Lazy Functional Components for Graphical User Interfaces

Rob Noble, BSc

Thesis submitted for the degree of DPhil in Computer Science

Department of Computer Science
University of York

November 1995

11

Abstract

The subject of this Thesis is the programming of applications that use a graphical user interface (GUI) in a lazy functional language.

In graphical user interfaces, many possible paths of interaction are open to the user at once. The central idea of this Thesis is to add concurrent processes to a lazy functional language, allowing GUI programs to be expressed more naturally.

An extended review includes an in-depth case study comparing two of the most advanced previous solutions. An extension to a lazy functional language facilitates the definition of concurrent processes that communicate through typed channels. Interface objects are made more re-usable by parameterising them on the communication channels used. An implementation of concurrent processes in the Gofer interpreter ensures static type-checking of process communication and composition. Unlike previous systems, this Thesis also formulates a screen manager within the functional setting. The validity of the approach is demonstrated through a series of application programs. These include the case-study used to measure two earlier systems, a board game, and a simulator for programs written for another system.

Contents

List of Figures

Acknowledgements

This work was supported by a Research Studentship from the Engineering and Physical Sciences Research Council, and was undertaken at the Department of Computer Science at the University of York.

I am indebted to my supervisor, Colin Runciman, whose enthusiasm, inspiration and patient guidance have kept me on target throughout.

Thanks also to Roger Took, who acted as my assessor and also supervised me for six months whilst Colin was away.

Malcolm Wallace provided an early process implementation, and was always ready to answer my fun programming questions from his desk beside mine.

Beyond York University, my colleagues (competitors?) in the world of functional GUIs have inspired and challenged me. Our discussions have always been useful. They include Sigbjørn Finne, Thomas Hallgren, Magnus Carlsson and all who attended the Glasgow GUIFest in July 1995. I thank the people I visited at the universities of Glasgow, Bristol and Kent, whose puzzled looks in response to my description of *System B* led to the development of *System C*.

Miscellaneous distractions from others in and out of the department prevented me from becoming permanently adhered to my computer screen. Thanks go to all those I have spent my lunch break with during my three years at York University. Andrew Hague frequently interrupted my train of thought with cups of coffee and computer games. Jenny Sykes has remained a friend since my undergraduate days and I appreciate our regular email conversations.

Finally I would like to thank my wife, Jackie, for all her love and support. I have benefited greatly from her encouragement and cheerful outlook. She has put up with my frequent absences (especially of mind), and kept me sane during the last throws of writing-up. I look forward to the next phase of our life together.

Author's Declaration

An earlier version of the material in Chapter 2 appeared in technical report YCS-223 (1994), published by the Department of Computer Science, University of York.

A paper presented at the Seventh International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP '95), entitled *"Gadgets: Lazy Functional Components for Graphical User Interfaces"*, contained some of the material now presented in Chapters 3 to 6.

Unless otherwise stated in the text, the ideas presented in this thesis are solely the work of the author.

Chapter 1

Introduction

Lazy Functional Languages have a number of advantages over strict functional and imperative languages. Peyton Jones sums up the advantages:

> *Functional programs are exceptionally concise and the absence of side-effects makes them easier to reason about than their imperative counterparts, as well as offering interesting opportunities for parallel execution* [Jon92].

These are the overall strengths of functional languages. A paper by Hughes explains another strength of functional languages — they provide a new kind of *glue* for putting together programs:

> *Our ability to decompose a problem into parts depends directly on our ability to glue solutions together. To assist modular programming, a language must provide good glue. Functional programming languages provide two new kinds of glue — higher-order functions and lazy evaluation. Using these glues, one can modularise programs in new and exciting ways, and we have shown many examples of this* [Hug90].

Whilst maintaining both these kinds of glue we add a third — the ability to break down a program into *concurrent processes*. This Thesis shows many examples of GUI programs that benefit from this form of modularisation.

Graphical user interfaces have been in use for many years now and are the norm in computer systems from home micros to workstations. If the benefits of functional languages are to be used in the "real world" then it must be possible for the programmer to provide the user with this type of interface.

At first sight, I/O in a pure functional language is not as straightforward as in imperative languages. In order to make use of lazy evaluation we must have referential transparency — there must be no side-effects. Most languages make use of side-effects for I/O. For lazy functional languages we must adopt some other method.

1.1 Functional Languages and GUIs

Previous solutions to I/O in lazy functional languages have been used in a number of different cases to interface functional languages to graphical user interfaces (usually The X Windows System [SG86]). The first of these attempts were simple and limited simulations of the early methods used in imperative languages to perform graphical interaction and did not make much use of the advantages of functional languages (eg. [Sin91, Sin89]).

More recent systems [CH93, PvE95] approach the problem of specification of a graphical interface from a different angle — one to which functional languages are better suited. The Fudgets system [CH93], for example, makes use of combinators, composition and abstraction to separate interface components from the event-processing loop.

Most current systems interface to The X Windows System or an X toolkit. There has been little thought about what it would mean to shift the functional/imperative borderline further towards the window manager.

1.2 The Aims and Claims of This Thesis

Our aims can be summarised as follows:

- to enable programs with GUIs to be written in a lazy functional language;
- to retain the benefits of lazy functional languages in such programs, and exploit these benefits in writing GUI programs;
- to move the boundary between functional program and imperative world towards the computer hardware.

The central claims of this thesis are:

- the addition of concurrent processes to a lazy functional language enables programs with a GUI to be expressed more naturally;
- process definitions are made more re-usable by parameterising them on the channels used to communicate with other processes. Process compositions are made more concise by the use of higher-order functions such as `map` to make multiple connections.
- allowing processes to receive from multiple sources in a non-deterministic manner benefits GUI programming;
- the boundary between the functional and imperative worlds can be *pushed back* by formulating a screen manager in the functional domain;

1.3 Structure of this Thesis

Chapter 2 begins by reviewing the main methods of incorporating I/O into a lazy functional language. A summary of work to integrate GUIs into functional languages is given, with a detailed review and case study involving two of these systems. We describe some desirable features of functional GUI systems, and measure the reviewed systems against them.

Chapter 3 argues that the addition of concurrent processes makes a lazy functional language more suitable for programming GUIs. We present a concurrent process extension to the lazy functional language Gofer. Communication between processes is through typed message channels that are passed as parameters to process definitions. Processes are capable of receiving messages from multiple sources in an indeterminate order. This frees the programmer from having to consider every sequence of events in a system in which the user affects the choice of evaluation order.

Chapter 4 discusses an implementation of concurrent processes. It consists of an extension to the Gofer interpreter and supports dynamic creation of processes and typed channels. The primitives chosen ensure statically type-checked communication between processes. The scheduling scheme is devised to minimise the amount of heap space taken by message queues.

Chapter 5 addresses the issue of screen management in a lazy functional language. A screen manager process is presented, programmed entirely in the functional lan-

guage. The screen manager uses the ability to pass functions in messages to remove unintuitive features from GUI programs, such as the need for exposure events.

Chapter 6 demonstrates the success of the completed system with a suite of example applications. Simple examples illustrate Gadget composition, layout, resizing, data sharing, use of concurrency and viewports. The case-study of Chapter 2 is revisited, providing a means to compare the style of programming with other systems. In support of our claim to generalise on the Fudgets system, a simulator enables Fudget programs to evaluate under the Gadgets system.

Chapter 7 presents overall conclusions and identifies possible lines of future work on the experimental system described in earlier chapters.

Chapter 2

Review

2.1 Introduction

In this chapter we review first the integration of general I/O into functional languages (§2.2), and see the solutions that have been proposed. We illustrate these with examples, and weigh up the relative merits. We look briefly at some of the graphical user interfaces that have been used with functional languages in §2.3. In §2.3.1 and §2.3.2 we look critically at two of these systems, and implement an example application in each. In §2.4 we consider a larger application as a case study. In §2.5 we look at some of the most recent work. We finish the chapter by listing some features of a good environment for graphical applications and measure these systems against them.

2.2 I/O in Lazy Functional Languages

A first guess at the means by which to obtain I/O in a functional language might be to use functions such as:

```
getChar :: Channel -> Char
```

which, given a channel identifier (eg. `<k>` might be a channel for the keyboard), returns the next character from the channel. However, this function is clearly not referentially transparent – identical applications may return different values at different points in a computation. Using a function such as this would cause problems. Suppose a function `readLine` uses `getChar` to read a line of characters from the keyboard, returning these as a String. If we try to evaluate

```
elem (getChar <k>) readLine
```

where `elem` is the standard function that returns `True` if its first argument is an element of its second, which would the user enter first, the character or the line? If the arguments were evaluated in parallel, the character might be read in the middle of reading the line, or an error might occur (depending on how contention for the keyboard channel were handled).

It might be possible to circumvent these problems by enforcing a rule that, for example, the arguments of a function that invoke I/O must be evaluated in left to right order. However, any function that *uses* an I/O function would also be considered to invoke I/O, and would thus suffer the same restriction. This effect could propagate through large portions of the program, enforcing a strict order of evaluation, losing the benefits of laziness.

2.2.1 Some Solutions

What is needed is a method that maintains referential transparency. We look at a number of solutions and see their relative advantages and disadvantages. The solutions we look at are streams, continuations, systems, history and monads.

Streams

We can use *lazy evaluation* to enable us to take the whole of the user's input as an argument, and return the whole of the program's output as the result. The input argument and output result will consist of lazy lists of characters, where each element (character) of the output is demanded in sequence by the output driver.

```
type Input = [Char]
type Output = [Char]
```

Laziness ensures that output may be interleaved with input — otherwise, the program would be a batch program requiring all of the user's input before producing any output. As an example, consider the function:

```
toUpper :: Char -> Char
```

that takes a character and, if it is a lower case letter, returns the upper case equivalent, or otherwise just returns the same character. We can use this function to make an interactive program:

```
upperCase :: Input -> Output
upperCase = map toUpper
```

For example (assuming that the terminal is echoing user input and buffering it line by line):

```
> upperCase <k>
one two
ONE TWO
Buckle My Shoe
BUCKLE MY SHOE
^D
> _
```

Combining I/O Functions To make more complex interactive programs, we need some higher-order combining functions (*combinators*) with which to glue together the basic building blocks (such as `upperCase`). Combinators are described in [O'D85], [Tho86] and [Dwe89]. We look at some of the combinators that Thompson described. Firstly, we define a slightly different type for an interactive function, to accommodate the passing of information between blocks of a program:

```
type Inter a b = (Input, a) -> (Input, b, Output)
```

That is, an interactive function takes as input the stream of input from the user and an initial state of some type `a`, and returns the *remainder* of the input, a final state of type `b`, and any output that is generated. The remainder of the input equals the input list of characters *less* any characters the function uses from the start of this list. The first combinator, `seq`, combines two interactive functions so that the actions they describe occur in sequence, one after the other:

```
seq :: Inter a b -> Inter b c -> Inter a c
seq one two (input, initState)
    = (rest, finalState, outOne ++ outTwo)
    where
        (midInp, midState, outOne) = one (input, initState)
        (rest, finalState, outTwo) = two (midInp, midState)
```

Other useful combinators include:

```
alt :: (a -> Bool) -> Inter a b -> Inter a b -> Inter a b
```

that takes a *test function* and two interactive functions and acts like the first function if the test applied to the state returns *True*, otherwise it acts like the second function. The `while` combinator:

```
while :: (a -> Bool) -> Inter a a -> Inter a a
```

takes a test function and an interactive function and produces an interactive function which operates repeatedly while the test on the state evaluates to *True*. The state returned by the function is fed back into the same function for the next iteration, hence the reason both input and output state types are labelled `a`. The function `apply`:

```
apply :: (a -> b) -> Inter a b
apply fn (inp, val) = (inp, fn val, "")
```

takes an ordinary (not interactive) function, `fn`, and turns it into one of interactive type, that does no input or output, but applies `fn` to the state.

An Example: Calculator We now use the combinators defined above to create a simple calculator. The user repeatedly enters a sum such as `3+4=` or `20+35+152=` and the program responds with the total as the answer.

To avoid nested brackets, we define a bit of *syntactic sugar*:

```
a $ b = seq a b
```

We may now write `in $ out` to mean *perform the interaction* `in` *followed by the interaction* `out`. The calculator program is shown in Fig. 2.1.

So, `calc` is a sequence of `zeroTotal` followed by `calc'`. The function `zeroTotal` passes on a state of integer zero representing the initial total held by the calculator. Function `calc'` is a `readInt` followed by an `addTotal` followed by a `readOp` followed by a choice depending on whether the last operator read was a '`+`' or a '`=`'. If it was a plus, the total is passed back to `calc'` for another iteration; otherwise, the total is displayed and we continue with the original calculator (with zeroed total).

In this style, it is fairly easy to see the flow of the interaction (eg. `readInt`, then `addTotal`). It is possible to place a debugging function between two blocks of an interaction to enable the programmer to see intermediate values. For example, to see the value of the total after `addTotal` the `calc'` function might be changed to:

```
calc' = readInt $ addTotal $ debug $
        readOp $ alt isPlus
                  (forwardTotal $ calc')
                  (showTotal $ calc)

debug (i, t) = (i, t, ("\ndebug: " ++ show t ++ "\n"))
```

```
calc :: Inter () ()
calc = zeroTotal $ calc'

calc' :: Inter Int ()
calc' = readInt $ addTotal $ readOp $ alt isPlus
                                          (forwardTotal $ calc')
                                          (showTotal $ calc)

zeroTotal :: Inter () Int
zeroTotal (i, ()) = (i, 0, "")

readInt :: Inter Int (Int, Int)
readInt (i, t) = (rest, (t, n), out)
                 where (n, out, rest) = parseInt

addTotal :: Inter (Int, Int) Int
addTotal (i, (t, n)) = (i, t+n, "")

readOp :: Inter Int (Int, Char)
readOp (i, t) = (rest, (t, op), out)
                where (op, out, rest) = parseOp

isPlus :: (Int, Char) -> Bool
isPlus (t, o) = o == '+'

forwardTotal :: Inter (Int, Char) Int
forwardTotal (i, (t, o)) = (i, t, "")

showTotal :: Inter (Int, Char) ()
showTotal (i, (t, o)) = (i, (), show t)
```

`atoi` and `show` are standard functions that convert integers to strings and vice-versa. `parseInt` and `parseOp` are input-parsing functions that return a tuple of the required integer/operator, the text to echo to the screen as output, and the remaining user-input after parsing.

Figure 2.1: Streams Calculator

If the program is changed so that the state passed between blocks encompasses all the information ever needed by the calculator (of the same type throughout the program) then the `seq` combinator may be changed to display the state after each block, and the program itself need not be changed at all. However, a state containing

all the values needed for an interaction might be likened to global variables in an imperative program, with all the disadvantages that these hold.

I/O Pitfalls The programmer needs to be aware of a problem (described in [Tho86, Tho87] and [Dwe87]) that may occur with lazy stream-based I/O (and also the continuations and systems models that are discussed later). For example, suppose we require a program that prompts the user for a name, eg. "Rob", and then replies with "`Hello Rob`".

```
Name? Rob
Hello Rob
```

This program might be written as in Fig. 2.2. The `()` symbol represents the null type and is used here where no state information is to be passed in or out of an interactive function. Function `splitat` takes a character `c` and a string `s` and returns two strings, the first being the front of `s`, up to the first occurrence of `c`, and the second being the rest of `s`. Thus in this example, `n` returned from `splitat` will be the first line of the user's input, and `r` will be the rest of the user's input.

```
prog :: Inter () ()
prog = ask $ read $ say

ask::Inter () ()
ask (i, x) = (i, x, "Name? ")

read::Inter () String
read (i, x) = (r, n, "")
      where (n, r) = splitat '\n' i

say::Inter String ()
say (i, n) = (i, (), "Hello "++n)
```

Figure 2.2: Pitfall Program

When the program is executed, what is displayed?

```
Name? Hello Rob
Rob
```

Not what is required! The "`Hello `" part of the output has skipped ahead of the user's input because `say` can evaluate the "`Hello `" part of its result without needing

the user's input. Dwelly solved this problem [Dwe87] by turning off the terminal's echoing of user input and requiring that the program echo what the user types. This means that both the user's input and the program's output appear in the output stream, and hence the interleaving of input and output is defined. In the example above, `read` is re-written as:

```
read :: Inter () String
read (i, x) = (r, n, n) where (n, r) = splitat '\n' i
```

Notice that the name returned as the final state is also returned as the output from `read`. It is important that `splitat` constructs the first element of its result as the characters of its argument become available, rather than once it has found the '`\n`' character, otherwise the user's input would not be echoed until return is pressed.

Thompson [Tho86], suggests another solution. A `wait` function can be used with the definition:

```
wait :: Eq a => Inter a a
wait (in, x) | x==x = (in, x, "")
```

The guard `x==x` causes `x` to be evaluated before it is available as the final state of the `wait` interaction. Our example program becomes:

```
prog = ask $ read $ wait $ say
```

Thompson proves using traces ([Tho87]) that using his combinators I/O occurs in the correct sequence.

Stoye [Sto85b] solves the problem by extending the streams model to streams of requests/responses (instead of simply streams of characters) and requires that each character of input is requested before it is received as a response.

Continuations

An alternative for gluing together functions in an interactive program is to use the *continuation passing style* (CPS), as described in [HM89] and [Per92]. First, as a non-interactive CPS example, consider the function:

```
add2 :: Int -> Int
add2 num = num + 2
```

In the continuation passing style, this function becomes:

```
add2 :: Int -> (Int -> a) -> a
add2 num cont = cont (num + 2)
```

The function now takes an additional parameter (`cont`), a function that represents the rest of the program. Instead of returning the sum `num + 2`, this is given to the rest of the program (`cont`) as an argument. The result of an I/O CPS function is the output it generates appended with the output of the rest of the program. As `add2` does not produce any output, its result is just the output from the rest of the program — the result of `cont`.

Calculator in CPS In the CPS, the output function in the calculator example becomes:

```
showTotal :: Int -> (Input -> Output) -> Input -> Output
showTotal total cont inp = show total ++ "\n" ++ cont inp
```

The value returned from this function is the total converted to a string appended with the output generated by the rest of the program.

The calculator in CPS is shown in Fig. 2.3. Notice that the type of the calculator is still the same as in the streams version. Function `calc` uses an auxiliary function `calc'` with the parameter `0` to start the calculator with a zeroed total. Function `calc'` reads an integer from the user's input; its continuation is a function which adds this number to the running total. `AddTotal`'s continuation reads an operator ('`+`' or '`=`') from the user's input, but passes that operator to its continuation function, `opAct`, which either passes the total on to its continuation or displays the total and passes a zeroed total on to its continuation.

The types of CPS functions may be complicated, but they reveal what information is passed on to the continuation function. A state consisting of multiple parts need not be packaged into a tuple, as required by the streams combinators. Each part is a separate argument to the continuation function.

Systems

A *system* is a data structure representing the state of all parts of the operating system relevant to I/O. In the systems model of I/O, every function that will perform I/O takes an extra parameter (system state before) and returns an extra value in the result (system state after). The dependency between systems passed to/returned from the functions of a program defines the order in which I/O occurs.

```
calc :: Input -> Output
calc = calc' 0

calc' :: Int -> Input -> Output
calc' t = readInt (addTotal t calc'')

calc'' :: Int -> Input -> Output
calc'' t = readOp (opAct t calc')

readInt :: (Int -> Input -> Output) -> Input -> Output
readInt cont inp = out ++ cont number rest
    where (number, out, rest) = parseInt inp

addTotal :: Int -> (Int -> Input -> Output) -> Int -> Input -> Output
addTotal t cont num = cont (t + num)

readOp :: (Char -> Input -> Output) -> Input -> Output
readOp cont inp = out ++ cont op rest
    where (op, out, rest) = parseOp inp

opAct :: Int -> (Int -> Input -> Output) -> Char -> Input -> Output
opAct t cont op = case op of
    '+' -> cont t
    '=' -> showTotal t (cont 0)
```

Figure 2.3: Continuations Calculator

In the systems model the output function for the calculator example might look like this:

```
type System = (String, String)
showTotal :: System -> Int -> System
showTotal (i, o) t = (i, o ++ show t)
```

The system contains two strings, one the user's input, and the other the program's output. `showTotal` returns the user's input part unchanged, but adds the display of the total to the program's output.

The systems model looks very similar to the streams model, and in this form is almost identical. Where the systems model differs is that the `System` type might contain other information about the I/O system (apart from the input and output stream) such as information relating to a window on a graphical terminal.

The systems model calculator is shown in Fig. 2.4. It is easy to see the order in which I/O operations occur because of the way the systems returned from functions

(`s1`, `s2`, etc.) are used as parameters to other functions. All the extra system parameters tend to clutter the program, but with lazy evaluation they are necessary to ensure the correct order of evaluation, and hence correct order of interaction.

```
calc :: System -> [System]
calc s = calc' 0 s

calc' :: Int -> System -> [System]
calc' t s = sys:rest
      where (tot, sys) = calc'' t s
            rest = calc' tot sys

calc'' :: Int -> System -> (Int, System)
calc'' t s = (t3, s4)
      where (num, s1) = readInt s
            (t2, s2) = addTotal s1 t num
            (op, s3) = readOp s2
            (t3, s4) = opAct s3 op t2

readInt :: System -> (Int, System)
readInt (i, o) = (num, (rest, o ++ out))
        where (num, out, rest) = parseInt i

addTotal :: System -> Int -> Int -> (Int, System)
addTotal (i, o) t n = (t + n, (i, o))

readOp :: System -> (Char, System)
readOp (i, o) = (op, (rest, o ++ out))
        where (op, out, rest) = parseOp i

opAct :: System -> Char -> Int -> (Int, System)
opAct (i, o) op t = case op of
                '+' -> (t, (i, o))
                '=' -> (0, showTotal t)
```

Figure 2.4: Systems Calculator

History

The FL language uses a model similar to the systems model (see [BWW86], and also [HD88] for a similar system in the *Meta* language), but with each function taking an extra *implicit* argument and returning an *implicit* additional value, the history. For example, the function application:

```
y = fn x
```

has an implicit history which, if made explicit, would cause the function application to be written:

```
(y, h') = fn (x, h)
```

where `h` is the history which is passed to the function, and `h'` is the altered history returned. The history represents the history of I/O operations performed, and has a similar role to the *system*, described above.

Because the dependency (of history) between functions is not specified in a program, FL relies on evaluation in a strict left-to-right order to ensure that I/O occurs as desired. As a result, interactive programs in FL are not cluttered with extra history or system parameters as in the systems model, but strict evaluation applies to the whole program and thus the benefits of laziness in the non-I/O parts of the program are lost.

Monads

The monadic model for I/O (described in [Wad90, Wad92, JP93]) consists of functions that represent *actions* and combinators that combine actions into a single action. The value of an interactive program is an action and executing the program will cause the action to be performed. An action may return a value. The type of an action is `IO a`, where `a` is the type of value returned. For example, the action function `putc` outputs a character as its action:

```
putc :: Char -> IO ()
```

Action functions such as `putc` that return no useful value may return the empty tuple `()`, the *null* type.

The two simplest monadic combinators are:

```
doneIO :: IO ()
seqIO :: IO a -> IO b -> IO b
```

Performing the action `x `seqIO` y` results in the action `x` being performed first, followed by action `y`, and the entire action returns the value returned from `y`. Action `doneIO` does nothing and is used in recursive definitions to terminate the recursion.

There are two corresponding combinators used with actions that return a value:

```
unitIO:: a -> IO a
bindIO:: IO a -> (a -> IO b) -> IO b
```

As with `doneIO`, `unitIO` performs no action, but returns the value passed to it. `bindIO` works as `seqIO`, but passes the value returned from the first action to the second action, and returns the value returned from the second.

The `IO` type is defined:

```
type IO a = World -> IORes a
data IORes a = MkIORes a World
```

where type `World` represents the state of the I/O system in much the same way as a *system*. However, the `World` is hidden from the programmer, being dealt with entirely by the combinators. Conceptually, I/O operators perform I/O by returning an altered World state and the runtime system interprets this. In fact, the `World` type contains no data, and is used only to enforce the order in which I/O occurs. I/O actions actually engage in I/O when evaluated, effectively updating the World *in-place*. This is safe because the World value is only handled by the monadic combinators that ensure it is only used in a single thread. By the use of suitable transformations, a compiler avoids generating any code to manipulate `World` data, and can generate code very similar to the equivalent C program.

The monadic style calculator is shown in Fig. 2.5. The style of the calculator in monad form is very similar to the streams or continuations versions. However, looking at one of the actions:

```
addTotal:: Int -> Int -> IO Int
addTotal t n = unitIO (t + n)
```

we can see that the `World` is not mentioned anywhere (it is hidden inside the `unitIO` action), which may be considered an advantage. Also, by using unboxed datatypes [JL91] and a method of calling C language functions from within the functional language, I/O is forced to occur at the point when an action is evaluated, and cannot be left until the resulting value of an action is required (as is normal with lazy evaluation). This means that the synchronisation problems (mentioned in §2.2.1) will not occur with monadic I/O.

The monadic calculator listed in Fig. 2.5 does not show the definition for `readInt`. When parsing a multi-character token such as an integer there is a problem with the basic form of monadic I/O. In previous I/O schemes the `readInt` function takes

characters from the input stream until a non-digit character, `c`, is reached, at which point the rest of the stream including `c` is passed on to the next function. In the implementation of monadic I/O in the Glasgow Haskell Compiler there is no stream of input characters — when a character is required it is read from the keyboard buffer directly (by a call to a C function) and the function which *requested* it must then deal with it. There is no stream of input characters in which to pass the character on to the next function.

One solution is to call to the C function `ungetc` to push the character back into the keyboard buffer, but only if keyboard buffering is in effect. For the calculator example, it was required that keyboard buffering be switched off so that the total could be displayed as soon as the user presses `=`, rather than after pressing Return, so this is not possible.

Alternatively, the `World` type could be changed to carry a *lookahead* character on to further parsing functions. With some real data contained in the `World` type the compiler optimisations mentioned before would be impossible.

Equivalence of Models

Hudak and Sundaresh [HS88] show that streams, continuations and systems, as described above, are equivalent in expressive power. Each may be described in terms of either of the others. However, with streams or systems described over continuations as primitive, there is an unavoidable *space leak*. For this reason, streams were originally chosen as primitive in the functional programming language Haskell, with continuations provided by higher-order *interpreter* functions which convert to streams. More recently, there has been a move towards the I/O monad as primitive, because of its efficiency of operation.

2.3 Graphical User Interfaces

We look briefly at the history of the use of functional languages with graphical user interfaces, and then look in more detail at some of the recent developments.

Functional graphics Henderson's paper on functional geometry started the ball rolling for functional languages used in graphical applications [Hen82a]. He describes a way of representing graphical pictures as data and functions for manipulating and

```
calc :: IO ()
calc = (unitIO 0) `bindIO` calc'

calc' :: Int -> IO ()
calc' t = (readInt `bindIO` (addTotal t)) `bindIO`
             (\t -> readOp `bindIO`
                (\t op -> case op of
                             '+' -> calc' t
                             '=' -> (putnIO t) `seqIO` calc ))

readInt :: IO Int
readInt = (see note in text)

addTotal :: Int -> Int -> IO Int
addTotal t n = unitIO (t+n)

readOp :: IO Char
readOp = getcIO `bindIO` (\c -> (putcIO c) `seqIO` (case c of
        '+'       -> unitIO c
        '='       -> unitIO c
        otherwise -> readOp))

putnIO :: Int -> IO ()
putnIO n = putsIO (show n)

putsIO :: String -> IO ()
putsIO s = foldr seqIO (putcIO '\n') (map putcIO s)
```

A full definition of readInt is not given here — see note in the text.

Figure 2.5: Monadic Calculator

combining pictures. The applications are not interactive, but use abstraction to structure a picture hierarchically.

Graphics and processes Arya took this lead and combined it with work on functional representations of processes [Hen82b, Sto86] to produce a functional animation system [Ary89] in which graphical objects are described by processes that may interact with each other, but not with the user.

Connection to a window manager A number of papers describe methods of connecting a functional language interpreter or compiler to a graphical user inter-

face. Singh describes an interface between the functional language Miranda and XView/X11 [Sin91]. He explains the technical method of connecting the two (using UNIX sockets to connect two processes, one running the Miranda interpreter and the other the window manager), but does not look much at the structure of an application written under such a system. The system does not support the use of *callbacks* as an equivalent C program would and provides no useful alternatives.

Sinclair did some similar work connecting the language LML to The X Window System [Sin89]. In his system a dialogue (based on a simplified subset of the X client-server dialogue) is communicated between the functional program and the window manager.

Sinclair developed this idea further into a system where the interface is programmed in an imperative language and communicates with the functional language via a dialogue [Sin92]. The interface can perform simple functions without need for communication with the functional side of the application.

The Yale Haskell compiler has an interface to The X Windows System by way of the I/O monad [Yal93] giving access to the Common Lisp X interface (CLX). Applications have a similar structure to imperative equivalents, with event processing loops, etc.

Using processes in GUI programs The Erlang language [AWV93] and the language CML with its X interface, eXene[GR93], are both based on defining concurrent processes. Both languages are functional, but have side effects, so are not *pure* functional languages.

Structuring GUI programs in a functional language Some papers consider the structuring of a graphical application in a functional language. Dwelly's paper [Dwe89] defines an `allcase` combinator that combines event processing functions for all possible events into a function that when recursively applied to a list of input events, generates a list of reactions to them. He also defines a similar *treecase* combinator in which the list of events the program can respond to changes from one iteration to the next (to cater for menus popping up on the screen, etc.).

Foubister and Runciman [FR91] do similarly with *transaction combinators* to build applications in LML under the "MGR" window manager.

Reid and Singh [RS93] encapsulate standard X Widgets inside Fudget wrappers (see the review of Fudgets in §2.3.1) so as to make use of existing Widgets without

having to reprogram their entire behaviour from the bottom up in the Fudget style. They call them *Budgets* because of the resultant saving in the amount of library programming needed. They suffer the same plumbing problems as the Fudgets system.

2.3.1 The Fudgets System

The Fudgets system [CH93, HM95] is a graphical user interface (GUI) toolkit for The X Window System written in the lazy functional language LML. The programmer may use it to write graphical applications in Haskell or LML, utilising the expressive power of functional languages to the full. This review was originally written based on version h3 of Fudgets with version 0.999.1 of the LML distribution. It has since been updated for version h9 of Fudgets and version 0.999.6 of the Chalmers Haskell compiler.

The toolkit uses the idea of a *Fudget* as its main abstraction — the functional equivalent of the *widget* used in imperative windows toolkits for The X Windows System and others. A Fudget may be thought of as a process (normally associated with an object on the screen, such as a button or a menu) that communicates with other Fudgets by message passing. A Fudget of type `F a b` receives messages of type `a` and sends messages of type `b`. Fig. 2.6 shows a diagrammatical representation of a Fudget. Note that the message streams are marked with the type of the messages they convey, not the type of the message stream.

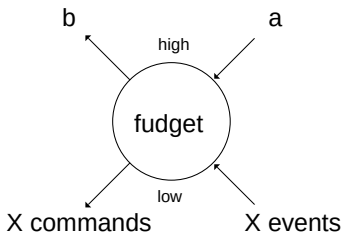


Figure 2.6: A Fudget of Type `F a b`

A Fudget has two streams through it, one for high-level messages and and one for low level messages. High level messages are used by the application program to convey pieces of information from one part of the program to another (such as an integer that has been entered into a box on the screen by the user); low level messages take the form of window events and commands (such as an event signifying that the

mouse has been clicked in a box on the screen) and are usually only manipulated by the library Fudgets.

Fudgets are hierarchical — they may be combined (using Fudget *combinators*), to form more complex Fudgets. A library of basic Fudgets implements a host of useful screen objects such as push buttons and menus. The library Fudgets deal with most of the low level messages that occur, and the Fudget combinators deal with routing messages to the correct Fudget. For the most part, Fudgets can be regarded as having their low level message stream directly connected to the relevant object on the screen (as in Fig. 2.7).

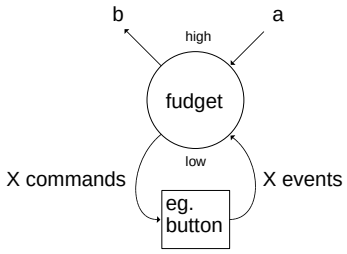


Figure 2.7: Low Level Fudget Messages

Fudget Combinators Perhaps the simplest combinator to understand is `>==<`. This is similar to the `seq` combinator for textual I/O, described by Thompson [Tho86]. It is best described with the aid of a diagram (see Fig. 2.8).

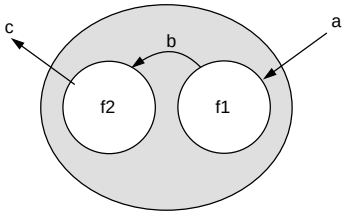


Figure 2.8: `f1 >==< f2`

Here (and in what follows) Fudgets diagrams are drawn without showing the low level stream unless necessary, and it should be assumed that low level messages are being routed to and from the relevant object on the screen. The dashed line denotes the Fudget constructed by the combination of Fudgets `f1` and `f2`. Fudget `f1` must output messages of the type Fudget `f2` expects to receive.

Using this combinator alone would only allow the Fudgets of an application to be connected in one long stream. The `>+<` combinator allows a Fudget to send a message to, or receive a message from, either of two other Fudgets using the `Sum` type `Either a b = Left a | Right b` (see Fig. 2.9).

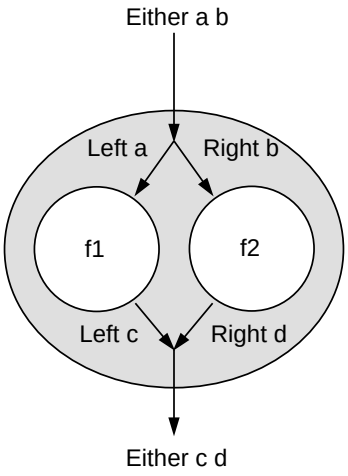


Figure 2.9: `f1 >+< f2`

Given three Fudgets, `f1`, `f2`, and `f3`, and using both the combinators we have seen so far, we may now combine the Fudgets so that `f1` may send messages to both `f2` and `f3`, expressing this as `(f2 >+< f3) >==< f1` (see Fig. 2.10).

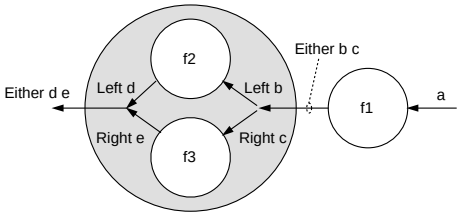


Figure 2.10: `(f2 >+< f3) >==< f1`

To combine them so that `f1` may receive messages from both `f2` and `f3` we use `f1 >==< (f2 >+< f3)` (see Fig. 2.11).

It is not possible to connect the output of a Fudget back to its own input using these combinators. To do this we must use one of the family of looping functions. For example, `loopLeft` applied to a Fudget of type `F (Either a b) (Either a c)` will produce a Fudget of type `F b c` in which the left-hand side of the output (of type `a`) is looped back to the left-hand side of the input (see Fig. 2.12).

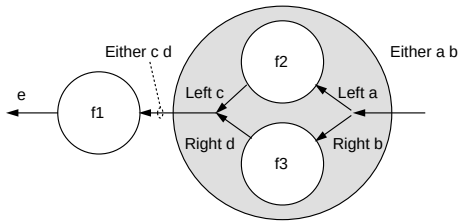


Figure 2.11: `f1 >==< (f2 >+< f3)`

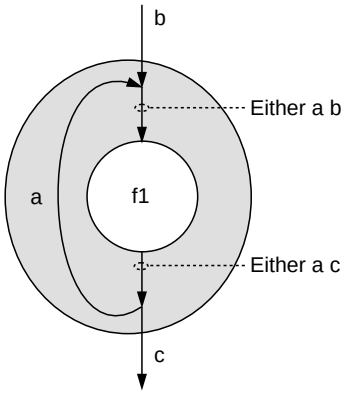


Figure 2.12: `loopLeft f1`

Abstract Fudgets

Most Fudgets have a related screen object, but the programmer may also define *abstract* Fudgets. These connect other Fudgets together, but have no direct effect on the screen display. An abstract Fudget does not use its low-level streams, but only inputs and outputs messages on the high level stream. An abstract Fudget is created by applying the `absF` function to a *stream processor* — the basic building block of processes in the system. Connecting a stream process `sp` to the output of a Fudget `f` with `absF sp >==< f`, or to the input with `f >==< absF sp` is a common requirement, so special combinators are provided for these cases. These combinators, `>^^=<` and `>=^^<`, join a stream processor to a Fudget, ie. `sp >^^=< f` or `f >=^^< sp`.

A common use of abstract Fudgets is to convert the messages in or out of a Fudget from one type to another. For instance we may have one Fudget that outputs words, and wish another Fudget to receive tokens representing the words (eg. "`start`" is converted to `StartTok`). To do this we use a stream processor form of the `map` function. For example, in Fig. 2.13 words output from a `readerF` Fudget

are converted to tokens for an `actionF` Fudget.

```
>^^=< :: SP b c -> F a b -> F a c
>=^^< :: F b c -> SP a b -> F a c

type Token = StartTok | StopTok | EndTok

tokeniseSP :: SP String Token
tokeniseSP = mapSP conv
    where conv :: String -> Token
          conv "start" = StartTok
          conv "stop" = StopTok
          conv "end" = EndTok

actionF >=^^< tokeniseSP >^^=< readerF
```

Figure 2.13: A Tokenising Stream Processor

In fact, the use of `mapSP` in abstract Fudgets is so common that a further pair of combinators, `>^=<` and `>=^<` are provided to connect a mapping function of type `a -> b` to a Fudget:

```
>^=< :: (a -> b) -> F c a -> F c b
>=^< :: F b c -> (a -> b) -> F a c
```

Another use of abstract Fudgets is to accommodate the parts of the program that are not related to the user interface. For example, in a database program, the parts that search the database might be contained in an abstract Fudget.

Layout

The layout of a set of Fudgets on the screen can be defined. The simplest way is for each Fudget to emit a message on its low level stream containing a pair of coordinates for the positions of its top-left and bottom-right. It is a laborious task for the programmer to calculate all these coordinates for a set of Fudgets, so a tool could be used to place the Fudgets interactively, using the mouse. A simple version of such a tool has been developed [Ahl93]. Alternatively, the Fudget system provides no less than four different ways of achieving automatic layout of Fudgets.

Default layout This is what you get if you don't specify any layout. Using the normal Fudget combinators, Fudgets are placed next to each other if they are connected by combinators.

Layout combinators To specify Fudgets side-by-side or one-above-another, *layout combinators* are what is required. By using the combinators, `>#+<` and `>#==<`, we can specify how Fudgets are placed next to each other. For example:

```
(f1, Above) >#==< f2
```

will result in `f1` being placed above `f2`. An important restriction: a Fudget must be *connected* to another Fudget to be *placed* next to it.

Named layout For a more flexible layout scheme, it is possible to *name* Fudgets with a label, then specify the layout at a higher level in the hierarchy of composition by referring to these labels. For example:

```
prog = nameLayoutF layout $ one >==< (two >+< three)
    where one = nameF oneN f1
          two = nameF twoN f2
          three = nameF threeN f3
          layout = listNL verticalP (map leafNL [oneN,threeN,twoN])
          (oneN:twoN:threeN:_) = map show [1..]
```

but we now have the overhead of labels.

Use a placer Fudget When wrapped around a Fudget, a *placer* applies a layout function to the contents of the Fudget (ie. the Fudgets composed to form the Fudget). For example:

```
prog = placerF horizontalP (f1 >==< (f2 >+< f3))
```

places `f1`, `f2` and `f3` in a line horizontally. The Fudgets in the composition are taken in left to right order.

An Example Fudget Application

We've looked at a fair number of the tools available to the programmer. We will now use these in an example. We have not said much about the library provided with the system. As the system is currently still under development, the final set of library Fudgets is not complete, but it is already large. We will introduce some of these in the course of this example.

The application program we will build will be a simple up/down counter. Two buttons (marked *up* and *down*) increase or decrease the number displayed in a box

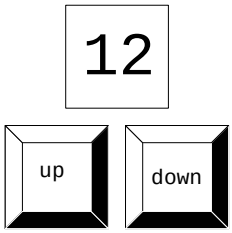


Figure 2.14: An Example Application — Counter

beside them. Fig. 2.14 shows a picture of how we would like the screen to depict this application.

The buttons and the box that displays the number are defined using Fudgets provided in the library (called `buttonF` and `intDispF` respectively). A `button` Fudget emits a `Click` message whenever it is *pressed* with the mouse pointer. The integer shown in the display is replaced by any new integer sent to it in a message. We would like to take advantage of the fact that we can pass any first class value in a message, and have the buttons send an *integer modifying* function to the display — eg. the *up* button will send a function that adds one to an integer. This will allow us to add further buttons later (eg. a *reset* button would send the function `const 0`) without having to change the displayer. To create this structure of program, we must create a new button Fudget (based on `buttonF`) that allows us to specify the message it sends when clicked. We must also create a new integer displayer, that accepts integer modifying functions instead of plain integers.

A diagram of the Fudgets is shown in Fig. 2.16 and the program in Fig. 2.15. Our new button Fudget, `buttonMsgF`, is defined simply as a `buttonF` with a message-mapping function attached to its output using the `>^=<` combinator. We have generalised the displayer side of the problem slightly by introducing a *state holder* process that is given an initial state of any type, not necessarily an integer, and accepts functions that modify the state. Each time the state is modified it outputs the new state. We connect this process to the input of the usual integer displayer (in `intHolderF`) resulting in the displayer we require.

The buttons are combined using the `>*<` combinator that merges their output messages into a single untagged stream. This is the binary operator equivalent of the `untaggedListF` combinator. (To add more buttons we can simply change to the `untaggedListF` combinator and put as many buttons as we require in a list.)

