames are written to a buffer and flipped at the appropriate time onto the screen. The programmer is effectively defining a sequence of these frames. Such a picture sequence is referred to as a *movie*.

### Creating movies

A movie consists of a sequence of pictures each of which is a set of polygons, closed to enable the modelling of opacity, and each polygon is described as a set of vertices, as shown in Figure 2.7. A cycle of key frames capturing the key elements of an action, such as a man walking on the spot, indefinitely repeated, is called a *character*.

```
MOVIE = [PIC]
PIC   = [[VEC]]
VEC   = N X N
```

Figure 2.7: Arya's representation of a picture.

A group of functions is defined that combine two movies in different ways, for example: `overlay :: MOVIE → MOVIE → MOVIE` takes two movies and returns a result in which the corresponding frames are overlaid. Other functions are used for cueing. These exploit the time ordering implicit in the sequence of pictures.

Beguiled, no doubt, by Escher, Arya also describes functions to convert from one picture to another in a given number of steps, and even to convert from one *movie* to another. For example what starts out as a walking man may turn into a flying bird. This is a version of "in-betweening" [16], a way of formally describing movement as the transition, in a number of steps, from one defined picture to another, as used extensively in cartoons.

So far the movies defined are but the building blocks for more complex varieties of animation. A character in a movie is, for example, likely to "move" across the screen, rather than staying at one spot, and may change in size or orientation. Accordingly, Arya defines a type: `BEHAVIOUR = [PIC → PIC]` which is a sequence of changes undergone by a character in a movie. Behaviours themselves may be combined by parallel or sequential composition for which Arya supplies infix operators inspired by CSP [48].

### Processes

Arya introduces the idea of *functional processes* as a formalisation within which the elements of a display may communicate with one another. The execution mechanism of processes is called `trace`, again inspired by CSP.

The input stream to a process is a series of messages, and each message is a sequence of pairs of the form (channel,value). The channel consists of an identifying string. Each value is a picture, or a number, or a vector, or a component of a behaviour. Generally a trace associates a single message with each frame of animation.

When a process is "listening" on a channel, it checks at each frame the elements of the message that contain that channel. Values associated with that channel may trigger appropriate continuations, for example in a movie consisting of a man and a vending machine as its two communicating processes, when the man reaches the vending machine he will turn round and walk away from it. The communication is dealt with using *actors*, a notion originally due to Hewitt [47].

Arya's work is interesting in that it brings together diverse areas of functional programming and makes effective use of them in an application that would not seem at first sight to be very amenable to this style of programming.

The actual pictures produced are disappointingly unrealistic, but he does emphasise that his focus is on the processes involved and their suitability for rapidly prototyping animation sequences. He tries to free the animator from machine-oriented patterns of thinking, thereby facilitating his creativity, so the skeletal nature of the examples is really unimportant.

## 2.3.3 Wray's spreadsheet

Wray [103] devised an interactive graphical system as a basis for discussion in his Ph.D. thesis. It is called ANS — A Novel Spreadsheet — and is unusual in that the cells of the spreadsheet can be positioned anywhere on the screen, rather than being in conventional columns and rows. The program is a function that: "takes a list of bytes from the keyboard/mouse, and sends a list of bytes to the screen".

While developing his system, Wray independently invented transaction combinators, that he refers to as complex and recursive *stream processing* functions. Another programming technique that he finds to be of use is "almost circular programming", the technique of "using the answer before it is all there". Aspects of this technique were first presented by Bird [12].

Wray's example of circular programming is in the central loop of his spreadsheet:

```
Letrec new_state = transition_function old_state new_state
```

Wray hit problems of unexpected ordering, in particular: when moving a cell to a new location, using mouse clicks, the cell would disappear as soon as it was selected rather than waiting until its new position was chosen. Fairbairn [32] points out that this results from answers being computed as soon as possible in a language with normal order semantics. The programmer has explicitly to ensure that the "remove ...redraw" sequence does not start until the destination is received.

Wray also discusses the other problem that dogged early interactive applications, that of space leaks: "Uncertainty about the time and space behaviour of functional programs is the worst blow to their credibility where guaranteed performance is needed." Recent techniques that serve to reduce such uncertainty are included in Chapter 4, in the review of monitoring.

### 2.3.4 Dwelly's Rubik cube

The LML distribution [9] includes example interactive graphics programs by Andrew Dwelly. He uses LML's TONEWS primitive, that directs string valued program output to the NeWS [38] window system. Particularly impressive is a multicoloured, mouse-click manipulable, Rubik's cube. This is convincing evidence that lazy functional programming is suitable for real interactive graphical applications.

Dwelly's definition of transaction combinators is given in Section 2.2.2. His bias towards graphical applications led him to a particular combinator which controls the behaviour of a *dynamic* graphical interface. The relevant code is presented in his FPCA paper [27].

He points out that his `AllCase` combinator mimics the event-response user interface described by Green [40] as the model with the greatest descriptive power of three models presented.

The `TreeCase` combinator allows the definition of a *dynamic* user interface, capable of modifying the list of rules that it uses. Finally Dwelly uses the `TreeCase` combinator to define a hypercard program that is both dynamic and multithreaded.

## 2.4 A declarative interface?

Even these early example applications show the benefits of using a lazy functional language in an interactive graphical context. Such a language is good at expressing the manipulation of graphical structures, and the potential problems of structuring referentially transparent I/O may be avoided.

More recent research has focused on the interfaces: between the user and the window system, and between the functional program and the window system.

This section looks at each of these is turn: first at the declarative modelling of the user interface, then at some of the practicalities of interfacing a declarative language with a procedural window system.

### 2.4.1 Models and prototypes

The advantages of using a functional programming language for both formally specifying and prototyping interactive programs were first claimed in the mid-eighties, in particular by Turner [91], Henderson [44] and Alexander [3].

#### Direct execution of prototype systems

Peter Henderson points out the potential of functional programming for reducing the cost of software development: with its simple mathematical basis, it facilitates the design of correct programs. He claims that "functional programs combine the clarity required for the formal specification of software designs, with the ability to validate these designs by execution". His prototyping language `me too` is a modeling tool for system designers. It is an implementation of a formal specification notation in a functional language that allows specifications to be directly executed as prototyping systems.

Formal definition of an interactive system is desirable for various reasons: to facilitate communication about a proposed system, to provide a standard by which an implementation may be assessed, and to allow proofs of formal properties to be carried out. Workers in the area find it convenient to separate out levels of description, and to use different techniques to define these different levels.

#### Levels of description

For example Heather Alexander conceives of a *presentation* layer, concerned with the details of screen appearance and device handling, and a *dialogue* layer, concerned with the protocol of exchanges between the user and the system. The description of the dialogue is expressed in notations that are effectively functional: `eventCSP`, which is a subset of Hoare's CSP for communicating sequential processes, is used to outline the order of events in a dialogue, and `eventISL` which is used to define the actual events, the primitive steps involved. The `eventISL` notation is adaptable to a host language in which it is embedded — in her case `me too` and the programming language `C`.

### Approaches to modeling dialogue

Mark Green [40] surveys three models of the dialogue between a user and an interactive computer system: *transition network*, *context-free grammar*, and so-called *event* models. This last model was not as established as the others at the time he was writing. Based on the concept of *input event*, it is particularly suited to the description of direct manipulation interfaces. Such interfaces were only then coming into widespread use, and there had not previously been any apparent need to account for multithreaded dialogues. Green concludes that the event model has the most descriptive power. However, as the other two may each be translated into the event model, a system designer may use whatever notation is most apposite for the particular application in hand, so long as the user interface management system provides run time support for the event model. As will be seen, the event model has relevance to techniques used in the selection of a transaction in the example program of the next Chapter.

### An early attempt to model a generic user interface with a functional program

An explicit attempt to model a generic user interface with a functional program is described in Steve Cook's paper [21]. The intention is to use generic components to develop families of interactive applications with common user interface characteristics. To do this he proposes using a functional language which has polymorphic functions, higher order functions, and a particular concept of *subtype*. At the time there was no language in which he could implement his ideas. But now there is Haskell with type classes and subclasses which exhibit the required properties. In Cook's parlance:

> "A type $\sigma$ is a subtype of another type $\tau$ ($\sigma \leq \tau$) if $\sigma$ has all the fields of $\tau$, and usually more, and the common fields are appropriately related."

This is very like the Haskell class system, where a subclass has all the *methods* of its superclass, and possibly some of its own. Haskell classes, however, are restricted in that they may only be parametrised on *one* variable, the instance of the class, so the system is *not* in fact used in the declarative description of the interface to be developed.

### The PIE model

Colin Runcimans's PIE model of interactive systems is extensively developed by Alan Dix [24]. It formalises the essence of such a system: there is input, and interpretation of this by the system to yield output. The input is labeled: `P`, for `Program` — meaning the sequence of commands directed to the system; the interpretation: `I` for `Interpretation`, and the output: `E` for `Effect` (hence the acronym `PIE`). The interpretation is a *function* from input to output.

The model, with appropriate extensions, may be used as a focus for detailed formal expression of principles of interaction at all sorts of level of sophistication and complexity. It is attractive because of its simplicity, its generic nature, and, in the context of this thesis, the possibility of directly expressing the model in a functional language [73]. This is an example of the creation of an executable prototype in fulfilment of a specification, that is effectively more than a mere prototype: it *is* the system that was specified.

## 2.4.2 The interface to the window system

In one sense the interfacing of a functional program with a window system is but a special case of the problem discussed in the first part of this chapter, of relating the pure declarative style to the (nasty) real world of side effects, sequencing and multithreading. It deserves separate consideration, however, because this special case is crucial to the increasing proportion of applications that require the use of graphical workstations, and because some proposed solutions to the problem exist already.

This section also serves to give a brief overview of the state of the art as context for the choice of the **MGR** window manager in the application to be described in the next chapter. This may be slightly misleading, however, as most of the systems to be described did not exist at the time the program was being developed.

### An intermediate imperative program

An obvious solution to the problem of interfacing a pure functional program with a window system is to use an intermediate imperative program. This interprets output from the functional program into commands for the window system, and translates output from the window system into input for the program. Merging of streams of input, for example from the mouse and keyboard, may be performed either by the window system, or by the intermediate program. This scheme is like that for I/O with a strict language: the declarative and non-declarative elements are kept strictly separate, so the aspects of the implementation that are amenable to transformation, for example, are clearly delineated.

### The use of MGR

**MGR**[5] is for "ManaGeR" [95]. It is highly suitable for use with a lazy functional program because there is no need for an intermediate imperative program, nor for the program to do any merging of input. Any language that can output strings can be used to write **MGR** applications as **MGR** responds to commands that are escape strings — strings the first character of which is ESC — and passes to standard output any that are not. **MGR** is also responsible for the merging of input from the mouse and input from the keyboard.

The functional program receives merged input from **MGR**, and calculates output including escape strings for the window manager. **MGR** does not provide features like scrollbars that programmers are beginning to expect, for example from X window system toolkits — though such features may be derived form the lower level facilities that *are* available. This is an advantage, in that the display is not pre-customised to a standard form, but also a disadvantage as the programmer has to define most details of the display explicitly. **MGR** was chosen for the application described in the next chapter.

### The TONEWS character in LML

In LML [9], the output of a program is normally printed on standard output. There are ways of directing output to files. There are also a number of special characters that will redirect the rest of the output. These include `TONEWS` which opens a channel to the `NeWS` [38] window server and permanently redirects both input and output to it.

Using `TONEWS` a functional program can set up communication between itself and the window manager. The language is also able to do polling and merging of input: if a program is in `hiatonic` mode it does not hang if there is no input, instead a `hiaton` is returned, indicating that no normal character is available.

Both `hiatons` and `TONEWS` are primitives which extend the language in a practical way. The next system to be described involves another extension to the LML compiler.

[5] **MGR** was developed at Bellcore by Stephen Uhler. It is freely available by ftp from `flash.bellcore.com`, and versions exist for various different platforms, including Sun 3, sparc, dec3100 and Macintosh.

### Fudgets

"Fudgets" is the name given to *functional widgets* (window gadgets) by a team working at Chalmers University [18]. Although they are using LML and Haskell, the principles involved are not language specific, and the GUI toolkit that they are implementing manipulates the X window system — though, again, the choice of window system is not crucial to the basic idea.

They have developed a library of fudgets that implement common user interface elements including buttons, menus and scrollbars. This will form the beginning of a comprehensive GUI toolkit. But "A fudget program is ... a hierarchy of concurrent processes communicating with each other and with the world" and the fudget concept has been used to do standard Haskell I/O, suggesting that the system being developed is a specialisation of a general way of structuring interactive functional programs.

### The Concurrent Clean system's I/O interfaces

Concurrent Clean [96] is an experimental pure, lazy, functional language that was originally designed to be used as an intermediate language between arbitrary functional programming languages and arbitrary machine architectures. It may also be used as a language in its own right, in which computations are expressed in terms of graph rewriting. As mentioned in Section 2.2.3, the language allows the definition of *unique* types, values of which have only one path to the root of the graph, *i.e.* are not shared so need not be copied when their value changes. This allows such unique objects to be updated without danger of losing referential transparency.

The relevance of Concurrent Clean here is that a programming environment for the language has been developed which provides amongst other things a "high level I/O interface with the Macintosh toolbox and with the X Window System". In conjunction with the use of unique types, this enables the functional programmer to write efficient graphical applications. Limitations are firstly that Concurrent Clean used as a programming language is very terse, so the programmer needs to customise it, and secondly that the explicit environment passing used to implement I/O, in particular in relation to *event* I/O used for graphical applications, requires that the number of interface objects be fixed.

### Glasgow Haskell's `ccalls` to X

Glasgow Haskell's `ccall` extension, also mentioned in Section 2.2.3, may be used to link a Haskell program to any other system, an in particular a window system such as X. Current work in Glasgow includes the implementation of combinators similar to fudgets, called *budgets* [72]. These are built on top of an interface to the Openlook widget set, and mostly correspond directly with widgets of OLIT (Open Look Intrinsics Toolkit).

### Yale Haskell's interface to CLX

Finally, the Yale Haskell implementation is now also offering an interface to the X Window System that is built on top of the Common Lisp X interface [79]. As with the Glasgow system it uses an IO monad to control the sequencing and single threading.

## 2.5 Motivation for the Escher program

This chapter has outlined various ideas for overcoming the apparent problems in writing interactive graphical applications in a lazy functional language. Pioneering applications are presented as evidence that this can be done. The next chapter describes a slightly larger application written to see whether the benefits of functional programming are still evident, or whether they become outweighed by performance considerations. The Escher program allows a declarative expression of the interface to be implemented, where a mouse click represents the application of a function that "interprets" the interface.

The program is also a preliminary exercise for the more substantial programming environment, implemented in Haskell, that is the basis for discussion of the second part of the thesis.

# Chapter 3

# The Escher program

## 3.1 Introduction

This chapter describes the implementation of an interactive graphical program in a lazy functional language. It investigates:

1. advantages and disadvantages of using a lazy functional programming language for such an application;
2. whether the performance of the program is satisfactory — *i.e.* the first aspect of "See how they run";
3. a declarative implementation of the user interface, including:
   - the representation of a mouse click as a function application;
   - the incorporation of principles of user interface design;
   - the viability of a generic functional model of interaction.

There is first an account of the application from the user's point of view; then the implementation is discussed, ending with an account of the interface; the program is reviewed according to each of the points above; finally a "Future work" section proposes possible extensions to the program, and work deriving from its implementation. The complete text of the Haskell version is given in Appendix A.

## 3.2 User's view of the program

The application of this chapter is an interactive graphical design program. It is potentially of more than recreational use, as the patterns that it enables users to create often resemble wrapping paper, or wallpaper[1] (Figure 3.1).

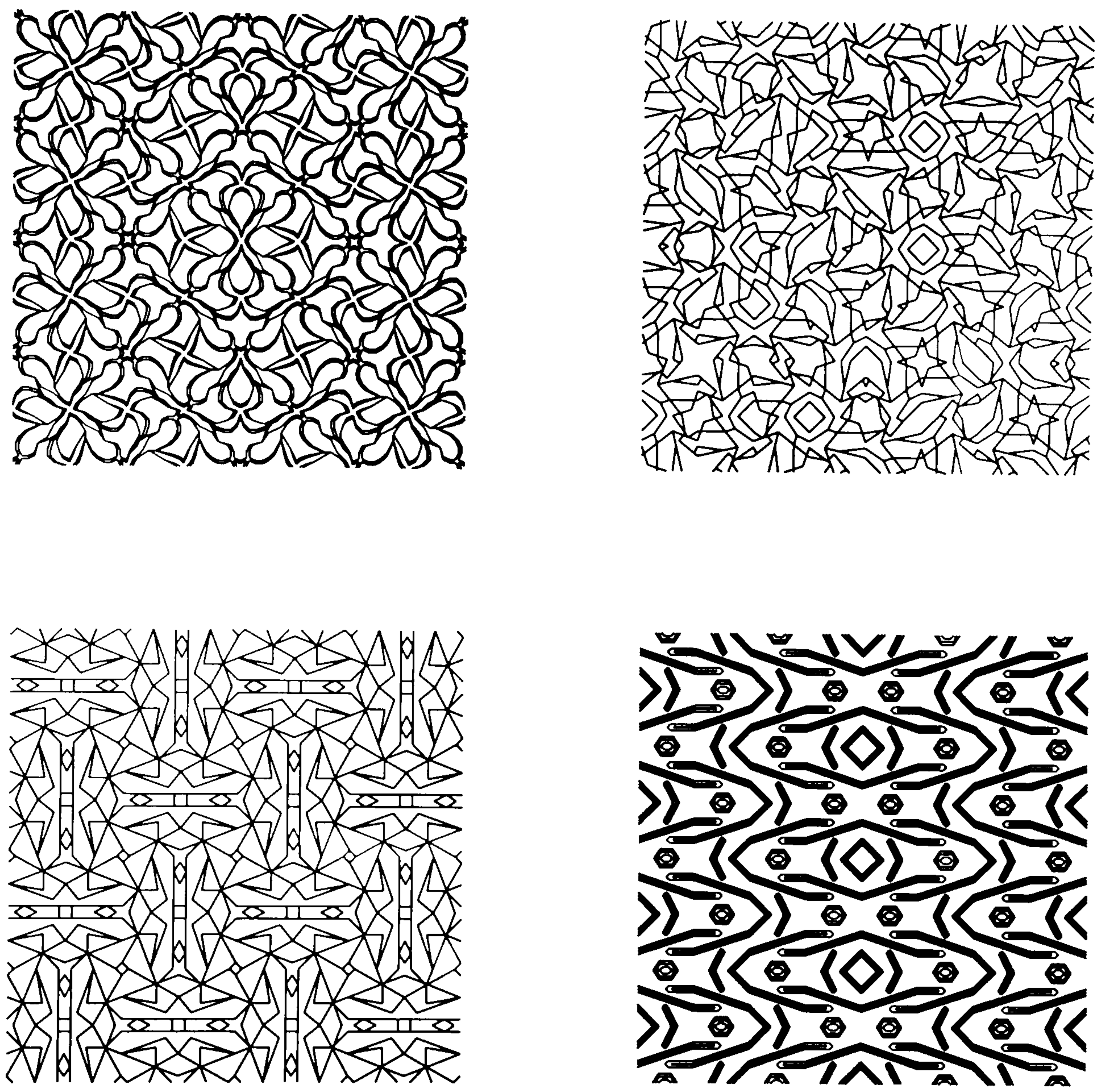


Figure 3.1: Some patterns created with the Escher program.

### 3.2.1 Outline of the program

The application builds on Henderson's work on Functional Geometry [45]. This article, written in 1982, only references a book about the artist M. C. Escher [31]. This program, too, was inspired by Escher and incorporates the functional manipulation of graphical patterns. There are important differences, however. Whereas Henderson's functions were aimed at

[1]There has, admittedly, been some concern expressed regarding my taste in wallpaper.

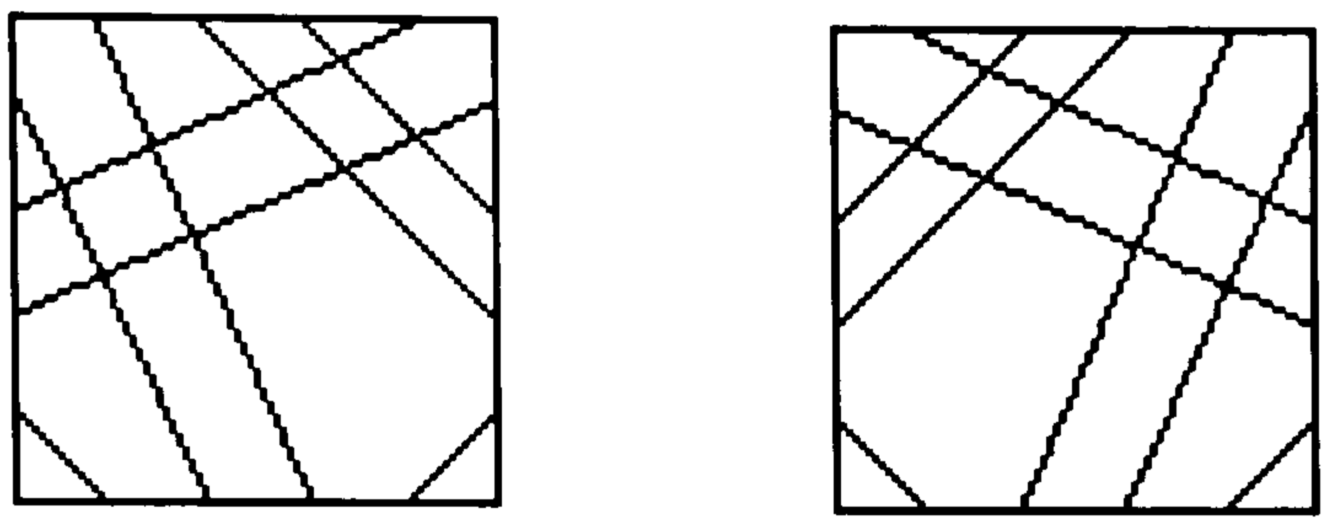


Figure 3.2: Escher's stamps.

combining *given* pictures that "fitted together" in certain combinations, the scheme described here helps the user to *design* pictures that can be combined in Escheresque ways, not just to do the combining. Another difference from Henderson's work is that the program uses interaction, in conjunction with the use of a workstation.

The program is based on an idea arising from a game used by Escher in 1942, described in *The magic mirror of M. C. Escher* [30]. He carved lines on a square stamp to intersect the four sides in the same relative places — when prints from a stamp are used for tiling, continuous lines are obtained, whatever sides of the square are adjacent. Escher also carved the mirror image of the first stamp (Figure 3.2). Using these two stamps in any given square, eight different prints may be obtained by rotation. From such prints Escher created patterns, illustrated in Figure 3.3.

### 3.2.2 Using the program

A user of the program creates designs, corresponding to stamps, and tiles a display area with their rotations and reflections to make a pattern (Figure 3.4). Phases of the design are re-

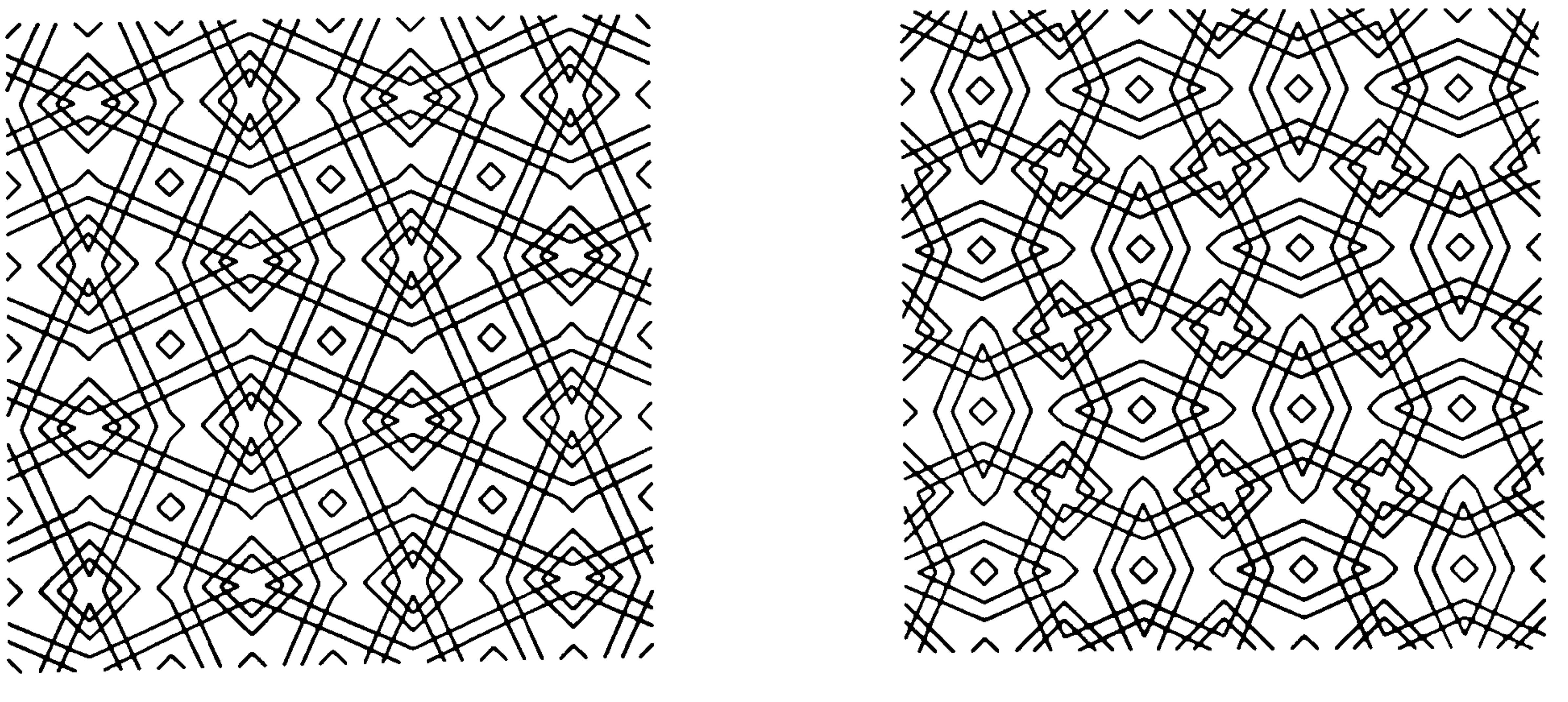


Figure 3.3: Escher's patterns.

flected in *modes* that determine the appropriate action associated with a mouse click.

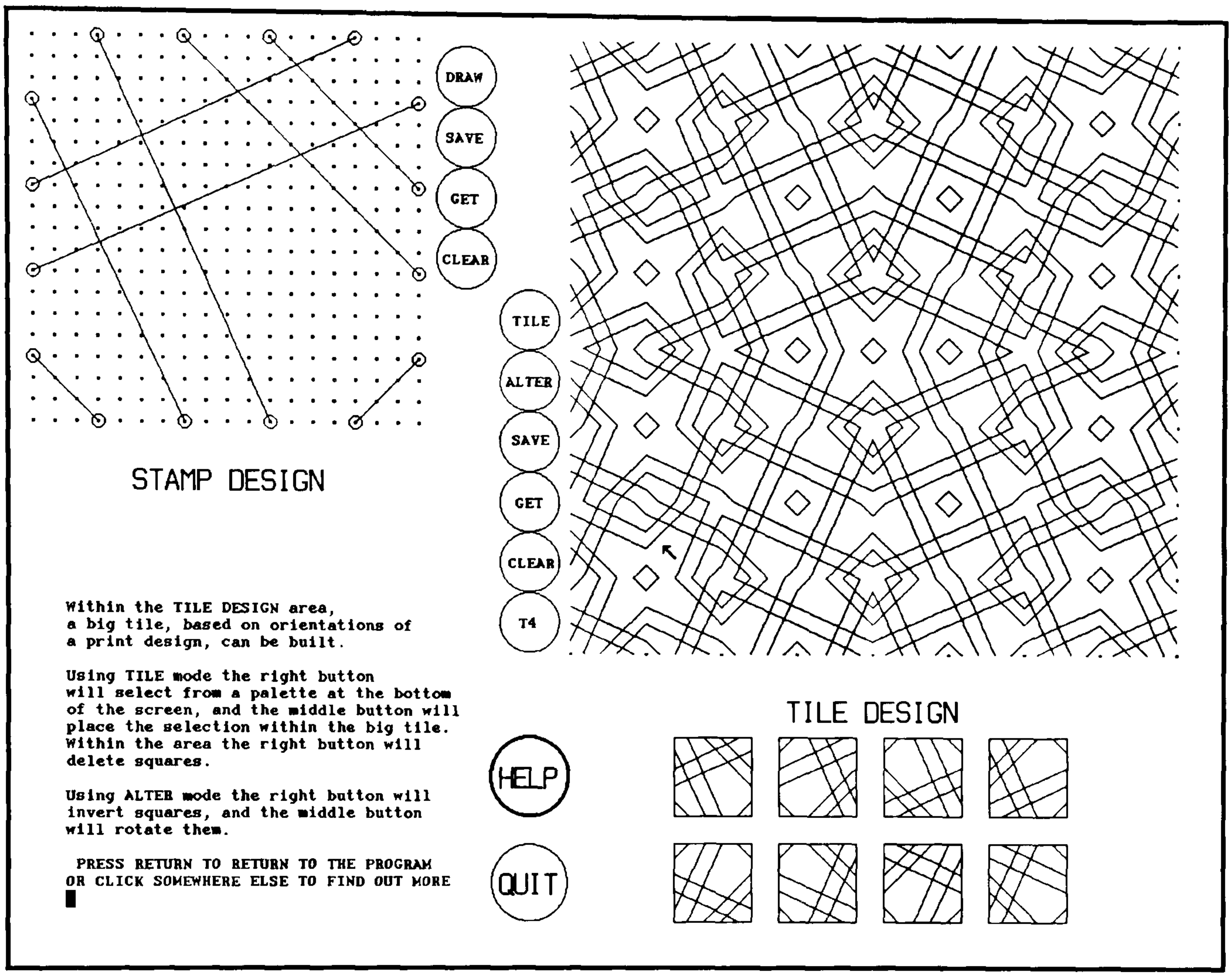



Figure 3.4: A sample screen.

### `Draw` mode: creating or modifying a stamp

There is a 19 × 19 grid in which to draw lines that define the stamp. Lines are drawn by depressing the middle button at the position of one end of the required line, and holding the button down until the position of the other end is reached. During this process the line is *rubberbanded* by the window manager until the button is released. Only if both ends of the line are within the `STAMP DESIGN` area is the line added to the existing stamp. The ends of the lines are adjusted to points on the grid, and gently curved lines may be simulated by polylines. Unwanted lines may be deleted by clicking with the right button near the middle of the targeted line. As the stamp is created, a miniature version of its progress may be observed in the lower right hand portion of the screen. The program encourages the user to make a stamp that will combine neatly with different orientations of itself, by marking all four edges of the square grid with little circles whenever a line is drawn that touches any one of them. These indicate the positions that must be incorporated into the stamp if designs built from it are to be continuous. This is illustrated in the STAMP DESIGN area of the sample screen in Figure 3.4, where the simulated Escher stamp is seen to touch the edge in two places: 3 dots and 7 dots in from the corner. Each of these is associated with 8 little circles, which are drawn at the same relative places from *all* corners. The sample screen also shows the eight orientations of Escher's stamp and yet another picture built from them.

Four circular screen buttons next to the STAMP DESIGN area form the DRAW menu. The top button, when marked, indicates that the program is in Draw mode, and that lines will be rubberbanded. Normally clicking on this button will put the program into Draw mode if it is not already. The SAVE button enables stamps to be "saved", coded as UNIX text files. Clicking on this menu button initiates a dialogue in which the user is prompted for the name of a file under which to save the stamp. In a similar way the GET button initiates a dialogue for *retrieving* a previously saved stamp — this replaces whatever design is present.

### Tile and Alter modes: building a pattern

Once the stamp has been formed, or a previous one restored, it can be used for creating a pattern on the larger grid. Next to this grid are five circular screen buttons that form the TILE menu.

Clicking on the TILE button puts the program into Tile mode, and causes all eight miniature stamps to be displayed. These may then be selected with the right hand mouse button and subsequently positioned with the middle button in the TILE DESIGN area — a $9 \times 9$ grid of dots, to enclose $8 \times 8$ stamps. Stamps in the TILE DESIGN area may be deleted by clicking over them with the right button.

Selecting the ALTER button allows prints that are already in place in the larger grid to be individually rotated (middle button) or inverted (right button).

The SAVE button is used to save a picture, and, as with the stamp SAVE, prompts the user for a filename.

When a previous picture is retrieved, through the use of the GET button, the *orientations* inherent in the picture are imposed on the *current* stamp. However the picture is also saved as a PostScript file to be printed out — from outside the program at present — or incorporated into a document (such as this one). The GET button may also be used to impose *predefined* patterns of orientation onto the current stamp, for example those used in the creation of Escher's pictures. The names of these predefined patterns are, however, not displayed — though they may be seen through the use of the HELP system (see below).

As with the DRAW menu there is a CLEAR button, which clears the grid.

Finally the `T4` button provides a token, and limited, form of tiling the whole area with a repeated pattern — ideally the largest patterned rectangle to be found in the grid, or that, for example, in the top left corner, but in fact it takes the 4 tile square in the top left corner, and patterns the area with this.

#### `Help` mode: the help system

Clicking on the `HELP` button puts the system into `Help` mode: in this mode a mouse click does not result in the action itself, but in the display of text *describing* the action. Pressing <<CR>> to leave the Help mode puts the program into `Draw` mode. Figure 3.4 shows the display in `Help` mode, with the `Help` button marked with an extra circle. A mouse click has occurred over the `TILE DESIGN` area, so text appropriate to that is shown.

#### The quit "mode"

The `QUIT` button allows the user to quit the application elegantly — though they can also quit by typing "`q`". This may be considered a "mode" as the action of mouse buttons is altered by their ceasing to have an effect on the output of the program, but `Quit` does not need to be coded as a mode.

### 3.2.3 How user interface principles are observed

Two examples of principles of interface behaviour are: that the user should be free to decide in what order to do things [23], and that there should be consistency in the use of the mouse buttons. Both principles can be followed if we arrange that each mouse click represents a function application, the result of which is clearly reflected to the user in the interface, and that there is consistency in the effects of each button's function applications. The user then has a good model of what is going on, and is free to do things in any order. The designer, in turn, does not need to anticipate the possibly idiosyncratic requirements of particular users. As will be seen in Section 3.3.4, the need for modes constrains the possible order of events to some degree, but even between modes there is consistency in the use of mouse buttons.

The left button is unavailable to applications that use **MGR** as it is permanently reserved for system use, so it is the action of the middle and right buttons that is in question. We use the middle button to do things, such as draw lines and place tiles, and the right button to complement this by *selecting* lines and tiles for deletion, and tile orientations for placing. The right button is also used to select screen menu buttons; and when the tiles in the design are individually rotated or inverted, the middle button does the rotating and the right one does the inverting, which can be regarded as selecting the other mirror image of the stamp.

The different ways in which the interface may respond to similar user actions, depending on the prior history of the interaction, correspond to what are referred to here as *modes*. Modes are needed because we have only two mouse buttons available, yet there are more than two transactions appropriate to each area of the screen. For example, while the graphics cursor is within the design area, a middle button press can be used to place a tile, or to rotate one, depending on the mode. Hence a *state* beyond the values intrinsic to the application, one which incorporates the mode, is required. As will be seen, the Escher program has a state that includes the `Mode`.

## 3.3 Implementation of the program

Here we have:

- a brief overview of the program which introduces the chosen window manager: **MGR**;
- a view of the program as the specialisation of a generic interaction function;
- the elements of the program state for the Escher program, in preparation for
- a fairly detailed description of the programming of the interface, which demonstrates how this may be regarded as a specialisation of a generic interface interpreting function.

### 3.3.1 Overall view

Here is the layered architecture of the system as a whole:

*Application Program*

**Functional Programming Language: Haskell**

Window system: **MGR**

`Workstation:  Sun 3 / Sparc`

The choice of window manager is **MGR** [95]: "Client programs communicate with **MGR** *via pseudo-terminals* over a reliable byte stream. Each client program can create and manipulate one or more windows on the display, with commands and data to the various windows

multiplexed over the same connection."
**MGR** is network transparent, like X Windows [78], though much smaller and simpler. The direct connection with the window manager obviates the need for an intermediate program in order to communicate with the functional program. The program runs happily on both Sun 3 and Sparc workstations.

### 3.3.2 Interaction

The interactive process exploits lazy evaluation: all the input that the program is going to receive has to be represented in the expression that is the program. It is essential for this style of interaction that the programming language allows unevaluated expressions to be manipulated by its programs. The interactive program evaluates the input *by need*, allowing lazy evaluation to enforce the desired sequentiality.

The program is an application of a generic, higher order, combinator called `inter`, to appropriate arguments. Figure 3.5 shows the definition of `inter` in Haskell.

```
inter :: (state -> [Input] -> Bool) -> TransD state ->
         (state -> [Input] -> Dialogue)
inter endp transf = inter'
     where
     inter' state input resps
            |endp state input = []
            |otherwise        = out ++ outs
         where
         (out,state',input',resps') = transf  state  input  resps
         outs = inter'  state' input' resps'
-- Transaction combinator modified to keep track of Responses
type TransD state = state -> [Input] -> [Response] ->
                        ([Request], state, [Input], [Response])
```

Figure 3.5: The `inter` combinator.

It is a wrapper function that extracts output, in the form of `Responses`, from successive applications of the transaction function `transf`. This has to be a transaction combinator with a type modified from that of Thompson/Dwelly combinators to incorporate `Requests` and `Responses` so that it conforms with Haskell's treatment of I/O — bearing in mind that a Haskell program is of type `Dialogue` *i.e.* `[Response]` $\rightarrow$ `[Request]`. The type of `transf` is also given in Figure 3.5: it takes a state, a list of inputs (each one in our case a `String`), and a list of `Responses`, and returns a quadruple consisting of a list of `Requests`, which is the output to be captured, a possibly modified state, and the input an `Responses` yet to be received. The program needs to keep track of `Responses` as it needs to refer to particular `Responses`, to access the contents of particular files, when retrieving previously saved stamps and patterns. `endp` is a condition on the state and inputs, that indicates that the program is finished — in the case of the Escher program, there is no terminal *state*, so a function `[Input] → Bool` would suffice.

A more general version of the `inter` function also outputs a prompt appropriate for the state at each step of the interaction. This is, however, unnecessary here as the screen display serves as sufficient cue to the user. Note that the user input is separate from the list of `Responses` in the definition. A wrapper function, in this case `main`, is needed to extract this from the `Response` to `ReadChan stdin`.

**MGR** directs user input, as a list of character strings, to the application program. It can also be asked to return a string when an *event* occurs, such as the press of a mouse button. Such strings are simply incorporated into the program's input. They may contain substitutable parameters: for example `%p` will be replaced by the coordinates of a mouse click. Indeed, most input strings to the program represent a mouse click, though some, such as the name of a pattern to retrieve, represent keyboard input. The list of `Requests` consists mainly of escape strings for **MGR**, directed to standard output. Some of them, however, direct text to files, *e.g.* PostScript coding of patterns.

### 3.3.3 Program state

The program `state` is defined in Figure 3.6.

```
type State  = (Mode, Stamp, Sel, (Board, Board),Flag)
data Mode   = Draw | Tile | Alter | Help
type Stamp  = [([Int],[Int])]
type Sel    = Int
type Board  = [((Int,Int),Sel)]
data Flag   = Dsave | Dget | Dclear | Act |
              Tsave | Tget | Tclear | T4
```

Figure 3.6: The Escher program state.

The `Mode` characterises the actions initiated by a particular button press, as described in Section 3.2.

The `Stamp` consists of the lines that make up the current design together with the coded position of their edge connections, if any.

The `Sel` is the current orientation used when putting stamps on the `Board`. It is coded as an `Int` ranging from 0 (blank) to 8. An alternative is to use the orienting function itself, but a coding scheme is needed for identification of selection boxes, and in the transcription to PostScript, so it is convenient to use the same type in the program state. It might be clearer, though, to use meaningful codings, *i.e name* the orientations, and translate these into numerical coding for the PostScript when needed.

A `Board` is a list of orientations, each associated with a square on the pattern grid. The second `Board` was introduced to eliminate a space leak when a previously saved pattern is being retrieved (see Section 3.4.1).

The `Flag` is used primarily to signal an interaction that involves file handling, and will incorporate more than one `Request -> Response` pair. Under "normal" circumstances, which involve a mouse click and a corresponding change in the screen display, the `Flag` is `Act`. Clicking on a menu button that entails a file interaction, causes the correct `Flag` to appear in the state. The transaction function is so defined that, when this happens, a special interaction appropriate to the flag is started. The type `Flag` is also used when temporarily marking menu buttons, hence includes representatives for all of them — apart from `Mode` buttons, as these are marked according to the current and previous `Mode` when the mode is changed.

**State transitions**

Figure 3.7 illustrates the state transitions in terms of `Mode` and `Flag` changes when a menu button is pressed. Mouse clicks over other areas initiate the appropriate interface transaction according to the overall Escher interface interpretation (see Section 3.3.4). When the program starts up the mode is `Draw`, and the final mode change is to `Quit` by a click on the `QUIT` button. When the mode is `Help` there is only one possible mode change, which is to go to `Draw` mode by pressing the carriage return key. In other modes the menus may serve to *change* mode, while leaving the `Act` flag operative. Menu buttons labelled `SAVE` and `GET` leave the mode unchanged, but initiate a transaction with the user that involves the output of a prompt, and the reading in of a filename.

### 3.3.4 The Interface

The implementation of the interface has to meet various requirements. We must observe principles of interface behaviour, take into account peculiarities of the particular window system being used, and exploit the lazy functional style.

The interface may be described as a collection of `areas`, each of which has a `display` element and `transactions` associated with it for each `button`. Functions are defined to

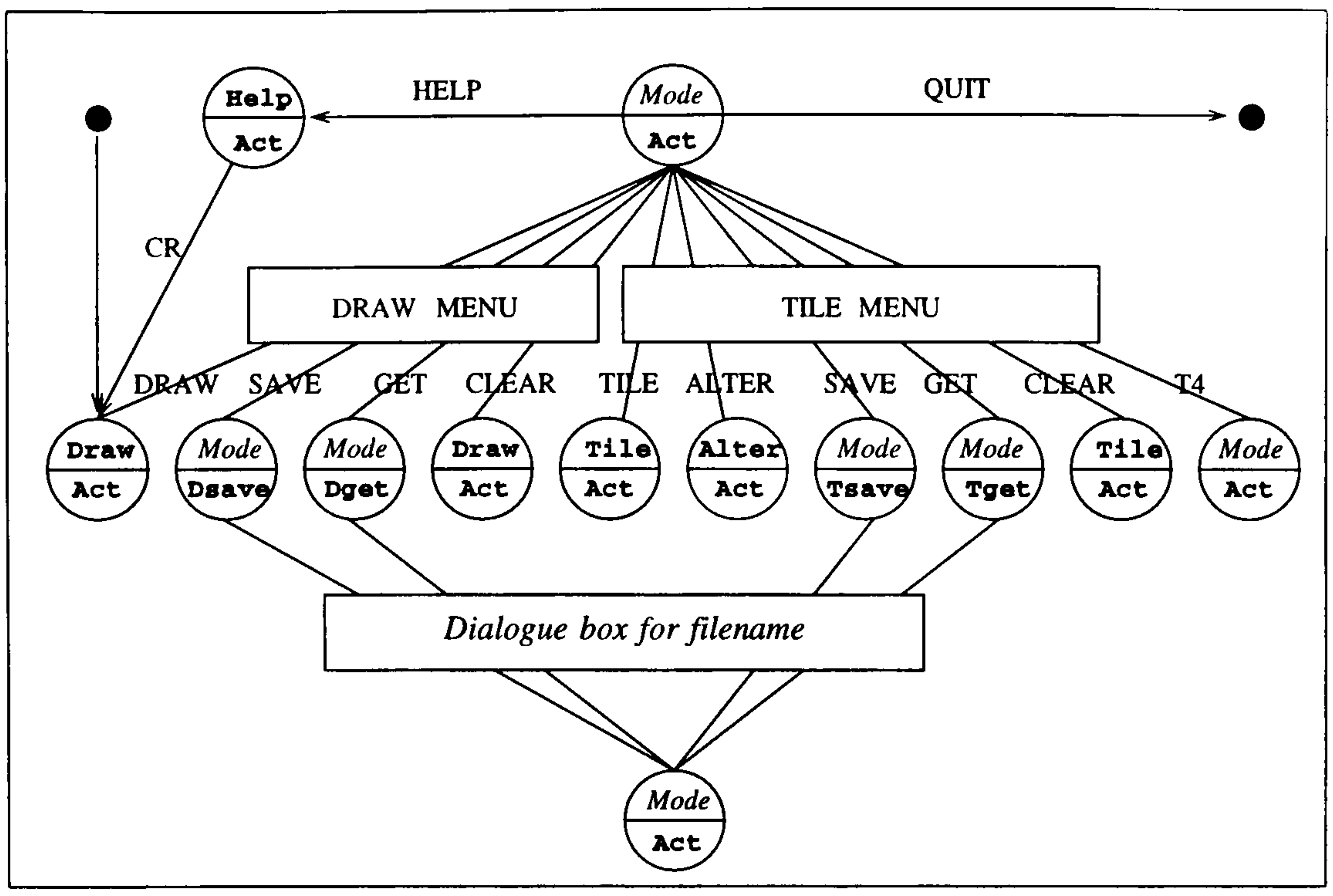



Figure 3.7: State transition diagram for the Escher program.

extract from the interface description the actual display or the function associated with a particular mouse click. Ideally a transaction would depend solely on which button was pressed and what area of the screen the graphics cursor is in at the time of a mouse click. This part of the interface description would have the type:

$$\texttt{Button} \rightarrow \texttt{Area} \rightarrow \texttt{Transaction}$$

However, mouse buttons have many uses in the program, and, as shown in Section 3.2.3, there may be more transactions appropriate to an area of the screen than there are mouse buttons available. This implies that the user must, on occasion, explicitly change mode before performing a desired action — for example, after placing a tile he may wish to invert it, but must first change from `Tile` to `Alter` mode.

### Peculiarities of MGR

Mode changes are also necessitated by the particular window manager being used. If **MGR** permitted mouse clicks to have a different effect over different areas of the screen, perhaps depending on the particular window under the cursor, there would be less need for mode changes. But, in **MGR** the need for rubberbanding in response to a mouse click has to be known in advance. It is not *essential* to hold such mode information in the functional program — as it *has*, in any case, to be held by **MGR** — but the simpler the program state, the more complex the messages for **MGR** must be. It is convenient to keep the mode in the program state, however, as the mode may then be subject to pattern matching. This is used, for example, in the definition of the function `tilef` given in Figure 3.8 and explained in the next subsection. The function `tilef` is applied when a button click is received in the `TILE` area.

```
tilef :: Button -> Coords -> State -> (String, State)
tilef button coords (mode, stamp, sel, (board,_),_) =
        ((undo . tplace) oldas ++ tplace new,
                                              -- MGR instructions
        (mode, stamp, sel, (board',[]), Act))
                                              -- new state
        where
        atile   = sqid coords                 -- the particular tile
        wcoords = wscale stamp                -- scaled stamp
        oldas   = assoc atile board           -- old orientation
        tcoords = btlocate coords             -- coordinates of the tile
        board'  = newas atile new board -- new board
        new     = case mode of                -- new orientation
                        Tile  -> case button of
                                      R -> 0          -- delete
                                      M -> sel        -- from state
                        Alter -> case button of
                                      R -> inv oldas -- invert old
                                      M -> rot oldas -- rotate old
        tplace o = put tcoords (orient xymax o  wcoords)
```

Figure 3.8: The action represented by a click in the `Tile` area.

Another possibility is to keep a *function* appropriate to the mode in the state, but this has the disadvantage that the mode cannot be directly accessed, yet is needed to enable the correct menu buttons to be labeled/unlabeled when changing mode. Of the four modes in the program: `Draw`, `Tile`, `Alter` and `Help`, the first three have as much overlap as possible so that the user need not usually be aware of the current mode, although there is clear visual indication of this. In particular a click on a menu button from any of these has a consistent action.

The presence of **MGR** as the windowing system also has implications for the nature of the program. Some of the facilities offered by **MGR** invite the application to hand over some of its control, for example the *event* strings mentioned earlier.

More significantly, **MGR** receives escape strings computed by the application and translates these into appropriate changes in the display and its own state. Thus an element of the program's output stream can have a *representation* that is a *change* in **MGR**'s state, and/or a *change* in the display. Nevertheless, as discussed in Chapter 2, referential transparency is *not* violated. We have a declarative program computing an imperative stream of messages. The fulfilment of the program specification depends on the mediation of **MGR**, as well as on the action of the program itself. However, almost of all of the application's complexity is coded in the functional program structure, not in the interpretation of the messages.

### Pattern matching on the mode — the `tilef` function

Here is a procedural account of the definition of `tilef` shown in Figure 3.8. The function `tilef` identifies a tile, `atile`, the square associated with the coordinates `coords` specified by the mouse click. It deletes the existing orientation of the stamp associated with that square, `oldas`, and places the `new` orientation there instead. This `new` orientation is obtained by pattern matching on the `mode`, `Tile` or `Alter`, and the button, `R` (right) or `M` (middle). A new state is also returned. This incorporates the new board, `board'`, leaving the `mode`, `stamp` and `sel` (current selection) unchanged, and confirming the current `Flag` to be `Act`.

### A mouse click as a function application?

The coordinates, `Coords`, that are used to determine in which display area a mouse click takes place, may also be needed in the transaction that the click represents. For example a right button click over one of the displayed orientations makes that orientation the current selection. The relevant screen area is the whole *group* of orientations, but the *particular* orientation over which the button was pressed is also needed to allow the transaction to proceed. Thus a screen area is associated with a transaction that also depends on the `Mode`, `Button` and `Coords`. A fixed interface is a collection of such elements. In the present case a *list* is suitable collection, but in another application, for example where disparate active areas are scattered over the screen, a tree structure might be more appropriate. The choice of data structure is dictated by how it is accessed in the program. The interpretation of an interface is a repeated cycle of identifying a display area that is subject to a button press, and applying the corresponding function to yield the transaction. Thus a mouse click does, indeed, symbolise a function application. The body of the text of the Escher `Interface` module is shown in Figure 3.9.