In §2.4.1 we look at a larger application and assess the strengths and weaknesses

```
import Fudgets

main = fudlogue (shellF "Up/Down Counter" updown)

updown = intHolderF >==< buttonsF

buttonsF = buttonUpF >*< buttonDownF
    where buttonUpF = buttonMsgF (+1) "up"
          buttonDownF = buttonMsgF (+(-1)) "down"

intHolderF :: F (Int -> Int) a
intHolderF = intDispF >=^^< stateHolderSP 0

stateHolderSP :: a -> SP (a -> a) a
stateHolderSP s = mapstateSP hold s
    where hold s f = let s' = f s in (s',[s'])

buttonMsgF :: m -> String -> F Click m
buttonMsgF m s = tomsg >^=< buttonF s
    where tomsg Click = m
```

Figure 2.15: The Example Fudget Application

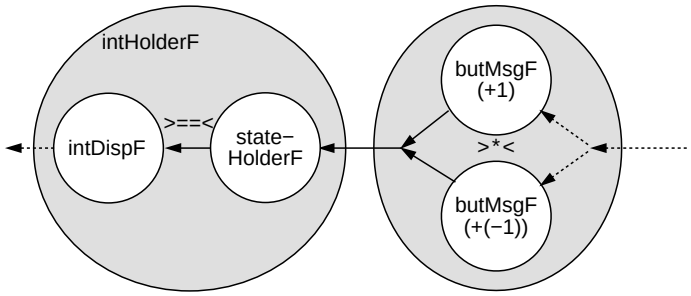


Figure 2.16: The Fudgets of the Example

of the Fudgets system.

2.3.2 The Concurrent Clean System

Concurrent Clean is a lazy functional programming language based on term graph rewriting [PvE93]. This review is based on version 0.8.3 of the Clean system [vEHN+93]. Since this review was undertaken a major new release of the system has been made available [PvE95]. We have noted some of the differences in this review, however as the new version had only just been made available at the time

of writing, we have not been able to test any of the new features. Function definitions are actually graph rewriting rules. For example the definition of the function representing the equation $Y = (x^2 + 1)x^2$ might be:

```
::      Y INT -> INT;
        Y x -> * (+ z 1) z,
                z: * x x;
```

The first line beginning :: indicates the type of the function. There are no infix operators — all functions use prefix notation. Identifier `z` is a NodeId, referring to the graph node `* x x` (that is, x^2). It is used in this instance to declare that the calculation `* x x` should only be done once, and the result shared.

NodeId definitions are not full counterparts of Haskell `where` clauses — a NodeId cannot be an application or a constructor pattern, but only a simple variable.

Strictness Annotations Strictness annotations may be used in a function definition to indicate that a function is strict in a certain argument (ie. that the argument should be evaluated before the function rewrite rule is applied). This enables more efficient programs to be written. A strictness annotation consists of the character '!' which may appear in the type definition of a function (where it affects the reduction order for all applications of that function), or in the rewrite rule (where it affects a particular function application).

Process Annotations Process annotations may be added to specify the parallel reduction of graphs. This is fine-grain parallelism. This form of parallel reduction is currently not supported on the Macintosh and Sun implementations, so is irrelevant to GUI applications. In the latest release, a program can be split into *concurrent* processes, but this is *conceptual* concurrency, where processing time on a single CPU is time-shared between processes.

I/O in Concurrent Clean

The Concurrent Clean answer to the problems of I/O in a pure functional language is to allow certain values, including the one that represents the state of the world, to be updated in-place. This is achieved through *uniqueness typing* — some types are marked as unique and may only be used in a single thread (ie. values of this type may not be duplicated). This is similar to the technique used in monadic I/O

where the world value is an abstract type and can only be accessed by the monadic combinators and primitive I/O functions, that guarantee to use it single threadedly. The difference is that while the monad World is protected by restricting access to it, the unique World has an annotated type and its uniqueness is checked by the compiler. If the World is duplicated an error is flagged by the compiler. This method has the advantage that the uniqueness annotation can be used for other types. For example, a large data structure might be made a unique type to increase efficiency of memory usage — only one copy of the large data structure is kept at any one time.

Unique Types A type with the 'UNQ' annotation, for example the unique state type:

```
TYPE
::      UNQ State -> INT;
```

is claimed by the programmer, and checked by the compiler, to be used only in a single thread (ie. accessible from the root of the graph by one path only. For example, in the following function definition, the programmer has broken the rule:

```
::      BadFunction INT -> (State, INT);
        BadFunction i -> (+ i 1, * i 2);
```

The function claims to return a unique type (as the first part of the tuple), but this is calculated using a subgraph (i) that is shared.

Using UNQ for I/O The Unique type annotation is used to declare a WORLD that is a structured type containing *environments* that represent the state of the I/O system. Library functions that alter these environments have an immediate side-effect (for example, one might output a character). But this happens safely because, being a unique type, the WORLD (and its environments) is guaranteed to have been used in a single thread only and therefore accurately represents the state of the I/O system at all times.

Without the unique guarantee, it would be possible for anomalies to occur such as two different views of the WORLD being held in separate parts of the program. Consider the example:

```
::      TwoWorlds FILE -> (FILE, FILE);
        TwoWorlds f -> (FWriteC 'a' f, FWriteC 'b' f);
```

where `FILE` is a part of the `WORLD` type which represents a particular stored file, and `FWriteC` is a function which has the side-effect of writing a character to a file. The function returns references to two files, but which one is correct? Regardless of the order in which the two parts of the returned tuple are evaluated, both file references are wrong because one expects to refer to a file containing the old contents with an '`a`' added, and the other expects the old contents with '`b`' added. In reality, the file referred to will contain the old contents with either '`ab`' or '`ba`' added, depending on what order the parts of the tuple were evaluated.

The `WORLD` type is passed to and from any function that is required to do some I/O, and the order of interaction is defined by the dependency on world values passed between functions. Non-I/O functions are not passed the `WORLD` type and so are not constrained to a particular order of evaluation, maintaining the benefits of laziness.

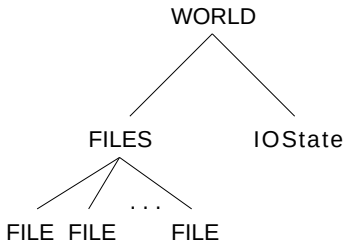


Figure 2.17: The Structure of the WORLD Type

Fig. 2.17 shows the structure of the `WORLD` type. The `FILES` environment provides access to the files available to a program. The library provides functions that work with the `FILES` environment to enable files to be opened, closed, read and written in a variety of ways. For example, a program to put the text "`Hello World!`" in a file named "greeting":

```
MODULE test;

IMPORT deltaFile;

RULE
::      Start UNQ WORLD -> UNQ FILE;
        Start world -> gfile1,
                gfile1: FWriteS "Hello World!" gfile,
                (gfile, files1): FOpen "greeting" FWriteText files,
                (files, world'): OpenFiles world;
```

This operates as follows: `OpenFiles` is applied to the world environment passed to the program, and yields a files environment and a new world in which it is recorded that the files are open. The files environment is used by the `FOpen` function to open a file called "greeting" for writing, returning a file environment and a new files environment in which the open status of the file is recorded. Lastly, the `FWriteS` function takes the file and returns the file containing the text "Hello World!", and this is the return value of the program.

Event I/O

Another part of the `WORLD` type is the I/O state. This contains references to graphical devices such as windows, menus, etc., and a stream of events (see Fig. 2.18).

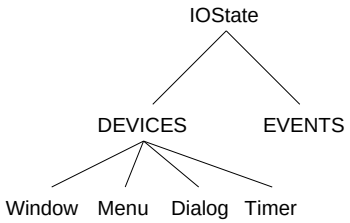


Figure 2.18: The Structure of the IOState Type

Event I/O is concerned with manipulating these devices rather than files. Events occur whenever a mouse button is clicked, or a key is pressed, and the program defines the response to these events. The response to any event could be defined by one large event-handling function mapping an event and some program state to a response and a new program state, but this function would quickly become over complex with increasing program functionality, because every possible state and event combination must be considered.

The Concurrent Clean I/O system breaks this problem into parts by having an event-handling function for each individual screen object. Events are automatically routed to the correct function, along with the program state and I/O state, and the function returns a new program state and I/O state. If the function alters the I/O state, using the library functions to do so, this has the effect of communicating some response to the devices.

The particular set and structure of devices used in an application program is defined by a complex hierarchical set of abstract data types. Event-handling functions are *registered* with the system by placing them within these data types. The

data types also contain other options as parameters (for example, the title text of a window). Each device has a unique identifying number (decided by the programmer) that can be used to alter the parameters or event-handler associated with that device. This allows, for instance, a menu selection to alter the behaviour of an icon button, using a library function. Devices come in four groups: menus, windows, dialogs and timers. We will look briefly at each of these. For more information, see [PvE95].

A **menu** presents a list of options to the user (possibly as a hierarchy rather than a list). Each option may be enabled or disabled and has a function which is called when the option is selected.

A **window** has various attributes (for example its title and whether it can be resized) and can be drawn in by the application using primitive drawing functions such as one that draws a line. The application must provide a function that is called by the system when an event occurs that requires the contents of the window to be redisplayed.

A **dialog** is a special type of window used to obtain some information from the user. It may contain buttons, text entry boxes and user-defined controls, and its sole purpose is to enable the user to set parameters or options of a program. Most dialog windows will have a button that is pressed to indicate that the user has finished the selection. The objects in a dialog window may be placed explicitly, or placed relative to other dialog items. There are a number of predefined dialogs. The *file selector* dialog enables the user to choose a file from a scrolling list of the files in a directory. The *about* dialog is displayed when the apple symbol on a Macintosh is clicked, or in the main application window on a Sun computer, and contains a short description of what the application program is *about* (hence the name). A *notice* dialog is used to warn the user about something (for example, if *Quit* has been selected but an altered file has not been saved).

A **timer** may be used to perform an action each time a certain interval of time has passed, such as updating the display of a clock every minute.

An Example Program

To illustrate the style of an application program in Concurrent Clean we present a small example (a screen picture is shown in Fig. 2.19). This application consists of two menus, `Exit` and `Shape`, and a window. The window displays either a circle

or a square — the user may choose which using the `Shape` menu. The `File` menu contains one choice, `Quit` which exits the application.

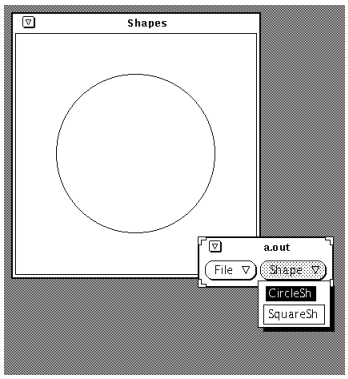


Figure 2.19: The Example Application

This example is different from the one chosen to demonstrate the Fudgets system because each system has certain operations that are easy to implement and others that are more difficult. Using the same example would be an unfair comparison. Later we use a larger common application to compare the style of programming in each system.

The top level of the program defines the global program state and the device components of the application:

```
MODULE excitingapp;

IMPORT deltaEventIO, deltaIOSystem, deltaWindow, deltaPicture;

TYPE
:: UNQ State -> SquareSh | CircleSh;
:: UNQ IO -> IOState State;

RULE
:: Start UNQ WORLD -> UNQ WORLD;
   Start w -> CloseEvents e' w',
      (s, e'): StartIO [MenuSystem [file, shape],
                        WindowSystem [disp]] initShape [] e,
      (e, w'): OpenEvents w,
      initShape: CircleSh,
```

`OpenEvents` extracts the list of events from the `WORLD` type and passes them to `StartIO` which initialises the devices that we require for our application and then ensures that all events are directed to the correct devices.

Next we define the menus:

```
file: PullDownMenu FileId "File" Able [
 MenuItem QuitId "Quit" NoKey Able Quit],
shape: PullDownMenu ShapeId "Shape" Able [
 MenuRadioItems CircleShId [
  MenuRadioItem CircleShId "CircleSh" NoKey Able (SetShape CircleShId),
  MenuRadioItem SquareShId "SquareSh" NoKey Able (SetShape SquareShId)]],
```

The *Shape* menu is made up of `MenuRadioItems` of which only one may be selected at a time. The `CircleId` parameter to `MenuRadioItems` indicates that the circle option will be the initial selection. We will come back to the event-handling function `SetShape` later.

The window definition follows:

```
   disp: FixedWindow WinId WinPos "Shapes" WinDom UpdateWin [];
```

A `FixedWindow` indicates a window of fixed size and with no scroll-bars. The `pos` is the initial position of the top-left corner of the window on the screen, and the `dom` is the rectangle defining the size of the display area of the window. `UpdateWin` is the function that will be called when the window's contents need displaying on the screen:

```
:: UpdateWin UpdateArea State -> (State, [DrawFunction]);
   UpdateWin ua cur -> StateAlt cur circle square,
      circle: [DrawCircle ((150,150),100)],
      square: [DrawRectangle ((50,50),(250,250))];

:: StateAlt State [DrawFunction] [DrawFunction] -> (State, [DrawFunction]);
   StateAlt CircleSh c s -> (CircleSh, c);
   StateAlt SquareSh c s -> (SquareSh, s);
```

The `SetShape` function is attached to the shape menu items. Because the state must remain unique, `SetShape` takes a shape id number instead of a new state as an argument, which is converted to a state by the `ShId2Sh` function:

```
:: SetShape INT State IO -> (State, IO);
   SetShape set s io -> DrawInWindowFrame WinId UpdateWin (ShId2Sh set) io2,
      io2: DrawInWindow WinId [EraseRectangle ((0,0),(300,300))] io;

:: ShId2Sh INT -> State;
   ShId2Sh CircleShId -> CircleSh;
   ShId2Sh SquareShId -> SquareSh;
```

that first erases the old contents of the window, then uses the window update function to redraw the window with the new shape.

Finally, the quit function and Id macro definitions:

```
:: Quit State (IOState State) -> (State, IOState State);
   Quit s io -> (s, QuitIO io);

MACRO
   FileId -> 1;    ShapeId -> 2;     WinId -> 1;
    QuitId -> 11;   CircleShId -> 21;
                    SquareShId -> 22;

   WinPos -> (100,100);
   WinDom -> ((0,0),(300,300));
```

Macro applications are reduced at compile time. Macros are useful for values that can be computed once during compilation and do not take up much memory in their reduced form.

2.4 Case Study

In this section we look at a larger application and assess the strengths and weaknesses of the Fudgets and Concurrent Clean systems.

2.4.1 Developing a Fudget Application

Using the Fudgets system, development of an application typically has three stages. The programmer knows what is required of the program, so first decides what should appear on the screen. Fig. 2.20 shows blocks for each of the highest hierarchical level of screen objects of an example application (ie. each of these objects may be composed of further screen objects). In the program these objects will be implemented as complex Fudgets. The application is a program similar to the Escher tile pattern program originally written by Foubister [FR91], and is described in more detail in the next section. The tile program was originally written as an exercise in writing interactive programs in a functional language. As such it seemed a good choice for evaluating the ease with which interactive applications may be developed using the Fudgets system. The arrows indicate where the objects communicate.

The second stage of the development is to decide how these objects will be combined as Fudgets. This is not straightforward. Fudgets only have one high level

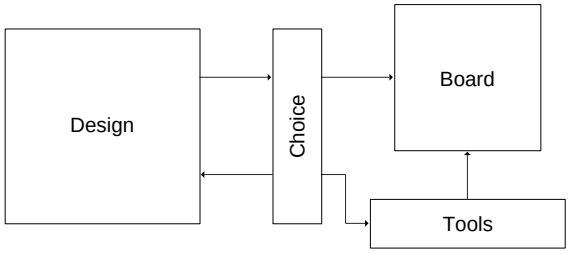


Figure 2.20: Connections Between Screen Objects in an Application

message path in and out, and these objects require many message paths (eg. `Choice` has three out and one in). We must combine the Fudgets in such a way that multiple message paths from or to an object become one message path between Fudgets, that conveys messages tagged to indicate which of the original paths they were intended for. The dotted lines in Fig. 2.21 show how the Fudgets might be combined. The key indicates which combinators are used. The `Id` Fudget (identity) simply passes through any messages it receives. This Fudget is used to enable the `Choice` Fudget to communicate messages to Fudgets it is not directly connected to. Choice would send message `md` to `Design` by outputting `Left md`, message `mb` to `Board` by outputting `Right (Left mb)`, and message `mt` to `Tools` by outputting `Right (Right mt)`. This is not the *only* way the Fudgets could be connected together, but there is no obvious simpler alternative.

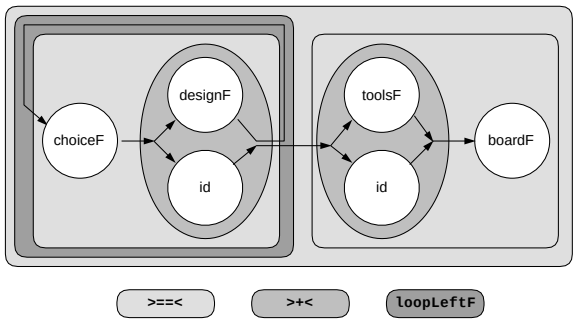


Figure 2.21: Combining the Fudgets

The combination of the `Design` and `Tools` Fudgets with the `id` Fudget may be factored out with the function:

```
throRight :: F a b -> F (Either a c) (Either b c)
throRight f = f >+< thro
    where thro = mapF id
```

The final stage of the development is to implement the Fudgets. It is not possible to implement them completely before the second stage because the connections must have been established for the programmer to know, for example, how `Choice` will address the three Fudgets to which it is connected for sending messages. If the programmer later decides that, for instance, another message stream is required between `Tools` and `Design`, then the connection process must be started again from the beginning. It would be better if message connections could be changed more easily.

Figure 2.22 shows a picture of the application window. The Choice and Tools buttons may be *pressed in* (indicated by the shaded surround) and each set form a *radio group* (ie. only one may be pressed at a time, pressing another button causes the first to *pop out*). Selecting a tile from one of the four tiles in Choice causes that tile to appear in the Design area, and eight rotations/reflections of the tile to appear in the Tools section. Clicking on two points in the Design area with the left mouse button causes a line to be added to the design. A similar action with the right button removes a line from the design. Clicking on the button under the Design updates the current tile and the eight rotations/reflections with the new design. What happens when the mouse is clicked in the Board area depends on which of the Tools is currently selected. If one orientation of the current tile is selected, then the tile is placed onto the board in the selected orientation, at the position where the mouse is clicked (there is an invisible grid on the board onto which tiles are placed). If a rotation or reflection tool is selected, clicking a tile on the board causes it to rotate or be reflected. The Board remembers which rotation/reflection is placed in each position, and if the button beneath the board is clicked, the tiles on the board will change to whatever tile design is currently displayed in the tools section, keeping their orientation/reflection.

Some operations (for instance, clicking the update button beneath Board) take longer to complete their action than others. These are generally the ones that have most effect on the screen.

Many of the main Fudgets of this application were implemented using two sub-Fudgets in a master and slave fashion. The master Fudget, an abstract Fudget, controls the slave Fudget and receives and sends messages to and from the rest of the program. The slave Fudget, representing a display object of some kind is connected only to the master controlling Fudget. With this arrangement, the function of a basic library Fudget, such as an integer displaying Fudget, may be supplemented by

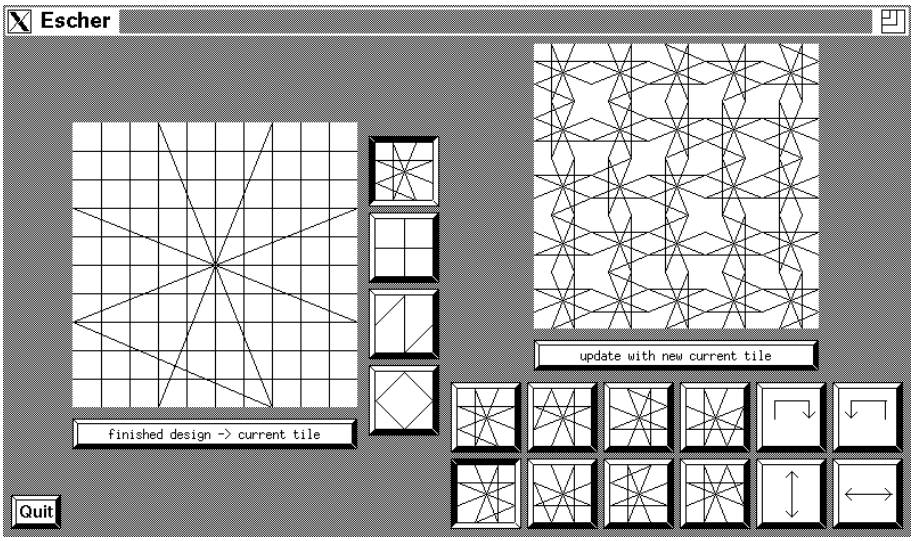


Figure 2.22: The Fudget Escher Tile Program

an abstract Fudget. For example, suppose we have an `intButtonF` Fudget that is a button that displays any integer sent to it and outputs a `Click` when pressed. We can make it into a button that emits a number starting at zero, increasing by one each time it is clicked, and displaying the number it will emit next click. Fig. 2.23 and Fig. 2.24 show the Fudget connections that are required and how this might be achieved.

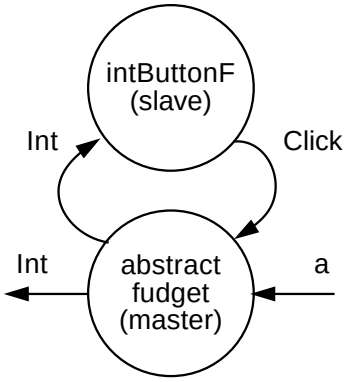


Figure 2.23: Supplemented Fudget Connections

A functional definition of this Fudget is given in Fig. 2.25. The `master` is defined as a *stream processor*, the general process of which Fudgets are made. A stream processor has one input stream and one output stream. Messages are input using the `getSP` function and output using `putSP`. A Fudget is constructed from a stream

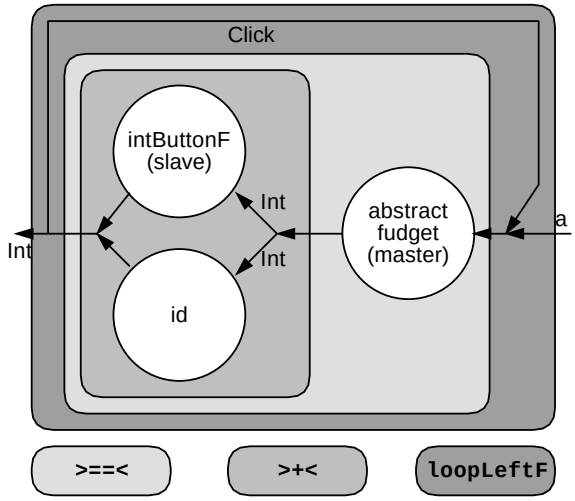


Figure 2.24: Supplemented Fudget's Components

processor by tagging messages with `High` or `Low` for the high and low level streams, though the programmer doesn't normally see these.

```
incButtonF :: F Click Int
incButtonF = loopLeftF ((throRight slaveF) >=^^< masterSP)

masterSP :: SP (Either Click Click) (Either Int Int)
masterSP =
    let initCount = 0 in
    putSP (Left initCount) $
    master initCount
        where
        master c = getSP $ \m -> case m of
            Left Click ->
                let c' = c + 1 in
                putSP (Right c) $
                putSP (Left c') $
                master c'

slaveF :: F Int Click
slaveF = intButtonF
```

Figure 2.25: Definition of the Supplemented Fudget

Advantages of Using a Functional Language

Unless at least some of the advantages of functional programming are available to the Fudget programmer they might as well use an imperative system. During the development of the Escher tile pattern example, one of the functional features found to be most useful was the ability to pass functions in messages, just as any other data value. A Fudget, `f1`, which is required to manipulate the *state* maintained in another Fudget, `f2`, has no need to receive that state in a message from `f2`, or maintain a copy of the state itself, in order to be able to change it. Instead, `f1` can send a state-manipulating function to `f2`, which applies the function to its state.

The alternative is much more complicated: `f1` sends a message to `f2` indicating that it needs a copy of the state in order to be able to change it; `f2` then sends its state to `f1` in a message, whereupon `f1` manipulates this state, and finally sends it back to `f2`. This takes three messages in total, compared with one message for the function-passing version, and additionally requires that `f2` has a message connection to `f1`.

The usual features of functional languages are available to aid the programmer. For example, `map` and `zip` are used to generate a list of messages that replace a grid of tiles with a new set of tiles. The imperative equivalent would have been much more complicated.

Fudget Type

The Fudget type `F a b`, with the input type on the left and output type on the right, might seem confusing as Fudget diagrams are usually drawn with input on the right and output on the left. Simply drawing the diagrams round the other way does not work because the Fudget combinators also assume the convention of input on the right, output on the left. This convention was chosen to match with that of function application (eg. in `f.g` the result from application of `g` is applied to `f`, as in `f >==< g` the messages from `g` are sent to `f`).

Testing Fudgets

During the development of an application it is useful to test Fudgets by *plugging* them into a *test-rig* to see that they operate correctly. With some Fudgets it is relatively easy to construct the required test-rig. For example, suppose a Fudget receives words and emits integers indicating the number of letters in each word.

This Fudget might be tested by connecting a menu containing five words (simple to construct using a library Fudget) to its input, and connecting its output to an integer displaying Fudget (again, from the library). The test data may then be fired at the Fudget by selecting words from the menu, and the display Fudget checked to see that it displays the correct number.

A Fudget that emits messages containing functions would be more difficult to test however. How would a function be *displayed* for checking?

A debugging *wrapper* can be useful. When placed around a Fudget, the wrapper displays messages entering and leaving the Fudget in a border placed around the Fudget on screen. Again, we cannot display some types of messages. At present a wrapper is available that displays the low level messages only, in the window from which the Fudget application was executed.

Classes, Hierarchy and Inheritance

Many GUI toolkits have object-oriented properties allowing screen objects to be grouped into classes, and a class to inherit properties of other classes, saving the programmer from duplicating work when objects have common functionality. Whilst developing the Escher tile program a Fudget was required that displayed line drawings. If it received a new drawing as a message then this drawing replaced the old one. A variation of this Fudget performs the same function, but also outputs a message containing a button number if the mouse is clicked on the drawing. Another variation outputs a message containing the displayed drawing when the mouse is clicked on it. Each of these variations was implemented as a separate Fudget. A better solution would have been to have one generic Fudget with a parameter that specifies which of its many possible functions to perform.

Problems Naming Message Streams Using Abstraction

One way to improve the way message streams are addressed from within a Fudget is to use a `let` declaration to name the streams. For instance, taking the example of the `Choice` Fudget described in the previous section, `Choice` must address the `Design`, `Tools`, and `Board` Fudgets by sending a message of type `Left m`, `Right (Left m))`, or `Right (Right m)`. Using a `let` declaration, the function implementing the `Choice` Fudget becomes clearer (see Fig. 2.26).

Naming the input message streams, however, is not as simple. This is because

```
choiceControlF =
 let startstate = 0
     step c m =
      let toButtons = Left
          toDesign = Right.Left
          toTools = Right.Right.Left
          toBoard = Right.Right.Right
      in case m of
       (Left (n, t)) -> -- from Buttons
        (n, [toDesign t, toTools (setT t), toBoard (setT t)])
       (Right t) -> -- from Design
        (c, [toButtons (c, t), toTools (setT t), toBoard (setT t)])
 in absF (mapstateSP step startstate)
```

Figure 2.26: Naming the Message Streams Using `let`

Fudgets generally have a case expression on the next input message to be processed, and the patterns of the case may not contain function names other than constructors. The definition of Fig. 2.27, for example, is not allowed.

```
boardF =
    let startstate = 0
        step c m =
            let fromTools = Left
                fromChoice = Right
            in case m of
                fromTools t -> ...
                fromChoice c -> ...
```

Figure 2.27: Illegal Pattern Matching in a Case Expression

2.4.2 The Concurrent Clean Escher Tile Program

Fig. 2.28 shows a Clean Escher tile program in action. From the user's point of view the program is very similar to the Fudgets version. The layout of screen objects is less of an issue in the Clean version because separate windows are used for each of the design, board, choice and tools areas. There is no support for automatically combining these areas into a single window, specifying layout, as there is in the Fudgets system.

A Concurrent Clean application typically consists of a set of *state transition*

functions, and functions to support these. The transition functions are *attached* to events (such as the pressing of a particular button), and are called when the event occurs. The information passed to, and returned from, such a function varies according to the type of event to which it is being attached. For instance, the event caused by a window exposure supplies its function with an update area (the part of the window that has been exposed) and the program state, and expects in return a new program state and a list of drawing commands to update the screen. For example:

```
::      UpdateDesignWin UpdateArea State -> (State, [DrawFunction]);
        UpdateDesignWin ua s -> (ns, pic),
                pic: Join [SetPenSize (1,1) | Grid]
                          [SetPenSize (5,5) | MagTile DTRatio dt],
                (dt, ns): GetCurrentTile s;
```

is the expose event function for the design window. `GetCurrentTile` is used to extract the current tile design from the program state (returning the design and a new, yet unaltered, program state). The drawing commands returned consist of a grid and the tile design, magnified to the required size. Commands to change the size of pen (so that the grid is drawn in thin lines and the tile design in thick lines) are interleaved in the required places between the commands.

Whilst writing the program it was found to be difficult to work on a small part of the program in isolation. This was mainly because of the global state, but also because some actions performed in one part of the program require an effect in another window. For example, clicking an icon in a window may require a drawing to rotate in another window. When writing the functions for the icon, it is necessary to know which other functions to use in order to effect the rotation. (In the latest version of Clean it would be possible to hide this using a suitable message-passing abstraction.)

Efficiency

Being able to define the graph of a function explicitly — indicating exactly which sub-graphs are shared — enables a much finer control on the graph structure of a program. In addition, the strictness annotations in a function allow the programmer to produce a more efficient program. In some cases the strictness analysis phase of compilation can compute the strictness required.

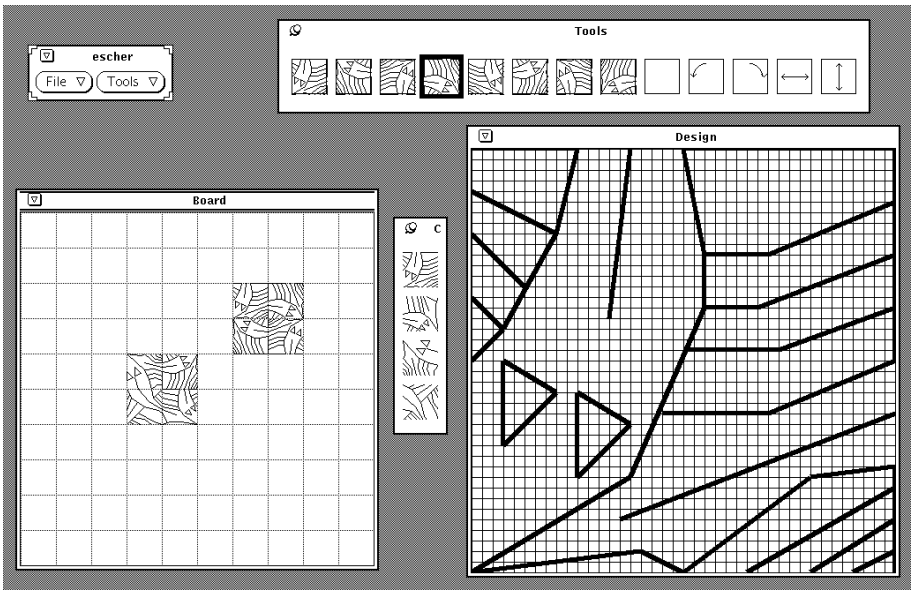


Figure 2.28: The Escher Tile Program

New Sweeter Version

At the time this review was first undertaken, the Concurrent Clean language lacked much of the syntactic sugar common to lazy functional languages. There were no list comprehensions, no infix operators, no `let` or `where` clauses and no class system or overloading.

Version 1.0 of Concurrent Clean now includes all these things and more. Among the additions are records, arrays, existentially quantified types and more refined control of strictness and uniqueness typing. It is also possible to define concurrent processes that may communicate by message passing, or via files or shared data.

The addition of records should improve access to the global state, as each separate part can now be named, and new parts added easily. As the Escher Tile program was written for version 0.8 however, these facilities were not used.

Callbacks vs. the Event-Polling Loop

One style of programming GUIs has an event-polling loop in which the program requests the next event from the OS and services this event before going back for the next. Another style is to register event-handling functions with the OS that are called on certain events. In systems that use an event-polling loop, all possible events must be catered for at all polling points (though in a functional language,

common event responses may be factored away using a suitable abstraction).

In Clean, the functions given as event-handlers for each device are rather like the *callback* functions used when writing C applications for The X Windows System. They have the advantage that, once *registered* with the system, they can be mostly forgotten about. The disadvantage is that the event-handler for a screen object only has access to any information the system passes it when the event occurs, and not information about other related objects. A global program state and I/O state are passed to most event-handlers along with details of the event that has occurred. The state associated with any part of the program is accessible by the event-handler, which gives good flexibility but poor localisation[1].

It is best to define a set of *access functions* on the program state, rather than use pattern matching in each function that uses a part of the state. Otherwise if the state needs to be extended, all the functions which use it will have to be changed, rather than just the access functions. This is a standard data type abstraction. In the latest version of Concurrent Clean the situation is improved with record types: these can be used to partition the global state neatly.

Restrictions on the use of Dialog Items

In version 0.8 dialog windows were separate from normal windows in the definition of an interface, and had a restricted behaviour. The abstract datatype defining a dialog contains a list of dialog items. Dialog items are screen objects such as buttons or controls of some kind. They are *leaf-nodes* in the description of an interface — they cannot be composed to form more complex dialog items.

As the event-handlers for dialogue items are only passed the dialogue state, we found we could not make a dialogue button have an effect in another window. In version 1.0 dialog windows have been integrated with normal windows and notices. The callback functions associated with any type of control can access the IOState, so the problems with version 0.8 appear to have been alleviated.

Lists of Id's

Every screen object is referred to by an Id. number. These Id.s are usually defined as macros to give them a meaningful name, for example:

[1] In the latest version of Concurrent Clean the state passed to an event-handler is split into global and local parts. The local part is private to the particular object

```
MACRO
        DrawingWinId -> 0;
        PatternWinId -> 1;
        OnIconId -> 0;
        OffIconId -> 1;
        SlowIconId -> 2;
```

It is a nuisance to have to maintain these lists of Id. numbers. It would be better if there were some other way of referring to objects, such as by their relationship to the object owning the current event (for example, if an icon is clicked its event handler could *close the window in which this icon is contained*).

Window Objects — Widgets

The window device allows the contents of a window to be drawn by using primitive drawing commands, such as line drawing. There is no support for defining any sort of *widgets* — predefined screen objects with a certain behaviour. Obviously the programmer could use a suitable abstract function to define a screen object and its behaviour in response to events, but shouldn't this be part of the standard library, to save the programmer time and so that all applications have a similar *feel* to them?

Generic Functions that Access the State

Suppose one defines a general function to implement a group of radio buttons, each needing to store some information in the program state. It would be necessary when declaring one *instance* of this radio group to pass in state access functions which access the state for *this particular* group. Otherwise this instance of the radio group would have to access a fixed part of the global state and it would not be possible to have multiple instances of the group used in the application program. In version 1.0 each window has its own private state, and this information can be stored there.

2.5 Some More Recent Functional GUI Systems

Since the comparison of Fudgets and Clean was undertaken more systems have become available. The *Haggis* system has many similarities to the system we describe in later chapters. *Tk-Gofer* offers a functional interface to an imperative GUI toolkit language. The *BriX* system provides a functional interface to The X Windows System whilst avoiding non-determinism. We look at each in turn.

2.5.1 Haggis

The Haggis System [FJ95] implements GUIs in an extended version of the Glasgow Haskell Compiler [PJGF96] that supports Concurrent Processes. Communication and synchronisation between processes is achieved through the use of *atomically-mutable state* that provides finite buffered channels of length one. Monads impose a sequence on evaluation and allow mutable state to be accessed safely. A value of the primitive type `MVar a` refers to a mutable store that is either empty or contains a value of type `a`. Primitive operations create a new `MVar`, take the value from an `MVar` (blocking the process until a value is placed there, if it was empty), or place a value into an `MVar`. Two processes can communicate if one reads from an `MVar` (blocking if no value is there) and another writes to it. It is an error for the sender to write to the `MVar` when it is not empty. A handshaking protocol must be built: a second line of communication is used in the other direction to acknowledge receipt of each message and indicate that the forward-direction `MVar` is empty. Other communication schemes can be built such as synchronous or asynchronous message passing, with finite and infinite buffered channels, process synchronisation, semaphores, etc..

A Haggis version of the Up-Down Counter is shown in Fig. 2.29. This counter is similar in construction to the Fudget version (see Page 27). Once again we take the basic *button* and generate one that emits a specified message when clicked instead of the default `()`. Lines of communication between button and display are implicit in the Fudget version, but named in the Haggis version. In the Fudget version information is communicated from buttons to an integer-holding display because they are combined using a combinator:

```
updown = intHolderF >==< buttonsF
```

In the Haggis version, we see this same line of communication named in a lambda expression as the buttons handle, `btns`:

```
buttons >>= \(btnsd,btns) ->
intDisp btns >>= \dispd ->
```

The advantage of naming lines of communication is that we can add more lines easily. Effectively we make a new combinator on the spot for every particular need. An advantage of the Fudgets version is that the values are passed *raw* — they are not wrapped up in any algebraic type. The display could be connected to any Fudget that emits integer-modifying functions. Similarly, the button could be connected

```
import Concurrent
import Haggis

main =
    buttons >>= \(btnsd,btns) ->
    intHolder btns >>= \dispd ->
    vBox [btnsd,dispd] >>= \maind ->
    wopen [] maind >>
    return ()

buttons :: IO (DisplayHandle, Button (Int -> Int))
buttons =
    buttonMsg (+1) "up" >>= \(buttonUpd,buttonUp) ->
    buttonMsg (+(-1)) "down" >>= \(buttonDownd,buttonDown) ->
    hBox [buttonUpd,buttonDownd] >>= \buttonsd ->
    combineButtons [buttonUp,buttonDown] >>= \buttons ->
    return (buttonsd,buttons)

intHolder :: Button (Int -> Int) -> IO DisplayHandle
intHolder but =
    mkLabel "0" >>= \(labeldisp,label) ->
    forkIO (dispUpdate 0 label) >>
    return labeldisp
        where
        dispUpdate :: Int -> Label -> IO ()
        dispUpdate n lab =
            getButtonValue but >>= \f ->
            let n' = f n in
            setLabel lab (show n') >>
            dispUpdate n' lab

buttonMsg :: m -> String -> IO (DisplayHandle,Button m)
buttonMsg m s =
    mkButton (text s) >>= \(buttond,button) ->
    let mbutton = mapButton (\_ -> m) (\_ -> ()) button in
    return (buttond,mbutton)
```

Figure 2.29: The Up-Down Counter in Haggis.

to any Fudget that accepts integer-modifying functions. In the Haggis version, the display expects a *button-handle* through which it receives an integer-modifying function. The display cannot be connected directly to another type of interface component or process.

In Haggis, processes can receive from more than one other process in a non-

deterministic way. A small *sucking* process is attached to each `MVar` (or communication protocol built on top of it) and blocks, waiting for any values passed through it. Values received by any of the sucking processes are fed into a single `MVar` from which the receiving process picks them up (see Fig. 2.30). Non-deterministic merging of values from all the sources is achieved because the sucking processes are all writing to a single `MVar`. In the counter program the sucking processes are set up by the `combineButtons` function.

If the `MVar` to which a process writes were not passed to it, but created by it and returned, then the sucking processes would not be necessary. The left-most `MVar` in the diagram could be passed to each source process.

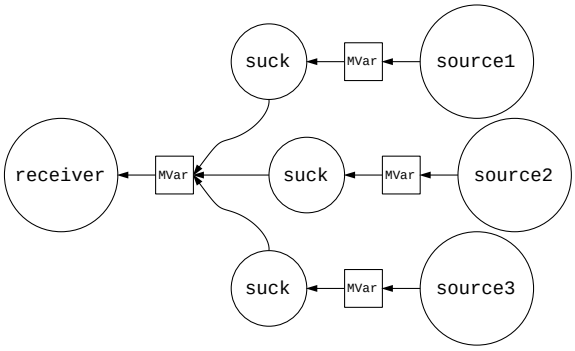


Figure 2.30: Sucking Processes Merge Multiple Sources of Input.

Widget functions return separate handles for *physical representation* and *behaviour*[2]. Handles for physical representation can be combined using layout combinators such as `vBox` and `hBox`. These return new physical representation handles for the combined layout.

Behaviour handles are used to communicate with the Widget. For example, using the behaviour handle returned from a *label* widget, `lab`, we can set the string contained in the label to `"text"` with `setLabel lab "text"`.

The picture of an interface part is described as a composition of *Glyphs*, small picture elements such as lines and circles, composed by placing them side-by-side or one above another. The stretchiness and squashiness of individual elements is controllable, allowing pictures to adjust to fit the available space. Interface parts are

[2]This is true of the most recent version of Haggis, though the earlier version described in [FJ95] combines the two handles into one. They have since been separated as it seems more natural to treat them independently

combined using the same graphical combinations to form a Widget and the Widget is displayed by applying the `wopen` function to it, which opens an X Window containing the Widget.

2.5.2 Tk-Gofer

The Tk-Gofer System [VTS95] consists of Widgets implemented in an imperative language (Tk/Tcl) but accessed through a functional interface from the functional language Gofer. In Sinclair's system [Sin92] the whole interface is defined in the imperative language leaving the higher-level details, such as what information will appear in windows, to the functional program. The functional program communicates with the interface through a dialogue of requests and responses.

In Tk-Gofer only a few basic interface components such as text entry boxes and buttons are defined in the imperative language. The layout and behaviour of the interface are defined in the functional language. Rather than use a dialogue, callback functions are registered for various actions that might be performed on the interface. The callbacks are described in a monadic style, and have access to mutable variables that allow interface components to have state.

The mechanism for converting a layout specification to an actual placement of Widgets on screen forms part of the imperative toolkit. The structure of an interface is declared in the functional language in a way similar to that of Concurrent Clean.

Figure 2.31 shows the Up-Down Counter in Tk-Gofer. A value of type `GIO a` is a monadic *action* that results in a value of type `a`. A mutable state variable is declared to hold the current number displayed. This is passed to the button function `but`, that uses it in the *callback* function attached to the button. The callback reads the current number, adds one to it, increases the displayed number then updates the state variable. The display is set up with a particular name, then altered from within the button. The layout combinator `^^` specifies that the display should appear above the button.

The program is very simple, but the components are not easily re-used. The mutable state variable `st` is *defined* at the top level of the program, *used* by the display `disp`, and *updated* from within the button `but`. In effect it is a global variable. Clicking the button causes the display to change by virtue of the fact that the display is named and this same name is used in the button callback function. The definition of each individual part of the program is inextricably linked to the

```
main :: IO ()
main = startProg updown

updown :: GIO ()
updown = do {
    st <- newState 0;
    openWindow [Title "Up-Down Counter"] (disp ^^ but st)
}

disp :: Widget
disp = labelW [Name "display",Relief Raised,Text "0"]

but :: Var Int -> Widget
but st = buttonW [Text "up",Command action]
    where
    action = do {
        s <- readState st;
        let {s' = (s+1)};
        set "display" (Text (show s'));
        writeState st s'
    }
```

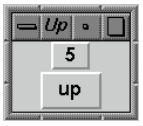


Figure 2.31: The Tk-Gofer Counter.

rest of the program, through the Widget names and state variable.

We can make a more re-usable program by adhering to a few programming conventions. Rather than using explicit names for Widgets, we can use a library function to obtain a unique name. A supply of names is hidden in the `GIO` monad, and a `tk_newName` action returns one of them. We pass this name to any other component of the interface that needs to interact with the Widget. Better still, we pass functions to perform specific tasks; that way we control what the other Widgets are able to do with the named Widget. For example, the display Widget of the counter can be defined:

```
disp :: GIO (Widget, Int -> GIO ())
disp = do {
    name <- tk_newName;
    let {wid = labelW [Name name,Relief Raised,Text "0"];
         update n = set name (Text (show n))
    };
    result (wid,update)
};
```

Whereas the old version of `disp` was a normal function, the new version is a monadic

action, giving access to the supply of names. The result of `disp` is a pair of the Widget and a function that when applied to an integer returns an action that changes the integer in the display. In effect, the display Widget is returning a display-changing callback function to the program that uses the Widget.

As far as possible, state variables should be defined in the place they are used. This makes for a more modular program. As the state variable in our counter is used to store the number being displayed, it should be declared and used in the display Widget. Then, instead of providing a callback to change the display with a particular integer, it provides a callback to alter the display with an integer-modifying function. This way we can alter the displayed number in a more flexible way. The *up* button can use the callback applied to the `(+1)` function as *its* callback, and we can easily add more buttons with other operations. The mutable state is now *safer* because it can only be altered using the callback provided by the display, and we have ensured that the state cannot be set to a new value without also updating the number displayed. The integer display is now easily reused in other circumstances. The improved counter is listed in Fig. 2.32.

The look and basic behaviour of interface components depends on imperative components but Tk-Gofer gives more control to the functional program than previous similar works.

2.5.3 BriX

The BriX system [Ser95] offers an interface to The X Windows System from the Haskell language, using monads for I/O and concurrent processes, with the special requirement that the system as a whole is deterministic. The reason is to keep programs entirely predictable. For example, in a non-deterministic system it may be impossible to predict the outcome of clicking two buttons in quick succession if the actions attributed to each button take different amounts of time to complete.

Monads are used to sequence events, and encapsulate the state of the I/O system. Functional counterparts of X Window system calls, such as `XCreateSimpleWindow` are created as monadic actions. GUI state such as window handles and graphics contexts are hidden away within the monad.

For a single window operated from a single thread there are no problems. To cater for multiple windows, there are two options: (1) keep a single thread and refer to separate windows with identifiers, or 2) create a separate, concurrently evaluating

```
main :: IO ()
main = startProg updown

updown :: GIO ()
updown = do {
    c <- counter;
    openWindow [Title "Up-Down Counter"] c
}

counter :: GIO Widget
counter = do {
    (dispW,update) <- intDisp 0;
    let {butW = buttonW [Text "up", Command (update (+1))]};
    result (dispW ^^ butW)
}

intDisp :: Int -> GIO (Widget,(Int->Int) -> GIO ())
intDisp i = do {
    name <- tk_newName;
    st <- newState i;
    let {wid = labelW [Name name, Relief Raised, Text (show i)];
         update f = do {
            s <- readState st;
            let {s' = s+1};
            set name (Text (show s'));
            writeState st s'
        }
    };
    result (wid,update)
}
```

Figure 2.32: A More Re-Usable Counter in Tk-Gofer.

thread for each window. The latter solution makes for much tidier programs as in the former programs tend to revolve around a single event-processing structure.

Introducing concurrency without non-determinism requires some ingenuity. Three examples are highlighted: (i) if two windows (concurrently operating) both try to raise themselves to the top of the display at the same time, this operation is non-deterministic. It is impossible to tell which will raise first, and which will end up on top. This problem is avoided by only allowing the parent process of the two windows to raise them. Then the order of raising is well defined.

(ii) shared access to a file. A method called *Object Coupling* solves this problem. Objects can be one of two sorts. Coupled objects are Owner Read, Owner

Write (OROW), ie. one process has exclusive access to a particular piece of state. Decoupled objects are Concurrent Read, No Write (CRNW), ie. a shared resource that cannot be changed. A file that is opened is treated as a coupled object, even if read-only (as there is some state encapsulated in the file pointer). A file can be shared amongst a number of processes by reading the contents into a list, and sharing this list as a decoupled object.

(iii) merging of streams from many processes to a single process. Take the up-down counter for example. Each of the two buttons can send information to the display. The display cannot tell where to look for a value first, from the up or down counter. This problem is solved by feeding the events that trigger each button to the display also, and arranging for the display to know which events are relevant to each button. This way, the display can test each event to see if it is relevant to a particular button and, if it is, receive a value from that button. The extra plumbing involved in passing events to the display as well, and instructing the display as to which events are relevant to each button, can be hidden away by abstraction.

The BriX system has shown that it is possible to construct a functional GUI without sacrificing deterministic behaviour. As yet, only a few simple examples have been tested to demonstrate the ideas.

2.6 Assessing Functional GUIs

In this section we argue that certain features are desirable in an environment for developing applications with graphical user interfaces. These features were chosen as a result of the practical experience of using various systems to write programs. For this reason, it is not an exhaustive list, and some features such as expressiveness of language and amenability to proof are left as implicit requirements. We measure the Fudgets, Clean, Haggis, Tk-Gofer and BriX systems against the features we describe.

2.6.1 Intuitive program structure

The structure of a GUI program should be close to the structure the user perceives by looking at the interface. For example, clicking a button on the screen has an effect on the program, usually shown in some other object on the screen. This link should be clear in the program. Conversely, a GUI program built around a central

event-polling loop is *not* similar in structure to the interface it implements.

If a program can be broken down so that the behaviour of each *object* is defined by a separate function or module, then the structure of the program will reflect the structure of the interface.

- **Fudgets:** a function in the continuation passing style describes a process and this defines the behaviour of a Fudget.
- **Clean:** a leaf element of the IOState represents an object. The event-handler defines the behaviour of the object by returning an altered state.
- **Haggis:** a function in the monadic style describes a process that defines an interface object.
- **Tk-Gofer:** the Widgets defined in the imperative language are objects. Call-back functions in the monadic style define the behaviour of each object.
- **BriX:** functions in the monadic style define each object.

2.6.2 Concurrent behaviour

We want objects to operate concurrently. This brings benefits in situations where two actions could be occurring at the same time. For instance, one object may be updating the screen at the same time as another is responding to mouse clicks. A more important reason for having concurrency is that it enables us to break down a program into parts. Each part may be specified independently, as if it is the only part in operation (except where parts interact with one another, when some assumptions must be made). Consider a menu object. The user can open the menu, move around multiple levels of selections (if it has a tree-like structure) and close the menu again, all without the menu object interacting with any other part of the program. While this is happening, another part of the program may be updating the contents of a window, or providing an animated display. In our definition of the menu, we want to concentrate on the operation of this object alone, and ignore whatever the rest of the program is doing. We do not want to have to release control from the menu object at regular intervals so that other objects can be *serviced*, as is necessary in event-polling loop style programs.

- **Fudgets:** Conceptually, Fudgets are processes that operate concurrently, and this is reflected in their definition. The syntax of Fudget definitions and combinations does not specify any kind of ordering on the evaluation. However, Fudgets are currently implemented as a purely sequential evaluation which limits their expressiveness.

- **Clean:** being based on callback functions, objects are defined independently, but the definition is given in terms of the response to an event, rather than being a definition of the continual operation of the object. Any state associated with the object must be explicitly returned by the callback function, rather than holding the state value in a recursive function.

- **Haggis:** objects are defined as concurrent processes. The state of an object may be held in a recursive function, or stored in a mutable state variable to reduce the plumbing in the function.

- **Tk-Gofer:** as with Clean, the behaviour is defined by callback functions. But state can be stored in mutable state variables instead of being returned by the callback.

- **BriX:** objects are defined as concurrent processes. The state of an object may be held in a recursive function, or stored in a mutable state variable.

2.6.3 Objects can communicate with one another

The objects of an interface interact with the user and with each other. For example, when clicked, a menu opens to reveal its contents. If one item of the menu represents a further menu then the user may again click to open. During these operations, the menu is interacting with the user. When the user clicks a menu option, the menu interacts with another object on the screen, perhaps opening a new window or deleting some highlighted text.

- **Fudgets:** are connected to each other through their high level streams, though each Fudget is restricted to one input and one output. Communication to or from more than one other Fudget is achieved by combining Fudgets into one. Fudgets interact with the user through their low level streams. The Fudget combinators direct messages to and from the I/O hardware to the relevant Fudget.

- **Clean:** objects interact with the user by returning a new IOState (thereby performing I/O). They interact with one another by altering a shared program state. Alternatively, in the new version of the system concurrent processes can be defined that communicate by message passing. The built-in interface components do not use concurrent processes however.

- **Haggis:** objects can communicate with each other through message channels. They communicate with the user by altering an I/O state.

- **Tk-Gofer:** objects communicate with the user through the Widgets defined in the imperative language. Objects can communicate by passing callbacks to each other as described in §2.5.2.

- **BriX:** objects communicate with the user by altering an I/O state. Objects communicate with each other through lazy streams or shared mutable state.

2.6.4 Specifying lines of communication between objects

We require that objects may be combined and the internal interaction between components of an object be made invisible from the outside. We require locality of definition and independence of specification. In other words, we want to define general purpose objects knowing only how they interface with their surroundings, not what they will be connected to.

When an object communicates with another, it may *address* the other object *directly* or *indirectly*:

Direct addressing This kind of address takes the form of a unique identifier. For example, an address might be *the object called 'mouse pointer'*. Confusion will arise if more than one object has a particular identifier, unless this is intended to be a means of communicating a message to more than one object at once. This type of addressing is useful for objects to communicate with OS devices such as the mouse pointer, or a general purpose warning box window object.

Indirect addressing With this form, objects are defined to communicate with other objects the identity of which is not known at the time of definition. The other objects are referred to with a label, and this label is *bound* to a particular object when the object is finally used in a specific setting. This is similar to the

way functions are defined with parameters, though we don't know the value of the parameters at the time of declaration. The function is applied to particular values later. This form of addressing is useful for defining general purpose objects, the use of which will be decided after their definition is complete.

- **Fudgets:** have a limited kind of parameterised addressing. Each Fudget has one input and one output. Other Fudgets are connected to each of these by Fudget combinators. Communications with a Fudget other than the two directly connected must be arranged with all the intervening Fudgets. It is down to the programmer to arrange these connections. It can be a complicated task and connections are difficult to modify.

  Fudgets address OS objects with a form of global addressing, but cannot address application Fudgets using the same method. Messages from OS objects to Fudgets are routed by the combinators to the correct Fudget, and this is equivalent to global addressing.

- **Clean:** the event-handling functions are equivalent to global addressing for OS objects communicating with application objects. Application objects use global addressing to communicate with OS objects, by calling library functions with the object ID as an argument. Application objects can only communicate by leaving messages in the global state. Parameterised addressing can be achieved by passing state-accessing functions to objects when they are created. These functions specify an "address" by accessing a particular piece of the global state. There is no direct way to *awaken* the receiving object to the fact that there is a message waiting in the state. The indirect way is to cause the OS object that has the application object's event-handler to call the event-handler.

- **Haggis:** each object creates unique addresses in the form of handles that are returned to the object that creates it. These handles are then passed as parameters to any object that wants to send information to the object.

  Access to I/O devices through the I/O state is equivalent to global addressing.

- **Tk-Gofer:** the Widgets defined in the imperative language are passed addresses as parameters. Objects defined in the functional language can communicate by passing each other monadic callback functions that when applied to a value, have the effect of communicating it to the object.

- **BriX:** objects can communicate via lazy streams or through shared mutable state. References to these streams or state are passed as parameters. There are restrictions as to which objects can communicate, due to the requirement that the program be deterministic.

  Access to I/O devices through the I/O state is equivalent to global addressing.

2.6.5 Type-checked communication

We want communication between objects to be statically checked so that we can be sure that the objects we have joined are capable of cooperation. It cannot ensure an object has a response to all possible values of a particular communicated type.

- **Fudgets:** connections between Fudgets are checked to see that messages of the same type are being communicated. However, each message stream is of a fixed type (possibly a sum type), so it is necessary for the programmer to define a new sum type for each required combination of message types. In Haskell the constructor names must be unique (across all types defined) which causes problems when several sum types have a common element — each must have a different name. This might tempt a programmer to define one sum type for all types of messages sent, and then to use this type whenever a subset of those types is required, effectively disabling the type-checking of messages. A better system would allow any of a set of types to be communicated between objects. Haskell-B's *existential types* might provide a means to do this [Aug93, Lau94].

- **Clean:** most communications happen *via* the global state. Though all functions that use or replace the state are type-checked statically, any part of the state may be accessed and there are no checks that the state is being used as intended by the programmer.

- **Haggis:** functions are provided that use the handles returned from Widgets to effect a communication, and these functions are applicable only to certain types.

- **Tk-Gofer:** callback functions passed between objects are statically typed-checked.

- **BriX:** both lazy streams and shared mutable state are statically typed.

2.6.6 I/O devices as objects

We want I/O devices to appear as objects, similar to objects of a program, but their internal workings will somehow be connected to the computer hardware. By giving access to I/O devices in this way we bring their use in line with program and library objects. Simple I/O devices can be wrapped up inside software layers and appear as more complex I/O devices. For example, a *keyboard* might simply emit characters whenever a key is pressed, not knowing which part of the user-interface should receive them. By wrapping the keyboard object inside a software object, we can provide an user-input object that can *connect* with multiple user-input fields in an interface.

Another benefit is that we can swap I/O device objects for program objects, and vice-versa, for the purposes of testing and debugging. For example, when designing an object that will eventually be connected to another object that sends it characters, it could be tested by connecting it to the keyboard object instead.

If we allow objects to communicate with the I/O hardware by altering an I/O state then we may have difficulty controlling access to resources. How should we stop two objects both writing to the same file? It would be possible to monitor the changes made to the I/O state by each object and disallow any conflicting changes. It would be clearer to give access to I/O devices through special objects, then wrap these objects in a software object that manages access to that resource, making explicit the fact that the resource is shared.

- **Fudgets:** access to I/O devices is through the low-level message streams. The Fudget combinators direct low level messages to and from the relevant Fudgets. Safe, shared access to the screen display is handled by the combinators. Access to files is achieved by integrating the Haskell I/O dialogue model into Fudgets.

- **Clean:** I/O devices are accessed through callbacks returning an altered IO-State. Functions are provided to alter the state, and these functions ensure there is no clash in the use of resources.

- **Haggis:** I/O devices are accessed by altering an I/O state that is passed through the monadic combinators. Only library defined monadic I/O functions can access the I/O state.

- **Tk-Gofer:** the Widgets defined in the imperative language have access to the

I/O devices. The callback functions generate a dialogue that communicates with the imperative Widgets.

- **BriX:** I/O devices are accessed by altering the I/O state that is passed through the monadic combinators. Only library defined monadic I/O functions can access the I/O state.

2.6.7 Composition of objects

We want the interface to be constructed from *building-blocks* or *objects*, that may be re-used in other similar situations. A library of standard objects can be provided and these may be used to build further more complex objects, or new objects may be built from scratch. Being able to compose objects easily and flexibly facilitates re-use of common components, saving time and effort.

- **Fudgets:** using combinators, Fudgets may be composed in a variety of ways to form more complex Fudgets.
- **Clean:** the hierarchy of device objects has a fixed number of levels, and objects cannot be composed to form a new sort of object of the same type as other objects.
- **Haggis:** objects are fully composable. At the top level of the hierarchy, a function is applied to the composition to create a Widget, but this Widget is of a different type and cannot itself be composed with anything else.
- **Tk-Gofer:** the imperatively defined Widgets may be composed but result in an object of a different type. However, these composed objects may themselves be composed, so it is not a problem. The only effect is that the basic Widgets are declared differently from composed objects.
- **BriX:** objects may be composed.

2.6.8 Objects have local state

Most user interface components have some state associated with them. Some examples are the integer in an integer displayer and the fact that the *thumb* in a scroll bar has been clicked and is being dragged by the mouse. All the systems we

have seen allow objects to have local state. The difference lies in how that state is accessed. Later, in §5.7.2, we look at the different ways of managing state in an object.

- **Fudgets:** each Fudget process can hold some local state as an explicit value in a recursive function.
- **Clean:** in the latest version the callback for each object is passed some private local state with the global program state when called.
- **Haggis, Tk-Gofer, BriX:** local state can be stored in mutable variables.

2.6.9 Static or dynamic objects

A system of static objects is one in which the number of objects is determined at compile-time, whereas a system of dynamic objects is one in which objects are created during the execution of the program. A system of static objects may be simulated within a dynamic system, but not vice-versa, so a dynamic system is a more general choice.

- **Fudgets:** processes are mostly determined at compile-time, though there is limited support for dynamically creating new Fudgets. A special list of Fudgets may have new Fudgets added to it. These are addressed by indexing.
- **Clean:** new I/O objects such as menus may be created and assigned an event-handling function (which corresponds to the process of an object). The rest of the program must somehow learn of the new object's identifier in order to interact with it though (via the global program state perhaps).
- **Haggis:** process (and hence object) creation is dynamic.
- **Tk-Gofer:** imperative Widgets are created on screen when a program is first executed, and it is not possible to create more during program execution.
- **BriX:** as objects each have a controlling process, it should be possible to create more dynamically. But as the system is in its early stages of development it is not yet clear how this is to be achieved.

2.6.10 GUIs in a functional style

We want to be able to build graphical user interfaces rapidly, and in a functional style. We would like to be able to use the tools of abstraction, higher-order functions and laziness, both in the definition of our objects, and in the combination of such objects. In *definitions*, for example, we want to be able to use abstraction for such things as generating a list of drawing primitives from an abstract definition of a picture. Object definitions should be concise and clear, such as is typical for a functional definition of an algorithm. When *combining objects*, for example, can we `zip` together a list of objects with a list of arguments for the objects, and connect them in a chain using `fold` on the list with an object combining function? Can we define a higher order function which takes a list of any type of object which may be *selected* by the user, and forms a radio-group with them, in which one of the objects is selected at any time, and selecting a new object deselects the previous one?

- **Fudgets:** within Fudget definitions, higher-order functions such as `map` and `foldr` can be used to good effect for generating lists of messages, for example to alter the drawings in a set of Fudgets. In combining Fudgets, a Fudget combinator may be used to `fold` a list of Fudgets.

- **Clean:** within event-handling functions, abstraction can be used to get away from primitive drawing commands, etc., but the lack of direct communication between application objects obviates the need for higher-order functions to generate messages. There are less opportunities for using abstraction outside the definitions of objects because the system lacks combinators for composing objects.

- **Haggis:** the use of handles has a slightly imperative feel to it, but in general laziness and abstraction are useful in writing Haggis programs.

- **Tk-Gofer:** if the programmer avoids hard-wiring the names of Widgets directly into definitions and passes functions to access mutable variables, then the power of abstraction is put to good use.

- **BriX:** as with Haggis, the use of handles gives a slightly imperative feel to the system, but otherwise abstraction and higher-order functions can be used well.

2.6.11 Conclusion

All the systems reviewed meet our requirements in some respects, but all fall short in some aspects. A combination of the good locality of object definition of the Fudgets system together with a more flexible means of communication between objects and the independent definitions of the process-based systems would be an improvement. This still leaves open the questions of how to safely access I/O devices and express dynamic creation of objects.

Chapter 3

Concurrency

3.1 Introduction

This thesis claims that by adding concurrent processes to a functional language we improve its suitability for programming GUIs. In this chapter we explain our reasons for wanting concurrency. We describe the development of three successive systems of extensions to a function interpreter that we use to test our ideas later in the thesis. Later we present some techniques we use in the extended language.

Why Concurrency?

On the face of it, functional languages might seem to be ideal for programming applications with graphical interfaces — the unimportance of the order of evaluation should map nicely to the unknown order of interaction present in a typical graphical interface [Dix87]. In practice, if we want to maintain the benefits of lazy-evaluation, things are not quite so simple. Although the order of evaluation of a pure functional program is not important, when adopting the lazy evaluation strategy a specific order *is* chosen. In a graphical interface each element (windows, icons, etc.), is a source of input. If each source of input is introduced into the graph at a different point, then parts of the graph will not be reducible until the relevant user-input arrives. In these circumstances, the order of evaluation *is* significant. We want the evaluation to proceed as pieces of input become available, not get *stuck* waiting for a particular piece of input because an arbitrary choice has been made about the order of evaluation.

One solution might be to alter the evaluator so that it can postpone an evalu-

ation if it turns out that a piece of input is required that is not yet available, and evaluate some other piece of the graph until the input arrives. This would require the evaluator to *mark* parts of the graph that are blocked, waiting for some input, so as to stop the evaluator trying again before the input has arrived. The system would have to *un-mark* these parts when the input did eventually arrive, by which time the evaluator might be evaluating some other part of the graph, and not return until that is complete. This solution is not multi-tasking — it would not, for instance, enable the program to respond to an event whilst another piece of output is happening. Neither does it allow us to describe a program where the order of interaction *is* important, eg. where a menu selection affects the way another part of the interface reacts.

Another solution would be to merge all the different sources of input into a single stream, in chronological order. For example, the solutions suggested by Dwelly and Stoye [Dwe89, Sto85b] extend the textual character streams method into streams of requests and responses. All input is merged into the stream of responses, and output is multiplexed through the stream of requests. A program consists of functions that respond to particular events, combined into one stream-processing function that is applied to the input responses to produce the output requests. Such a program cannot easily respond to one event whilst producing output for another, even if the response stream *can* consist of multiple outputs spliced together. To achieve this, each event-processing function must specifically be programmed to allow others to take over from time to time during its evaluation of the response to some input. Communication between parts of the program designed to service different areas can only be achieved through a shared global state. This makes the definition of modules that enter into private communication impossible and can lead to a large and unwieldy state that is difficult to maintain when the program is changed.

Our solution has a number of evaluations running concurrently, feeding separate sources of user input to separate processes, and similarly for output. When one evaluation cannot proceed until some user-input is received or some event occurs, another evaluation takes over. This might be viewed as a specialisation of the first suggestion, in which the points at which the evaluation may be postponed are specified by the application programmer by defining separate processes.

By specifying concurrent processes we can indicate directly where concurrent interactions are allowable, and where operations must be performed in a set sequence. The response to an event need not be completed before another event can be ser-

viced — a context change between processes may occur at other times aside from the case where input is not available.

It is often desired that user input from one source affect the response to user input from another source. For example, in a word-processor, highlighting a section of text results in the *Delete selection* menu item becoming available. To facilitate this kind of functionality, we allow processes to communicate with one another by message passing. So in the word-processor example, when the user highlights some text, the process responsible for text highlighting sends a message to the process responsible for the menu, indicating that something is now highlighted. The menu process responds by making the *Delete selection* item appear on the menu.

Developing a Concurrent Process Functional Language

We have added some primitives to a functional language to provide concurrent processes. We have progressed through a number of systems, testing each one for suitability and making improvements for the next. Our earliest experiments used concurrent processes that communicated via a *sorting office*, but suffered some drawbacks relating to the types of messages sent and values used to identify processes. Our second system overcame these problems, but was awkward to use, requiring the programmer to include some information in the program that, if forgotten, would cause a run-time error. The final system removes this necessity, and is the one we use to program GUIs, described in later chapters. In the following sections we describe each of the systems we have developed, highlighting the advantages and disadvantages of each, and the reasons for requiring further development. The final system is described in detail.

3.2 System A — Embedded Gofer

This is a modified version of the Gofer System [Wal95] adapted from the original by Malcolm Wallace who wished to use it to program embedded systems. In Embedded Gofer a number of functional expressions represent processes that are evaluated concurrently. Processes communicate by asynchronous message passing. Messages are stored in queues, one for each process, until a receiving process takes them one at a time. Each process is a function from a single input stream of incoming messages to a single output stream of outgoing messages, and has a unique process

identifier (PID). Outgoing messages are tagged with the PID of the destination process. The run-time system includes a sorting-office [Sto85a] that is responsible for taking output messages from processes and inserting them into the input stream of the process with the specified PID. An incoming message is tagged with the PID of the sending process.

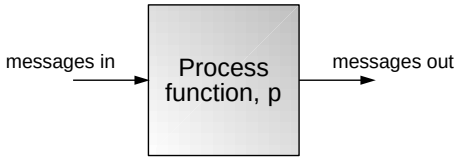


Figure 3.1: A Process in Embedded Gofer.

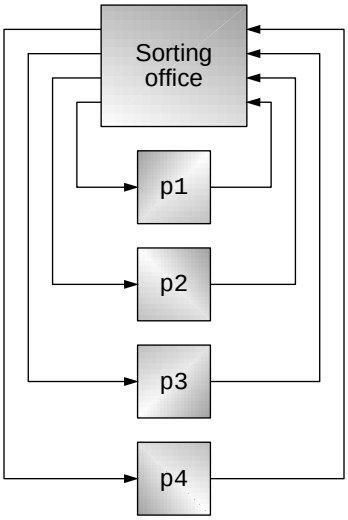


Figure 3.2: A Program in Embedded Gofer.

The sorting office performs a non-deterministic merge on the messages sent to each process, posting them to the correct process in the order they arrive. Because of this, a program given the same input twice may give different output each time, depending on the exact arrival time of each piece of input. In effect, user input involves not only the input data (a mouseclick at a certain point, for example), but also the timing of such data, even though the timing information is not available to the program.

A program built from a network of processes is specified by a list of process declarations. When such a list is evaluated, the resulting processes are created and set into action. A process declaration is of type `ProcDec`:

```
data ProcDec pid msg = Proc pid (Process pid msg)
```

where a `Process` is a function from a list of messages paired with the PID of the sending process, to a list of messages paired with the PID of the destination process:

```
type Process pid msg = [(pid,msg)] -> [(pid,msg)]
```

A simple example program is shown in Fig. 3.3. It consists of three processes that together display the running total of the sequence of numbers 1, 2, 3... The first, `Counter`, outputs a stream of messages to `Summer`, starting with `(NextNum 1)`, then `(NextNum 2)`...Process `Summer` receives these numbers and outputs a running total. It sends the total, converted to a string, to the `Printer` process. The definition of `Printer` is not shown here so as not to complicate the example with the mechanisms associated with output to the screen.

```
data Pid = Counter | Summer | Printer
data Msg = NextNum Int | NewSum String

prog = [Proc Printer printer, Proc Summer summer, Proc Counter counter]

counter :: Process Pid Msg
counter = map toSummer [1..]
    where
    toSummer n = (Summer,NextNum n)

summer :: Process Pid Msg
summer = summer' 0
    where
    summer' t ((Counter,NextNum n):ms) =
        (Printer,NewSum (show nn ++ " ")):summer' nn ms
    nn = t+n
```

Figure 3.3: An Integrator built from Processes

3.2.1 Shortcomings

The system described above has a number of shortcomings with respect to the features we require for programming GUIs, as described in the final section of Chapter 2. These relate to the re-usability of processes, type-checking of messages, support for dynamic processes and scheduling. We look at each in turn:

I. Re-use of processes The PIDs and tags used by the processes in the previous example are *hard-wired* into the definition. For example, the `summer` process receives a message with tag `NextNum` from a process with PID `Counter`, and sends a message with tag `NewSum` to a process with PID `Printer`. Suppose we wish to re-use the `summer` in another program. It will be connected to processes with different PIDs and in most cases will use different tags for the messages too. A simple process like this could be altered by changing the definition, but this is not practical for a larger process, or a large number of processes, and in a system supporting separate compilation would require the process to be re-compiled. What if we parameterise the process on the PIDs and tags used? There are two problems with this, both caused by limitations of pattern matching:

Parameterisation of PIDs

Consider an example process `intMemory`:

```
intMemory :: Int -> Process
intMemory n ((IntProducer,IntMsg n'):ms) = intMemory n' ms
intMemory n ((Trigger,UnitMsg):ms) = (IntConsumer,IntMsg n):intMemory n ms
intMemory _ _ = error "Unexpected message or source"
```

The `intMemory` receives integers from a process with PID `IntProducer` and stores them. It sends the stored integer to a process with PID `IntConsumer` when a unit message `()` is received from a process with PID `Trigger`. The process is declared to store integer zero initially with:

```
intMemory 0
```

We cannot test statically that it will only receive messages from the `IntProducer` or `Trigger`, and that in each case the associated message will be the correct sort. The `IntProducer` may send a unit message for example. We catch all combinations of errors in the last line.

Leaving the message tags as they are for the moment, consider a parameterised version. The parameters `i`, `t` and `o` replace the constructors `IntProducer`, `Trigger` and `IntConsumer` in the definition:

```
intMemory :: Pid -> Pid -> Pid -> Int -> Process
intMemory i t o = intMemory'
    where
    intMemory' n ((i,IntMsg n'):ms) = intMemory' n' ms
    intMemory' n ((t,UnitMsg):ms) = (o,IntMsg n):intMemory' n ms
    intMemory' _ _ = error "Unexpected message or source"
```

This definition is not valid because we cannot pattern match on a parameter. Constructors in patterns must be constants. Neither can we use a `case` expression for similar reasons. The alternative is to compare each PID for equality in turn:

```
intMemory :: Pid -> Pid -> Pid -> Int -> Process
intMemory i t o = intMemory'
    where
    intMemory' n ((src,msg):ms) =
        if src == i then case msg of
            IntMsg n' -> intMemory' n' ms
            otherwise -> error "Unexpected message"
        else if src == t then case msg of
            UnitMsg -> (o,IntMsg n):intMemory' n ms
            otherwise -> error "Unexpected message"
        else error "Unexpected source"
```

We can connect the `intMemory` as previously hard-wired with:

```
intMemory IntProducer Trigger IntConsumer 0
```

We can now connect it to different processes without changing its definition, so long as the other processes also use the `IntMsg` and `UnitMsg` tags. This is no less expressive but is inconvenient. The `Pid` type must be of class `Eq`. The integer can only be extracted from the message using pattern matching after the source PID has been identified, because up until that point we do not know which tag the message has. Again, it cannot be checked statically that messages will only be received from one of the stated sources, and that in each case the message will be the correct one. Now that we are not using pattern matching, we have to test for these errors in more than one place.

Parameterisation of message tags

Previously we used pattern matching to extract the contents of a message (the integer in the example). We cannot use pattern matching with a parameter, so instead we pass in a function to extract the value. This function also tests the equality of the message tag:

```
intMemory :: (Pid,Msg->Maybe Int) -> (Pid,Msg->Maybe ()) ->
                                    (Pid,Int->Msg) -> Int -> Process Pid Msg
intMemory (i,ti) (t,tt) (o,co) = intMemory'
    where
    intMemory' n ((src,msg):ms) =
        if src == i then case (ti msg) of
            Yes n' -> intMemory' n' ms
            None -> error "Unexpected message"
        else if src == t then case (tt msg) of
            Yes () -> (o,co n):intMemory' n ms
            None -> error "Unexpected message"
        else error "Unexpected source"
```

We can connect this `intMemory` as previously hard-wired with:

```
intMemory (IntProducer,unpackInt) (Trigger,isUnit) (IntConsumer,IntMsg) 0
    where unpackInt (IntMsg n) = Yes n
          unpackInt _ = None
          unpackUnit UnitMsg = Yes ()
          unpackUnit _ = None
```

The definition is now re-usable, but much more cluttered. We can even increase the degree of parameterisation by changing each instance of `Int` in the type of `intMemory` to a polymorphic type parameter, and the function name to `memory`:

```
memory :: (Pid,Msg->Maybe m) -> (Pid,Msg->Bool) -> (Pid,m->Msg) ->
                                                    m -> Process Pid Msg
```

The process is now polymorphic in the type of messages it stores. This was not possible before because the hard-wired message tags determined the type of message. We can define type synonyms for the parameters associated with input and output from a process, to clarify the type of a process:

```
type In m = (Pid, Msg -> Maybe m)
type Out m = (Pid, m -> Msg)
```

and the type of our `memory` can now be expressed:

```
memory :: In m -> In Bool -> Out m -> m -> Process Pid Msg
```

II. Type-checking of messages This problem has already been encountered in the previous section. The processes of a program are declared in a single list, so they must be of the same type, thereby forcing them to expect and emit the same types of messages as each other. All types of messages that are used in a given

application are encoded in one abstract datatype (eg. `Msg` in the example of §3.2). The type-checker will not spot errors where a message sent *is* of the right type (ie. the message type) but does not use the constructor that the receiving process is expecting. The best we can achieve is to signal a run-time error.

Wallace developed a solution to this problem using a system of classes that ensures message passing is type-checked for each sort of message [WR94]. However, his solution is usable only in a system in which processes are *static*.

III. Composition of Processes Suppose we wish to create a new kind of `memory` process that can store values from two inputs, and emit these from two outputs on receipt of a single trigger. We would like to re-use the `memory` process already defined by combining two of these into something that looks like a single process. A number of processes could be combined manually by writing a new definition based on a number of old ones, but this would be a time consuming task. We would prefer to make use of a number of definitions we already have. What we want is to arrange a set of processes in such a way that they appear to the rest of the program to be a single process.

The first problem is that the program structure does not allow for *hierarchy*. A program consists of a number of processes that are all joined on the same level. If we make process definitions return a list of processes instead of a single process, then a function describing a single process can return a list containing one process, and a composition can return a list of many. We can `concat` these lists to form a single list to define the program.

The first step is to change the type of processes:

```
type Process' pid msg = [(pid,msg)] -> [(pid,msg)]
type Process pid msg = [ProcDec pid (Process' pid msg)]
```

Previously the PID was associated with each process at the top level of the program in the process declaration list. As the processes in a composition need to know each other's PIDs to communicate with one another, we must know the PIDs inside the definition of a composition. If we choose a type of PID values that can be extended (eg. `String`) then we can pass a single PID to each function, and if the function defines a composition it can extend the PID to form multiple PIDs. Now process functions can return a list of process declarations — process functions attached to PIDs. Our `dualMemory` returns a list of three process declarations (assume the

declaration of `memory` has been changed so that it accepts a PID and returns a list of process declarations):

```
memory :: Pid -> In m -> In t -> Out m -> m -> Process Pid Msg

tee :: Pid -> In m -> Out m -> Out m -> Process Pid Msg
tee pid (i,ti) (o1,co1) (o2,co2) = [ProcDec pid teeproc]
    where
    teeproc :: Process' Pid Msg
    teeproc ((src,msg):msgs) =
        if src==i then case ti msg of
            Yes m -> (o1,co1 m):(o2,co2 m):teeproc msgs
            None -> error "Unexpected message"
        else error "Unexpected source"
dualMemory :: Pid -> (In m,In n) -> In t -> (Out m,Out n) -> (m,n) ->
                                                              Process Pid Msg
dualMemory pid (i1,i2) t (o1,o2) (s1,s2) =
    tee tpid t (s1pid,packUnit) (s2pid,packUnit) ++
    memory s1pid i1 (tpid,snd t) o1 m ++
    memory s2pid i2 (tpid,snd t) o2 n
        where
        tpid = pid++"t"
        s1pid = pid++"1"
        s2pid = pid++"2"
        packUnit = ?
```

Unfortunately, we do not know how to tag the messages sent from `tee` to each of the `memory`s (hence the `?` definition of `packUnit`). We have access to an unpacking function for the message because it happens that the messages sent between `tee` and the `memory`s use the same tag as trigger messages sent into the composition. We could alter the `tee` process to copy the message with the same tag (removing the need for the `packUnit` constructor):

```
tee :: Pid -> Pid -> Pid -> Pid -> Process Pid Msg
tee me i o1 o2 = [ProcDec me teeproc]
    where
    teeproc :: Process Pid Msg
    teeproc ((src,msg):msgs) =
        if src==i then (o1,msg):(o2,msg):teeproc msgs
        else error "Unexpected source"
```

but then the type of `tee` does not follow the same conventions as other processes, and this solution is no help if two processes in a composition wish to communicate messages of a type not sent into or out of the composition. In this case, suitable tags must be passed into the composition, revealing the fact that the process is a composition and increasing the burden on the programmer.

IV. Declaring and maintaining PID and message tag types The PID and message tag types are declared at the top level of a program to include PIDs for all processes used, and a message tag for every type of message sent between processes. Using a type of PID whose values are extensible, such as the `String`s used in the previous section removes the need to allocate all PIDs at the top level of a program. We cannot perform a similar trick for message tags because messages are of different types. Every time the programmer wants a new type of message in a program, the global message type must be altered to include a new tag, and *packing* and *unpacking* functions passed into the relevant process definitions.

V. Dynamic processes Evaluation of a program in this system has two distinct phases. In the first phase, the list of process declarations is evaluated and the processes created. No communication occurs during this phase. In the second phase, the processes are evaluated and communicate with one another. No further processes are created during this phase. The first phase effectively creates a *static* network of processes — all processes exist at the start of process execution. This is fine for programming real-time systems (the original motivation for developing the system), but dynamically created processes would be useful for building dynamic graphical user interfaces in which new objects arise during execution of the program.

VI. Scheduling The scheduling of processes in this system is quite simple and can cause problems. Processes have a priority ordering specified by the process declaration list. The first process declared has the highest priority. Each process first has the opportunity to perform some initialisation before receiving any messages. As soon as a process sends or tries to receive a message, it ceases to be initialising. When the scheduler is invoked, it picks the process with the highest priority that has a message waiting in its incoming message stream. This can result in one process or a group of processes *taking over*, and not letting the rest have any chance to evaluate. In certain circumstances a process might never be run (after initialisation) because other higher priority processes always have messages waiting, and therefore are always chosen for evaluation in preference to the lower priority process. Simply raising the priority of the process often reverses the problem, with another process remaining suspended instead.

3.3 System B — Gofer with Components

System B attempts to address the problems of system A.

Components In system B, processes have a number of *pins* through which messages are passed. We call processes with pins, *Components*. Communication between Components is defined by connecting pins with *wires*. Each pin or wire conveys messages of a particular type. Messages in wires are not tagged, but travel as raw values. A wire can only connect an input pin that expects messages of type `m` to an output pin that emits messages of type `m`. Connecting two inputs pins, two output pins or an input and output pin of different message types will cause a static type error.

Components can be defined directly or composed of other Components. From the outside, a composition cannot be distinguished from a direct definition. Components are defined using the continuation passing style (CPS) to sequence operations.

Components can create new wires and other Components dynamically at any point in their evaluation. Evaluation of a Component program is begun by *launching* a single Component that creates a composition of the top level of Components in the program. Each of these Components may itself be a composition, so a program forms a hierarchy.

New Scheduling Algorithm In system A the scheduler caused problems that resulted in queues of messages building up. To address this problem, in system B the scheduling decision is based upon the number of messages waiting for each Component. A count of *messages waiting* for each Component is required. In Embedded Gofer it is difficult to keep a message count because messages are consumed by the process function evaluating the incoming message stream. Evaluation of a process is performed by the normal evaluator in the Gofer interpreter. This does not treat the evaluation of a process as different from any other expression, so it is not possible for it to know when to alter a message counter.

To solve this problem, we use primitive functions to send and receive messages. When the primitive function that sends a message is evaluated, the message count of the receiving Component is increased. The primitive that receives a message reduces the count.

Inside Components Inside a Component, a stream-processor operating on lazy lists of messages still defines the operation of the process. As there is a single stream in and a single stream out, messages *inside* a Component *are* tagged. In contrast to system A, however, the message tags are not global to the whole program. Two sets of tags are defined per Component, one for input and one for output messages. Each tag is associated with one of the Component's pins. The association is defined by the Component at the start of its operation. The operation of a Component is illustrated in Fig. 3.4. The function defining a Component is the square at the centre of the diagram. The rest is handled by the run-time system.

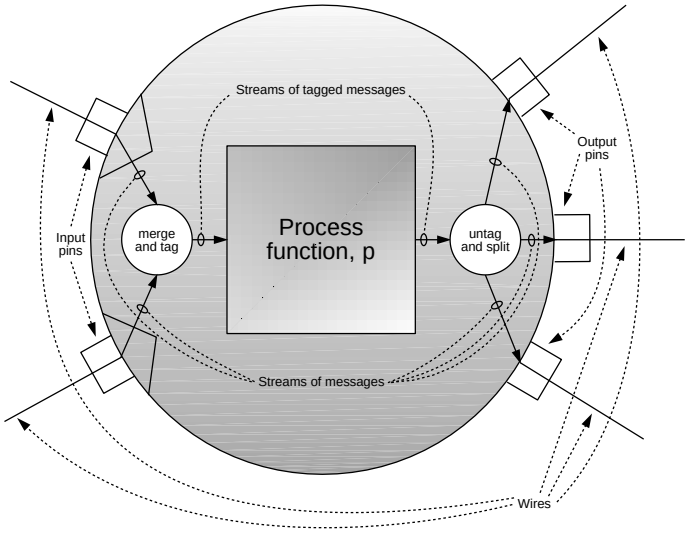


Figure 3.4: Inside a Component.