The `interpret` function searches through the list of `FindActs`; when it finds one where the `pt`, the point clicked on, is `inFA`, this identifies the action, `actFA`, to be applied. `Trans` represents the transaction combinator type. A dynamic interface might extend the type of a display element to include the function for its own display. This could be neatly encapsu-

```
type Interface = [FindAct]

-- FindAct has two functions: one to recognise mouse clicks,
-- the other to return the appropriate action
data FindAct   =
     FA (Coords -> Bool) (Mode -> Button -> Coords -> Trans)

inFA :: FindAct -> Coords -> Bool
inFA (FA pb _) pt = pb pt

actFA :: FindAct -> (Mode -> Button -> Coords -> Trans)
actFA (FA _ tfun) = tfun

interpret :: Interface -> Mode -> Button -> Coords -> Trans
interpret [] _ _ _           = notrans []
interpret (fa:rest) m b pt = if   inFA  fa pt
                             then actFA fa m b pt
                             else interpret rest m b pt
```

Figure 3.9: `Interface` type and associated functions.

lated in an extended interface element type, that includes the information to be sent to the window manager to display the area, ToMGR:

```
data DrawFindAct =
DFA ToMGR (Coords → Bool) (Mode → Button → Coords → Trans)
```

The display information, and the function for finding the displayed area, can be defined together to ensure that these are synchronised, *i.e.* the relevant active area correponds to the one displayed. There can then be a menu building function, that at once displays the menu, including button labels, and defines its active areas and their actions.

In the Escher program there is no need for such a menu function, as the menu buttons are permanently displayed; a menu, however, is described as an area which itself is an interface to be interpreted. The definition of the Escher interface as it appears in the program is given in Figure 3.10.

It can be seen, for example, that the *action* when the `tilemenu` is selected with a mouse click is to use that same click in the interpretation of the *tile menu* interface: `tmenu`.

## 3.4 Assessment

Here the implementation of the Escher program is assessed in relation to the points mentioned in the introduction to this chapter.

```
escher_interface :: Interface
escher_interface = [FA indesign   desfun                ,
                    FA indesmenu  (interpret dmenu)    ,
                    FA inbigtile  tilefun               ,
                    FA intilemenu (interpret tmenu)    ,
                    FA inpicarea  orifun                ,
                    FA inhelp     helpfun               ,
                    FA inquit     quitfun ]
```

Figure 3.10: The Escher interface.

### 3.4.1 Advantages and disadvantages of using a lazy functional language

Some of the factors listed under "The virtues of functional programming" in Chapter 1 are exploited in the application: directness, use of higher order functions and lazy evaluation.

#### Directness

Just as in the implementation of functional programs there is "delight in the close interplay of theory and practice" [68], there is also tremendous gratification resulting from the expression of ones ideas directly in the code, without assuming any details about implementation. The use of a functional language enables and encourages a precise reflection of the structure and functionality of the program in the structure and detail of the code. Take for instance the `Escher` module. This encapsulates the program's interaction. It imports the overall active areas from `EscherAreas`: the design, design menu, tile and tile menu areas, the orientation boxes and the Help and Quit buttons. The `escher-interface` function relates each of these to its action — including relating the menu areas to their respective actions as defined in the imported `Tmenu` and `Dmenu` modules. This conceptual grouping reflects the specification of the program in the code. It also echoes the *visual* grouping of areas, so the three views of the program: the specifier's, the implementor's and the user's, are consistent and plainly related. The whole code of the program is included as Appendix A so that the detail of the interrelationships may be examined.

#### Higher order functions

There is extensive use of higher order functions in the program. For example a picture is a list of lines, each represented by a list of end coordinates: `[x0,y0,x1,y1]`. The function `toright` which moves a picture to the right, is defined entirely by composition of other functions: `toright ::  Int → [Line] → [Line]`

```
toright = map .  mapx .  (+)
```

Functions, and partially applied functions, may be passed between modules as values. For example a function for drawing a grid, defined in a `Geometry` module, is used for drawing both the stamp design and tiling areas as grids of dots. The `Draw` and `Tile` menus, however, may also be drawn as grids: single column grids of circles. The definition is shown in Figure 3.11.

```
grid :: Int -> Int -> Int -> Int ->
        (Int -> Int -> [Char]) -> Coords -> [Char]
grid xgap ygap xlength ylength drawf [xor,yor] =
        concat [drawf x y | x <- x0list, y <- y0list]
        where
        x0list = gridlist xor xgap xlength
        y0list = gridlist yor ygap ylength
        gridlist orig gap len = take len (iterate ((+) gap) orig)
```

Figure 3.11: The `grid` function.

It has several arguments: the `xgap, ygap :: Int` determine the horizontal and vertical spacing of the grid elements; `xlength, ylength :: Int` are the number of grid elements in each direction; `drawf` is a drawing function that, given a pair of coordinates will return a string which, when picked up by the window manager, causes the appropriate shape to be drawn; finally, `[xor,yor] :: Coords` represent the origin of the grid — changing this moves the whole grid on the display. The `grid` function is defined with a list comprehension that says "apply the function `drawf` to all pairs of points of which one is drawn from the list of possible `xs`, and the other from the list of possible `ys`."

### Laziness

Lazy evaluation is essential to the control of the interaction. This is described in section 3.3.2.

### A note on debugging

This controlled expression of interaction may be used to provide a convenient channel for debugging. Under the transaction combinator model there is output at every stage of an interaction, even if this is on occasion an empty string. This was exploited in the insertion of "debug statements" during the program development, directly connected to the output stream of the program. For example: wrong lines were being deleted in the creation of a stamp, so the code was changed to augment the list of `Requests`, that included the request to **MGR** to "delete the line with such and such coordinates", by directing to a file the parameter and intermediate result values of the function that identified the line, revealing an instance of numeric overflow. All the "plumbing" necessary for such debugging of a pure functional program [42] is already present if transaction combinators are used. However, one still has to be careful not to affect the strictness properties of the program, only tracing the values of expressions that are also needed for the untraced computation.

**Disadvantages?**

The only restrictions encountered during the development of the program were limitations imposed by the chosen window system — which itself could be extended if necessary. There were minor problems in performance, especially when earlier versions of the Chalmers' and Glasgow Haskell compilers were being used.

### 3.4.2 Satisfactory performance?

The program runs satisfactorily, with only minor "embarrassing pauses" for garbage collection. When it was subjected to heap profiling a source of unnecessary space usage became apparent: retrieving a predefined pattern involved retaining the string of **MGR** instructions to display it. Each individual line of the pattern, coded as an escape string for **MGR**, was repeated in its relevant orientations 64 times as part of the `Request` to redraw the big tile. These escape strings were joined together to form the long argument to that one `Request`. This caused the program to slow down noticeably if the pattern was complex, *i.e.* had more than about twenty lines in it. The effect was negligeable with small examples. Now only the new *orientations* are put into the program state at first. A series of transactions that consume no input, each triggered by the program state, place the tiles from the second `Board` to the first, one by one, until there are no more to place. This is reflected in the first clause of `tiletrans`:

```
tiletrans :: State -> [[Char]] -> [Response] ->
               ([Request], State, [[Char]], [Response])
tiletrans state inpt ~(_:resps) |tilestoput state =
               ([AppendChan stdout str],newstate,inpt,resps)
                where
                (str,newstate) = tput state
```

The condition `tilestoput` on the state does the checking. The rest of the definition of `tiletrans` is given in Appendix A. In this clause the `tput` function returns the string to draw an individual tile, and the state after this has been transferred from the "new" board to the current one.

### 3.4.3 Declarative implementation of the interface?

The program architecture conveniently reflects the interface that is being described. The description of the interface is to a large extent declarative, in particular a mouse click is represented as a function application, as shown in Section 3.3.4: in a given context, it extracts the required behaviour from the system. There is a danger, however, of attempting to adapt a specification to accommodate a simple declarative model, rather than fitting the model to the specification. It is not always appropriate for the limits of an active area to coincide with a displayed outline. In the Escher program, though menu selection requires a mouse-click *strictly within* a menu item as displayed, when drawing lines in the grid the user should be allowed the same margin of error at the edge as in the centre, so end-points *slightly outside* the grid are acceptable. Not only must such margins be incorporated in a general purpose region-selection function, but also there is complexity in labeling menu boxes: the displayed grids have many elements, but only one label, which could be above, below, or to the side of the grid, or even somewhere in the middle of it; every element of the tile and draw menus, by contrast, requires a properly placed label. In addition the label itself may involve a font that is not fixed width. We have not yet implemented a model which permits such flexibility.

#### Incorporation of principles of user interface design

The presence of `Modes` and `Flags` in the program state highlight tensions between a simple declarative expression of the interaction and the particular nature of the application. The user of the system need not, naturally, be aware of any implementation details — it is the state of the *display*, not that of the *system*, of which the user is aware. Thus the declarative nature of the program may, perversely, be used to *hide* the system state from the programmer who, for example, has no inkling that a `Flag` lies behind his dialogue with the program. On the other hand, as we have seen, the user must *explicitly* change mode under certain circumstances, and is made aware, by the marking of mode buttons, of relevant aspects of the current state — such as which transactions are currently possible. Thus usability properties of predictability and observability ([25] page 318) are present, and, where it matters, the state of the system *is* mirrored in the state of the display.

#### A generic functional model of interaction?

The `inter` combinator of Figure 3.5 is one of a family of wrapping functions that may be used to encapsulate interactive programs. Such functions may differ in details, such as the presence or absence of an explicit prompt, but also in the type of their overall result. For example an intermediate stage of interaction, in a more complex program than the Escher one, may need to keep track of the input and responses in order to hand these to the next stage, so the wrapper function itself may be of the type of an interaction combinator.

With the help of such functions, interactive programs may be defined by finding the exact nature of their arguments — in particular the overall transaction combinator. Thus we have a generic model of interaction which, when filled out with the details of an application becomes an executable specification that *is* the application.

## 3.5 Future work

The program is fun to use and frequently results in the creation of satisfying patterns. One can envisage extensions to the program, such as the use of colour and a scrolling area in which to view the overall pattern, which would require but minor enhancements to an implementation that used a window manager offering the appropriate facilities. Various suggestions for other extensions and enhancements have been received. For example: users would like to combine different stamps with the same edge connections in the tiling area. A palette of suitable stamps, previously saved, could be provided for this. The tiled area itself forms a mega-Escher-tile, which would fit with orientations, and even suitably scaled versions, of itself. The Escher program, and its possible extensions, has sufficient variety of features to make it a good vehicle for exploring other styles of interaction, and more sophisticated ways of interfacing with window systems, such as the "fudgets" (functional window gadgets) system being developed at Chalmers' University of Technology in Sweden [18], to interface between LML/Haskell and X Windows.

We note with Dwelly [27] that:

> "...one area of computer science that has still to benefit from graphic user interface design, is that of software environments for functional languages ..."

The Escher program was a preliminary exercise to the design of such an environment, one that will enable the fulfilment of the other aspect of "See how they run" (the visualization of lazy functional computation), by displaying the functional computation as it proceeds. The ability of a program to change its user interface dynamically, together with a mouse click representing a function application, imply that the user can change the program. This is being exploited in the environment where the user is developing a functional program as well as using one. The environment is described in Chapter 5, following a review of current approaches to monitoring and profiling.

# Chapter 4

# Monitoring and profiling

## 4.1 Introduction

The programmer wishing to write a correct and efficient program needs to understand what happens when it runs. But until recently there have been very few tools to monitor the behaviour of functional programs.

If a program is not behaving the way it should, the aim of debugging may be either a program that yields a correct result, or one that runs at the required speed or within a required amount of memory. Usually an inappropriate speed implies "runs too slowly", but it may be that the program runs too fast. For example in the context of a graphical application, a display may not be held for long enough. In all cases of program misbehaviour the problem may be revealed, and understood, if the programmer can *see* what's going on. This understanding may then be used as the basis for changing the program so that it runs correctly. The aim is to gain *insight.*

There was a mention in Chapter 3 of the ease with which "debug statements" may be inserted into the output stream of an interactive functional program. However this technique is restricted to *interactive* programs, and can only be used to investigate a limited aspect of program performance: the nature of fully evaluated intermediate values. In general, monitoring the behaviour of lazy functional programs is problematic. The order of evaluation, though precisely determined in any given sequential implementation, is often not intuitively obvious. So debugging information may be produced in a surprising, and apparently jumbled, order. More importantly, structures involved in the reduction process, such as *closures*, do not have an obvious textual representation.

For these reasons, and because many systems aim to preserve referential transparency, it is not possible to use arbitrary "print statements" to see what is going on. Tools like the UNIX `prof` [71] may not be very revealing as large sections of a computation may appear to be attributed to a higher order function such as `map`.

### 4.1.1 What to monitor?

It is the programmer's task to relate the desired program behaviour with the actual program behaviour — a debugging tool merely presents information. There are three levels of error at which debugging may usefully be directed.

**Errors detectable by automatic checking of the program text.**

These range from insignificant syntax errors, to type inconsistencies that may reflect trivial errors, but, on the other hand, may be symptomatic of, and pointers to, *semantic* errors. Focused and eloquent error messages from the type-checker may be a great boon to the functional programmer. It may also be revealing to allow the compiler to infer the typing of a program, and to compare this with the typing that the programmer intended.

**Errors in the design and implementation of an algorithm.**

These may be detected, for example, by checking assertions about the relationship of a function's arguments to its result. Applying a suspect function to a range of arguments is a technique used by several of the researchers mentioned below [54, 64].

**Errors in performance of a program.**

That is: errors in the speed at which a program runs, and errors in the amount of memory used. These two aspects of performance appear to be closely related — a lazy functional program that is doctored to make use of less space will usually run faster [75] as less time need be spent on garbage collection, and there is the complementary space-time tradeoff where the more memory that the program has *available*, the faster it will run, as memory management takes up a smaller proportion of the program's running time.

Most of the work described below concentrates on algorithmic and semantic debugging, but there are also a few papers concerned with performance debugging [43, 75, 76, 20]. The two are not entirely unconnected: machine considerations may be the cause of apparent semantic errors, for example numeric overflow may result in apparent errors in the otherwise blameless implementation of a correct algorithm.

### 4.1.2 How to monitor?

There are two main approaches to monitoring: one is gathering statistics about the program as it runs; the other is causing diagnostic reports to be output. These reports may incorporate the display of cumulative information, so the approaches are not entirely separate. And whatever data are gathered, there has to be some textual or graphical display of them to the programmer/user.

The rest of this chapter considers approaches to monitoring and profiling under the categories of:

- Routine collection of statistics (Section 4.2)
- Side effecting tracing (Section 4.3)
- Debugging without side effects (Section 4.4)
- Purpose built environments (Section 4.5)
- Profiling graph reduction (Section 4.6)

The requirements for the system to be implemented are then determined in the light of this existing work.

## 4.2 Routine collection of statistics

Various measurements are available from many existing implementations, such as the number of reductions performed, the size of the heap reported at garbage collection time and the number of garbage collections *e.g.* [9, 88, 52]. These can be used as a guide to writing efficient applications, but are not useful in locating either specific errors, or specific sources of excessive time and space usage.

An early example of a profiler that is intended as an aid to tuning the performance of functional programs is described in a paper [5] that is included in the Standard ML distribution [6]. This profiler uses standard techniques of counting function calls and execution time measurement similar to `prof` [71] and `gprof` [39]. The authors describe a modification of such techniques. They maintain a so-called "*pointer-to-current-function-entry*" to determine which function's call-count to increment, rather than using the program counter. There is also a scheme for coping with anonymous functions, that involves making up names for them, such as `f.anon`. These can then be treated like any other function. This low level accounting allows the statistics collected to be more accurately associated with elements of the source code.

Recent work to provide more detailed measurements, and to display their results in a comprehensible form, includes *heap profiling* [20, 77, 75]. Unlike the "window into the store" [54] that can, for example, give the values of variables at stages in the evaluation of an imperative program, the view of the heap offered by heap profiling reveals instead detailed, lower level information about the state of a running program. This information is based on a census of appropriately tagged elements in active memory. Such profiling of the program graph is discussed further in Section 4.6.

## 4.3 Side effecting tracing

The most direct equivalent to "just put in a print statement" in implementations of functional languages is to get the *implementation*, rather than the functional program itself, to do something similar.

### 4.3.1 The Chalmers `hbc` compiler

Indulgent environments such as the Chalmers LML/Haskell system, developed from the original Lazy ML compiler [9], provide side-effecting tracing facilities which can be used during program development without necessarily compromising the referential transparency of the final program. A compilation option results in the availability of vast quantities of trace material, which may be examined using checkpoints of named function applications. A particular argument may be evaluated to weak head normal form, and printed. While the tracer is on, it prints messages indicating, for example, that a traced function is just about to be entered.

There is a potential weakness here, as the value requested might not ordinarily be required at this point, so its calculation may not terminate. Also this tracing may affect the space properties of the program, so is not generally suitable for investigating space faults.

### 4.3.2 Kieburtz' proposal

Kieburtz [54] offers a proposal for the structured debugging of a functional language. This is in addition to techniques such as writing *show* functions for appropriate argument and result types in order to print out the effects of applying a suspect function to different arguments. Given that the programmer is willing to do this, the problem in the middle of a larger, more complex computation, is rather to *obtain* the values of the arguments than to display them.

The values may already be defective because of the faulty behaviour of some other function. There is also the problem of representing *functional* values.

The proposal is to use ML's exception mechanism to trace the history of values, and to enable the programmer to examine this context incrementally. Any function that may raise or propagate an exception is given an *exception continuation* as an extra argument. This will be applied to any exception that arises to produce the result.

He gives as an example a putative exception generated by arithmetic overflow. This, however, is a reminder of a comment by Hall and O'Donnell [64] that error values tend to be oriented towards handling exceptional numeric conditions and are less useful in other circumstances.

### 4.3.3 Instrumentation of the SML-NJ compiler

Tolmach and Appel [87] describe a system that uses automatic instrumentation of the user's code. They note that programmers will "instrument" their code to print out values, or trace the flow of control, when attempting to locate an error. The key idea is to insert such instrumentation automatically wherever an identifier is bound (to report its value), and wherever a function is called (to report the caller and the callee). The debugger is implemented in ML as an extension to the SML-NJ compiler. Since the instrumentation is part of the code, debugging information is not distorted by the compiler's code transformations, and there is no need to attempt to map machine code back to the original source code. The whole approach is motivated by the SML-NJ's implementation of the `callcc` primitive [6]. Information available from the debugger is only generated on request, and this minimises the overhead on performance that it causes.

Potential breakpoint locations are called *events*. These occur at each value declaration, at the top of each function, at the top of each case branch, and prior to each function call. These locations are also convenient points at which to collect the values of bound variables. The debugger maintains a counter which is incremented whenever an event takes place. The value of this counter is referred to as the *current time*.

The debugger supports reverse execution, using a primitive routine `gotoTime`. A series of state checkpoints is maintained to facilitate time-travel within the computation.

The user is allowed to set breakpoints at particular source program locations, or at particular times in the program's execution history. Tolmach and Appel also plan to have their system support modification of store values, as do O'Donnell and Hall [64] (see below).

### 4.3.4 A snapshot tool for fly

Fly [89] is a programming environment based on an eager SECD machine. It supports a purely functional, higher-order, strict language.

The debugging tool that it provides logs the application of suspect functions, showing the arguments, and optionally the results, of each application. The eager evaluation strategy means that these arguments and intermediate results are fully evaluated, so their value can be directly represented. User defined and primitive functional values are represented by their names. *Error values* as proposed by Mycroft [61] (see Section 4.4.1) are used to represent undefined results.

In the case of a non-terminating computation, it is necessary to interrupt the process before displaying a trace. The trace represents the computation up to the interrupt. See, for example, the (abbreviated) trace generated by an interrupted infinite list of ones in Figure 4.1.

```
fly> Define ones() → 1 : ones().
[ones]
fly> ones().
^C
interrupt
fly> Trace.
ones() →
      1 : ones() →
          1 : ones() →
              1 : ones() →
                  1 : ones() →
                       1 : ones() →
                           1 : ones() →
```

Figure 4.1: Tracing in `fly`.

As in the terminating case, the snapshot tool performs a *separate* computation on the state of the machine. The result is pretty-printed to give a source-level snapshot of the interrupted computation.

### 4.3.5 A snapshot tool for glide

In the same paper [89], Toyn and Runciman also describe a snapshot tool for an environment based on *lazy* combinator graph reduction: glide. Both environments are described in more detail in Toyn's thesis [88].

Some of the problems faced by the snapshot tool for the environment for a lazy language are the same as those for fly. In both cases the intention is to be able to offer finite source-level textual representations of an interrupted functional computation. Both systems need to be able to show: primitive functions; data values; identifiers; and derivations. See the `ones` example in the glide version in Figure 4.2.

```
glide> Define ones → 1 : ones
glide> ones
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
 1 , 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1^C , 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1interrupt
glide> Trace
 ones ->
        (1: ones
)
```

Figure 4.2: Tracing in `glide`.

The snapshot tool in the lazy environment, however, has the additional problem that intermediate results may not be fully evaluated. The solution is to display the text of an expression representing them. Here is another example:

```
glide> Define gen f x → x : gen f (f x)
glide> gen ((+) 1) 0
^C[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13interrupt
glide> Trace
(14: gen ((+) 1) 15)
```

Figure 4.3: Representing a partially evaluated expression.

## 4.4 Debugging without side effects

An alternative approach to debugging *exploits* the constraints imposed by the applicative style rather than trying to circumvent them. It does not depend on side effects to show what is going on in the evaluation [41]. One such approach was first suggested by Mycroft [61].

### 4.4.1 Errors as values

Mycroft's concern is to ensure that code transformations do not change the semantics of the language. He points out that: if system functions are implemented to return special error values, rather than to cause an interrupt to be generated, not only is referential transparency preserved, but a backtrace of an error is automatically built up. Such a system allows *wrong* values, *i.e.* values that are conceivable and well typed, but incorrect, and *error* values, to be treated in the same way.

`[1/0, Hd(NIL), 3+5]` should return
`[Error: Division by zero, Error: Hd of NIL, 8]`
which also demonstrates, by returning separate results for the items in the list, that this scheme is suited to parallel processing. Here is an example he gives of a backtrace:

The expression `1/0 + 3` might return:

```
Arg for PLUS not number:  error + 3
Error:  Division by zero:  1/0
```

### 4.4.2 The Daisy "debug" tool

Another way to make use of the applicative style is to get functions to return debugging information as part of their result. This is used in the work of Hall and O'Donnell [64], and most recently in the monadic style of error handling [101].

Hall and O'Donnell [42, 64, 41] discuss debugging techniques in the context of a purely functional language called Daisy [51] that uses lazy evaluation. They claim that debugging tools written in the functional language itself are effective in helping the programmer find such bugs as do occur.

Their approach is to use *shadow variables.* It is a specialisation of a technique whereby primitive functions return, in addition to their normal value, a message specifying their inputs. Every function is transformed into a debugging version that returns a pair of the return value and debugging information. Functions must be capable of receiving and propagating debugging values embedded in their inputs. When tracing information is *not* required, functions just ignore the debugging components of their inputs.

Hall and O'Donnell have automated such transformation of user defined functions. They created a system function that is built round a template which contains debugging code and user code place holders. This function replaces the place holders with the user's code, returning a new function which the user may name and apply to interesting arguments.

They give the transformation of the definition of the factorial function `fact`, which incorporates messages such as " `fact receives ... returns ...`", to include the relevant intermediate results when the message is output. This incorporates, for example, the transformed definition of the primitive `mpy` (multiply) that will return a pair consisting of the two arguments multiplied together: `result`, together with the debugging message giving the inputs to `mpy` and the result:

```
mult = \ [x y] ·
     let result = mpy :  [x y]
     in [result !
         "mult receives" x y "returns" result]
```

With the definition of fact modified in a similar way, the output of fact 4 is:

```
[[[[[[[[[] [fact receives 4]] [fact receives 3]] [fact receives 2
]] [fact receives 1]] fact receives 0 returns 1] [mult receives
1 1 returns 1] fact returns 1] [mult receives 2 1 returns 2] fact
returns 2] [mult receives 3 2 returns 6] fact returns 6] [mult
receives 4 6 returns 24] fact returns 24]
```

Full tracing produces too much output to be conveniently useful. Hall and O'Donnell's solution to this is to use an interactive debugging package. A program to be debugged is modified to include input and output streams. As Daisy uses an interpreter, the source code is easily available to the debugger. The debugging package goes through the original source program responding to the user's enquiries. Controversially, the user is allowed to change the value of a variable, for example to see whether functions applied after a given point return a correct result given the correct input. It is also at any point possible for the user to ask to see a listing of all the bound variables in scope.

### 4.4.3 Kishon

Kishon [55] presents a *monitoring semantics* to capture the *monitoring activity* found not only in Kieburtz' proposal, and Hall and O'Donnell's work, but all kinds of debuggers, profilers, tracers and monitoring daemons. It is an extension to a language's standard denotational semantics, parametrised with respect to the specifications of the monitoring. Not only can this monitoring activity be formally described, but the semantics, he claims, can be used as a *practical* basis for building effective monitors.

He points out the advantage of enabling programmers to write their own monitors without fear of changing program behaviour. While developing a program the emphasis is on getting the program to behave as it should. Using a formalised extension to the standard semantics of the language during that phase lets the development proceed in a more structured way. By specialising a monitoring semantics with respect to a source program, an *instrumented* program is created, in which code to perform monitoring actions has been automatically embedded.

Monitor semantics consists of a language (monitor syntax) to specify monitoring operations, monitor domains as value spaces in monitoring semantics, and monitoring functions to map a language's abstract syntax annotated with monitor syntax to "monitoring meaning" drawn from semantic and monitor domains.

## 4.5 Purpose built environments

Programming environments for declarative languages, with monitoring facilities, have tended to be geared more towards pedagogic than practical applications, because it is easier to display clearly the evaluation of a small example than that of a large and complex program.

A precedent for the monitoring of realistic examples, and one that can be adapted for the use of the beginning student or the advanced programmer, is TPM, the Transparent Prolog Machine [29]. Although this does not involve a *functional* language, it has exemplary features and some of the principles it embodies could usefully be applied in the context of a functional language.

### 4.5.1 The Transparent Prolog Machine

This is an execution monitor and graphical debugger for Prolog. Some of the insights from TPM are relevant here:

- "It is possible to display an execution space involving thousands of nodes on today's graphics workstations."

That this has been shown to be both possible and effective is encouraging to the developer of a graphical debugger for functional programming.

- "When a Prolog programmer is debugging a program which he or she has personally been developing over a period of weeks or months, an overall graphical view of the execution space of that program is highly meaningful to that programmer because it conveys its own gestalt ..."

This is illustrated with diagrams that demonstrate that parts of the program can be recognised even when labeling of the nodes is removed, thus allowing a smaller scale diagram (showing more nodes) to be meaningfully displayed. If the execution graph of a *functional* program can be represented in an analogous way, it may be that incongruous features such as unnecessary space leaks can also be identified. They could then be subject to closer investigation.

### 4.5.2 Lieberman's Zstep

Lieberman's Zstep [58] is a stepper for Lisp designed to facilitate locating the code responsible for a bug, as Lieberman notes that this identification of the relevant code is often the main debugging task. Zstep integrates an editor with a stepper, and when a function is invoked, its definition is retrieved as a text file and displayed in the editor's window. During the stepping evaluation, Zstep visually *replaces* an expression, or sub-expression, by its value "conforming to an intuitive model of evaluation as a substitution process". As it is usually not known whether a particular evaluation needs to be examined more closely until after a result has been obtained, Zstep allows the user to delay the decision until then. Lieberman suggests an analogy of checking alibis against fact in a criminal investigation — if the result of evaluating a sub-expression is not as expected, this evaluation deserves closer inspection. In order to help programmers locate a bug, the system allows them to have an overview of a process which can subsequently be examined in more and more detail as required. This "zooming in" will also be seen in the work of Taylor [84].

Menus appear under two conditions:

- Just *before* evaluating an expression: Do you want the details?
- Just *after* returning a value from the evaluation of an expression: Continue stepping or step back?

Zstep uses error objects to handle exceptional conditions. If evaluating some code causes an error, Zstep substitutes the error message for the code that caused the error, where normally it would substitute the value. Error objects are propagated: a function applied to one yields an error object with the same error message.

He gives an example of this: having defined the function FACT as:

```
(DEFUN FACT (N)
        (COND ((ZEROP N)
         1)
         ((TIMES N
                  (FACT (N - 1)))))
```

Figure 4.4: Definition of FACT in Zstep.

he attempts to apply FACT to the *string* ``FOO''. This causes the error message shown in Figure 4.5.

```
''The argument given to the ZEROP function,
''FOO'', was not a number''.
```

Figure 4.5: Error message in Zstep.

Stepping back allows the user to see where this message arose in the context of the original code. One window displays the original definition while another shows the definition with the `N` replaced by `''FOO''.` and `(ZEROP N)` replaced by the error message. Another example locates a missing parenthesis with a message:

`''The function CAR was called with too many arguments'',`

again with the relevant portion of the source code highlighted.

### 4.5.3 Nilsson and Fritzson

Nilsson and Fritzson [62] describe an "algorithmic debugger". The user is allowed to concentrate on the declarative aspects of a program's semantics, without needing to consider the order in which computations take place. As this order is not easily predictable, it is an advantage not to have to take it into account. The user need only ask himself: "Does this function applied to these results yield the correct result?".

The debugger first executes the program, and builds an execution trace tree. It then searches for the bug by traversing this execution tree in a preorder manner. At each node the debugger interacts with the user by asking whether or not the behaviour of the procedure invocation corresponding to the node is correct. Where the arguments to the function would be partially evaluated expressions, the system uses a process of *strictification*, looking forward to their fully evaluated forms where possible.

Figure 4.6 shows an example from the paper of the user interaction generated by working down the execution tree of an erroneous `sort` program. The user can answer "yes", "no", or "maybe". The system remembers the answers so that, unless the answer was "maybe", the same question is not asked twice.

There are problems with the system though: answers to an average of 50 or 60 questions are needed to find a bug even in a "toy" program. Building the entire execution tree causes a large time and space overhead; and subtle problems arise in the implementation of strictification.

Nevertheless the technique could be used with some larger programs if they were suitably modularised, and if the user were willing to isolate sections for scrutiny.

```
sort(in:list=[2,1,3], out:sort[3,1])?
no
sort(in:list=[1,3], out:sort[3,1])?
no
sort(in:list=[3], out:sort[3])?
yes
insert(in:elem=1, in:list=[3], out:insert=[3,1])?
no
insert(in:elem=1, in:list=[], out:insert=[1])?
yes
A bug has been located inside the body of the function insert
```

Figure 4.6: Nilson and Fritzon's debugger in action.

### 4.5.4 Kamin's Centaur

The next system to be discussed, Kamin's Centaur [53], has the more usual pattern of the user initiating the debugging interaction.

Centaur is a generic interactive programming environment. It works with abstract syntax trees (ASTs) that are supported by a *Virtual Tree Processor*. There are tools to: describe a language's concrete syntax, to translate concrete syntax trees into ASTs, and to *pretty-print* the ASTs as programs and traces.

One language implemented is a minimal functional language with lazy semantics. As a debugging system, Centaur allows the programmer to home in on an interesting part of an *execution trace* by using a hypertext approach. This approach is used to deal with the problem of information overload that is associated with trace based debugging. Whereas following lazy evaluation step by step may be impenetrably confusing, the overall pattern of an evaluation may be simple, so that such traces may be valuable.

In Kamin's trace semantics, the value of an expression is a *history* of evaluation steps. A trace may be regarded as a tree, each node representing the evaluation of an expression, and its children the trees of its subexpressions. If the expression is the application of a closure, the node also has a child giving the trace of the body of the closure.

As the trace of the history of the evaluation may be very big, the aim is to provide a *hypertext* interface for exploring it. A user of the debugging tool may click on a value and ask it to "explain itself" getting a choice of:

- the expression, the evaluation of which produced the value;
- the environment in which the expression was evaluated (*i.e.* the bindings to the values in the $\lambda$ expression);

- the *history* of the value — meaning the sequence of function applications that led to it;

also, for closure values:

- the $\lambda$ expression contained in the closure;
- the environment contained in the closure.

Environments are displayed by opening new windows for them. Expressions are represented textually, with closures depicted as a symbol: $<< - >>$. Clicking on a closure symbol causes the lambda expression stored in the closure to be highlighted in the window that contains the source code. The closure can then be further explored:
selecting `Show closure env` from a menu, for example, causes a new window to appear which contains bindings of locally bound variables.

A major problem with Centaur is that it is not able to debug programs that enter an unproductive loop: as the program does not produce any trace, there is no value for which to request the history.

### 4.5.5 Snyder's "Lazy Debugging"

A similar reconstruction of source level debugging information, from a combinator based machine, is used by Snyder [80]. He envisages debugging as searching the *reduction-history space* of a computation. He uses the phrase "lazy debugging" to mean delaying until run time the decision as to what part of the reduction history to investigate at source level. In addition to reconstructing information that can be related to the source program, his system makes use of a history mechanism that can reverse reductions. His browser's facilities include:

- single or multiple stepping to the next or previous reduction;
- moving up or down a level in the abstract syntax tree;
- displaying an accessible variable binding or function definition.

After correctness has been established, the programmer's main concern is efficiency. Snyder mentions profiling tools as a useful first step in identifying the time and space consuming parts of a computation, and alludes to features that have been useful in providing diagnostic profiling information:

- colour coding of a visual representation of the node space: for tags and reference counts;
- colour coding of the displayed parse trees: to identify variable types, and sharing information;

- statistics on the number of reductions by category.

Snyder puts forward the idea of running the reduction in the manner of a motion picture, so that the programmer can easily detect changes in the program graph.

### 4.5.6 Taylor's Prospero

This "movie" analogy is central to the Prospero system developed by Taylor [84, 85]. Prospero is a teaching tool for students who are learning Miranda [93]. It uses simple graph rewriting in its implementation. It can evaluate Miranda programs and display the stages of evaluation to the user as a graphical display.

Taylor proposes a system of *filters* in order to focus the display on particular aspects of an evaluation. One variety of filter he calls *simple* filters. These take a representation of an expression and return a new representation, usually removing low level information from the graph. For example apply nodes may be omitted in the representation of a constructor function applied to its arguments — these arguments are then shown as direct descendants of the constructor function node. Users are allowed to combine basic filters to create their own, to enable them to view the program in a way that helps them gain insight into the reduction process, or to look for the source of a particular error.

The other sort of filter Taylor calls *temporal* filters. These change the appearance of an expression over a period of time, as opposed to the "one reduction step" lifetime of simple filters. Temporal filters involve: searches through the evaluation history for the start of a section to be observed; a *mask* that determines the appearance of the expression of interest throughout the scope of the temporal filter; and a stop condition.

Unlike O'Donnell and Hall, and Kieburtz, Taylor normally avoids allowing the user to change the direction of an evaluation as this might result in unnecessary non-termination. But in his proposed searching strategies for start conditions for a temporal filter there are options to evaluate arguments prematurely — though any such evaluation is subsequently thrown away.

The Prospero system is the one most resembling that implemented and used for the purposes of this thesis, so is of particular interest: both display graph reduction steps in source level terms; both have systems of spatial and temporal filters, though as will be seen the approach is not identical, and the terminology is different; and both allow the user to define filters appropriate for the particular computation to be observed. There are naturally differences in the interfaces, and in the implementation and target languages used. But the main differences lie in the display of the graph and in the definition of filters. Where Prospero is not concerned with crossing of arcs nor, for example, keeping the traditional display of having the function to the left and the argument on the right of an apply node, the system to be described converts the graph into a tree so that there is no crossing of arcs, and the usual left-right ordering of nodes is always possible. My system also incorporates a metalanguage for user definition of filters. This permits the conditions for the compaction of the graph, and the choice of breakpoints to be more precisely defined.

## 4.6 Profiling graph reduction

Profiling graph reduction could be seen as an extreme form of compacting the information from a reduction graph. Such pictures of the graph that are shown consist of representations, not of program nodes, but of statistical data garnered from them. Three profilers are presented: from Glasgow, York and UCL. But first there is an account of one of the earliest examples of this technique as applied to functional programming.

### 4.6.1 Hartel and Veen

The precursor to the work described below is that of Hartel and Veen [43]. They investigate the process of combinator graph reduction. Using four small and four medium-sized SASL programs as examples, they measure the size and composition of the combinator graph at intervals while the program is running. Their analysis of the graph is, however, not as detailed as that of the more recent systems. For example nodes are classed as application or constructor nodes, and not further subdivided by function or constructor *name*.

They note that all major transitions in the size of the graph can be related easily to the algorithm. In most cases the graph grows to a certain size which remains fairly constant until the final phase where it reduces to the result. Most nodes have a very short life "60% of nodes witness no more than 10 reduction steps", and on average one node is reclaimed per reduction step. 94% of the nodes, in the case of their medium sized programs, represent structure rather than data values, suggesting that further implicit coding of structure could yield savings in time and storage. For example they suggest special constructor nodes for arrays and records.

### 4.6.2 The Glasgow profiler

Peyton Jones and Sansom have been working in Glasgow on a profiler that concentrates on time and space problems [77].

They point out that a possible reason for the paucity of tools for measuring the dynamic space and time behaviour of lazy functional programs is that the program is executed in an order that is not immediately apparent from the source code. It is also not easy to relate dynamically gathered statistics to the original code.

Their solution to this is to use a concept of *cost centres*. These are labels with which the user may annotate source code expressions. During execution statistical information is gathered about the expressions being evaluated and attributed to the appropriate cost centre. This is intended to enable the programmer to identify "critical parts" of the program that account for much of the space and time used. A cost centre is determined by annotating source expressions with a *set cost centre* expression construct `scc`.

For example `scc ''foo'' (map (f x) list)` causes the evaluation of `(map (f x) list)` to be attributed to the cost centre `''foo''`, though *not* the evaluation of `x` nor `list`. If these were required to be monitored, nested cost centres could be used. Costs are only attributed to a single cost centre.

Their scheme does imply that the user needs to have a clue in advance what sections of code will be of interest, in order to identify useful cost centres to set. For each cost centre they collect aggregate information about its associated pieces of source code:

- the time spent evaluating instances of the expressions;
- the amount of memory allocated;
- the number of instances of the expressions that were evaluated.

Serial profiles can then be produced either by aggregating the information collected for each time interval, or by sampling the execution state during the interval. They point out that:

> "*any* runtime event, or heap closure property of interest, can make use of the cost centre mechanism to relate the information back to the different parts of the source"

The proposed heap profiling has similarities to that described by Runciman and Wakeling below [75].

### 4.6.3 The York profiler

The York heap profiler [75, 74] is an innovative profiling tool that emphasises the analysis of memory space. It consists of two parts: a modified Lazy ML compiler which generates profiling information as a program executes, and a display program which converts such data into profile graphs expressed in postscript. The particular facts that the prototype version focuses on are: the composition of the heap in terms of constructor nodes and closures of named functions, and the names of the functions that produced the nodes in the course of the evaluation of expressions containing their application.

The graphs suggest possible target functions for reducing space consumption, akin to the critical parts mentioned above. The example on which the system was first tried exhibited five problems that were subsequently remedied, two of them involving changes to the compiler, and three to the code of the example program. This resulted in a reduction of space used from 1.3Mb to 9Kb. The program also ran twice as fast.

The profiler has subsequently been used to analyse the translator in the LML compiler itself [74], but with whole types as contructors and whole modules as producers. Here again the execution cost was significantly reduced. This illustrates that such a tool can be used effectively, regardless of the size of the target program.

A version of the York heap profiler has now been added to the Chalmers' Lazy ML/Haskell distribution.

### 4.6.4 The UCL profiler

Yet two more profiling techniques are being developed at UCL [20, 19] by Clayman, Parrott and Clack.

The first involves the use of a *cost* function. However, this is unlike the Peyton Jones and Sansom cost centre. It writes the cost of evaluation of an expression to a special output stream, without maintaining any cumulative information. The authors are not happy with this, however, as the cost function is dependent on its context. This means that in a parallel implementation, where the order of evaluation of expressions may vary from one run of the program to another, timings returned may not be consistent. They prefer a different technique that is not affected by the properties of run time behaviour. This technique they call *lexical profiling*.

In lexical profiling function *definitions* rather than expressions are profiled: only the costs of expressions *textually* contained in a function definition are attributed to that function; and statistics are collected over a whole program run. This is like making the function definition something like a Glasgow cost centre. The data collected for a profiled function consists of:

- the space usage of the function over time;
- the time spent in its evaluation;
- the number of times it was called;

- and the number of calls it made, and to whom.

One of the authors' chief design objectives is to help the programmer to identify parts of a program which consume a disproportionate amount of resources. They demonstrate the usefulness of relating results collected during the run of a program to the source code: tail strictness is introduced into a program that uses `foldr` by giving it as argument a function that unnecessarily pattern matches on its own second (list) argument; the profile clearly shows that this errant function is being repeatedly called by another that uses `foldr` in its definition, and not merely that it is repeatedly called by `foldr` in the execution of the program.

The system does not yet cope with source code at the Haskell level, as it has been developed using intermediate level code so that names assigned by the compiler to functions created by lambda lifting and optimisation may be used directly. But the scheme looks as though it could be of real practical use in detecting the origin of space faults when it has been developed further: although the programmer can still not "see what's going on", the evidence from lexical profiling may give even more clues than, say, the York profiler, so this sounds like a potentially very useful tool.

## 4.7 Discussion

The various strands of recent work on monitoring have the common theme of observing the reduction of functional programs, but with several different aims. These include:

1. finding errors in the source code;
2. optimising execution performance;
3. illustrating what is going on for teaching purposes.

There are also aims that are outside the scope of this thesis:

4. optimising compiler performance;
5. exploring parallelism.

In relation to points 1 – 3 above, what tools would the programmer ideally like to have, and how far does existing work go to provide them?

### 4.7.1 Finding errors in the source code

In the absence of any automated assistance in searching for bugs, the programmer has various lines of attack on the problem. If there is no clue as to where the problem arises, each function of the program needs to be tested separately for accuracy of output, given correct input. The action of suspect functions may be tested by changing them to return debugging information. In a strict world this would be sufficient, and could be directly automated, though the problem of representing *functional* values needs to be resolved; in a lazy world the situation is complicated by intermediate stages of the computation involving closures rather than easily displayable values. These may be as arguments to the function, as well as resulting from its application.

The programmer does *not* normally want to know what is going on in the reduction process, but only which functions are misbehaving. A view of the process is required that will show where the error(s) occur, and perhaps suggest remedies.

Which of the existing schemes provide a solution? Kieburtz [54], Hall and O'Donnell [64] and Kishon [55] all include in their conception, implemented in some form by the last two, a trace of the computation available to the user. Tolmach and Appel [87] get the compiler to instrument the user's code. This has an advantage that transformations to the code are reflected in transformations to the debugging information, so there is no problem in relating the two. The `-T` flag in the Chalmer's LML compiler has a similar effect, and *does* allow checkpointing, but the output produced is hard to control and understand.

Lieberman's Zstep [58] is a step in the right direction but, because of the amount of trace information generated, would be exceedingly tedious to use on a large example. He suggests that setting breakpoints might help the navigation through the evaluation, but he didn't implement this. Nilsson and Fritzon's "algorithmic debugger" [62] is also along the right lines, but again is only suitable for small examples because of the number of questions the user would have to answer for a larger one. Centaur's [53] hypertext system, if used in conjunction with checkpointing, which Kamin does not do, might be a solution that could be used for larger examples — but his implementation of the system, keeping a history of evaluation steps, seems not very efficient. The inability to deal with programs that loop is a serious flaw, as such looping may be the very symptom of the bug one wants to investigate.

The glide system can helpfully show the state of a non-terminating computation, but when tracing a computation that terminates, merely indicates which clauses of the function definitions have been tried, not with the actual arguments. Finally Snyder's reduction history is yet another method of approaching the requirements, also unsuitable for large examples.

So there is as yet no satisfactory general purpose tool for finding bugs in lazy functional programs. Kishon's monitoring scheme [55] may be the way forward here. An ideal programming environment should offer the programmer facilities for sytematically creating an idiosyncratic tracing mechanism appropriate for his particular application program, perhaps by joining together multiple monitors to create new ones. This may include the possibility of stepping backwards or forwards in a computation, and the creation of breakpoints by, for example, identifying suspect functions. So we can identify some first requirements for the system to be developed:

**Requirement 1** Let the user adapt the tracing to particular applications.

**Requirement 2** Allow the user to step through the reduction.

**Requirement 3** Permit the creation of breakpoints.

### 4.7.2 Optimising execution performance

In order to optimise execution performance it is helpful to be able to identify sections of code that cause space faults. The complexity of the evaluation process, in the context of a lazy functional language and a compiler that does some transformation, is such that theorising about the program behaviour from cold is unproductive, because it is error prone.

This is where heap profiling comes in. The ability to see detailed statistics about the composition of the program graph has already proved to be effective in identifying code that is inefficient with regard to space usage [75]. It may be that the efficiency of the running program is closely linked with the implementation of the compiler. For example a local prototype of the York profiler included an option to use Wadler's suggested scheme to obviate the problem of the elements of tuples not being efficiently garbage collected [98]. Although this was only implemented for pairs, programs would run using this modified version of the compiler that would run out of space using the conventional version.

At this early stage in the development of usable implementations for lazy functional languages the programmer may need to have control of options such as this, and to use them in conjunction with the particular code being written. An alternative is to modify the code, where possible, to take account of the compiler's foibles. For example a coding trick can be used to avoid the problem with tuples mentioned above: an extra function is introduced to be given as argument the expression that reduces to the tuple; the elements of the tuple are then accessed by pattern matching rather than projection so the construction is effectively broken up. This allows programs to run that otherwise crash with "Out of heap space" messages, but does impair the readability of the code.

The success of heap profiling suggests that it should be an intrinsic part of every serious environment for lazy functional programming. It will assist in the development of pragmas for the lazy programmer, both general ones and others geared to particular implementations. The only disadvantage of heap profiling is that, although details of the reduction processes may be inferred from the views provided of the heap, there is no explicit account of what is going on. The programmer has to make informed guesses as to what is happening from the overall view. From this another requirement for the proposed system emerges:

**Requirement 4** Give detailed information about the reduction process.

### 4.7.3 Illustrating the reduction process

The problem of giving a view of the reduction process, for teaching purposes, is easier to solve, because at the stage at which students need such a teaching tool, small examples suffice. There is little information to display and a relatively small number of reduction steps.

Whether it is appropriate to display a representation of the *actual* reduction steps of the computation is another matter. It is probably best at first to present the process using a simple graph reduction model. Taylor's Prospero [84] does this, and offers filters which allow small examples to be shown. Techniques of filtering and focusing need to be further developed for such displays to be of real, practical, help in exploring larger examples. This suggests a further requirement for the proposed system:

**Requirement 5** Provide powerful techniques of filtering and focusing so that the display may be of practical use for large examples as well as small ones.

### 4.7.4 What we need now ...

Bringing together the requirements enumerated throughout this section, what we need now is a programming environment that will:

1. let the user adapt the tracing to particular applications;
2. allow the user to step through the reduction;
3. permit the creation of breakpoints;
4. give detailed information about the reduction process;
5. provide powerful techniques of filtering and focusing.

# Chapter 5

# A monitoring interpreter

## 5.1 Introduction

This chapter presents the design of a monitoring interpreter for a lazy functional language, that will fulfill the needs enumerated at the end of the previous chapter. The scale of the exercise demands something simpler than an interpreter for, say, full blown Haskell. So a language has been devised that is sufficiently sophisticated to enable the system to give convincing results, but otherwise as simple as possible. The language is for the most part a subset of Haskell, so is called `h`. The interpreter is called `hint`, as it is an **h** **int**erpreter, and because it is designed to give *hints* as to what is going on in the reduction process. The implementation of this interpreter is in Haskell, providing further evidence of the benefits and limitations of using a lazy functional language for an interactive graphical application. Techniques and algorithms involved in the implementation are described and assessed in the next chapter.

The hypothesis is that given a display of what is going on in a computation, at an appropriate level of detail, the functional programmer will be able to make use of the information to write "better" programs, *i.e.* more efficient in terms of space and time usage. Until implementations of lazy functional programming languages become really efficient, such considerations may make the difference between a program running and not running. The idea is to have a system that displays a series of program graphs that represent a computation as it proceeds: a sort of electronic animation of textbook presentation of graph reduction. These graphs may be very large. And there may be very many reduction steps. Hooking up the reduction to some pre-existing graph display package is not adequate except for small examples, as in general one has to exploit the characteristics of the particular type of graph to tailor, and compact, the display meaningfully. Showing a computation in great detail could take an inordinate length of time. Thus the focus is on *depicting graph reduction on a small screen in a reasonably short time*. Subsidiary questions arising from this are:

- How to implement the reduction?
- How to decide which reduction steps to show?
- How to simplify and compact the display without losing its original meaning?

**Outline of chapter**

The structure of the rest of this chapter is as follows:

- the nature of the language to be interpreted is described and justified; (Section 5.2)
- an account is given of the reduction process; (Section 5.3)
- problems involved in displaying graph reduction are identified, and solutions involving filters are proposed; (Section 5.4)
- a metalanguage is described for defining functions to compact the display, and to determine which reduction steps to show; (Section 5.5)
- finally an overview of the prototype system is given. (Section 5.6)

## 5.2 The `h` language

The intention is that the information provided by `hint` during the reduction of an `h` expression offer an accurate view of the reduction of an equivalent expression in Haskell by a conventional implementation. It is important to ensure that an `h` program is not misleading with respect to corresponding Haskell programs in regard to expressiveness and the reduction process to which it is submitted. Ideally `h` programs would be a subset of Haskell programs. This is not quite the case, as pattern matching in `h` is expressed slightly differently from that in Haskell: in `h` patterns may not occur in lambda abstractions and function definitions, and there are no list comprehensions. As discussed below there is a pattern matching case expression, as in Haskell, but one that always returns a function with the same arity as the constructor found *i.e.* either a constant or an expression to apply to the arguments of the constructor on which the pattern matching succeeded. This forces pattern matching to be explicit in the reduction graph which facilitates observation of the reduction. And, so long as this difference is taken into account, Haskell programs may be expressed in `h`, after suitable transformation, with only minor syntactic modification.