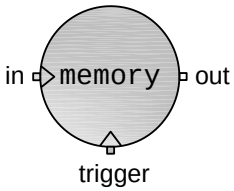


Figure 3.5: The memory Component.

Example Component We use the `memory` example from the previous section. In system B the type of the `memory` Component is:

```
memory :: In m -> In t -> Out m -> m -> Process MemoryIn MemoryOut
```

This looks very similar to the final version of `memory` in system A, but for the process type. The process type is parameterised on the input and output tag types, `MemoryIn` and `MemoryOut`. The pin types have the same names as those in system A, but they are different also. An `In i` parameter is the end of a wire that carries messages of type `i` *into* a Component. Similarly, an `Out o` parameter is the end of a wire that carries messages of type `o` *out* of a Component. The tag types are defined:

```
data MemoryIn m = MemoryIn m | MemoryTrigger ()
data MemoryOut m = MemoryOut m
```

Process types and primitive functions The type of a process is:

```
type Process i o = [i] -> [o]
```

and the functions that receive and emit a message have type signatures:

```
rx :: (i -> Process i o) -> Process i o
tx :: o -> Process i o -> Process i o
```

These functions use the continuation passing style (CPS) to thread the streams of input and output messages through the program invisibly. Expanding out the `Process` type synonyms in the type of `rx` and `tx` we get:

```
rx :: (i -> [i] -> [o]) -> [i] -> [o]
tx :: o -> ([i] -> [o]) -> [i] -> [o]
```

The first parameter of `rx` is a *continuation*. It represents *the rest of the program* after performing the `rx` operation. The `rx` primitive takes a message from the head of the input stream (the second parameter to `rx`) then applies the continuation to the message and the remaining input, to result in the output stream of messages. The `tx` primitive returns a result of the output message (the first parameter) followed by the result of the continuation (second parameter) applied to the input stream (third parameter).

Two further continuation functions are used to specify the link between a pin and the constructor tag associated with that pin, one for input pins and one for output pins:

```
linkIn :: In i -> (i -> t) -> Process t u -> Process t u
linkOut :: Out o -> (o -> t) -> Process s t -> Process s t
```

The `linkIn` primitive provides the information used by the *merge and tag* operation of Fig. 3.4, `linkOut` provides the information for *untag and split*. Their use is best explained with an example. The definition of the `memory` Component is given in Fig. 3.6. A new value is received from pin `i` with a `MemoryIn` tag. A trigger message is received from pin `t` with a `MemoryTrigger` tag. Values are output from pin `o` with `MemoryOut` tag. Even though there is only one output we still need to tag the output message as the run-time system expects a tag.

```
data MemoryIn m = MemoryIn m | MemoryTrigger ()
data MemoryOut m = MemoryOut m

memory :: In m -> In t -> Out m -> m -> Process MemoryIn MemoryOut
memory i t o s =
    linkIn i MemoryIn $
    linkIn t MemoryTrigger $
    linkOut o MemoryOut
    memory' s where
    memory' s = rx $ \m -> case m of
        MemoryIn s' -> memory' s'
        MemoryTrigger () -> tx (MemoryOut s) $ memory' s
```

Figure 3.6: A Component Definition

The Components to which the `memory` is connected are not mentioned at any point in the definition. Only the pins are are identified. Although message tags are still used, they are defined and used within the Component definition, so we do not need to specify any tags when using the Component in a program.

The state of wires There is an implicit *state* value held in a Component that records which wires a Component is linked to. The `linkIn` and `linkOut` primitives alter this state. The actual value of this implicit state is contained in the run-time system, in the data structures that record information about Components, pins and wires. As the primitives are combined into a sequence by the use of CPS, a new value of the implicit state is passed from one continuation through to the next. Because the state cannot be duplicated and used in two different expressions at once, we can be sure that there will only be one version of the implicit state in existence at any given point. This allows the linking primitives to update the state value in-place, knowing that there will not be any references in the program to an older version.

Transmission of messages For each message in the output stream of a Component, the run-time system takes off the tag and uses it to identify which pin the message is to be emitted from. From this, the destination of the message can be found, by following the wire connected to the pin. This leads to a particular pin of another Component. The run-time system applies the tag associated with this pin to the message, and inserts it into the input stream of the Component, where input messages queue until the Component processes them.

Questions arising about system B *As there is a one-to-one correspondence between wires and tags, why can't they be the same thing?* The reason for making a distinction between them is our requirement that message passing is type-checked. Tagged messages entering a Component must be of one type, as they still enter through a single lazy list input. As the type of message each wire conveys may potentially be different, each wire-end attached to the pins of a Component is a different type. For example, a Component might be given an `In Char` and an `In Int` (where `In a` is the type of one end of the wire of type `Wire a`). If wires and tags were the same thing, then because the tags of a Component must all be of the same type, the wires would also have to be of the same type, and we would lose the benefit of wire connections being type-checked.

If the wires convey messages between Components then why aren't messages delivered to a Component function through the pin parameters (ie. the pin parameter is a lazy list of messages arriving at that pin)? Although this would seem the most intuitive way of delivering messages to a Component, we cannot do this because we want the Components to operate concurrently. We do not know which input pin of a Component the next message will arrive on. We can only guess which incoming message stream to evaluate next. If we choose a pin that turns out to have no waiting messages, the Component is unable to evaluate further until a message arrives. Any messages arriving on other pins cannot be received until after that has happened.

Why not provide a primitive function whose result indicates which wire to receive the next message from, then simply pass the wire's contents as a lazy list? Such a function would have to be applied to a value representing the current contents of the wires, or else it wouldn't be referentially transparent. Assuming this could be achieved, what value would the function return, to refer to the wire that contains the next message? It could not return the wire itself, as wires are of differing types.

Creating and wiring-up Components The pins of Components are connected together using wires. A Component obtains a wire using the function:

```
wire :: (Wire a -> Process i o) -> Process i o
```

A `spawn` function:

```
spawn :: Process a b -> Process i o -> Process i o
```

creates new Components dynamically. As an example, consider how to create two Components, `c1` and `c2`, wiring `c1` to `c2` and `c2` to `c1` (ie. each may send messages to the other). The Components have type signatures:

```
c1 :: In Char -> Out Int -> Process CompOneIn CompOneOut
c2 :: In Int -> Out Char -> Process CompTwoIn CompTwoOut
```

where `CompOneIn`, `CompOneOut`, `CompTwoIn` and `CompTwoOut` are datatypes containing constructors for the input and output pins of each Component. We begin each constructor tag with the name of the Component to avoid conflict with other datatypes. It is unfortunate that Gofer does not provide us with local datatype definitions for a function or module.

Fig. 3.7 shows the steps that create these Components, *wiring* them together. Functions `ip` and `op` return the ends of the given wire suitable for connection to an input or output pin, respectively.

```
wire $ \a ->
wire $ \b ->
spawn (c1 (ip b) (op a)) $
spawn (c2 (ip a) (op b))
```

Figure 3.7: Creating Components wired together

Although there are two tag-types for each Component, one for input tags and one for output tags, and types are defined at compile-time, this does not imply that only a single instance of a Component with these tags may exist. Multiple instances of a Component are differentiated by the Components they are connected to. There may be many instances of a particular Component in a single application, each connected to a unique set of other Components via its pins. What we cannot do is create new *types* of Component dynamically, because that would require new

tag types to be created during program execution. Anyhow, it is not obvious *how* new types of Component would be defined dynamically, or what use it would be.

Trying to connect one end of a wire to more than one pin causes a run-time error. Merging or broadcasting of messages is achieved through Components dedicated to the task.

3.3.1 Reappraisal — Improvements Over System A

Here we test system B for the shortcomings identified in system A. The same paragraph headings are used.

I. Re-use of processes The pins and wires of system B make it much simpler to re-use Components. In system A it is necessary to pass PIDs along with functions to pack or unpack a message, a tedious overhead for the programmer. A composition of processes requires packing and unpacking functions for messages passed privately within the composition. In system B, a Component requires only a wire-end to identify a communication link. The packing and unpacking of messages is handled by the run-time system.

II. Type-checking of messages With a different set of tags for each process pins are individually type-checked and messages are no longer lumped into one algebraic type. Even though types are static, Components can be dynamic because many instances of a particular Component may exist using the same type of tags. An instance of a Component is defined by what its pins are connected to, not by its tags. What *cannot* be created dynamically are new *types* of Components. This is similar to the fact that new types of functions cannot be created during the evaluation of a program, so is not seen as a shortcoming.

III. Composition of processes Compositions of Components can communicate messages privately without help from outside. For example, the `dualMemory` can be defined:

```
dualMemory :: (In m,In n) -> In t -> (Out m,Out n) -> (m,n) -> Process i o
dualMemory (i1,i2) t (o1,o2) (s1,s2) =
    wire $ \t1 ->
    wire $ \t2 ->
    spawn (tee t (op t1) (op t2)) $
    spawn (memory i1 (ip t1) o1 m) $
    spawn (memory i2 (ip t2) o2 n) $
    end
```

In this example, the wires `t1` and `t2` communicate the same type of message as passed into the composition, but they could have communicated any type. The composition does not require anything of the rest of the program, except to be connected to wires of the correct type.

IV. Declaring and maintaining PID and message tag types In system B, Components do not have PIDs. They are distinguished by what their pins are connected to. There are no PID types to maintain. Messages travel between Components without tags. Inside a Component definition tags are required, and tag types must be maintained. Each time a Component is changed to include a new pin, a new tag must be added to a tag type. This is a much more manageable task than maintaining a message tag type global to the program.

V. Dynamic processes Components can be created dynamically, in response to an event from the user for example.

VI. Scheduling The scheduling algorithm is the same for System B as for System C, and is described in detail in Chapter 4, *"Implementation"*. It avoids groups of processes taking over the program, leaving others starved. In addition, it tends to keep queues of messages as short as possible (because it picks the process with the most number of messages waiting), which keeps the amount of heap space used low.

3.3.2 Shortcomings

Insecurities of message passing Dynamic processes give more power to the programmer, but also create more opportunity for error. There are a number of places where the programmer must follow certain rules to avoid the risk of a message being undeliverable. These rules are common sense ("a message sent along a wire that has not been connected at the receiving end cannot be delivered"), but it

would be better if such mistakes were brought to light because they caused a static type-error.

VII. Messages sent along a wire unattached at the output end In system B, it is possible to obtain a wire but only connect one end of it to a process. A message sent along a wire that is not connected at the other end causes the program to halt with an error. The implementation could be altered so that such messages are *lost* rather than causing the program to stop. This would make programs unpredictable. The `emit` primitive could pass on a parameter to the continuation in the sending Component indicating success or failure. The error would still only be detected at run-time. As this type of error is caused by a mistake in the program, it ought to be detected at compile time.

This type of error can occur even when the program appears to connect Components to both ends of a wire. If the sending Component is created first, and transmits a message before the receiving Component has been created or has linked up to the wire, then the error will occur. This problem is avoided in most cases because the scheduler ensures that a sequence of `spawn`s in a composing process are evaluated one after another without the composer being suspended. So all the Components of a composition are created before any of them start sending messages. Also, newly created Components are scheduled at high priority until the point when they start communicating messages, so that any linking done during their initialisation phase happens in preference to message passing.

VIII. Messages arriving at pins that have not been linked with a constructor tag A process must link its wires with tags first, before attempting to receive or send any messages.

IX. Messages sent to a process that has terminated If a Component at the receiving end of a wire terminates and a message is subsequently sent along the wire, the results are unpredictable. This is because memory used to hold information about the process is deallocated when the process terminates, but the wire remains pointing to the area. At best, nothing will happen. At worst, the message transmission will have some undesired side-effect, as the message transmission primitive accesses the memory that used to contain process information.

X. A Component has direct access to its message streams It is important that the programmer does not side-step the primitive functions to send and receive messages, otherwise the message count will become invalid. There is nothing to stop the programmer doing this though. The message streams should be an abstract type that can only be accessed through the primitive functions.

XI. Two sets of tags needed for each sort of Component The disadvantage of having two sets of tags for each Component is that a lot of algebraic data types must be created, which leads to long constructor names to avoid clashes. If Gofer had modules we could hide elements of a Component's definition. However, the tag types form part of the type of a Component, so we cannot hide these without hiding the Component as well.

If two Components have the same type of pins then they may share input and output tags between them, but otherwise two sets of tags are required for every sort of Component. If two share the same set then this does not cause any ambiguity in deciding which of them will receive a particular message, as communication between Components is defined by wires connecting their pins, rather than by their message tags. It may be confusing for the programmer though.

There is a way of removing the tag types from a Component's definition. Compositions don't use any tags, as they don't do any tagging or untagging. They simply feed their wires into the Components of the composition. We can take advantage of this and make every direct definition of a Component into a composition of one process. Then the tag types don't appear in the type. For example, we can make the `memory` Component into a composition like this:

```
data MemoryIn m = MemoryIn m | MemoryTrigger ()
data MemoryOut m = MemoryOut m

memory :: In m -> In t -> Out m -> m -> Process i o
memory i t o s = spawn (mem i t o s) $ end
    where
    mem :: In m -> In t -> Out m -> m -> Process MemoryIn MemoryOut
    mem i t o s =
        linkIn i MemoryIn $
        linkIn t MemoryTrigger $
        linkOut o MemoryOut
        memory' s where
        memory' s = rx $ \m -> case m of
            MemoryIn s' -> memory' s'
            MemoryTrigger () -> tx (MemoryOut s) $ memory' s
```

The disadvantage of this method is that an extra process (the composer) is created and then terminated.

3.4 System C — Component Gofer

The need to link pins and tags in system B opens up many opportunities for run-time errors and makes the system difficult to comprehend. System C eliminates the need for tags altogether, increasing security and making the system simpler to understand and use.

The differences As in system B, Components are processes with pins that are connected together with wires. They differ in how messages get in and out of a Component. A Component is a *state transformer*, where the state incorporates a value that represents the state of the *world of wires*. Messages are held in the world state, queued up in the wires, instead of in the input stream of a process. Primitive functions that operate on the world state place messages and remove messages from wires. Whereas system B Components have messages from different pins fed in and out of the process function through tagged streams, system C Components are passed a series of *world* values containing wires that hold messages.

The similarities Scheduling is the same as for system B. Most of the types and primitives are retained with the same names as system B. The type of a Component looks very similar, but there are no tag types. For example, the type of the `memory` Component is:

```
memory :: In m -> In t -> Out m -> m -> Component
```

Types and primitives In system C, the type of a process is:

```
type Process s = s -> s
```

where `s` is an *explicit* state value. The world state is *implicit* in the definition of a process. If we made it explicit, the type of a process would be:

```
type Process s = (s,World) -> (s,World)
```

The reason for making it implicit is explained in a later section. For simple Components, the *explicit* state holds no useful value. It is just a place-holder to mark the place of the implicit world state:

```
data ComponentState = ComponentState
type Component = Process ComponentState
```

Later chapters describe more complex types of Components that keep useful values in the state.

As in system B, wires are obtained with a `wire` primitive, the wire-ends are selected with `ip` and `op` functions, and are passed in as pin parameters of Component functions. These wire-end values refer to wires in the world state. To transmit a message, a Component uses the `tx` primitive applied to an `Out m` wire-end and a message value of type `m`:

```
tx :: Out m -> m -> Process s -> Process s
```

As in system B we are using CPS to sequence operations, so each primitive expects a further parameter, the continuation.

To receive a message, a Component supplies to the `rx` primitive a separate continuation for each pin that a message might have been received on. The `rx` primitive takes the next message from the world state and applies the relevant continuation to it. The list of continuations supplied to `rx` have a different form from normal continuations because they can be tested to see if they apply to the next message. These *guarded* continuations have the type `Guarded s`, where `s` is the (explicit) state of the Component:

```
rx :: [[Guarded s]] -> Process s -> Process s
from :: In m -> (m -> Process s) -> [Guarded s]
```

A Component does not need to link pins with tags. However, a Component does need to `claim` any wire-ends it will receive messages from:

```
claim :: In m -> Process s -> Process s
```

The need for claiming is explained: if a Component tries to receive a message when none is available, it will be *suspended*. By claiming a wire-end, a Component tells the scheduler that it wants to be woken up (unsuspended) when a message is sent on that wire. In §3.7 we discuss an alternative implementation that removes the need for claiming.

If a second Component is to claim a particular wire, the first must disown it first:

```
disown :: In m -> Process s -> Process s
```

New wires and Components are created in the same way as in system B:

```
wire :: (Wire m -> Process s) -> Process s
spawn :: Process s' -> Process s -> Process s
```

Example Component We use the `memory` Component again:

```
memory :: In m -> In t -> Out m -> m -> Component
memory i t o s =
    claim i $
    claim t $
    m s where
    m s = rx [from i $ \s' -> m s',
              from t $ \_ -> tx o s $ m s]
```

This Component is typical of the form of many simple Components. It performs some initialisation first, claiming the inputs it will later receive from. It follows with a tail recursive loop, receiving a message and acting upon it. The loop holds an explicit state value, in this case the contents of the memory (`s`).

In a guarded continuation such as `from i $ \s' -> m s'` the part on the left of the `$` indicates the input pin the guard refers to. The part on the right takes the message received and returns the continuation to use in the event that this pin receives the next message. The type of the `from` primitive ensures that the type of message expected by the continuation matches the pin type. As `from` returns a value that is independent of the message and pin type, guarded continuations for pins of different types can be passed to `rx` in a normal list.

3.4.1 Reappraisal — Improvements over systems A and B

Issues relating to the re-use of processes, typechecking of messages, composition of processes, PID and message tag types, dynamic processes and scheduling (items I to VI) are the same for system C as for system B.

VII. Messages sent along a wire unattached at the output, or VIII. Messages arriving at pins that have not been linked In system C, neither of these situations cause a run-time error. Messages queue up in the wire until such time as a Component claims the wire and receives them. It does not matter if a sender starts transmitted messages before the receiver has had a chance to connect up to the wire.

IX. Messages sent to a process that has terminated Rather than cause a run-time error, messages sent on a wire that used to be connected to a Component that has since terminated will queue up and remain in the wire.

X. A Component has direct access to its message streams In system C, messages are held in wires that are hidden in the implicit world state. Access to them is only through the primitive functions, which need a wire-end value to pick a message out of a particular wire.

XI. Two sets of tags needed for each sort of Component No message tags are needed in system C, so this is not a problem.

3.4.2 Shortcomings

Despite all these improvements there are a few shortcomings of system C.

XII. `rx` needs a guard for every claimed pin The `rx` primitive must be supplied with a guarded continuation for every input wire a Component has claimed, otherwise a run-time error will occur when a message is sent on an unguarded pin and the `rx` is evaluated.

XIII. Receiving from wires not claimed If a guarded continuation for a wire that has not been claimed by a Component is passed to the `rx` primitive, it will cause a run-time error when evaluated. This is similar to the error of not linking a wire with a tag in system B. In system C we could improve matters by making a claimed wire a different type:

```
claim :: In m -> (Claimed m -> Process s) -> Process s
from :: Claimed m -> (m -> Process s) -> [Guarded s]
```

Then the `memory` would be defined:

```
memory :: In m -> In t -> Out m -> m -> Component
memory i t o s =
    claim i $ \ci ->
    claim t $ \ct ->
    m s where
    m s = rx [from ci $ \s' -> m s',
              from ct $ \_ -> tx o s $ m s]
```

It would still be possible for a Component to pass a claimed wire to another Component that tries to receive using it.

XIV. Claiming wires already claimed If a second Component tries to claim a wire that has already been claimed by another Component, a run-time error will occur.

3.5 Programming Techniques in system C

System C (also known as *Component Gofer* meets our requirements for programming GUIs sufficiently so in this section we present a few useful functions and methods to use when programming.

3.5.1 Derived functions

The `sequence` function:

```
sequence :: [Component -> Component] -> Component -> Component
```

performs a list of *actions*. An action is a function that when applied to a continuation `c` forms a process that performs some I/O operations, then continues with `c`. For example, the following expression emits the numbers 1 to 5 in five messages through pin o:

```
sequence [tx o m | m <- [1..5]]
```

The `accumulate` function is similar but operates on actions that *return* a result. By *return* we mean *pass on to their continuation*.

```
accumulate :: [(a -> Component)] -> ([a] -> Component) -> Component
```

The results from each action are collected and passed on in a list to the the rest of the program. For example, this expression evaluates three copies of `wire` collecting the three wires that result:

```
accumulate [wire, wire, wire] $ \ws ->
```

Collecting three wires into a list forces them to be of the same type. If three wires of different types are required then this is what is necessary:

```
wire $ \a ->
wire $ \b ->
wire $ \c ->
```

The `wires` function, found in the Component library:

```
wires :: Int -> ([Wire a] -> Component) -> Component
wires n = accumulate (copy n wire)
```

performs the same trick as described in the `accumulate` example, but for a given number of wires. As before, all the wires will be of the same type.

The `duplex` function supplies a pair of wires arranged back-to-back (ie. one part of the pair is the input of the first wire with the output of the second wire, and vice-versa):

```
duplex :: (Duplex a b -> Component) -> Component
```

where the type of a duplex wire is:

```
type Duplex a b = ((In a, Out b), (In b, Out a))
```

3.5.2 Components with Lists of Pins

A Component may have a list of pins of the same type, such that the number of pins in the list is unknown at the time the Component is defined. For example, the `merger` Component has a list of input pins and a single output pin. A message received on any of its inputs is emitted from its output:

```
merger :: [In m] -> Out m -> Component
merger is o =
    sequence (map claim is) $
    m where
    m = rx [ from i $ \v -> tx o v $ m | i <- is ]
```

Sometimes we would like to be able to identify which of a list of input pins a message is received from. In this situation, the `froms` function (defined in terms of `from`) is useful:

```
froms :: [(In m,b)] -> (m -> b -> Process s) -> [Guarded s]
froms p c = [from i $ \m -> c m d | (i,d) <- p]
```

It is applied to a list of pairs where each pair consists of an input pin and a value to identify the pin by. When a message is received from one of the pins, both the message and the *identifying value* is given to the continuation: For example, a Component `tag` has a list of input pins and a single output pin. Each input pin is associated with a number and when a message is received, the message is sent out of the component paired with the number of the pin through which it was received:

```
tag :: [In m] -> Out (m,Int) -> Component
tag is o =
    sequence [claim i | i <- is] $
    t where
    t = rx [ froms (zip is [1..]) $ \m n -> tx o (m,n) $ t ]
```

The `froms` function is most useful in programming *servers*. A list of clients can be connected to the server through a list of duplex wires. Each duplex wire consists of an input pin (from the client) paired with an output pin (to the client). If we apply `froms` to the duplex wire, then when a message is received it is given to the continuation together with the output pin through which to return a result message to the client.

3.5.3 Creating New Pins on a Component as it Operates

A Component isn't limited to just the pins it is passed when it is created. More wire ends can be passed to it in messages. For example, a `chat` Component is created with a single duplex connection to a `talker` Component. The `chat` Component has the potential to connect itself to more `talker` Components. A `talker` either sends strings to `chat`, which copies them to all `talker`s connected, or an existing `talker` introduces a new `talker` by sending a duplex connection to the new `talker` to `chat`. A `talker` sends `Talk` messages to `chat` and receives plain `String`s back:

```
data Talk = Say String | NewTalker (InOut Talk String)
```

The `chat` Component is defined:

```
chat :: InOut Talk String -> Component
chat t = chat' [t] where
    chat' ts =
        rx [ froms ts $ \m _ -> case m of
            Say s -> sequence [tx o s | o <- map snd ts] $ chat' ts
            NewTalker t -> chat' (t:ts) ]
```

It keeps the duplex connections in a list, and uses `froms` to receive messages from any of the `talkers`. The `froms` function results in the message received and the output part of the duplex connection, but `chat` ignores the latter and sends the string to all `talkers`.

An example of a server Component can be found in §6.5.

3.6 Related Work

Henderson [Hen82b] uses recursive stream-processing functions connected by networks of lazy streams to program operating systems. A special *merging* function (not a pure function, but it merges messages in chronological order) allows processes to receive messages from multiple streams non-deterministically. To identify the source of messages they are *tagged*. An *untagging* function picks messages with a certain tag out of a stream. By a combination of tagging and untagging the communication paths between processes are defined. The resulting programs are spagetti-like in nature. Karlsson [Kar81] and Stoye [Sto85a] improve the situation by employing a *sorting office* that all processes are connected to. The sorting office forwards messages from one named process to another. Type checking of messages is lost because all processes must communicate messages of the same type. Turner [Tur87] develops the idea further by enforcing the rule that values are turned into messages using *wrapper* functions supplied by the receiving process, thus re-introducing type-checking of messages.

Holyer and Carter [HC93] describe a concurrent process system that is unique in that it does not introduce non-determinism to programs. Rather than merging input from separate sources non-deterministically, a new process constitutes a separate demand for lazy evaluation. The purely deterministic system requires the structure of some programs to be re-thought, and examples are given. A simple interface to the Xlib library has been implemented in this system [Ser95].

Jones and Hudak [JH93] describe a concurrent process implementation with the purpose of expressing parallelism. Concurrent evaluation of two processes is initiated with a `fork` primitive. Evaluation of `fork` yields a pair of the results from the two forked evaluations. Communication between processes is through typed message channels. A process receives messages from multiple channels by forking a process for each channel.

Peyton Jones, Gordon and Finne [PJGF96] describe a concurrent process exten-

sion to Haskell. Processes are defined in the monadic style. Type-checked communication between processes and process synchronisation is achieved through shared atomically-mutable state. Processes are created dynamically by a `forkIO` primitive. This language is used as the basis of a GUI system A process receives messages from multiple sources by creating a process to input from each source.

3.7 Possible Improvements

System C meets our initial aims sufficiently to support the rest of the work in this thesis. If more time were to be spent on the concurrent process implementation we would like to see the following additional features of the language:

- to be able to `rx` from a subset of the input wires claimed by a process. This would enable a Component to ignore some wires until a certain message had been received. For example, we could define a modified version of `memory` that is not given an initial value, but instead ignores the trigger input until a value is received. The scheduling algorithm of the present system assumes Components are ready to receive from any of their claimed wires, so this would need to be changed to accommodate this feature.

- to remove the need for the `claim` primitive. In the present implementation the need to claim wires is related to scheduling. It may be the case that the changes required to implement the previous feature would also remove the need for `claim`.

- enabling many Components to receive from a single wire. In system C, multiple Components can transmit on a single wire, but only one can claim and receive from a wire. If multiple Components each need to receive a copy of messages sent on a particular wire, it must be copied explicitly by a `broadcast` Component, which requires multiple messages to be transmitted.

3.8 Summary

In this chapter we argued for adding concurrent processes to a functional language, to allow GUIs to be expressed more naturally. Through three systems we developed an extended language that suits our requirements, as specified at the end of Chapter 2.

Some programming techniques for our final system and some possible improvements are discussed.

Chapter 4

Implementation

4.1 Introduction

In this chapter we describe the implementation of concurrent functional processes that we use to test the ideas of this thesis. It consists of a modification to the Gofer functional language interpreter [Jon93], a program written in the imperative language C, along with supporting declarations of process types and primitive functions in the functional prelude.

A process is a functional evaluation. The function is a *state-transformer*. It takes a *World state* argument whose value represents a history of all communications made between processes. The function returns an altered World state, the changes reflecting any communications that have taken place.

The World is an abstract type that can only be accessed by some primitive I/O functions. This limits the I/O actions that can be performed to creating processes and wires, and transmitting and receiving messages along those wires.

Special primitive I/O device processes provide links between the World of processes and the real world in which the keyboard, screen and mouse exist.

The implementation of the Gofer interpreter is described in [Jon94]. We have modified the interpreter so that when a `launch` primitive is applied to a process function and evaluated, the process is created. Primitive functions within a process definition are used to `spawn` a new process, create a `wire`, `claim` a wire for receiving messages, `disown` a wire, `tx` (transmit) a message, and `rx` (receive) a message `from` a particular wire.

The heap is shared between all processes. We can do this without risk of conten-

tion because a context switch between processes can happen only at certain points, when we are certain that the heap is in a stable state.

Conceptually, each process has its own stack that is used during evaluation to hold intermediate values. In fact, it turns out that the stack pointer returns to the same place at every change of context, so we can safely have a single stack for all processes. This saves extra overheads in terms of stack memory for each process and the time taken to switch stacks.

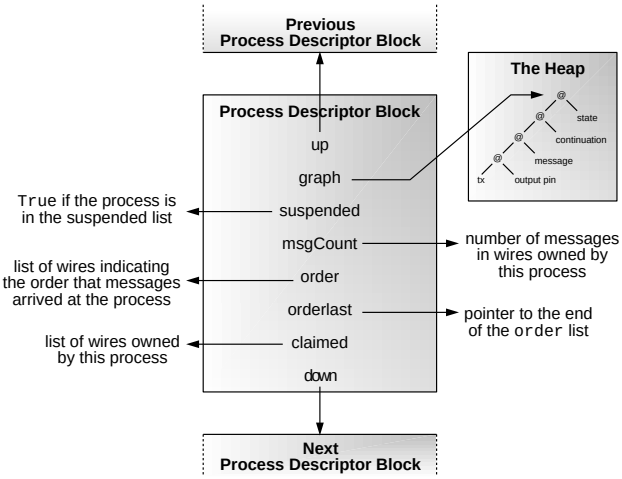
We begin by describing the data structures used to implement processes and the way they integrate into the interpreter. Then we look into the implementation of each primitive and see how processes, scheduling and message-passing are implemented in terms of data structures and algorithms. In the final section we discuss some weaknesses of the implementation and suggest lines of future work.

4.2 Data Structures

A block of memory holds information about each process. These blocks are linked into a list.

```
record ProcBlock
  graph     :: Cell
  suspended :: Bool
  msgCount  :: Int
  order     :: ptr to WireList
  orderlast :: ptr to WireList
  claimed   :: ptr to WireList
  up        :: ptr to ProcBlock
  down      :: ptr to ProcBlock

ProcBlock this;
```



- `graph` points to the state-transformer function in the heap that defines the operation of the process.
- `suspended` is a boolean indicating whether the process is currently suspended.
- `msgCount` is an integer count of the number of messages waiting to be received by this process, ie. the sum of the number of messages in all the wires claimed by this process.

- `order` is a pointer to the head of a wire list that indicates the order that messages have been sent to the process. The number of wires in the list equals the `msgCount`. When this process next receives a message with `rx`, it will be from the wire referred to at the head of the `order` list.

- `orderlast` is a reference to the last element of the `order` list. When a message is sent on a wire owned by this process, a reference to the wire is added to the end of the `order` list using this pointer.

- `claimed` is a list of wires claimed by this process for receiving messages from. The order of this list is not important, so every time a wire is `claim`ed it is added to the front of this list.

- `up` is a pointer to the next process block up the list. In the runnable queue `up` points to a process with a greater or equal `msgCount` than this one. In the initialising queue `up` points to a process nearer the top of the queue. The suspended queue is unordered.

- `down` is a pointer to the next process block down the list. In the runnable queue `down` points to a process with a lesser or equal `msgCount` than this one. In the initialising queue `down` points to a process nearer the end of the queue.

A global variable `this` points to the process block of the currently executing process.

Queues Processes are held in one of three queues, *initialising*, *runnable* or *suspended* (Fig. 4.1). The queues are linked lists of process blocks, and pointers to the first and last process in each queue are stored so that new processes can be added to the bottom or taken from the top directly. The runnable queue is kept in order of message count, highest first. The initialising queue is arranged into first-in first-out (FIFO) order. The suspended queue is not kept in any particular order. A particular process is removed from the suspended queue when there is a message for it to receive.

Wires The wire lists referred to in the process block are linked lists of `WireID`s that index a global array of *wire blocks* (Fig. 4.2). A wire can either be a hardware wire or a software wire. If it is a hardware wire then sending a message on it results

```
record Queue
  first :: ptr to ProcBlock
  last  :: ptr to ProcBlock
  type  :: QueueType

enumumerated QueueType {Initialising, Runnable, Suspended}
Queue initQ, runnableQ, suspendedQ;
initQ.type := Initialising;
runnableQ.type := Runnable;
suspendedQ.type := Suspended;
```

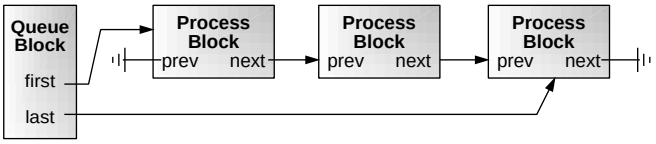


Figure 4.1: The Queue Block and Three Queues.

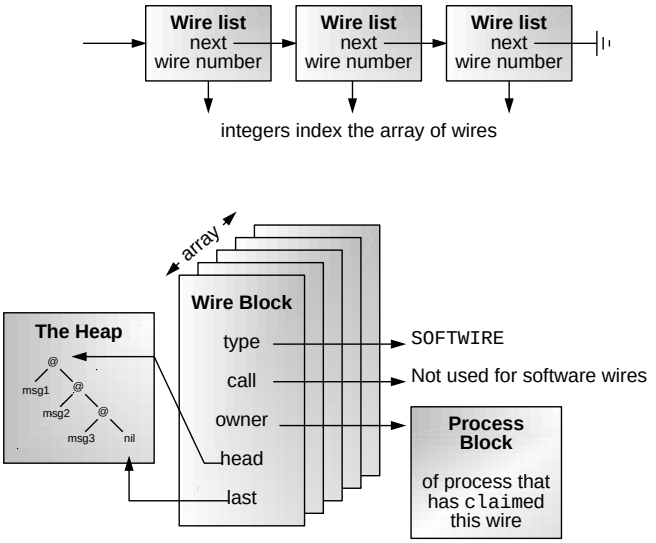
in a C function being called with the message as parameter. If it is a software wire then sending a message on it increases the `msgCount` of the process that owns the wire, and *awakens* it if it was suspended.

```
typedef WireID int

record WireList
  number :: WireID
  next   :: ptr to WireList

record WireBlock
  type :: WireType
  call :: ptr to C function
  owner :: ptr to ProcBlock
  head :: Cell
  last :: Cell

WireBlock wires[MAXWIRES];
```



The wire block illustrated is a software wire.

A hardware wire is illustrated later in Fig. 4.3.

Figure 4.2: The Wire Block, Wire List and Array of Wires.

The `WireId` of the next wire to be allocated is kept in a global variable. It is initialised to 0 and incremented each time a wire is created by the program:

```
nextWire :: WireID
```

Global variables store the `WireId`s of wires to which device outputs are connected. These variables are set when the relevant device processes are created. When

user input becomes available (eg. the user clicks the mouse) messages indicating this are inserted into these wires:

```
mouseInput    :: WireID
screenInput   :: WireID
keyboardInput :: WireID
```

4.2.1 Operations on Data Structures

C functions are defined for placing a process in a queue (`addProcessToQueue`), removing a process from a queue (`removeProcessFromQueue`), moving a process from one queue to another (`moveProcess`), creating a new process in a queue (`addNewProcess`) and checking that a process is in the correct place in a queue (`checkPositionInQueue`).

4.3 Functional Types

Wires used to connect Component pins have two ends: the `In m` end carries messages of type `m` into a Component, the `Out m` end carries messages of type `m` out of a Component. A `Wire m` consists of a pair of the ends:

```
data In m = In WireID
data Out m = Out WireID
type WireID = Int
type Wire m = (In m, Out m)
```

A process is described by a *state-transformer* function:

```
type Process s = s -> s
```

The state is used to hold hidden wires in certain types of Components. This is discussed more in Chapter 5, *"Screen Management"*.

The World State There is also another *implicit* piece of state associated with a process. This is the state of *The World*, a value representing the history of I/O interactions. Primitive I/O functions alter this state to *perform* I/O. We could make the world state explicit:

```
type Process s = (World,s) -> (World,s)
```

In other systems using a similar *World* state [JP93, FJ95, PvE95], the World is an abstract type on which the only operations are the I/O functions provided in the library. As the Gofer interpreter does not have a module system with which we could make the World type abstract, we have chosen instead to make the World state *implicit* in the definition of a process, and restrict access to the World in this way.

The World state contains information relating to the messages contained in wires, including the order in which messages are sent, the set of processes in the program, and a supply of wires for connecting pins. The value of the World state is actually held in the process blocks, wire blocks and global variables described previously. Because we take care to use the World state single-threadedly there is only ever one World state value in existence at any one time, so we can update these blocks and variables in-place without the risk of a reference in the program pointing to an older version of the state.

We could have used Gofer's *restricted type synonyms* to ensure that only the supplied I/O functions attempt to access parts of the World state. However, as all of I/O functions that access the World state are primitive functions, we have no need to specify the structure of the World state in the functional domain, so we have chosen to make the World state implicit instead.

4.4 Primitive Functions

A program consists of an expression to be evaluated in the context of a set of function definitions. The program expression is stored as a graph structure in the heap. The program is evaluated by transforming the graph by a sequence of *reductions*, until a form is reached where no further reductions can take place. The graph now contains the result of the program. Each reduction is defined by one of the functions in the program, in the prelude, or by a primitive function. The reductions performed by a primitive function are defined in C. We use primitive functions to gain access to the data structures defined in the previous sections. When a primitive function is called it is passed pointers to the cells in the heap that hold the parameters to the function. The function returns a result by overwriting the function application. This is achieved using the `updateRoot` function. The result may be reducible further, for example the `tx` primitive results in a continuation. In this case the evaluator will evaluate the result further as necessary.

Evaluation of a primitive may cause a process to become suspended. In this case, the graph is not updated, and control returns to the scheduler (`scheduler()`) which chooses another process to evaluate. The current position in the graph is saved in the `graph` part of the `ProcBlock`, and when the suspended process becomes runnable again, evaluation is restarted at the point it left off.

4.4.1 Process-related primitives

Program evaluation primitive When the launch primitive is applied to a process and evaluated, it initialises the three queues to be empty, adds the specified process to the initialising queue and hands over control to the scheduler. Evaluation of `launch` results in the unit value (`()`) when all processes have terminated.

```
launch :: Process s -> s -> ()
launch(process, state) =
    initQ := runnableQ := suspendedQ := empty;
    addNewProcess(process, state, initQ);
    nextWire := 0;
    scheduler();
```

Process creation primitive The `spawn` primitive allows a process to add a new process to the initialising queue. It achieves this in much the same way as `launch`, and carries on with its continuation. `ap(a,b)` allocates a new cell in the heap that is an application of `a` to `b`. An application of a function to more than one parameter consists of nested partial applications. For example, `f a b c` is represented by `((f a) b) c` in the heap.

```
spawn :: Process s' -> s' -> Process s -> Process s
spawn(process', state', continuation, state) =
    addNewProcess(process',state',initQ);
    updateRoot(ap(continuation,state));
```

Process termination primitive The `end` primitive is used to signal the termination of a process. It returns the final state instead of a continuation, so it cannot be evaluated any further. The evaluation, which was initiated by the scheduler, ends and control returns to the scheduler. The scheduler removes the process from the runnable queue and the memory taken by its `ProcBlock` is released.

```
end :: Process s
end(state) =
    updateRoot(state);
```

4.4.2 Wire-related primitives

Wire creation primitive A process obtains new wires with the `wire` primitive. The next block in the array of wires is set to a software wire, and the result of this primitive is the continuation applied to this wire.

```
wire :: (Wire m -> Process s) -> Process s
wire(Cell continuation, Cell state) =
    wires[nextWire].type := SoftWire;
    wires[nextWire].head := wires[nextWire].last := [];
    nextWire := nextWire + 1;
    updateRoot(ap(ap(continuation,(In nextWire, Out nextWire)),state));
```

Wire claiming and disowning primitives To `claim` a wire, the `owner` field of the `WireBlock` is set to point to the process evaluating `claim` and the wire is added to the list of `claimed` wires in the `ProcBlock`. For every message stored in the wire, the process' message count is incremented and one copy of the wire's ID is added to the end of the process' message order list. This is so that the process can receive any messages that were already held in the wire when it `claim`ed it. The result returned by the primitive is the continuation.

Claim run-time error Only one process can claim a wire. If a process tries to claim a wire already claimed by another process, a run-time error occurs.

```
claim :: In m -> Process s -> Process s
claim(Cell wire, Cell continuation, Cell state) =
    if (wires[wire].owner != NULL)
        error "Wire already owned by another process"
    else
        add wire to this.claimed;
        for each message held in wire
            increment the msgCount and
            add the WireID to the end of the process' order list
        checkPositionInQueue(this,runnableQ);
        updateRoot(ap(continuation,state));
```

Similarly, to `disown` a wire, the owner part of the wire block is set to *empty*, and every occurrence of the `WireID` in the process' message `order` list is removed, decreasing the message count by one for each one found. Any messages left in the wire can be received by a Component that subsequently `claim`s the wire.

Disown run-time error If a process tries to disown a wire it has not claimed, a run-time error occurs.

```
disown :: In m -> Process s -> Process s
disown(Cell wire, Cell continuation, Cell state) =
    if (wires[wire].owner != this)
        error "Wire not owned by this process"
    else
        remove wire from this.claimed;
        for each occurrence of wire in proc.order
            proc.msgCount := proc.msgCount - 1;
            remove wire from proc.order list;
        wires[wire].owner := NULL;
        checkPositionInQueue(this,runnableQ);
        updateRoot(ap(continuation,state));
```

4.4.3 Communication-related primitives

Message transmission primitive If a process tries to `tx` a message when it is initialising, the process is moved to the runnable queue, and the scheduler is called. The reason for this is explained in §4.5. If the process was already runnable, the wire block is consulted to see if the wire is a hardware or software wire. If it is a hardware wire, the stored C function is called with the message as a parameter (see §4.4.4). Otherwise, the message is added to the end of the list of messages held in the wire. If the wire is owned by a process, the message count of the process is incremented, and the `WireID` of the wire is added to the end of the process' message order list. If the receiving process is suspended, it is moved to the runnable queue. By adjusting the message count of the receiving process, the runnable queue may become out of priority order, so the process is moved up if necessary. The scheduler is called, as the receiving process may, having just received a message, have a higher priority than the one currently evaluating.

Queue manipulation The sending process is moved down the runnable queue below any other processes with the same message count. This action achieves round-robin scheduling for processes with the same number of messages waiting. Round-robin scheduling is important for generator processes that only transmit messages and never receive any. Without round-robin or a similar strategy, one process could deny others from being evaluated.

```
tx :: Out o -> o -> Process s -> Process s
tx(Cell output, Cell msg, Cell continuation, Cell state) =
    if (this process is in initQ)
        moveProcess(process,initQ,runnableQ);
        this.graph := root;
        scheduler();
    else
        if (wires[output].type == HardWire)
            (wires[output].call)(msg);
        else
            add msg onto end of wires[output] message list;
            receiver := wires[output].owner;
            if (receiver != empty)
                (add output to receiver.orderlast);
                if (receiver.suspended)
                    moveProcess(receiver,suspendedQ,runnableQ);
                checkPositionInQueue(receiver,runnableQ);
        while(this.down.msgCount == this.msgCount)
            moveProcessDown(this,runnableQ);
        updateRoot(ap(continuation,state));
        this.graph := root;
        scheduler();
```

Message reception primitives Message reception is achieved through the use of the `rx` primitive applied to a list of guarded continuations (*guards* for short). A guard is generated by an application of the `from` primitive to an input pin and a continuation that should be applied to a message from that pin. The type of `from` is:

```
from :: In m -> (m -> Process s) -> Guarded s
```

The `Guarded s` type is like a process, but it returns a value of type `Maybe s` instead of `s`:

```
type Guarded s = s -> Maybe s
```

We explain the operation of a guard: suppose we are evaluating `from` applied to input pin `i :: In m`, continuation `c :: m -> Process s` and state `s`, where the process is of type `Process s`. If the next message[1] to be received by the process, `m`, matches pin `i` then it is removed from the world resulting in a new world state `s'`, and `Yes (c m s')` is the result of the guard. If the next message does not match pin `i`, then `None` is the result. This allows the `rx` primitive to try each guard in turn until the matching one is found.

[1] by *next message* we mean the message sent to this process longest ago

Unclaimed wire run-time error A run-time error occurs if a guard is evaluated for which the associated wire has not been claimed by the process. This is discussed further in §4.6.2.

```
from :: In m -> (m -> Process s) -> Guarded s
from (Cell input, Cell continuation, Cell state) =
    if (wires[input].owner != this process)
        error "tried to receive from a wire not owned by this process";
    else
        if (input != this.order.number)
            updateRoot(None);
        else
            msg := first message in wires[input] removed;
            updateRoot(ap(ap(ap(Yes,continuation),msg),s));
```

If a process tries to `rx` when it is initialising and there is a message for it to receive then it is moved to the runnable queue and the scheduler is called. If there is no message for it to receive (initialising or runnable) then it is moved to the suspended queue. These actions are explained in §4.5. Otherwise, each guarded continuation is tried in turn until one is found that matches the pin on which the next message is to be received.

```
rx :: [Guarded s] -> Process s -> Process s
rx (Cell guards, Cell state) =
    if (this process is in initQ)
        if (process.msgCount == 0)
            moveProcess(this,initQ,suspendedQ);
        else
            moveProcess(this,initQ,runnableQ);
            checkPositionInQueue(this,runnableQ);
        this.graph := root;
        scheduler();
    else
        if (this.msgCount == 0)
            moveProcess(this,runnableQ,suspendedQ);
            this.graph := root;
            scheduler();
        evaluate guards until one returns Yes s;
            updateRoot(s);
            this.msgCount := this.msgCount - 1;
            checkPositionInQueue(this,runnableQ);
        or else if all of the guards return None then
            error "No guarded continuation for next message";
```

Missing guard run-time error If no matching guarded continuation is found then a run-time error occurs because messages must be received in the order in which they were sent. It is tempting to try and eliminate this run-time error by allowing messages to be received *out* of order. This is discussed further in §4.6.2.

Efficiency A quick test reveals that evaluating each guard requires at least seven reductions (more if there is abstraction on top of the `from` primitive, eg. using `froms`). For Components with a large number of inputs (eg. servers like the Screen Manager) that input messages repeatedly, these evaluations could become significant. An alternative implementation might create an *array* of guards in order of `WireId` on the first evaluation of an `rx` application, and in subsequent evaluations obtain the correct guard immediatedly from the array. As the system stands, we can help to minimise the number of reductions by putting the most frequently used guards first in the list given to `rx`.

4.4.4 Primitive I/O device Components

The definitions of the I/O device Components are primitives. When `spawn`ed, a primitive Component records the `WireID` of the wire attached to its output pin in a global variable. It also converts the wire attached to its input pin into a hardware wire, setting the stored function to a C function that performs the desired output action. For example, the wire attached to the output pin of the `screen` Component is stored in the global variable `screenInput`. When input arrives from the screen display (eg. a window exposure event) this variable indicates which wire to send a message along (see Fig. 4.5). The wire attached to the input pin is converted to a hardware wire and the function `screenCmnd` (that responds to messages such as `DrawLine` and `DrawSetColour`) is stored the wire block (see Fig. 4.3 and Fig. 4.4).

```
screen :: In [ScreenCmnd] -> Out ScreenEvnt -> Process s
screen (Cell in, Cell out, Cell state) =
    screenInput := out;
    wires[in].type := HardWire;
    wires[in].call := screenCmnd();
    updateRoot(state);
```

Alternatives We could have given access to the hardware through the World state. Primitive functions would *perform* I/O by returning an altered World state.

We chose not to do this because it would be difficult to control the sharing of resources — any Component can access the World state. Alternatively we could access the hardware through special wires. We chose to use special Components instead because it follows the pattern of programs in general. Primitive I/O device Components are wrapped up in compositions to create more complex I/O devices. These in turn are composed to form more complex devices.

Multiple devices run-time error Despite our claim that using Components for I/O allows us to control sharing of a resource, it is possible for a program to try to create multiple instances of a particular I/O device. This causes a run-time error. This is discussed further in §4.6.2.

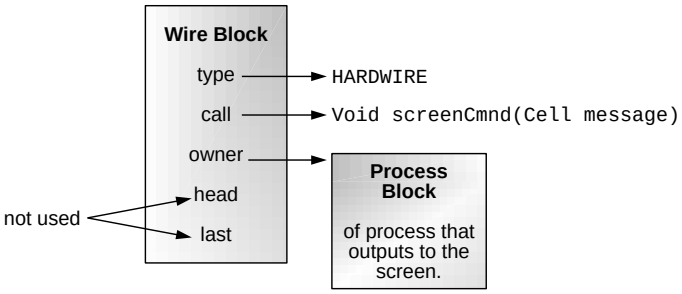


Figure 4.3: A Hardware Output Wire Connected to the `screen` Component.

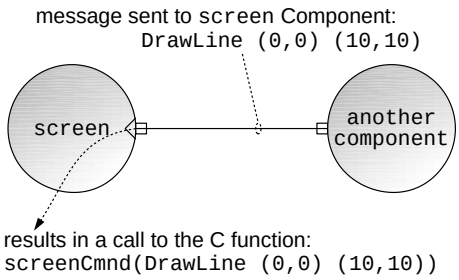


Figure 4.4: A Component Sends a Message to a Primitive Output Component.

4.5 Scheduling

Scheduling options There are a number of options in choosing a scheduling algorithm. Is the scheduling to be pre-emptive? Can the scheduler change context

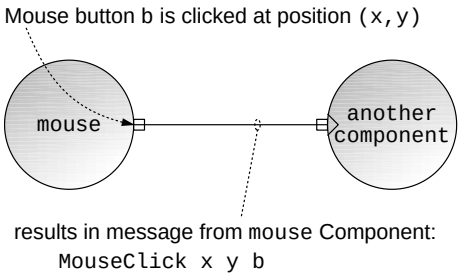


Figure 4.5: A Primitive Input Component Sends a Message to a Component.

to another process at any point? Or must a process explicitly give control back to the scheduler at regular intervals? How should the scheduler choose a process? Do processes have a priority? If so, is the priority stated in each process definition, or calculated by some other means?

As the scheduling decisions are not central to the work of this thesis, we have chosen a scheduling algorithm that is simple to implement. We do need pre-emptive scheduling to program GUIs. However, we do not want to have to mark the points at which a change of context can occur explicitly, because that would reveal the fact that processes are sequentially evaluated in process definitions. Therefore, we chose a compromise in which a change of context can occur at the point of receiving or transmitting a message. Control is implicitly returned to the scheduler at these points.

We want to avoid giving processes an explicit priority, again because it reveals the fact that processes do not really operate concurrently but share a single processor. On the other hand, the simple scheduling algorithm of earlier systems tends to result in some processes not evaluating often enough to consume the messages they are sent. Therefore, we chose to prioritise processes on the number of messages waiting for each to receive. This helps keep the number of messages queuing in wires low, reducing the amount of heap space used.

Scheduling algorithm The scheduler is called for the first time by the `launch` primitive. Control returns to the scheduler whenever a process transmits a message or is suspended. In general, the process with the most messages waiting will be the next to be evaluated. Maintaining a message count for each process depends on processes claiming the wires they wish to receive from. When a message is sent on a wire, a check is made to see which process owns the wire. If any process has

claimed the wire, the message count of that process is incremented. This may raise the priority of the receiving process above that of the process ahead of it in the queue, so the queue is reordered. If the receiving process is suspended, it is made runnable. If we did not insist that processes claim wires they wish to receive from we would not know when to wake up a suspended process.

Initialising processes have priority over runnable processes to ensure that processes have a chance to `claim` any wires they wish to receive from. The message counter for a particular process only includes messages in wires claimed by the process. If there were no initialisation queue, and new processes were inserted straight into the runnable queue on creation, their priority would be low compared to existing runnable processes. They might never get to claim their wires if other processes are constantly busy. Once a new process has claimed its wires in the initialisation queue it is ready to move into the runnable queue where its priority will accurately reflect the number of messages waiting for it. The diagram in Fig. 4.6 depicts a process as a finite state machine that has states Initialising, Runnable and Suspended. In Fig. 4.7 the transitions from state to state are further explained.

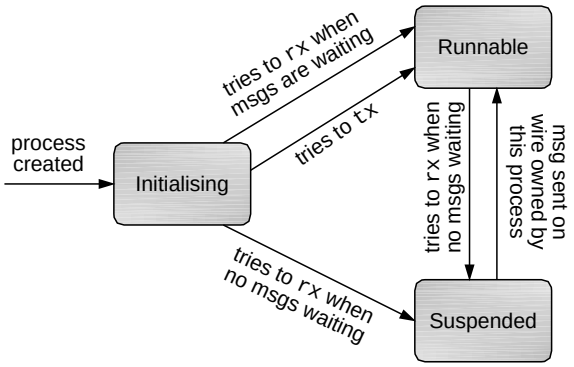
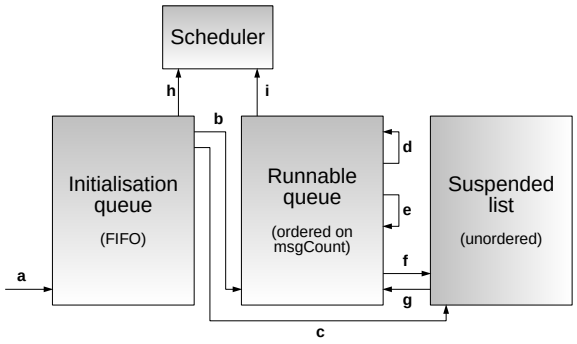


Figure 4.6: A Process as a Finite State Machine.

As the runnable queue is kept in message-count order we avoid having to search a list for the runnable Component with highest count. The process at the top of the runnable queue has the highest priority.

The `scheduler` polls for pending input from the user, then looks for a process to evaluate with `switchContext` (Fig. 4.8). All processes may be suspended, in which case it polls for input again. We cannot assume that if all processes are suspended then the program has deadlocked because some user-input may awaken a suspended process. If there was an initialising process to evaluate, or failing that a runnable



Letters in the following list refer to the arrows in diagram above.

- New Components are placed onto the end of an *initialisation queue* (a). Here they are chosen in preference to runnable Components so that they may claim the wires they wish to receive from.
- If an initialising Component attempts to send a message it is moved to the back of the *runnable queue* (b). The message is not sent until the Component is chosen for execution from the runnable queue.
- If an initialising Component tries to receive a message and there is one waiting it is moved to the *runnable* queue (b).
- If an initialising Component attempts to receive a message when there are none waiting it is moved to the *suspended list* (c).
- If a Component in the runnable queue receives a message that causes it to have more messages than the Component above it in the queue, then the Component moves foward in the queue (d). If a Component in the runnable queue receives a message, and as a result has less messages left to receive than the Component below it in the queue, then the Component moves backward in the queue (e).
- A message delivered to a Component on the suspended list moves the Component to the back of the runnable queue (g).
- Whenever a change of context is required, the scheduler picks the first Component in the initialisation queue (h), or if the initialisation queue is empty then the scheduler picks the first Component in the runnable queue (i). If the runnable queue is also empty then the scheduler continues polling until the arrival of some user input moves a process from suspended to runnable.

Figure 4.7: The Scheduling Scheme.

process, the `graph` stored for that process is evaluated. This evaluation will end either when the process terminates, or when an evaluation of an `rx` or `tx` primitive in the process evaluation makes a call to the scheduler.

The `pollEvents` function (Fig. 4.8) checks for input from the user. If there is any, a message is generated from it, and fed into the wire connected to the relevant I/O device. The `WireIds` of wires connected to I/O devices are stored in global variables `screenInput`, `mouseInput` and `keyboardInput` when the I/O devices are `spawn`ed in the functional program.

```
scheduler() =
    while (at least one process left in any queue)
        pollEvents();
        this := switchContext();
        if (this != NULL)
            evaluate(this.graph);
            removeProcessFromQueue(this,runnableQ);

pollEvents() =
    msg :: Cell
    wire :: WireID
    receiver :: ProcBlock

    e := getEventFromHardware();
    case e of
        None -> return();
        Exposure ->      msg := Expose (e.rectangle);
                         wire := screenInput;
        ButtonPress ->   msg := MouseClick e.x e.y e.button;
                         wire := mouseInput;
        ButtonRelease -> msg := MouseUnClick e.x e.y e.button;
                         wire := mouseInput;
        KeyPress ->      msg := KeyPress e.char;
                         wire := keyboardInput;
    (perform 'tx wire msg' - as per tx primitive);

switchContext() =
    if (there is at least one process in initQ)
        return(first process in initQ);
    else if (there is at least one process in runnableQ)
        return(first process in runnableQ);
    else
        return(NULL);
```

Figure 4.8: Scheduling functions.

Consequences of this scheduling algorithm Because a change of context can only occur at the point of receiving or transmitting a message, a process will take over the processor if it has to evaluate an expression before receiving or transmitting a message. An expression is not evaluated before being sent in a message so delays due to evaluation of a complicated expression do not always occur where expected. For example, when a Gadget sends a new drawing function to the Screen Manager (see Chapter 5, *"Screen Management"*) it is not the Gadget that evaluates it, nor is it the Screen Manager. It is the `screen` I/O device Component that requires the values in the list returned from the drawing function.

4.6 Shortcomings of this Implementation

4.6.1 `Wire` should be an abstract type

Currently it is possible to abuse an arbitrary integer value as a wire instead of using the `wire` primitive to generate one. It would be better if the `Wire`, `In` and `Out` type were abstract. Without a module system we have no means to make it abstract. We cannot use the same trick as we used with the `World` type as there is usually more than one wire in a program and we need the `WireID` value to distinguish them.

4.6.2 Run-time errors

A large part of the correctness of a Component program is checked statically. Wires can only be connected to pins of the same type. Messages can only be transmitted on wires of the correct type. Guarded continuations must expect messages of the same type as the pin they are associated with. There remain a few instances where an error will not be flagged until run-time:

An I/O device Component multiply spawned (see §4.4.4)

If Gofer had a module system we could hide the declaration of each primitive I/O device in the module that contains its manager Component.

Missing `from` in `rx` (see §4.4.3)

The list of guarded continuations that the function `rx` is applied to must have a `from` for every wire the Component owns. If not, and the next message to be received

is destined for an unguarded pin, a run-time error will occur when the Component tries to `rx`. If the pin is seldom used the error could go un-noticed for a long time.

Receiving messages out of order A run-time error occurring in the `rx` primitive could be avoided by allowing messages to be received in a different order from that sent. When the `rx` primitive does not find a guarded continuation for the next message, the associated wire is "suspended" until such time as an `rx` primitive with a guard for that wire is evaluated. Messages transmitted on a suspended wire do not increase the receiving process' `msgCount` and do not *wake* it if it is suspended. When a guard for a suspended wire is encountered in a different application of the `rx` primitive, the wire is *woken up* and the process' `msgCount` incremented by the number of messages in it.

`claim` errors (see §4.4.3 and §4.4.2)

A run-time error occurs if a Component tries to receive from a wire not `claim`ed. The implementation could be changed so that if a Component tries to receive from a wire it doesn't own, then the wire is automatically claimed. But efficient scheduling depends on Components claiming their wires first.

If a process `disown`s a wire it has not previously `claim`ed, a run-time error will occur. Similarly, if a process `claim`s a wire already claimed, either by itself or by another process, a run-time error will occur.

Alleviating the `claim` problems One way to trap the problem of forgetting to claim a wire before trying to receive from it is to make claimed inputs a different type.

```
claim :: In m -> (Token m -> Process s) -> Process s
from :: Token m -> (m -> Process s) -> Guarded s
disown :: Token m -> (In m -> Process s) -> Process s
```

Now a wire must be claimed first to obtain a token that may be used to receive from the wire. However this cannot stop a Component from passing the token to another Component that tries to receive from it as well. One way to avoid that problem would be to integrate the `claim`ing operation into `rx` (see solution in §4.6.3).

4.6.3 Cannot `rx` from a subset of claimed wires

Sometimes a Component definition would be simpler if on occasions it could receive messages from a subset of the wires it owns. For example, the `memory` Component could be initialised with no contents, and ignore its trigger pin until a value arrived at its input pin. Trigger messages would queue up in the trigger wire and be serviced once a value had arrived. Instead we have to take messages in the order they arrive, storing trigger messages internally until a value arrives.

Allowing `rx` from a subset of `claim`ed wires Rather than keeping a list of *claimed* wires for each process, a process could re-specify the wires it is interested in receiving from every time it tries to `rx`. This list would be the wires for which there is a guarded continuation in the particular application of `rx`. To do this it would be necessary to extract from each guard the wire to which it refers.

4.6.4 Cannot inspect a message in a wire

Sometimes it would be useful to be able to receive a message from a pin only if it satisfies a certain predicate. For example, a Component that takes sorted messages from multiple inputs and merges them to a single pin in sorted order could use the predicate to pick the highest message from a group of wires, leaving the rest in their wires. Currently, we have to read messages in whatever order they come. We can buffer messages within a Component but this may upset the scheduling scheme because internally buffered messages are not counted in the process' message count. If a process could read messages in a different order to that sent, it would be an easy mistake to make a Component that leaves messages building up in one wire while receiving messages in preference from another.

4.6.5 Only one process can receive from each wire

A Component cannot `claim` a wire already `claim`ed by another Component. This is a pity because there are two useful mechanisms that this action could support.

(1) **Multiple-servers:** *many* server Components listen to a wire and for each message sent on it, *one* server receives the message and responds to it. This is useful where the response to a message involves communications with other Components and may take some time. Other servers can continue to service requests whilst

one deals with the first request. Currently we could achieve this using a *server manager* that delegates tasks to a group of servers. This would require messages between servers and server-manager that would not be required if all the servers could connect to a single wire. The extra messages would tell the manager when a server is ready to process another request.

(2) **Broadcast:** multiple Components receive the same messages from a single wire; every message sent on the wire is received by all the Components listening on it. Currently, using a `broadcast` Component to send copies of messages to a group of Components requires multiple messages to be transmitted.

Multiple receivers on a single wire To solve this problem requires a means to specifiy which of the two actions are required: (1) each message goes to one receiver, or (2) each message goes to all receivers. This could be specified by having two different types of wire, *broadcasting* and *distributing*. Alternatively each receiving Component could specify if it requires *sole* or *shared* receivership of messages. On a wire with a sole receiver and several shared receivers, the shared receivers would only get messages whilst the sole receiver was not trying to `rx`. It may also be required that a Component can *grab* a wire to prevent any messages going to other receivers. Other receivers on the wire would get no messages until the one that has grabbed it *releases* it.

Outline of one possible implementation Each wire block has a list of *listening* processes that are waiting for input from the wire. Whenever a Component evaluates an `rx`, the whole list of guards is scanned and a pointer to the process added to the listening list of every wire refered to in a guard. When a message is sent on a *sole receivership* wire, a single process is picked from the listening list, woken (if suspended) and sent the message. When a message is sent on a *shared receivership* wire, all the processes in the listening list are woken (if suspended) and sent the message. A process may be interested in receiving from many wires, but in the end this particular invocation of `rx` will only receive from one of them (the first to carry in a message). For this reason, each reference to a process in a listening list must be in a doubly linked list connecting all the other references to the process in other wire blocks. When a message is received, the reference to the process can be removed from all the other wire blocks. The process may subsequently try to receive from the same set of pins again, thus putting all the references back, but it may not, so

they must all be removed in the mean-time.

This implementation would be much more expensive than the current one, and that is the reason we have not adopted it.

4.6.6 Fairness

The scheduler described here does not guarantee complete fairness. There are two ways in which one or more processes may monopolise the processor and cause other processes to be starved. One way is that high priority processes, ie. ones with a large number of messages, may starve lower priority processes. The other is that a single process may spend a long time processing between I/O interactions, or even get stuck in an infinite loop, preventing other processes from running. Technically, this lack of fairness may result in a loss of referential transparency, in the sense that one expression may terminate, while a supposedly equivalent one may not. In practice, with typical reactive programs, problems are rare, but to make the system safe, a preemptive scheduler with a better priority system would have to be implemented.

4.7 Efficiency

To measure the efficiency of Components and message-passing compared with passing values around a program using lazy streams, we evaluated two programs that pass a stream of the numbers `1` to `100` from one object to another that sums them and prints the total at the end. The purpose of this limited test is to give some idea of the overhead involved in using Components. To enable the Component version to detect the end of the stream, we pass the numbers with a `Maybe` tag: `Yes n` for integer `n`, `None` for the end of the stream. The streams version uses the tags also, to make the comparison fair. The programs are presented in Fig. 4.9 and Fig. 4.10. Evaluating both programs, we get the following results:

Method	Reductions	Cells
Streams	913	1629
Components	1551	3682
Ratio	58.9%	44.2%

So, roughly speaking, using Components doubles the number of reductions and cells used compared with using plain streams. Of course, the streams version of `consume`

could not cope with two input streams without knowing the order to consume messages from them, whereas the Component version, `consumeC` could, and this is why it is worth the extra overhead.

```
generate :: [Maybe Int]
generate = (map Yes [1..100]) ++ [None]

consume :: [Maybe Int] -> Int
consume = consume' 0
consume' n [None] = n
consume' n (Yes h:t) = consume' (n+h) t

stream_eff :: Int
stream_eff = consume generate
```

Figure 4.9: Efficiency test: streams.

```
generateC :: Out (Maybe Int) -> Component
generateC o = sequence [tx o n | n <- generate] $ end

consumeC :: In (Maybe Int) -> Component
consumeC i =
    claim i $
    consumeC' 0 where
    consumeC' :: Int -> Component
    consumeC' n = rx [ from i $ \m -> case m of
        Yes h -> consumeC' (n+h)
        None -> typeout (show n) $ end ]

comp_eff :: Component
comp_eff =
    wire $ \w ->
    spawn (generateC (op w)) $
    spawn (consumeC (ip w)) $
    end
```

Figure 4.10: Efficiency test: Components.

4.8 Summary

We have developed a concurrent process extension to a lazy functional language according to the requirements of GUI programming. Through successive systems

we have refined the language to a point where we are able to program GUIs with the features and aspects that we want. Some shortcomings remain that, if solved, would make Component programming easier. However, we have a system that meets our requirements, so we now move on to test our ideas relating to GUI programming. The main points covered in this chapter were:

- The data structures used to store processes, queues and wires, and the operations on these data structures;
- An explanation of the *World state*;
- Implementation of each primitive function;
- Implementation of I/O device Components;
- Process scheduling;
- A discussion of the shortcomings of the implementation;
- An efficiency comparison with programs using lazy streams.

Chapter 5

Screen Management

5.1 Requirements of Previous Chapters

At the end of Chapter 2, *"Review"*, we listed some desirable features of a system for programming GUIs in a functional language. The extended functional language described in Chapter 3, *"Concurrency,"* and Chapter 4, *"Implementation,"* has met these requirements in the following ways: Concurrent processes can be used to implement the behaviour of objects (§2.6.1 and §2.6.2). Processes communicate by message-passing (§2.6.3). Processes have *pins* through which they send messages. Processes with pins are called *Components*. Lines of communication in a group of Components are defined by linking their pins with *wires*. So Components have locality of definition and independence of specification (§2.6.4). Wire connections are statically type-checked (§2.6.5). Components communicate with the user through special *I/O devices* (§2.6.6). Components may be composed to form more complex Components (§2.6.7). Private state may be held in a Component evaluation (§2.6.8). Components and wires are created dynamically (§2.6.9). The system is functional in nature and requires only a small set of imperatively programmed primitives that have functional types. Messages may be any first class value. We can use laziness, higher-order functions and abstraction in the definition of Components, and in wiring them together. (§2.6.10)

5.2 SM Requirements

Whilst our extended functional language has met our requirements in terms of program structure, we have yet to provide Components with *shared* access to the I/O hardware. The `screen`, `keyboard` and `mouse` I/O devices allow only a single Component to access the hardware. We list our requirements for screen management, and explain why they are needed:

5.2.1 Components can have a screen image

Components that implement objects in the GUI require an image on the screen display. We call Components with an image *Gadgets*. The fact that the screen may be shared by many Gadget images is hidden from a Gadget, so it can be defined without knowing what other Gadgets will be used in the same program. This is similar to our requirement of §2.6.4.

5.2.2 Mouse and keyboard input

Keyboard input can be directed to a particular Gadget. This Gadget is said to have the *input-focus*. A Gadget might request the input-focus when it is selected with the mouse, when it is created, or in response to some other event.

Mouse input is directed to Gadget whose image is beneath the mouse pointer at the time of the click. The coordinates of a mouse click are given relative to the corner of the image, so that a Gadget's definition is independent of the absolute image position.

5.2.3 Gadgets can be composed

Just as Components can be composed (§2.6.7), so can Gadgets. This enables us to make more complex Gadgets out of simpler ones. A Gadget creates a composition of Gadgets, just as a Component creates a composition of Components. When composing Gadgets we specify not only the wire connections between them, but also the relative layout of their images on the screen. Images in a composition appear inside the image of the composing Gadget, but can be positioned in any format within that image.

5.2.4 Images can overlap

The amount of space on the screen is limited, so images can overlap. The fact that *layers* of images are *projected* onto a flat screen display is hidden from the Gadgets. They do not have to redisplay their image if it becomes *exposed* when an overlapping image moves. This is in keeping with the requirement that Gadgets are independent (§5.2.1).

5.2.5 Images are dynamic

New Gadgets may be created and destroyed dynamically during the course of program execution. For example, this will be useful in programs such as editors in which multiple views can be created by selecting a menu option. This matches our requirement for dynamically created Components (§2.6.9).

5.3 In This Chapter

In this chapter we describe a Screen Manager Component, called *SM*, that stands between the I/O hardware and the Gadgets of a program (see Fig. 5.1). SM provides Gadgets with a hierarchy of overlapping images on a bitmapped screen with mouse and keyboard input.

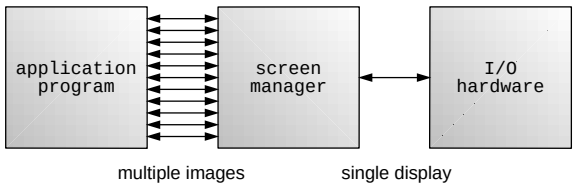


Figure 5.1: The Screen Manager interfaces program to display.

The rest of the chapter divides as follows: In §5.4 we look at the structure of a Gadget program and the type of a Gadget. In §5.5 the interaction between SM and Gadgets is described. Section 5.6 deals with the specification of layout and hierarchy *outside* SM. In §5.7 we look *inside* SM and explain how it provides each Gadget with an image on the screen. We explain the data structures used and associated operations on them. These operations have been made efficient though they are defined in a functional language. In §5.8 we summarise the work on screen

management and discuss the success of SM. Sections following this concern further issues related to screen management. Coping with Gadgets with a large number of attributes is discussed in §5.9.1. Maintaining a large piece of state in a Component such as SM is discussed in §5.9.2. Finally we conclude in §5.10 with some possible future work.

5.4 The Structure of a Gadget Program

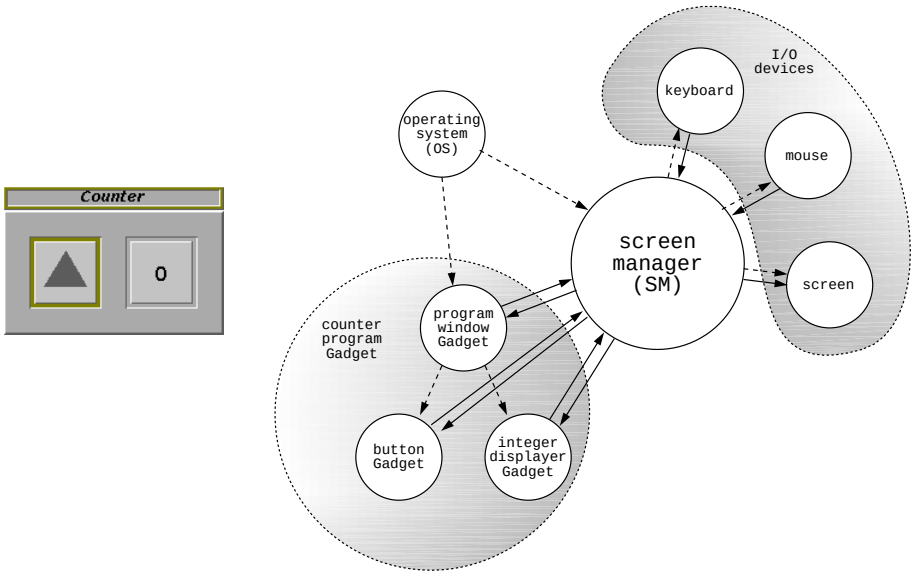


Figure 5.2: The Processes in a Gadget Program. Dotted lines indicate process creation, and are drawn from parent to child. Wires are shown as solid lines.

Figure 5.2 shows a diagram of the Components and images of a simple counter Gadget. SM maintains a hierarchy of overlapping images, consisting of a single *root window* (not shown in the diagram) at the top containing many overlapping *program windows* that each contain images of buttons, icons and other Gadgets. A Gadget alters this data structure when it changes its image. SM is responsible for *projecting* this data structure onto the two dimensional plane that is the screen, and maintaining the display when the data structure changes.

The operating system (OS) Gadget is the first to be created when a Gadget program is run. OS is the Gadget that owns the *root window* image. OS starts SM, then each of the programs to be run, creating the image for each program window within the root window. Each window, icon, button or other interface part

on the screen has a corresponding Gadget. When a Gadget creates a composition of Gadgets, it is responsible for connecting them all to SM. For example, the `counter` Gadget creates the program window, button and display Gadgets, and connects them to SM.

5.4.1 The Type of a Gadget

On Page 86 we saw how a Component is a process that holds some state of type `ComponentState`. Another type of process is a `System` that has a state of type `SystemState`. A `System` Component has a connection to OS, so that it can obtain connections to system resources like a file manager. The ends of the wires connecting it to OS are held in the `SystemState`.

A Gadget is a Component with a connection to OS, SM and a layout manager Component (layout is discussed later in §5.6). Like a `System` the wire-ends of the connections to OS and SM are held in the process state of type `GadgetState`.

`System` and `Gadget` processes keep these connections in the state so that the wires do not have to be explicitly mentioned in the process definition. The connections are made by a function called at the start of the process' evaluation (eg. `initGadget`), and accessed by library functions (eg. `txSM` which sends a request to SM).

5.5 The Gadget ↔ Image Connection

The connection between a Gadget and SM behaves as if it were a direct connection to the Gadget's image on the screen (Fig. 5.3). If the process sends a new picture through this connection, it appears as the image on the display wherever the image is visible (ie. not covered by other images). If the mouse is clicked on an image, a message is sent to the Gadget that owns the image.

5.5.1 Using SM

Ideally the programmer is rarely concerned with communication between Gadget and SM. A large library of Gadgets supplied with the system provides the programmer with many ready-made interface Components, so programs consist of compositions of library Gadgets and Components. A tutorial on Gadget and Component composition is given in Appendix A, *"The Gadget Gofer Manual"*. However, this chapter

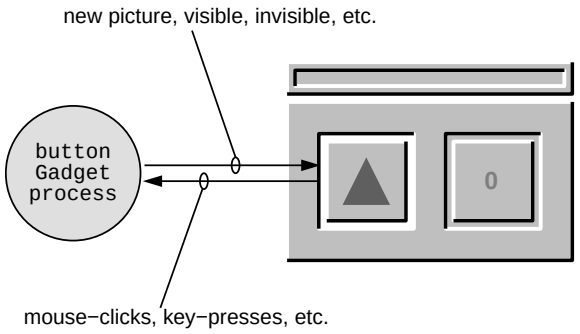


Figure 5.3: A Connection to SM is Like a Connection to an Image on the Screen.

describes how the system works rather than how to use the system, so we include details of SM requests and responses here.

5.5.2 SM Requests

Gadgets send messages of type `SMRequest` *to* the SM and receive messages of type `SMResponse` *from* SM. The requests a Gadget can make are:

```
data SMRequest
    = SMNewCon (In SMRequest,Out SMResponse)
    | SMDrawFun DrawFun
    | SMUpdateDrawFun DrawFun DrawFun
    | SMInvisible
    | SMVisible
    | SMClickAction ClickAction
    | SMUnClickAction ClickAction
    | SMMulti [SMRequest]
    | SMClaimFocus
    | SMDisownFocus
    | SMUpdates [DisplayChange]
```

We look at each request in turn:

`SMNewCon (In SMRequest,Out SMResponse)`

When a Gadget wishes to create a new *child* Gadget to appear within its own image, it spawns the Gadget with a duplex pair of wires attached for connection to SM (Fig. 5.4). It passes the other ends of the wires to SM in a `SMNewCon` message. SM takes these wire-ends and makes a new connection to itself with them.

Being able to pass wires, wire-ends and Components in messages is a very useful technique in programs whose structure changes during evaluation. It is discussed further in Chapter 7, *"Conclusions and Future Work"*.

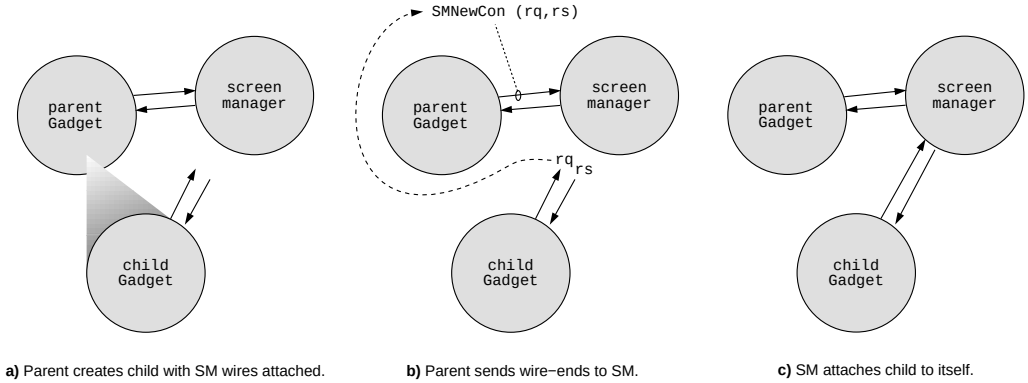


Figure 5.4: A Gadget Creates a Child Gadget.

`SMDrawFun DrawFun`

Gadgets describe the picture in their image with a drawing-function. When applied to the size of the area it has to draw in, and the part of the area that needs drawing, a drawing function computes a list of drawing commands that result in a picture.

```
type DrawFun = Size -> Area -> [ScreenCmnd]
```

The drawing could be described directly by a list of drawing commands, so why use a function? There are three reasons. (1) by using a function, the picture can be made image-size independent. (2) There is the potential to take up less memory in between redraws. A simple list of drawing commands, once evaluated, will take up space for the whole list as long as the list may be required to redraw the image. A function can produce the same list of drawing commands but this list can be thrown away afterwards if all that is held is the function (un-applied to *image-size* and *area-changed*). (3) A function can be made sensitive to partial image exposures. If the drawing is complex and only part of the image is exposed, then it may be quicker for the drawing function to filter out those parts of the drawing that are not required. Otherwise it generates drawing commands for the whole drawing, only to have the majority of them clipped to the area being redrawn.

`SMUpdateDrawFun DrawFun DrawFun`

If only a small change has to be made to the picture in an image, this request can make that change without forcing the whole image to be redrawn. Two drawing functions are sent. The first describes the change to the image already displayed. The second describes the whole picture, and will be used if the image subsequently needs redrawing because it becomes exposed.

SM could calculate the second drawing function by combining the update with the one it already holds, but such a function would return a longer list of screen commands with every `SMUpdateDrawFun` message sent. Sending the second drawing function with an update avoids this. For instance, if the update function changes one word to another by *blotting out* the old with a filled rectangle before writing the new, then the second drawing function can omit drawing the first word and filled rectangle, and just draw the new word. A combination of the update with the old drawing function would draw the old word, the filled rectangle and the new word.

Although the second drawing function should draw the current picture plus the given change, there is no way to enforce this.

`SMInvisible` **and** `SMVisible`

These requests make the image become invisible or visible again. Applications include a *pop-up* image such as a menu. A pop-up image could be created and destroyed dynamically, but if there is only ever going to be one instance of the pop-up, then this method can be used. It requires less computation to make an image invisible or visible than to create or destroy a composition of Components and images.

`SMClickAction ClickAction` **and** `SMUnClickAction ClickAction`

Normally SM responds to mouse-clicks and unclicks by sending a message to the Gadget whose image has been clicked. We can redefine the response by changing the click-action or unclick-action. For example, we could redefine the click-actions of a set of buttons in a keypad Gadget to send their messages to the image containing the buttons. This has the effect that the containing image *intercepts* the mouse-click messages sent to its children. The `ClickAction` type is discussed later in §5.7.1.

`SMMulti [SMRequest]` **and** `SMUpdates [DisplayChange]`

SM updates the screen display after each message it receives from a Gadget. The `SMMulti` message enables a Gadget to send a list of requests to SM. The screen is only updated *once* after the whole list has been processed. For example, a Gadget can change its size and picture in one go. This speeds up the operation and prevents intermediate states being displayed.

The `SMUpdates` message enables a whole set of changes to be made to the images on the display in one go. It is used when the Gadgets of a program are initially created, and when the layout of a set of Gadgets changes. We discuss this further in §5.6.

`SMClaimFocus` **and** `SMDisownFocus`

Some Gadgets are able to accept key-presses from the keyboard. It only makes sense for one Gadget to be accepting key-presses at any one time. This Gadget is said to have the *input-focus*. A Gadget obtains the input-focus by sending a `SMClaimFocus` message to SM. Normally the Gadget sends this message when it is clicked, but it could claim the input-focus in response to some other event, for example when it is first created. On receipt of a `SMClaimFocus` message SM informs the previous holder of the focus that it has lost the input-focus. Key presses are forwarded to the focus owner, until such time as another Gadget claims the focus, or it gives up the focus by sending a `SMDisownFocus` message. Normally a Gadget will highlight itself in some manner to indicate that it has the focus.

There are other ways keyboard input could be handled. The way we have chosen cannot enforce a common mode of highlighting a Gadget with the input-focus, which may confuse the user. This issue is discussed further in §5.10.

5.5.3 SM Responses

SM sends `SMResponse` messages to a Gadget. There is *not* a one-to-one correspondence with `SMRequest` messages:

```
data SMResponse
    = SMMouseClick Int Int Int
    | SMMouseUnClick Int Int Int
    | SMKeyPress Char
    | SMLoseFocus
```

We look at each response in turn:

`SMMouseClick Int Int Int` **and** `SMMouseUnClick Int Int Int`

When the mouse is clicked (or unclicked) SM determines which image the mouse is over, taking into account the overlapping of images and clipping of children to the borders of parent, and forwards a message to the correct Gadget. The message indicates the position (x and y coordinates relative to the top left of the image) and the number of the button pressed or released on the mouse.

`SMKeyPress Char` **and** `SMLoseFocus`

If any Gadget owns the input focus when a key is pressed, the character will be forwarded in a `SMKeyPress` message to that Gadget. Keyboard input is discussed further under the section on the `SMClaimFocus` request on Page 129.

If a Gadget claims the input focus while another holds it, SM informs the latter that it has lost the focus with this message. A Gadget should respond by unhighlighting itself.

5.6 Gadget Layout

A Gadget could change the size and position of its image in the same way as it changes its picture by sending requests to SM. However, a Gadget is defined independently of its place of use. It knows nothing of the Gadgets it will be placed beside on the screen. For example, the buttons of a numeric keypad Gadget do not know where to place themselves in order to form a tidy grid of buttons. It is up to the parent of the buttons to arrange them.

A Gadget may decide what size its image will be, then subsequently change its size. For example a text-displaying Gadget may need to display more text. If a *parent* Gadget is to arrange the position of its *child* Gadgets neatly then it needs to keep track of their sizes. For example, if one of the buttons in a keypad Gadget grows, then the positions of some of its neighbours may have to change, and the parent may have to grow to accommodate the whole set.

To satisfy these needs, the position and size of Gadgets are *not* set by sending requests directly to SM. Instead, Gadgets send the size they require to their parent. A parent collects the sizes of all its children, then calculates the position of each

child according to the desired layout (eg. all children placed in a horizontal row). The parent sends its new size, together with positions for all its children, to *its* parent. So the size and position details work their way to the top of the Gadget image tree. The OS owns the root window that is the top of the tree and sends a single `SMUpdates` message. SM uses this message to make all the changes and update the screen in one traversal of its data structure. The layout mechanism is illustrated in Fig. 5.5.