### 5.2.1 Functions

Definitions are equational. Functions are constructed with explicit or implicit lambda expressions at the top level. Identical lambda expressions would be legal Haskell. Function application, as in Haskell, is expressed by juxtaposition, so `f a` means: `f` applied to `a`.

Lambda expressions in `h` are only allowed at the top level, so that there is a "function name" to which they can be related. There are no local definitions of any sort to complicate this. Most compiled implementations of functional languages lift local definitions out of the outer lambda expression within which they occur. Thus by writing auxiliary global definitions in `h` we can effectively program at the supercombinator level. Binary functions may be used in infix form, as in Haskell, by enclosing the function name in grave accents. The syntax of `h` is given in Figure 5.1.

<table>
<tr><td>command</td><td>::=</td><td>def | expr</td></tr>
<tr><td>def</td><td>::=</td><td>iden var* = rhs_expr</td></tr>
<tr><td>expr</td><td>::=</td><td>const | applic | casexpr |<br>fun | data | cond | ( expr )</td></tr>
<tr><td>rhs_expr</td><td>::=</td><td>(\ var →)* (expr | rhs_expr)</td></tr>
<tr><td>fun</td><td>::=</td><td>iden | preop | applic | prim</td></tr>
<tr><td>data</td><td>::=</td><td>char | string | int | list | pair | constr</td></tr>
<tr><td>applic</td><td>::=</td><td>expr expr |<br>expr inop expr |<br>( applic )</td></tr>
<tr><td>casexpr</td><td>::=</td><td>case expr of casepr$^+$</td></tr>
<tr><td>casepr</td><td>::=</td><td>expr → expr</td></tr>
<tr><td>preop</td><td>::=</td><td>( inop )</td></tr>
<tr><td>list</td><td>::=</td><td>[ (expr (, expr)* )$^o$ ]</td></tr>
<tr><td>pair</td><td>::=</td><td>(expr , expr)</td></tr>
<tr><td>prim</td><td>::=</td><td>head | tail | null | fst | snd |<br>mod | div</td></tr>
<tr><td>cond</td><td>::=</td><td>if expr then expr else expr</td></tr>
<tr><td>inop</td><td>::=</td><td>+ | – | == | : | ++ | * | ‘iden‘<br>> | < | >= | <= | && | ||</td></tr>
</table>

Figure 5.1: Syntax of `h`.

### 5.2.2 Types

There are built in primitive types in `h` corresponding to `Int`, `Bool`, `Char`, `List` and `Pair` (and `Error`). In addition to these, users' own types may be used. There are no *user defined* types, so they might be called *user implied* types: `hint` recognises constructors by their initial capital letter. It is as though all constructors are regarded as instances of a Universal type. The system requires that all constructions be saturated, as, otherwise, it has no way of knowing the assumed arity of constructors. Indeed no user defined function is *ever* checked for arity, so the system can express functions of *variable* arity. For example see the definition of Turner's tautology checker [94] in Figure 5.2. Here the tautology `f` is of type `Bool` when `n` is `0`, `Bool` → `Bool` when `n` is `1`, `Bool` → `Bool` → `Bool` when `n` is `2`, and so on. The original application of `taut` requires `n` to be the number of Boolean arguments to the tautology function. Thus `n` is `2` when checking the validity of De Morgan's law.

```
hi> taut n f =
      case n of
        0 -> f
        _ -> taut (n-1) (f True) && taut (n-1) (f False)
hi> taut has been defined
hi> demorgan p q = not (p && q) == not p || not q
hi> demorgan has been defined
hi> taut 2 demorgan
hi> True
```

Figure 5.2: Turner's tautology checker.

### Type checking

There is no static type checking in `h`, and dynamic checking only insofar as the application of a primitive to the wrong sorts of arguments results in an error *value* that incorporates an error message. The lack of type checking permits functions of variable arity as shown above. It also, for example, admits lists of mixed type,

But how significant is this lack of type checking in relation to the intention to keep the expressiveness and reduction properties of `h` close to those of Haskell? A type system in general *constrains* what can be expressed. The "switching off" of typechecking does not radically alter the graph reduction process, yet allows the interpreter to be quicker and simpler than it might otherwise be.

### Pattern matching

As in core Haskell there is no pattern matching in `h` of the implicit kind: recognising argument patterns to choose which equation in the definition of a function to apply. Instead `h` expresses pattern matching at an intermediate level with a *case expression* which maps constructors to a function with the same arity as the constructor. This allows pattern matching to be displayed in a manner that is consistent with the rest of the reduction, and ensures that the display is not complicated by a need to include argument variables. The use of such a case statement is illustrated in Figure 5.3.

h version:
```
take n list = case list of
                 []     -> []
                 (h:t)  -> take' n
take' n h t = case n of
                 0      -> []
                 _      -> h : take (n-1) t
```
Haskell version:
```
take _ []     = []
take 0 _      = []
take n (h:t)  = h : take (n-1) t
```

Figure 5.3: The pattern matching case statement.

In the definition of `take` the case expression pattern matches on the `list`: if this is null, an empty list is returned; if this is a list with head `h` and tail `t`, the function `take' n` is returned, to be applied to `h` and `t`. The case expression in the definition of `take'` pattern matches on `n`, in the case of `n == 0` returning `[]`, otherwise returning an expression that involves another call to `take`.

Sometimes when tracing a computation a surprising amount of reduction is seen to be necessary before pattern matching can be resolved, *e.g.* the insertion sort example, page 132 in Chapter 7. Here the intuition of "inserting the head of a list into the sorted tail of the list" is shown, in the display, to involve a cascade of *case* expressions, representing cumulative unresolved pattern matching at every element of the list, that can only be disentangled once the empty list at the end is reached.

### 5.2.3 Primitives

A limited selection of primitive functions and operators is provided. These are required to be saturated for simplicity of implementation, though their partial application may be achieved by creating a user defined function with the same effect, *e.g.* `plus x y = x + y`. Infix operators may be used in prefix form if parenthesised as in Haskell. There is a conditional construct, `if...then...else`, and constructors and projector functions for lists and pairs.

### 5.2.4 Lambda lifting

As noted in Section 5.2.1, local definitions in Haskell are replaced by named auxiliary functions in h. This creates two problems. One is that the process of lambda lifting by hand is *tedious* and *error prone*. This could be overcome by automating it, perhaps retrieving Haskell code from a Haskell compiler after the lambda lifting phase, while ensuring that supercombinators are tagged with a name derived from their function of origin. The other problem is a danger of *loss of laziness*, as supercombinators that have been derived by lambda lifting may have built in to them values of arguments to the original function. This can only be emulated by defining a specialised version of the function in h. Again, the problem may be overcome by properly automating the lambda lifting.

As an illustration, Figure 5.4 shows a modified version of an example function from Stoye's thesis [82], also discussed in [67]. The Haskell definition of f has two possible h counterparts:

Haskell version:
```
f x = g
      where
      g 0 = 0
      g n = ef x + g (n - 1)
```

h version *without* sharing:
```
nsf  x n =  case n of
                 0 -> 0
                 _ -> nsf' x (n - 1)
nsf' x n = ef x + nsf x n
```

h version *with* sharing:
```
sf      x n = case n of
               0 -> 0
               _ -> sf' (ef x) n
sf'  efx n = case n of
               0 -> 0
               _ -> sf'' efx n
sf'' efx n = efx + sf' efx (n - 1)
```

Figure 5.4: The danger of losing sharing when lambda lifting.

nsf loses the sharing of ef x, and sf maintains it. Here ef represents an *expensive function* to emphasise the requirement that its repeated application is to be avoided, if possible. In the sharing version ef x is only calculated once, and all other instances are shared. Its value becomes incorporated in a partial application of sf' in a similar way to the Haskell version where `ef x` is incorporated into `g`. The version without sharing results in `ef x` being calculated `n` times.

## 5.3 The reduction model

The simplest way of deriving a graph to display from the state of the reduction is to use graph reduction by template instantiation in the implementation. So the reduction of `h` expressions is implemented using graph reduction, and the program graph is available for display at any point in the process. The Haskell functions that implement the reduction are a declarative expression of the target language's reduction rules. These are given in Appendix B. An account of the implementation of the reduction is given in Chapter 6.

### 5.3.1 Graph reduction

Graph reduction is a form of expression rewriting that includes *sharing* by means of pointers [102]. The process may be visualised by showing a succession of graphs, each representing an intermediate stage in the reduction of the expression (Figure 5.5). The vertices of

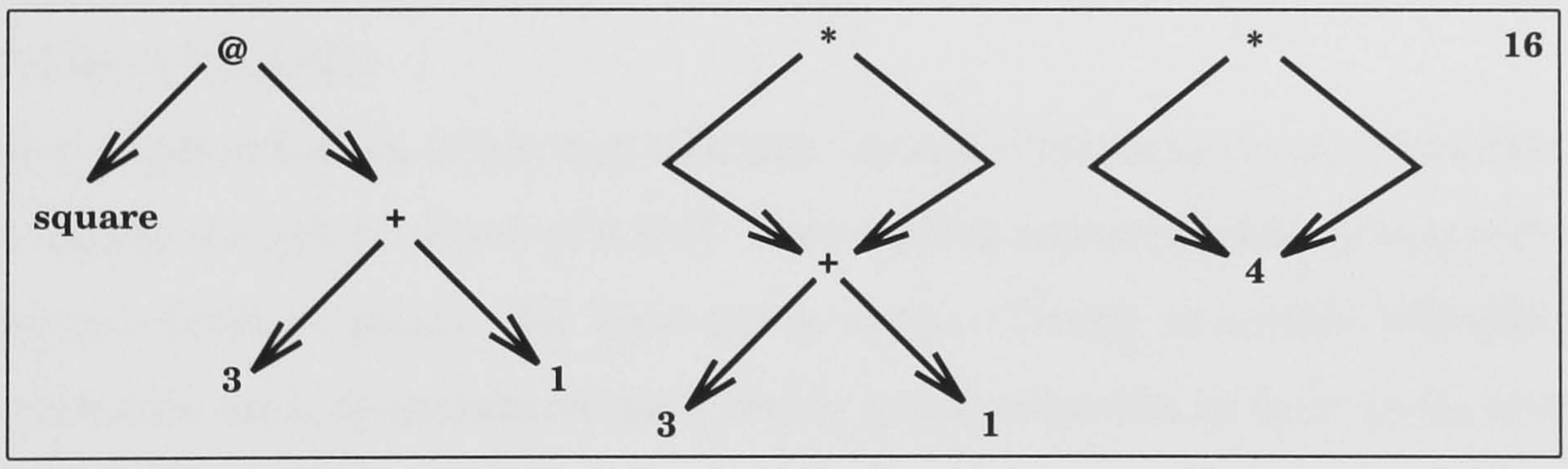



Figure 5.5: The reduction of `square (3 + 1)`.

the graph are elements of the intermediate expression: values, built-in functions and variable names. There may also be "apply" nodes (`@`), representing function application, that are implicit in the functional expression. The arcs of the graph build the expression's abstract syntax tree, with sharing expressed by more than one arc going to the same vertex.

### 5.3.2 Rewrite rules

The reduction process is determined by a series of *graph* rewrite rules. There is a "current node to be reduced" which is the root of the next subexpression to be reduced. This is originally the root of the main expression to be evaluated. Before an expression can be reduced it may be that some of its subexpressions need to be evaluated, either completely or to a *normal*, canonical, form. In Figure 5.5 the `3 + 1` needs to be evaluated before the `*` may be applied.

### 5.3.3 Order of evaluation

The order of evaluation of sub-expressions in a functional program does not affect the final result, though it may affect whether the reduction terminates or not. Sub-expressions may even be evaluated in parallel, but this is not our concern here. In a sequential evaluation there are two principal orders possible: *outermost* and *innermost*. Outermost reduction, also called *normal order* or *lazy* evaluation, only reduces a sub-expression so far as its result is needed to reduce the main expression. Innermost reduction, also called *applicative order* and *eager* evaluation, causes the arguments to a function to be fully evaluated *before* the function can be applied.

Our main interest here is in exploring *lazy* sequential reduction. The order of evaluation in such an implementation is deterministic, but not necessarily intuitive. Our aim is to enable the user of our system to write "better" programs — *i.e.* demanding fewer resources for their execution — through understanding what is going on in the reduction process in terms of the constitution of the graph.

Naïve graph reduction is not very efficient. Actual implementations of functional languages usually use optimisations of it [67]. The resulting reduction process may still be represented as a series of graphs, but these are no longer directly associable with the source code. However, such optimisations have simple graph reduction as their basis, so observing simple graph reduction is potentially useful in understanding what is going on in more sophisticated schemes of reduction.

An important question relating to the display of the reduction is the *level* at which this should take place. The aim is to be able to relate what is going on to the user's source code in more detail than can be obtained from something like heap profiling, yet not to overwhelm the user with *too much* detail. The raw use of the program graph may well provide too much detail for anything larger than toy examples. However it is a clear way of presenting *sharing*, and offers a well articulated framework from which, as will be shown, a compacted display may be produced.

As an example of the display of a graph with no sharing, Figure 5.6 shows three stages in the evaluation of the expression `foldr plus 0 [1,2,3,4]`. This suggests the usefulness of a system like `hint` for teaching. The definition of `foldr` is given in Figure 5.7.

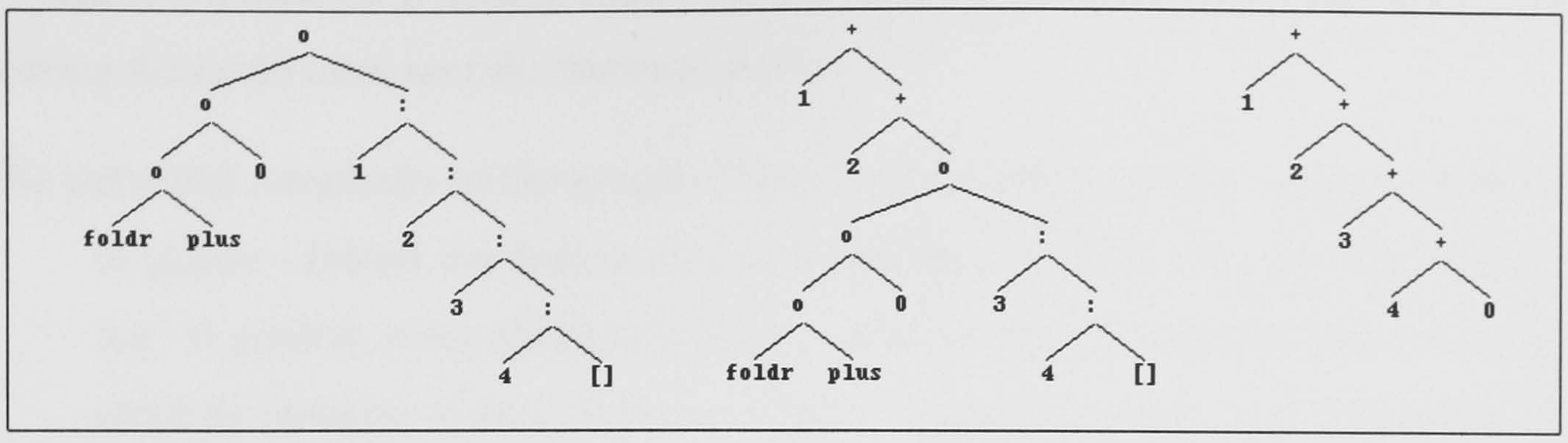



Figure 5.6: Three stages in the evaluation of `foldr plus 0 [1,2,3,4]`.

```
foldr _ z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Figure 5.7: (Haskell) Definition of `foldr`.

The steps from the display of the reduction confirm the intuition of the cons `(:)` nodes of the list being replaced by the functional argument `f`, and the terminal `[]` being replaced by `z`. The figure shows the expression before any further reduction takes place, an intermediate step — where some of the `(:)` cells have been replaced by $+$, and finally the graph just before the elements of the list become added together. (Here "Apply" nodes are shown as o, rather that `@`: as discussed in Section 7.5 this is not quite satisfactory either.)

## 5.4 Visual representation of graph reduction

The idea is to display the program graph at each stage in the reduction *i.e.* after every change engendered by a rewrite rule, or at less frequent intervals on request.

### 5.4.1 Problems in displaying the reduction

The reduction mechanism involves the application of a successor function for reduction states. Some "steps" do not involve a change to the program graph. For example: steps that push another node onto a stack of nodes to be reduced. Such changes are not intended to be displayed to the user who is observing the graph, though it would be possible to extend an implementation to show the more detailed mechanics of the reduction if this were required. From the point of view of the observer, however, a *step* is a change in the reduction state that also involves a change in the program graph. This will include any pattern matching reductions associated with the case expression.

There is a need for a display algorithm. But in addition to the general question of displaying the graph three specific problems arise:

**The potential *complexity* of the graphs** There is no guarantee that the program graph will be planar – indeed, the features of a lazy language: sharing, recursion, and "knot tying" in general, make planarity unlikely; so the display of the graph may be complicated by crossing of arcs, or by potentially long and unwieldy arcs if maximal planarity is attempted. A proposed solution to this is the creation of *graph-trees* (§ 5.4.2).

**The potential *size* of the graphs** The program graph is a detailed and low-level structure. It will be very large in all but trivial examples. Two solutions are proposed for this: one is to use *browsing*, with a miniature version of the graph as a map (§ 5.4.3); the other is to compact the graph by regarding certain connected patches of graph each to be *one* cluster in a graph of clusters. We refer to this as *spatial filtering* (§ 5.4.4).

**The potential *number* of graphs to show** The problem of there being too many reduction steps may be resolved by regarding the sequence of program graphs itself as a graph. (If alternative reduction paths were allowed, for example in a system that offered a "strict" option, it might be more than a linear sequence.) A similar scheme to the filtering of individual program graphs may be used to compact this graph of graphs, collapsing a whole chain of steps into one. We refer to this as *temporal filtering* (§ 5.4.5).

### 5.4.2 Overcoming complexity: Graph-trees

One way of simplifying the display is to avoid any crossing of arcs. Rather than trying to display every arc in the graph, display a spanning tree enhanced with *display leaves* to represent arcs that would otherwise not be shown. Display leaves are labeled with a reference to the vertex to which they represent an arc. The resulting tree is a graph that is homomorphic to the original one, but the problem of *graph* display is now limited to that of *tree* display. The special kind of tree being displayed is referred to as a *graph-tree* (Figure 5.8). The shared + node is now represented by its *instantiation* (on the right) labeled with a display reference: `{0}`, and by a display leaf (on the left) labeled *only* with the display reference.

This might seem a rather drastic solution. In this example we already had planarity, and there will be non-planar graphs where planarity might be achieved much more simply, without the need to convert the graph into a tree. Figure 5.10 shows an example of this. The first display is that provided by the prototype system from the definition of `fib` shown in Figure 5.9. Numbers in curly brackets are display references. The second display illustrates a

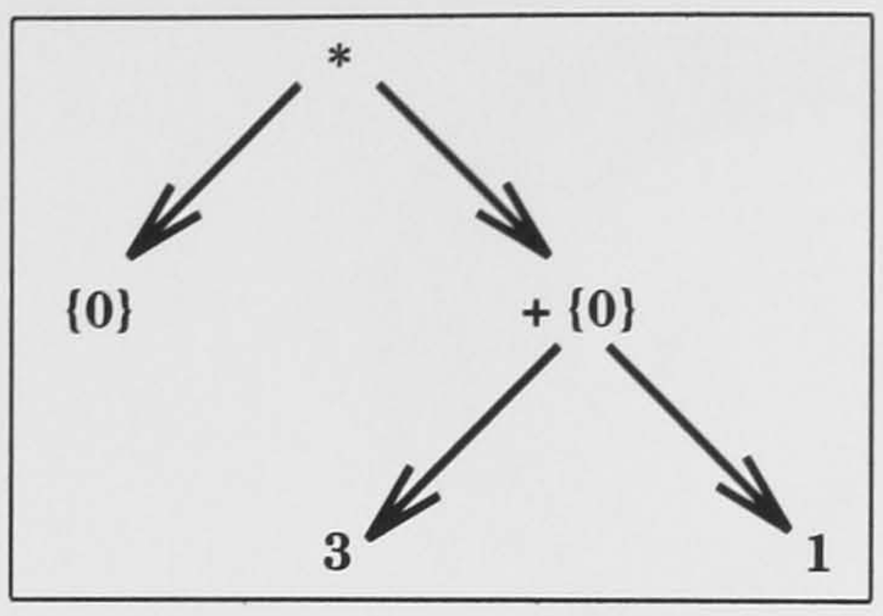



Figure 5.8: `square (3 + 1)` as a graph-tree.

corresponding planar graph. Although the arcs do not cross it is not easy to see at a glance where each leads. With the use of graph-trees there is a danger of replacing the problem of deciphering a display complicated by crossing of arcs by the problem of disentangling *display references*: in order to identify a display leaf one has to find its instantiation by matching some visual label. However the graph-tree:

- offers a simple and consistent technique;
- proves convenient when it comes to browsing and compacting the display;
- may be a useful intermediate representation from which to derive a cyclic yet still planar graph: joining display leaves to the vertex to which they refer, so long as this does not involve crossing an existing arc. The dual technique only breaks an arc if it crosses another — but lacks the advantage of intermediate representation as a tree which is more conveniently subjected to filtering.

```
fib = fib' 1 1
fib' n1 n2 acc =
     if (acc == 0) then n1 else fib' n2 (n1 + n2) (acc - 1)
```

Figure 5.9: Definition of `fib`.

### 5.4.3 Overcoming the problem of size I: Browsing

The problem of size, compounded by the addition of display leaves, may be simply resolved in two ways: by reducing the scale of the display, or by only showing part of it. But these both introduce further difficulties. Reducing the scale makes the labels harder to read; and showing only part of the display may cause the viewer to become disoriented in relation to the graph as a whole.

However the two solutions may be effectively combined by showing a *minigraph* scaled to fit exactly onto a small window, and using this as a map for browsing, as advocated by

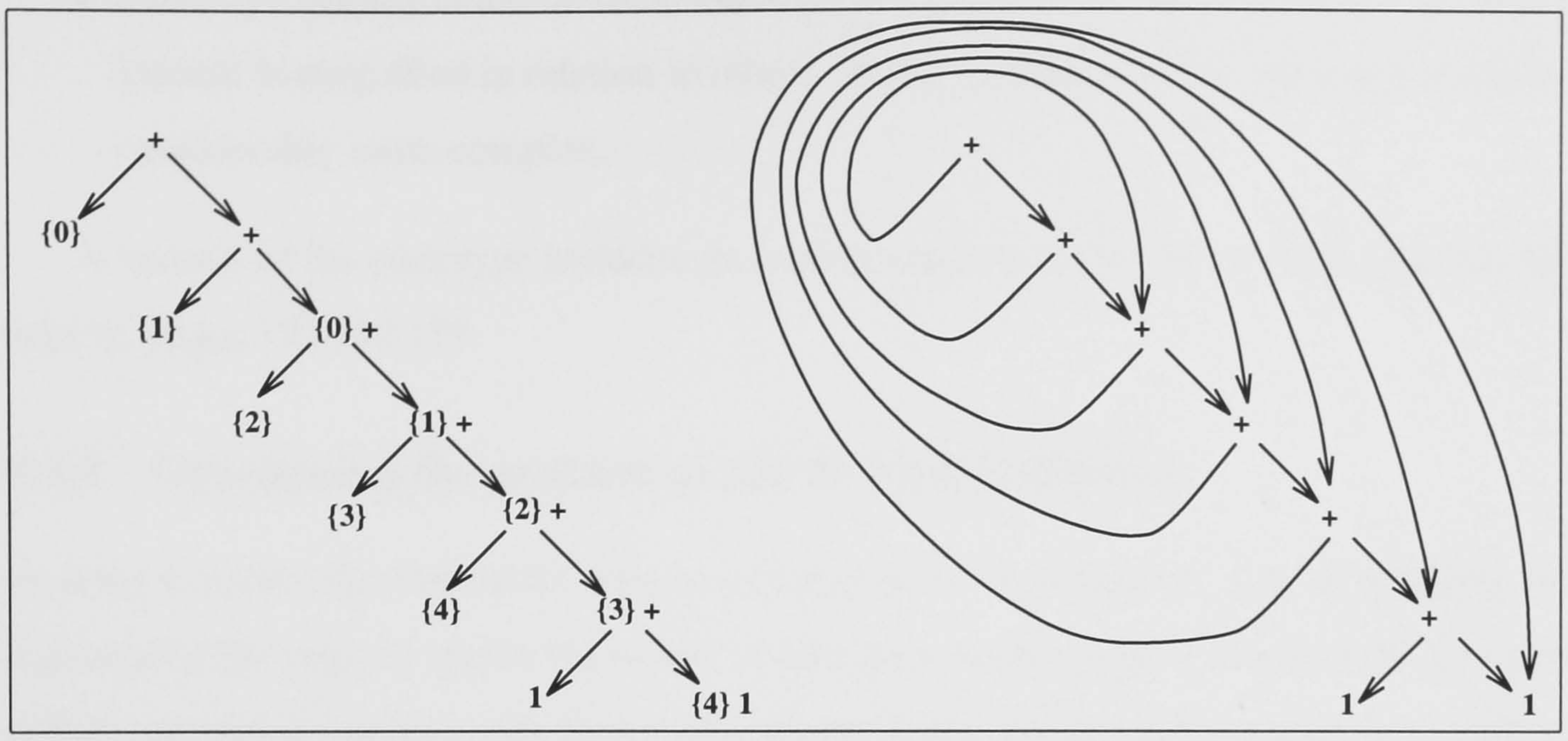



Figure 5.10: Two possible displays of `fib 7`.

Beard and Walker [11]. This has the advantage that all the structure is available to scrutiny if required, yet distinctive patterns within this, possibly hidden by the complexity of the full scale graph, may be revealed in the minigraph. The minigraph, which is a graph-tree, has the shape it would have if labels were present, for concordance with the main display, but no labels are shown. The main display is in a larger window, but on a fixed scale, so the graph-tree may have to be pruned. Arcs to vertices off the display are truncated to form *stubs*. These features are all illustrated in the `isort` graph on page 132.

### Possibilities for browsing

There are several possibilities for browsing:

- browsing may be governed by a click in the main display area, the coordinates of the click determining the new display root. The user may be offered *every* displayed node as a potential root, or be confined to moving either to the display parent of the root of the display or to one of the stubs;
- the display may be moved up, down, right or left, by a click in a control panel, a click in the region of the display to which movement is required, or even by keyboard commands;
- browsing may be governed by the minigraph display area – an outline of the main display is shown in the minigraph window, and this may be changed by a mouse click to another region of the graph-tree;

- a fish-eye display could be used where *all* the graph is on the display, but an area of interest is magnified in relation to others. In this case the scaling functions would be considerably more complex.

A version of the prototype includes an implementation of the first of these. See the figures on pages 132 and 133.

### 5.4.4 Overcoming the problem of size II: Spatial filtering

In order to reduce the number of vertices in the graph to be displayed, without violating the meaning of the original graph, the notion of a *homosemantic* graph is introduced. The idea is that a cluster of vertices with their interconnecting arcs becomes **one** vertex in a graph of clusters. This vertex inherits all the arcs from the vertices it incorporates that connect with the rest of the graph. The value of the new vertex is the piece of graph that it represents. The label for the cluster may reflect any aspect of the part of the graph that it symbolises. This technique of condensing the graph is referred to as *spatial filtering*. Figure 5.11 illustrates the effect of a hypothetical `PLUSINT` filter on the first stage of `square (3 + 1)`. The filter

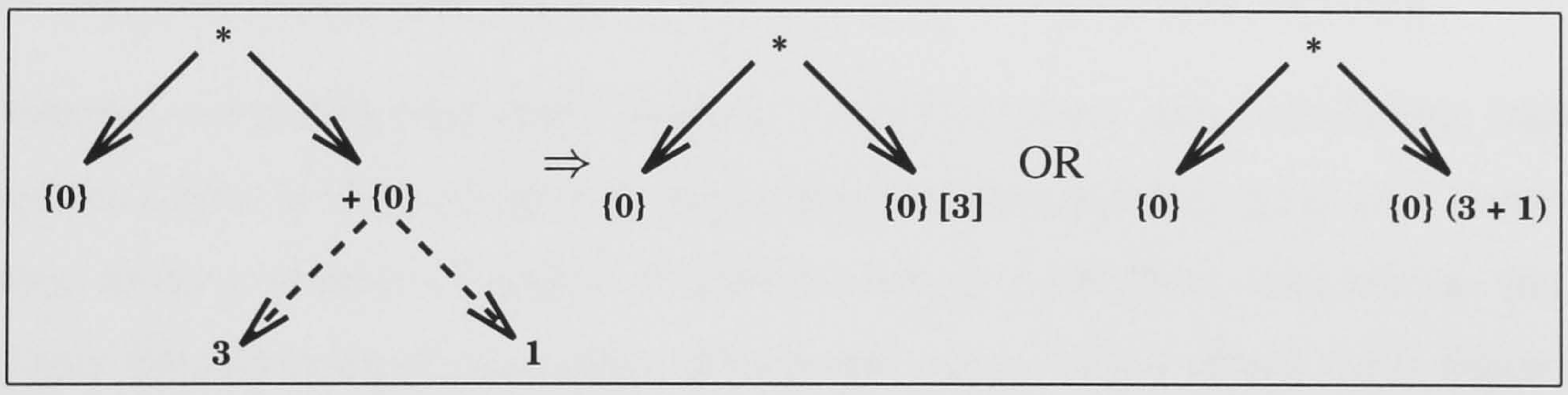



Figure 5.11: Subjecting a graph to a `PLUSINT` filter.

causes adjacent nodes that are + or `Integer` to be part of the same cluster. The arcs to be collapsed are indicated by dashed lines. The cluster is shared, so has a display reference, and two possible labels are shown: `[3]`, indicating the number of nodes in the cluster, and `(3 + 1)` indicating the expression that the cluster represents. Although the full structure of the original graph is **not** retained in the display, the condensed graph has the same *meaning* so long as all information relevant to the user's requirements is displayed in the cluster label.

#### The condensed graph retains meaning

The particular rule by which a graph is partitioned has created a view of the graph that regards nodes within the same cluster to be homologous, and the articulation between them to be immaterial. Information that is temporarily hidden either is not significant to the view, or *must* be available to the viewer through the cluster label. The label associated with such a spatial filter must expose every relevant detail that would otherwise be obscured. For example a filter called NOAPPLY condenses a chain of Apply nodes, together with the function to which they directly or indirectly belong. The name of the function must be retained in the label unless the view is to regard all functions as effectively identical. This is illustrated in Figure 5.12. Again dashed lines indicate arcs that will be collapsed.

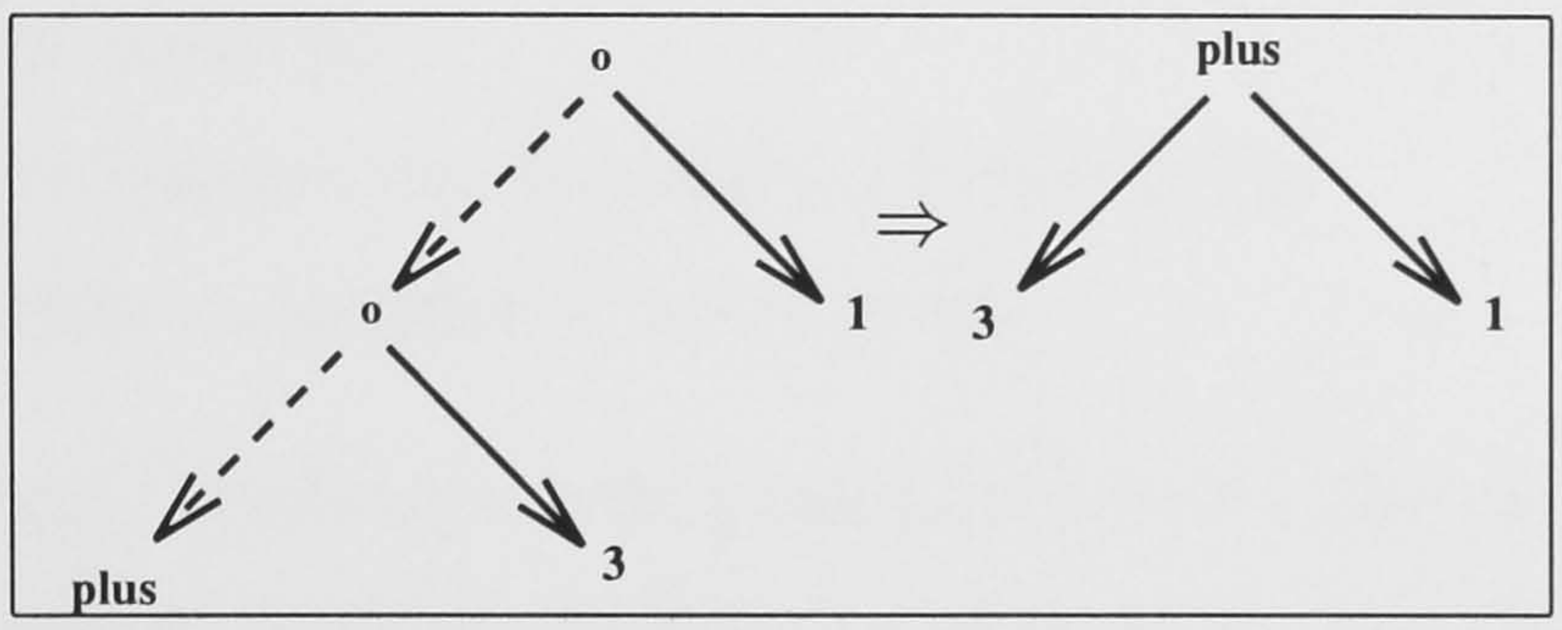



Figure 5.12: The effect of the NOAPPLY filter.

### 5.4.5 Overcoming "Too many graphs to show": Temporal filtering

The sequence of graphs may also be filtered so as only to show stages of interest. Defining the temporal filter in terms of adjacent graphs that may be regarded as equivalent rather than in terms of the properties of graphs of interest achieves a satisfying consistency: the user need only think in terms of compaction rules in both cases. This is illustrated in Figure 5.13 where, again, arcs to be collapsed are indicated by broken lines. G1, G2 etc. represent graphs in a reduction sequence. Here graphs 2,3 and 4 are coalesced. Note that the compaction does not determine what will be displayed as a representation of the collapsed chain of graphs. Usually, however, the required display will be of the first graph or the last in the collapsed

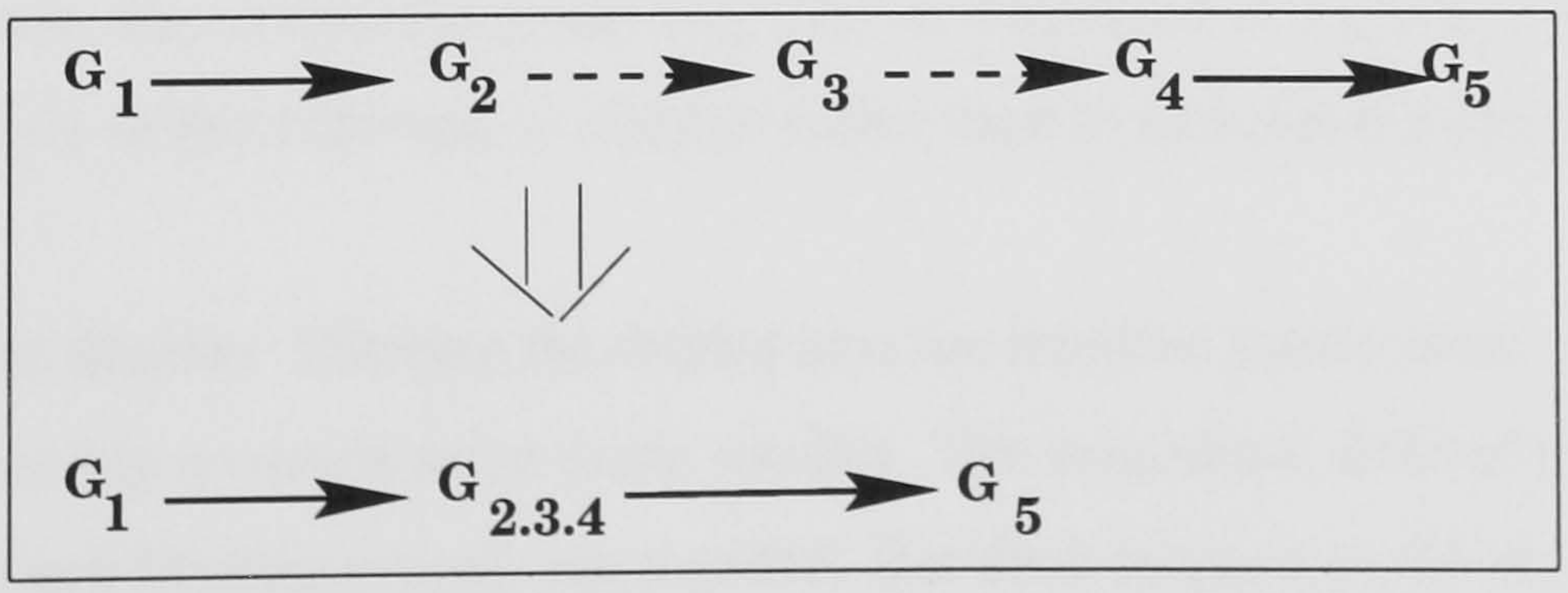



Figure 5.13: Collapsing a graph-chain: temporal filtering.

series. This is discussed further in Section 5.5.3.

The proposal is that the requisite quotient graphs be obtained in both spatial and temporal filtering by the definition of equivalence rules which determine whether two nodes are part of the same cluster, or two graphs part of the same series.

## 5.5 Defining the compaction

Given that the graph is going to be displayed as a tree, there seem to be two main options for creating a filtered graph-tree:

1. filter the graph, then convert the result to a graph-tree, or
2. convert the graph to a graph-tree, and filter that.

**1. Filtering the graph** Intuitively it is the graph itself that one wants to filter. One approach is to assume a filtering *process*: initially each node in the graph would be a single node cluster; a particular filtering rule would be determine whether or not a node is to be added to an existing cluster; such a filtering rule would be applied recursively through the graph. But problems arise. If the order of compaction affects the final structure of the graph, the definition of filters must take this into account. The order of compaction is potentially significant when dealing with any graph which is not a tree, as the treatment of a shared node may depend on the direction from which it is approached. There is the minor inconvenience of needing to keep track of nodes that have been visited during the filtering process. Also, a filtering rule may need information about ancestors to a node, that is not directly available.

An alternative approach is to regard the filtered graph as a quotient graph, and to change the emphasis of the problem from the *means* of reaching the compacted structure, to its *nature*. What is needed is the rule which says whether two nodes are part of the same equivalence class, and may thus be regarded as part of the same cluster. Given such a rule, the compacted graph may then be displayed as a graph tree with the display leaves in fact referring to clusters rather than to individual nodes or the original graph.

**2. Filtering the display** Filtering the display also has intuitive justification: it is the *display* that is too big so needs to be made smaller. The procedural difficulties encountered in the *graph* filtering process are avoided. But there is a new problem: the filter must take display leaves into account. Under what conditions will they become single node clusters rather than be merged with their parent?

Again the filtering may be described in terms of an equivalence rule. To determine whether an arc in the raw graph-tree is to be collapsed, the rule merely has to state the conditions under which two adjacent nodes are in the same cluster. This scheme is able to take ancestors into account, by reference to information collected during the creation of the graph-tree. Thus any node has access to information about the whole graph through its parent and children. Not all the problems mentioned disappear. For example display leaves have to be given special attention in the definition of the filter. But the approach has an attractive simplicity and has been chosen for the prototype.

### 5.5.1 `whiff` – a metalanguage for defining filters

Both spatial and temporal filters require a compaction rule and a labeling function in their definition. The spatial filter comprises a quotient rule which determines whether two graph tree nodes are part of the same equivalence class, *i.e.* part of the same vertex in the clustered graph, and a labeling function which extracts information from the cluster tree at each vertex and formats this into a string label. The temporal filter analogously needs an equivalence rule to decide whether adjacent graphs are to be "collapsed" together, and a function which both selects a graph to display from the collapsed series, and may also collect information from the series of graphs to show with it, as a caption.

A bank of suitable rules and functions could be built into the system, possibly with facilities for composing them to give more flexibility. Ideally, however, the user would have the power to write his or her own filters in a *metalanguage* for expressing filtering rules. Such a metalanguage has been incorporated into the prototype, called `whiff` — for **w**riting **h** **i**nterpreter **f**ilter **f**unctions.

For the system to be completely flexible and the user to have access to every aspect of the reduction process, the implementation of reduction and of filtering could be so closely tied that the user becomes an implementor. The ideal filtering provision lies between the two extremes of offering a choice of primitive filters and effectively exposing the implementation. What is needed is a simplified model of the interpreter that is consistent with the actual implementation.

### 5.5.2 Spatial filters

*Spatial compaction rules*

A spatial compaction rule determines whether two adjacent nodes are part of the same cluster, so that the arc between them will be "collapsed" in the clustered graph-tree.

```
type SpCompact = Node → Node → Bool
```

To express the rule, `whiff` offers primitives by which to refer to properties of a node. These primitives involve a view of *nodekind* that does not force the user to think in terms of datatypes that may be used in the implementation. For example, the user may wish to express: "Is it a value?", "Is it an integer?", "Is it 12?". In whiff these become: `is Val`, `is Int`, and `itis 12`. Other attributes of a node reflect the information gathered by the interpreter during the reduction. In the prototype this includes: `producer` — the name of the function the application of which caused the node to be created; and `step` — the step number of its creation, thus indirectly, by reference to the current step number, also its `age`.

However the condition under which two nodes are to be part of the same cluster may involve their context in the display. For example a rule may only apply to nodes that are not descendants of the node currently being reduced. There is, then, a need for "family relationship" functions that transfer a whiff function to the relevant other node(s): *e.g.* `parent` — the display parent; `anydescs` — at least one descendant; `child i` — the child node at position `i`. As an example: `parent (is Apply)` is `True` if the display parent is an Apply node. The `parent` function can return any appropriate whiff value, and these may be combined using `h` functions and primitives in the definition of filters.

The availability of the `parent` function suggests that the spatial compaction rule may be defined solely in terms of a condition on the child node: if reference to the display parent is needed, the `parent` function may be used. The spatial compaction rule becomes a predicate which determines whether a node is coalesced with its parent:

```
type SpCompact = Node → Bool
```

A simple example of a spatial compaction rule is the NOAPPLY filter, illustrated in Figure 5.12, which may be defined at the `hint` interface as:

```
NOAPPLY = parent (is Apply) && ownpos == 0
```

*Labeling the clustered graph tree*

The compaction rule does not fully determine the appearance of the compacted graph. For this a labeling scheme is needed. Each vertex in the compacted graph represents a graph tree. A labeling function determines how the graph tree at each vertex is presented in the display. There may be alternative labeling functions for the same compaction rule.

As with the spatial filters, labeling functions might be provided as primitive. An example would be "the leftmost node of the cluster tree", for use with the `NOAPPLY` filter. Other options could include a representation of the cluster tree as an expression, its size (number of raw graph tree nodes), the age of its root node in reduction steps, *etc.*. But here again it is preferable to offer flexibility to users to define their own labels. The prototype system uses a folding function over the cluster tree, for which the user has to provide:

**unit:** a function to apply to leaves of type: `/Node →/ info`

**join:** a function to apply at inner nodes of type: `/Node →/ [info] → info`.
The `[info]` is from the graph tree nodes below.

**display:** a function of type `/Cluster →/ info → String`,
which controls the final formatting of the cluster label

The first two functions have the *node* in question as an implicit argument that may be accessed by `whiff` primitives; the display function has the *cluster* as an implicit argument, and its own set of `whiff` primitives. A `whiff` primitive represents a function application to an implicit argument. In the functional expression it has the type of the result of this application, but there is no need for the user to refer explicitly to the "node" or "cluster" that will be the argument. The `whiff` primitives are explained further in the next chapter in the description of the implementation of `whiff` (Section 6.4).

Here is an example of a labeling function definition. It labels application clusters created by the `NOAPPLY` filter with the function names. The unit function is: `u = show`, which shows a representation of the node, the join function: `j = head`, and the formatting function: `d = id`.

The treefold arguments are associated using "keywords" `u`, `j` and `d` thus:

```
NASHOW = u show j head d id
```

The compaction rule and labeling function, which are defined separately and may be changed separately, may also be associated to create a named spatial filter thus:

```
NA = (NOAPPLY,NASHOW)
```

*Examples of spatial filtering*

To illustrate the definition, composition and visual effect of spatial filtering we take the computation of the series of prime numbers using the sieve of Eratosthenes. Figure 5.14 gives the h definition of primes.

```
primes          = sieve (from 2)
sieve    pl     = case pl of
                    (p:l) -> sieve'
sieve'   p   l  = p : sieve (pfilter p l)
pfilter  p  xl  = case xl of
                    (x:l) -> pfilter' p
pfilter' p x l  = if (x 'mod' p) == 0
                  then pfilter p l
                  else (x:pfilter p l)
from n          = n : from (n + 1)
```

Figure 5.14: An h definition of primes using the sieve of Eratosthenes.

Figure 5.15 shows the raw graph tree as the second prime number, 3, is just about to be output, and the effect of applying the NOAPPLY filter to this. The labeling function has been modified to include marking the node currently being reduced with ~~~, and representing display leaves as display references, here an integer between curly brackets. The {0} is a display reference which represents 3 in each case, as may be seen in the bottom level of the tree where the display reference is associated with its instantiation. At the top of each diagram is a modified *cons* node, displayed as --:--. This represents an *output node*, a device used in the implementation of the stepping interpreter that permits the display of the reduction of a constructor argument to be associated with the rest of the display even though the constructor itself is no longer part of the graph.

Figure 5.16 illustrates a slightly different view of the graph tree, using a spatial filter that collapses tree sections that represent *any* arithmetic expressions – not merely those consisting of + and integers, as in the PLUSINT filter of Figure 5.11. In this case it is labeled by the expression that it represents, using an "infix" function defined in h. The whiff definition of the compaction function, ARI is as follows:

```
ARI = parent AR
AR = is Mathop && alldescs (is Int || AR)
```

The labeling function shows leaf nodes unless they are display leaves, clustered sections are represented by infix expressions, and each vertex is also marked as appropriate with a display reference (sref) and whether it is the current focus of the reduction (sfocus).

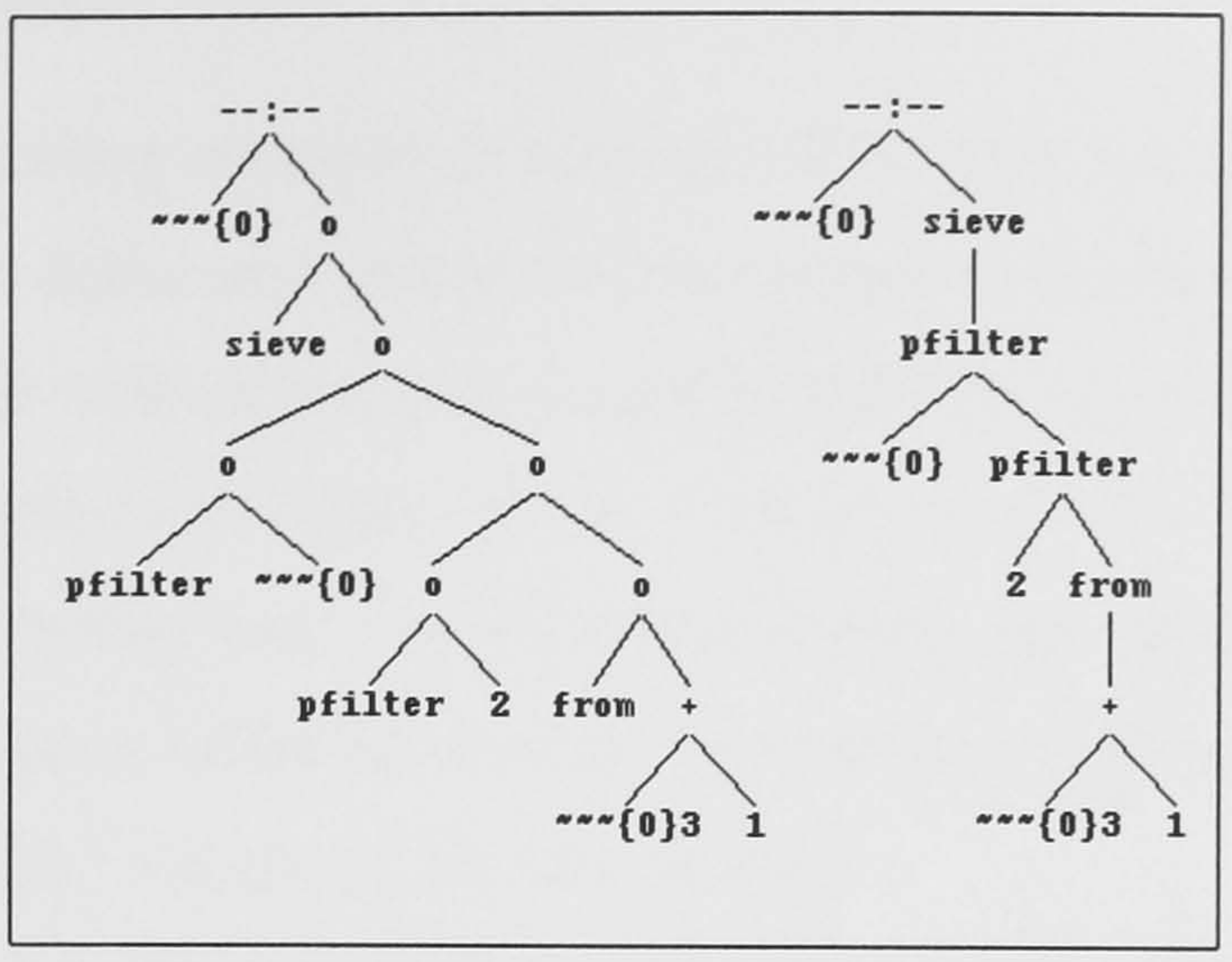



Figure 5.15: The raw graph and the effect of the NOAPPLY filter.

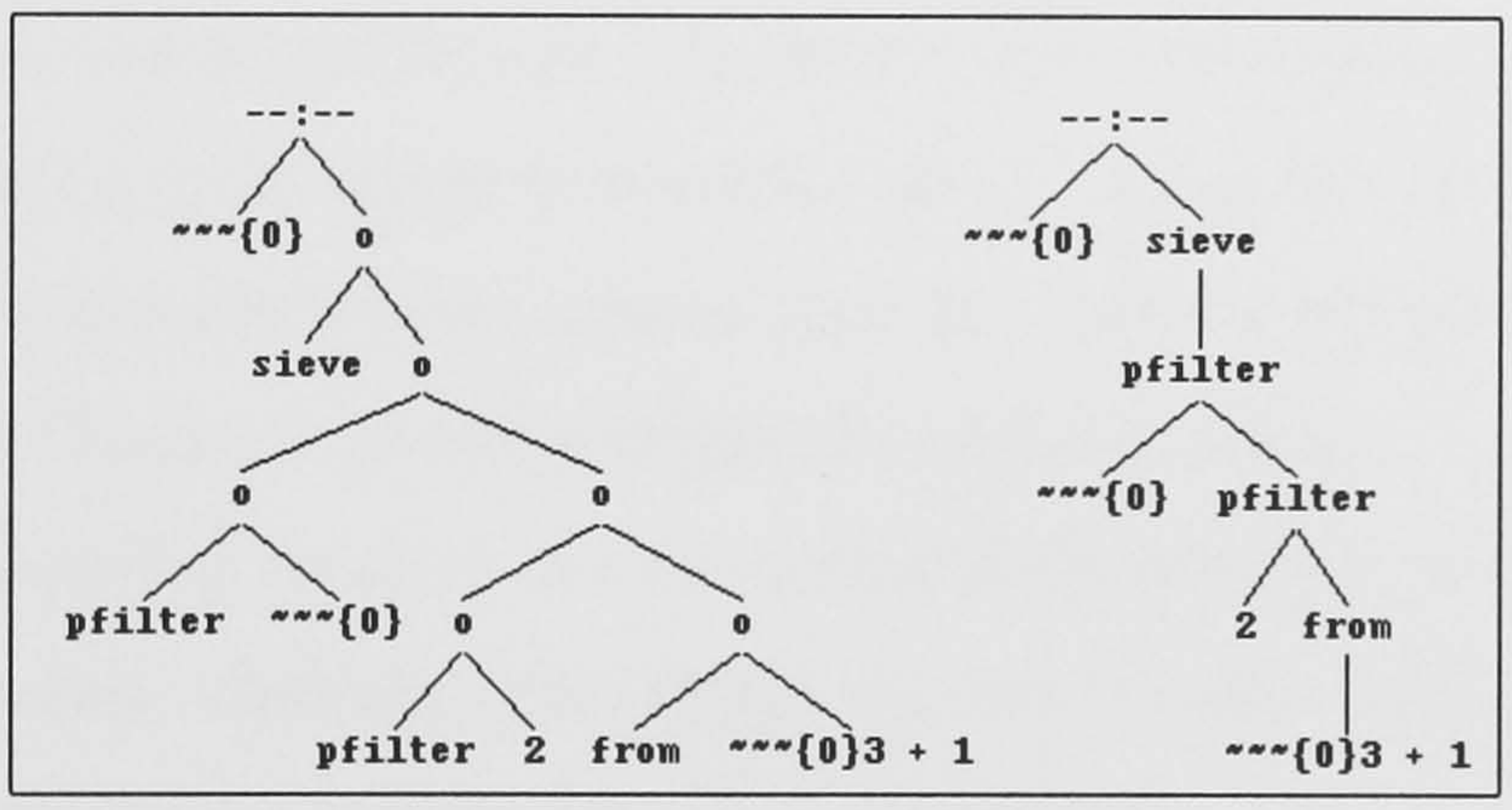



Figure 5.16: The ARITH filter, then this composed with the NOAPPLY filter.

```
ASHOW = u show j JAR d DAR
JAR lss = infix show lss
DAR ct = sfocus ++ sref ++ id ct
```

The spatial filter associates the compaction with the labeling: `ARITH = (ARI,ASHOW)`

Figure 5.16 also shows this arithmetical filter composed with the NOAPPLY filter: `NOAPAR = NOAPPLY || ARI`. This is associated with a labeling function that needs to take the nature of the cluster into account:

```
SCOMP = u show j JCOMP d DAR
JCOMP lss = if is Mathop && alldescs (is Int || AR)
              then infix show lss else head lss
```

### 5.5.3 Temporal filters

The compacting and labeling elements of temporal filters have been combined in the prototype: the user is able to define *checkpoints* which determine the graphs to show, and cumulative data are presented with each displayed graph, but there is no extra "caption" information. A checkpoint *implies* a compaction rule: a step that is not a checkpoint is implicitly coalesced with the following step. Graphs at checkpoints are displayed under the current spatial filter, and cumulative information about the reduction: the number of function applications, the "step number" and the graph size, is shown.

It is possible that some other graph than the first or the last (or both) in a series might be required, an obvious example being the largest in the series. Other information such as the number of reduction steps between particular checkpoints could also be of interest, though this may be obtained indirectly through reference to the step number at each stage. But it is quite possible that the most useful information about a subsection of the reduction might be provided by some complementary scheme such as a version of heap profiling, which, as will be discussed in Chapter 8, is one possible line of future work.

Both the checkpointing function and the spatial filter, or either of its elements, may be changed at a checkpoint. Amongst other things this enables the user to see different views of a stage in the computation without the need to recompute.

*Defining checkpoints*

As the most likely graphs to be shown in a compacted series are the first and/or the last, the checkpointing `whiff` primitives all exist in two forms to allow the user to choose which one as appropriate. For example the graph just before or just after any function application may be seen using definitions such as:

`CHECKBEFORE = isfun` or `CHECKAFTER = wasfun`, and to see both:

`CHECKBOTH = isfun || wasfun` To see graphs just before the application of a *particular* function, or set of functions one may use a definition such as:

`TARGETS = isin ["fname1","fname2",` *etc*`]`

The type of a checkpointing function is: `ReductionState` $\rightarrow$ `Bool`. Ideally all aspects of the reduction state would be accessible to the user *via* such primitives, including cumulative information such as the "function meter" that keeps track of the number of applications of each function,

Another variety of checkpoint is the "output event". A checkpoint to catch the step illustrated in Figures 5.15 and 5.16 would be:
`FIGSTEP = hasout`: the step has some output, or even `FIGSTEP = outis "3"` indicating "show the graph when a 3 is about to be output". If the *next* graph had been wanted, the `hadout` and `outwas` primitives could have been used instead. A safeguard is needed to allow for a checkpoint never being reached in a non-terminating computation. This can be established by making use of the `gstep` primitive which returns the step number of the graph, to determine a default checkpoint after some predefined large number of steps.

Other conditions on the overall state may also be of interest, such as the presence of particular application chains, but this would necessitate capabilities that the prototype system does not yet offer.

## 5.6 Overview of `hint`

A prototype programming environment for `h` has been developed to investigate the effectiveness of different techniques for presenting sequences of program graph. In addition to making and undoing function and filter definitions, and typing in expressions to be evaluated, possibilities for the user are itemized below, under the headings of the wish list reached at the end of Chapter 4.

1. **Let the user adapt the tracing to particular applications.** The user can define and apply spatial filters that enable a summarised view of the graph to be shown. Both these spatial filters and any temporal filters (or checkpoints) may be tailored precisely to the particular program, for example with the use of named functions.
2. **Allow the user to step through the reduction.** The user may step through the reduction either by single steps or between checkpoints defined as temporal filters;
3. **Permit the creation of breakpoints.** The user may observe the program graph at every reduction step — or less frequently as desired; the temporal filter may be changed at any breakpoint;
4. **Give detailed information about the reduction process.** The user may browse the program graph at each step at which it is displayed, receive details regarding both local and global information, and change their view of the graph by applying different spatial filters.
5. **Provide powerful techniques of filtering and focusing.** These are provided by the spatial and temporal filtering schemes.

This section describes the user interface, and gives an account of the features implemented in the prototype.

There are four main display areas needed: a prompt-response interface for typing in expressions, and for defining and undefining functions, filters and filter auxiliaries; a minigraph display area to give an overall view of the graph; a main display area; and a control panel. A sketch of the layout is given in Figure 5.17.

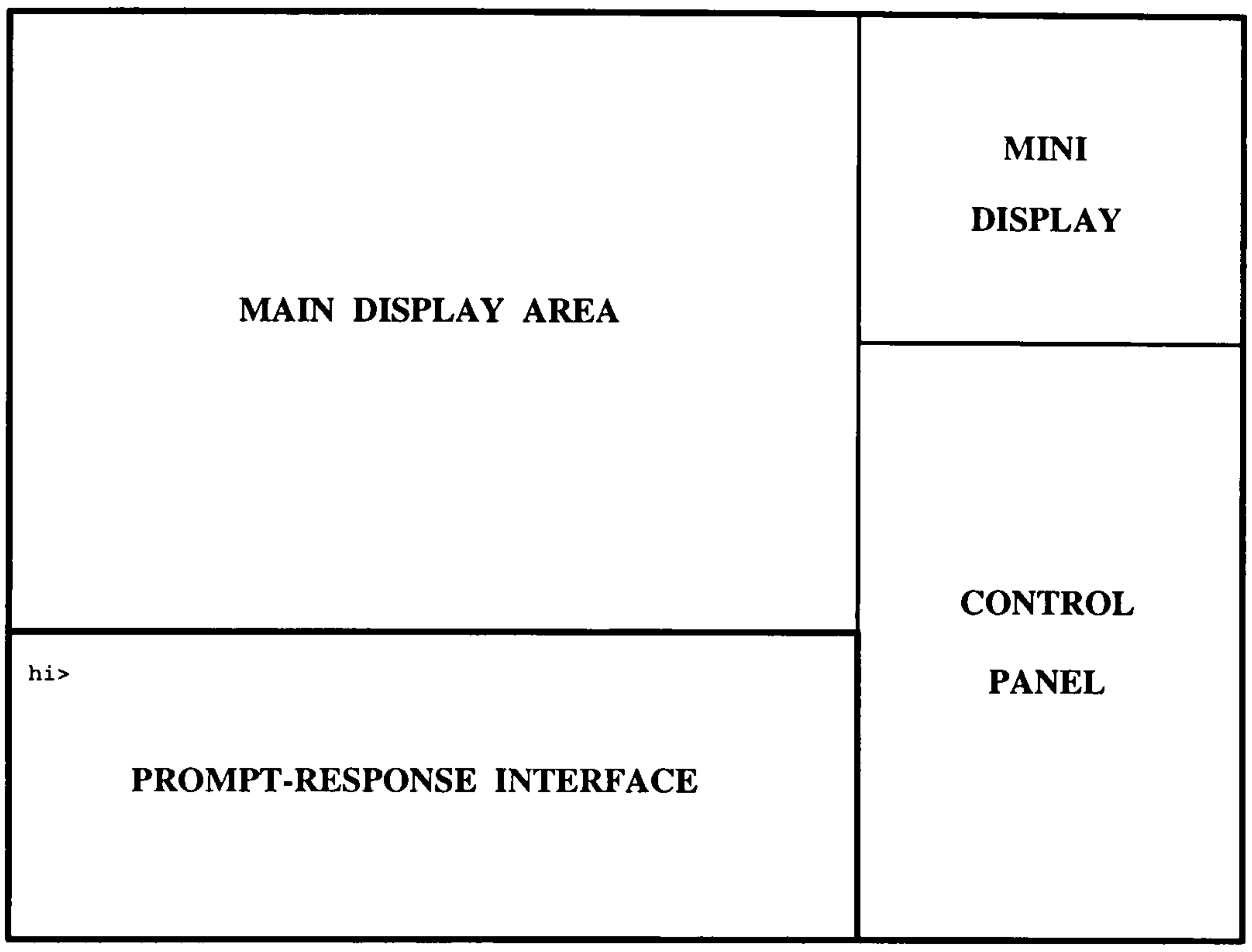



Figure 5.17: The layout of the `hint` screen.

### 5.6.1 The prompt-response interface

The prompt-response interface is not affected by the tracing: whether or not tracing is switched on the user may make or undo function and filter definitions, or offer an expression for evaluation. In return the system gives a message indicating whether the function/filter has been successfully defined/undefined, or the result of evaluating the expression. When the reduction is being monitored, the result is output progressively, if appropriate, as the computation proceeds.

### 5.6.2 The minigraph display

The whole program graph is scaled to fit in the minigraph display area, after being subjected to any current spatial filter. The intention is to give an overview of the graph rather than to present its details; labels would be too small to be readable in a large graph: so no labels are shown.

### 5.6.3 The main display area

The program graph, or as much of it as will fit, is shown in the main display area after compaction with any current spatial filter. The labeling is also determined by the spatial filter.

### 5.6.4 The control panel

The control panel in the prototype is used to display the *function meter*, a table of function and primitive names each associated with the total number of its applications so far. An "ideal system" would also use this area of the screen for locating control buttons for browsing and stepping, and for changing the current filtering elements, but this functionality is currently accomplished by entering commands at the keyboard.

### 5.6.5 Implementation and use of `hint`

The next two chapters discuss the implementation and use of the `hint` environment.

Chapter 6 outlines the implementation of the environment. It shows that Haskell is very apt for some aspects of the implementation such as: the direct expression of the reduction rules, graph representation, the transformation of the program graph into a graph-tree, the spatial filtering and display of a graph-tree, and the interpretation of the interface.

Chapter 7 is about the use of `hint`: possible benefits of using graph display in the teaching of functional programming; locating errors; exploring a program through browsing; the definition and composition of suitable spatial filters; the problem of labeling; and limitations of the system.

# Chapter 6

# The implementation of hint

## 6.1 Introduction

The implementation of `hint` is in Haskell as a contribution to the investigation of the appropriateness of using a lazy functional language for such an application.

A simple interpreter would evaluate a suitably parsed expression directly and return the result. This is not sufficient here, as the computational steps need to be separately identifiable for tracing purposes. So the reduction procedure is strongly governed by the requirement to create an original program graph which is transformed step by step until the final result of reducing the given expression is reached. Another consideration is the need to collect information about the reduction for possible display, both cumulatively and at each reduction step.

Aspects of the implementation that are of interest are those that conveniently exploit the use of a lazy functional language, and those which are necessary elements in the creation and filtering of the graph-trees that are used for display.

### Outline of chapter

This chapter discusses:

- the reduction process; (Section 6.2)
- the display of the program graph; (Section 6.3)
- the implementation of spatial and temporal filtering; (Section 6.4)
- and the hint interface. (Section 6.5)

## 6.2 Implementing the reduction

The reduction model is essentially a template instantiation model, the "simplest possible implementation of a functional language" as described in Peyton Jones and Lester [69]. This fulfills the requirement of providing a program graph for possible display at each reduction step, the nodes of which are directly associable with the user's source code.

This section describes:

- an overview of the reduction process; (Section 6.2.1)
- lexical analysis and parsing; (Section 6.2.2)
- the reduction state; (Section 6.2.3)
- the mechanics of function application; (Section 6.2.4)
- declarative implementation of the reduction rules; (Section 6.2.5)
- stepping through the reduction. (Section 6.2.6)

### 6.2.1 Overview of expression reduction

Users type in text representing expressions to be evaluated, and `h` and `whiff` definitions. This is parsed into an abstract syntax tree, which is bound and, in the case of an `h` expression, added to the *heap* to create the initial program graph. This undergoes successive transformations, the reduction steps, until the final program graph is reached. The process is summarised in Figure 6.1.

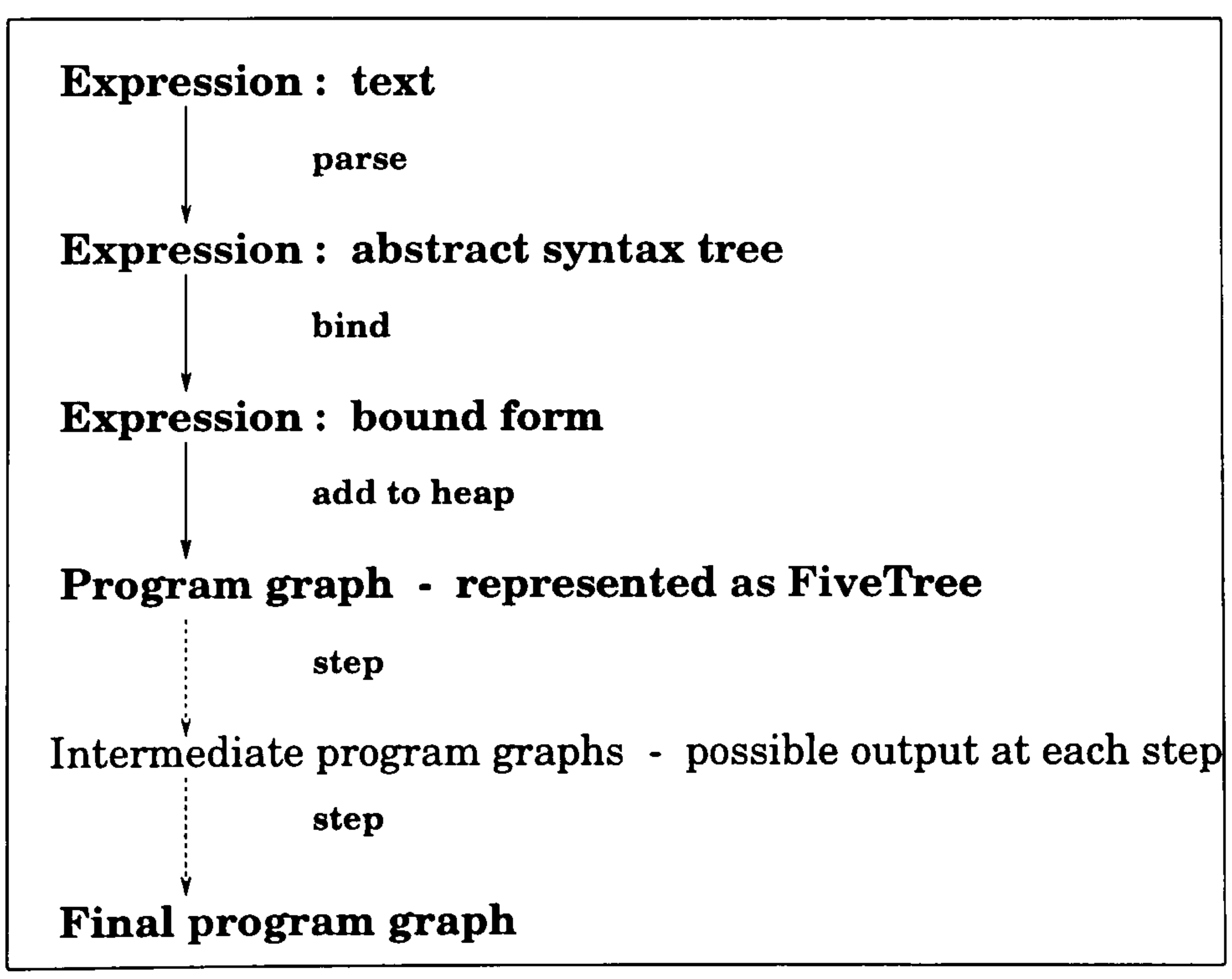



Figure 6.1: Stages in the reduction of an expression.

### 6.2.2 Lexical analysis and parsing

Lexical analysis and parsing in `hint` exploit well known techniques that have frequently been used to demonstrate the suitability of lazy functional languages for such applications (*e.g* [99, 33, 69]). The implementation uses a `Parser` type which takes a list of lexical tokens, and returns a triple: whether or not the parse has been successful, maybe a parse tree, and the remaining lexical tokens.[1]

```
data Deft      = Deftree      Identifier (Expr Identifier)
data WhiffDeft = WhiffDeftree Identifier (Expr Identifier)


data PT = E (Expr Identifier) | -- Abstract syntax tree
          D Deft              | -- Definition
          WD WhiffDeft        | -- Whiff definition
          ERROR String

type Parser = [Lexs] -> (Bool,Maybe PT,[Lexs])
```

Figure 6.2: The `Parser` type.

The `Parser` type is shown in Figure 6.2, which also shows the parse tree type, `PT`. There are legitimate intermediate forms of parse tree that are not shown — for example for gathering arguments to a primitive function.

```
data Expr a = ENAT Int                              |
              EVAR a                                |
              EPRIM Identifier [Expr a]             |
              ECONSTR Identifier [Expr a]           |
              EAPP (Expr a) (Expr a)                |
              ELAM Identifier (Expr a)              |
              ENIL                                  |
              ECONS (Expr a) (Expr a)               |
              EPAIR (Expr a) (Expr a)               |
              EBOOL Bool                            |
              ECHAR Char                            |
              ECASE Identifier (Expr a) [Expr a]    |
              ECASEPR CaseMatch (Expr a)            |
              EERR String
```

Figure 6.3: The `Expr` type.

[1] A parse may be successful yet return `Nothing`, hence the need for both the `Bool` and the `Maybe` type.

### The expression type

The `Expr` datatype (Figure 6.3) is parametrised on the type of value associated with variables, which is initially `Identifier`, but becomes `(Binding tag)` (Figure 6.4) when an expression is bound prior to becoming part of the program graph.

```
data Binding tag = ToThisIs Nid                    |
                   Func (FunRule tag)              |
                   Caf Identifier Nid [Identifier] |
                   BindError Identifier
```

Figure 6.4: The `Binding` type.

### The binding of expressions

The potential of parametrising the expression type is discussed in Peyton Jones and Lester [69] — the `Binding` type here is an example of their *binder*. An expression to be evaluated is transformed into a *bound* form, then its elements become tagged nodes in a *program graph* in which the reduction takes place.

A function name is replaced by an element that includes the function name, for display purposes, as well as the relevant function application rule. CAFs have within them the address of the expression that they represent.

### Program nodes

A constituent node of the program graph has three essential aspects: the sort of node it is, the addresses of its successor nodes, and a tag for garbage collection (See Section 6.2.3).

*The node type*

The sorts of node are enumerated in the `Nodeop` definition in Figure 6.5. As well as values, primitives, functions, constructors, application and indirection nodes (`ThisIs`), there are also:

- output nodes to synchronise the display of the graph and the output — for example keeping a constructor in the display of the graph while its arguments are being evaluated, even after its own representation has been output;
- `Cons` and `Pair` nodes, which though unnecessary at this level in their role as constructors, reflect the special syntax of lists and pairs, and facilitate their display;

```
data Nodeop tag = Val Value                       |
                  Prim Funid                      |
                  Closure SoFar (FunRule tag)     |
                  Constr Identifier               |
                  Apply                           |
                  ThisIs                          |
                  Out  Identifier                 |
                  Cons                            |
                  Pair                            |
                  Case Identifier                 |
                  Casepr CaseMatch                |
                  CaseApply Identifier
```

Figure 6.5: Sorts of node.

- `Case` nodes for the pattern matching case statement, and related to these: *case pair* nodes, `Casepr`, which associate a constructor with the appropriate function, and *case apply* nodes, `CaseApply`, which coordinate the application of the function to the actual arguments of the constructor being matched.

*The node class*

In order to be able to use nodes with different types of tag, a `Node` *class* is defined. This is in anticipation of using this field for holding information as well as garbage collection, for future use in the creation of quotient graphs. The definition of the `Node class` is given in Figure 6.6. The class requires the following operations:

`mark` — to mark the tag
`clear` — to clear the tag
`marked` — is the tag marked?
`scs` — addresses of successor nodes
`newscs` — change the addresses of successor nodes
`indi` — is this an indirection node?

The module includes functions related to these: `rna`, real address, and `rnv`, real node value, for following an indirection chain to the node it represents — the *ultimate referent*, and `nodelist` to return the successors of the node at a particular address. `Nid`[2] is defined as a synonym for `Ind` (index).

[2]`Nid` for *Node Identity*, but also to imply the nest (address) in the FiveTree (see page 104) that it represents.

```
module Nodes where

import FiveTree  (Ind(..),FT(..), nodevalue)

type Nid = Ind
type Freelist = [Nid]
type Busylist = [Nid]

class Node a where
        mark        :: a -> a
        clear       :: a -> a
        marked      :: a -> Bool
        scs         :: a -> [Nid]
        newscs      :: a -> [Nid] -> a
        indi        :: a -> Bool

realnode ::  (Node a) => FT a -> Nid -> (Nid, a)
realnode fta nid = case (indi node) of
                     False -> (nid, node)
                     _      -> realnode fta nid2
                               where
                               [nid2] = scs node
                   where
                   node = nodevalue fta nid

rna :: (Node a) => FT a -> Nid -> Nid
rna  = (fst .) . realnode

rnv ::  (Node a) => FT a -> Nid -> a
rnv      = (snd .) . realnode

nodelist :: (Node a) => FT a -> Nid -> [Nid]
nodelist = (scs .) . nodevalue
```

Figure 6.6: The node class.

The use of a `FiveTree` (see below) is assumed, but the heap might instead be represented by any of a class of types, including `Arrays`, that allow the appropriate updating and look-up facilities.

*The tag*

It may seem strange that the node type in Figure 6.5 has to be parametrised on the type of the tag. It is because the node type that embodies a function, the `Closure` constructor, carries within it a template instantiation rule that represents the function application. However this usually involves the creation of graph nodes, which are tagged. Thus the template instantiation rule has to be parametrised on the type of the tag, hence also the `Closure` constructor, hence the `Nodeop` type as well. Arbitrarily specifying the type of the tag to avoid this would make the implementation easier to read, but harder to change.

### 6.2.3 The reduction state

When an expression is to be reduced in `hint` a *reduction state* is derived from the overall evaluation environment. This reduction state is parametrised on the type of the node tags and on the type of global information to be collected. It has six component fields: the heap, a busylist, a freelist, *stacks* of nodes to be reduced, an output field, and a global information field.

#### The heap

```
data FT a = Finger a |
            Hand (FT a) (FT a) (FT a) (FT a) (FT a)
```

Figure 6.7: The FiveTree type.

The heap is an association of node-address pairs. For historical reasons[3] *FiveTrees* are used to keep track of the node/address associations. The type is given in Figure 6.7. Addresses are implicit in a given FiveTree. Figure 6.8 shows the implicit addresses in a two generation FiveTree. Figure 6.9 demonstrates how these addresses are assumed in the FiveTree lookup function.

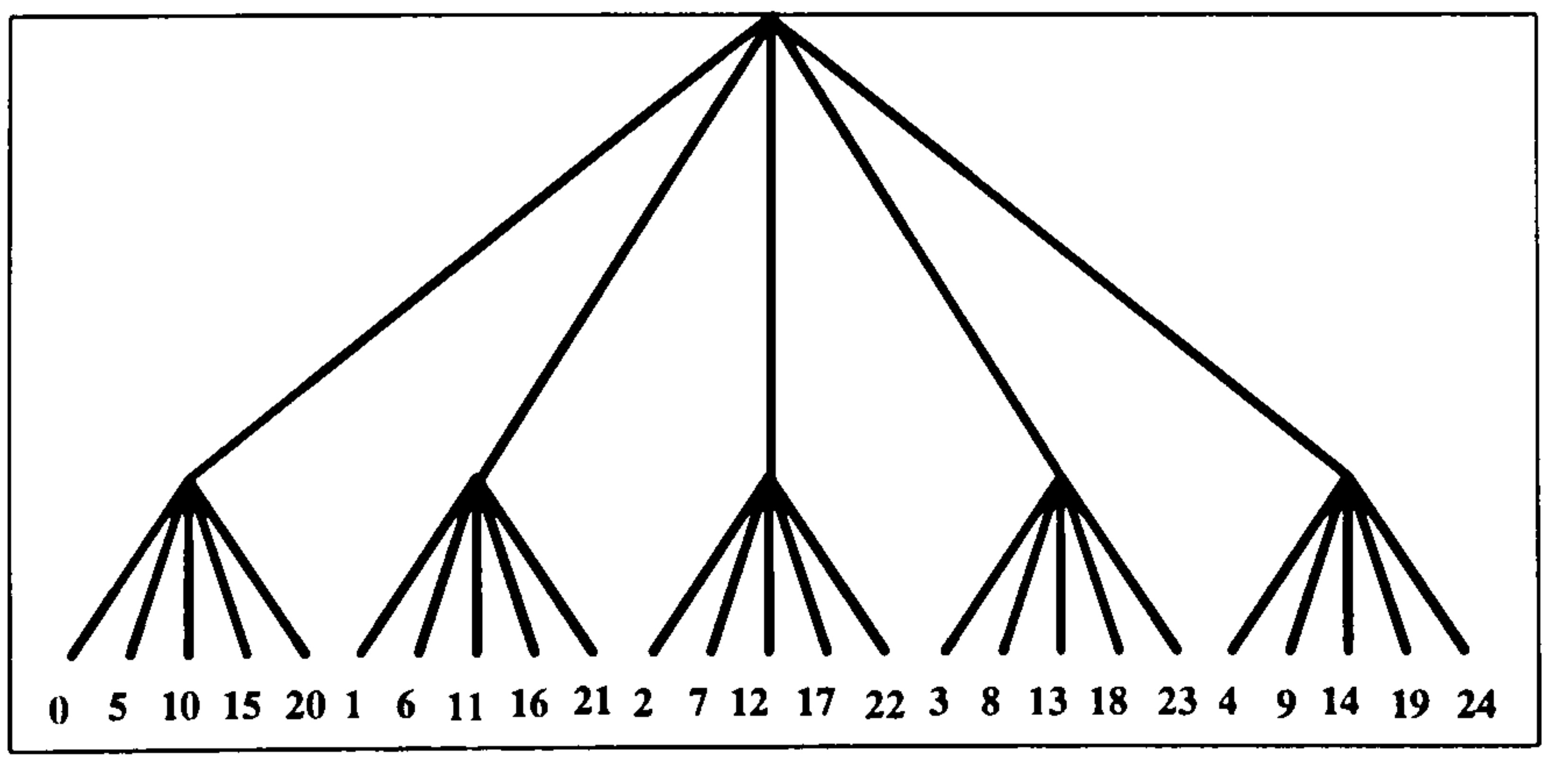



Figure 6.8: Implicit addresses in a two generation FiveTree.

[3]When the prototype was first being developed there was no efficient implementation of Haskell arrays. In addition the Haskell compiler that was being used (Glasgow: Version 0.4) did not support constructor functions with more than five arguments!

```
locateFT :: FT a -> Ind -> FT a
locateFT (Hand ft0 ft1 ft2 ft3 ft4) n=
        locateFT ft (n `div` 5)
        where
        ft = case newn of
               0 -> ft0
               1 -> ft1
               2 -> ft2
               3 -> ft3
               4 -> ft4
        newn = n `mod` 5
locateFT f@(Finger _) _ = f
```

Figure 6.9: Look-up in a FiveTree.

### The graph

```
data Graph index value =
         G index (index -> value) (index -> [index])
```

Figure 6.10: The graph type.

The graph type is parametrised on index and value types: indices uniquely identify vertices, values are vertex labels, not necessarily unique. A rooted directed graph is represented as the index of its root together with two characteristic functions. The first characteristic function maps indices to values, the second maps indices to their successors in the graph. Figure 6.10 shows an expression of this in Haskell. In the context of graph reduction, `index` is the type of addresses in the program graph, and `value`, the value type, is that of the program nodes. The first characteristic function is implemented by a look-up in the FiveTree, and the second relies on the successor operation, `scs`, of the `Node` class (see page 103).

### The stacks

The stacks are lists of nodes waiting to be reduced. With laziness and sharing it is possible that a node may be incidentally reduced before its turn in the stacks, but this does not matter as a check is always made whether the next node to be reduced is already in weak head normal form. There is one "final" stack, which originally has the node corresponding to the root of the overall expression to be reduced. If the final result is a base value, this will appear as a single node here in normal form. If the final result is a composite construction, the final stack will have more than one node in it corresponding to the number of arguments to the constructor node, and any intermediate output nodes which exist in order to synchronise the output. In the notation for the reduction rules in Appendix B, the Stack is the final stack of the reduction state and the Dump represents the rest of the stacks.

### The output field

When a final value has been completely or partially evaluated (for example the head of a list may be available), a string version of this is put in the output field of the reduction state. The next step is always to display this output, before proceeding with the reduction.

### Global information

The information field of the reduction state holds information that cannot be derived from the current program graph alone. This may include, for example, the current step number. Such information may be used in the display, but may also be transferred to the tag of nodes created by a function application.

### The free list, the busy list and garbage collection

The busy list contains the addresses of roots of live subgraphs, the nodes of which are not available for allocation. This includes the roots of predefined constant applicative expressions (CAFs). As the node type contains within it the addresses of successor nodes, the busy list is used to protect all the nodes involved in the CAF expressions. The busy list comes into play when the heap is full, and the reduction process calls the garbage collector to create a new free list. In addition to the busy list, the garbage collector must be given the address of the root of the program graph, from which all live nodes may be accessed, and any nodes that are currently in the process of being inserted into the graph. Figure 6.11 shows the complete garbage collection module, including the definition of the garbage collection function. It uses the higher order function `foldl` to apply the state transition function `live` recursively. The `foldl` function is also used in the binding of functions (Section 6.2.4). The *free list* has the addresses that *are* available when adding new nodes to the graph. The function to add a new node uses the head of the free list as the address at which to put it. As the busy list only contains the *roots* of needed expressions it is not the case that nodes are either on one list or the other.

```
module GC (gc) where

import Nodes    (Nid(..), Freelist(..),
                 Busylist(..), Node(..), rna)
import FiveTree (FT(..), nodevalue, updateFT, ftlmapFT, Ind(..))
import Normal   (Normal(..))

live :: (Normal a, Node a) => FT a -> Nid -> FT a
live fta nid =
  if marked node then fta
  else foldl live fta' (scs node')
  where
  node  = nodevalue fta nid
  node' = if indi node then newscs (mark node) [rna fta nid']
          else mark node
  [nid'] = scs node
  fta'   = updateFT nid node' fta

-- gc relies on the previous gc having cleared all nodes
gc :: (Normal a, Node a) => FT a -> Busylist -> (Freelist, FT a)
gc fta nids = ftlmapFT (not . marked) clear interfta
              where
              interfta = foldl live fta nids
```

Figure 6.11: The Garbage Collection module.

### 6.2.4 Function application

A function is applied when a `Closure` node of arity `n` is next to be reduced, and is saturated (*i.e.* already has `n` arguments). An expression built by the function rule within the closure is placed in the graph with its root at the address of the `Closure` node. The formal parameters of the function are replaced in the expression by indirection nodes that point to the addresses of the real arguments. The `ToThisIs` constructor in the bound expression is used for this: every `EVAR variable_name` in the function definition, that does not itself represent a function, becomes an indirection (`ThisIs`) node to the real argument in the relevant position of the argument list. Any references to function names in the function definition will become bound as they are added to the graph, including the calling function itself if it is recursive.

The implementation of mutually recursive bindings provides a good example of the use of circularity, a feature offered by lazy functional languages that allows part of the result of a functional application to contribute to the calculation of that result. This happens outside the environment for a particular reduction, in the more general evaluation environment for the interpreter. The function `tofroc` – "to function rule or CAF" – binds a single function or constant definition with reference to an existing group of bound functions and constants. In the definition of `multifroc`, which simultaneously binds a group of definitions, `tofroc` is directed by `foldl` to create a new evaluation environment with reference to the bound functions and constants *of the new environment that is being created.*

Before any expression evaluation in the `hint` environment, a check is made whether any new definitions have occurred since the last evaluation: if so, all functions and constants become rebound to create an up-to-date environment in which to bind the expression to be evaluated.

```
-- A circular definition for binding a group of definitions
-- before evaluating an expression
multifroc :: [Deft] -> Evalenv tag -> Evalenv tag
multifroc [] env = env
multifroc defs@(_:_) env = finalenv
          where
          finalenv = foldl (tofroc (fracs finalenv)) env defs
```

Figure 6.12: Circularity in the binding of a group of functions.

### 6.2.5 Declarative implementation of the reduction rules

The `step` function takes a reduction state and returns the next one in the series. It is effectively a concise declaration of the reduction rules. These are given in Appendix B.

The `h` language has primitive functions, from which all others are built, and these may correspond directly to Haskell functions. As `h` has no explicit type checking, the consistency of a primitive application has to be ensured before a value is passed to Haskell. For example the expressions `(2 > 3)` or `('a' > 'b')` will both yield `False`, but the expression `(2 > 'a')` will yield an error message from `hint`: the appropriate values will not be passed to the underlying Haskell which would cause the whole system to crash.

### 6.2.6 Stepping through the reduction

Stepping through the reduction involves moving from one step of interest to the next. With no temporal filtering in place this means displaying the graph after every application of the `step` function. When filtering is in place, this determines which steps are of interest. When function application is used for checkpointing, the graph may be shown before and/or after the function is applied. Figure 6.13[4] shows, for example, the first application of `take`[5] in the expression `take 1 [1,2,3]`

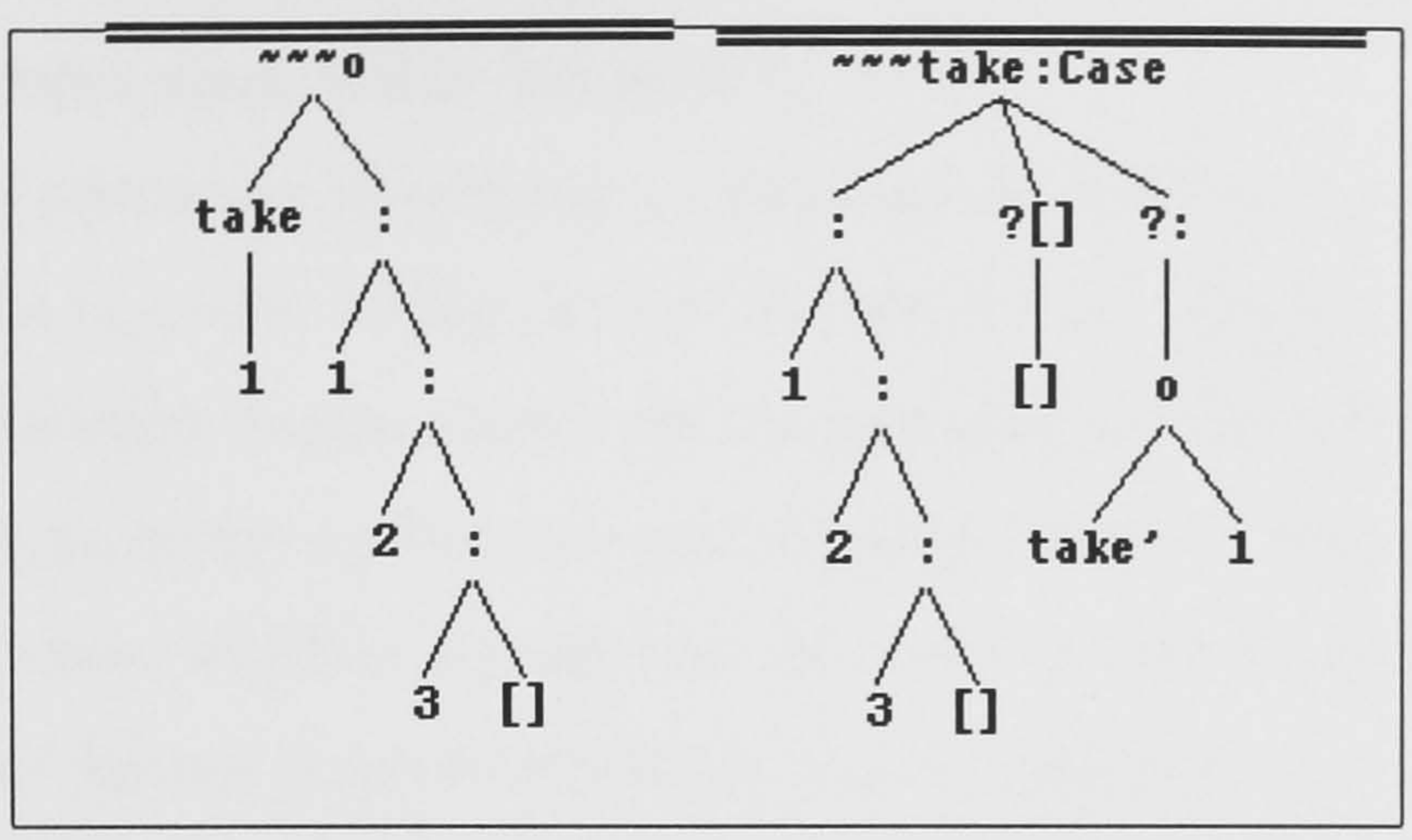



Figure 6.13: The application of `take`.

The first picture illustrates the marking, with ~~~, of the next node to be reduced. In this case it is an `Apply` node. The second shows the pattern matching case expression that arises from the function application: if it's [ ] return [ ]; if it's `(:)` apply `take' 1` to the arguments of the constructor.

## 6.3 Displaying the program graph

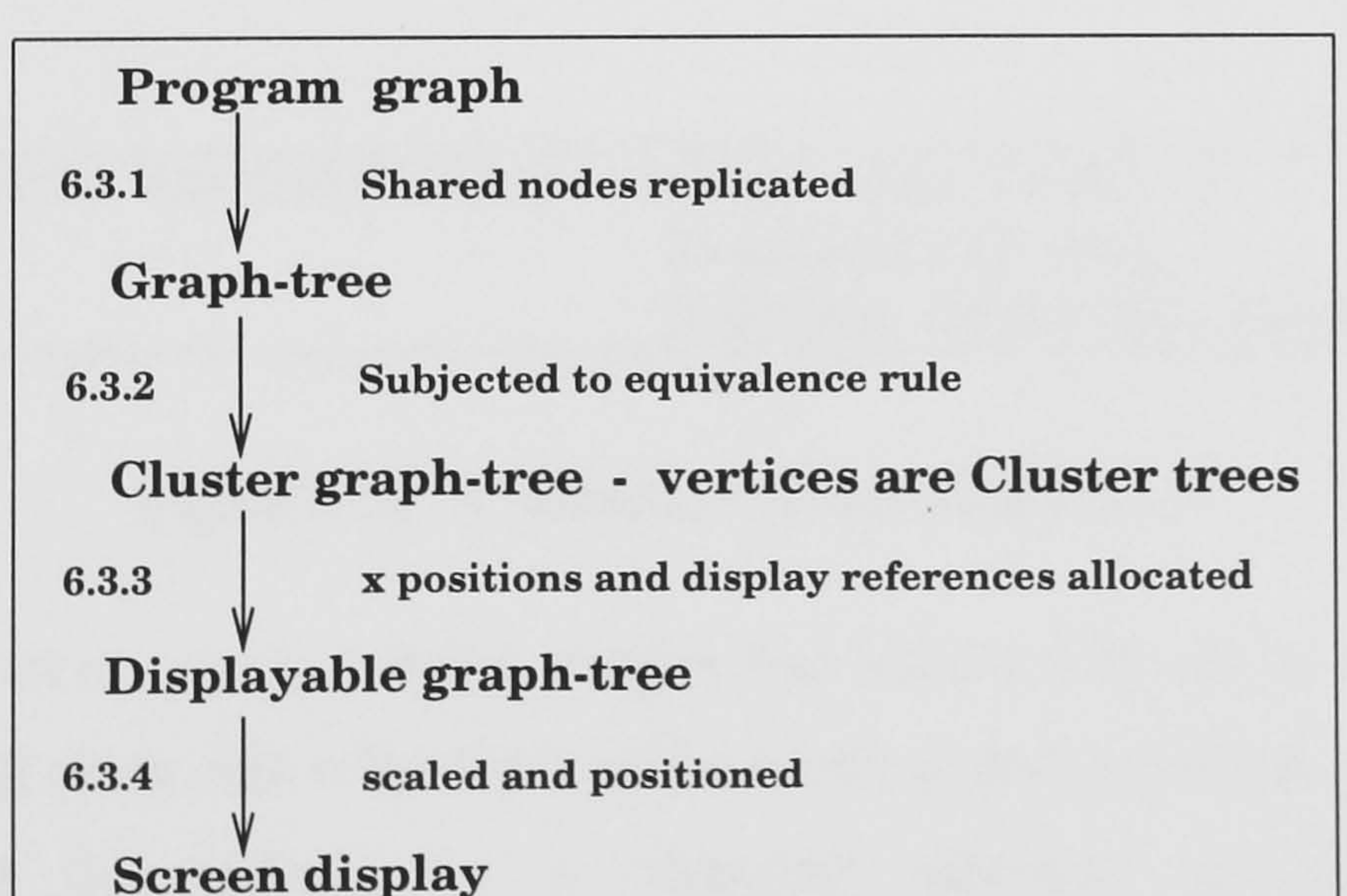



Figure 6.14: Stages in the display of a program graph.

[4]The thick horizontal lines that appear above some of the screendumps are part of the border of the display window, and have no other significance

[5]The h definition of `take` is given in Chapter 5, Figure 5.3.

The implementation of graph-trees exploits both laziness and the use of higher order functions. The relationships between the various graphs and trees involved in the display of a program graph are summarised in Figure 6.14. The numbers down the left hand side of the figure refer to subsections within this section.

A graph-tree is created by identifying a spanning tree of the program graph, and replicating shared nodes to create display leaves (Section 6.3.1). Sharing information is kept as an association between display leaves and the particular nodes that they represent. The graph-tree is subjected to the equivalence rule embodied in the current spatial filter to create a *cluster* graph-tree, which is a graph-tree the nodes of which represent appropriately collapsed regions of the raw graph-tree (Section 6.3.2). Labels are allocated to the clusters, and this enables x positions to be determined (Section 6.3.3). The display of this *displayable* graph tree then depends on the allocation of an x scaling factor, and a y distance with which to separate the generations (Section 6.3.4).

### 6.3.1 Graph-trees

As graphs in their own right, graph-trees can also be described using the `Graph` type. But because they include display leaves, they need to use an extended version of the index type of the graph from which they are derived. An example of an extended index type is given in Figure 6.15. Here the constructor `GraphNode` builds extended indices from original graph

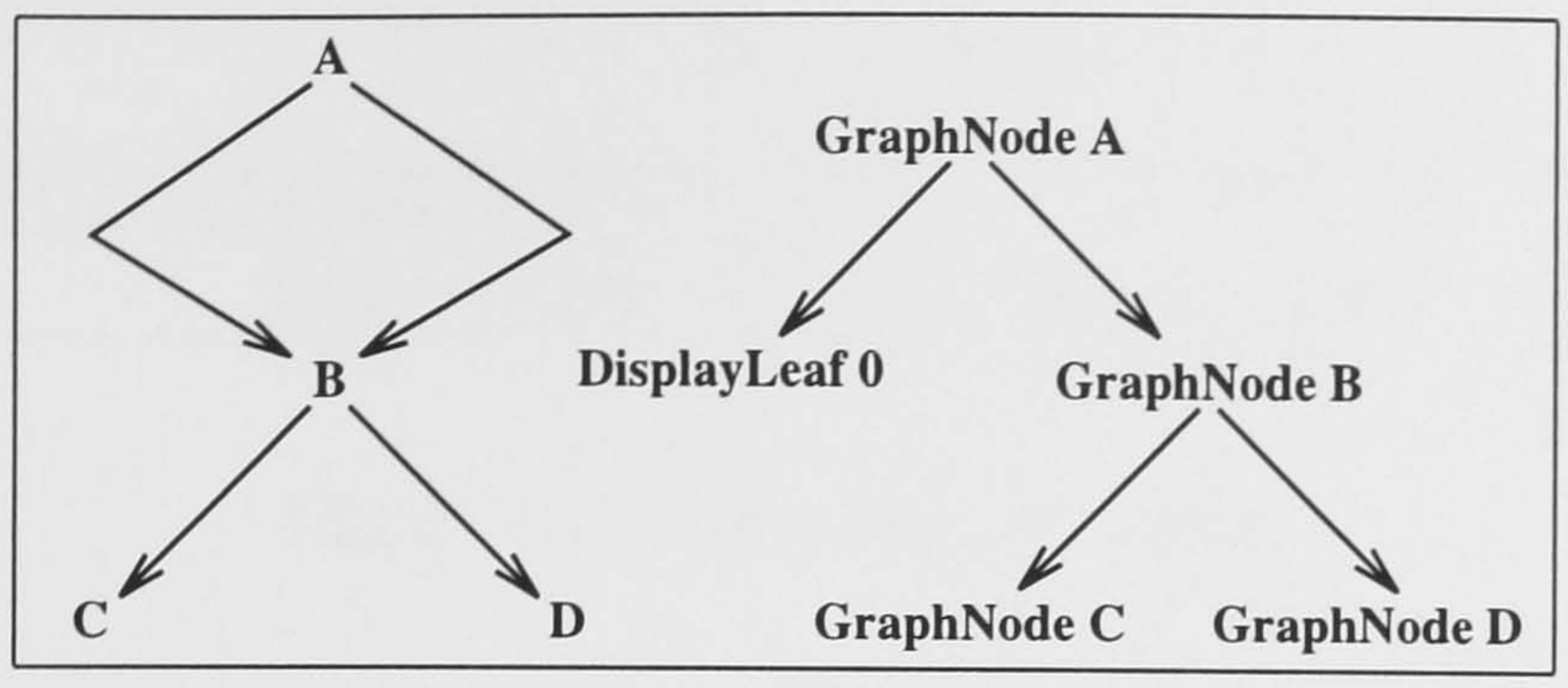
```
data ExtIndi index = GraphNode index |
                     DisplayLeaf Int |
                     NoIndex deriving (Ord)
```

Figure 6.15: A definition of extended indices.

node addresses; `DisplayLeaf` builds display leaf indices with unique integers. Finally `NoIndex` is a null value; this is the index of the parent of the root of a graph-tree, when the graph-tree type is defined as a threaded structure as described below. Figure 6.16 shows indices and extended indices for the graph-tree in Figure 5.8 on page 84.

#### The graph-tree type

A graph-tree is a represented as a function from an extended index to a pair consisting of the extended index of the parent and those of the child nodes. The graph-tree type is parametrised




Indices for a piece of graph with corresponding *extended* indices.

| Index | Value | Extended index | Index |
|---|---|---|---|
| A | * | GraphNode A | A |
| B | + | GraphNode B | B |
| C | 3 | GraphNode C | C |
| D | 1 | GraphNode D | D |
| | | DisplayLeaf 0 | B |

Necessary associations.

Figure 6.16: Extending indices.

on the type of the original index (Figure 6.17). The type is effectively a representation of two functions: from an extended index to its parent, and from an extended index to its children. The graph-tree type is used in conjunction with associations between extended indices and original indices. By reference to this and to the original graph, an extended index may be associated with a *value*, as well as with its predecessor and successors. By extending the index type in this way, it is possible to determine from the constructor of an extended index whether it refers to a display leaf or to an instantiated node.