5.7 Inside SM

SM remembers all the Gadget images, how they fit into the hierarchy, what is displayed in them, what size they are, etc. It processes requests from Gadgets that change the data, and updates the screen display accordingly. SM can make several updates to the data structure, then update the screen display in one go. In this section we look at the data structures of SM and the operations performed on them.

5.7.1 Internal Types of SM

Coordinates on the screen are represented by a pair of integers:

```
type Coord = (Int, Int)
```

Rectangular areas of the screen or within an image are represented by a pair of coordinates (top left and bottom right of the rectangle):

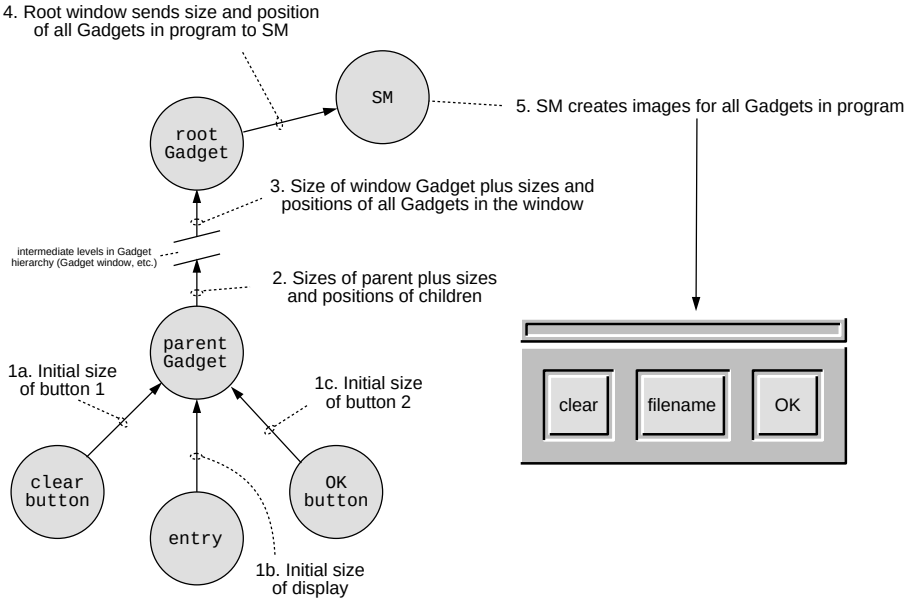
```
type Area = (Coord, Coord)
```

The placement of an image within its parent is specified by the area it covers relative to the origin of the parent:

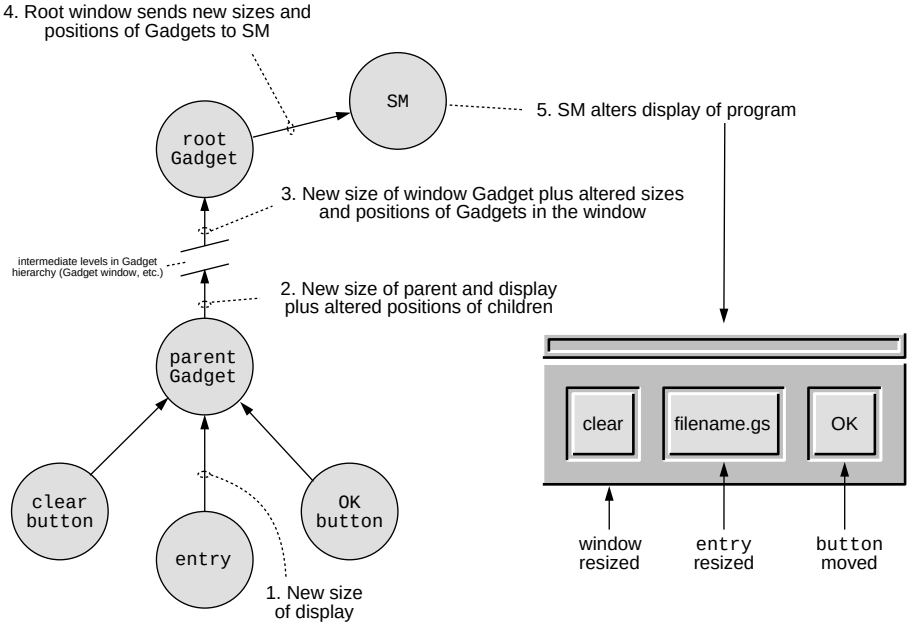
```
type Placement = Area
```

The position is given relative to the parent rather than as an absolute position on the screen so that when an image is moved, all its children move with it, without changing their specified position. The visibility of an image is denoted by a boolean, true for visible:

```
type Visibility = Bool
```



The Creation of the `fileentry` Gadget.



Resizing the `fileentry` Gadget.

Figure 5.5: Consider a simple example. A Gadget consists of a filename-entry window. A filename is entered into the central box. Clicking the left-hand button clears the entry box, the right-hand button is clicked when the filename is correct. If the filename entered becomes too long to fit into the entry box, it resizes.

The children of an image are given in a list, front-most child first:

```
type Children = [Display]
```

Children are placed in layers although this will not be apparent unless they overlap. We can make some display operations more efficient if we know whether the children of a Gadget overlap or are disjoint. So when the children of a Gadget are created, the parent states whether the children will overlap or not. SM remembers this fact for each parent. The display operations can be made more efficient because SM does not have to calculate which children will be obscured by others. By using the layout combinators, for example `<->` and `<|>`, disjoint displays will automatically be used.

```
type Overlap = Bool
```

A click- or unclick-action is described by a function. Its arguments are the coordinates of the mouseclick, the mouse button number and a continuation `c`. Its result is a function of the Component state that performs some action (eg. transmits a message along a wire), then continues with `c`. When the mouse is clicked SM applies the click-action of the image beneath the mouse to the mouse coordinates (relative to the top left of the image), the mouse button number, and the continuation given is the continued operation of SM. So the actions of the click-action are *inserted* into the operation of SM.

```
type ClickAction = Coord -> Int -> Component -> Component
```

The original motivation for doing this was to allow the type of message sent to a particular Gadget to be changed without changing the definition of SM. This is possible because the type of the message does not appear in the type of the button action function. It could be used to instruct SM to send a Gadget a `Click` message that ignores the mouse coordinates and button number. For example:

```
clickact :: ClickAction
clickact _ _ c = tx (op w) Click $ c
```

sends `Click` on wire `w`. The type of the message sent (`Click`) does not appear in the type of `clickact`.

This is a very flexible approach, though somewhat experimental. Care must be taken to call the supplied continuation eventually, as SM will not respond to any further messages until then. It must call the continuation *soon* too, if SM

is to continue operation without any noticable delays. SM could be mutated into a completely different Component by ignoring the continuation supplied by SM! There is no way to check statically that this won't happen. To make this facility more secure, an algebraic type could be used to indicate what operations the SM is prepared to let the click-action engage in.

The situation could be slightly improved by changing the `ClickAction` type to:

```
type ClickAction = Coord -> Int -> Component
```

where the click action applied to coordinate and button number returns a process that SM spawns when a Gadget is clicked. This process performs the desired action (sending a message or whatever) and then terminates. SM continues operating concurrently. The problem now is that we cannot check statically that the process will terminate, so the number of active processes could increase by one every time the Gadget is clicked. This method is slightly more expensive too, as it requires a process to be spawned and then terminate.

5.7.2 State Inside SM

SM maintains three items of state (see Fig. 5.6):

- a *display* data structure representing the hierarchy of images.

```
data Display = Image
                    ImageID   -- identifies the image
                    ImageInfo -- see below
                    [Display] -- list of children
                    Overlap   -- True if children overlap

data ImageInfo = ImageInfo
                    Placement   -- position of image inside parent
                    DrawFun     -- the drawing function
                    Visibility  -- True if this image is visible
                    ClickAction -- the button click action
                    ClickAction -- the button unclick action
                    Size        -- size of image
```

- a list of screen rectangles (areas of the screen, *not* images) that need redrawing to reflect changes in the display structure, of type `[Area]`.

- a list of *cached* changes to be made to the display data structure. Each cached change takes the form of the id. of the image to be changed paired with a function to change it.

```
type DisplayChange = (ImageID,ChangeFun)
type ChangeFun = Display -> Coord -> [Area] ->
                                  [Area] -> [Area] -> (Display,[Area])
```

Multiple changes sent in an `SMUpdates` or `SMMulti` message (see §5.5.2) are placed into the update cache. They are cached so that multiple changes can be applied in a single traversal of the display data structure.

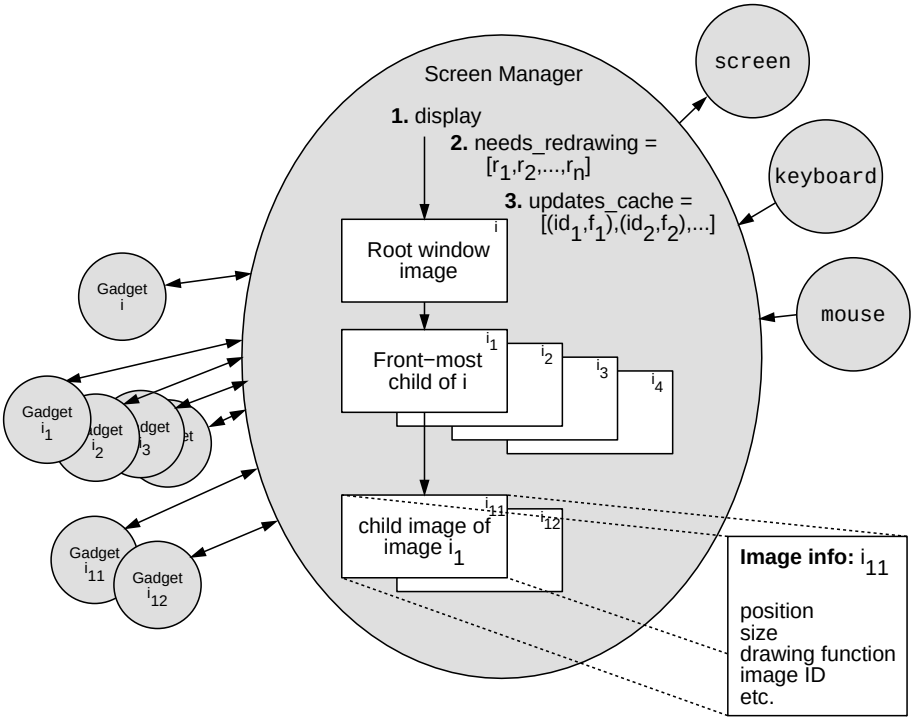


Figure 5.6: Inside the Screen Manager.

5.7.3 Screen Manager Operations

There are three operations we perform with the display structure:

Determine which image a mouse-click *lands* in. This operation picks the image immediately underneath the tip of the mouse pointer, taking into account the fact that an image is not visible outside the borders of its parent, and images

may be obscured by other images. For example, by looking at the coordinates of the images in Fig. 5.7, the click appears to be within the borders of image C. However, the mouse-click is associated with image A because image C is not visible outside the borders of its parent, image B.

Redraw parts of the screen where areas have been changed. SM keeps a list of screen areas that need updating to reflect changes in the display data structure. Redrawing involves generating a list of screen commands to redraw the areas in this list.

Update the display structure with some new information for a particular image (eg. move image to a new position). This operation may result in more rectangles that need redrawing. For example if the image in Fig. 5.8 is moved as shown then the shaded areas will be divided into rectangles and added to the redraw-list. Only rectangles containing visible parts of changed images are added to the list.

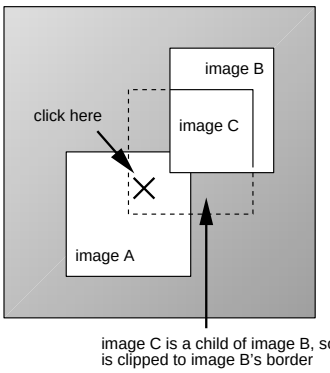


Figure 5.7: Determining the Destination of a Click.

5.7.4 Defining SM Operations

SM is defined in an interpreted lazy functional language that does not evaluate programs particularly quickly. It is important therefore to program SM operations as efficiently as possible. Operations that update SM's display data structure change pieces of information stored within it. Because we are using a lazy functional language we cannot use side-effects to update the information in-place. We could use mutable variables [LJ94] to introduce in-place updates, and maintain SM state in

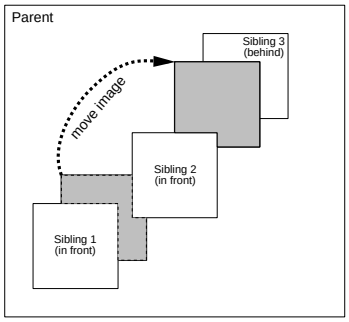


Figure 5.8: Move an Image: shaded areas need redrawing.

much the same way as would be done in an imperative language. We discuss this option further in §5.10. Instead we chose to keep to a purely functional definition that creates a new version of the display data when a change is required. We re-use parts of the old data where possible. We try to traverse the data structure as few times as possible, gathering whatever information we need to construct the new display data. There is a lot of information to collect. Some of it is stored directly within the data (for example, the drawing function, needed when redrawing the screen). Some must be calculated from the data (for example, the visible areas of an image, needed to discover which areas of the screen need redrawing when an image's picture is changed).

To simplify the definition of these operations we use *Attribute Grammars* [Knu68] to specify some extra information about each image in addition to the information stored.

5.7.5 Attribute Grammars

An *attribute grammar* is a context-free grammar augmented with semantic rules that define a fixed set of attributes associated with each non-terminal in the grammar. By evaluating the attributes we *decorate* the parse tree. The rules for a given production in the grammar, such as:

$$X0 \quad \rightarrow \quad X1\ X2\ \ldots\ Xn$$

may be accompanied by equations defining *synthesized* attributes of X0, and *inherited* attributes of each of X1, X2,...,Xn. Synthesized attributes propagate upwards in the parse tree; inherited attributes propagate downwards. An inherited attribute

`a` of non-terminal X is written `X`↓`a`; a synthesized attribute `a` is written `X`↑`a`. In addition to the rules defining attributes for each production in the grammar we also define a special case, the *start production*, matching only the root of the parse tree.

Example Consider the grammar of a binary tree whose leaves contain integers:

Tree → `Integer`
Tree → $Tree_1 \wedge Tree_2$

We can define an inherited attribute, ↓`depth`, that indicates how many levels down the tree we are, and a synthesized attribute, ↑`sum`, indicating the sum of a sub-tree's *weights*, where the *weight* of a leaf node is the integer multiplied by its depth in the tree and the *weight* of an internal node is the sum of the weights of its branches:

Tree → `Integer`
Tree↑`sum` = `Integer`.value * *Tree*↓`depth`

Tree → $Tree_1 \wedge Tree_2$
$Tree_1$↓`depth` = *Tree*↓`depth` + 1
$Tree_2$↓`depth` = *Tree*↓`depth` + 1
Tree↑`sum` = $Tree_1$↑`sum` + $Tree_2$↑`sum`

and the start production:

Tree → *tree*
tree↓`depth` = 1
Tree↑`sum` = *tree*↑`sum`

Now if we want to know the sum of all the integers multiplied by their depths in the tree, we simply decorate the tree and take the value of ↑`sum` at the root. Generating a function to decorate a tree with attributes is very simple and the technique is described in [Joh87]. For example, the tree summing operation translates to the function given in Fig. 5.9.

In this case, the function could easily have been defined without using attribute grammars. In more complex cases involving many attributes, some of which depend on the values of others, the functions are more difficult to comprehend, but can be expressed more simply with attribute grammars.

5.7.6 The Display Grammar

The grammar that describes the display data structure of SM has a single terminal `Image` and non-terminals *Display* and *Children*. The start symbol is *Display*. The

```
data Tree = Leaf Int | Branch Tree Tree
type Depth = Int
type Sum = Int

sum :: Tree -> Sum
sum = sum' 1
    where sum' :: Depth -> Tree -> Sum
          sum' i_depth (Leaf n) = i_depth * n
          sum' i_depth (Branch t1 t2) =
              let t1_i_depth = i_depth + 1
                  t2_i_depth = i_depth + 1
                  s_sum = t1_s_sum + t2_s_sum
                  t1_s_sum = sum' t1_i_depth t1
                  t2_s_sum = sum' t2_i_depth t2
              in s_sum
```

Figure 5.9: The Tree Summing Function.

productions are:

$$\begin{array}{rcl} \textit{Display} & \rightarrow & \texttt{Image}\ \textit{Children} \\ \textit{Children} & \rightarrow & \textit{Display}\ \textit{Children} \mid \varepsilon \end{array}$$

Some information for each image is stored in the data structure, identified by attaching a label or name to the terminal with a full stop. The types of information, inherited attributes and synthesized attributes are listed in Fig. 5.10.

The Click Destination Operation

Using the three attributes ↓`offset`, ↓`bounds` and ↑`dest`, we define the click-destination operation. The grammar annotated with rules for each attribute is given in Fig. 5.11. The ↓`offset` attribute is used to calculate the absolute position of each image on the screen. The ↓`bounds` attribute is used to clip each image to the borders of its parent. For a image `i`, `i`↑`dest` is either (1) the ID of one of its descendants that encompasses the mouse-click coordinate. If there is none then (2) `i.id` if `i` encompasses the mouse-click coordinate. If it does not then (3) an 'empty' value (`i`↑`dest` is a `Maybe` type — it may contain a value, or may be empty). Children are always in front of their parent, so they take precendence for catching mouse-clicks. At the top of the parse tree (ie. the top of the display data structure) the value of the ↑`dest` attribute determines the image, if any, that receives this mouse-click.

`i.rpos` — the image's position relative to its parent

`i.children` — the image's children

`i.df` — the drawing function describing the image's picture

`i.id` — the image's ID

`i.ca` — the image's click-action

`i.uca` — the image's unclick-action

`i.overlapping` — 'true' if the image's children overlap

`i.size` —the image's size

`i.vis` — 'true' if the image is visible

`i↓offset` — this is the offset to be added to the image's position (`i.rpos`) to get coordinates relative to the screen origin. This is equal to the absolute position of the parent's origin. The offset is used for calculating the image's absolute position on the screen.

`i↓bounds` — these are the borders of the parent image. An image is invisible outside of these bounds.

`i↓redraw` — the areas of the screen that need updating after updating or redrawing the images in front of this one.

`i↓visible` — the areas of the image's parent that are visible after taking the borders and other siblings that are in front of this image into account.

`i↓changes` — the list of change functions that remain to be applied before this image has been updated.

`i↑covers` — this is the area covered by an image, after clipping the image to its parent's border. This is useful for calculating the areas of an image that are covered by other images.

`i↑redraw` — the areas of the screen that need updating as a result of changes made to images in front of and including this one.

`i↑scrcmnds` — a function of the form `(sc++)` where `sc` is a list of screen commands generated to update the display to reflect changes in the display data structure. We use a function of this type rather than a plain list of screen commands so that each list is traversed once to concatenate them.

`i↑changes` — the list of changes to images in front of and including this one.

`i↑display` — the image in the display data structure after any changes have been made.

`i↑dest` — this attribute is used in the operation of finding the image a mouse-click lands in to hold either the id. of the destination image if the destination is this image or one in front of it, or nothing.

Figure 5.10: The Information Associated with an Image `i`.

Conversion of this attribute grammar to a function is straight-forward (Fig. 5.12). We do not have to consider the order of evaluation for attributes as laziness ensures the correct order is chosen for us. We have used a cut-down version of the display datatype and ignored the image `visibility` flag for brevity. The algebraic datatype and click destination function are given in Fig. 5.12.

The Screen Redraw Operation

Using the attributes ↓`offset`, ↓`redraw`, ↑`scrcmds` and ↑`redraw`, we define the redraw operation. The ↓`offset` attribute is used as before. For a particular image, the ↓`redraw` attribute indicates the areas of the screen that need updating and haven't been redrawn by the images in front of the current one. The ↑`scrcmds` attribute is used like an *accumulator* to collect the list of screen commands to redraw the screen. The ↑`redraw` attribute holds what is left of the ↓`redraw` list after an image has been redrawn. At the top of the parse tree (ie. the top of the display data structure) the value of the ↑`scrcmds` attribute supplies the resulting list of screen commands. The grammar annotated with rules for each attribute is given in Fig. 5.13. Conversion of this attribute grammar to a function is found in Fig. 5.14.

The Update Operation

Using the attributes ↓`offset`, ↓`visible`, ↓`redraw`, ↓`changes`, ↑`display`, ↑`covers`, ↑`redraw` and ↑`changes` we define the update operation.

The `update` function traverses the `Display` data structure, calculating the absolute origin of each image[1] using the `.offset` attribute, and the areas of each image that are visible[2]. Where `i.id` corresponds to the ID of change in the cache, the `i`↑`display` equals the change function applied to

`i`↓`image` the portion of the `Display` data structure that describes the image.

`origin` calculated from `i.rpos` and `i`↓`offset`.

`i`↓`visible` a list of areas inside the parent image still visible after covering by other children in front of this child.

`c-visible` the `visible` list less the areas covered by the image's *own* children.

[1] the `Placement` is relative to the origin of its parent stored for each image

[2] taking into account the borders of its parent and any *siblings* of the image that obscure it

Display → `image` *(the start production)*

`image`↓`offset` = (0,0)
`image`↓`bounds` = *dimensions of screen*
Display↑`dest` = `image`↑`dest`

Display → `Image` *Children*

apos = `Image`↓`offset` + `Image.rpos`
visible = apos ∩ `Image`↓`bounds`
thisone = if click 'within' visible then `Image.id` else ε

Children↓`offset` = apos.topleft
Children↓`bounds` = visible

`Image`↑`dest` = if *Children*↑`dest` == ε then thisone
else *Children*↑`dest`

Children → *Display* $Children_1$

Display↓`offset` = *Children*↓`offset`
Display↓`bounds` = *Children*↓`bounds`
$Children_1$↓`offset` = *Children*↓`offset`
$Children_1$↓`bounds` = *Children*↓`bounds`

Children↑`dest` = if *Display*↑`dest` == ε then $Children_1$↑`dest`
else *Display*↑`dest`

Children → ε

Children↑`dest` = ε

Figure 5.11: The Click Destination Attribute Grammar.

```
data Display = Image ImageID Placement Children

data Children = Children Display Children | NoMore

data Maybe a = Yes a | None

click_dest :: Display -> Coord -> Maybe ImageID
click_dest display click = cd (0,0) scrDimen display
  where
  cd :: Coord -> Area -> Display -> Maybe ImageID
  cd i_offset i_bounds (Image imid rpos children) =
    let apos = rpos `addArea` i_offset
        visible = apos `intersectArea` i_bounds
        thisone = if click `within` visible then Yes imid else None
        c_i_offset = topleft apos
        c_i_bounds = visible
        c_s_dest = cc c_i_offset c_i_bounds children
    in s_dest = if c_s_dest == None then thisone else c_s_dest
  cc :: Coord -> Area -> Children -> Maybe ImageID
  cc i_offset bounds (Children display children) =
    let d_i_offset = i_offset
        d_i_bounds = i_bounds
        c_i_offset = i_offset
        c_i_bounds = i_bounds
        d_s_dest = cd d_i_offset d_i_bounds display
        c_s_dest = cc c_i_offset c_i_bounds children
    in s_dest = if d_s_dest == None then c_s_dest else d_s_dest
  cc i_offset bounds NoMore = None
```

Figure 5.12: The Click Destination Function.

Display → `image` *(the start production)*

`image`↓`offset` = (0,0)
`image`↓`redraw` = `id`
Display↑`scrcmnds` = `image`↑`scrcmds`
Display↑`redraw` = `image`↑`redraw`

Display → `Image` *Children*

apos = `Display`↓`offset` + `Image.rpos`
visible = apos ∩ `Display`↓`redraw`
origin = apos.topleft

Children↓`offset` = origin
Children↓`redraw` = visible

Display↑`scrcmds` =
```
   foldr (.) Children↑scrcmds
      (map (generate origin Image.df Image.size)
                                        Children↑redraw)
```
Display↑`redraw` = `Display`↓`redraw` - apos

Children → *Display* $Children_1$

Display↓`offset` = *Children*↓`offset`
Display↓`redraw` = *Children*↓`redraw`
$Children_1$↓`offset` = *Children*↓`offset`
$Children_1$↓`redraw` = *Display*↑`redraw`

Children↑`scrcmds` = ($Children_1$↑`scrcmds` . *Display*↑`scrcmds`)
Children↑`redraw` = $Children_1$↑`redraw`

Children → ε

Children↑`scrcmds` = `id`
Children↑`redraw` = *Children*↓`redraw`

```
where
generate origin df size area =
    (DrawSetClip area:moveOrigin origin (df size (area - origin)))
```

Figure 5.13: The Screen Redraw Attribute Grammar.

```
data Display = Image ImageID Placement Children

data Children = Children Display Children | NoMore

redraw :: Display -> [ScreenCmnd]
redraw display = fst (rd (0,0) [] display)
 where
 rd :: Coord -> [Area] -> Display ->
                                  (([ScreenCmnd] -> [ScreenCmnd]),[Area])
 rd i_offset i_redraw (Image imid rpos df size children) =
   let apos = rpos `addArea` i_offset
       visible = intersectArea apos i_redraw
       origin = fst apos
       c_i_offset = topleft apos
       c_i_redraw = visible
       (c_s_scrcmds,c_s_redraw) = cc c_i_offset c_i_redraw children
       s_scrcmds = foldr (.) c_s_scrcmds (map (generate origin df size)
                                                          c_s_redraw)
         where generate origin df size area =
          (DrawSetClip area:moveOrigin origin (df size (area - origin)))
       s_redraw = i_redraw - apos
   in (s_scrcmds, s_redraw)
 rc :: Coord -> [Area] -> Children ->
                                  (([ScreenCmnd] -> [ScreenCmnd]),[Area])
 rc i_offset i_redraw (Children display children) =
   let d_i_offset = i_offset
       d_i_redraw = i_redraw
       c_i_offset = i_offset
       c_i_redraw = d_s_redraw
       (d_s_scrcmds, d_s_redraw) = rd d_i_offset d_i_redraw display
       (c_s_scrcmds, c_s_redraw) = rc c_i_offset c_i_redraw children
       s_scrcmds = (c_s_scrcmds.d_s_scrcmds)
       s_redraw = c_s_redraw
   in (s_scrcmds, s_redraw)
 rc i_offset i_redraw NoMore =
   (([]++), i_redraw)
```

Figure 5.14: The Screen Redraw Function.

`i↓redraw` the current list of areas of the screen that need redrawing.

to yield `i↑display` and `i↑redraw`. The areas added to `i↓redraw` to form `i↑redraw` are calculated from the other parameters to the change function, depending upon how the image has been changed. For instance, if the picture is being changed, the area of the image intersecting with `c-visible` is added, i.e. those parts of the picture not obscured by siblings or its own children. If the image is being moved, then the old area of the image intersecting with `visible`, plus the new area of the image intersecting with `visible`. This time we include the areas covered by the image's own children because they move with the image.

The grammar annotated with rules for each attribute is given in Fig. 5.15. At the top of the parse tree (ie. the top of the display data structure) the value of the `↑display` attribute holds the updated display data structure and `↑redraw` the list of screen areas that have changed. This attribute grammar is implemented by the function is defined in Fig. 5.16.

5.8 Discussion

A summary of the achievements described in this chapter On a foundation of a concurrent process functional language with access to the I/O hardware at a simple level, we have built a framework for GUI programs. This consists of one process per interface part, called a *Gadget*. Gadgets are composable and can be interconnected with layout Components that keep their relative layout constant despite changes in Gadget size. Each Gadget is connected to a Screen Manager (SM) that provides each with an image on a *virtual* display. The virtual display presents a hierarchy of layered images to both the user and the GUI programmer. SM is a pure functional evaluation. It holds a data structure that describes the display, and messages received by it trigger operations on that data. Attribute grammars were used to give a declarative definition of these operations.

5.8.1 Evaluating the success of SM

Measuring against the requirements

Looking back at our requirements in §5.2, we find that we have achieved all bar one — the ability to dynamically destroy an interface component. Gadgets have

Display → `image` *(the start production)*

`image`↓`offset` = (0,0)
`image`↓`visible` = `[scrDimen]`
`image`↓`redraw` = `[]`
`image`↓`changes` = *list of (image id., change function) pairs*
Display↑`display` = `image`↑`display`
Display↑`covers` = `image`↑`covers`
Display↑`redraw` = `image`↑`redraw`
Display↑`changes` = `image`↑`changes`

Display → `Image` *Children*

(match,mismatch) = `span ((==Image.id).fst)` *Display*↓`changes`
(`Image'`,redraw') = `applyAllChanges` match `Image` *Display*↓`offset`
Display↓`visible` *Children*↑`visible` *Display*↓`redraw`
apos = *Display*↓`offset` + `Image'.rpos`
visible = apos ∩ *Display*↓`redraw`

Children↓`offset` = apos.topleft
Children↓`visible` = visible
Children↓`redraw` = redraw'
Children↓`changes` = mismatch
Display↑`display` = `Image'` *Children*↑`display`
Display↑`covers` = visible
Display↑`redraw` = *Children*↑`redraw`
Display↑`changes` = *Children*↑`changes`

Children → *Display* $Children_1$

∀`a`. *Display*↓`a` = *Children*↓`a`
$Children_1$↓`offset` = *Children*↓`offset`
$Children_1$↓`visible` = *Display*↑`visible`
$Children_1$↓`redraw` = *Display*↑`redraw`
$Children_1$↓`changes` = *Display*↑`changes`
∀`a`. *Children*↑`a` = $Children_1$↑`a`

Children → ε

Children↑`display` = *Children*↓`offset`
Children↑`covers` = `[]`
Children↑`redraw` = *Children*↓`redraw`
Children↑`changes` = *Children*↓`changes`

Figure 5.15: The Update Attribute Grammar.

```
update :: Display -> (Display,[Area])
update display = (s_display,s_redraw)
 where
 (s_display,_,s_redraw,_) = rd (0,0) [] display
 ud :: Coord -> [Area] -> [Area] -> [DisplayChange] -> Display ->
               (Display, [Area], [Area], [DisplayChange])
 ud i_offset i_visible i_redraw i_changes image@(Image imid _ _) =
  let (match, mismatch) = span (==imid.fst) i_changes
      (image', redraw') =
       applyChanges match image i_offset i_visible c_s_visible i_redraw
      Image _ rpos children = image'
      apos = rpos `addArea` i_offset
      visible = intersectArea apos i_redraw
      c_i_offset = topleft apos
      c_i_visible = visible
      c_i_redraw = redraw'
      c_i_changes = mismatch
      (c_s_display, c_s_visible, c_s_redraw, c_s_changes) =
       uc c_i_offset c_i_visible c_i_redraw c_i_changes children
      s_display = Image imid rpos c_s_display
      s_covers = visible
      s_redraw = c_s_redraw
      s_changes = c_s_changes
  in (s_display, s_covers, s_redraw, s_changes)
 uc :: Coord -> [Area] -> [Area] -> [DisplayChange] -> Children ->
               (Display, [Area], [Area], [DisplayChange])
 uc i_offset i_visible i_redraw i_changes (Children display children) =
  let d_i_offset = i_offset
      d_i_visible = i_visible
      d_i_redraw = i_redraw
      d_i_changes = i_changes
      c_i_offset = i_offset
      c_i_visible = d_s_visible
      c_i_redraw = d_s_redraw
      c_i_changes = d_s_changes
      (d_s_display, d_s_covers, d_s_redraw, d_s_changes) =
       ud d_i_offset d_i_visible d_i_redraw d_i_changes display
      (c_s_display, d_s_covers, c_s_redraw, c_s_changes) =
       uc c_i_offset c_i_visible c_i_redraw c_i_changes children
      s_display = c_s_display
      s_covers = c_s_covers
      s_redraw = c_s_redraw
      s_changes = c_s_changes
  in (s_display, s_covers, s_redraw, s_changes)
 uc i_offset i_visible i_redraw i_changes NoMore =
  (i_offset, [], i_redraw, i_changes)
```

Figure 5.16: The Update Function.

access to a screen image through a connection to SM (§5.2.1). Mouse and keyboard input is directed to the relevant Gadget (§5.2.2). Gadgets can be composed just like Components, specifying their layout as well as wire connections (§5.2.3). Images can overlap on the screen (§5.2.4). Images can be dynamically created (§5.2.5) but cannot be destroyed. This is not solely an issue of removing an image from the display. It requires the Components that implement the interface part to be removed as well. As Components can only terminate themselves, removing a Gadget requires the Gadget's cooperation. Termination of Components is discussed in Chapter 3, *"Concurrency"*. Neither problem is insurmountable, but no time has been spent on implementing solutions to either.

Execution speed

The system is not as fast in execution as an imperative equivalent. It is difficult to compare without the same program written in functional and imperative languages, but screen update typically takes an order of magnitude more in our system than we would expect to see in an imperative language. However, it is certainly fast enough to be useable. Implementing the primitives for use in a compiler for a functional language instead of the interpreter we use would bring additional speed increases.

Efficiency brought about by single traversal

To measure the efficiency gained by introducing SM operations that traverse the data structure once for many changes, we altered SM so that it applied changes one at a time (updating the display after each change) then measured the number of reductions, cells and time taken to adjust the layout of a set of Gadgets. The program used in this test was the Escher Tile program of Chapter 6. A re-layout of Gadgets was triggered by reprogramming a Gadget button to grow when clicked. The top-most *Choice* button was triggered to enlarge, causing eighteen other Gadgets to be moved to accommodate its new size. The results are given in the table in Fig. 5.17. A similar operation involving twenty-five Gadgets in the Explode program (also from Chapter 6) revealed even better reductions (Fig. 5.18). Clearly the improvement depends on the number of Gadgets affected, and varies according to the exact layout chosen, but these results give some idea of the magnitude of the improvement. The improvement might not be as much as expected given that, for example, in the second test the screen was updated twenty-five times less often, and

yet the improvement was only a factor of between three or four. The reason is that there is far more work to be done in one update than in each of the twenty-five updates.

Method	Reductions	Cells	Seconds
Multiple traversals	236009	722597	10
Single traversal	138910	363922	6
Ratio	58.9%	50.4%	60.0%

Figure 5.17: An Efficiency Comparison (Escher).

Method	Reductions	Cells	Seconds
Multiple traversals	278020	815363	9
Single traversal	66627	169069	3
Ratio	24.0%	20.7%	30%

Figure 5.18: An Efficiency Comparison (Explode).

5.9 Further Issues

5.9.1 Gadget Attributes

Gadgets have *attributes* such as width, height and colour. With many attributes to set, passing them all as arguments leads to clutter and confusion. Even with the attributes of width, height and colour only a button application might be:

```
button (op w) Click 50 60 c
```

where "`Click`" is the message the button sends when clicked, "`50`" is the width, "`60`" is the height and "`c`" a colour. With only three attributes the button has five parameters, and it must be remembered which parameter controls each one.

Instead, we define an algebraic type that specifies all a Gadget's parameters, and assume that the Gadget has a *default value* of these *attributes*. We can pass the gadget a function to alter the default value. Take the `button` for example:

```
button' :: (ButtonAttributes -> ButtonAttributes) ->
                                                Out m -> m -> Gadget
```

This is the *default Gadget*. It must be applied to some attribute-changing function before it becomes a Gadget. To leave the attributes at the default, we apply the default Gadget to the function `id`. We define the *unprimed* `button` to be equal to the default button:

```
button = button' id
```

Alternatively, *attribute modifying functions* may be composed and used to alter the attributes. For example, a button of default width and colour, but with a height of 100 and a *momentary* action (ie. the button pops out again when the mouse button is released):

```
button' (height 100.buttonMomentary) (op w) Click
```

Some aspects of a Gadget's operation are peculiar to the type of Gadget and have no sensible default value (eg. for a button, the output wire and message), so these values are separate parameters to the Gadget. Some Gadget attributes are specific to a particular Gadget (eg. `buttonMomentary`) but can have a default value. Others are common to many Gadgets (eg. `width` and `height`), and we would like to use the same name regardless of the particular type of Gadget. Here the *type classes* of Haskell prove useful [HHJW94]. A class is defined for each sort of attribute that is common to more than one Gadget, and the attributes type for such a Gadget is an instance of that class. For example, there is a `HasWidth` class, of which `ButtonAttributes` is an instance. Each type of Gadget that can have its width set defines its own overloaded version of the modifier `width` to alter its particular attribute datatype. Now `width` can be used on the attributes of any Gadget that accepts it. Modifiers for attributes specific to a particular Gadget are not overloaded.

5.9.2 Managing State in Components

In several of the Components we have defined for this system, and most notably the Screen Manager Component, a large state value consisting of multiple parts is used. This state is maintained as a parameter in a tail-recursive function. The individual parts of the state are unpacked using pattern matching, then repacked, replacing

whichever parts are to change, in the recursive function. This adds extra plumbing work to the program. For example, the main part of the Screen Manager definition is[3]:

```
smloop (display,uncl,smrr) =
    rx [
        froms smrr $ \smrq smrs ->
            service smrq smrs (display,uncl,smrr) $
                \(display',uncl',smrr') ->
                    let (suc,display'') = update_and_redraw display' in
                    tx screenCo suc $
                    smloop (display'',uncl',smrr'),
        from mouseEv $ \e -> case e of
            MouseClick x y b ->
                case click_dest display (x,y) of
                    None -> smloop (display,uncl,smrr)
                    Yes ((x1,y1),(cx,cy),(ca,uca)) ->
                        ca (cx,cy) b $
                        smloop (display,Yes ((x1,y1),uca),smrr)
    ]
```

Programs would be tidier and therefore easier to read and understand if parts of the state were only mentioned when they were to be altered.

Using state access functions

One way to cut down the plumbing is to use state access functions. So for SM definition above we could define the following state access functions:

[3] simplified here for brevity

```
type SMState = (Display, Maybe (Coord,ClickAction),
                        [InOut SMRequest SMResponse])

getDisplay :: SMState -> Display
getDisplay (d,_,_) = d

getUnclick :: SMState -> Maybe (Coord,ClickAction)
getUnclick (_,u,_) = u

getGadgetCons :: SMState -> [InOut SMRequest SMResponse]
getGadgetCons (_,_,g) = g

setDisplay :: Display -> SMState -> SMState
setDisplay d (_,u,g) = (d,u,g)

setUnclick :: Maybe (Coord, ClickAction) -> SMState -> SMState
setUnclick u (d,_,g) = (d,u,g)

setGadgetCons :: [InOut SMReqeust SMResponse] -> SMState -> SMState
setGadgetCons g (d,u,_) = (d,u,g)
```

and the Component can now be defined:

```
smloop state =
    let smrr = getGadgetCons state in
    rx [
        froms smrr $ \smrq smrs ->
            service smrq smrs state $ \state' ->
            let display' = getDisplay state'
                (suc,display'') = update_and_redraw display' in
            tx screenCo suc $
            smloop (setDisplay display'' state'),
        from mouseEv $ \e -> case e of
            MouseClick x y b ->
                let display = getDisplay state in
                case click_dest display (x,y) of
                    None -> smloop state
                    Yes ((x1,y1),(cx,cy),(ca,uca)) ->
                        ca (cx,cy) b $
                        smloop (setUnclick (Yes ((x1,y1),uca)) state)
    ]
```

A slight improvement over the original. The overhead of having to define the state access functions is probably more than the improvement in the Component definition. The advantage is that we can add new items to the state without having to alter the whole of the Component definition. Just the state accessing functions need to be altered.

Record types

The situation would be better if the language had record types. Essentially this would give us the state accessing functions for free, perhaps with some additional syntactic sugar as well[4]:

```
record State = {display :: Display, unclick :: Maybe (Coord, ClickAction),
                gadgetcons :: [InOut SMRequest SMResponse]}

smloop {smrr:state} =
    rx [
        froms smrr $ \smrq smrs ->
            service smrq smrs state $ \state' ->
            let (suc,display') = update_and_redraw {display:state'} in
            tx screenCo suc $
            smloop {state':display = display'},
        from mouseEv $ \e -> case e of
            MouseClick x y b ->
                case click_dest {display:state} (x,y) of
                    None -> smloop state
                    Yes ((x1,y1),(cx,cy),(ca,uca)) ->
                        ca (cx,cy) b $
                        smloop {state:unclick = (Yes ((x1,y1),uca))}
    ]
```

Now we can extend the state by adding new fields to the record. We don't have to generate new state accessing functions.

Mutable state variables

Another alternative would be to use mutable state variables [JP93]. This is updatable state, held and accessed by monad in a thread.

[4] We are using an imaginary syntax here

```
newMutVar (initDisplay :: Display) $ \displayV ->
newMutVar (initUnclick :: Maybe (Coord, ClickAction)) $ \unclickV ->
newMutVar (initGadgetCons :: [InOut SMRequest SMResponse]) $ \gadgetconsV ->
smloop where
smloop =
    readVar gadgetconsV $ \smrr ->
    rx [
        froms smrr $ \smrq smrs ->
            service smrq smrs displayV gadgetconsV $
            let suc = update_and_redraw displayV in
            tx screenCo suc $
            smloop,
        from mouseEv $ \e -> case e of
            MouseClick x y b ->
                click_dest (x,y) displayV unclickV $
                smloop
    ]
```

Now the definition is shorter because the auxilliary functions `service`, `update_and_redraw` and `click_dest` access and update the state directly, so don't need to pass back new values.

Mutable state for increased efficiency in SM

As well as improving the readability of the definition of SM, mutable state could also increase the efficiency There is only one copy of the display data structure in existence at any one time. Currently we generate new values for any parts we wish to change, and they occupy new cells in the heap. The old values are left in the heap. If the display data structure were held in mutable variables, it could be updated in-place, reducing the amount of garbage collection that has to be done.

5.10 Further Work

5.10.1 Mouse could carry messages

Instead of delivering click messages to images, the mouse could be *armed* with messages to deliver. This could be used, for example, in implementing drag-and-drop style actions, where an icon representing a file in one window is dragged to a text-editor to instruct the editor to load the file. The mouse message could be the contents of the file. We could alter the Screen Manager's handling of mouse input

to simulate this. A problem is that programs should specify the type of messages conveyed by the mouse, but the type would be specified by the Screen Manager.