```
data GraphTreeFun index =
          ExtIndi index -> (ExtIndi index, [ExtIndi index])
```

Figure 6.17: The graph-tree type

### 6.3.2 Cluster-trees: vertices of a compacted graph-tree

A cluster in the filtered graph tree must retain the structure of the part of the graph that it summarises. This may be done using a structure such as a *cluster-tree*, as shown in

Figure 6.18. The type is parametrised on an index type, which will in fact be an *extended*

```
data ClusterTree index =
            Unit index | Join [ClusterTree index]
```

Figure 6.18: The cluster-tree type.

index type. The intuition for cluster-trees is that they are either a unit `Unit`, or a composite `Join`. The composite has the root of the cluster-tree at the head of the list followed by its children, which are themselves cluster-trees.

The filtering process involves the creation of a *cluster-graph*, by the use of a particular filtering equivalence relation with reference to the raw graph-tree and its associated sharing information, and to the original graph. Again the representation of the structure involves various associations, here expressed as a binary search tree. Haskell definitions are given in Figure 6.19.

```
type ClusterGraph index =
    BinSearchTree index
            (ClusterTree index,[(index, Maybe Ref)],[index])

data Ord a => BinSearchTree a b =
       Empty |
       Branch a (BinSearchTree a b) (BinSearchTree a b)

type Ref = Int -- references coded as integers
```

Figure 6.19: The cluster graph and associated types.

The cluster graph is parametrised on its (extended) index type. The binary search tree is used to derive a memoised function from such an index to:

- the cluster-tree of which it is the root;
- indices of the constituent nodes of that cluster-tree, each associated with a reference if it is shared;
- the indices of the roots of child clusters.

### 6.3.3 *Displayable* graph-trees

The laziness of the implementing language is again exploited in the creation of the final structure used for the display and browsing of the filtered graph.

The displayable graph-tree type is a *threaded* structure of which an impression is given in Figure 6.20. Again this is based on the little example in the previous chapter (page 84). Each element represents a view of the *whole* graph-tree since each contains its parent, which in its turn contains the original element among its children. The diagram is simplified in that the cluster-tree at each vertex, and the *displayable graph-tree* in clusters deriving from display leaves, are not illustrated. The threadedness is of use when browsing, as a mouse click with the cursor over the root of the display may cause the parent of that graph-tree node to become the new root of the display. All the information needed for the new display is already encapsulated there.

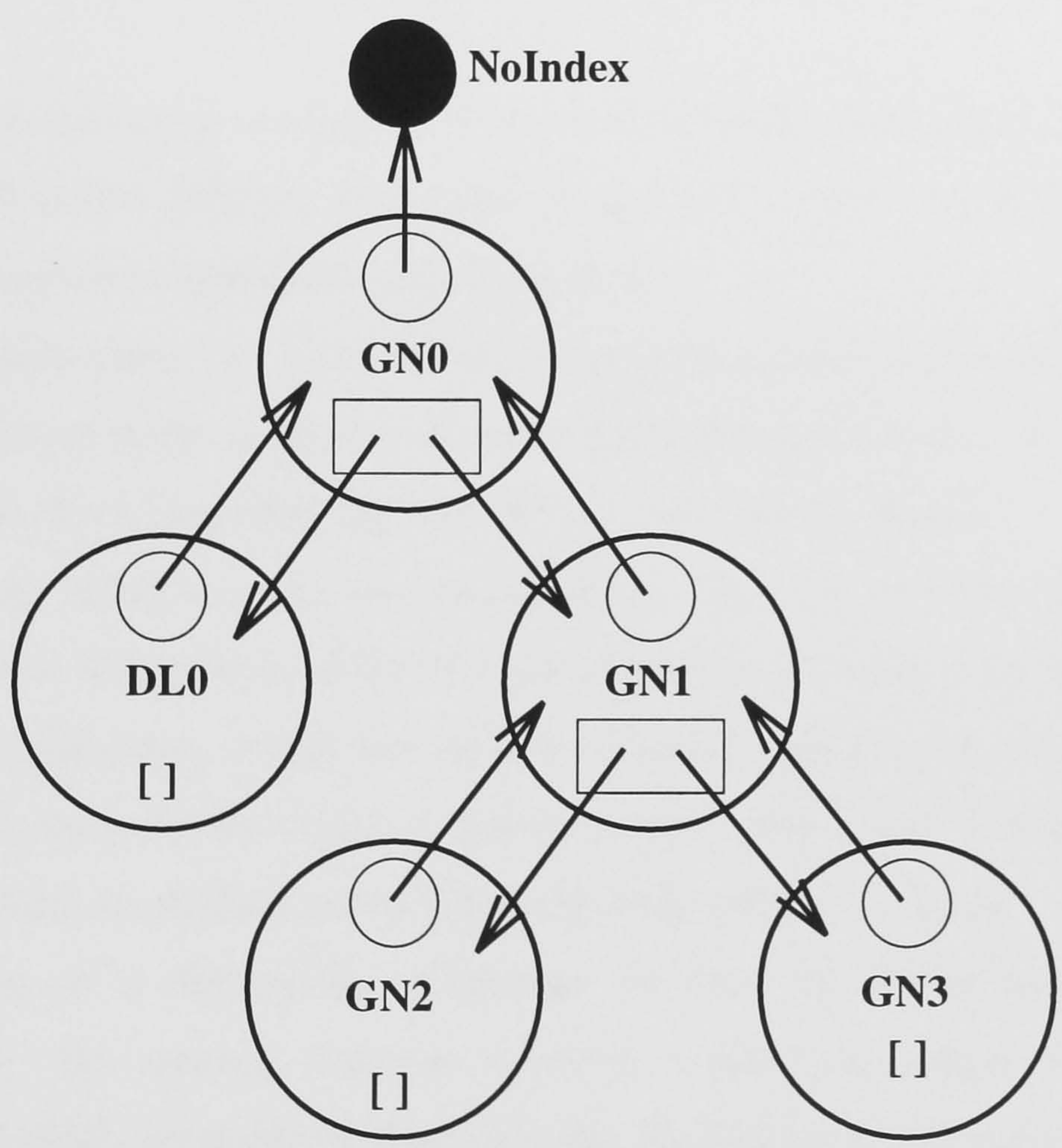



Figure 6.20: Threading.

The vertex of a displayable graph-tree includes the relevant cluster-tree, unless it represents a display leaf in the cluster graph, in which case it includes the *displayable graph-tree* appropriate to the cluster to which that leaf represents an arc.

Figure 6.21 shows the formulation of the displayable graph-tree and vertex types in Haskell. The `Xpos` is a provisional position on the x axis that may be scaled to an actual x coordinate. `Gen` is an integer for the *generation* – the depth from the *graph-tree* root as

```
data DispGraphTree index =
      DGT Xpos Gen (Vertex index) [DispGraphTree index] |
      NoDGT

data Vertex index =
     Val    index (ClusterTree   index) (Cref,Nref)|
     RefVer index (DispGraphTree index) (Cref,Nref)
```

Figure 6.21: The displayable graph-tree and vertex types.

opposed to the *display* root. The `Vertex` is parametrised on the (extended) index type as its instantiated version, `Val`, has a `ClusterTree` which is parametrised on the extended index.

The `[DispGraphTree index]` is a list of displayable graph-trees consisting of the parent, followed by the children, of the one in question. `NoDGT` is needed for the parent of displayable graph-tree representing the root cluster.

An instantiated vertex `Val` has its index, a cluster-tree, and a cluster and node reference. The cluster reference is `Nothing` unless *any* of the nodes in the cluster tree is shared. The node reference is `Nothing` unless the *root* of the cluster-tree is shared.

A display leaf vertex `RefVer` also has its index, which by reference to sharing information collected in the creation of the original graph-tree allows it to be associated with a value in the *original* graph, which may be needed for the construction of its display label. The display leaf vertex has the cluster reference of the vertex to which it represents an arc, and a node reference depending on the particular node within the cluster.

The creation of a displayable graph-tree involves an almost circular definition (see page 107): the creating function is given a parent as argument to use in the (parent:children) field, but those children have the displayable graph-tree *that is being created* as parent. Circularity is also involved in the allocation of references: when a cluster-tree is first encountered its corresponding vertex is given the reference that it *will have* in the fully completed structure. If it is not shared, this will be `Nothing`.

Provisional `x` positions are allocated according to a modification of Vaucher's algorithm [97]. This makes allowance for variations in the number of children and the length of vertex labels (but these are still restricted to a fixed number of lines of text). This has been adequate for demonstrating some of the problems and potential of displaying filtered graphs; but the question of labeling clusters, discussed in the Chapter 7, is complex and deserving of further work.

### 6.3.4 The display of the graph-tree

The display of a displayable graph-tree requires a scaling function that enables the `Xpos` to be translated into an actual x coordinate. It also requires a *y distance* by which to separate the generations.

In the main display the root of the graph is placed so that as much as possible of the graph is shown, given a fixed `y` distance, and a fixed separation between adjacent node labels. The `x` scaling function is calculated from this root position.

In the minigraph both the `x` scaling function and the `y` distance are calculated so that the *whole* graph may be displayed. The graph may change size fairly drastically over a short series of steps, so the scaling in the minigraph might be fixed so that the largest graph in that sequential group will fit, an example of "display inertia".

#### Browsing a graph tree

Clicking on a node brings it to the root of the display. The `x` position of the mouse cursor is associated with the cluster in the appropriate generation according to the `y` coordinate. By virtue of "containing" its display parent and children, the new displayable graph tree has all the information it needs to be redisplayed. The effect of browsing may be seen in Figure 7.12 in Chapter 7.

## 6.4 Implementing the filtering metalanguage

The aim of `whiff`, the filtering metalanguage, is to allow the user to express compaction rules, labeling functions and checkpoints in accordance with the model of reduction explained in Chapter 5. The user is provided with a palette of `whiff` primitives and primitive expressions with which to define filters. Auxiliary `whiff` definitions may also be used. This is illustrated in Chapter 5 where, for example, the `ARITH` filter on page 92 is defined using the auxiliary `AR`. Users write definitions of filtering functions and auxiliaries in `h` enhanced with the primitives that `whiff` provides. So `h` primitives and functions may be incorporated in `whiff` definitions.

Spatial filters require a compaction function and a labeling function to be associated by pairing. Checkpoints and compaction functions may be defined directly. Labeling functions are defined using the `u`, `j`, `d` "keyword" scheme described on page 91.

### 6.4.1 `whiff` primitives

<table>
<tr><td>def</td><td>::=</td><td>iden var* = rhs_expr</td></tr>
<tr><td>rhs_expr</td><td>::=</td><td>(\ var →)* (expr | rhs_expr)</td></tr>
<tr><td>expr</td><td>::=</td><td>hexpr | applic | prim |<br>cond | ujd | ( expr )</td></tr>
<tr><td>hexpr</td><td>::=</td><td><< h expression >></td></tr>
<tr><td>applic</td><td>::=</td><td>expr expr |<br>expr inop expr |<br>( applic )</td></tr>
<tr><td>prim</td><td>::=</td><td>nprim | vprim | rprim</td></tr>
<tr><td>cond</td><td>::=</td><td>if expr then expr else expr</td></tr>
<tr><td>inop</td><td>::=</td><td><< h infix operator >></td></tr>
<tr><td>preop</td><td>::=</td><td>( inop )</td></tr>
<tr><td>nprim</td><td>::=</td><td>producer | age | step |<br>ownpos | node | nodenl |<br>is (argnode | nodekind) | get argnode |<br>itis argnode val | relpfun</td></tr>
<tr><td>vprim</td><td>::=</td><td>vsize | sleft | sright | sage |<br>sstep | sroot | sfocus | ssize |<br>sref | srefs | showexpr | lit string</td></tr>
<tr><td>rprim</td><td>::=</td><td>isfun | wasfun | isprim | wasprim |<br>gstep | hasout | hadout | getout | gotout |<br>isin fnames | wasin fnames | outis string | outwas string</td></tr>
<tr><td>argnode</td><td>::=</td><td>Char | Bool | Prim | Function |<br>Constr | Casepair | Caseapply |<br>Output | Int</td></tr>
<tr><td>nodekind</td><td>::=</td><td>Val | Mathop | Ordop | Boolop |<br>Cons | Pair | Apply |<br>Dleaf | Focus | Case</td></tr>
<tr><td>relpfun</td><td>::=</td><td>child int | parent | allancs | someancs |<br>alldescs | somedescs | allkids | somekids</td></tr>
<tr><td>ujd</td><td>::=</td><td>u expr j expr d expr</td></tr>
<tr><td>val</td><td>::=</td><td>int | char | bool | string</td></tr>
</table>

Figure 6.22: Syntax of `whiff`.

The `whiff` primitive expressions may be regarded as returning result values of the expected `h` basic types. For example, `is Int: Bool`, `get Function: [Char]`, `gstep: Int`.

There are primitives that implicitly refer to:

- nodes, mainly for use in the display compaction;
- cluster-trees, mainly for use in the labeling functions;
- and reduction states, for the temporal filtering.

The labeling function may make use of *node* primitives, as well as the cluster specific ones. When this happens the node primitives are applied to the root of the cluster. The syntax of `whiff` is given in Figure 6.22.

### 6.4.2 Haskell functions to implement `whiff` primitives

Here are illustrations of Haskell functions underlying the three groups of `whiff` primitives.

#### Node primitives

When a `whiff` definition refers to `is Int`, for example, this is translated into the *Haskell* function `isInt` of type `(FilterArgs,ExtIndi) → Bool`. This first checks that the node represented is a `Value` node, then that it is indeed an integer. When a definition refers to a particular integer value, as in `itis Int 3`, this invokes the Haskell function `isIntN` of type: `Int → (FilterArgs,ExtIndi) → Bool`, which is itself defined in terms of `isInt`.

As another example, the `parent` primitive, which transfers a `whiff` function to the display parent of the node in question, invokes the Haskell `sfparent` function, defined as follows:

```
sfparent :: ((FilterArgs,ExtIndi) → a) → (FilterArgs,ExtIndi) → a
sfparent f = f . parenid
```

`parenid` is of type `(FilterArgs,ExtIndi) → (FilterArgs,ExtIndi)`, returning the `ExtIndi` of the display parent paired with the `FilterArgs`. This transfers the application of the function `f` to the display parent.

#### Cluster primitives

There is a similar relationship between the `whiff` cluster primitives and the underlying Haskell functions. Here the type is usually `(FilterArgs,Vertex) → String`. For example the `whiff` primitive `ssize` invokes the Haskell function `sizeshow` which returns an empty string when applied to a vertex which is a display leaf (paired, as ever, with the `FilterArgs`), but returns a string version of the size of the cluster in other cases.

#### Reduction state primitives

Finally here is an example of a reduction state primitive. A criterion for a checkpoint may be that a particular value is about to be output. Again the `whiff` primitive, `hasout` *string* directly reflects a Haskell function. It is of type `String → [ReductionState] → Bool`. The string argument to the `whiff` function is passed to this. The Haskell function is applied to the series of reduction states and returns the appropriate Boolean result. The Haskell function needs first to check the presence of output, then that it matches that described in the temporal filter.

### 6.4.3 The compilation of `whiff` expressions.

The compilation of a `whiff` expression involves the invocation of any such auxiliaries. The compiler returns an `h` expression. This incorporates values returned when the relevant Haskell auxiliaries are applied, in context, to a particular node, vertex, or series of reduction states. The "context" here is encapsulated in a type called `FilterArgs`, and includes all the information about the graph-tree that a filtering function might conceivable require. Whilst the argument to a `whiff` primitive is conceived of as "Node", "Vertex" or "Reduction State", the Haskell auxiliary is in fact applied to an `ExtIndi`, a `Cluster-tree`, or a series of reduction states, each associated (by pairing) with the current value of `FilterArgs`.

The interpretation of the resulting `h` expressions makes use of a different (and less complex) mechanism than that involved in the stepping interpreter. There is no need for stepping. More importantly, the `whiff` evaluator works at the `h` *expression* level – *i.e. values* of `whiff` computations are `h` *expressions.* This simplifies both the composition of `whiff` expressions, and the incorporation of `h` expressions into `whiff` definitions.

A filtering or labeling function is applied to its argument in an environment, `env`, of which the relevant components for the `whiff` compilation are a *context*, `cxt`, and *associations* between `whiff` identifiers and their definitions, `defs`.

The context here has two elements: aspects of the reduction state encapsulated in the `FilterArgs` type, and the focus of the filtering primitive. The `FilterArgs` type includes, for example, associations between the identifier of a node and that of its display parent, and details of the reduction state such as the current step number. The focus is a node in the case of a spatial filtering primitive, a cluster in the case of a labeling primitive, and a series of reduction states in the case of a temporal filtering function. The context is thus expressed as one of three different Haskell types. So the compilation of a `whiff` expression is effected by one of three Haskell functions, which apart from the type of the context element of the environment are otherwise very similar. Their action may be summarised in a small set of "compilation rules" shown in Figure 6.23.

Here $\mathcal{W}_{env}$ represents compilation in an environment consisting of: the associations between `whiff` names and their definitions, $env\{def\}$; and the node, cluster-tree, or series of reduction states in context, $env\{cxt\}$. $Hask$ represents the Haskell filtering auxiliary associated with a primitive `whiff` expression.

1. $\mathcal{W}_{env}$ $[\![$ $id$ $]\!]$ $= \mathcal{W}_{env}$ $[\![$ $env\{def\}$ $id$ $]\!]$
2. $\mathcal{W}_{env}$ $[\![$ $w$ $]\!]$ $= Hask_{env\{cxt\}}$ $w$
3. $\mathcal{W}_{env}$ $[\![$ $f$ $e$ $]\!]$ $= \mathcal{W}_{[\![f]\!]env}$ $[\![$ $e$ $]\!]$
4. $\mathcal{W}_{env}$ $[\![$ $v$ $]\!]$ $= v$
5. $\mathcal{W}_{env}$ $[\![$ $c\, e_1 \ldots e_n$ $]\!]$ $= c\, (\mathcal{W}_{env}\, [\![\, e_1\, ]\!]) \ldots (\mathcal{W}_{env}\, [\![\, e_n\, ]\!])$
6. $\mathcal{W}_{env}$ $[\![$ $(\lambda\, id.e)expr$ $]\!]$ $= \mathcal{W}_{env}((\mathcal{W}_{env}\ e)[expr/id])$
7. $\mathcal{W}_{env}$ $[\![$ $e_1\ e_2$ $]\!]$ $= (\mathcal{W}_{env}\, [\![\, e_1\, ]\!])(\mathcal{W}_{env}\, [\![\, e_2\, ]\!])$
8. $\mathcal{W}_{env}$ $[\![$ $h\ e_1 \ldots e_n$ $]\!]$ $= h\ (\mathcal{W}_{env}\, [\![\, e_1\, ]\!]) \ldots\ (\mathcal{W}_{env}\, [\![\, e_n\, ]\!])$
9. $\mathcal{W}_{env}$ $[\![$ $\lambda\, id\ e$ $]\!]$ $= \lambda\, id\ (\mathcal{W}_{env}\, [\![\, e\, ]\!])$
10. $\mathcal{W}_{env}$ $[\![$ `case` $e$ `of` $cp_1 \ldots cp_n$ $]\!]$ $=$
    `case` $(\mathcal{W}_{env}\, [\![\, e\, ]\!])$ `of` $(\mathcal{W}_{env}\, [\![\, cp_1 ]\!]) \ldots (\mathcal{W}_{env}\, [\![\, cp_n ]\!])$
11. $\mathcal{W}_{env}$ $[\![$ $c\ \longrightarrow\ e$ $]\!]$ $= c\ \longrightarrow\ (\mathcal{W}_{env}\, [\![\, e\, ]\!])$

Figure 6.23: Compilation rules for `whiff` expressions.

Other notation is as follows:

- $id$ – an identifier;
- $w$ – a `whiff` primitive expression;
- $f$ – a "Family relationship" `whiff` primitive;
- $v$ – an expression representing a basic value;
- $c\, e_1 \ldots e_n$ – a constructor with its arguments;
- $e_1\ e_2$ – $e_1$ applied to $e_2$;
- $h\ e_1 \ldots e_n$ – an `h` primitive function with its arguments;
- `case` $e$ `of` $cp_1 \ldots cp_n$ – a case expression;
- $c\ \longrightarrow\ e$ – a case pair.

The three rules that are crucial to the compilation are rules 1, 2 and 3.

An identifier that represents a user defined `whiff` expression is replaced by the body of the definition (Rule 1), through a look up in the `defs` element of the environment. Other identifiers may be names of `h` primitives or user defined `h` functions. These are unaffected by the compilation, and returned unchanged as part of the `h` expression that is the result.

A `whiff` primitive expression is replaced by the result of applying its associated Haskell filtering auxiliary in the relevant environment (Rule 2). For example:

**Node primitive** The `whiff` expression `is Int` is replaced by the (`h` expression version of the) Boolean returned when the Haskell filtering auxiliary `isInt` is applied to the particular node with the associated reduction information.

**Cluster primitive** The primitive `vsize` is replaced by the (`h` expression version of the) Integer returned when the Haskell filtering auxiliary, also called `vsize` is applied to the particular cluster.

**Reduction state primitive** The `whiff` primitive `isfun` – does the current reduction state represent the application of a function? – is similarly replaced by the (`h` expression version of the) Boolean returned when the Haskell filtering auxiliary `isFun` is applied to the current reduction state.

In all cases the `cxt` element of the environment is the relevant focus (node, cluster or series of reduction states) paired with the `FilterArgs`.

The third crucial rule, Rule 3, specifically applies to *spatial* filtering: it is "family relationships" between graph-tree nodes that are in question. One can imagine, however, using similar functions in the context of a series of reduction states – with, for example, the current reduction state having a similar relationship to the previous reduction state as a graph-tree node has to its display parent, *i.e.* the one before.

The "family relationship" function `f` is one of:

| | | |
|---|---|---|
| parent | – | the display parent |
| child n | – | the child node at position n |
| anyancs | – | any display ancestor |
| allancs | – | all display ancestors |
| anydescs | – | any display descendant |
| alldescs | – | all display descendants |
| anykid | – | any child node |
| allkids | – | all child nodes |

The effect of such a primitive is to transfer the application of its first argument, which is functional, to an environment in which the context element of the environment is changed appropriately. The `FilterArgs` is not changed, but the focus moves from a particular node to its parent/child node/display ancestors etc.. For example where `f` is `parent` the context changes from a node to its display parent; where `f` is `child 0`, the function is applied to the leftmost child of the node currently in focus.

The other compilation rules ensure that these crucial rules are applied wherever `whiff` primitive expressions or identifiers are encountered, not just at the top level.

### 6.4.4 Incorporating filters in the display

When a filtering function is applied, it is in the context of a particular node, cluster, or series of reduction states (together with relevant other data represented by `FilterArgs`). Haskell auxiliaries involved in the filtering return values that are incorporated into the `whiff` compiled `h` expression. This is reduced, using the non-stepping expression interpreter, to yield an `h` expression from which the appropriate Haskell value may be obtained. For example the Haskell value `True` is derived from the `h` expression `EBOOL True`. The value is of type `Bool` in the case of the spatial compaction and checkpointing filters, and `String` in the case of the labeling function.

## 6.5 The `hint` interface

### 6.5.1 The control panel

There are many possibilities for the `hint` interface. As well as the various browsing schemes described in Section 5.4.3, there is also the option of exploiting the potential offered by the control panel window. In the prototype this is used only to display cumulative information about the reduction. It would also be an appropriate area for displaying cumulative information as specified by a caption function. But the idea behind the control panel is to help the user choose and define spatial and temporal filters for the display of the reduction.

### 6.5.2 The interaction

The implementation of the prototype uses the generalised version of the overall interaction function used in the Escher program in Chapter 3 (Figure 3.5). As the interaction is almost entirely controlled at the keyboard, the system of interpretation of active areas used in the Escher program is not necessary. Apart from browsing, where the sites of the displayed graph-tree nodes become points within one big active area, there is no need to locate input.

However the `interpret` function could be incorporated if the definition of compacting functions, and the changes of checkpoint and of spatial filter were to be accomplished through dialogue boxes, rather than by simple text commands as at present. Then the relevant areas within the control panel window would be programmed to initiate appropriate dialogues. This would have the benefit that the user could, for example, be prompted for the three components needed in the definition of a labeling function. Warnings could be given where a user refers to a `whiff` component that has yet to be defined. Lists of appropriate named functions could be displayed, for example when changing filters. Such lists could be merely reminders, or could themselves become menus from which the user could select the required element. Such guidance in the definition of filters and their components would make the process simpler for the user and help to reduce errors.

Another function of the control panel might be to change modality, so that the effects of a mouse-button click in a particular area are not restricted to two: the number of buttons. For example a button click in the displayed graph-tree area might have one of *three* effects: to change the root of the display, to expose the structure of a cluster-tree, or to change the label shown.

### 6.5.3 Appearance of the display

The appearance of the display of the prototype is more easily conveyed with screendumps than with words, so the next chapter uses screendumps to illustrate the use of the prototype. There is also a discussion of the problems of labeling, and of limitations of the system.

# Chapter 7

# The use of hint

## 7.1 Introduction

The `hint` environment may be used for various purposes. It may be regarded as an interpreter for defining functions and evaluating expressions at the prompt/command/response interface. The "stepping" through the reduction may be used to illustrate the mechanics of simple graph reduction. The system may be used to investigate the cause of a wrong result, or, if the result is correct, on the cause of a space fault or, indeed, on confirmation that there *is* no space fault. If the error is that the computation does not terminate, checkpoints may be set to monitor its progress in the expectation that the state of the graph will help pinpoint where the problem lies.

This chapter presents screendumps that illustrate how `hint` may be used in teaching and in finding errors (Sections 7.2 and 7.3). There is then a longer example which demonstrates the effect of browsing, and shows how a spatial filter can be tailored to the compaction of a particular display (Section 7.4). There is then discussion of the problem of labeling subgraphs (Section 7.5). This is a vital aspect of the monitoring as the label has to show clearly just enough detail to give the user the information needed to understand both the nature of the particular cluster that it represents and the relationship of this to other collapsed subgraphs on the display. Finally there is a section concerning the limitations of the `hint` system (Section 7.6), followed by a chapter summary.

## 7.2 Visualizing simple graph reduction

One use of the `hint` environment is to illustrate the mechanics of simple graph reduction for teaching purposes. In this section we give further examples of this:

- a graphical representation of the `map` function;
- an illustration of the cumulative filters involved in the realisation of the "sieve of Eratosthenes" definition of `primes`;
- and a comparison of the higher order list-processing operators `foldl` and `foldr`.

### 7.2.1 The `map` function

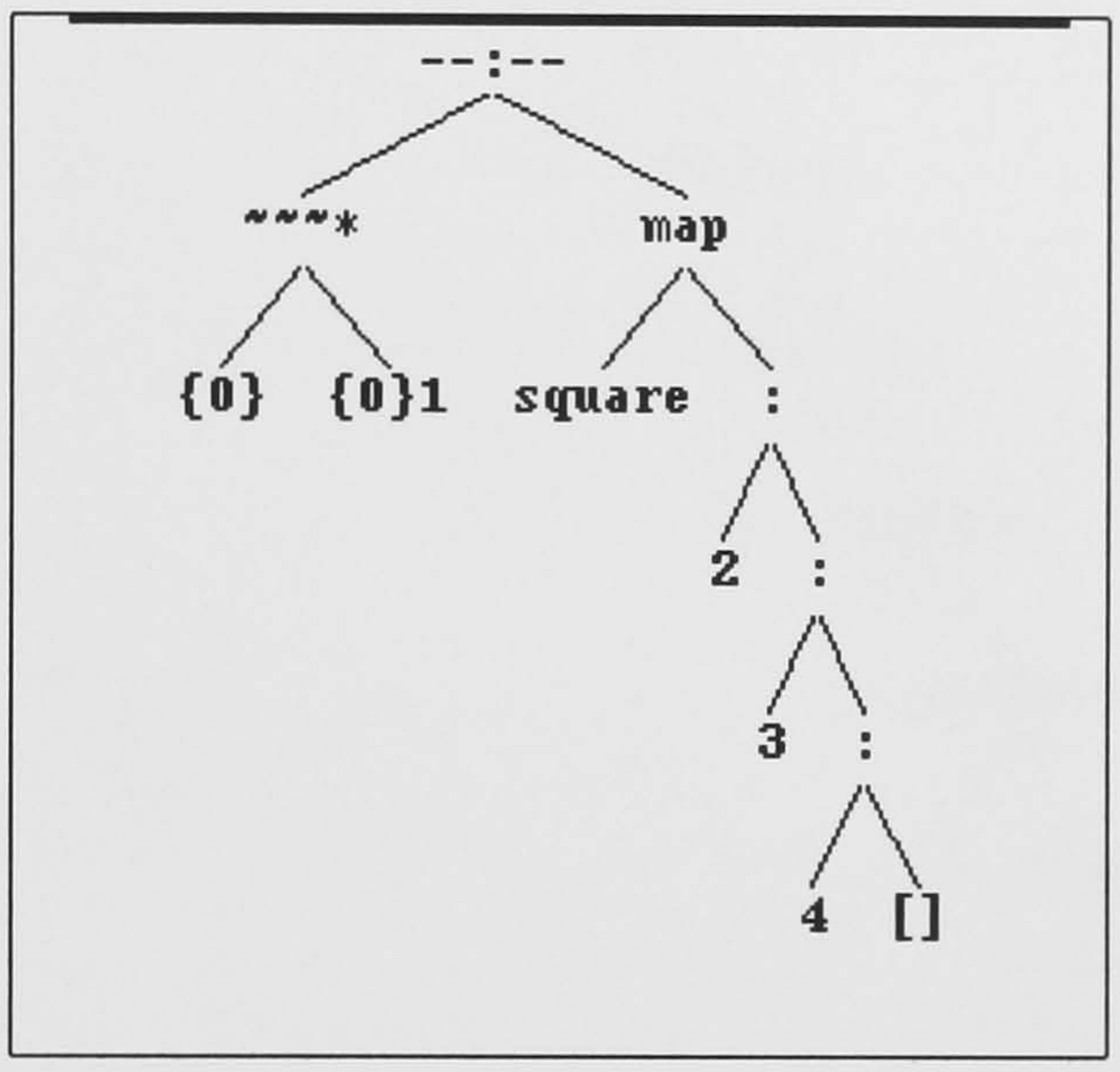



Figure 7.1: The `map` function.

Figure 7.1 illustrates elements involved in the display of a simple reduction under the `NOAPPLY` filter (defined on page 90). It shows the step in the reduction of the expression: `map square [1,2,3,4]` where `square 1` has been identified as the first item to output, and the `square` function has just been applied. The output node, `--:--`, indicates that the value of reducing the expression on its left is to be output, and is there to enable the rest of the program graph to be displayed during this reduction. The sharing of the `1`, the argument to `square` is shown in the display reference: `{0}`. The focus of reduction is the `*` node, as indicated by the ~~~ before it. The "rest of the graph", on the right of the output node, is seen to represent the expression `map square [2,3,4]`.

### 7.2.2 The sieve of Eratosthenes

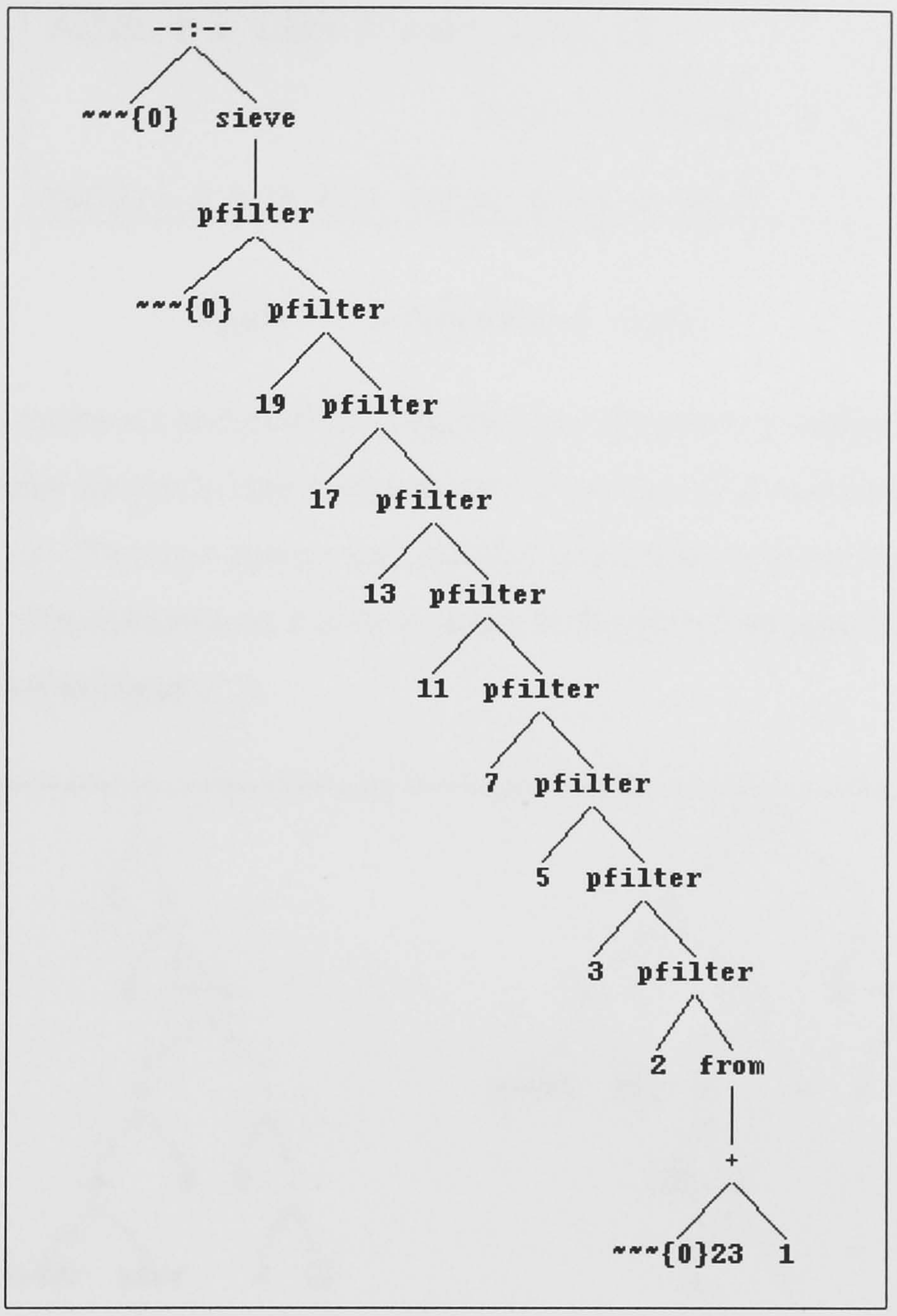



Figure 7.2: A barrage of filters.

Figure 5.14 on page 92 showed an example of the sieve of Eratosthenes used to compute the list of prime numbers to illustrate the use of the NOAPPLY filter. Figure 7.2 shows it a few steps later as the number 23 is about to be output. This shows the amassing of the barrage of filters through which a new number must pass.

Again the sharing is indicated by the display references $\{0\}$, each of which represents the 23 which is clearly marked in the graph, in all its instances, as being the next node to be reduced.

### 7.2.3 The two list `fold` operators

```
foldl f z list = case list of
                      []    -> z
                      (h:t) -> foldl' f z

foldl' f z h t = foldl f (f z h) t
```

Figure 7.3: `h` definition of `foldl`.

Introductory textbooks on functional programming often have a section which compares the two higher-order list processing functions `foldr` and `foldl` (sometimes called `reduce` and `accumulate`). The `hint` system may, through its pictures, help the student understand the comparison. The definition of `foldr` is given in Figure 5.7 on page 82. The definition of `foldl` is shown in Figure 7.3.

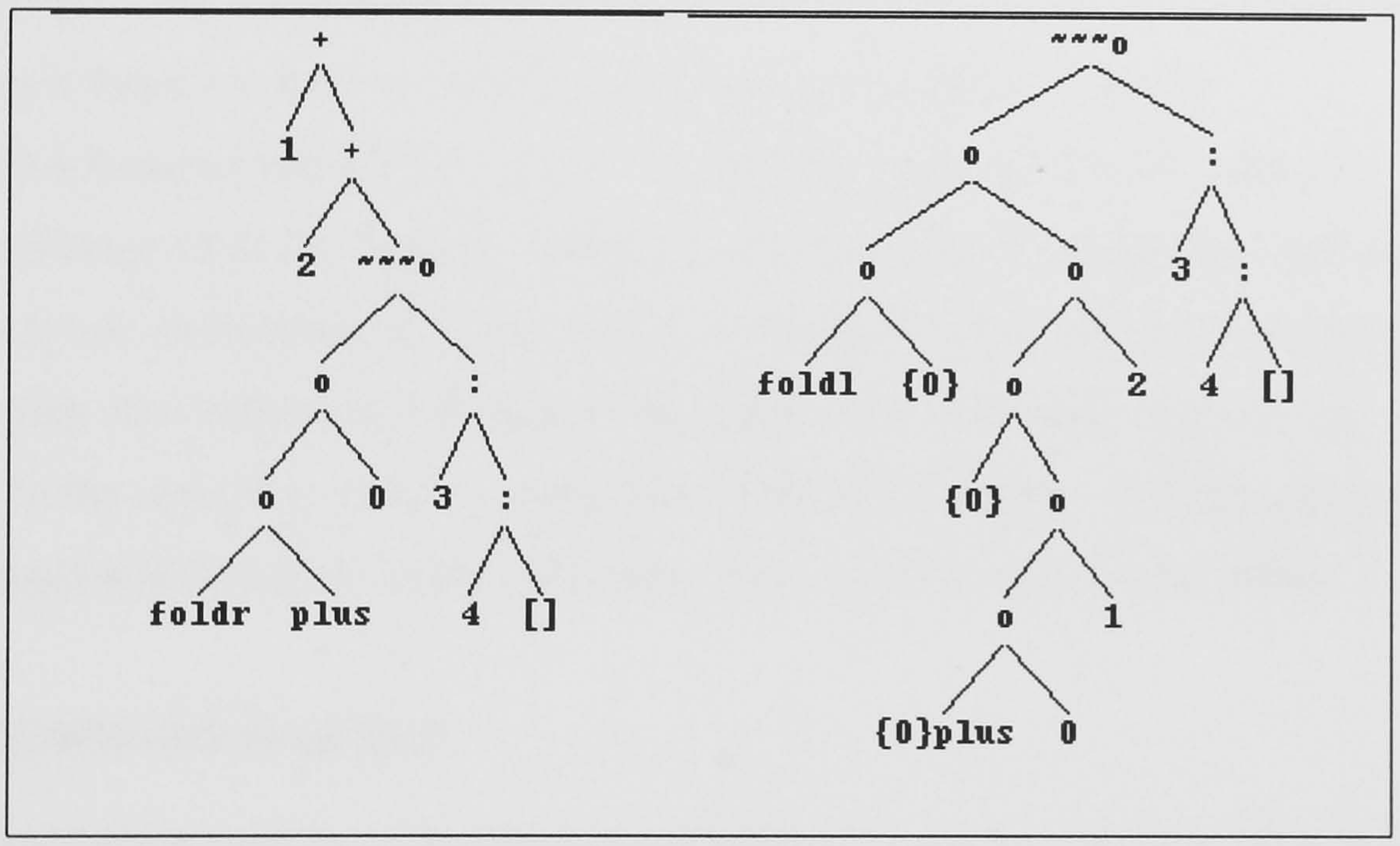



Figure 7.4: Comparison of `sum` defined in terms of `foldr` and `foldl`.

Figure 7.4 gives a visual basis for comparison of the use of `foldr` and `foldl` in the definition of `sum`. The expression in each case is `fold plus 0 [1,2,3,4]`. In the case of the use of `foldr`, in the screendump on the left, it may be observed that the + node at the root might have been applied, had `hint` had the associativity of + built into its reduction rules, as soon as the `2` became available. The `foldl` example shows that a strictness annotation on the second argument to the `foldl` function would in this instance save space as the numbers would be summed eagerly, rather than lazily, into the accumulator. In the example shown, the expression `plus (plus 0 1) 2` would be represented by a single `3` node.

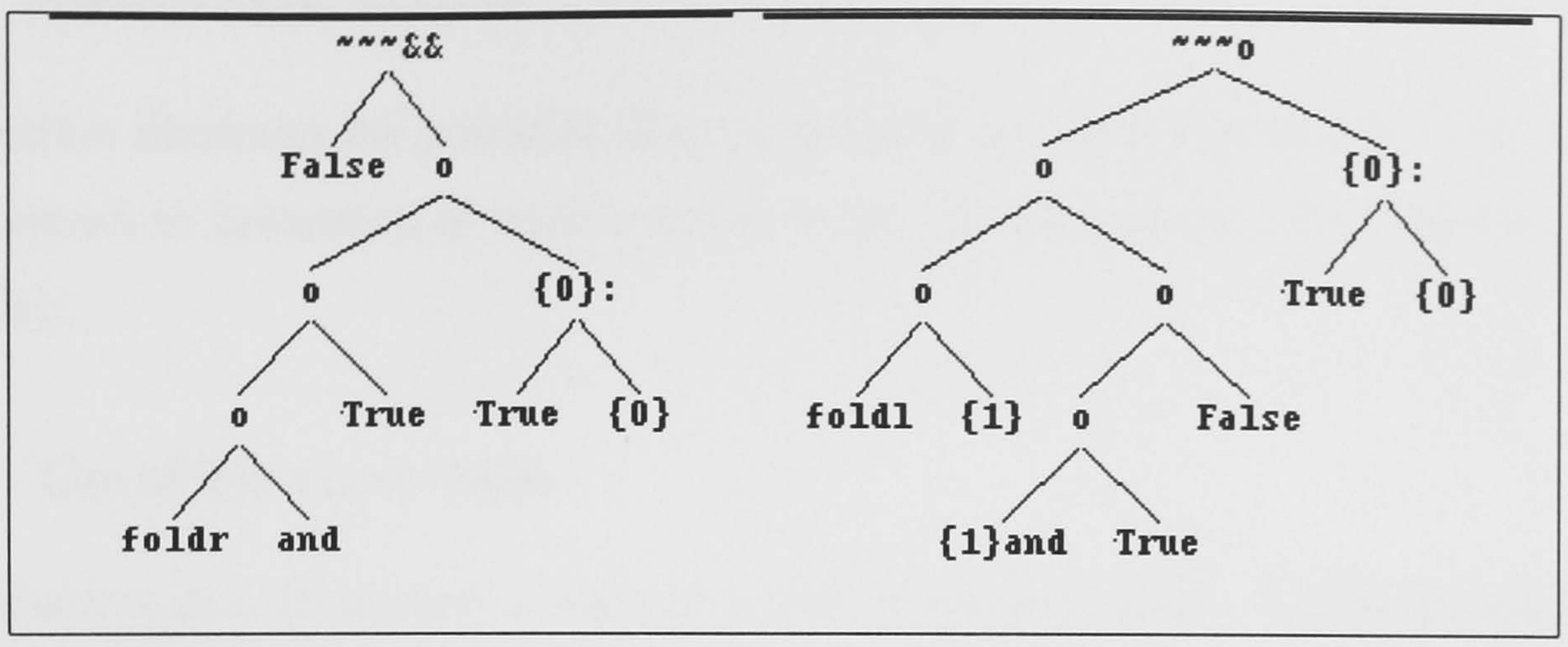



Figure 7.5: Comparison of `andlist` defined in terms of `foldr` and `foldl`.

Figure 7.5 illustrates a potential space leak in `foldl`. It gives visual confirmation of the relative efficiencies of `foldr` and `foldl` in the context of defining a list version of `and`, as described in Bird and Wadler [13] (page 151). They point out that given a list: $\mathtt{xs} = [\mathtt{x_1}, \mathtt{x_2}, \ldots, \mathtt{x_n}]$, and assuming that some element of this list, $\mathtt{x_i}$, is `False`, the expression `foldr and True xs` requires but $\mathcal{O}(i)$ steps for its evaluation, whereas `foldl and True xs` needs $\mathcal{O}(n)$ steps. The expression involved in the figure is: `fold and True (False:trues)`, where `trues` represents an infinite list each element of which is `True`. In the case of `foldr`, `False` is encountered as the first element of the list, and the very next reduction will reduce the graph to but one node representing the value `False`. In the case of `foldl` the computation will not terminate. Adding strictness to the second argument to `foldl` would save space, but would not affect termination.

### 7.2.4 Animated diagrams

The examples that have been presented suggest that showing stepwise reduction to a student may help in conveying insight. It could be argued that showing the procedure of reduction is not conducive to declarative thinking: the particular order of reduction, and the details of the implementation should not need to be taken into account. Yet teachers of functional programming do use diagrams that look uncannily like `hint` screendumps! Used with discretion, `hint` could usefully animate such diagrams for the teacher. For practical purposes the implementation *does* need to be taken into account, a recurrent theme in this thesis.

## 7.3 Identifying errors

This section illustrates the potential of `hint` to locate and understand errors. The first example shows an invocation of `hint`'s `Error` value. The second example exposes a wrong definition.

### 7.3.1 Use of the `Error` value

The existence in `h` of an `Error` value is useful in locating errors. Unlike an application of `error` in Haskell, the creation of such a node does not stop the reduction, so it may be seen, normally being involved in the creation of other error nodes when it is found to be an inappropriate argument or whatever. Here is a tiny example, where `foldr` has been wrongly defined as in Figure 7.6. The recursive call to `folde` omits `f` as the first argument. When

```
folde f z xs = case xs of
                  []    -> z
                  (h:t) -> folde' f z
folde' f z h t = f h (folde z t)
```

Figure 7.6: An erroneous definition of `foldr`.

this is applied in the expression `folde plus 0 [1,2,3]` it yields an error value shown in Figure 7.7. In the preceding step, `folde` is applied to only two arguments, so is regarded by `hint` as a partial application: an inappropriate argument to the + primitive.

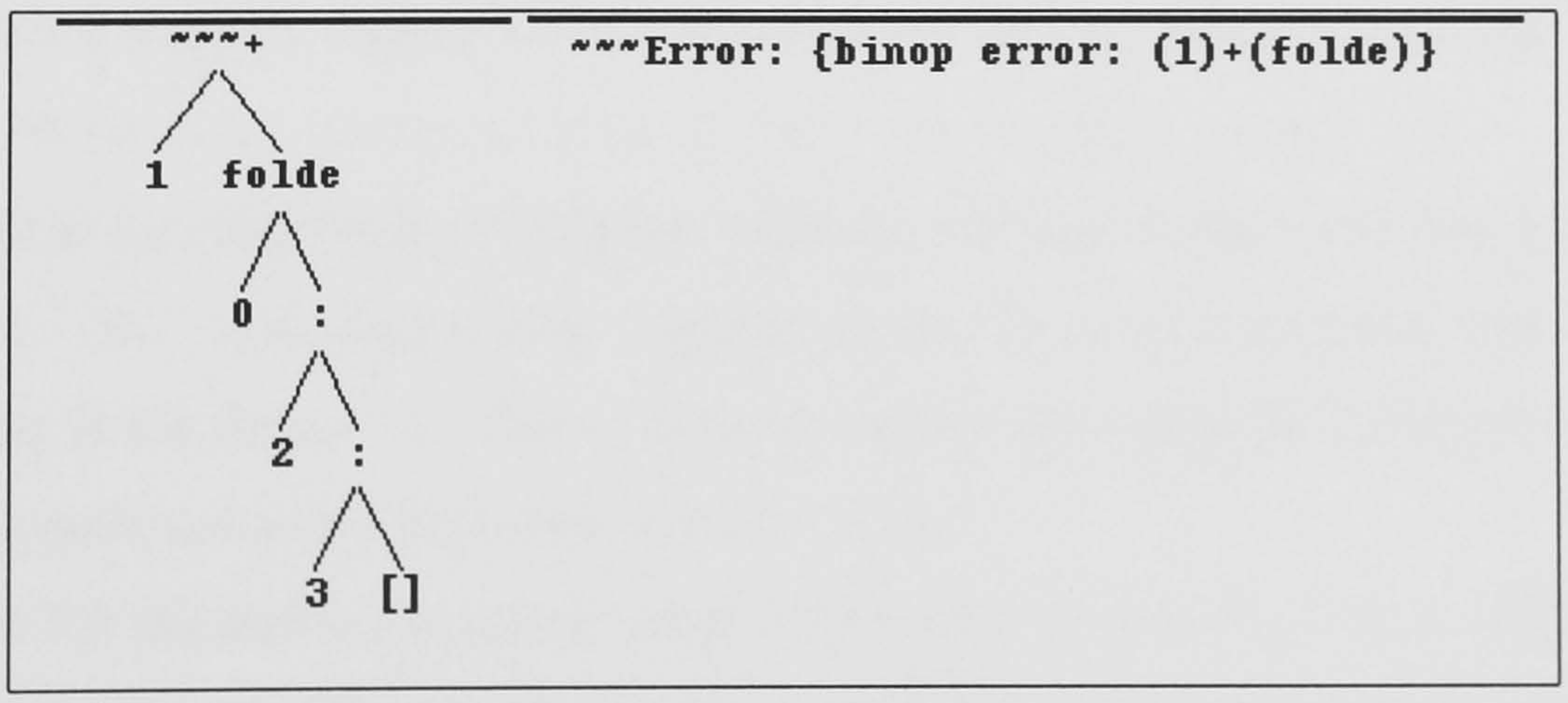



Figure 7.7: The error message preceded by the step before.

### 7.3.2 Locating a semantic error

If hint had a typechecker, an error such as that described above would have been caught before the function could be applied. Indeed the main purpose of the Error value is to prevent such type errors from crashing the system. The system may, however, also help the user detect semantic errors in a type correct function definition. Figure 7.8 gives a definition of

```
mt tree          = case tree of
                      (L n)      -> id
                      (B t1 t2) -> mt'
mt' t1 t2        = min (mt t1) (mt t2)
min n1 n2        = if n1 < n2 then n2 else n1
allnew n tree    = case tree of
                      (L v)      -> toleaf n
                      (B t1 t2) -> totree n
toleaf n         = L n
totree n t1 t2 = B (allnew n t1) (allnew n t2)
mintree tree     = allnew (mt tree) tree
```

Figure 7.8: Definition of mintree.

mintree, a function to replace a binary tree of integers by another of the same shape but with every leaf replaced by one containing the smallest value of the leaves of the original tree. The (user implied) type of the tree is:

`data BT = L Int | B BT BT`

The error is in the definition of min which returns the wrong (greater) value.

Figure 7.9 illustrates stages in the reduction of an expression, under the NOAPPLY filter, that expose the mistake. Figure 7.9 (**a**) shows, on the left, the initial expression:

`mintree (B (B (L 1) (L 10)) (B (L 2) (B (L 9) (B (L 5) (L 6)))))`

On the right is the stage in the evaluation where the left part of the result tree is about to be output. The --B-- is an output node representing the top level tree constructor. The value in the leaves is the shared 10. This is about to replace the values in the leaves of the right part of the result tree as the first argument to allnew.

Figures 7.9 (**b**) and (**c**) represent steps intermediate to the two others. Figure 7.9 (**b**) shows the calculation of the minimum value, initiated by the need to output the value at the first leaf. At this point the value is shared, but in order to reach the final result the tree will in fact have to be traversed twice: once to find the minimum value, then again to propagate this through the tree. This is because h, lacking local definitions, cannot express circular definitions [12], where part of the result is used in the calculation of that result.

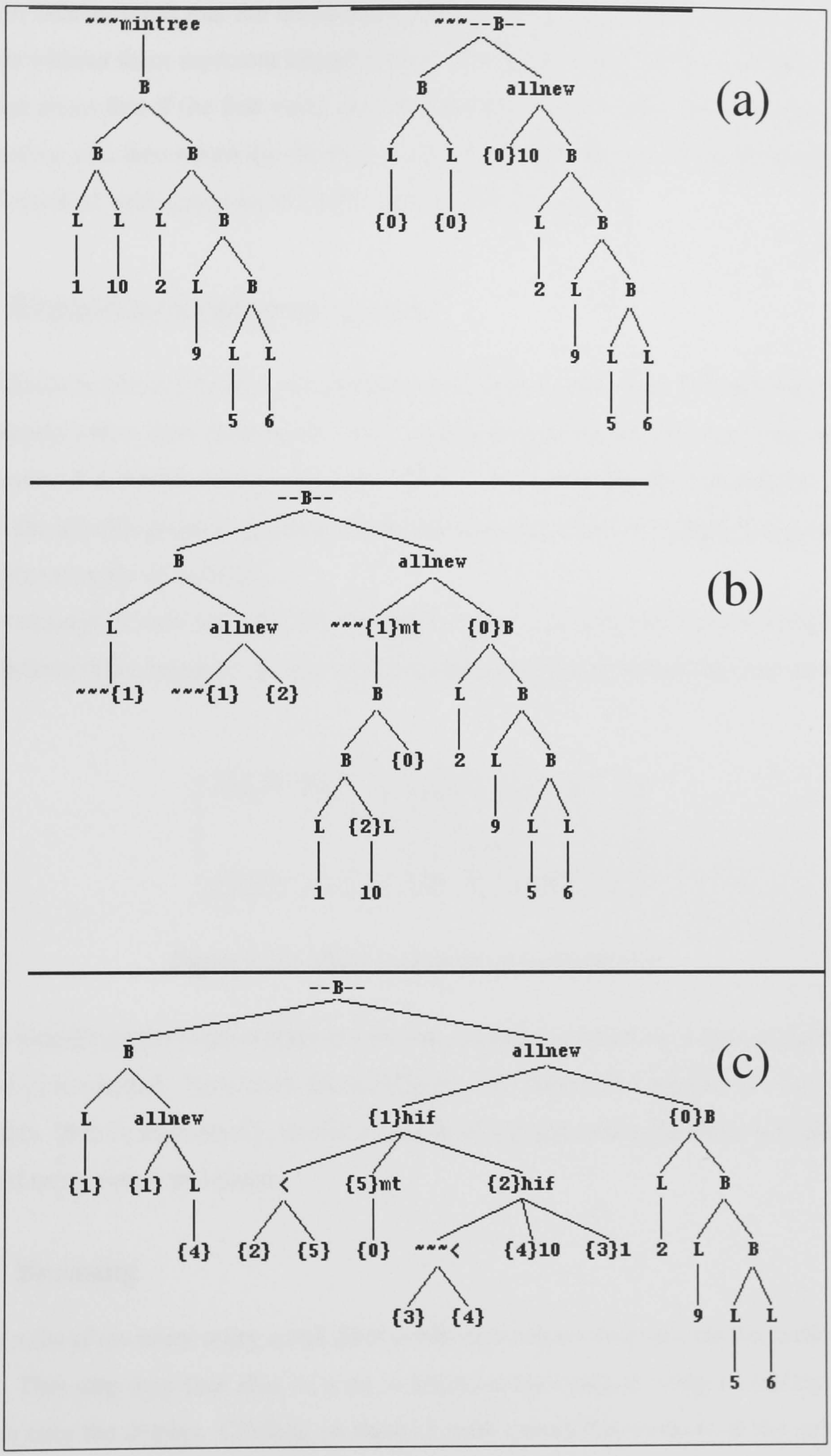



Figure 7.9: Error in `mintree`.

Figure 7.9 (c) exposes the erroneous definition. In interpreting the diagram it is necessary to bear in mind that the numbers in curly brackets are display references whereas numbers without these represent integer values. The lower of the two `hif`, conditional, expressions states that if the first value ({3} representing `1`) is smaller than the second ({4} representing `10`), then return the second (`10`) otherwise the first (`1`). Thus the wrong value will be returned, and `mintree` has been defined as `maxtree`.

## 7.4 Exploring a program graph

The example in this section does not exhibit a space *fault* as such, but illustrates that thinking declaratively rather than procedurally may mislead the programmer into ignoring essential space costs. It is further used to show the effect of browsing, and the tailoring of a spatial filter to the specific graph in question to achieve a similar effect by compacting parts of the graph not currently of interest.

The example is insertion sort. The definition of an `isort` function is given in Figure 7.10. The intuition of inserting the head of a list into the sorted tail of the list does not necessarily

```
isort list = case list of
           [] -> []
           _  -> isort'
isort' h t = ins h (isort t)
```

Figure 7.10: The h definition of insertion sort.

involve visualizing the implementation waiting to pattern match at every item in the list until the final [ ] is reached. Yet it must do, and Figure 7.11 shows how it looks in `hint` (with a null filter). This is, incidentally, another example of a graph where planarity may almost be achieved using pencil and paper.

### 7.4.1 Browsing

Sorting a list of six items using a null filter produces a display that runs off the main display screen. This step may thus also be used to illustrate browsing to bring a missing section of graph onto the display. Clicking on the `hif` node causes this to move to the root of the display, revealing the nature of what were, in the original display, *stubs*: the double circles indicating that there is more. The browsed version is shown in Figure 7.12.

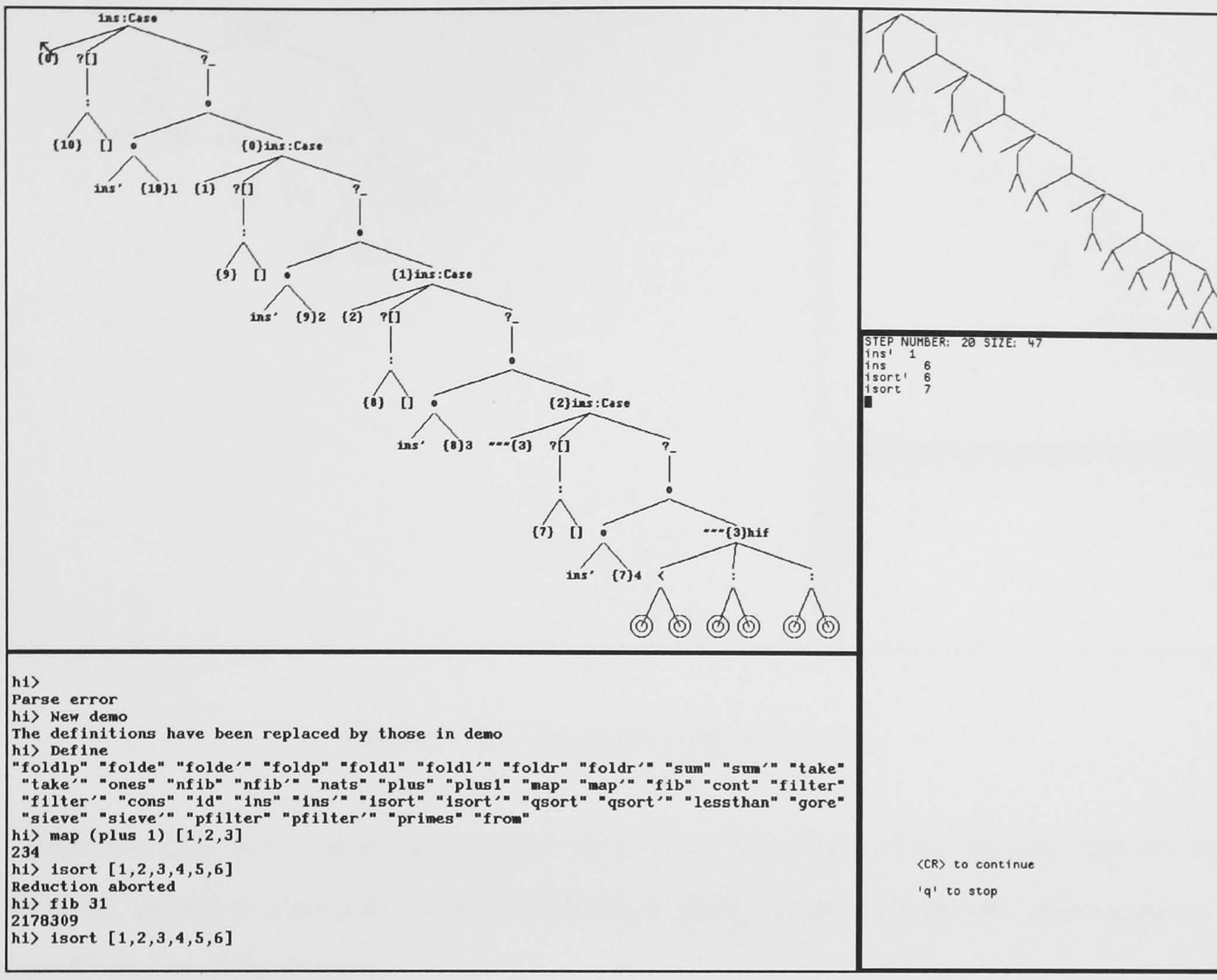



Figure 7.11: The `isort` graph.

### 7.4.2 Tailoring the compaction

Figure 7.13 illustrates how spatial filtering can be used here, instead of browsing, to compact the graph so that the part that was off the display can be seen.

The raw graph in Figure 7.11 reveals a pattern of linked trees, each with a `Case` node at the top resulting from the application of the `ins` function (`ins: Case`). This is the effect of the pattern matching on the tail of the list at each list element. In order to see the part of the graph currently off the display we would like to compact these trees, but to leave the `hif` structure at the bottom in full display. We leave also the `Case` nodes themselves in order to keep an outline of the graph-tree structure. So the rule is that a node is collapsed with its display parent if it is a descendant of a `Case` node, but not itself a `Case` node:

```
CASE = someancs (is Case) && not (is Case)
```

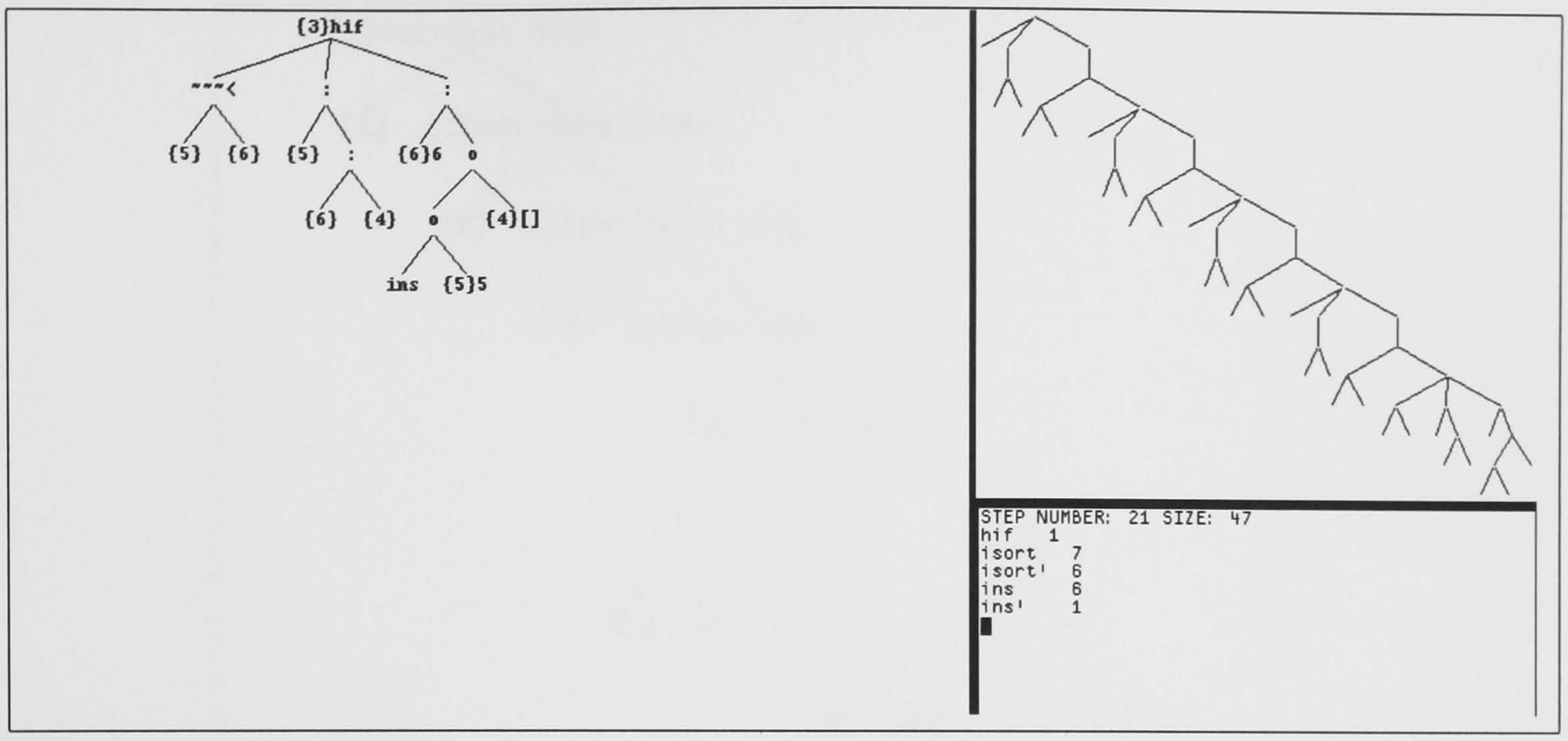



Figure 7.12: The browsing of `isort`.

However this would also collapse the section at the bottom of the display that we wish to observe, as this is also a descendant of a `Case` node, so we exclude the "next node to be reduced" and its descendants:

```
CASE = someancs (is Case) &&
       not (is Case || is Focus || someancs (is Focus)
```

That is the compaction rule. Now what is to be the display rule? Single node clusters are to be displayed as the node they represent, unless they are display leaves in which case we would like to see the display reference only. The `whiff` primitive for this is `nodenl` (see Section 7.5). Clusters that are larger than one node are those deriving from the `CASE` rule, and will be labeled with the `Case` node at their root. Thus the `join` function discards nodes below, and shows the node at the "join":

```
JCASE _ = node
```

The final formatting function might be required to: mark the next node to be reduced, `sfocus`; show any display references, `sref`; and, if the size, `vsize`, is greater than 1, put the letter "S" followed by the size, `ssize`, of the cluster:

```
DCASE ct = sfocus ++ sref ++ ct ++
(if vsize > 1 then (lit " S:") ++ ssize else "")
```

The display function is thus:

```
SHOWCASE = u nodenl j JCASE d DCASE
```

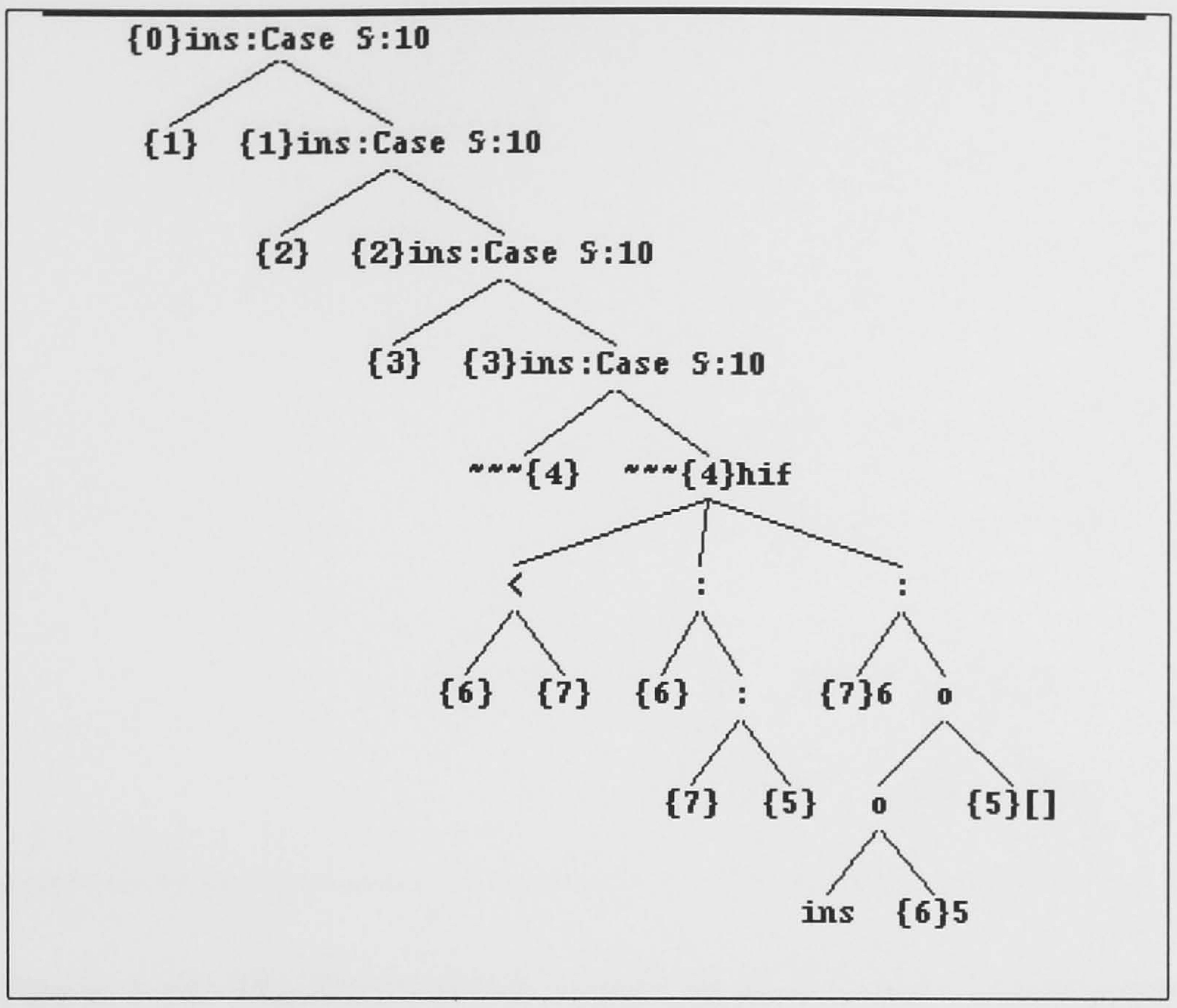



Figure 7.13: The NOCASE filter applied to the `isort` graph.

And the complete spatial filter is defined as:

```
NOCASE = (CASE,SHOWCASE)
```

A further refinement might be to compose the CASE filter with the NOAPPLY filter, which in this display would get rid of the two `Apply` nodes in the expression `ins 5 []` at the bottom right of Figure 7.13.

```
NCA = CASE || NOAPPLY
```

However the labeling of a cluster then needs to be related to the rule the application of which created it. Using the SHOWCASE labeling scheme here would result in the topmost `Apply` node being shown, rather than the name of the function being applied. The unit (`u`) and display (`d`) functions are the same for both kinds of cluster, but the join (`j`) function has to take account of the node at "the join".

```
JNEW lss = if is Apply then head lss else node
```

The labeling function is then:

```
SHOWNCA = u nodenl j JNEW d DCASE
```

and the new spatial filter:

```
NOCASEAPPLY = (NCA,SHOWNCA)
```

The effect of applying this composite spatial filter is shown in Figure 7.14.

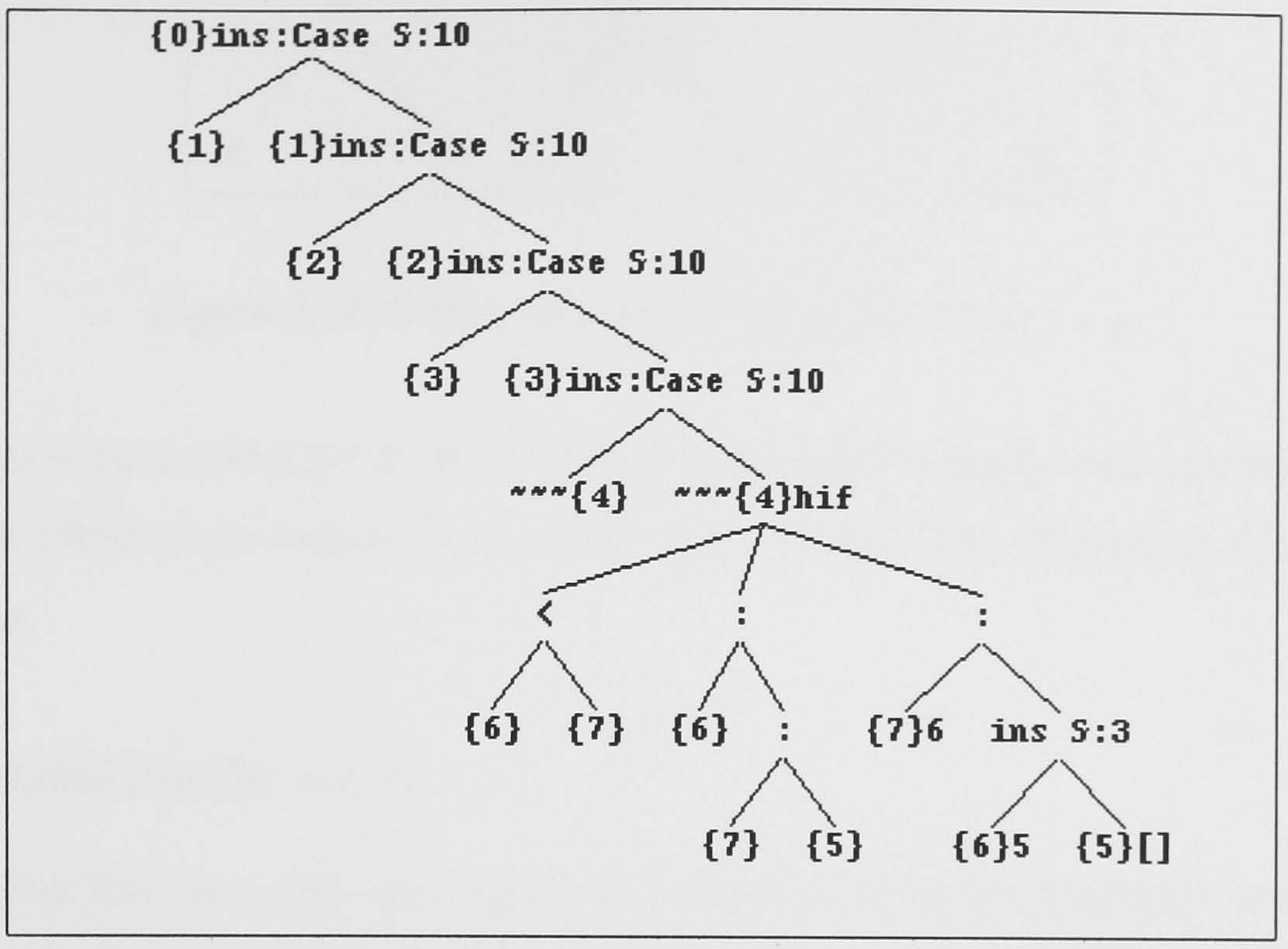



Figure 7.14: The NOCASEAPPLY filter applied to the `isort` graph.

## 7.5 The problem of labeling

Spatial filtering has been introduced both to tailor the compaction of raw program graphs, as illustrated in Section 7.4, and to present distinctive views of the graph, giving the user insight into the reduction process. Such views are largely characterised by the labeling scheme in use. This section discusses some of the problems encountered in labeling, and solutions found, or proposed, for overcoming these. There is discussion of:

- the display of `Apply` nodes;
- the need for *two* show primitives for `whiff`;
- the indication of sharing;
- the marking of the focus of reduction;
- strictification of sections of the graph;
- and some other possibilities for the display of labels.

### The display of `Apply` nodes

In general the presentation of single node clusters is clear. An exception is the `Apply` nodes, often shown as `@` in the literature, and in `hint` "simplified" to `o`. Arguably the application should be conveyed solely by the articulation of the graph, just as in the textual form application is denoted by juxtaposition. Figure 7.15 illustrates just how much more straightforward and uncluttered is the third version, where this is the case.

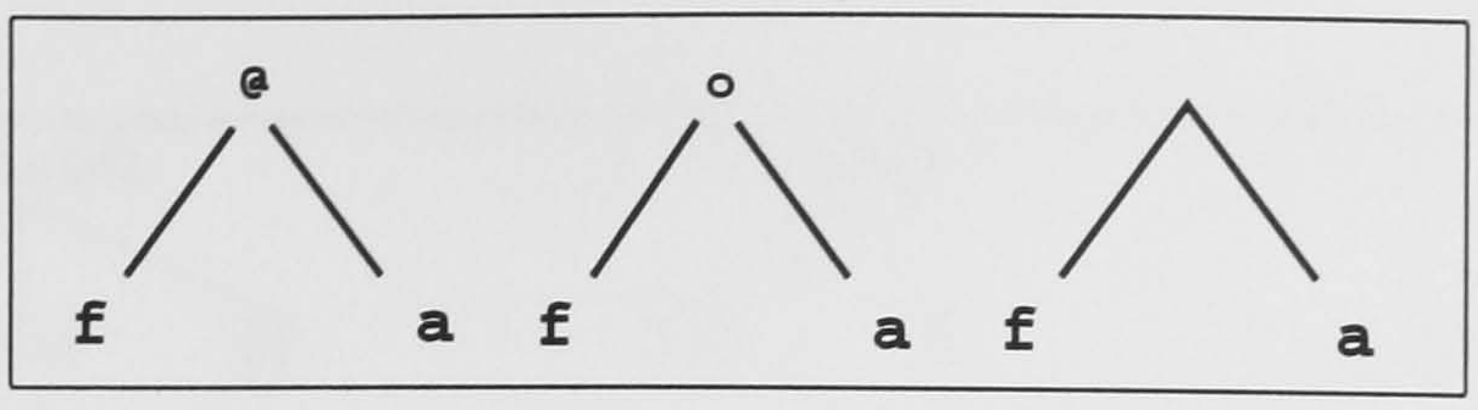



Figure 7.15: Three versions of the apply node in `f a`.

This may seem a minor point, but in a large and complex display such niceties may have a significant effect in the reduction of clutter on the screen. This makes the task of the observer easier.

**Two `show` primitives for `whiff`**

The `show` function for nodes may represent a display leaf in the graph tree as its ultimate referent. For example a display leaf that refers to an `Int 2` node it may simply be labeled `2`. Alternatively a display leaf may be left as an empty string (see Figures 5.15 and 5.16 in Chapter 5). The advantage of *not* displaying the node's value is that there is no redundant display information, thus again reducing "noise" in the display of the overall structure. In those examples there is a display reference which identifies the ultimate referent. But there are also advantages in showing the value of a display leaf: not only does the viewer know that this node is shared, but also what it represents. This saves effort if the ultimate referent is not instantiated on the current display.

In fact `whiff` offers *two* show primitives to the user, `node` and `nodenl` (node, but **n**ot display **l**eaves), so that the more appropriate version may be chosen.

However, this choice may have further implications. For example where a function node is shared under the `NA` filter (No Apply), it is not necessarily the root of the cluster, so using one-to-one references the display reference will not appear. If, in addition, it is a display leaf, and the labeling scheme join function transfers the label of the leftmost node at each junction, an empty string will appear instead of the function name. Figure 7.16 shows three versions of a graph in the reduction of `foldl plus 0 [1,2,3,4]` that illustrate the effect of various combinations of node labeling and final display functions.

In Figure 7.16 **(a)** the `node` show primitive is used. Because of this, both of the shared `plus` nodes are labeled with the function name. In fact the other `plus` nodes are also shared (see Figure 7.4). Where the shared function node is not a display leaf, however, it is also not the root of the cluster of which it forms a part, so its display reference is not available. This display of sharing is further misleading in that it is `plus`, not `plus 0 1`, that is shared

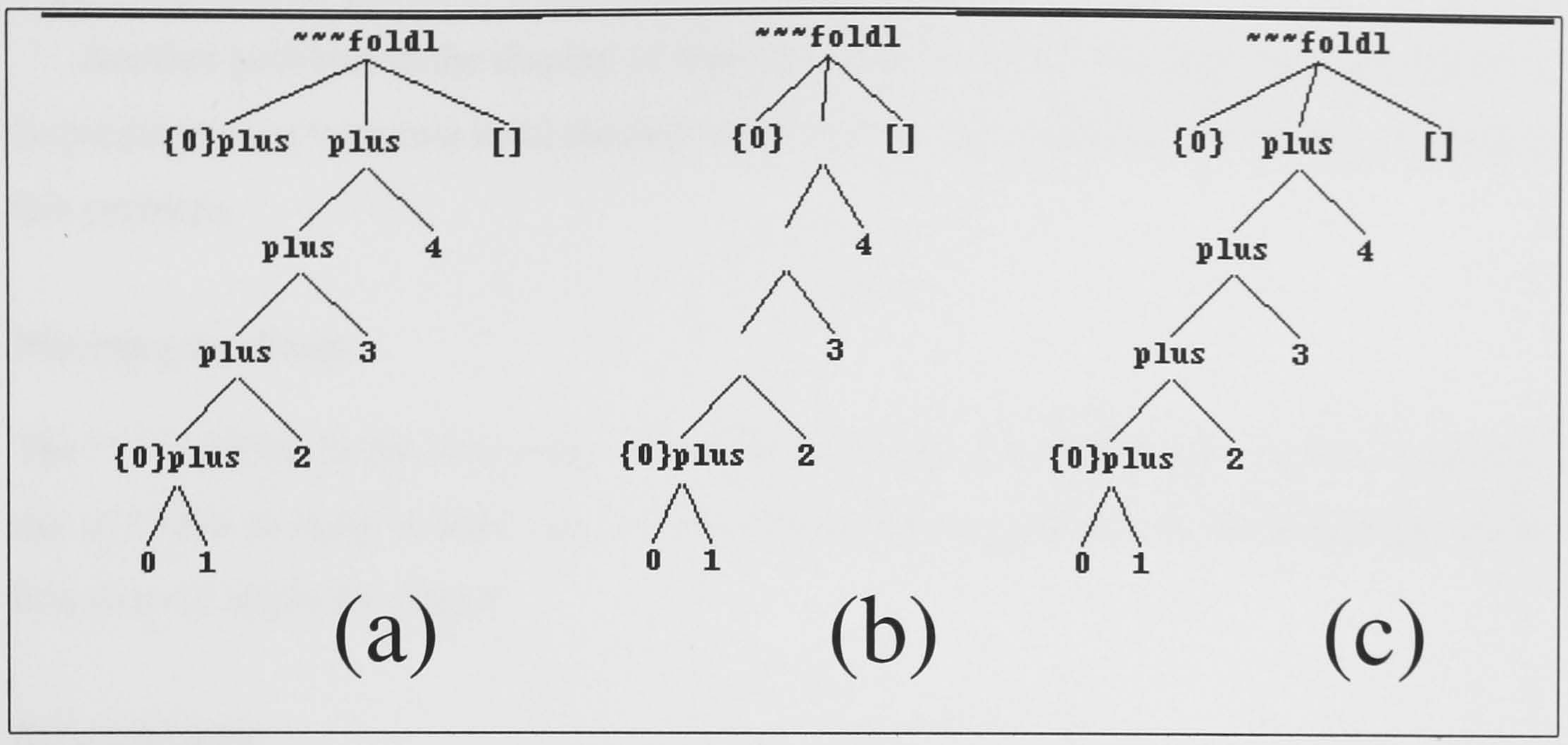



Figure 7.16: The result of various labels for the NOAPPLY filter.

(again, see Figure 7.4). This particular node is labeled somewhat arbitrarily with the display reference because, according to the display algorithm, it is the *instantiation* in the graph-tree of the function node. Whenever *any* spatial filter is applied, the display of sharing can become confused in this manner.

In Figure 7.16 (**b**) the `nodenl` show primitive is used. This gets rid of the duplication of the label in the display leaf for the first argument to `foldl`. But it also results in an empty string being propagated to the level of cluster label, leaving blanks in the display. This is because the intermediate `plus` nodes are display leaves but not single node clusters.

Figure 7.16 (**c**) gives the most satisfactory picture. Here the `node` version of the show primitive is used, so display leaves have their value reflected in the `unit` label. But now the `display` function removes the name from display leaves:

```
DNA ct = sfocus ++ sref ++ (if is Dleaf then "" else id ct)
```

### Showing display references

In `hint`, sharing of clusters is shown as a number in curly brackets. These curly bracketed references can be hard to distinguish from other labels: when the referent is itself an integer it is difficult to tell at a glance which is the value and which the referent. Possibilities for overcoming this include representing the reference in a different font or in inverse video, or placing it differently in relation to the node. Tufte writes, in a slightly different context:

> "...color effortlessly differentiates between annotation and annotated" [90]

suggesting another possible solution, given a suitable environment.

Another problem in the display of sharing is that shared nodes may be within clusters. Experimentation with two level sharing labels has not yet yielded a satisfactory solution to this problem.

### Marking the focus

The ~~~ marking of the next node to be reduced is also not entirely satisfactory. Again the use of colour to make it stand out, or even to have a flashing node (or flashing nodes where it is shared) might be clearer.

### Strictification

The `ARITH` filter, illustrated in Figure 5.16 in Chapter 5, suggests that a *strictification* primitive might be usefully provided by `whiff`. According to the `ARI` compaction rule involved (see page 92), clusters consist entirely of arithmetic operators and integer nodes, or are single node clusters, so could take advantage of such a primitive – perhaps displaying a cluster as the result of its evaluation.

The values obtained from fully reduced clusters might be used not only in the labeling, but in determining checkpoint criteria. The user would need to beware of the possibility of non termination, though perhaps the system could help with this by, for example, keeping track of the number of reductions and abandoning the attempt to strictify if necessary. The primitive would then effectively be: "strictify if possible within a limited number of reductions". Another problem would be that large examples might take noticeable time to process the necessary calculations: using the current scheme, the whole graph needs to be compacted and labeled for the `x` positions to be allocated to the graph tree – even regions of the graph that are not initially displayed.

In a sense this strictification would be a formatting primitive. Other, simpler, such primitives might, for example, display a list of characters as a string, and a list of other elements using the `h` syntax of square brackets and commas.

### Some other possibilities

Here are some other possibilities for the display of labels.

**Richer labels** A more general algorithm, in the spirit of those presented by Bloesch [14], could display labels of varying depth. This could allow clearer differentiation between elements of a textual label.

Using `ASCII` text for labels limits the flexibility of the display. If labels of different depths are allowed this opens the way to using pictorial representations, for example where filters of different kinds are composed characterising the rule that produced a particular cluster.

**Miniature graph-trees** Each cluster could be displayed as a miniature version of the graph-tree it represents. This could be combined with an option to expand a cluster to its constituent graph-tree in place, or to display it elsewhere on the screen such as in the minidisplay window.

**The use of colour** The use of colour has already been mentioned. Its potential for structuring the display is much more than for helping the viewer decipher display labels. For example the age of clusters could be expressed on a scale of darkness in a particular hue. Labels on display leaves could be in a characteristic colour to make the distinction obvious. Different fields of cluster labels could each have a distinguishing colour.

**Display the graph as a graph?** An assumption in the design and implementation of `hint` has been that the display of the raw program graph, or of a compacted version of this, would be too complex to decipher. This could be mistaken. It may be that an ideal system by default displays the graph rather than a graph-tree, with the option of converting to a graph-tree as required. Techniques for isolating parts of the graph for further exploration could be devised. The main problem with this, apart from the display considerations, would be the relative complexity of defining compaction rules: no longer can a node be assumed to have but one parent.

The display of the graph introduces the problem that graph-trees avoid: the crossing of arcs. Conversely it avoids the problems of graph-trees: the need for display references, and an increased number of nodes to display.

## 7.6 Limitations of the system

Here are some of the limitations of the `hint` system. This includes both restrictions due to the specific implementation of the prototype, and others to do with the general approach taken.

### Size

A future `hint` may well be used with large examples to help solve, in particular, the problem of locating the source of mysterious space leaks. Indeed during the development of the system itself such insight would have been invaluable. However the prototype has not yet been used with computations involving more than around 30,000 steps, mainly because of its own exemplary space leak. One might observe that if a program such as `hint` itself could easily be made to behave properly using existing tools, then a `hint`-like tool would be unnecessary anyway.

At present the system cannot deal with other than quite small examples, because of the space leak. This is not an insurmountable problem, however. In principle a similar system would be able to apply the power of the compaction, labeling and checkpointing scheme that has been implemented in `whiff` to any size of graph.

As the space characteristics of a target program do not interfere with performance in small examples, the `hint` system so far has been used to demonstrate these, but not to identify problems relating to space usage. However this is the area where such a system might well be most useful from a practical point of view.

### Sharing

The problems of displaying sharing in a compacted graph have been described above. Another aspect of sharing that might usefully be included is a *sharing index*. It is not easy to find the appropriate definition of this. It would be something along the lines of weighting shared nodes with the number of descendant nodes, and comparing this with the total number of nodes. But the possibility of direct and indirect cycling in the graph complicates this. As sharing of nodes suggests less space usage, and less reductions, it may be that a high sharing index would denote an efficient program. Given a suitable index, `hint` could keep track of it, though such a facility is not built into the system. However, in defining cluster labels, the user may take the number and proportion of display leaves into account.

### Profiling

Unlike heap profiling, the `hint` system does not offer statistics relating to the program graph, nor does it give diagrammatic summaries of the constitution of the graph. In theory it could, as it has the raw information explicitly available. The aim of the system in the context of the thesis has been to give insight into the reduction process by displaying a (simplified) view of the graph. In a sense though, pictures used to display statistics about the graph are giving yet another view of the structure, since heap profiles also provide a (highly) simplified view of the graph. Heap profiles amalgamate information from disparate regions of the graph, whereas compaction rules in `hint` compel the user only to collapse together regions of the graph that are adjacent. However the *labeling* of a cluster may involve analysis of the cluster's constitution, so, in `hint` terms, heap profiling corresponds to a special labeling of a completely compacted graph. It may be that a `hint`-like system would ideally offer both sorts of view, and a facility for switching between them.

## 7.7 Summary

This chapter illustrates the use of hint, discusses some of its limitations and gives some ideas for its potential development.

The display of the program graph as a teaching aid may reinforce concepts, such as the effects of reduction order, by giving the additional visual dimension. To some extent the use of `hint` is an extension of existing practices, and offers an animation of text book-like diagrams. Errors in definitions may be pinpointed by detailed observation of the reduction in action.

Space characteristics of a program graph may be explored using spatial filters, and the progress of a reduction may be monitored using temporal filters. An example is given of a spatial filter tailored to the compaction of the display of a particular program graph. In principle the `hint` system has the power to compact very large graphs, though in practice this has not been achieved because of a space leak in its own implementation.

The usefulness of a compaction scheme lies mainly in the ability of cluster labels to convey the needed information. The `hint` system offers a very flexible labeling scheme, but further possibilities are proposed. These include additional labeling primitives to be available to the user, and alternative labeling schemes involving graphics and colour, rather than the existing one line, black and white, ASCII labels.

In contrast with heap profiling, the `hint` scheme is designed to enable and assist the user to gain a detailed view of the articulation of the program graph. Indeed one of the aims was to provide the sorts of view that heap profiling could not give. It does not, in its present form, provide summaries of aspects of the whole graph in graphical format. In principle, though, this would be possible, through novel schemes of labeling a completely compacted graph.

# Chapter 8

# Conclusions and future work

## 8.1 Introduction

The previous chapter on the use of `hint` shows, amongst other things, how such a system may help programmers understand the space characteristics of their programs. This brings us back to the questions in the introduction:

Using current implementations can we provide evidence that this style of programming is viable? Well, two medium sized implementations are used to illustrate this thesis that are themselves evidence. And they illustrate as intended the two aspects of "See how they run": the performance of the Escher program is discussed in Chapter 3; and the `hint` environment allows the programmer to watch the program reducing.

What information does the programmer need in order to write efficient programs? Moreover, does an environment such as `hint` provide this information? Here the evidence is more confused. Certainly `hint` shows the reduction in a novel way, and enables information about the reduction to be compacted in a meaningful way. But it is not certain that this is sufficient basis for the programmer to program efficiently.

This chapter discusses the possibility that the programmer *must* take the implementation into account: Section 8.2. There is then a discussion of the potential usefulness of a system such as `hint`, and of possible future developments: Section 8.3. In Section 8.4 the Escher program is revisited, and its structure reviewed in the light of the subsequent work on `hint`. Finally Section 8.5 concludes and ties together the various strands of the thesis.

## 8.2 It's a lie!

It's a lie, of course. You have to take the implementation into account. "This style won't 'work' because there'll be a space leak ...Oh no, it's alright, because the compiler we're using has a 'Sparud' option" [81]; and debugging is hard: what functional programmer has not encountered "Fail: head [ ]" and reacted either with "Oh bother I forgot to account for ..." or worse "Where on earth...?" This is an analogous to the situation with the heap profiler when (++) is seen to be both creating and taking up an inordinate amount of space. Here the user may define their own append for every module in order to isolate the one creating the problem, but this is extremely tedious.

And what about the reputed conceptual clarity of functional programming? The simplicity of functional programming, the directness of thinking always in terms of function argument and result were not always so obviously appealing. Many writers of functional programming theses of seven or eight years ago felt obliged to offer an introductory section to explain basic concepts of functional programming. Now that these may be assumed new apparent complications arise. For example when referring to a lazy system one may glibly mention that a function returns "the input that it has yet to receive", or that it "makes use of part of the final result in creating that result"; and in the realm of I/O monads we talk about a "World" on which actions may be made without compromising the referential transparency of the program that does not even have to mention it. Logically these too are simple concepts, but intuitively they are so incongruous as to create a psychological barrier towards systems that involve them.

In practice too, no way is the use of lazy functional programming the concise, clear, expressive medium that I would like to make it out to be. I have spent hours, nay weeks, chasing space leaks. The structure of the Escher program that does indeed nicely reflect its specification was only reached through the most tortuous routes. This resulted mainly from inherent problems in the nature of the style. The very aspects of lazy functional programming that make it so attractive: the lack of need to be concerned with memory management, the possibility of compactly defining functions, the blissful ignorance of order of evaluation — each has a corresponding, and potentially lethal, drawback. Without direct control of memory allocation the programmer cannot be sure that the program is going to behave "properly": there may be chains of partially evaluated expressions and the laziness of the system ensures that they *do not* get fully reduced unnecessarily, and through this means the program may run out of memory; functions may be neatly composed — yet the resulting system may create closure chains so that the effect is not at all as "neat" in terms of performance as expected; the order of evaluation actually occurring may be such that the programmer is misled by the "strict" thinking that the declarative style encourages that reductions will take place "as written", whereas in fact they may not. The programmer regards formulae as being equivalent to the values to which they (may) reduce; in terms of absolute meaning they are, but in a lazy system the reduction will only take place if the result of the reduction is needed. The conception of an expression as the result of its evaluation may be useful in grasping the essence of a function definition. But as discussed in Chapter 5 this may lead the programmer to imagine that an expression is reduced when it may well not be. Conversely, on occasion the result of evaluation may take up more space than the redex from which it arose, so the delay of an evaluation is sometimes a good thing from the space point of view.

It appears that the programmer needs to take the implementation into account, but may do this through having an appropriate mental model rather than a detailed knowledge of the low level processes involved. Even with awareness of implementation details, the programmer needs to think in higher level terms. The mental model may be used as a yardstick to assess the practicality of a particular approach to a function definition. It is important that the mental model be not misleading, hence the need in Chapter 5 to justify the use of `h` and its implementation using simple graph reduction and template instantiation as being relevant to "real" *Haskell* implementations. It is obviously relevant to the `h` implementation as it reflects it directly. In fact textbook presentation of functional programming to the programmer is also usually at this level of abstraction, precisely for the same reasons that are used to justify its use in `hint`: that it enables the reduction process to be seen in source level terms, and that it offers a view of the reduction that is compatible, for example, with actual implementations using supercombinators.

## 8.3 `hint` to assuage the lie?

Given that `hint` offers a view of program reduction at an appropriate level, various questions arise:

1. Can the use of a monitoring system such as `hint` bridge the gap between the need to take the implementation into account, and the desire to program at a level that does not need to?
2. How far does the prototype system go towards this?

3. What more could it do to give the programmer the required insight into the reduction?

4. If such a system is worthwhile, what problems are envisaged in developing a version to handle full blown Haskell?

### 8.3.1 Bridging the gap

The solution to having the advantages of lazy functional programming without the disadvantages is to have just as much control as is necessary over the elements that one would ideally prefer not to have to consider. A monitoring system, as such, is evidently not sufficient to allow the programmer to assume control of any aspect of the reduction. A system such as `hint` does have the advantage over one that displays the information about the graph in purely statistical terms. The programmer may in small examples trace what is going on in the reduction, for example the occurrence of an error with its appropriate error message may be noted. In larger examples he may summarise the graph in different ways in order to explore its structure in more detail.

However if an environment *did* offer options for the user to have some control over the reduction process, it could provide the best of both worlds. In general, as discussed in Chapter 4, reducing the amount of memory needed by the program also reduces the time it takes to run as there is less memory management overhead. The main problems are, then, those that cause too much space to be needed. As illustrated in the heap profiling work [74] this may be caused by *too little* as well as by *too much* laziness. In the second case, at least, strictness annotation may offer a cure. There is also a case for strictness *declaration* where the programmer would indicate the expected strictness of a function application, which could then be checked along with the type of the function. Another situation where there may be excessive space usage, as mentioned above, is one where the value of a reduced expression takes up more room than the redex from which it was derived. Here the programmer might wish an option to cause it to revert to its unevaluated state [92]. However this may only be necessary when such a value needs to be present for a relatively long time after its creation, and is, moreover, either not needed in its evaluated form during that time, or easily reconverted.

### 8.3.2 Limitations of the prototype

The prototype `hint` does not give the user any control over the reduction. It does, though, allow the reduction process engendered by different versions of a function to be observed so that the apparently more efficient may be chosen. As it notes the age of nodes it also has the potential to keep track of the age of evaluated nodes, so that if there were an option to cause them to revert to the unevaluated form, the age could be used as the criterion for when to do this. The idea of the monitor as having an overview of the computation, and the facility to step *backwards* within it, suggests that it might be possible to use this to recreate the expression from which the result was derived. The prototype in fact does not have this facility, so this is one urgently needed next step. The provision of spatial filters in `hint` offers options to compact the graph in a flexible way so that even when the graph is large the user of the system may, through different views of a particular reduction step, reach an understanding of the graph's composition.

### 8.3.3 Potential development

Despite the limitations, the facilities incorporated in the prototype `hint` are sufficient to illustrate the points made in the thesis. For it to be developed into a more generally useful tool various changes and improvements are envisaged. These may be grouped into:

1. planned adjustments/improvements to the existing system;
2. further ideas for the ideal system.

#### 1. Towards a more sophisticated prototype

#### Stepping

Although it is possible in the prototype to move from one reduction step to the next, or from one checkpoint to the next according to the current temporal filter, it is not possible to step back, nor to define checkpoints such as "The step where this node ceases to be part of the graph". One would like to have a much more flexible mechanism for investigating the reduction, analogous perhaps to incorporating Snyder's "reduction-history space" [80]. Facilities such as stepping back to the creation of a particular node, or to an instance of a particular application chain might help the programmer find out what is going on. Keeping a particular address at the root of the display, rather than invariably placing the root of the graph there is another technique that might be worth exploring. As any particular node is not guaranteed to be present from one step to the next, some default, such as returning to the root of the program graph, would be needed to allow for this.

The enhancements suggested are: to introduce stepping back as well as stepping forwards, and to offer extra primitives for the description of checkpoints.

### Browsing

Even with spatial filtering in place, graphs will be large when `hint` is used with bigger programs. This is particularly because, unlike the case with heap profiling, similar clusters in the graph are not merged — it is not a sorting of the graph into statistically related elements, but a partitioning of the graph. This is deliberate, with the intention of maintaining the relevant articulation of the graph, so giving the user a view of the reduction state that may enable them to understand the process better.

But without a browsing facility this exercise is very limited. Exploring the program graph is analogous to exploring the reduction space: there is a lot of detailed information that may or may not be relevant. The aim is to help users identify and isolate the sections that will give the them insight into the process. Partly this is done through the filtering. Spatial filtering collapses together patches of the graph the detailed structure of which is irrelevant to a particular view of the graph. Temporal filtering similarly collapses together stretches of the reduction that are not of interest to a particular view of the reduction. But just as stepping is a vital element in exploring the reduction, browsing is a vital element in exploring a particular program graph.

So in the ideal `hint` users should be able to move around freely in the graph, jumping from display leaves to their referents, seeing what lies beyond stubs, opening up clusters.

### 2. The ideal `hint`

Further development of a system like `hint` would undoubtedly be worthwhile — both for teaching and for helping programmers understand their programs better, whether they could directly influence the reduction process or merely affect the program behaviour through the functional code. This section discusses some of the features that were either rejected from the prototype as not being essential for the thesis, or that were beyond its scope for other reasons. For example the use of colour, while potentially a great asset to a practical `hint`, is not vital to the argument that presenting the reduction in the `hint` style can be of help to the programmer.

### Instantiation

The arbitrary instantiation of nodes according to the graph tree creation algorithm can create problems in the display: for example a patch of interconnected graph may become widely dispersed and its structure effectively lost to the viewer. The ideal `hint` would offer solutions to this. Possibilities include allowing the user to:

- change the spanning tree;
- change the instantiation of a particular node;
- click on a node and be alerted to all its referents, perhaps by flashing;
- selectively join display leaves to the clusters that they represent.

### Colour

The use of colour in labeling might remove some of the confusion which labeling with an ASCII string currently causes. Other possibilities for labeling are discussed in Chapter 7 Section 7.5, in particular reflecting age bands by colour. This would mainly be of use when the active spatial filter includes age amongst its criteria as otherwise there is no reason to expect nodes in a particular cluster to be of similar age. Another example of potential use of colour is to differentiate between the names of *producer* and *consumer* functions — the consumer function being the one the application of which is going to cause this node to become detached from the graph.

### Reduction mechanism

If the `hint` user could specify strictness through annotation of the original function definitions, or at run time, the environment would fulfil the requirements of Section 8.3.1. There could also be simulated parallelism — several parts of the graph being reduced simultaneously: an effective target for the viewing mechanism, but opening a new can of worms with its own problems, so that in the short to medium term this would be both counterproductive and hard to implement. A more practical option might be to have a strict version of `hint` with its own set of reduction rules that the user could switch to.

As many of the problems of lazy functional programming arise when I/O is involved, it would be good to monitor this. A very simple early prototype of `hint` had a miniature "screen". It allowed the strings to and from **MGR** to be observed, as lines and circles were drawn and deleted. In the context of the thesis work it was not appropriate to follow this up further, and using a window manager other than **MGR**, where the messages to and fro are already strings, would involve decoding of the messages to make them readable. Despite this I think it would be a worthwhile and revealing exercise.

#### Interrupt

It would sometimes be useful to interrupt an `h` computation in between checkpoints without crashing the environment, and possibly with the display of information about the reduction at the point of the interrupt. The question of implementing this is a separate problem, depending of course on the implementing language — for example LML's hiatons might be used.

### 8.3.4 A `hint` for Haskell?

Scaling up the system to include full blown Haskell would involve three main elements: type checking, local definitions, and conventional pattern matching. The monitoring would be optional so that the overhead it represents does not affect the normal running of the system. This is reminiscent of the Glide system [88] where the display of trace information involves a separate calculation to that used in the non-monitoring reduction. Type checking is well researched and would complicate the implementation, but should not be problematic.

#### Local definitions

Local definitions are really needed: for example *circular* programs cannot be investigated using the current `hint`, yet their very circularity would make this of interest. Name clashes could be overcome, even allowing for anonymous definitions. A simple solution, for example, would be to append the main function name and the local name: `fname.lname`. In the case of anonymous local definitions, they might be numbered as they occur in the text: `fname.1`, `fname.2` etc..

#### Pattern matching

The pattern matching should be displayed, as illustrated in the `isort` example in Chapter 7 on page 132. Here `Case` expressions cascade, each "waiting" for the resolution of the next one. The problem will be the translation of Haskell pattern matching to one that may be meaningfully displayed without overly complicating the display with pattern variables.

#### Strictness annotation and declaration

If the system is to allow strictness annotation and declaration, this too will need to be taken into account. Indulgent existing Haskell implementations such as the Chalmers `hbc` already feature strictness annotation as a pragmatic extension to the language. Strictness *declaration* could be checked along with the type declarations. Ideally both strictness annotation and declaration would be part of the standard language.

#### Window system

The use of **MGR** is ideal for the prototype system as it enables the interfacing to be a very minor part of the implementation — highly appropriate for a thesis that is not focusing on that aspect. The `hint` for Haskell must take into account the window systems that people tend to use. This suggests it should be implemented in X windows. As described in Chapter 2 there is a lot of current work on interfacing lazy functional languages to X windows that may be exploited here.

## 8.4 Escher revisited

What changes, if any, might be made to the design of the Escher program in the light of the implementation and use of `hint`?

### 8.4.1 Escher

The Escher program was originally written in Lazy ML, but translated into Haskell as implementations became available. The specification evolved along with the program, so rewriting would involve a more direct approach: the concept of the program as interface description may now be implemented directly, though the specification should first incorporate the changes proposed in Chapter 3. As mentioned in that chapter, the advent of heap profiling was exploited to locate and eliminate a space leak. However the use of `hint` has not so far provided insight that might be applied to the Escher program.

### 8.4.2 Interface interpretation in `hint`

On the other hand, the `interpret` function described in Chapter 3, together with a description of the interface in terms of active areas, could be applied to `hint`. There is scope for both fixed active areas such as buttons to regulate stepping in the control panel and dynamic active areas such as the location of particular clusters for use when browsing. As there are potentially a lot of displayed nodes, and each is associated with a view of the threaded displayable graph tree structure, it is important that the calculation of the interface is done lazily. As ever, strictness properties have to be given prominence!

## 8.5 Conclusion

We have seen that, in the context of a lazy functional language, the programmer may, to some extent and by devious means, control time/space factors without changing the implementation. Pragmatically, though, the programmer has to take the particular implementation into account. This suggests that monitoring systems that give a view of the reduction at an appropriate level of abstraction are likely to be invaluable. A prototype monitoring interpreter is used to explore various problems that arise in attempting to observe the reduction process, in particular size and complexity, as well as concern for authenticity. Solutions to these problems have been suggested (Chapter 5), implemented (Chapter 6), and demonstrated (Chapter 7).

The study took place in the context of an investigation into the pragmatics of writing interactive graphical applications in a lazy functional language. Two exemplars were used, one the monitoring interpreter itself, the other a graphical design program. Although in both cases the implementation process offered evidence of some of the problems inherent in the style, they nevertheless benefited from the use of a lazy functional language in their implementation.

The conclusions are that, even using current implementations, lazy functional languages are not only capable but well suited to writing interactive graphical applications. However the problems inherent in laziness need to be tackled by allowing strictness annotations and by further development of monitoring facilities such as those prototyped here.

# Appendix A

# Code of Escher program

Here is the code of the Escher program discussed in Chapter 3. The modules are in alphabetical order:

| | |
|---|---|
| Design.hs | MGR.hs |
| Dmenu.hs | MagicNos.hs |
| Dtrans.hs | Main.hs |
| Escher.hs | Maths.hs |
| EscherAreas.hs | PostScript.hs |
| Etrans.hs | Rational.hs |
| Geometry.hs | State.hs |
| Help.hs | T4.hs |
| Interact.hs | Tile.hs |
| Interface.hs | Tmenu.hs |
| Layout.hs | Transact.hs |
| Lib.hs | Ttrans.hs |
| Lines.hs | |