5.10.2 Standard treatment of input-focus

Highlighting of a Gadget that has the input focus and controlling when a Gadget claims the input focus should be standard across all Gadgets. Presently, the highlighting is implemented within each Gadget separately. We could add an extra tag to each image in the Display data structure, indicating that it can receive keyboard input. SM could then control how input-focus is chosen, and still direct key presses only to Gadgets that require them. This method is similar to that used by window managers such as The X Windows System. We chose a method that was simpler to implement as it did not require SM data structure to be changed.

5.10.3 Gadget layout in a limited space

In the current system, messages relating to the allocation of screen space propagate up the hierarchy of Gadget compositions. Gadget's specify the amount of space they require, and their parents must make room for them. There is no way to work in the opposite direction, starting with a certain Gadget size and fitting its children into the space by altering their size.

Layout stretchiness and squashiness

If Gadgets specified a size along with values indicating relative willingness to squash or stretch horizontally and vertically, then sizes could be adjusted to fit the space available. The gaps between Gadgets in a composition should be attributed with squashiness and stretchiness as well so that the programmer has control over where surplus space goes.

5.10.4 Graphical Composition

The layout of Gadgets within a composition is specified using layout combinators. The picture in a directly defined Gadget is defined by the drawing function that provides a list of screen commands to render the picture. There is nothing in the Gadget system to help the programmer construct the drawing function.

What's in a Gadget?

What is needed is a set of combinators for simple graphical elements such as lines and circles. Functions compose these elements to form new elements. Other functions perform transformations on a graphic, such as scaling or rotation. The graphic elements are described by an algebraic datatype:

```
type Vector = (Coord, Coord)

data Graphic id = GPoint
                | GLine Vector
                | GRectangle Vector
                | GCircle Int
                | GText String
                | GHSpace Int
                | HVSpace Int
```

where `GHSpace w` is an invisible element with no height but width `w`. These elements may have some attributes applied to them. Each attribute changes some elements and does not affect others:

```
                | GColour Colour (Graphic id)
                | GFilled (Graphic id)
                | GLineWidth Int (Graphic id)
                | GFont String
```

or transformations applied to them:

```
                | GRotate Float (Graphic id)
                | GScale (Float,Float) (Graphic id)
```

Graphical elements have a *stretchiness* (their willingness to grow in size) and a *squashiness* (their willingness to shrink in size) that are used when fitting a graphic into an image. These attributes apply to horizontal and vertical directions independently:

```
                | GStretchy (Float,Float) (Graphic id)
                | GSquashy (Float,Float) (Graphic id)
```

Elements are composed by placing them beside or above one another:

```
                | GBeside [Graphic id]
                | GTop [Graphic id]
                | GBottom [Graphic id]
                | GAbove [Graphic id]
                | GLeft [Graphic id]
                | GRight [Graphic id]
```

where `GAbove` centers, `GLeft` left aligns and `GRight` right aligns. Infix operators, `>-<` and `>|<`, corresponding to horizontal and vertical layout, are provided to give a more convenient description of a picture:

```
(>-<),(>|<) :: Graphic id -> Graphic id -> Graphic id
(>-<) l r = GBeside [l,r]
(>|<) a b = GAbove [a,b]
```

Finally, elements may be given an identifier so that they can be altered at a later stage:

```
                | GID id
```

The function to alter a graphic:

```
alterG :: Graphic id -> id -> (Graphic id -> Graphic id) ->
                                    (Graphic id, DrawFun, DrawFun)
```

takes the graphic to change, the id. of the element to change and a function to alter the element, and returns the altered graphic, an update drawing function and a drawing function. See §5.5.2 of Chapter 5, *"Screen Management"* for a description of the purpose of update drawing functions.

A function converts a graphic to a drawing function that fills a Gadget's image with the graphic according to the stretchiness and squashiness of each of the elements.

```
render :: Graphic id -> DrawFun
```

This drawing function makes use of the *area to be redrawn* parameter to redraw only those elements that have been exposed. For example, the picture in a text-editor Gadget might be described by a sequence of `GText` elements composed with `>|<`. If the last line of the text-editor Gadget is exposed, the drawing function generated by `render` is able to calculate that only the last line of text is visible, so returns screen commands to draw that only.

Integrating Gadget layout into graphical composition

The calculations involved in positioning the elements of a graphic are similar to those involved in Gadget layout. The difference is that a Gadget can choose to

change its size, affecting the layout, whereas graphical elements are *lifeless* although they can stretch or squash.

Despite this difference, Gadget layout could be integrated into the composition of graphical elements:

```
                | GGadget Gadget
```

Now our counter of Chapter 6, *"Applications"* is expressed:

```
counter :: Gadget
counter =
    wire $ \a ->
    let b = button' (picture uparrow.width 30.height 30) (op a) (+1)
        d = display' (displayInput (ip a).width 30.height 30) 0 in
    initGadget (render (GGadget b >-< GGadget d)) $
    end
```

where `initGadget` initialises the `counter` Gadget and sends the graphic (converted to a drawing function by `render`) to SM. Now that a graphic contains Gadgets that can change their size dynamically, `render` must also create a *layout manager* Component to monitor their sizes and alter their position and the rest of the graphic accordingly. To do this, the type of `render` must be changed to give the rendering function access to the Gadget's SM connection and allow it to change the layout connection. We will not consider the implementation of `render` any further here.

The integration of Gadget layout in graphical composition holds other implications for the structure of Gadget programs too. Altering a graphic that contains Gadgets may require some Gadgets and the corresponding layout manager Component to be terminated. At present there is no simple and common way to terminate other Components (see §7.2.6). This issue would need to be addressed before Gadget layout could be integrated with graphical composition.

As well as removing the need for two different types of layout specification, this integration would bring other benefits as well. Composed Gadgets could be integrated with graphics. For example, consider a Web Browser Gadget. This Gadget contains a composition of graphical text elements with *hyperlink* Gadgets containing highlighted text that, if clicked, take the browser to a new web page. A function could convert HTML text contained in a `String` to a graphic value.

Comparison with Haggis

The contents of top-level windows in Haggis are described using a graphical data structure similar to the one suggested here. The widgets within a window do not

have separate windows as Gadgets do, but a description of their picture is composed into the top-level window's picture. A widget's picture is changeable by virtue of the fact that the picture value is held in a mutable variable so that the process governing a widget can alter it. A screen updating process regularly *walks* each window's picture to check for changes, and updates the screen accordingly.

5.10.5 More manager Components

It was always intended that the OS Component would give access to a variety of resources by allowing Components to connect to various *managers*. At present there is only one manager, for the screen. Two new managers that might have been implemented (time allowing) would give access to the filing system and a clock.

File manager

The file manager allows Components to read and change the contents of files. A Component can sends the file manager requests. It can request the contents of a file, the "file" itself which is later returns (possible changed), or send a function to alter a file. If a Component makes a request relating to a file currently "held" by another Component, the request is not serviced until the holder returns the file. Consider a simple example. A *text-editor* Gadget requests a file that contains the source code of a program. While the editor is working on the file, a *compiler* Component requests the contents of the same file. The file manager delays sending the contents of the file to the compiler until the text-editor returns a new version of it. This mechanism is similar to that discussed in [HC95].

```
type FileName = String

data FMRequest = FMReadFileContents FileName
               | FMGetFile FileName
               | FMReturnFile FileName String
               | FMChangeFile FileName (String -> String)
```

The file manager answers requests with responses:

```
data FMResponse = FMFileContents FileName String
                | FMFile FileName String
                | FMError String
```

Both `FMFileContents` and `FMFile` return the same information, however conceptually the requesting Component "holds" the file when the later is sent to it, and must

return it later with a `FMReturnFile`. An error may be returned in circumstances such as the Component requesting a file that does not exist.

Timer manager

Apart from giving programs access to the time of day, it is sometimes useful to be able to put off an action for a certain amount of time. One example of this can be found in the multiple counters program of Chapter 6, *"Applications"*. When this program is in a certain mode a constant stream of messages is sent from a Component, as fast as they can be produced. This has the unfortunate effect of stopping another message getting through because the task of sending and receiving this stream of messages takes over the program. Besides, it is undesirable that the rate of flow of the stream of messages is determined by the speed at which the program evaluates.

To address this and other similar problems, access to a clock could be provided through a timer manager. We assume that time can be represented by a value of type `Time`. The exact representation is not important here. The `clock` Component (a primitive I/O device) accepts messages of type:

```
type Action = Component -> Component


data ClockRq = ClockAlarmSet (Time -> Time) Action | ClockTellTime

data ClockRs = ClockLaterAlarm Time Action
             | ClockAlarm Action
             | ClockTimeIs Time
```

The `clock` is connected to a *timer manager* Component (TM for short). The TM keeps a list that consists of pairs of an alarm time and action to perform at the alarm time. The list is kept in order of *next to occur* first.

To set the alarm clock, the TM sends a `ClockAlarmSet` message to the `clock` device along with a function that when applied to the current time returns the alarm time (so an alarm can be set for `n` seconds time by adding `n` to the time), and an action to perform at that time. If the `clock` does not have an alarm already set, this alarm is set. If the `clock` already had an alarm set, the *earlier* of the two alarms (the one stored and the one just sent) is held in the clock, and the *later* is returned to the TM in a `ClockLaterAlarm`. The returned one is inserted into the correct place in the list held in the TM. When the time reaches that of the alarm set, the

`clock` sends the corresponding action to the TM in a `ClockAlarm` message, where the TM performs the action. The TM resets the `clock`s alarm with the head of its list.

The TM may also request the current time (on behalf of another Component) by sending a `ClockTellTime` message. This results in a `ClockTimeIs` message from the `clock`.

The reason we have two separate Components, `clock` and the timer manager, is that the clock is a primitive Component defined in an imperative language that has access to the clock hardware, whereas the timer manager is defined in the functional language. This way we keep the amount of imperative programming to a minimum.

We could reduce the imperative programming even further by simplifying the operation of the `clock` to sending `ClockTimeIs` messages every second. The disadvantage with this solution would be that the timer manager Component must process a message every second. The former solution requires only a few messages to set up and execute each timed action.

5.11 Summary

This chapter divides into two sections. In the first section we have:

- defined our requirements for screen management;
- described a screen manager Component that stands between the screen display hardware and Gadget programs;
- introduced Gadget Components;
- listed and described the *request* and *response* messages communicated between a Gadget and the Screen Manager;
- explained the mechanism of Gadget layout;
- revealed the internal state of the Screen Manager and the operations performed on it;
- defined these operations using *attribute grammars*;
- evaluated the success of the work in respect of our initial aims.

In the second section we have considered some further issues:

- coping with Gadgets that have many attributes, and with groups of Gadgets that have common attributes;
- managing state in Components
- a library of graphical composition combinators for describing the picture in a Gadget, and describing the layout of Gadgets;
- plans for more manager Components.

Chapter 6

Applications

6.1 Introduction

In this chapter we look at various example Component and Gadget programs that illustrate some of the benefits and problems of the system. The aim is to introduce various aspects of the system through examples. We begin with a simple text-only example that demonstrates the definition, composition and re-use of Components. Then we move on to graphical programs, using the screen manager Component, and introducing Gadgets, their layout and attributes. A program displaying multiple views of data uses a client/server structure. A simple board game introduces purpose-written Component combinators and the use of concurrency in the non-GUI parts of a program. The Escher Tile Program of Chapter 2 provides a comparison of programming style with the Fudgets and Concurrent Clean systems, and a context in which to discuss Radio Buttons. A couple of examples demonstrate features of screen management: the relationship between parent and child Gadgets is put to use in a Viewport Gadget; automatic layout and resizing of Gadgets feature in a filename entry Gadget. Lastly, we consider a Fudget Simulator in the Gadgets System, and a Gadget Simulator in the Fudgets system.

6.2 Hamming Number Generator

Motivation

The *Hamming problem* [Dij76, Hoa80] was used as an early test of concurrent processing in Embedded Gofer [Wal95]. We present the same problem here for compar-

ison, re-written in the Component Gofer style. It shows the Components definition, wiring and composition. Later we discuss the re-use of Components we have created for this example.

Problem

The problem is to combine multiplication processes to generate the sequence of Hamming Numbers (with only 2, 3 and 5 as prime factors) in ascending order (see [Hoa80]).

For the purposes of his concurrency test, Wallace simulated laziness by sending messages demanding the next piece of data between processes. For example, the colon process demands the next number from the sorted merger. This was necessary because the scheduling scheme in Embedded Gofer simply gave processes a static priority ordering. We do not simulate laziness in our example because the scheduler in Component Gofer picks processes according to the number of messages waiting for them, so the scheduler will already choose processes in an order that minimises the number of messages queued in wires.

Solution

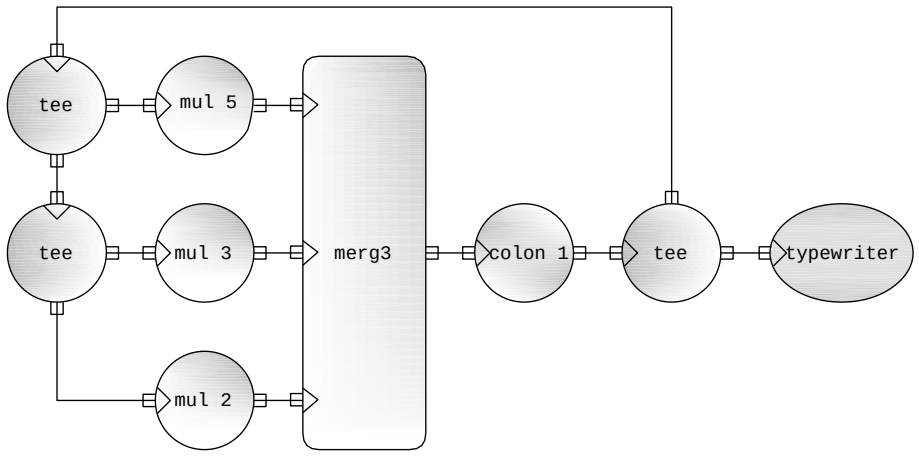


Figure 6.1: The Components of the Hamming Number Generator.

Wiring the Components of the Hamming Generator At the top level of the hierarchy, the main Components of the Hamming generator (shown in Fig. 6.1) are wired together:

```
hamming :: Component
hamming =
    wires 11 $ \[a,b,c,d,e,f,g,h,i,j,k] ->
    spawn (merg [(ip a),(ip b),(ip c)] (op d)) $
    spawn (colon 1 (ip d) (op e)) $
    spawn (tee (ip e) (op f) (op g)) $
    spawn (typewriter (ip f)) $
    spawn (tee (ip g) (op h) (op i)) $
    spawn (mul 5 (ip h) (op c)) $
    spawn (tee (ip i) (op j) (op k)) $
    spawn (mul 3 (ip j) (op b)) $
    spawn (mul 2 (ip k) (op a)) $
    end
```

As the wires all carry messages of the same type (integers) we can obtain them all together in one list, using the `wires` function. The ends of each wire are connected to pins of a Component, for example wire `f` is connected to an input pin on the `typewriter` Component, and to an output pin on the `tee` Component. The `hamming` Component simply has the job of creating and connecting these Components together. Once this is complete, it `end`s. The Components it has created are left wired together and appear from the outside to be one Component. The `hamming` Component has effectively set up a static network — we don't require dynamic reconfiguration of Components for this example.

The Components `tee` and `typewriter` are defined in the library. The `tee` Component has one input that duplicates received messages on its two outputs. Messages sent to the `typewriter`, which must be of class `Text`, are displayed on `stdout`. The `typewriter` is an I/O device Component.

The `mul` Component, that multiplies integers received by a set factor and passes them on, is one application of a generalised *mapping* Component found in the library. The Component, `mapC`, is like `map`, but operates on streams of messages in wires, rather than a list.

```
mul :: Int -> In Int -> Out Int -> Component
mul n = mapC (*n)
```

The `colon` Component (analogous to 'lazy cons' in Henderson's solution [Hoa80]) is similar to the library operator `(:)`. It feeds one message through its output pin before copying through any messages received on its input to its output. Our definition of `colon` is polymorphic in the type of messages it passes through, and might be useful in other situations.

```
colon :: a -> In a -> Out a -> Component
colon n i o =
    claim i $ tx o n $ c
    where
    c = rx [ from i $ \n -> tx o n $ c ]
```

The *sorted merger* Component `merg`[1] is the most complicated Component in this example. Its function is to merge messages from its list of inputs into ascending order, removing duplicates. The merger must have a message from each input pin before it can test to see which is the lowest, so messages are buffered inside the Component (in the buffer `bs`).

```
merg :: Ord a => [In a] -> Out a -> Component
merg is o =
    sequence (map claim is) $ m' (map (const []) is)
    where
    m' bs =
        if (not. any null) bs then
            let hs = map head bs
                l = minimum hs
                bs' = map (removeheadequal l) bs
                removeheadequal e (h:t) = if e==h then t else (h:t)
            in tx o l $
            m' bs'
        else
            rx [ froms (zip is [1..]) $ \n i -> m' (appendto i n bs) ]
                where
                appendto 1 n (b:bs) = (b++[n]):bs
                appendto i n (b:bs) = b:(appendto (i-1) n bs)
```

We have not made any effort to produce an efficient implementation — items are added to the end of the buffer by a singleton append operation! The production of a more efficient version is left as an exercise for the reader (see [Oka95] for some help).

The Hamming program is shown in operation in Fig. 6.2.

Evaluation

Wiring Knowing the structure of processes required, the Hamming Number Generator was simple to construct. The wiring of Components in `hamming`, although simple to define, is not as clear as a diagram of the processes. Wiring together a

[1] `merg` is spelt without the final 'e' to avoid conflict with the `merge` function in the standard prelude

```
GG> launch hamming
1 2 3 4 5 6 8 9 10 12 15 16 18 20 24 25 27 {Interrupted!}

(2579 reductions, 6591 cells)
GG>
```

Figure 6.2: The Hamming Number Generator Program in Operation.

composition is a low-level task, and prone to errors. It is difficult to construct combinators to ease the job because Components are so varied in their type. A program to enable the plumbing to be performed graphically would be useful.

In the `merg` Component some messages are buffered internally as a way of changing the order in which they are processed. In some situations this kind of internal buffering could upset the scheduling because messages held within a Component will not be counted when the scheduler is choosing the next Component. With a more sophisticated approach to scheduling, this would be less likely to be a problem.

Re-usable Components The multiplier is a good example of a re-usable Component. It is defined in terms of the general `mapC` Component, but could easily have been defined directly:

```
mul :: Int -> In Int -> Out Int -> Component
mul n i o = claim i $ mul'
    where
    mul' = rx [ from i $ \v -> tx o (v*n) $ mul' ]
```

Compare this with the version of the multiplier from Embedded Gofer:

```
mul :: Int -> SoftProc Pid Msg Stt
mul i (SMsgI Merge NextPlease: msgs) =
    GetStt: mul i msgs
mul i (State s: msgs) =
    UpdStt (movePtr i):
    SMsgO Merge (Val (i * readVal i s)):
    mul i msgs
```

This has extra functionality to simulate laziness with *demand* messages (`NextPlease`), and the list of output values is stored in a global state value. Simplifying the definition to obtain a functionality of a similar level to the Component definition, we get:

```
mul :: Int -> SoftProc Pid Msg Stt
mul n (SMsgI Colon (Val v):msgs) = SMsgO Merge (Val (v*n)):mul n msgs
```

We can see that the function is a stream processor, operating on an input list of messages (tagged with the address of the sender, `Colon`, and type of message, `Val`), and resulting in a list of output messages (tagged with the address of the receiver, `Merge`, and the type of message, `Val`).

In the Component Gofer version, we have no need to specify directly the sender or receiver of messages, but instead messages are sent or received via pins that will later be connected to sender and receiver. In addition, if only one type of message is sent per pin, then messages can be sent as a plain value without a tag indicating the type. Both these issues make Components easier to re-use.

Henderson's lazy streams version of the program [Hoa80] is similar in that streams are passed in and out of the functions representing processes as parameters.

Wallace later devised an alternative way to achieve this using type classes and overloading, but only for static networks of processes [WR94].

The mapping Component (Fig. 6.3) used in the Hamming program is useful in many situations. It can be used to translate messages from one type to another, for example the `mapC show` Component converts messages of class `Text` into a message of type `String`.

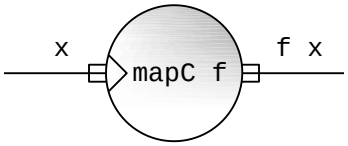


Figure 6.3: The `mapC` Component.

6.3 Simple Counter

Motivation

The counter is a popular example in papers describing functional GUIs [FJ95, HM95, VTS95]. It illustrates the basic technique of Gadget composition, where a new Gadget is formed by connecting several others. This is similar to Component composition, but the layout of the Gadgets is specified as well.

Problem

A button and an integer display are connected: clicking the button increases the number in the display. A suitable display is shown in Fig. 6.4.

Figure 6.4: The Counter Example.

Solution

The program consists of a composition of two library Gadgets:

```
counter :: Gadget
counter =
    wire $ \a ->
    let b = button' (picture uparrow.width 30.height 30) (op a) (+1)
        d = display' (displayInput (ip a).width 30.height 30) 0
    in b <-> d
```

The button is passed the output end of a wire `(op a)` and the display is passed the input end `(ip a)`. The screen layout of these Gadgets is defined in the last line: the `<->` combinator specifies that button and display should be placed side-by-side horizontally.

The button sends a function to the display that is applied to the current value stored to yield a new value. We could have programmed the display to respond to simple `Click` messages instead, incrementing the number itself, but by passing a function we keep the display more general. We can add more buttons to the counter by adding buttons that emit new functions. For example, a *Clear* button would send the function `const 0`. Multiple Components may send messages on the same wire[2] so we don't even need an extra wire to add a clear button to the counter. The extended counter is shown below.

[2]though only one Component may receive from a wire

```
counter :: Gadget
counter =
    wire $ \a ->
    let countbut p f = button' (picture p.width 30.height 30) (op a) f
        d = display' (displayInput (ip a).width 30.height 30) 0
    in countbut uparrow (+1) <-> d <-> countbut clear (const 0)
```

The Gadgets in this example have some common attributes, `width` and `height`. Couldn't we abstract the common attributes away? For example:

```
counter :: Gadget
counter =
    wire $ \a ->
    let b = button' (picture uparrow.c) (op a) (+1)
        d = display' (displayInput (ip a).c) 0
        c = width 30.height 30
    in b <-> d
```

Unfortunately we cannot do this because the `height` and `width` attribute modifiers are *overloaded*. A different instance is used for each type of Gadget. If the Gadgets in this example had been the same then we could have factored out the common sizing attributes. There is a way we could avoid this problem. If all Gadgets take the same attributes and just ignore the ones that do not apply, then the attribute modifiers do not need to be overloaded. Unfortunately if we do this we lose the ability to check statically that we are applying the Gadget to appropriate attributes.

Evaluation

Because communication between Components is defined at the level at which they are composed, and not within the definition of each, we can see immediately which parts interact. The events associated with the X server such as window exposure and redrawing are completely hidden. As a result the counter program is simple and uncluttered.

Passing functions as messages has made this program simpler to extend. We have used this technique in many of the examples in this chapter.

The Gadget counter program is slightly shorter that the Fudget equivalent (see Page 27), but this is mostly because Gadget buttons and displays have more functionality than their Fudget counterparts. In most respects the programs are very similar. It could be argued that it is necessary to remember a lot of different types of combinators for use in Fudgets. On the other hand, the Gadget version has the addition of wires in the definition. We shall see in later examples that as programs get

more complex, wires are more flexible than combinators, because arbitrary networks of processes can be set up, and not just rigid hierarchies.

6.4 Resizing Filename Entry Box

Motivation

The Gadget layout mechanism provides for Gadgets that change size during program operation. This example demonstrates the use of resizing Gadgets.

Problem

A GUI is required to allow the user to enter a filename in a text-entry area on the screen. Clicking a *Clear* button should delete any filename already entered. Clicking an *Ok* button submits the entered filename to the program. We want to be able to view the whole filename, even if it is longer than the initial size of the text-entry area.

Figure 6.5: The File Entry Gadget.

Solution

The `editor` library Gadget, which is used for text entry of a class of `Enterable` types, resizes when the text being entered becomes too large to display. Most of the function of the file entry box is managed by this Gadget. We present the part of the `editors` definition that does the resizing in Fig. 6.7. The `editor` is configured to emit a new string every time a key is pressed, and this is fed into a `memory` Component. The *Ok* button is connected to the `memory`'s *trigger* input, so when clicked, the last stored filename in the `memory` is emitted from the file entry Gadget. The *Clear* button is connected to send a function to the `editor` to set its contents to an empty string. The editor is configured to copy received changes to its contents through to its output, so pressing *Clear* results in an empty string stored in the `memory` also. The file entry Gadget definition is given in Fig. 6.6.

Inside the definition of the `editor` Gadget, a change of size is requested using the `setSize` library function. The `editor` remembers its width in an argument `sx` so that it can know when the text it is displaying has become too long to fit. The section of the `editor` definition that alters the size is given in Fig. 6.7, simplified by excluding details irrelevant to the resizing. The new size (`sx'`) is equal to the width of the text if the text is wider than the old size, or else is equal to the old size. The `setSize` function extracts from the Gadget state the connection to the Gadget's layout manager, and sends the new size to the manager (see §5.6 for a description of the layout mechanism). The hierarchy of layout managers alter the sizes and positions of Gadgets as necessary to accommodate the new Gadget size.

```
fileentry :: Out String -> Gadget
fileentry o =
  wire $ \cb -> wire $ \okb -> wire $ \fn ->
  let fo = "*-lucida-medium-r-*-120-*"
      co = col "black"
      clrbut = button' (picture (text fo co "Clear")) (op cb) (const "")
      okbut = button' (picture (text fo co "Ok")) (op okb) ()
      fnentry = editor' (editorOutput (op fn).editorInput (ip cb).
                         editorInit "".editorSendOnAny.
                         editorCopyThrough) in
  spawn (memory (ip okb) (ip fn) o) $
  clrbut <-> fnentry <-> okbut
```

Figure 6.6: The Filename Entry Gadget.

```
editorloop s sx =
    rx [
        fromSM $ \r -> case r of
            SMKeyPress k ->
                let (s',changed) = prokey s k  -- keypress alters string
                    (tw,_) = textSize fn s' -- new string size
                    sx' = if tw > sx then tw else sx in
                when changed (
                    when (sx' /= sx) (setSize (sx',h)) $
                    txSM (SMDrawFun (df o f s'))
                ) $
                editorloop s' sx'
```

Figure 6.7: A Section of the `editor` Definition.

Figure 6.8: The File Entry Gadget After Resizing.

Evaluation

This example, though simple, illustrates the ease with which a Gadget may alter its size. It might be argued that most similar filename entry interfaces available commercially *scroll* the filename within the box if it becomes too long, but perhaps this is because it is too difficult to resize the box in other systems?

Resizing Gadgets in Viewports A resizing Gadget will not always cause other Gadgets to be moved. For example, a text-editor Gadget could be implemented by displaying the text in a Gadget that is large enough to fit the whole file (possibly larger than the screen), then displaying this Gadget in a viewport (see §6.8). When the user presses *Return* to start a new line in the text file, the text displaying Gadget grows one line taller. No other Gadgets have to move as it is the only Gadget contained in the vieport's image. This arrangement has a style similar to the lazy evaluation of a filter applied to a large list, because the text Gadget generates the whole of the text display, whilst the screen manager picks out only those parts that are visible to show on the screen. This is good because we do not need to consider what will actually be on display whilst programming the text Gadget. We can pretend that the screen is large enough to display the whole file.

6.5 Multiple Counters

Motivation

Here we consider a program in which some common data must be shared by a number of Components. The shared data is displayed in multiple views. If the data is changed through one view, the change is reflected in the other views. This is an important issue to cover because Components cannot freely access expressions that form part of the function describing another Component. The program was specifically devised at the Glasgow GUI Workshop (July 1995) as a test of functional GUI systems.

Problem

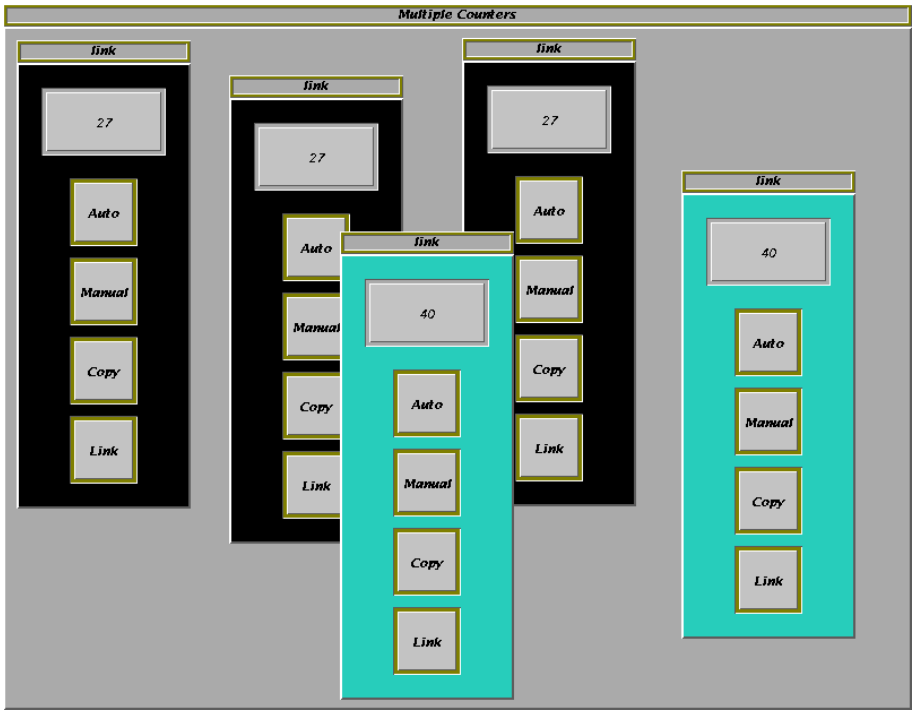


Figure 6.9: Multiple Counters in Operation.

The multiple counters are shown operating in Fig. 6.9. A counter consists of a window containing an integer display and four buttons, *Link*, *Copy*, *Manual* and *Auto*. The purposes of the buttons are as follows:

Link opens a new counter window that is *linked* to this one and has the same background colour as it. It displays the same number, and if the number is incremented in one window, it increments in the other.

Copy opens a new counter window that is *independent* and has a different background colour to it. Initially, the new window displays the same value, but incrementing the number in the new counter does not affect the number in the one that created it, and vice-versa.

Manual increments the number in the counter and in any counter *linked* to it.

Auto sets the counter into a mode in which it increments itself at regular and frequent intervals[3]. *Linked* counters also increment. The counter is returned

[3]the duration of the interval is determined by the speed of execution of the program only

to manual mode by clicking the *Manual* button.

Solution

The program is in Fig. 6.10 and Fig. 6.11. When a group of Components wish to share some data, one Component is designated to store the most up-to-date value. We call this the *server*. Any of the group of *clients* may send a function to the server, which it will use to update the value, then send a copy to each client. (There is only one copy of the data held in memory. When the server sends the value to each Component in the group, it actually sends a pointer to the value in memory, though this is transparent to the functional programmer.)

It would not be satisfactory for clients to send *new values* to the server. For example, suppose the server holds an integer, and each of the clients increments the integer from time to time. If two clients both decide to increment the number at approximately the same time, they will both add one to their copy of the current state and send it to the server. The server will receive two new values, but the value will only increment once. The problem is caused by the fact that there is no way to stop one of the clients sending a new value in between another client sending a new value and that value being posted to all clients. By sending a function instead we alleviate the problem. Each client can send a `(+1)` function. This method is reliable because the server will respond completely to any message sent to it before it goes on to process the next message.

The multiple counters in this example are a slightly simplified version of this scheme. Rather than sending a function to modify the integer state we send a token representing the function. For example, sending a `Manual` token is equivalent to modifying the state with a `(+1)` function, but also switches the server back to manual mode if it was in automatic mode. Using a token instead of a function enables us to state explicitly the operations the server is willing to perform on its state.

The program consists of two different types of Components. A *counter server* (`countserv`) keeps track of the value in one counter and all its links. It can be manually incremented, set into auto mode in which it increments itself, told to copy itself, or create itself a new link. The definition of `countserv` is given in Fig. 6.10.

The `when` function is defined in the library. It is like an `if` expression without the `else` part. If the condition is True, the functions of the second parameter are

```
data CountRq = Manual | Auto | Matic | Copy | Link
type Name = String

countserv :: Out Window -> Name -> Int -> Bool -> Component
countserv w id n a =
   wire $ \i ->
   let c' os n a s =
         rx [let m a' =
                  let n' = n + 1
                  in sequence [tx o n' | o <- os] $
                  c' os n' a' s
             in from (ip i) $ \rq -> case rq of
                Manual ->
                   m False
                Auto ->
                   when (not a) (tx (op i) Matic) $
                   c' os n True s
                Matic ->
                   if a then tx (op i) Matic $ m True else c' os n a s
                Link ->
                   wire $ \o ->
                   tx w (window (link id n (ip o) (op i)) "link") $
                   tx (op o) n $
                   c' (op o:os) n a s
                Copy ->
                   let nextId = (chr (s+97):id)
                   in spawn (countserv w nextId n a) $
                   c' os n a (s+1)
         ]
   in tx (op i) Link $
   claim (ip i) $
   c' [] n a 0
```

Figure 6.10: The Definition of `countserv`.

inserted into the sequence of continuations. Otherwise, they are skipped.

A `link` is a Gadget that provides one view of the count in a counter server. It is the client. Its definition is given in Fig. 6.11. The `Name`s are used to calculate a new colour for each set of links. We assume that `CountRq` is of class `Text`.

A `counters` Gadget sets up the first counter server and link, placing the link window on a `wall`[4]:

```
counters =
  wire $ \w ->
```

[4] A `wall` is a Gadget to place `windows` on

```
link :: Name -> Int -> In Int -> Out CountRq -> Gadget
link id n i o =
    wire $ \w ->
    spawn (mapC const i (op w)) $
    let d = display' (editorInput (ip w)) n
        b m = button' (picture (text fn co (show m))) o m
            where fn = "*lucida*"
                  co = col "black"
        k = nameToColour id
    in wrap' (border 20.picture (colourbox k))
               above (l++(map b [Auto,Manual,Copy,Link]))
```

Figure 6.11: The Definition of `link`.

```
spawn (countserv (op w) "" 0 False) $
wall' (width 800.height 600.picture (colourbox (col "light grey"))) (ip w)
```

Evaluation

The use of a client-server structure enables Components to share common data. It is possible for the server to restrict the operations performed on the shared data. It is necessary for each client Component to explicitly receive new updates of the state, though this could be hidden using abstraction.

The counters do not indicate visually that they are in *auto* mode, except by the fact that their integers are increasing by themselves. The *Auto* button should stay lit whilst the counter is in auto mode, returning to normal when *Manual* is pressed.

If the *Copy* or *Link* buttons are pressed whilst a counter server is in auto mode, the action is not completed until the server returns to manual mode (when *Manual* is pressed on one of the links). This is because the `Matic` messages that the counter server sends itself *take over* the machine and don't leave any time for the creation of a new link or copy. This problem could be solved if each message sent could be given a priority. A `Matic` message would be given a lower priority than `Copy` and `Link` messages.

Alternatively, access to some sort of timer to restrict the rate of flow of `Matic` messages would solve the problem. By sending the timer a message it could be *programmed* to send a `Matic` message to the counter server say once a second, then instructed to stop when a `Manual` message arrives. This solution is better because it gives precise control over the frequency of update. However, we then move into

the realm of real-time constraints: problems could arise if the frequency proved to be too high for the program to respond to a message before the next timer message was sent.

6.6 Grid Explode

Motivation

Concurrent processes are not only useful for building GUIs. In this example we put concurrency to good use in mechanising the rules of a board game.

Problem

The game of Grid Explode is played on a rectangular grid of cells. Each player has a supply of stones of a unique colour. Players take turns placing a stone into any cell that does not already contain another player's stones. The *neighbours* of a cell are the cells above, below, to the left and to the right of the cell. The *capacity* of a cell is the number of neighbours it has. If the total number of stones in a cell remains below the capacity after a player places a stone into it, then the turn is over and it is the next player's turn. When a cell reaches its capacity, it *explodes* invading each of its neighbours with one stone. A stone invading a cell turns any occupants to its colour, and may cause further explosions. The turn is over when all cells are below their capacity. A player wins if his or her move results in perpetual explosions or the explosions cease leaving every cell containing at least one stone of their colour.

Solution

The Gadget version assumes two players only. The picture on the right-hand side of Fig. 6.6 shows an example screen image. Each cell is a Gadget, with a duplex connection for each of its four sides. This is the type of a *Grid Gadget* with four duplex wires, one for each side:

```
InOut m m -> InOut m m -> InOut m m -> InOut m m -> Gadget
```

A higher-order function, `grid`, wires grid Gadgets to their neighbours and places them in a grid on the screen:

```
grid :: [InOut m m] -> [InOut m m] -> [InOut m m] -> [InOut m m] ->
     [[InOut m m -> InOut m m -> InOut m m -> InOut m m -> Gadget]]
     -> Gadget
```

`grid` takes a list of duplex connections for each side of the grid and a list of rows of grid Gadgets. In this application, we do not use the edge connections in the grid, so give a list of `(nci,nco)` values for each edge, where `nci` and `nco` are special values of type `In a` and `Out a`, respectively, indicating that a pin is a *not-connected input* or *output*. In the main `explode` function, we set the capacity of cells and wire them to a `referee` Component, then pass all the cells to `grid`. The cells are wired as shown on the left-hand side of Fig. 6.12.

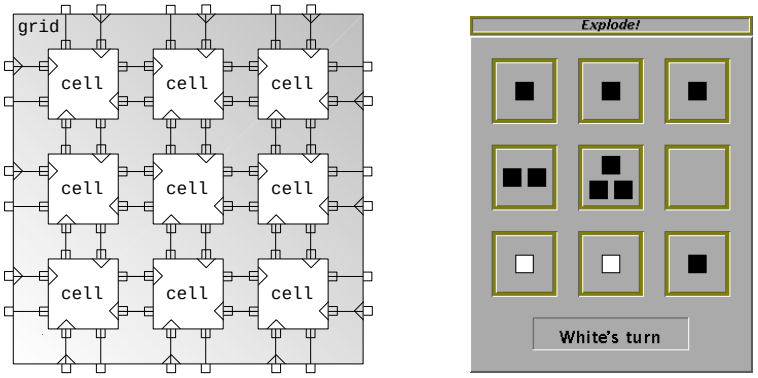


Figure 6.12: The Programmer and Player Views of the Grid.

To achieve a similar wiring in the Fudgets system would require a significant amount of multiplexing. Our initial design of the Explode game did not foresee the need for a referee Component (see below), so additional links were introduced at a later stage. A new duplex pin was added to the cell and wired to the referee by a partial application of each cell function, before passing the resulting grid Gadgets to `grid`. To modify the wiring in this way with Fudgets would require the Fudget equivalent of `grid` to be completely re-written.

Stones are transmitted in messages to adjacent cells when an explosion occurs. Each cell knows when it has been clicked, how many stones it contains (if any) and what colour they are, but knows nothing about the other cells in the grid. There are three problems to solve in this distributed implementation: (1) How does a cell determine what a click represents (ie. which player)? (2) How is termination detected? (3) How is the next player prevented from moving until the explosions from the previous move have completed?

To solve these problems each cell has a further duplex connection to a *referee* that negotiates the players turns and decides when a player has won. Cells send the referee messages of type `Notify (Maybe Player)`:

```
data Notify m = ClickOver m
              | Bonk m
              | Boom Int m
```

A `ClickOver` message indicates the colour of any stones already within the cell; a "`Boom n c`" message indicates an explosion, where `n` is the number of neighbouring cells exploded into and `c` is the colour of any stones previously in the cell; a cell that receives a stone, but does *not* explode, sends a "`Bonk c`" message, where `c` is the colour of any stones previously in the cell. The `cell` (Fig. 6.13) remembers its contents with a state value of type `Maybe (Player,Int)` where `Maybe a = Yes a | None` indicating that either the contents is empty with "`None`", or that there are `n` stones belonging to player `p` with "`Yes (p,n)`".

The referee (Fig. 6.14) sends cells `Rulings`:

```
data Ruling = Invasion Player | ClearSmoke
```

The referee remembers whose turn it is and when a cell sends a message saying it has been clicked and currently contains colour `c`, the referee checks that the move is valid, and if so returns an `Invasion` ruling, but otherwise makes no response. The referee is able to tell that explosions have ceased by keeping a tally. On receipt of a "`Boom n c`", it increases the tally by `n`-1; On receipt of a "`Bonk c`", it decreases the tally by one. When the tally reaches zero, the explosions have stopped, exploded cells are sent a `ClearSmoke` message and the next player may take a turn. To detect the end of the game the referee keeps a count of how many cells contain stones of each colour and which cells have exploded this move.

Evaluation

Concurrency was useful in this particular application, because cells react at times other than in response to user input (ie. when stones arrive from other cells). However, in a game like *noughts and crosses* where each cell on the board changes only in response to user input, concurrency would not be needed for the game algorithm.

It was simple to make a new combinator to connect together the cell Gadgets of the board. When we found that each cell required an extra connection to a referee Component, we could go back and add one easily.

```
cell :: Int -> InOut Ruling (Notify (Maybe Player)) ->
         Invade -> Invade -> Invade -> Invade -> Gadget
cell capacity (fromReferee,toReferee) l t r b =
   wire $ \d ->
   wire $ \c ->
   giveImage (button (pictureIn (ip d)) (op c) ()) $
   let draw = op d
       click = ip c
       cell' o =
          let neighbours = [l,t,r,b]
              invasion p o =
                 let (op,n') = case o of
                       None -> (None,1)
                       Yes (p,n) -> (Yes p,n+1)
                 in tx draw (stones p n') $
                 if n' < capacity then
                    tx toReferee (Bonk op) $
                    cell' (Yes (p,n'))
                 else
                    tx draw boom $
                    tx toReferee (Boom capacity op) $
                    sequence [tx n p | (_,n) <- neighbours] $
                    cell' None
          in rx [
             from click $ \_ ->
                let o' = case o of
                       None -> None
                       Yes (p,_) -> Yes p
                in tx toReferee (ClickOver o') $
                cell' o,
             from fromReferee $ \m -> case m of
                Invasion p -> invasion p o
                ClearSmoke -> tx draw blank $ cell' o,
             froms neighbours $ \p _ ->
                invasion p o
          ]
   in claim fromReferee $ claim click $
   sequence (map (claim.fst) [l,t,r,b]) $
   cell' None
```

Figure 6.13: The Definition of a `cell` in Grid Explode.