```
-- Design.hs

-- Functions for the design of the stamp, including
-- the little circles associated with the ends of the lines.
-- It also has the orientations to be applied to the stamp
-- when it is displayed in the boxes, and the display of these
-- boxes (showoris).

module Design (nearx, neary, deline, orient, cs, wwscale,
               wscale, towcoords, wline, showoris) where

import MGR      (circle, line, undo)
import Lines    (place, rotatecw, antirotate, tbinvert,
                 lrinvert, undraw, mapx, mapy)
import Maths    (square, diff, between)
import Layout   (picbox)
import Rational (rdiv, rsub, radd, rmul, rmin, rabs, torat,
                 intval)
import Lib      (concmap3, remove1, listremove1)
import MagicNos (dpxyorig, dpxynum, dpxygap, picxorig, picyorig,
                 xymax, pixdist)

-- These codings are used for the eight pictures,
-- for the program state, and for the postscript file
-- The zero orientation is an blank tile
orient :: Int -> Int -> [[Int]] -> [[Int]]
orient m n = case n of
             0 -> (\_ -> [[0,0,0,0]])
             1 -> (\x -> x)
             2 -> rotatecw m
             3 -> rotatecw m . rotatecw m
             4 -> antirotate m
             5 -> tbinvert m
             6 -> tbinvert m . rotatecw m
             7 -> lrinvert m
             8 -> lrinvert m . rotatecw m

-- is a point on a line? For deleting lines in the design
online :: [Int] -> Int -> Int -> Bool
online [x0,y0,x1,y1] xp yp =
    if y0 == y1 then between x0 x1 xp && abs (y0 - yp) < pixdist
    else if x0 == x1 then
                    between y0 y1 yp && abs (x0 - xp) < pixdist
    else b2 <= a2 + c2 && c2 <= a2 + b2 &&
                              intval (rmin dx dy) < pixdist
    where
    k1 = rdiv (torat (x0 - x1)) (torat (y0 - y1))
    k0 = rsub (torat x0) (rmul k1 (torat y0))
    xp' = radd k0 (rmul k1 (torat yp))
    yp' = rdiv (rsub (torat xp)  k0) k1
    a2 = square (diff x0 x1) + square (diff y0 y1)
    b2 = square (diff x1 xp) + square (diff y1 yp)
    c2 = square (diff x0 xp) + square (diff y0 yp)
    dx = rabs (rsub (torat xp) xp')
    dy = rabs (rsub (torat yp) yp')

-- remove a line from the design, together with the little
-- circles if it's the last one at that position
deline :: [([Int],[Int])] -> [Int] -> ([Char], [([Int],[Int])])
deline ls [px,py] =
    deline' ls
    where
    deline' [] = ("",ls)
    deline' (pl:pls) =
      if online thisline px py then
        (undraw thisline ++ (undo . wline) thisline ++ decircs,
                                                  remove1 ls pl)
      else deline' pls
      where
      (thisline, thesecircs) = pl
      restcircs = listremove1 (concat (map snd ls)) thesecircs
      decircs = (concat . map decirc) (thesecircs \\ restcircs)

-- functions to do with the drawing of lines and marking of
-- circles in the design phase. As the x and y lists for the
-- design area are the same, the function onedge can be defined
-- without specifying onedgex and onedgey
onedge :: Int -> Bool
onedge n =
        n == dpxyorig || n == dpxyorig + (dpxynum -1) * dpxygap

-- similarly the method of finding the nearest x or y points
```

```
-- on the grid are equivalent
nearest :: Int -> Int
nearest n = if n - n1 < n2 - n then n1 else n2
        where
        n1 = dpxyorig + ((n - dpxyorig) `div` dpxygap) * dpxygap
        n2 = n1 + dpxygap

-- but the cursor is not symmetrical in its deficiencies,
-- so we have:
nearx, neary :: Int -> Int
nearx x = nearest (x - 4)
neary y = nearest (y - 5)

-- numassoc is to give points on the edge an associated number
-- with which to code edge intersections. It gives the number
-- of dots from the nearest corner.
numassoc :: Int -> Int
numassoc n = if n1 <= 9 then n1 else 18 - n1
             where
             n1 = (n - dpxyorig) `div` dpxygap

-- circ6 for drawing the little circles of radius 6 pixels
-- to mark the possible intersections with the edge of the
-- design area.
circ6 :: Int -> Int -> [Char]
circ6 x y = circle [x,y,6]

-- circsym for identifying symmetrically placed dots and
-- drawing circles round them.
circsym :: Int -> Int -> ([Char], [Int])
circsym xn yn =
           if onedge xn then (symcircs yn,[numassoc yn])
           else if onedge yn then (symcircs xn,[numassoc xn])
           else ("",[])

-- From this list paired with its reverse may be obtained the
-- eight postions on the edge that correspond to n in the design
sympat :: Int -> [Int]
sympat n = [n, 400-n, 380, 380, 400-n, n, 20, 20]

-- draw the eight little circles associated with edge point n
symcircs :: Int -> [Char]
symcircs n = concat (zipWith circ6 (sympat n)
                                             (reverse (sympat n)))

-- assumes the coordinates have already been corrected to allow
-- for the deficiencies of the cursor, and to fit into the grid
cs :: [Int] -> ([Char], [Int])
cs [x0,y0,x1,y1] =
        (line [x0,y0,x1,y1] ++ circles0 ++ circles1, ids0++ids1)
        where
        (circles0,ids0) = circsym x0 y0
        (circles1,ids1) = circsym x1 y1

-- when a line is deleted that was the only connection to a
-- particular edge connection, the little circles have to go away
decirc :: Int -> [Char]
decirc n  = (undo . symcircs) (n * dpxygap + dpxyorig)

-- for scaling lines for the tiles in the tile area
-- and for the postscript version
scale :: Int -> Int -> Int
scale factor n = (n - dpxyorig) `div` factor

-- wwscale for the lines in postscript
wwscale :: Int -> Int
wwscale = scale 10

-- wscale for the lines in the wee square
-- scaling down by 20% the size of the design to fit on a tile
wscale :: Int -> Int
wscale = scale 5

towcoords :: [([Int],[Int])] -> [[Int]]
towcoords = map ((map wscale) . fst)

wline :: [Int] -> [Char]
wline = line .
        mapx (\x -> x + picxorig) .
        mapy (\y -> y + picyorig) .
        map wscale
```

```
-- display the eight orientations of the tile
showoris :: [[Int]] -> Int -> [Char]
showoris coords n =
        place x y (((orient xymax) n . map (map wscale)) coords)
        where
        [x,y,w,h] = picbox n

-- Dmenu.hs

-- The design area menu defined as an Interface

module Dmenu where

import Transact   (filefun, mi, mkm)
import Dtrans     (todesign', dclear)
import Layout     (dmc)
import Interface  (FindAct(..), Interface(..))
import Geometry   (Coords(..))
import State      (Mode(..), Button(..), Trans(..), Stamp(..),
                   Sel(..), Board(..), State(..), Flag(..))

-- dmenu is an interface that uses dmc to tell whether
-- a click is within a particular button,
-- and dmas to associate buttons with transactions
-- dmas also associates help text with each transaction
dmenu :: Interface
dmenu = mkm dmc dmas

dmas :: [Mode -> Button -> Coords -> Trans]
dmas = [mi todesign'                              (
         "This button puts you in drawing mode.\n"        ++
         "Lines can be drawn in the STAMP DESIGN\n"       ++
         "area by holding down the middle button,\n"      ++
         "and deleted by clicking\n"                      ++
         "with the right one.\n"                          ++
         "\nLittle circles will appear if a line\n"       ++
         "touches the edge of the square.\n"              ++
         "These indicate positions on all the sides\n"    ++
         "that would contact that line in each of\n"      ++
         "the possible orientations of the print.\n"      ++
         "Unless special effects are being sought,\n"     ++
         "the recommendation is that all little\n"        ++
         "circles be attached to a line."    )
  ,
     mi (filefun Dsave)                      (
       "\nClicking on this SAVE button allows\n"        ++
       "you to save a design for future\n"              ++
       "re-use.\n\n"                                    ++
       "You will be prompted for a file name\n"         ++
       "in which it will be saved\n\n"                  ++
       "To restore the design, click on GET\n"          ++
       "and type in the name of the file in which\n"    ++
       "it has been kept."                   )
    ,
     mi (filefun Dget)                        (
       "\n\n\nTo restore a print that has previously\n" ++
       "been saved, click on GET and then\n"            ++
       "type in the filename."               )
  ,
     mi dclear                                (
       "\n\n\nThis clears the PRINT DESIGN grid.\n"     ++
       "The print currently being worked on\n"          ++
       "will be lost, unless it has been\n"             ++
       "explicitly SAVEd first"              )
    ]

-- Dtrans.hs

-- The transactions for the Design Menu, other than
-- the Save and Get that involve file transactions

module Dtrans where

import State     (Mode(..), Button(..), Trans(..), Stamp(..),
                  Sel(..), Board(..), State(..), Flag(..),ssel,
                  sstamp)
import Geometry  (Coords(..))
import Tile      (unmark, initalist)
import Design    (deline, wline, cs,
                  nearx, neary, showoris)
import Layout    (chmode, cleara, tilearea, picarea, picgrid,
                  tpgrid, menumark, unmenumark, newdraw)
import Interface (FindAct(..), Interface(..))
```

```

-- dclear to clear the design area
dclear :: Coords -> Trans
dclear _ state inpt =
            ( menumark Dclear ++ newdraw ++
              unmark (ssel state) ++ unmenumark Dclear ++ str
            , (Draw,[],1,(initalist,[]),Act)
            , inpt)
            where
            (str,_) = chmode state Draw

-- todesign' for changing the mode to Draw
todesign' :: Coords -> Trans
todesign' _ state inpt =
        ( cleara picarea  ++ picgrid ++
          cleara tilearea ++ tpgrid  ++
          showoris (map fst (sstamp state)) 1 ++ modestring
        , newstate
        , inpt)
        where
        (modestring, newstate) = chmode state Draw
```

```
-- Escher.hs
-- This module exports the program expressed as the
-- interpretation of the escher_interface that the module also
-- defines, in terms of the active areas and their respective
-- actions.

module Escher (escher) where

import Interface   (FindAct(..), Interface(..), interpret)
import Transact    (mi, helptrans, notrans)
import Help        (endmes, helpsetup)
import State       (State(..), Mode(..), Stamp(..), Sel(..),
                    Board(..), Button(..), Trans(..), Flag(..))
import Geometry    (Coords(..))
import EscherAreas (indesign, indesmenu, inbigtile, intilemenu,
                    inpicarea, inhelp, inquit)
import Dmenu       (dmenu)
import Tmenu       (tmenu)
import MGR         (clear)
import Etran       (tilef, select, drawf, starthelp, doquit)

escher = interpret escher_interface

escher_interface :: Interface
escher_interface = [FA indesign   desfun               ,
                    FA indesmenu  (interpret dmenu) ,
                    FA inbigtile  tilefun              ,
                    FA intilemenu (interpret tmenu) ,
                    FA inpicarea  orifun               ,
                    FA inhelp     helpfun              ,
                    FA inquit     quitfun ]

tilefun :: Mode -> Button -> Coords -> Trans
tilefun mode button =
   case mode of
        Help  -> helptrans (
                   "\n\n\nWithin the TILE DESIGN area,\n"
                    ++ "a big tile, based on orientations of\n"
                    ++ "a print design, can be built.\n"
                    ++ "\nUsing TILE mode the right button\n"
                    ++ "will select from a palette at the bottom\n"
                    ++ "of the screen, and the middle button"
                    ++ "will\n"
                    ++ "place the selection within the big tile.\n"
                    ++ "Within the area the right button will\n"
                    ++ "delete squares.\n"
                    ++ "\nUsing ALTER mode the right button will\n"
                    ++ "invert squares, and the middle button\n"
                    ++ "will rotate them."
                    ++  endmes  )
        Tile  -> tilef button
        Alter -> tilef button
        Draw  -> notrans

desfun :: Mode -> Button -> Coords -> Trans
desfun mode button =
   case mode of
        Help  -> helptrans (
                    "\n\nThis is the area in which to design\n"
                    ++ "your print.\n"
                    ++ "\nDraw lines by holding down the\n"
```

```
                   ++ "middle button.\n"
                   ++ "Delete lines by clicking with the\n"
                   ++ "right button.\n\n\n"
                   ++ "A print that has previously been saved\n"
                   ++ "can be restored by clicking on GET\n"
                   ++ "then typing in the filename "
                   ++ "at the prompt.\n"
                   ++ endmes  )
        Draw  -> drawf button
        Tile  -> notrans
        Alter -> notrans

orifun mode button =
   case mode of
        Help  -> helptrans (
                   "\nThese boxes show the eight possible\n"
                   ++ "orientations of the print that is\n"
                   ++ "to be used in tiling\n"
                   ++ "\nWhen in tiling mode, clicking with the\n"
                   ++ "right button over one of these\n"
                   ++ "will make it the \"current selection\".\n"
                   ++ "Clicking with the middle button in\n"
                   ++ "the TILE DESIGN grid, will put that\n"
                   ++ "orientation of the print at that place"
                   ++ endmes  )
        Tile  -> case button of
                      R -> select
                      M -> notrans
        Draw  -> notrans
        Alter -> notrans

helpfun =
     mi starthelp
        (helpsetup ++ clear ++
                "\n\n\nTo find out the use of a particular\n"
                ++ "menu button or region of the screen,\n"
                ++ "click over the item you wish to\n"
                ++ "investigate." )

quitfun  = mi doquit ("\n\n\nClicking on QUIT allows you\n"
                        ++ "to leave the program." )
```

```
-- EscherAreas.hs

-- EscherAreas has finding functions for the areas of the Escher
-- interface.

module EscherAreas (intilemenu, indesmenu,inpicarea, indesign,
                    inbigtile, inhelp, inquit) where

import Geometry (Coords(..), inrect, incirc)
import State    (Flag(..), State(..), Mode(..), Stamp(..),
                 Sel(..), Board(..))
import MagicNos (tpxorig, tpyorig, tpxygap, tpxynum,
                 dpxyorig, dpxygap, dpxynum,
                 tmxygap, tmxnum, tmynum, tmxorig, tmyorig,
                 dmxygap, dmxnum, dmynum, dmxorig, dmyorig,
                 dmcircr, tmcircr,
                 picxorig, picyorig, picxnum, picynum, picxygap,
                 helpbr, helpx, helpy,
                 quitbr, quitx, quity)

inbigtile, indesign :: Coords -> Bool

-- in the design area
-- (a bit wider than the displayed grid)
indesign =
   inrect ((dpxygap * (dpxynum - 1) + 18) )
          ((dpxygap * (dpxynum - 1) + 18) )
          [dpxyorig - 6, dpxyorig - 6]

-- in the tiling area
inbigtile = inrect tpwh tpwh [tpxorig,tpyorig]
            where
            tpwh = tpxygap * (tpxynum - 1)

-- in the tile menu area
intilemenu :: Coords -> Bool
intilemenu = inrect (tmxygap * tmxnum) (tmxygap * tmynum)
                    [(tmxorig - tmcircr), (tmyorig - tmcircr)]

-- in the design menu area
indesmenu :: Coords -> Bool
```

```
indesmenu = inrect (dmxygap * dmxnum) (dmxygap * dmynum)
                   [(dmxorig - dmcircr), (dmyorig - dmcircr)]


-- in the area where the eight orientations are displayed
inpicarea :: Coords -> Bool
inpicarea = inrect (picxygap * picxnum) (picxygap * picynum)
                   [picxorig -1,picyorig -1]

-- THE HELP AND QUIT BUTTONS
inhelp, inquit                       :: Coords -> Bool

-- in the help button
inhelp = incirc helpbr [helpx,helpy]

-- in the quit button
inquit = incirc quitbr [quitx,quity]
```

```
-- Etrans.hs

-- The Escher transactions for the whole interface are mostly
-- defined in modules relevant to particular menus. Here are
-- the drawing functions for the design area, and the tile
-- area

module Etrans where



import State      (Mode(..), Button(..), Trans(..), Stamp(..),
                   Sel(..), Board(..), State(..), Flag(..),smode,
                   sstamp, ssel, sboards)
import Geometry   (Coords(..))
import Tile       (sqid, btlocate, inv, rot, inbox, mark, unmark)
import Design     (towcoords, orient, deline, wline, cs,
                   nearx, neary)
import Lib        (assoc, newas)
import Lines      (put)
import MGR        (undo)
import Layout     (chmode)
import Help
import MagicNos   (xymax)
import Interface (FindAct(..), Interface(..))

drawf :: Button -> Coords -> Trans
drawf button coords state inpt = (out, newstate, inpt)
        where
        (out, newstate) = drawf' button coords state

drawf' button coords state =
        (out, (Draw,newdlist,ssel state,sboards state,Act))
        where
        (out, newdlist) =
                case button of
                      R -> deline dlist coords
                      M -> (linecircs ++
                           (wline nstoilrest), (newele:dlist))
        nearline [x0,y0,x1,y1] =
                       [nearx x0, neary y0, nearx x1, neary y1]
        nstoilrest              = nearline coords
        cssr                    = cs nstoilrest
        newele                  = (nstoilrest,snd cssr)
        linecircs               = fst cssr
        dlist                   = sstamp state

tilef :: Button -> Coords -> Trans
tilef button coords state inpt = (out, newstate, inpt)
        where
        (out, newstate) = tilef' button coords state

tilef' button coords state =
        ((undo . tplace) oldas ++ tplace new,
        (mode, dlist, sel, (newtilist,[]), Act))
        where
        mode      = smode state
        dlist     = sstamp state
        sel       = ssel state
        tilist    = (fst . sboards) state
        atile     = sqid coords
        wcoords   = towcoords dlist
        oldas     = assoc atile tilist
        lsrest    = btlocate coords
        newtilist = newas atile new tilist
```

```
        new       = case mode of
                         Tile  -> case button of
                                     R -> 0
                                     M -> sel
                         Alter -> case button of
                                     R -> inv oldas
                                     M -> rot oldas
        tplace o = put lsrest (orient xymax o  wcoords)

starthelp :: Coords -> Trans
starthelp _   state inpt =
        (str ++ inithelp, newstate, inpt)
        where
        (str, newstate) = chmode state Help

select ::  Coords -> Trans

-- the mode will be Tile
select rest (_,dlist,sel,tilists,_) inpt =
        (unmark sel ++ mark newsel,
                        (Tile,dlist,newsel,tilists,Act),inpt)
        where
        new    = inbox rest
        newsel = if new == 0 then sel else new

doquit :: Coords -> Trans
doquit _ state _  = ("",state,[])
```

```
-- Geometry.hs

-- A module that exports functions for drawing and locating
-- rectangles and circles, dots and grids.

module Geometry (Coords(..),drawdot, grid, squ, circ,
                 rectangle, inrect, incirc) where

import MGR        (line, circle)
import Maths      (diff, square)
import Lib        (allpairs)

type Coords = [Int]

rectangle :: [Int] -> [Char]
rectangle [x1,y1,x2,y2] = line [x1,y1,x2,y1] ++
                          line [x2,y1,x2,y2] ++
                          line [x1,y1,x1,y2] ++
                          line [x1,y2,x2,y2]

fillrect :: [Int] -> [Char]
fillrect [x0,y0,x1,y1] = shade (diff x0 x1)
                         where
                         m = min x0 x1
                         vline n = line [n,y0,n,y1]
                         shade 0 = vline m
                         shade n = vline (m+n) ++ shade (n-1)

squ :: Int -> Int -> Int -> [Char]
squ n x y = rectangle [x, y, x+n, y+n]

circ :: Int -> Int -> Int -> [Char]
circ n x y  = circle [x,y,n]

-- a dot is a 3 X 3 filled square
drawdot :: Int -> Int -> [Char]
drawdot x y = fillrect [x-1, y-1, x+1, y+1]

-- grid -- a function that draws a grid.
-- The function drawf is applied to each x y pair in the grid
-- For example a grid of dots would use drawdot
-- The Coords is the top left origin of the grid

grid :: Int -> Int -> Int -> Int -> (Int -> Int -> [Char]) ->
                                          Coords -> [Char]
grid xgap ygap xlength ylength drawf [xor,yor] =
        concat [drawf x y| x <- x0list, y <- y0list]
        where
        x0list = gridlist xor xgap xlength
        y0list = gridlist yor ygap ylength
        gridlist orig gap len = take len (iterate ((+) gap) orig)

-- The second versions of inrect and incirc allow the Coords
-- to be the ends of a line rather than just one point.
```

```
inrect :: Int -> Int -> Coords -> Coords -> Bool
inrect w h [x,y] [xp,yp]        = xp > x && xp <= x + w &&
                                  yp > y && yp <= y + h
inrect w h [x,y] [x1,y1,x2,y2] = inrect w h [x,y] [x1,y1] &&
                                  inrect w h [x,y] [x2,y2]

incirc :: Int -> Coords -> Coords -> Bool
incirc r [xc,yc] [xp,yp]       =
                 square (xp - xc) + square (yp - yc) <= square r
incirc r [xc,yc] [x1,y1,x2,y2] = incirc r [xc,yc] [x1,y1] &&
                                  incirc r [xc,yc] [x2,y2]
```

```
-- Help.hs

-- The help text for the various areas is now included in the
-- definition of the transactions for that area, so that it can
-- also serve as a comment.

module Help (endmes, helpsetup, -- printcommand, predefinedpats,
             helpend, inithelp, errmes) where

import Layout (helptextarea, cleara)
import MGR    (textregion, font, clear, smallfont, largefont)

helpsetup, helpend :: [Char]

helpsetup = textregion helptextarea ++ smallfont

helpend = cleara helptextarea ++ largefont

inithelp, errmes, endmes:: [Char]

inithelp =
        helpsetup ++ clear ++  "\n\n\n\n "         ++
        "To find out the use of a particular\n"   ++
        "menu button or region of the screen, \n" ++
        "click over the item you wish to\n"       ++
        "investigate.\n" ++
         endmes

errmes = "\n\n\n"                               ++
         "You have clicked over an area \n" ++
         "of no particular interest.\n"     ++
          endmes

endmes = "\n\n PRESS RETURN TO RETURN TO THE PROGRAM\n" ++
             "OR CLICK SOMEWHERE ELSE TO FIND OUT MORE\n"
```

```
-- Interact.hs

-- The function for interaction. It used to allow for a prompt
-- at each stage in the interaction appropriate to the state.
-- It turns out that, in this application, the prompt is,
-- in every case, an empty string.
-- So this aspect of the interaction function has been dropped,
-- thus making it more efficient, but less generic.

module Interact (inter) where

-- Using [[Char]] for input
inter :: (state -> [[Char]] -> Bool) ->
         (state -> [[Char]] -> [Response] ->
         ([Request], state, [[Char]], [Response])) ->
         (state -> [[Char]] -> [Response] -> [Request])

inter endp transf = inter'
     where
     inter' state input resps |endp state input = []
                              |otherwise        = out ++ outs
        where
        (out,state',input',resps') = transf  state  input  resps
        outs                       = inter'  state' input' resps'
```

```
-- Interface.hs

-- The interface module defines the type of an interface.
-- As the possible active areas are more or less fixed in this
-- application, the more general description:
-- data DrawFindAct =
-- DFA [Char] (Coords -> Bool)
--                  (Mode -> Button -> Coords -> Trans)
-- where the [Char] is the string to MGR to display the area,
-- is unnecessary. If the display changed significantly with each
```

```
-- change of mode, this might, however, be more useful. Each mode
-- would have associated with it an interface, including the
-- display element.

module Interface (FindAct(..), Interface(..), interpret) where

import State      (Mode  (..), Stamp(..), Sel  (..), Board(..),
                   Button(..), Trans(..), State(..), Flag (..))
import Geometry   (Coords(..))
import Transact   (notrans)



data FindAct =
    FA (Coords -> Bool) (Mode -> Button -> Coords -> Trans)

-- Interface is here a list of fixed interface elements
type Interface = [FindAct]

infa :: FindAct -> Coords -> Bool
infa (FA pb _) pt = pb pt

funfa :: FindAct -> (Mode -> Button -> Coords -> Trans)
funfa (FA _ tfun) = tfun

-- argument order so that:
-- escher = interpret escher_interface
interpret :: Interface -> Mode -> Button -> Coords -> Trans
interpret [] _ _ _           = notrans []
interpret (fa:rest) m b pt   = if       infa fa pt
                               then   (funfa fa m b) pt
                               else interpret rest m b pt
```

```
-- Layout.hs

-- Layout has the display routines for the various areas
-- that are defined for the program. It also has the chmode
-- function which needs to know where the mode buttons are
-- in the display in order to mark and unmark them appropriately.
-- Menu buttons also have strings to MGR to mark or unmark them.

module Layout( setup, closedown,
                invisibletext, vistextreg, textarea, helptextarea,
                designarea,tilearea, picarea, dpgrid,tpgrid,
                picgrid, clearit, cleara, picbox, newdraw,
                dmc, tmc, menumark, unmenumark, markmode,
                unmarkmode, chmode)
                 where

import MGR      (invertmode, insertmode, absolute, circle,
                 stringto, smallfont, shapewindow, textregion,
                 clear, textreset, go, aligntext, undo)
import Geometry (Coords(..), inrect, incirc, drawdot, grid,
                 circ, squ)
import State    (Flag(..), State(..), Mode(..), Stamp(..),
                 Sel(..), Board(..), setmid, setright)
import MagicNos (tpxorig, tpyorig, tpxygap, tpxynum,
                 dpxyorig, dpxygap, dpxynum,
                 tmxygap, tmxnum, tmynum, tmxorig, tmyorig,
                 dmxygap, dmxnum, dmynum, dmxorig, dmyorig,
                 dmcircr, tmcircr, picsqu,
                 picxorig, picyorig, picxnum, picynum, picxygap,
                 helpbr, helpx, helpy,
                 quitbr, quitx, quity,
                 dmcirc, tmcirc, amcirc, hmcirc,
                 menustrings)

-- THE DESIGN AREA

dpfun :: Int -> Int -> [Char]
dpfun    = drawdot

dpgrid :: [Char]
dpgrid = grid dpxygap dpxygap dpxynum dpxynum dpfun
                                                [dpxyorig,dpxyorig]

designarea :: [Int]
designarea = [dpxyorig - 6,
               dpxyorig - 6,
              ((dpxygap * (dpxynum - 1))) + 15,
              ((dpxygap * (dpxynum - 1))) + 15]

-- THE TILE AREA
```

```
tpfun :: Int -> Int -> [Char]
tpfun   = drawdot

tpgrid :: [Char]
tpgrid = grid tpxygap tpxygap tpxynum tpxynum tpfun
                                            [tpxorig,tpyorig]

tilearea :: [Int]
tilearea = [tpxorig - margin,
            tpyorig - margin,
            margin + tpxygap * (tpxynum - 1),
            margin + tpxygap * (tpxynum - 1)]
           where
           margin = 3

-- THE TILE MENU
tmfun :: Int -> Int -> [Char]
tmfun   = circ tmcircr

tmgrid :: [Char]
tmgrid = grid tmxygap tmxygap tmxnum tmynum tmfun
                                            [tmxorig,tmyorig]

tmc :: Int -> Coords -> Bool
tmc n = incirc tmcircr [tmxorig, tmyorig + n * tmxygap]

-- THE DESIGN MENU

-- The drawing function for the design menu
dmfun :: Int -> Int -> [Char]
dmfun   = circ dmcircr

-- The design menu as a single column grid
dmgrid :: [Char]
dmgrid = grid dmxygap dmxygap dmxnum dmynum dmfun
                                            [dmxorig,dmyorig]

-- Is a click within a particular design menu button?
dmc :: Int -> Coords -> Bool
dmc n = incirc dmcircr [dmxorig, dmyorig + n * dmxygap]

-- THE ORIENTATIONS
-- pic definitions relate to the display of the eight
-- orientations of the print

picfun :: Int -> Int -> [Char]
picfun   = squ picsqu

picgrid :: [Char]
picgrid = grid picxygap picxygap picxnum picynum picfun
                                [(picxorig -1),(picyorig-1)]

picarea :: [Int]
picarea = [picxorig -1,
           picyorig -1,
           picxygap * picxnum,
           picxygap * picynum]

-- the coordinates of the box to mark/unmark one of the eight
-- orientations which are in two rows coded 0 0 to 1 3
picbox :: Int -> [Int]
picbox n = [ picxorig + n4  * picxygap,
             picyorig + n4' * picxygap,
             picsqu2, picsqu2 ]
           where
           n4  = case n of
                 4 -> 3
                 8 -> 3
                 _ -> (n 'mod' 4) - 1
           n4' = if n <= 4 then 0 else 1
           picsqu2 = picsqu - 2

-- THE TEXT AREA FOR HELP AND INTERACTION
-- LEAVE, but maybe rationalise later and make them the same area
textarea :: [Int]
textarea = [50,550,300,300]

-- vistextreg is the region into which to type filenames
vistextreg :: [Char]
vistextreg = textregion [50,615,200,100]

helptextarea :: [Int]
```

```
helptextarea = [50,500,380,400]

clearit :: [Char]
clearit = cleara textarea

cleara :: [Int] -> [Char]
cleara area = textregion area ++ clear ++ invisibletext

-- invisibletext sets up a scrolling text region, then moves the
-- text cursor out of it.

invisibletext :: [Char]
invisibletext = vistextreg ++
                go [500,500] ++
                aligntext ++ "\n"

-- MODE BUTTONS

-- marks a mode button
markmode :: Mode -> [Char]
markmode mode =
        case mode of
          Draw  -> dmcirc
          Tile  -> tmcirc
          Alter -> amcirc
          Help  -> hmcirc

unmarkmode :: Mode -> [Char]
unmarkmode = undo . markmode

-- OTHER MENU BUTTONS
-- These initiate transactions when activated
-- They're coded according to their position in the menu
menumark :: Flag -> [Char]
menumark flag = circ radflag xof (yof + f * g)
   where
   ((radflag, xof, yof, g), f) =
                case flag of
                   Tsave  -> (tmf, 2)
                   Tget   -> (tmf, 3)
                   Tclear -> (tmf, 4)
                   T4     -> (tmf, 5)
                   Dsave  -> (dmf, 1)
                   Dget   -> (dmf, 2)
                   Dclear -> (dmf, 3)
                where
                tmf = (tmcircr + 2, tmxorig, tmyorig, tmxygap)
                dmf = (dmcircr + 2, dmxorig, dmyorig, dmxygap)

unmenumark :: Flag -> [Char]
unmenumark = undo . menumark

-- Draw the Help and Quit buttons
buttons :: [Char]
buttons = circle [helpx,helpy,helpbr] ++
          circle [quitx,quity,quitbr]

-- The original set up of the screen and MGR buttons:
-- use absolute coordinates
-- make this big window with no text region
-- clear it, draw areas, buttons and labels
-- set the buttons to return coordinates appropriately for
-- Draw mode and mark the Draw mode button
setup :: [Char]
setup = absolute ++
        shapewindow [0,0,1150,900] ++
        textregion [0,0,0,0] ++
        clear ++
         tpgrid ++ dpgrid ++ tmgrid ++ dmgrid ++ picgrid ++
          buttons ++ invisibletext ++ menustrings ++
        setmid "%l" ++ setright "%p" ++ markmode Draw

-- The closedown of the program
closedown = shapewindow [0,0,500,500]    ++
            smallfont ++ textreset ++ clear ++
            insertmode

-- newdraw clears and redraws the design area, and the picarea.
-- also the tile area
-- It is used by dclear and by dget
-- doesn't need the state, as Mode is already Draw
newdraw :: [Char]
```

```
newdraw = cleara designarea ++
          dpgrid            ++
          cleara picarea    ++
          picgrid           ++
          cleara tilearea   ++
          tpgrid            ++
          invisibletext     ++
          setmid "%l"

-- for marking and unmarking the mode buttons it is more
-- convenient to hold the mode as a data type than as
-- a function.
chmode :: State -> Mode -> ([Char],State)
chmode (mode1,dlist,sel,tilist,_) mode2 =
     (drawspecial ++ unmarkmode mode1 ++ markmode mode2, newstate)
     where
     newstate    = (mode2, dlist, sel, tilist, Act)
     drawspecial = case mode1 of
                        Draw -> setmid "%p"
                        _    -> case mode2 of
                                   Draw -> setmid "%l"
                                   _     -> ""
```

```
--Lib.hs

module Lib (pam, newas, putline, totext, assoc, stoil,
            allpairs, concmap, concmap3, pamcat, concrep,
            remove1, listremove1) where

pam :: (a -> b -> c) -> [a] -> b -> [c]
pam f xs y = map (\x -> f x y) xs

newas :: (Eq a) => a -> b -> [(a,b)] -> [(a,b)]
newas i e [] = [(i,e)]
newas i e ((g1,g2):gs) = if g1 == i then (i,e) : gs
                         else (g1,g2) : newas i e gs

assoc :: (Eq a) => a -> [(a,b)] -> b
assoc i ((j,v):ivs) = if i == j then v else assoc i ivs

totext :: [[Int]] -> [Char]
totext = concat . map putline

putline :: [Int] -> [Char]
putline [x0,y0,x1,y1] = 'M': show x0 ++ " " ++ show y0 ++ " " ++
                             show x1 ++ " " ++ show y1 ++ "\n"

stoil :: [Char] -> [Int]
stoil = map read . words

concmap = (concat .) . map

concmap3 :: (a -> b -> c -> [d]) -> [a] -> [b] -> [c] -> [d]
concmap3 f (x:xs) (y:ys) (z:zs) = f x y z ++ concmap3 f xs ys zs
concmap3 f _      _      _      = []

pamcat :: [(a -> [b])] -> a -> [b]
pamcat (f:fs) a = f a ++ pamcat fs a
pamcat [] a = []

concrep :: Int -> [a] -> [a]
concrep x y = concat (take x (repeat y))

-- remove1 xs y is xs with 1st occurrence (if any) of y removed
remove1 :: (Eq a) => [a] -> a -> [a]
remove1 (l:ls) i = if i==l then ls else l : remove1 ls i
remove1 []     i = []

listremove1 :: (Eq a) => [a] -> [a] -> [a]
listremove1 = foldl remove1

-- used in Drawfuns for drawing grids
allpairs _ [] _ = []
allpairs _ _ [] = []
allpairs f (x:xs) ys = map (f x) ys ++ allpairs f xs ys
```

```
-- Lines.hs

-- Functions that yield strings for placing, displacing,
-- and removing lines (under MGR)

module Lines (put, place, toright, down, mapx, mapy,
              rotatecw, antirotate, tbinvert, lrinvert,
```

```
                  undraw) where

import MGR (line, undo)

xs, ys, swapxy :: [Int] -> [Int]

xs [x1,y1,x2,y2] = [x1,x2]
ys [x1,y1,x2,y2] = [y1,y2]
swapxy [x1,y1,x2,y2] = [y1,x1,y2,x2]

mapx, mapy :: (Int -> Int) -> [Int] -> [Int]

mapx f [x1,y1,x2,y2] = [f x1, y1, f x2, y2]
mapy f [x1,y1,x2,y2] = [x1, f y1, x2, f y2]

toright, down :: Int -> [[Int]] -> [[Int]]

toright = map . mapx . (+)
down = map . mapy . (+)

origin :: Int -> Int -> [[Int]] -> [[Int]]
origin x y = (toright x) . (down y)

-- place x y takes a print and outputs a string that
-- is interpreted by MGR with the result that
-- the print is drawn at x y

place :: Int -> Int -> [[Int]] -> [Char]
place x y = drawlines . (origin x y)

put :: [Int] -> [[Int]] -> [Char]
put [x,y] = place x y

lrinvert, tbinvert, rotatecw, antirotate ::
                                        Int -> [[Int]] -> [[Int]]

lrinvert   m = map (mapx  (m -))
tbinvert   m = map (mapy  (m -))
rotatecw   m = map (swapxy . mapy  (m -))
antirotate m = map (swapxy . mapx  (m -))

undraw :: [Int] -> [Char]
undraw = undo . line

drawlines :: [[Int]] -> [Char]
drawlines = concat . map line
```

```
-- MGR.hs

-- Functions that yield escape strings to manipulate MGR

module MGR ( aligntext, circle, clear, font, go, line, newwin,
             rcircle, smallfont, mediumfont, largefont,
             selectwin, setevent, shapewindow, stringto,
             textregion, textreset, deletemode, insertmode,
             invertmode, absolute, activate, undo) where


-- most MGR functions yield escape strings
-- this "escape command" function builds them
escom :: [Char] -> [Int] -> [Char]
escom str ns = '\ESC' : foldr f "" ns
               where
               f n "" = show n ++ str
               f n s = show n ++ "," ++ s

-- align text with the graphics cursor
aligntext = '\ESC' : "l"

-- draw a circle at x y of radius r
circle = escom "o"   -- x y r

-- clear the text area
clear = "\FF"

-- change to font x
font x = escom "F" [x]
smallfont  = font 8
mediumfont = font 12
largefont  = font 13

-- select drawing mode
func mode = escom "b" [mode]
```

```

-- move the graphics cursor to point x y
go = escom "g"        -- x y

-- draw a line
line = escom "l"      -- x0 y0 x1 y1

-- create a new window
newwin = escom "Z"    -- x y w h

-- draw a circle of radius r at the current graphics point
rcircle r = escom "o" [r]

-- make window n active
selectwin n = escom "Z" [n]

-- used for example when setting the string to be returned
-- by a mouse button click
setevent event str = escom ("e"++str) [event, length str]

-- set various parameters, such as whether the window
-- is to measure in absolute or relative coordinates
setmode mode = escom "S" [mode]

-- reshape window to the given dimensions
shapewindow = escom "W" -- x y w h

-- write string str at x y on window win
-- use 0 for current window
-- need to use invertmode for this
stringto win x y str = escom ("."++str) [win,x,y,length str]

-- create a text region of given dimensions within the window
textregion = escom "t" -- x y wide high

textreset = '\ESC':"t"

deletemode, insertmode, invertmode :: [Char]
deletemode = func 0

insertmode = func 15

invertmode = func 4 -- essential for stringto

absolute, activate :: [Char]
absolute   = setmode 7

activate   = setmode 8

-- to undo lines by drawing them again in inverse mode
undo :: [Char] -> [Char]
undo f = deletemode ++ f ++ insertmode
```

```
-- MagicNos.hs

-- Numbers used in coordinates for both drawing and finding
-- interface areas Also the size of an element in the tiling
-- area (xymax), and the distance from a line in the drawing
-- area that is small enough to indicate that this is the
-- line to delete (pixdist).

module MagicNos where

import MGR (circle, invertmode, insertmode, stringto, smallfont,
            mediumfont, largefont)

-- the help button
helpbr, helpx, helpy :: Int
helpbr = 36
helpx  = 485
helpy  = 712

-- the quit button
quitbr, quitx, quity :: Int
quitbr = 36
quitx  = 485
quity  = 812

-- dp definitions relate to the design phase of the program
-- x and y origin, gap and number are the same
dpxyorig, dpxygap, dpxynum :: Int
dpxyorig = 20
dpxygap  = 20
```

```
dpxynum  = 19


-- tp definitions relate to the tiling phase of the program
tpxorig, tpyorig, tpxygap, tpxynum :: Int
tpxorig = 524
tpyorig = 20
tpxygap = 72
tpxynum = 9

-- tm definitions relate to the menu for the tiling phase
tmxorig, tmyorig, tmxygap, tmxnum, tmynum, tmcircr :: Int
tmxorig = 485
tmyorig = 282
tmxygap = 57
tmxnum  = 1
tmynum  = 6
tmcircr = 28 -- the radius of the tile menu buttons

-- dm definitions relate to the menu for the design phase
dmxorig, dmyorig, dmxygap, dmxnum, dmynum, dmcircr :: Int
dmxorig = 425
dmyorig = 54
dmxygap = 57
dmxnum  = 1
dmynum  = 4
dmcircr = 28 -- the radius of the design menu buttons

-- pic definitions relate to the display of the eight
-- orientations of the print
picxorig, picyorig, picxygap, picxnum, picynum, picsqu :: Int
picxorig = 624
picyorig = 676
picxygap = 100
picxnum  = 4
picynum  = 2
picsqu   = 74

-- MODE BUTTONS
-- These are the circles round the mode buttons used for marking
-- the mode could be related to menus etc. (apart from Help)
dmcirc, tmcirc, amcirc, hmcirc :: [Char]
dmcirc = circle [425,54,30]
tmcirc = circle [485,282,30]
amcirc = circle [485,339,30]
hmcirc = circle [485,712,38]

-- xymax is for use by T4: the size of the square in
-- the big tile
xymax :: Int
xymax = 72

-- pixdist nearness to a line that may be deleted in the design
pixdist :: Int
pixdist = 10

-- MENUSTRINGS
-- these are strings to go in the menu boxes
menustrings :: [Char]
menustrings = invertmode ++
              smallfont ++
              stringto 0 405 64 "DRAW" ++
              stringto 0 405 121 "SAVE" ++
              stringto 0 410 178 "GET" ++
              stringto 0 402 235 "CLEAR" ++
              stringto 0 468 292 "TILE" ++
              stringto 0 462 349 "ALTER" ++
              stringto 0 468 404 "SAVE" ++
              stringto 0 471 463 "GET" ++
              stringto 0 464 520 "CLEAR" ++
              stringto 0 474 577 "T4" ++
              mediumfont ++
              stringto 0 457 729 "HELP" ++
              stringto 0 457 829 "QUIT" ++
              largefont ++
              stringto 0 112 450 "STAMP DESIGN"++
              stringto 0 730 666 "TILE DESIGN"++
              insertmode
```

```
-- Main.hs

-- escherprint is the interactive program,
-- a specialisation of the inter function,
```

```
-- that uses the interpretation of the escher interface
-- as its essential transaction, but changes the mode
-- of the program state to enable a file handling transaction
-- when this is required.
-- The file handling functions, that involve more than
-- one Response, are also in this module, as fdget calls
-- tiletrans to place the lines of a retrieved pattern.

module Main (main) where

import PostScript (pos8head, introline, lf)
import MagicNos   (xymax)
import Interact          (inter)
import Design     (orient, towcoords, wwscale, wscale)
import Lines      (put)
import Layout     (chmode, unmenumark, newdraw, clearit, cleara,
                   tilearea, tpgrid, setup, closedown)
import Tile       (initalist, sqas, squas, turn, alistind,
                   ineights, tpatformat, initstate)
import Help       (helpend)
import Lib        (totext, stoil, pam, newas)
import State      (State(..), Flag(..), Trans(..), Button(..),
                   Board(..), Sel(..), Stamp(..), Mode(..),
                   smark, sflag, setmid, smode,tilestoput)
import Escher     (escher)
import Geometry   (Coords(..)) -- for hbc995

tilequit :: State -> [[Char]] -> Bool
tilequit state _ |tilestoput state = False
tilequit state (('q':_):_)       = case sflag state of
                                     Tget  -> False
                                     Tsave -> False
                                     Dget  -> False
                                     Dsave -> False
                                     _     -> True
tilequit _ []                    = True
tilequit _          _            = False

tiletrans :: State -> [[Char]] -> [Response] ->
        ([Request], State, [[Char]], [Response])
tiletrans state inpt ~(_:resps) |tilestoput state =
        ([AppendChan stdout str],newstate,inpt,resps)
        where
        (str,newstate) = tput state
tiletrans state inpt resps  = trans (case sflag state of
                                     Tget  -> ftget
                                     Tsave -> ftsave
                                     Dget  -> fdget
                                     Dsave -> fdsave
                                     _     -> ftrans)
                              where
                              trans f = f state inpt resps
ftrans s i ~(_:resps) =
        ([AppendChan stdout out], s', i', resps)
        where
        (out, s', i') = transact s i

-- the first clause assumes that CR will be pressed
-- only when the user wants to get out of Help mode
-- but won't cause too much harm if this is done inadvertently
-- It puts the program into Draw mode.

transact :: Trans
transact state ("":inpt) = (helpend ++ out, newstate, inpt)
        where
        (out, newstate) = chmode state Draw

transact state (mgrstr:inpt) =
        if button == M || button == R
        then escher mode button coords state inpt
        else transact state inpt
        where
        mode = smode state
        (button, coords) = decode mgrstr

-- decode takes a line of input, and returns
-- the button pressed and the coordinates associated with this
decode :: [Char] -> (Button, [Int])
decode ln =
        case rest of
             [] -> (L,[]) -- error from keyboard input
             _  -> (b,rest)
```

```
        where
        b     = case h of
                   'M' -> M
                   'R' -> R
                   _   -> L   -- an error
        rest  = stoil t
        (h:t) = ln

tput :: State -> (String,State)
tput (mode,dlist,sel,(tilist,((inds,ori):reqs)),_) =
          (put lsrest (orient xymax ori wcoords),
                        (mode,dlist,sel,(newtilist,reqs),Act))
          where
          newtilist = newas inds ori tilist
          lsrest = squas inds
          coords = map fst dlist
          wcoords = map (map wscale) coords

escherprint :: State -> [[Char]] -> [Response] -> [Request]
escherprint = inter tilequit tiletrans

---------------- FILE HANDLING FUNCTIONS --------------------
-- ftget, ftsave, fdget, and fdsave
-- involve more than one response, as they either
-- need to read, or to write to, a file.

fdget state@(_,dlist, sel, tilists, _) (hin:tin)
                                         ~(r: ~(_:resps)) =
        (reqs, smark Act newstate, restin, resps)
        where
          reqs = [ReadFile hin,
                AppendChan stdout (result ++ unmenumark Dget)]
        conddraw = if dlist == [] then "" else newdraw
        (result, newstate, restin) =
            case r of
              Failure _    -> (clearit, state, tin)
              Str contents ->
                  (clearit ++ conddraw ++ out, s, inp)
                  where
                  (out,s,inp) =
                      transact (Draw,[],sel,tilists,Act)
                                               (lines contents ++ tin)

-- A pattern may be imposed on the big tile
-- without the need to change to Tile mode
ftget (mode,dlist, sel, (tilist,_), _) (hin:tin)
                                            ~(r: ~(_:resps)) =
        (reqs, (mode,dlist,sel,(tilist, snd infromfile),Act),
                                                     tin, resps)
        where
        wcoords = towcoords dlist
        patfile = if hin == "" then hin
                  else if head hin == '*' then lib ++ tail hin
                       else hin ++ ".pat"
        lib = "/u1/staff/sandra/anubis/sandra/escher/potato/"
        out = cleara tilearea ++
              fst infromfile  ++
              clearit         ++
              unmenumark Tget
        reqs = [ReadFile patfile,
                AppendChan stdout out]
        infromfile =
          case r of
            Failure _ -> (tpgrid, initalist)
                                         -- as tile area is cleared
            Str ls8   -> ( "" , zip alistind orilist )
                             where
                             orilist = concat (map stoil (lines ls8))

fdsave state@(_,dlist,_,_,_) (hin:tin) ~(_: ~(_:resps)) =
        (reqs, smark Act state, tin, resps)
        where
        reqs = [WriteFile hin (totext (map fst dlist)),
                AppendChan stdout (clearit ++ unmenumark Dsave)]


-- now puts .ps on the postscript files
-- as well as .pat on the pattern files
-- Assuming Success from the WriteFiles and the AppendChan
ftsave state@(_,dlist,_,(tilist,_),_) (hin:tin)
                                       ~(_: ~(_: ~(_:resps))) =
        (reqs, smark Act state, tin, resps)
```

```
        where
        reqs = [WriteFile (hin ++ ".ps" ) pstext,
                WriteFile (hin ++ ".pat") pattext,
                AppendChan stdout (clearit ++ unmenumark Tsave)]
        pstext  =
           pos8head (tops dlist) ++
           introline ++
           (concat . (map lf)) ((reverse . ineights)
                    (map (turn . snd) tilist)) ++ "\nshowpage\n"
        pattext = (tpatformat . ineights . map snd) tilist
        tops = (map (map wwscale)) . (map fst)


main :: Dialogue
main ~(Str fromMgr : ~(_ :resps)) = (ReadChan stdin: su: out)
  where
  su  = AppendChan stdout setup
  out = escherprint (Draw,[],1,(initalist,[]),Act)
                                     (lines fromMgr) resps ++ end
  end = [AppendChan stdout closedown]
```

```
-- Maths.hs

-- Functions for Ints, pretending they're Nats.
-- (Which are appropriate for numbers that refer to pixels)

module Maths (diff, bcroot, square, between) where

square :: Int -> Int
square n = n*n

diff :: Int -> Int -> Int
diff a b = if a>b then a-b else b-a

bcroot :: Int -> Int
bcroot n = root' 0 n
           where root' a b = if a+1>=b then b
                             else if s<n then root' m b
                             else if n<s then root' a m
                             else m
                             where
                             m = (a+b) `div` 2
                             s = m*m

between :: Int -> Int -> Int -> Bool
between n1 n2 n = (n1 <= n && n2 >= n) || (n1 >= n && n2 <= n)
```

```
-- PostScript.hs

-- Functions used when creating the postscript text when a
-- big tile is saved, so that it can be printed out.
-- The postscript file is given an appropriate header,
-- the eight orientations of the tile specified as
-- "print0..print8", then the actual order of orientations
-- translated from the tile state

module PostScript (pos8head, introline, lf) where

import Design   (orient)
import Lib      (pamcat, concrep)

pos8head :: [[Int]] -> [Char]
pos8head coords =
          header ++ pamcat (map newf [1 .. 8]) coords
          where
          header = "%!PS-Adobe-1.0\n0.75 setlinewidth\n" ++
                   "/print0\n{\n} def\n"
          topos [x1,y1,x2,y2] = show x1++" "++show y1++
                                                " moveto\n"++
                                      show x2++" "++show y2++
                                                " lineto\n"
          fpat h f coords =
                   h ++ (concat . map topos . f) coords ++
                                            "stroke} def\n"
          newf n = fpat ("/print" ++ show n ++ "\n{")
                                            (orient psmax n)

introline, rowline, ss :: [Char]
introline = "400 400 translate"
rowline = "\n-288 36 translate"
ss = "\n36 0 translate\nprint"

sq :: Int -> [Char]
```

```
sq num = ss ++ show num

lf :: [Int] -> [Char]
lf list = rowline ++ concat (map sq list)

-- 36 is the size of the postscript square
psmax :: Int
psmax = 36
```

```
-- Rational.hs

-- Simple emulation of rational numbers, used when calculating
-- which line in the design to delete.

module Rational (radd, rsub, rmul, rdiv, rmin, rabs, intval,
                 torat) where

norm :: (Int, Int) -> (Int, Int)
norm (x,y) = (u `div` d, v `div` d)
             where
             u = if y > 0 then x else -x
             v = abs y
             d = gcd (abs u) v

radd, rsub, rmul, rdiv :: (Int, Int) -> (Int, Int) -> (Int, Int)
radd (x,y) (u,v) = norm (x * v + u * y, y * v)
rsub (x,y) (u,v) = norm (x * v - u * y, y * v)
rmul (x,y) (u,v) = norm (x * u, y * v)
rdiv (x,y) (u,v) = norm (x * v, y * u)

rmin (x,y) (u,v) = if a > 0 then (u,v) else (x,y)
                   where (a,b) = rsub (x,y) (u,v)

rabs :: (Int, Int) -> (Int, Int)
rabs (x,y) = if x<0 then (-x,y) else (x,y)

-- if y is zero we have an error condition
intval :: (Int, Int) -> Int
intval (x,y) = x `div` y

--for debugging
--show_rat :: (Int, Int) -> [Char]
--show_rat (x,y) = show x ++ "/" ++ show y

torat :: Int -> (Int, Int)
torat n = (n,1)
```

```
--State.hs

-- Various algebraic types used in the description of
-- the program state, and the setting of Buttons in MGR
-- as their contribution to the Mode: i.e. the middle button
-- is set to rubberband when the program is in Draw mode,
-- otherwise a mouse click returns the coded identity of
-- which button was clicked, together with the coordinates
-- of the cursor at that point.

module State (State(..), Flag(..), Mode(..), Button(..),
              Stamp(..), Sel(..), Board(..), Trans(..), smark,
              setmid, setright, smode, sstamp, ssel, sboards,
              sflag,tilestoput) where

import MGR (setevent)

data Flag = Act | Dsave | Dget | Tsave |
            Tget | T4 | Tclear | Dclear deriving (Eq)

-- the second Board is only non-null when a predefined pattern
-- of tiles is being retrieved
type State = (Mode, Stamp, Sel, (Board, Board),Flag)

-- coordinates of line with associated circles
type Stamp = [([Int],[Int])]

-- square identifiers with their Sel
type Board  = [((Int,Int),Sel)]

-- the current selected orientation
type Sel = Int

data Mode = Draw | Tile | Alter | Help deriving (Eq)

type Trans = State -> [[Char]] -> ([Char], State, [[Char]])
```

```

data Button = M | R | L deriving (Eq) -- the L is for an error,
-- as in fact left button presses don't communicate
-- with the program.

smode :: State -> Mode
smode (m,_,_,_,_)  = m

ssel :: State -> Sel
ssel (_,_,s,_,_)   = s

sstamp :: State -> Stamp
sstamp (_,stamp,_,_,_)   = stamp

sboards :: State -> (Board,Board)
sboards (_,_,_,boards,_) = boards

sflag :: State -> Flag
sflag (_,_,_,_,fl) = fl

-- tilestoput to indicate there are tiles to place when retrieving
-- a pattern
tilestoput :: State -> Bool
tilestoput = not . null . snd . sboards

-- smark used by filefun to set the flag to call the appropriate
-- Dialogue handling
smark :: Flag -> State -> State
smark flag (mode,dlist, sel,tilist,_) =
                                    (mode,dlist,sel,tilist,flag)

setbutton :: Button -> [Char] -> [Char]
setbutton button str = setevent b (c:str++"\n")
        where
        (b,c) = case button of
                  M -> (2,'M')
                  R -> (1,'R')

setmid, setright :: [Char] -> [Char]
setmid   = setbutton M
setright = setbutton R
```

```
-- T4.hs

-- Functions special to the tile 4 button
-- They could be generalised so that the program would allow
-- a user to define a patch of tiles of undetermined size
-- that was to be replicated over the entire tiling area.

module T4 (t4, tile)  where

import MagicNos (xymax)
import Lines        (toright, down, place)

-- a function specifically for the potatoprinting program
-- ss is the square size
t4 :: [[[Int]]] -> [[Int]]
t4 [c1,c2,c3,c4] = c1 ++
                   toright ss c2 ++
                   down ss c3 ++
                   (down ss . toright ss) c4
                   where
                   ss = xymax

-- a tile function specifically for use with t4
tile :: Int -> Int -> Int -> Int -> [[Int]] -> [Char]
tile _ _ _ 0 coords = ""
tile _ _ 0 _ coords = ""
tile x y c r coords =
         col x y r coords ++
         row  (x + 2*xymax) y (c-1) coords ++
         tile (x + 2*xymax) (y + 2*xymax) (c-1)(r-1) coords

col, row :: Int -> Int -> Int -> [[Int]] -> [Char]

col x y 0 coords = ""
col x y n coords = place x y coords ++ col x y' (n-1) coords
                   where
                   y' = y + (2 * xymax)

row x y 0 coords = ""
row x y n coords = place x y coords ++ row x' y (n-1) coords
                   where
```

```
                x' = x + (2 * xymax)

-- Tile.hs

-- Functions used when the program is in Tile or Alter mode
-- Identification of particular tiles to manipulate,
-- the selection of orientation of the design from
-- the eight displayed.
-- Grouping of tiles in eights for use when creating the
-- postscript file to print out.

module Tile (alistind, initalist, mark, unmark, sqid, sqas,
             btlocate, ineights, tpatformat, rot, inv, turn,
             squas, inbox, initstate) where

import Layout       (picbox)
import EscherAreas  (inbigtile)
import MGR          (undo)
import Geometry     (Coords(..), rectangle, inrect)
import Maths        (square)
import Lib          (pam, newas)
import State        (State(..),Mode(..),Sel(..),Board(..),
                     Flag(..),Stamp(..))
import MagicNos (tpxorig, tpyorig, tpxygap)

-- to get the (0,0)..(7,7) part of the state of
-- the tiling area
-- size of the row in the big tile
rowsize :: Int
rowsize = 8

nextrow :: Int -> Int
nextrow n = (n + 1) `mod` rowsize

nop :: (Int, Int) -> (Int, Int)
nop (n1,n2) = if n2 == (rowsize - 1) then (nextrow n1, 0)
              else (n1, nextrow n2)

indlist :: (Int, Int) -> [(Int, Int)]
indlist n1n2 = n1n2 : (indlist . nop) n1n2

alistind :: [(Int,Int)]
alistind = take (square rowsize) (indlist (0,0))

initalist :: [((Int,Int),Int)]
initalist = map (\x -> (x,0)) alistind

initstate :: State
initstate = (Draw,[],1,(initalist,[]),Act)

-- the mark to show the current selection

unmark :: Int -> [Char]
unmark = undo . mark

-- The picboxes are marked by a rectangle 3 pixels away from
-- the picture boundary
mark :: Int -> [Char]
mark 0 = ""
mark n = rectangle [x-3, y-3, x + w + 3, y + h + 3]
         where
         [x,y,w,h] = picbox n

-- to find the x of the top left corner of
-- the square in which the middle button is pressed

tlx, tly :: Int -> Int
tlx = \x -> tpxorig + (((x - tpxorig) `div` tpxygap) * tpxygap)
tly = \y -> tpyorig + (((y - tpyorig) `div` tpxygap) * tpxygap)

-- counting squares to give it an id

tlidx, tlidy :: Int -> Int
tlidx = \x -> ((x-tpxorig) `div` tpxygap)
tlidy = \y -> ((y-tpyorig) `div` tpxygap)

-- sqas -- square associated with
-- refers to tiling area
-- gives top left coordinates of the square

sqas :: Int -> Int -> [Int]
sqas x y = [tlx x, tly y]
```

```
-- sqid -- square id
-- refers to tiling area
-- gives id of the square as reflected in the state

sqid :: [Int] -> (Int,Int)
sqid [x,y] = (tlidy y, tlidx x)

-- squas returns the coordinates associated with a particular
-- tilist square.

squas :: (Int,Int) -> [Int]
squas (ln1,ln2) =
          [tpxorig + ln2 * tpxygap, tpyorig + ln1 * tpxygap]

-- btlocate -- locate in the big tile
-- if it's not there gives a default [0,0]

btlocate :: [Int] -> [Int]
btlocate [x,y] = if inbigtile [x,y] then sqas x y else [0,0]

-- for grouping tiles in rows for printing them out

-- This 8 is really rowsize
ineights :: [a] -> [[a]]
ineights [] = []
ineights ns = take 8 ns : ineights (drop 8 ns)

-- For the alter button, rot returns the code of the clockwise
-- rotation of the current code
rot :: Int -> Int
rot n = case n of
          0 -> 0
          4 -> 1
          8 -> 7
          7 -> 6
          6 -> 5
          5 -> 8
          n -> n + 1

-- turn is used in the creation of the postscript file
turn :: Int -> Int
turn n = if n==0 then 0 else
                   (if n == 4 then 8 else (n + 4) `mod` 8)

-- Because of the arrangement of the 8 pictures
-- inv is effectively tbinvert in this version

inv :: Int -> Int
inv = turn

inbox :: Coords -> Int
inbox coords = inbox' 1
                where
                inbox' n =
                  if n > 8 then 0
                  else if inrect w h [x,y] coords then n
                  else inbox' (n+1)
                  where
                  [x,y,w,h] = picbox n

tpatformat :: [[Int]] -> [Char]
tpatformat [] = ""
tpatformat (ln:lns) = formline ln ++ "\n" ++ tpatformat lns
                      where
                      formline (n:ns) =
                            if (ns /= [])
                            then show n ++ " " ++ formline ns
                            else show n
```

```
-- Tmenu.hs

-- The tile area menu defined as an interface

module Tmenu where

import Interface (Interface(..),FindAct(..))
import Layout    (tmc)
import Transact  (mkm,mi,filefun)
import Ttrans    (totile', tofiddle',tclear,t4')
import State     (State(..),Button(..),Mode(..),Trans(..),
                  Board(..),Flag(..),Sel(..),Stamp(..))
import Geometry  (Coords(..))

```

```
-- tmenu is an interface that uses tmc to tell whether
-- a click is within a particular button,
-- and tmas to associate buttons with transactions
-- tmas also associates help text with each transaction
tmenu :: Interface
tmenu = mkm tmc tmas

-- map2 mi [action] [text] is not used as this
-- would divorce the action from its associated help text
tmas :: [Mode -> Button -> Coords -> Trans]
tmas = [mi totile'                              (
              "In this mode, orientations of the print\n"      ++
              "can be placed in the TILE DESIGN area\n"        ++
              "to create an 8 X 8 big tile\n"                  ++
              "\nUse the right button to select a print\n"     ++
              "and the middle button to place it.\n"           ++
              "\nWithin the tiling area the right button\n"    ++
              "may also be used to delete a square\n"          ++
              "\nTo rotate or invert squares within\n"         ++
              "the big tile it may be more convenient\n"       ++
              "to use the ALTER mode."              ) ,
        mi tofiddle'                            (
              "In ALTER mode, squares within the\n"            ++
              "big tile can be adjusted.\n"                    ++
              "The middle button causes them to \n"            ++
              "rotate clockwise.\n"                            ++
              "The right button causes them to\n"              ++
              "invert."                             ) ,
        mi (filefun Tsave)                      (
              "This SAVE button prompts for a filename.\n"     ++
              "It creates a file from which\n"                 ++
              "the actual big tile may be printed out\n"       ++
              "(from outside this program)\n"                  ++
              "with the command: " ++ printcommand             ++
              "\nThe filename can also be used\n"              ++
              "to retrieve the pattern of orientations used\n" ++
              "in the big tile, so that this may be used\n"    ++
              "in conjunction with another print.\n"           ++
              "\n\nThe pattern is retrieved by use of the\n"   ++
              "GET button\n"                        ) ,
        mi (filefun Tget)                       (
              "GET enables previously stored patterns\n"       ++
              "of orientations to be retrieved.\n"             ++
              "Type in the name of the pattern\n"              ++
              "to be retrieved\n"                              ++
              "\n\nIn addition to this there are some\n"       ++
              "predefined patterns that can be imposed\n"      ++
              "on the current print: " ++ predefinedpats       ++
              "\nFor these, type the pattern name\n"           ++
              "preceded by *\n"                     ) ,
        mi tclear                               (
              "This CLEAR button clears the TILE DESIGN\n"     ++
              "region and draws an empty grid." ) ,
        mi t4'                                  (
              "The T4 button tiles the whole big tile\n"       ++
              "with the pattern of the four squares\n"         ++
              "in the top left hand corner\n"  )
       ]


printcommand = "cat filename|lp -apple1"
predefinedpats = "\n        quartet\t     quartets\n" ++
                 "        lwheel\t     rwheel\n"    ++
                 "        wheels1\t     wheels2\n"  ++
                 "        pic1\t    pic2\n"         ++
                 "        escher1\t     escher2\n"  ++
                 "        symcols\t     plain"
```

```
-- Transact.hs

-- Setting up interfaces, and file handling interactions,
-- and basic transactions to be used by active areas.

module Transact (filefun, mi, mkm, helptrans, notrans) where

import State      (Mode(..), Button(..), Trans(..), Stamp(..),
                   Sel(..), Board(..), State(..), Flag(..),
                   smode, smark)
import Geometry   (Coords(..))
import MGR        (clear, invertmode, insertmode, stringto)
import Layout     (vistextreg, clearit, menumark)
```

```
import Help      (errmes, endmes)
import Interface (FindAct(..), Interface(..))

-- filefun sets up an interaction that will involve
-- input from , or output to, a file, by marking the appropriate
-- menu button, and prompting the user for a filename.
-- Has been given dummy coords to fit in with FindAct model
filefun :: Flag -> Coords -> Trans
filefun flag _ state inpt = (outflag, stateflag, inpt)
        where
        outflag   = menumark flag ++ prompt
        stateflag = smark flag state

-- prompt sets up a user interaction area, and prompts
-- the user to type in a filename
-- The prompt is placed just above the text area, hence
-- the use of stringto. The 50 600 relate to the 50 615
-- top left corner of the text area.
prompt :: [Char]
prompt = clearit ++
         vistextreg ++
         invertmode ++
         stringto 0 50 600 "Type in filename: " ++
         insertmode

-- mi is a function for menu items that return help text
-- in Help mode, and an action when the right button is pressed
-- in any other mode

mi :: (Coords -> Trans) -> String ->
                              Mode -> Button -> Coords -> Trans
mi ct str mode button =
        case mode of
              Help -> helptrans (str ++ endmes)
              _    -> case button of
                             R -> ct
                             _ -> notrans

--  mkm makes a menu that is an interface (doesn't draw it)
mkm :: (Int -> [Int] -> Bool) ->
                [Mode -> Button -> Coords -> Trans] -> Interface
mkm = mkm' 0

mkm' :: Int -> (Int -> [Int] -> Bool) ->
        [Mode -> Button -> Coords -> Trans] -> Interface
mkm' _ _ [] = []
mkm' n f (ma:mas) = (FA (f n) ma) : mkm' (n + 1) f mas

helptrans :: [Char] -> Coords -> Trans
helptrans str _ s i = (clear ++ str, s, i)

notrans _ state inpt = (out, state, inpt)
        where
        out = case smode state of
                Help -> clear ++ errmes
                _    -> ""
```

```
-- Ttrans.hs

-- The transactions for the Tile Menu, other than
-- the Save and Get that involve file transactions

module Ttrans where

import State     (Mode(..), Button(..), Trans(..), Stamp(..),
                  Sel(..), Board(..), State(..), Flag(..))
import Geometry  (Coords(..))
import Tile      (initalist, alistind)
import Design    (towcoords, orient, showoris)
import Lib       (assoc, pam, concrep)
import T4        (t4, tile)
import Layout    (chmode, cleara, tilearea, tpgrid,
                  menumark, unmenumark)
import MagicNos  (tpxorig, tpyorig, xymax)
import Interface (FindAct(..), Interface(..))

tclear, t4' :: Coords -> Trans

tclear _ state@(_,dlist,sel,(tilist,_),_) inpt =
    ( menumark Tclear ++
      cleara tilearea ++
      tpgrid          ++
      modestring      ++
```

```
      unmenumark Tclear
    , (Tile,dlist,sel,(initalist,[]),Act)
    , inpt)
    where
    modestring = fst (chmode state Tile)

t4' _ state@(mode,dlist,sel,(tilist,_),_) inpt =
    (out,(mode,dlist,sel,(newtilist,[]),Act),inpt)
    where
    orilist   = pam assoc [(0,0),(0,1),(1,0),(1,1)] tilist
    wcoords   = towcoords dlist
    pic       = t4 (pam (orient xymax) orilist wcoords)
    newtilist = zip alistind (concrep 4 (cr12 ++ cr34))
                where
                cr12          = concrep 4 [n1,n2]
                cr34          = concrep 4 [n3,n4]
                [n1,n2,n3,n4] = orilist
    out       = menumark T4                    ++
                cleara tilearea                ++
                tile tpxorig tpyorig 4 4 pic ++
                unmenumark T4

totile' _ state@(_,dlist,_,_,_) inpt =
        ( concat (map (showoris coords) [1..8]) ++ modestring
        , newstate
        , inpt)
        where
        coords                = map fst dlist
        (modestring, newstate) = chmode state Tile

tofiddle' _ state inpt = (str, newstate, inpt)
        where
        (str, newstate) = chmode state Alter
```

# Appendix B

# Reduction rules for hint

Here are the reduction rules referred to in Chapter 6. The relevant elements of the reduction state are the Stack and Dump, which together constitute the `Stacks` element of the `hint` reduction state, and the nature of the next node to be reduced.

REDUCTION STATE ELEMENTS:

| | |
|---|---|
| $S=[\,]$ or $n : ns$ | — The Stack is empty or $n$ is next |
| $D=[\,]$ or $ss$ | — The Dump, the rest of the stacks, is empty or not |
| $n : [\![ t ]\!]$ | — $t$ is the node at address $n$ |
| $n : whnf$ | — the node at address $n$ is in weak head normal form |

NODE TYPES:

| | |
|---|---|
| $< [\![ v ]\!] >$ | — Value |
| $< [\![ a ]\!]\ n_1\ n_2 >$ | — Apply node with its arguments |
| $< [\![ f^a ]\!]\ n_1 \ldots n_m >$ | — Function $f$ of arity $a$ with $m$ arguments |
| $< [\![ p^1 ]\!]\ n >$ | — Unary primitive with its argument |
| $< [\![ p^2 ]\!]\ n_1\ n_2 >$ | — Binary primitive with its arguments $p_0^2$ — `cons`, `pair`<br>$p_1^2$ — `(&&)`, `(||)`, `(++)`<br>$p_2^2$ — all other binary operators |
| $< [\![ if ]\!]\ n_1\ n_2\ n_3 >$ | — Conditional with its arguments |
| $< [\![ co ]\!]\ n_0 \ldots n_m >$ | — Constructor $co$ with $m$ arguments |
| $< [\![ cs ]\!]\ e\ cp_1 \ldots cp_n >$ | — Case node with discriminating expression $e$ and case pairs $cp_1$ to $cp_n$ |
| $< [\![ cp ]\!]\ co\ e >$ | — Case pair with its distinguishing constructor $co$, and expression $e$ that is an expression to apply to the arguments of the constructor |
| $< [\![ ca ]\!]\ e\ n_0 \ldots n_m >$ | — Case-apply node with the expression to return if there are no arguments, or to apply to these arguments otherwise |
| $< [\![ o ]\!]\ n_1 \ldots n_m >$ | — Output node with list of nodes to be evaluated and output |
| $< [\![ ti ]\!]\ n >$ | — An indirection node "This Is:" address $n$ |
| $< app\ p\ n_1 \ldots n_m >$ | — the node resulting from the application of primitive $p$ to its arguments |

The rules are expressed as follows. On the left is any necessary property of the state. For example

$$D = [\,]$$

means that this rule applies when there are no nodes in the dump. Then the rule depends on the condition above the line. If this holds the next step is the condition below the line. Any output resulting from the step is written on the right of the rule.

REDUCTION RULES:

The end of the program has been reached when there are no longer any nodes to reduce.

$$\frac{S = [\,]\ \ D = [\,]}{END}$$

When the Dump is empty, value nodes have their value output, constructor nodes cause the output of the constructor name and the creation of an `Output` node. `Output` nodes direct arguments of the constructor to the Stack to be evaluated. A partial application of a function is recognised as such.

$$D = [\,]\ \frac{S = n : ns\ \ n : [\![\, v \,]\!]}{S = ns}\{output :\ \ [\![\, v \,]\!]\}$$

$$D = [\,],\ S = n : ns\ \ \frac{n :< [\![\, co \,]\!]\ n_0 \ldots n_m >}{n :< [\![\, o \,]\!]\ \ co\ \ n_0 \ldots n_m >}\{output[\![\, co \,]\!]\}$$

$$D = [\,]\ \frac{S = n : ns\ \ n :< [\![\, o \,]\!]\ co\, n_0, n_1 \ldots n_m >}{S = (n_0 : n : ns)\ \ n :< [\![\, o \,]\!]\ co\, n_1 \ldots n_m >}$$

$$D = [\,]\ \frac{S = n : ns\ \ n :< [\![\, o \,]\!]\ co >}{S = ns}$$

$$D = [\,]\ \frac{S = n : ns\ \ n :< [\![\, f^a \,]\!]\ n_1 \ldots n_m >\ \ (m < a)}{S = ns}\{output :\ \text{``partial application of''}\ [\![\, f \,]\!]\}$$

If there are nodes in the Dump, and the next node to be reduced is in weak head normal form — pop it.

$$D = ss\ \frac{S = n : ns\ \ n : whnf}{S = ns}$$

If the next node to be reduced is an `Apply` node, and its first argument is another `Apply` node, the first argument is pushed on the Stack.

$$\frac{S = n : ns \quad n :< [\![ a ]\!] \, n_1 \, n_2 > \quad n_1 : [\![ a ]\!]}{S = n_1 : n : ns}$$

If the next node to be reduced is a saturated `Closure`, apply the function at the `Closure` node. The next node to reduce is never an indirection node as these are transparent. Such nodes need explicit mention, however, for their role here in the instantiation of the formal parameters of a function application

$$S = n : ns \quad \frac{n :< [\![ f^a ]\!] \, n_1 \ldots n_m > \quad (m = a)}{n :< [\![ f ]\!] \, (a_1 / < [\![ ti ]\!] \, n_1 >) \ldots (a_m / < [\![ ti ]\!] \, n_m >) >}$$

The pattern matching case statement is represented by `Case` nodes which have the constructor on which to match, and a series of case pairs. If the left hand argument of the first pair does not match, the next one is tried. If there are no case pairs left, an error node is created with an appropriate message.

$$S = n : ns \quad \frac{n :< [\![ cs ]\!] \, e \; < [\![ cp_1 ]\!] \, co_1 \, e_1 >, cp_2 \ldots cp_n > \quad e :< [\![ co_e ]\!] \, n_0 \ldots n_m > \quad (co_e \neq co_1)}{n :< [\![ cs ]\!] \, e \, cp_2 \ldots cp_n >}$$

$$S = n : ns \quad \frac{n :< [\![ cs ]\!] \, e \; < [\![ cp_1 ]\!] \, co_1 \, e_1 > \ldots cp_n > \quad e :< [\![ co_e ]\!] \, n_0 \ldots n_m > \quad (co_e = co_1)}{n :< [\![ ca ]\!] \, e_1 \, n_0 \ldots n_m >}$$

$$S = n : ns \quad \frac{n :< [\![ cs ]\!] \, e > \quad e :< [\![ co_e ]\!] \, n_0 \ldots n_m >}{n :< [\![ v : VERR \, (\text{“}Error\ in\ case\ match\ for\text{”} \; [\![ co_e ]\!]) ]\!] >}$$

When there is a case match a `CaseApply` node is created. This applies the right hand side of the case pair to the actual arguments of the constructor on which the match was made, unless the right hand side of the pair is a constant, in which case this is pushed onto the Stack.

$$S = n : ns \quad \frac{n :< [\![ ca ]\!] \, e \, n_0 \ldots n_m > \quad e :< [\![ f ]\!] \, n_1 \ldots n_p >}{n :< [\![ f ]\!] \, n_1 \ldots n_p, n_0 \ldots n_m >}$$

$$S = n : ns \quad \frac{n :< [\![ ca ]\!] \, e > \quad e :< [\![ v ]\!] >}{n :< [\![ v ]\!] >}$$

The conditional primitive evaluates its first argument. If this is `True`, the primitive node is replaced by its second argument; if not, its third.

$$\frac{S = n : ns \;\; n :< [\![\, if \,]\!] \, n_1 \, n_2 \, n_3 > \;\; n_1 : [\![\, v : True \,]\!]}{S = n_2 : ns}$$

$$\frac{S = n : ns \;\; n :< [\![\, if \,]\!] \, n_1 \, n_2 \, n_3 > \;\; n_1 : [\![\, v : False \,]\!]}{S = n_3 : ns}$$

$$\frac{S = n : ns \;\; n :< [\![\, if \,]\!] \, n_1 \, n_2 \, n_3 >}{S = n_1 : n : ns}$$

The constructor primitives `cons` and `pair` create `Cons` and `Pair` nodes without evaluating their arguments.

$$S = n : ns \;\; \frac{n :< p_0^2 \, n_1 \, n_2 >}{n :< app \, p_0^2 \, n_1 \, n_2 >}$$

Unary primitives are head strict.

$$S = n : ns \;\; \frac{n :< [\![\, p^1 \,]\!] \, n_1 > \;\; n_1 : whnf}{n :< app \, p^1 \, n_1 >}$$

$$\frac{S = n : ns \;\; n :< [\![\, p^1 \,]\!] \; n_{1>}}{S = n_1 : n : ns}$$

Left strict primitives need to evaluate their first argument.

$$S = n : ns \;\; \frac{n :< p_1^2 \, n_1 \, n_2 > \;\; n : whnf}{n :< app \, p_1^2 \, n_1 \, n_2 >}$$

$$\frac{S = n : ns \;\; n :< p_1^2 \, n_1 \, n_2 >}{S = n_1 : n : ns}$$

Bi-strict primitives need to evaluate both their arguments.

$$S = n : ns \;\; \frac{n :< p_2^2 \, n_1 \, n_2 > \;\; n_1 : whnf \;\; n_2 : whnf}{n :< app \, p_2^2 \, n_1 \, n_2 >}$$

$$\frac{S = n : ns \;\; n :< p_2^2 \, n_1 \, n_2 >}{S = n_1 : n_2 : n : ns}$$

# Bibliography


[1] S. Abramsky and R. Sykes. SECD-M: a Virtual Machine for Applicative Multiprogramming. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 81–98, Nancy, France, 1985. Springer-Verlag.

[2] P. M. Achten, J. H. G. van Groningen, and M. J. Plasmeijer. High level specification of I/O in functional languages. In John Launchbury and Patrick Sansom, editors, *Functional Programming, Glasgow 1992*, pages 1–17. Springer-Verlag, 1992.

[3] Heather Alexander. ECS – A technique for the formal specification and rapid prototyping of human-computer interaction. In M. D. Harrison and A. Monk, editors, *People and Computers: Designing for Usability*, pages 157–179. Cambridge University Press, 1986.

[4] L. Allison. Circular programs and self-referential structures. *SOFTWARE – Practice and Experience*, 19(2):99–111, 1989.

[5] A. W. Appel, B. F. Duba, and D. B. MacQueen. Profiling in the presence of optimisation and garbage collection. Technical Report CS-TR-197-88, Princeton University, 1988.

[6] A. W. Appel and D. B. MacQueen. A Standard ML compiler. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 301–324. Springer Verlag, 1987.

[7] Kavi Arya. *The Formal Analysis of a Functional Animation System*. PhD thesis, Oxford University PRG, 1988.

[8] Kavi Arya. Processes in a Functional Animation System. In *FPCA '89 Conference Proceedings*, pages 382–395. IBM T.J.Watson Research Centre, 1989.

[9] L. Augustsson and T. Johnsson. The Chalmers Lazy ML compiler. *The Computer Journal*, 32(2):127–141, 1989.

[10] J. Backus, J. H. Williams, E. L. Wimmers, P. Lucas, and A. Aiken. FL Language Manual parts 1 and 2. Technical Report RJ 7100, IBM Research Division, 1989.

[11] David V. Beard and John Q. Walker II. Navigational techniques to improve the display of large two-dimensional spaces. *Behaviour and Information Technology*, 9(6):451–466, 1990.

[12] R. S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21:239–250, 1984.

[13] Richard Bird and Philip Wadler. *Introduction to Functional Programming*. Prentice Hall, 1988.

[14] Anthony Bloesch. Aesthetic Layout of Generalised Trees. *SOFTWARE – Practice and Experience*, 28(3):817–827, 1993.

[15] William H. Burge. *Recursive Programming Techniques*. Addison-Wesley, 1975.

[16] Peter Burger and Duncan Gillies. *Interactive Computer Graphics*. Addison-Wesley, 1989.

[17] R. Burstall, D. MacQueen, and D. Sanella. HOPE: An Experimental Applicative Language. In *Conference Record of the 1980 LISP Conference*, pages 136–143, Stanford, 1980.

[18] Magnus Carlsson and Thomas Hallgren. FUDGETS: A Graphical User Interface in a Lazy Functional Language. In *FPCA '93 Conference Proceedings*, pages 321–330. ACM Press, 1993.

[19] Chris Clack, Stuart Clayman, and David Parott. Lexical Profiling: Theory and Practice. Technical report, University College London, 1993 (to appear in the *Journal of Functional Programming*).

[20] Stuart Clayman, David Parrott, and Chris Clack. A Profiling Technique for Lazy, Higher-Order Functional Programs. Technical Report RN/92/24, University College London, 1991.

[21] S. J. Cook. *Modelling Generic User-Interfaces with Functional Programs*. Cambridge University Press, 1986.

[22] John Darlington. Software Development in Declarative Languages. In Susan Eisenbach, editor, *Functional Programming*, pages 71–85. Ellis Horwood, 1987.

[23] A. J. Dix. Giving control back to the user. *Human-computer interaction – Interact'87*, 1987.

[24] Alan Dix. *Formal methods and interactive systems: Principles and practice*. PhD thesis, University of York, 1988.

[25] Alan Dix, Janet Finlay, Gregory Abowd, and Russell Beale. *Human-computer interaction*. Prentice Hall, 1993.

[26] Andrew Dwelly. Synchronising the I/O behaviour of functional programs with feedback. *Information Processing Letters*, 28:45–51, 1988.

[27] Andrew Dwelly. Functions and Dynamic User Interfaces. In *Proceedings of the conference of Functional Programming Languages and Computer Architecture*, pages 371–381, Imperial College, London, 1989. Addison Wesley.

[28] J. L. Locher (ed). *The world of M. C. Escher*. Harry N. Abrams, Inc, 1971.

[29] M. Eisenstadt and M. Brayshaw. The Transparent PROLOG Machine (TPM): an execution model and graphical debugger for logic programming. *Journal of Logic Programming*, 5(4):277–342, 1988.

[30] Bruno Ernst. *The Magic Mirror of M. C. Escher.* Ballantine, New York, 1976.

[31] George A. Escher. *M. C. Escher at work.* Elsevier Science Publishers B. V., North-Holland, 1986.

[32] Jon Fairbairn. Design and implementation of a simple typed language based on the lambda-calculus. Technical Report 75, University of Cambridge, 1985.

[33] Jon Fairbairn. Making Form Follow Function: An Exercise in Functional Programming Style. *SOFTWARE – Practice and Experience*, 17(6):379–386, 1987.

[34] Joe Fasel, Paul Hudak, Simon Peyton Jones, and Phil Wadler. Special issue on the functional programming language Haskell. *ACM SIGPLAN Notices*, 27(5), 1992.

[35] Sandra P. Foubister. Graphical Design. In C. Runciman and D. Wakeling, editors, *Applications of Functional Programming.* UCL Press, 1995.

[36] Sandra P. Foubister and Colin Runciman. After Escher... Patterning the graphical interface in the functional style. *Proceedings of the International Conference for Young Computer Scientists (ICYCS'91), Beijing*, pages 151–155, 1991.

[37] Sandra P. Foubister and Colin Runciman. Techniques for Simplifying the Visualization of Graph Reduction. In Kevin Hammond, David N. Turner, and Patrick M. Sansom, editors, *Functional Programming, Glasgow 1994*, pages 66–77. Springer, 1995.

[38] J. Gosling, D. Rosenthal, and M. Arden. *The NeWS Book.* Springer-Verlag, 1989.

[39] S. L. Graham, P. B. Kessler, and M. K. McKusick. An Execution Profiler for Modular Programs. *SOFTWARE – Practice and Experience*, 13:671–686, 1983.

[40] Mark Green. A survey of three dialogue models. *ACM Transactions on Graphics*, 5(3):244–275, 1986.

[41] C. V. Hall and J. T. O'Donnell. Debugging in a side effect free programming environment. *ACM SIGPLAN Notices*, 20(7):60–68, 1985.

[42] Cordelia Hall, Kevin Hammond, and John O'Donnell. An algorithmic and semantic approach to debugging in Haskell. In *Third Annual Glasgow Workshop on Functional Programming*, pages 44–53. Springer, 1990.

[43] P. H. Hartel and A. H. Veen. Statistics on Graph Reduction of SASL Programs. *SOFTWARE – Practice and Experience*, 18(3):239–253, 1988.

[44] P. Henderson. Functional programming, formal specification and rapid prototyping. *IEEE Transactions on Software Engineering*, 12(2):241–250, 1986.

[45] Peter Henderson. Functional Geometry. In *Conference record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 179–187. Oxford University PRG, 1982.

[46] Peter Henderson. Purely functional operating systems. In J. Darlington, P. Henderson, and D. A. Turner, editors, *Functional Programming and its Applications*, pages 177–192. Cambridge University Press, 1982.

[47] C. Hewitt. Design of the APIARY for Actor Systems. In *Conference Record of the 1980 LISP Conference*, pages 107–118, Stanford, 1980.

[48] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[49] Paul Hudak and Raman S. Sundaresh. On the Expressiveness of Purely Functional I/O Systems. Technical report, Yale University Department of Computer Science (YALEU/DCS/RR-665), 1988.

[50] John Hughes. Why Functional Programming Matters. *The Computer Journal*, 32(2):98–107, 1989.

[51] S. Johnson. Daisy language manual. Technical report, Computer Science Department, Indiana University, 1987.

[52] Mark P. Jones. Gofer - functional programming environment Version 2.20, 1991.

[53] Samuel Kamin. A debugging environment for functional programming in Centaur. Technical Report 1265, INRIA - Sophia Antipolis, 1990.

[54] R. B. Kieburtz. A Proposal for Interactive Debugging of ML Programs. In *Proceedings of the Workshop on Implementation of Functional Languages*, pages 151–155. Chalmers University of Technology, Programming Methodology Group, Report 17, 1985.

[55] Amir Kishon, Paul Hudak, and Charles Consel. Monitoring Semantics: A Formal Framework for Specifying, Implementing, and Reasoning about Execution Monitors. Technical Report YALEU/DCS/RR-850, Yale University Department of Computer Science, 1991.

[56] P. W. M. Koopman. Interactive Programs in a Functional Language: A Functional Implementation of an Editor. *SOFTWARE – Practice and Experience*, 17(9):609–622, 1987.

[57] P. J. Landin. A correspondence between ALGOL 60 and Church's Lambda Notation. *Communications of the ACM*, 8(2):89–101, 1965.

[58] H. Lieberman. Steps Toward Better Debugging Tools for Lisp. In *ACM Symposium on LISP and Functional Programming*, pages 247–255, 1984.

[59] Robin Milner. A Proposal for Standard ML. In *ACM Symposium on LISP and Functional Programming*, pages 184–197, Austin, Texas, 1984.

[60] James H. Morris. Real programming in functional languages. In J. Darlington, P. Henderson, and D. A. Turner, editors, *Functional Programming and its Applications*, pages 129–176. Cambridge University Press, 1982.

[61] Alan Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs.* PhD thesis, University of Edinburgh, 1981.

[62] H. Nilsson and P. Fritzson. Algorithmic Debugging of Lazy Functional Languages. In M. Bruynooghe and M. Wirsing, editors, *Programming Language Implementation and Logic Programming*, pages 385–389, Leuven, Belgium, 1992. Springer Verlag.

[63] J. T. O'Donnell. Dialogues: A basis for constructing programming environments. *SIGPLAN Notices*, 20(7):19–27, 1985.

[64] J. T. O'Donnell and C. V. Hall. Debugging in applicative languages. *Lisp and Symbolic Computation*, 1:113–145, 1988.

[65] Nigel Perry. I/O and Inter-language calling for Functional Languages. In *Proceedings of the XVth Latin American Conference on Informatics*, 1989.

[66] Nigel Perry. *The implementation of practical functional programming languages*. PhD thesis, Imperial College of Science, Technology and Medicine, 1990.

[67] Simon Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[68] Simon Peyton Jones. UK research in functional programming. *SERC Bulletin*, pages 24–25, 1992.

[69] Simon Peyton Jones and David Lester. *Implementing functional languages: a tutorial*. Prentice Hall, 1992.

[70] Simon Peyton Jones and Philip Wadler. Imperative Functional Programming. In *ACM Conference on the Principles of Programming Languages*, pages 71–84, 1993.

[71] UNIX programmer's manual. `prof` command, 1979.

[72] Alastair Reid and Satnam Singh. Implementing fudgets with standard widget sets. In J. T. O'Donnell and K. Hammond, editors, *Proceedings of the 1993 Glasgow Workshop on Functional Programming*, pages 222–235. Springer, 1994.

[73] Colin Runciman. From abstract models to functional prototypes. In M. Harrison and H. Thimbleby, editors, *Formal methods in human-computer interaction*, pages 201–232. Cambridge University Press, 1990.

[74] Colin Runciman and David Wakeling. Heap profiling of a lazy functional compiler. In *Functional Programming, Glasgow 1992*, pages 203–226. Springer-Verlag, 1992.

[75] Colin Runciman and David Wakeling. Heap profiling of lazy functional programs. Technical Report 172, University of York, 1992.

[76] Patrick Sansom. *Execution profiling for non-strict functional languages*. PhD thesis, University of Glasgow, 1994.

[77] Patrick M. Sansom and Simon L. Peyton Jones. Profiling Lazy Functional Languages. Technical Report (draft), University of Glasgow, 1992.

[78] Robert. W. Scheifler and Jim Gettys. The X Window System. *ACM Transactions on Graphics*, 5(2):79–109, 1986.

[79] Liang Sheng. *Yale Haskell X Interface*. Yale University, 1993. (distributed with the Yale Haskell compiler).

[80] Robin A. Snyder. Lazy Debugging of Functional Programs. *New Generation Computing*, 8:139–161, 1990.

[81] Jan Sparud. Fixing some Space Leaks without a Garbage Collector. In Peter Dybjer, John Hughes, Andy Moran, and Bengt Nordström, editors, *Proceedings of El WinterMöte*. Programming methodology group, University of Göteborg and Chalmers University of Technology, 1993.

[82] William Stoye. *The implementation of functional languages using custom hardware*. PhD thesis, University of Cambridge, 1985.

[83] William Stoye. Message-based functional operating systems. *Science of Computer Programming*, 6:291–311, 1986.

[84] J. Taylor. A System For Representing The Evaluation of Lazy Functions. Technical Report 522, Department of Computer Science, Queen Mary and Westfield College, 1991.

[85] Jon Taylor. *Presenting the evaluation of lazy functions*. PhD thesis, Department of Computer Science, Queen Mary and Westfield College, 1995 (forthcoming).

[86] Simon Thompson. Writing Interactive Programs in Miranda. Technical report, University of Kent at Canterbury (UKC Computing Laboratory Report No 40), 1986.

[87] Andrew P. Tolmach and Andrew W. Appel. Debugging Standard ML Without Reverse Engineering. In *ACM Conference on LISP and Functional Programming*, pages 1–12, Nice, France, 1990. ACM Press.

[88] Ian Toyn. *Exploratory Environments for Functional Programming*. PhD thesis, University of York, 1987. YCST 87/02.

[89] Ian Toyn and Colin Runciman. Adapting combinator and SECD machines to display snapshots of functional computations. *New Generation Computing*, 4:339–363, 1986.

[90] Edward R. Tufte. *Envisioning information*. Graphics Press. Cheshire, Connecticut, 1990.

[91] D. Turner. Functional programs as executable specifications. In C. A. R. Hoare and J. C. Shepherdson, editors, *Mathematical logic and programming languages*, pages 29–54. Prentice Hall, 1985.

[92] D. A. Turner. A new implementation technique for applicative languages. *SOFTWARE – Practice and Experience*, 9(1):31–50, 1979.

[93] D. A. Turner. Miranda: A Non-Strict Functional Language with Polymorphic Types. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 1–16, Nancy, France, 1985. Springer-Verlag.

[94] David Turner. SASL Language Manual. Technical Report CS/79/3, University of St.Andrews, Department of Computational Science, 1976 (revised 1979,1983).

[95] Stephen A. Uhler. *MGR – C Language Application Interface*. Bell Communication Research, 1988.

[96] Marko van Eekeken, Halbe Huitema, Eric Nöcker, Sjaak Smetsers, and Rinus Plasmijer. *Concurrent Clean language manual*. University of Nijmegen, 1993.

[97] Jean G. Vaucher. Pretty-printing of trees. *SOFTWARE – Practice and Experience*, 10:553–561, 1980.

[98] P. Wadler. Fixing Some Space Leaks with a Garbage Collector. *SOFTWARE – Practice and Experience*, 17(9):595–608, 1987.

[99] Philip Wadler. How to Replace Failure by a List of Successes. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 113–128, Nancy, France, 1985. Springer-Verlag.

[100] Philip Wadler. Comprehending Monads. In *ACM Conference on LISP and Functional Programming*, pages 61–78, Nice, France, 1990. ACM Press.

[101] Philip Wadler. The essence of functional programming. In *ACM Conference on the Principles of Programming Languages*, pages 1–14. ACM Press, 1992.

[102] C. P. Wadsworth. *Semantics and pragmatics of the lambda calculus*. PhD thesis, Oxford University PRG, 1971.

[103] S. C. Wray. Implementation and programming techniques for functional languages. Technical Report 92, University of Cambridge Computer Laboratory, 1986.