6.7 The Escher Tile Program

Motivation

The Escher Tile program [Fou95] was used to test ease of programming in the Fudgets and Concurrent Clean systems (see Chapter 2, *"Review"*). The main prob-

```
referee :: [InOut (Notify (Maybe Player)) Ruling]->Out Turn->Component
referee cs turn =
  sequence (map (claim.fst) cs) $
  tx turn (Turn Black) $ p Black 0 everyone [] ih
  where
  p::Player->Int->[CellID]->[CellID]->[(Player,Int)]->Component
  p c t b s h =
    rx [
      froms cs $ \r toCell ->
        let newcount h o n = map (adj (+1) n.adj (\x->x-1) o) h
            adj _ None h = h
            adj f (Yes np) (p,n) = (p,if p==np then f n else n)
            t' = t - 1
        in case r of
             ClickOver v ->
               if (v == None || v == Yes c) && t == 0 then
                 tx turn NoTurn $
                 tx toCell (Invasion c) $
                 p c 1 b s h
               else p c t b s h
             Bonk pc -> let h' = newcount h pc (Yes c)
                            s' = s \\ [toCell]
               in if t' == 0 then
                 if (any (==nc) (map snd h')) then
                   tx turn (Win c) $
                   end
                 else
                   let c' = opponent c
                   in tx turn (Turn c') $
                   sequence [tx e ClearSmoke | e <- s'] $
                   p c' 0 everyone [] h'
               else p c t' b s' h'
             Boom n pc -> let b' = b \\ [toCell]
               in if b' == [] then tx turn (Win c) $ end
               else p c (t'+n) b' (toCell:s) (newcount h pc None)
    ]
  everyone = map snd cs
  ih = zip players (repeat 0)
  nc = length cs
```

Figure 6.14: The Definition of `referee` in Grid Explode.

lems found were related to the inability to break a program down into modular parts that were independent in definition. We now present a Gadget Gofer version for comparison. It illustrates a number of features often found in GUI-based

programs.

Problem

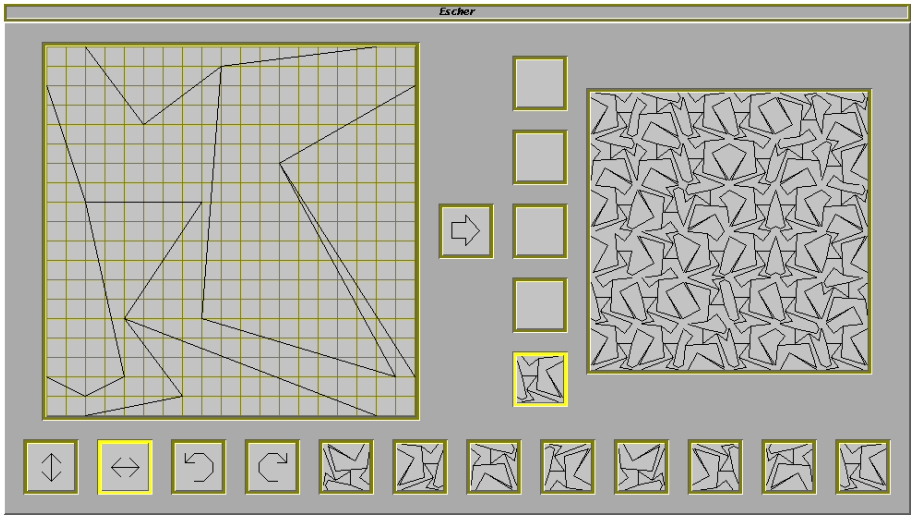


Figure 6.15: The Escher Tile Program.

A picture of an Escher program display is in Fig. 6.15. The program allows tiles to be designed that may then be placed onto a grid in different rotations and reflections to form a pattern. There are four main parts to the program, each with an area of display on the screen. The *design* area allows tiles to be designed. Clicking the mouse on two points on the design adds a line, except where a line already exists, in which case the line is removed. The *choice* area displays five different tile designs. One of the designs is *selected*, indicated by a highlighted border around the tile. The selected tile is also found in the design area and the tools area. The *tools* area displays a selection of tools that may be used to alter the contents of the board. One of the tools is *selected*, and it is this one that will be applied when a cell of the board is clicked. The tools consist of four rotation/reflection tools, that when selected alter a tile already placed on the board, and eight relections/rotations of the currently tile, that when selected replace tiles on the board. The *board* contains a grid of cells that may be empty or contain a tile. When a cell on the board is clicked, the currently selected tool is used to alter the contents of the cell. For example, if the *mirror vertically* tool is selected (the left-most tool) and a cell on the board clicked, the tile in the cell will turn upside down.

Solution

The structure of the program is similar to the Fudgets version described in Chapter 2, *"Review"*, but without the explicit multiplexing and routing of message streams that was necessary with Fudgets.

Tiles are represented by drawing functions that produce a rectangular picture containing lines only. We have some functions that rotate and mirror drawings:

```
rotateCW, rotateACW, mirrorH, mirrorV :: DrawFun -> DrawFun
```

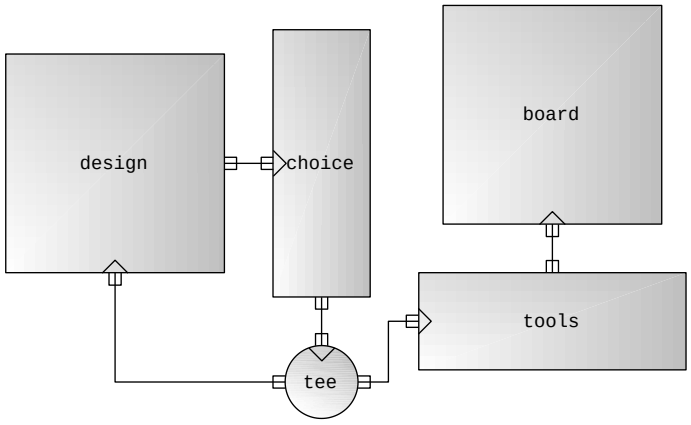


Figure 6.16: Major Component Connections in the Escher Tile Program.

The main Components of the Escher program are shown in Fig. 6.16. At the top of the Component hierarchy in the Gadget Escher program we see the main sections of the program connected exactly as shown in the process diagram.

```
escher_prog :: Gadget
escher_prog =
    wire $ \dc -> wire $ \ct -> wire $ \td -> wire $ \tt -> wire $ \tb ->
    spawn (tee (ip ct) (op td) (op tt)) $
    let d = design (ip td) (op dc)
        c = choice 5 (ip dc) (op ct)
        t = tools (ip tt) (op tb)
        b = board (6,6) (ip tb)
    in setGapSize 20 $ d <-> c <-> b <|> t
```

Design Gadget The design Gadget has one input and one output, both for messages containing tile designs. It consists of an area where lines can be added and removed to a tile design and a button to emit the current design to an outside Gadget. New tiles arriving on the input replace the current design. The `paper`

Gadget from the library already provides most of the functionality we require, but it outputs a new design every time a line is added. We want to control the output of new designs with a button, because they will be sent to the *tools* area and we do not want the tool area to be updated every time a line is added as this takes a few moments. We alter the `paper` Gadget by connecting its output to a memory Component that stores the latest tile design, and connect a button to trigger the memory to send out the design when the user has finished designing. The definition is a simple composition of `paper`, `memory` and `button`:

```
design :: In DrawFun -> Out DrawFun -> Gadget
design i o =
    wire $ \d -> wire $ \b ->
    spawn (memory (ip b) (ip d) o) $
    let p = paper' (gridlock (20,20).width 400.height 400.
                    pictureIn i.pictureOut (op d))
        s = button' (picture rightarrow) (op b) ()
    in p <-> s
```

Choice Gadget The `choice` Gadget consists of a set of buttons connected in a *radio group*, ie. exactly one is *on* at any one time. In addition we require that a tile sent to the choice group is directed to the currently selected button. We use a *radio gate* for this purpose. Radio groups and gates are provided in the library but since they were developed first for the Escher program, we describe them here before we look at their use in the `choice` Gadget.

Radio Buttons The choice and tool buttons are each examples of *radio-buttons*. At all times one of the buttons in each group is *on* whilst the rest are *off*. Clicking another button in the group deselects the previously selected button, and selects the clicked one. It is not possible to deselect a button by clicking it again, one button must always be selected.

Groups of radio buttons are created using ordinary buttons, but using a couple of extra pins on each button that are not normally used, called *set* and *notify*. The *set* pin is a boolean input to the button that enables the state of the button (*on* or *off*) to be set (`True` = *on*). The *notify* pin is a boolean output from the button that indicates a change of state. Whenever the button changes state by being clicked, the new state is emitted from the *notify* pin as a boolean value. Connections to the *set* and *notify* pins are made as attributes of the button. The default value of these attributes is that the pins are not connected. When used in a radio group, the

attribute `buttonPon` is also used to make a button that can be **P**ressed (clicked) **on**, but cannot be pressed off. The *set* and *notify* pins are connected to a radio-group control Component, as shown in Fig. 6.17.

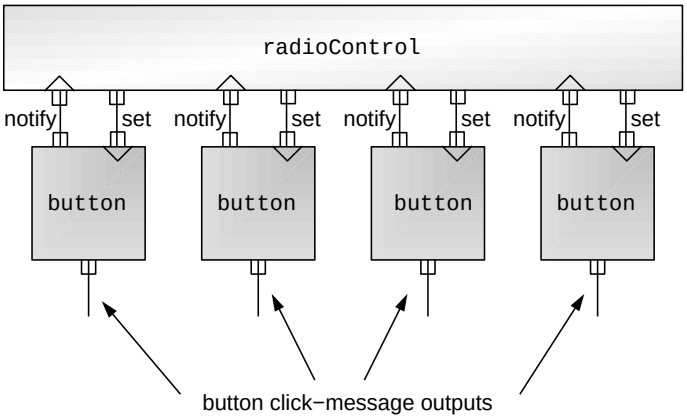


Figure 6.17: Buttons Wired to a Radio-Group Controlling Component.

The radio-group controller begins by sending a *set* message to the first of the buttons (so that the groups starts with one button selected). A *notify* message from any other button triggers the radio-group controller to send a `False` *set* message to deselect the previously selected button.

In the choice radio group we require slightly more functionality. A new tile design sent to choice should replace the currently selected tile. To do this we use another controller Component, called a `radioGate`. This is attached to the radio buttons as shown in Fig. 6.18.

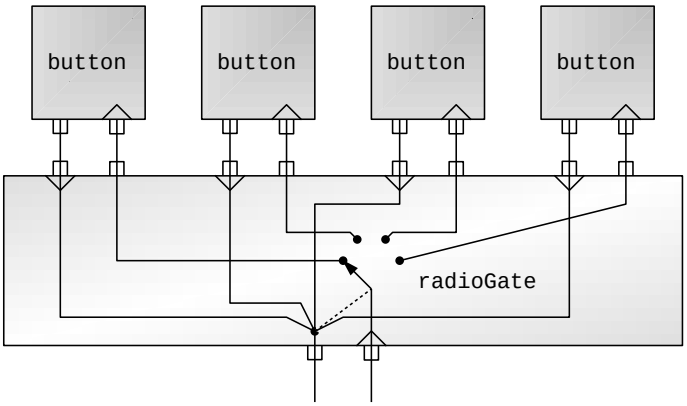


Figure 6.18: Radio Buttons Wired to a Radio Gate Controlling Component.

A message input to the gate controller is sent to one of the buttons, depending

on the position of the gate switch. Messages output from the buttons are merged into a single output from the gate controller, *and set the gate switch to the button that sent the message*. So, a message sent to the gate controller is passed to the last button that sent a message out. In the Escher program, a tile sent to the choice Gadget is directed by the radio gate controller to the last button that was selected.

The radio group and gate controllers can be used with other Components apart from buttons. Any set of Components that have *set* and *notify* pins can be used as a radio group. Any set of Components that each have an input and an output (all inputs must be of the same type and all outputs must be of the same type for the whole set) can be connected to a radio gate.

Using the Radio Group and Gate Controllers Functions are provided for connecting Components to a radio group or gate controller. The function `radioGroup` is applied to a list of Components. The Components in the group may have other pins, but as we do not know their type we leave this unspecified in the type: `b`. Each Component is partially applied to a duplex radio controller connection, and the resulting Components (of type `b`) passed on to the continuation in a list.

```
radioGroup :: ComponentClass a =>
    [InOut Bool Bool -> b] -> ([b] -> Process a) -> Process a
```

We might have insisted that the components connected to a radio group have the type:

```
InOut Bool Bool -> Process s
```

and then the `radioGroup` function could create each of the Components directly instead of passing it on to the continuation. However, this would make it impossible to connect a group of Components to both radio group *and* radio gate controllers at the same time.

The mechanism for connecting to a radio gate is much the same as for a group:

```
radioGate :: ComponentClass d =>
    In a -> Out b -> [InOut a b -> c] -> ([c] -> Process d) -> Process d
```

The gate has two extra pins, one that will forward messages *to* the currently selected Component, and one that forwards messages *from* the currently selected Component.

Returning to the Choice Gadget Having defined the radio group and gates, we now return to the definition of the `choice` Gadget. It consists of a radio group of `pictureStoreButtons`. These are buttons that display a picture. When they are clicked, they emit their picture as a message. They can also be sent a new picture, changing their display. A new tile sent to the `choice` Gadget is forwarded to the currently selected button via a radio gate. Whenever the user clicks one of the `choice` buttons, the tile emitted is forwarded via the radio gate and emitted from `choice`:

```
choice :: Int -> In DrawFun -> Out DrawFun -> Gadget
choice n i o =
    radioGroup (copy n pictureStoreButton) $ \rb ->
    radioGate i nco rb $ \cb ->
    above cb
```

The `pictureStoreButton` is a composition of a normal `button`, a `tee` and a `memory` (Fig. 6.19). Drawings sent to it are duplicated by the `tee`, passed to the `button` (to replace the picture displayed) and to the `memory` to be stored. An empty tile drawing (`transparent`) is placed into wire `a` as a way of setting the initial contents of the `memory`. When the `button` is clicked, its click-message triggers the `memory` Component to emit the last stored drawing, and this is emitted from the `pictureStoreButton`. The *set* and *notify* pins of the `button` are connected to pins of the `pictureStoreButton` so that it can form part of a radio group:

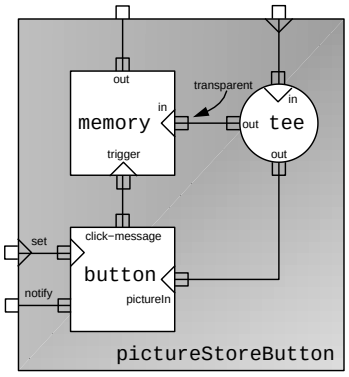


Figure 6.19: `pictureStoreButton`

```
pictureStoreButton :: InOut Bool Bool -> InOut DrawFun DrawFun -> Gadget
pictureStoreButton (se,no) (pi,po) =
  wire $ \a -> wire $ \b -> wire $ \c ->
  spawn (tee pi (op a) (op b)) $
  tx (op a) transparent $
  spawn (memory (ip c) (ip a) po) $
  button' (buttonPon.pictureIn (ip b).
           buttonSet se.buttonNotify no) (op c) ()
```

Tools Gadget The tools Gadget (Fig. 6.20) consists of a set of twelve buttons: two rotation buttons, two reflection buttons and eight tile buttons, each containing a different rotation/reflection of the current tile. The tools Gadget has an input for a new tile drawing and an output for a tile-changing function. A new tile received becomes the current tile. The pictures on the tile buttons change to display the new tile in the rotation/reflection associated with each one. Clicking any of the tool buttons causes a tile-changing function to be emitted from the tools Gadget. If a rotation or reflection tool is clicked, the function emitted takes a tile drawing and returns the tile rotated or reflected accordingly. If a tile tool is clicked, the function emitted is `const t`, where `t` is a drawing of the tile in the reflection/rotation associated with the tool. In the Escher program the tools Gadget receives a new tile from the choice Gadget and sends a tile-changing function to the board Gadget.

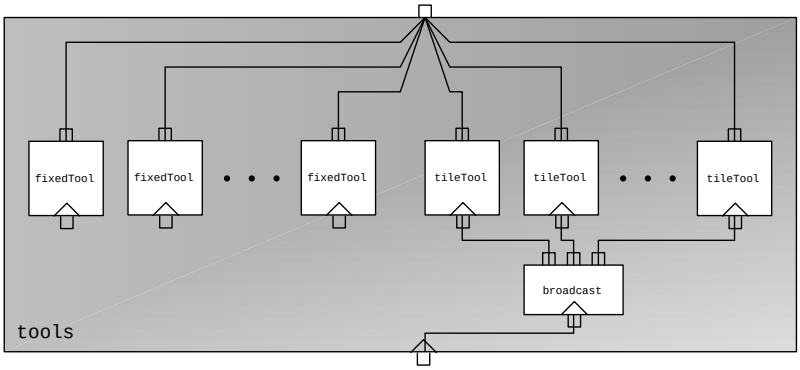


Figure 6.20: The Components of the `tools` Gadget.

A tile drawing received by `tools` is `broadcast` to each `tileTool`. Only one Component can receive messages from a wire, hence the need for the `broadcast` Component. Tile-changing functions from any tool are merged to the output pin of `tools`. We do not need an explicit `merger` Component because any number of Components can send messages on a single wire. The tool Gadgets are connected

into a radio group to ensure that only one is active at any one time:

```
tools :: In DrawFun -> Out (DrawFun -> DrawFun) -> Gadget
tools i o =
    wiresIn tileTools $ \tileTools' bi ->
    spawn (broadcast i bi) $
    radioGroup (tileTools'++fixedTools) $ \rt ->
    let gs = map (\g->g o) rt
    in beside gs
```

The `wiresIn` function attaches wires to input pins on a list of Components and returns the list of connected Components and a list of the other ends of the wires.

The tile rotation/reflection tools do not change their function when a new tile is received, so we call them `fixedTools`. Each fixed tool is parameterised on the picture it displays (eg. a clockwise-rotating arrow for the clockwise-rotation tool), and the tile-changing function it emits (eg. `rotateCW`). The fixed tools are implemented as `buttons`:

```
fixedTools = map fixedTool [(rotCWIcon,rotateCW), (rotACWIcon,rotateACW),
                            (mirHIcon,mirrorH), (mirVIcon,mirrorV)]

fixedTool :: (DrawFun, (DrawFun->DrawFun)) -> InOut Bool Bool ->
                                    Out (DrawFun->DrawFun) -> Gadget
fixedTool p f (se,no) o = button' (buttonSet se.buttonNotify no.
                                   picture p.buttonPon) o f
```

The tile tools are parameterised on the function that when applied to a tile, produces the reflection/rotation that they represent.

```
tileTools = map tileTool [id, rotateCW, (rotateCW.rotateCW),
                          rotateACW, mirrorV, (rotateCW.mirrorV),
                          (rotateCW.rotateCW.mirrorV),
                          (rotateACW.mirrorV)]

tileTool :: (DrawFun->DrawFun) -> In DrawFun -> InOut Bool Bool ->
                                        Out (DrawFun->DrawFun) -> Gadget
tileTool f i (se,cl) o =
    wire $ \a -> wire $ \b -> wire $ \c -> wire $ \d -> wire $ \e ->
    spawn (mapC f i (op a)) $
    spawn (tee (ip a) (op b) (op e)) $
    spawn (mapC const (ip b) (op c)) $
    spawn (memory (ip d) (ip c) o) $
    button' (buttonSet se.buttonNotify cl.
             pictureIn (ip e).buttonPon) (op d) ()
```

The `tileTool` (Fig. 6.21) is similar to the `pictureStoreButton` of the `choice` Gadget, but not similar enough to re-use it. The difference is that a `mapC f` Component is used to convert a tile received into the correct rotation/reflection, and a `mapC const` converts the resulting tile into a tile-changing function.

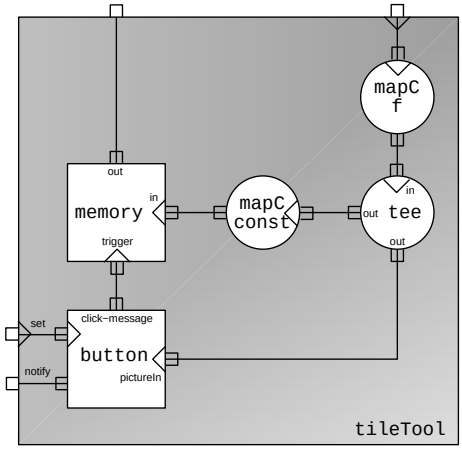


Figure 6.21: The `tileTool` Gadget.

Board Gadget The `board` Gadget consists of a grid of cells that hold tile drawings. A tile-changing function, sent to the `board` in a message, is applied to the tile in any cell that is clicked. The `board`'s only pin is an input for tile-changing functions, connected to the tools Gadget. The `board` is set up as follows:

```
board :: (Int,Int) -> In (DrawFun->DrawFun) -> Gadget
board (sx,sy) i =
    let bcs = copy (sx*sy) boardCell in
    wiresInOut bcs $ \bs ws ->
    spawn (boardControl i ws) $
    in setGapSize 0 $ wrap' (border 5) (squareof sx bs)
```

For a `board` of size `sx` by `sy`, `sx*sy` copies of a `boardCell` Gadget are attached to wires. The ends of these wires are passed to a `boardControl` Component. The cells are positioned in a square of width `sx`.

The `boardControl` Component (Fig. 6.22) holds the current tile-changing function. We choose to keep the function here rather than broadcast it to every cell as broadcasting many messages makes the program slow. Initially the identity function `id` is held.

```
boardControl :: In (DrawFun->DrawFun) ->
                 [InOut () (DrawFun->DrawFun)] -> Component
boardControl f bcs = claim f $ sequence (map (claim.fst) bcs) $ bc id
    where
    bc c = rx [ from f $ \c' -> bc c',
                froms bcs $ \_ r -> tx r c $ bc c ]
```

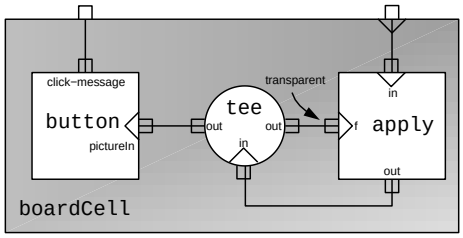


Figure 6.22: The `boardCell` Gadget.

```
boardCell :: InOut (DrawFun->DrawFun) () -> Gadget
boardCell (i,o) =
    wire $ \a -> wire $ \b -> wire $ \c ->
    spawn (apply i (ip a) (op b)) $
    spawn (tee (ip b) (op a) (op c)) $
    tx (op a) transparent $
    button' (pictureIn (ip c).border 0) o ()
```

When a `boardCell` is clicked it sends a unit `()` message to the `boardControl`, causing it to respond with the current tile-changing function. When this function is received it enters an `apply` Component. The `apply` Component is like a `memory`, but instead of sending any message to trigger it, a function is sent that is applied to the contents of the memory and the result output. In the `boardCell` an `apply` Component holds the tile drawing displayed. When the tile-changing function arrives at the `apply` it is applied to the current tile. The resulting tile is passed to the `button` to display and also back to the `apply` to replace the stored tile.

Evaluation

Functions as Messages Selecting a tool sends a function to the board. When a cell in the board is clicked, the function is applied to the current contents of the cell resulting in a new tile. Rather than broadcast a function sent from the tools to all cells in the board, a `boardControl` Component holds the lastest function received, and cells request the function when they are clicked.

As with the counter example, passing functions to modify the state held in another Component is a more flexible method than programming the Component to change its state in a number of set ways. For example, a new tool could easily be added to remove a tile from a cell on the board.

Comparison Between Fudget, Concurrent Clean and Gadget Versions The Gadget version is most similar to the Fudget version in program structure (Fig. 6.23). Whilst the main components of the program (board, tools, etc.) are similar, unsurprisingly it is the *plumbing* of these components that is most different. For example, consider the top level of program definition where the design, choice, board and tools components are connected. In the Fudget version the structure of the program is obscured by combinators such as `loopLeftF` and `throRight`. In the Gadget version the structure is clearer; we can see directly what the wires connect. In the top level of the Concurrent Clean version we cannot see which components communicate at all! All we see are the names of functions that describe each component of the program. Communication between them is defined *inside* each component, through the common use of certain parts of the global state.

The top levels of each program are a fair representation of the rest of the program. Levels further down are programmed in a similar style to the top level in each system.

6.8 Gadget Viewport

Motivation

This example demonstrates a useful property of the relationship between parent and child images in the Gadget image hierarchy.

Problem

Gadgets take as much room on the screen as they require, regardless of how much room is available. A Viewport is a wrapper for a Gadget that gives a display through which the Gadget is viewed. Horizontal and vertical scroll bars enable the piece of Gadget being viewed to be panned across the viewport. Sliders can be moved by dragging the *thumb* or by clicking buttons on either end of the slider. The size of the *thumb* in each slider indicates what proportion of the child is visible. A corner

```
escher :: Gadget
escher =
    wire $ \dc -> wire $ \ct -> wire $ \td ->
    wire $ \tt -> wire $ \tb ->
    spawn (tee (ip ct) (op td) (op tt)) $
    let d = design (ip td) (op dc)
        c = choice 5 (ip dc) (op ct)
        t = tools (ip tt) (op tb)
        b = board (6,6) (ip tb) in
    d <-> c <-> b <|> t
```

```
escherF = placerF (revP horizontalP) (playF >==< workF >==< quitButtonF)

playF :: F (Either [Tile] [Tile]) c
playF = (boardF,Above) >#==< throRight toolsF

workF :: F b (Either [Tile] [Tile])
workF = loopLeftF ((worka,LeftOf) >#==< workb)

workb :: F (Either Tile c) (Either Tile (Either [Tile] [Tile]))
workb = choiceF >=^< fromLeft

fromLeft :: Either a b -> a
fromLeft (Left x) = x

worka :: F (Either Tile a) (Either Tile a)
worka = throRight (designF)
```

```
:: Start UNQ WORLD -> UNQ WORLD;
   Start world -> CloseEvents events' world',
      (ps', events'): StartIO [menus,timer] StartState [] events,
      (events, world'): OpenEvents world,
      menus: MenuSystem [file, tools],
      file: PullDownMenu FileId "File" Able
         [MenuItem QuitId "Quit" (Key 'Q') Able Quit],
      tools: PullDownMenu ToolsId "Tools" Able
         [MenuItem ShowToolsId "Show tools" (Key 'T') Able ShowTools,
          MenuItem ShowChoiceId "Show choice" (Key 'C') Able ShowChoice,
          MenuItem ShowDesignId "Show design" (Key 'D') Able ShowDesign,
          MenuItem ShowBoardId "Show board" (Key 'B') Able ShowBoard,
          MenuItem ShowDebugId "Show debug" NoKey Able ShowDebug],
      timer: TimerSystem [Timer 1 Able TicksPerSecond ReDrawDebug];
```

Figure 6.23: The Top Level of the Gadget, Fudget and Clean Escher Programs.

icon allows the viewport to be resized. For example, the Escher Tile program in a viewport is show in Fig. 6.24.

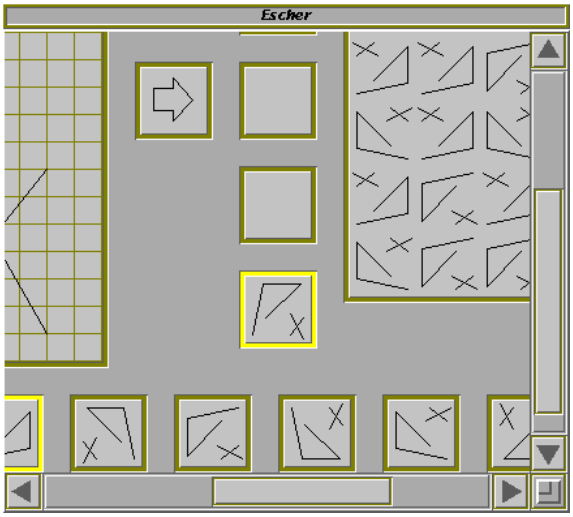


Figure 6.24: The Escher Tile Program in a Viewport Gadget.

Solution

Because the image of a Gadget is clipped to the borders of its parent, we can achieve the required effect by making a parent Gadget of the size required of the viewport and placing the larger child Gadget that is to be viewed inside it. By moving the origin of the child relative to its parent we can view different parts of it (see Fig. 6.25).

The Components of the `viewport` are shown in Fig. 6.26. The heart of the `viewport` is a `viewer` Gadget. This is the parent of the child to be viewed. It has two inputs, one for changing the size of the view, and one for changing the position of the child. The size of the view is changed if the `viewport` is resized. The position of the child is changed if one of the `sliders` is moved. The `viewer` has one output that indicates a new size of the child if the child Gadget grows. This indicates that the *thumb* part of each `slider` should be resized so that it continues to give an indication of the proportion of child visible.

The horizontal and vertical sliders consist of a `slider` Gadget together with a couple of arrow buttons. The `slider` contains a *thumb* that can be moved horizontally or vertically within the boundaries of the slider. For the horizontal slider, the thumb fills the slider vertically, so can only be moved horizontally.

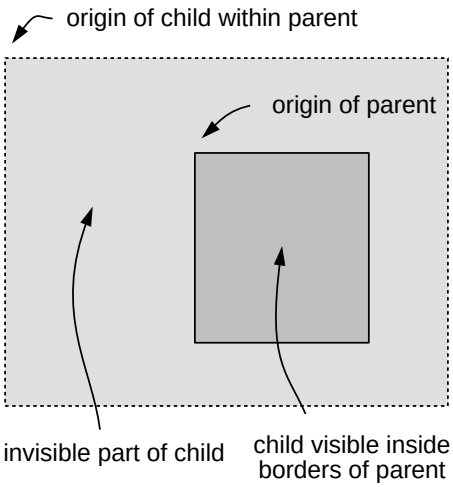


Figure 6.25: Larger Child Within Small Parent.

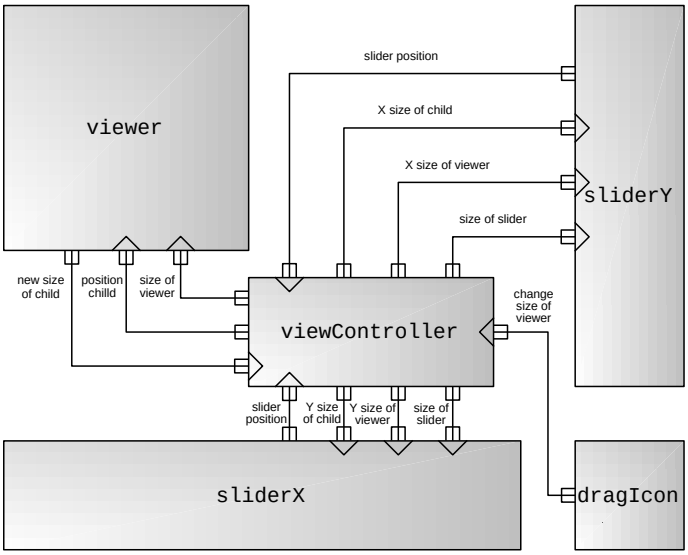


Figure 6.26: The Components of a Viewport

A `dragIcon` is attached to the corner of the `viewport`. Dragging this Gadget resizes the `viewport`.

The `viewer`, `sliders` and `dragIcon` are connected to a `viewController` that governs the exchange of information between them. For instance, when the vertical slider is moved down, the `viewController` sends a message to the `viewer` to change the origin of the Gadget.

Evaluation

The fact that child Gadgets are not visible outside the borders of their parent has made the viewport very easy to implement. Instead of using this feature of the Gadget image hierarchy, we could intercept the drawing function sent from the viewed Gadget to SM, and *clip* it to the viewing area.

Our `slider` Gadgets are composed in much the same way as the scrollbar used by Duke and Harrison to describe *Interactors* [DH93]. Interactors are a formal model of the elements of a GUI. Communication between Interactors is in the form of common *events*, which can be used to model message-passing on channels.

Other systems have interface Components that size themselves both according to the space they require *and* what space is available. This is one possible improvement of the Gadget Gofer system.

The buttons on either end of the slider move the slider a set amount. In some applications it is desirable to be able to set this amount. For example, if the viewport were being used to display some text, the vertical slider buttons might shift the text a line's height up or down.

6.9 A Fudget Simulator

Motivation

The Fudgets system was one of the main motivating factors of this work. We wanted a system that was like Fudgets but without the restriction on number of inputs and outputs and the requirement for communication only between adjacent components[5]. That is, a *generalisation* of Fudgets. To show that Gadgets do generalise Fudgets, we present here a Fudget simulator written within the Gadgets system. Fudget programs can be loaded into the Gadgets system and evaluated just as if they had been compiled with the Fudget libraries. Thomas Hallgren and Magnus Carlsson (of Fudget fame) were the first to try this idea. Our own simple formulation is in essence the same solution as theirs.

[5] the latest version of Fudgets decouples layout from communication

Problem

A Fudget is like a Gadget with one input and one output. The big question is how to give access to the screen. In the Fudgets system it is through the low level messages in and out of a Fudget, that are routed through the hierarchy of Fudgets and result in a dialogue between Fudgets and The X Windows System. We could simulate X events and program respond to X commands, so that all Fudgets from the most primitive upwards can be simulated. This would require a lot of work, however, as many features of The X Windows System are used. As Gadgets do not have access to these features, they would have to be emulated. Alternatively we could implement the most primitive Fudgets (buttons, integer displayers, etc.) at Gadgets and give them a Fudget wrapper, then use actual Fudget definitions at all levels above that. We choose the latter solution, to avoid the copious amounts of work needed to emulate The X Windows System.

Solution

A simulated Fudget is a Component with one input and one output pin:

```
type F a b = In a -> Out b -> Component
```

We define the basic combinators `>==<`, `>+<` and `>*<` for our simulated Fudgets in Fig. 6.27. In these definitions we ignore the possibilty of using *named layout* or *placer Fudgets* to specify layout in a program. These combinators use *default layout* only. (See §2.3.1.) In Fig. 6.28 we define some combinators and functions concerned with abstract Fudgets and stream processors. We wrap some Gadgets to give them Fudget types in Fig. 6.29. For simplicity we do not implement all the features of the corresponding Fudgets in the real system. For example, the real Fudget `buttonF` is sized to fit the text printed on it.

The core simulator defined here supports enough combinators and Fudgets to run the Fudget counter of §2.3.1. The Fudget program is *exactly* as given on page Page 27. The screen display is not exactly the same as the Fudget version because we have not simulated every aspect of the look and behaviour of the `buttonF` and `intDispF` Fudgets, but the functionality of the simulated version is the same. The display could be made exactly the same with a more complicated wrapper around the Gadgets that simulate `buttonF` and `intDispF`.

```
type F a b = In a -> Out b -> Gadget

data Either a b = Left a | Right b

(>==<) :: F a b -> F c a -> F c b
(>==<) a b i o =
    wire $ \w ->
    (a (ip w) o) <-> (b i (op w))

(>+<) :: F a b -> F c d -> F (Either a c) (Either b d)
(>+<) a b i o =
    wire $ \il -> wire $ \ir ->
    wire $ \ol -> wire $ \or ->
    let split_untag :: In (Either a b) -> Out a -> Out b -> Component
        split_untag i l r =
            claim i $
            su where
            su = rx [ from i $ \tm -> case tm of
                Left m -> tx l m $ su
                Right m -> tx r m $ su ]
        tag_merge :: In a -> In b -> Out (Either a b) -> Component
        tag_merge l r o =
            claim l $ claim r $
            tm where
            tm = rx [ from l $ \m -> tx o (Left m) $ tm,
                      from r $ \m -> tx o (Right m) $ tm ] in
    spawn (split_untag i (op il) (op ir)) $
    spawn (tag_merge (ip ol) (ip or) o) $
    (a (ip il) (op ol)) <|> (b (ip ir) (op or))

(>*<) :: F a b -> F a b -> F a b
(>*<) a b i o =
    wire $ \ia ->
    wire $ \ib ->
    spawn (tee i (op ia) (op ib)) $
    (a (ip ia) o) <|> (b (ip ib) o)
```

Figure 6.27: A Fudget Simulator in Gadgets.

Evaluation

We have shown that Fudgets and Fudget combinators can be simulated in the Gadgets system. The combinators we have chosen to simulate are not a sufficient *generating set* that any of the Fudget combinators can be built on top of them. To achieve this we must simulate combinators for *stream processors*, the base on which

```
data SP a b = NullSP | GetSP (a -> SP a b) | PutSP b (SP a b)

absF :: SP a b -> F a b
absF sp i o =
    initGadget (0,0) blank $
    claim i $
    absf sp where
    absf NullSP = $ end
    absf (PutSP m c) = tx o m $ absf c
    absf (GetSP c) = rx [ from i $ \m -> absf (c m) ]

(>=^^<) :: F a b -> SP c a -> F c b
(>=^^<) f sp = f >==< absF sp

(>^=<) ::  (a -> b) -> F c a -> F c b
(>^=<) fun fud = absF (mapSP fun) >==< fud

mapSP :: (a -> b) -> SP a b
mapSP f = GetSP (\m -> PutSP (f m) (mapSP f))

mapstateSP :: (a -> b -> (a, [c])) -> a -> SP b c
mapstateSP f s = GetSP
    (\m -> let (s',ms) = f s m in putsSP ms (mapstateSP f s'))
        where putsSP [] c = c
              putsSP (m:ms) c = PutSP m (putsSP ms c)
```

Figure 6.28: Simulating Fudget Stream Processors in Gadgets.

```
data Click = Click

buttonF :: String -> F Click Click
buttonF t i o =
    spawn (mapC id i o) $
    button' (picture (text fn co t)) o Click
        where fn = "*lucida*"
              co = col "black"

intDispF :: F Int a
intDispF i _ =
    wire $ \w ->
    spawn (mapC const i (op w)) $
    display' (editorInput (ip w)) 0
```

Figure 6.29: Wrapping Gadgets to act as Fudgets.

Fudgets are implemented. We must also interpret low-level commands and generate low-level events if Fudgets that use these are to work in the simulator. However, the point of this exercise is to show that the structures of a Fudget program can be simulated in a Gadget program, and we have achieved this.

6.10 A Gadget Simulator in Fudgets?

Motivation

If we can simulate Fudgets on top of Gadgets, can we also simulate Gadgets on top of Fudgets?

Problem

Whilst we can make a Fudget out of a Gadget with two wires, we cannot simply add an arbitrary number of message streams to a Fudget to make a Gadget.

Solution

Once again Thomas Hallgren and Magnus Carlsson were first to attempt a solution. It was not perfect, but worked sufficiently well to demonstrate the feasibility of Gadget simulation. They make use of existential types to allow Fudgets to simulate Gadgets by receiving and emitting a stream of differing types of messages paired with their `WireId`s. Wire connections remain statically type-checked, due to the typing of functions like `from`, `tx`, `ip` and `op`.

Simulated Gadgets do not connect to a Screen Manager. Instead, a wrapping function intercepts SM messages and translates them to low-level Fudget messages, and vice-versa. The Fudget mechanisms for layout and screen redrawing are used. As in the Fudget simulator, simple Gadgets such as `button` are defined by wrapping Fudgets to give them a Gadget type. A simulated version of the Gadget Explode game has been seen to operate almost perfectly, and significantly faster due to Fudget programs being compiled rather than interpreted. The simulator did not interpret screen clipping `ScreenCmnd`s properly, so screen update was not always correct. Notably, all aspects of message-passing and process scheduling are handled in the functional domain. A scheduler function holds lists of messages in wires and lists of processes in queues, both existentially typed.

Evaluation

With the benefit of hindsight, and considering the amount of effort that went into our implementation (see Chapter 4), it might have been better to make use of existential types and keep more of the system functional. However, this solution was not apparent at the time this work was started.

6.11 Summary

In this chapter we have seen many examples of Component and Gadget programs. We have seen how easy they are to construct, and a sample of what can be achieved. Where appropriate, the advantages of using a lazy functional language have been highlighted.

Through example applications we have demonstrated Gadget composition, layout and resizing. We have seen how a client/server structure can be used to share data. A simple game has shown uses of concurrency apart from the construction of GUIs. The Escher Tile program served as a larger application and a demonstration of radio buttons. Viewports illustrate the usefulness of the Gadget image hierarchy. Finally we compared the expressiveness of Gadgets against Fudgets by simulating each in the other.

Chapter 7

Conclusions and Future Work

In this chapter we look back at our aims and requirements and assess whether we have been successful in meeting them. We summarise some lines of future work in Gadget Gofer that have been suggested at various points in the thesis.

7.1 Conclusions

Our aims, defined in Chapter 1, are to enable GUI programs to be written in a lazy functional language whilst retaining the benefits of lazy evaluation, abstraction, higher-order functions and polymorphism. These give rise to more concise programs using functions that are simple to generalise and re-use. These features are important for GUI programs where common elements appear repeatedly and in different forms.

7.1.1 Concurrency

Concurrency is useful for breaking up the large number of events in a graphical program into manageable, independent chunks. Concurrency removes the need to combine all event processing functions into a single poll-and-dispatch point. We have seen this in all the programs we have written in Gadget Gofer, even in the simplest *up-down counter* (see §6.3).

Concurrency does not help in the definition of the *inside* of a Component where the receipt of one message may affect the response to a subsequent message. In these situations, a message received alters the *state* of the Component, so the Component must be evaluated in a single thread. The state is threaded through a sequential

evaluation, the order of which is determined at run-time by the order of events, or rather the order of messages received.

Despite the aim to avoid imperative-style programming constructs, the inside of a Component definition is like an event-processing loop. Each `rx` *polls* for events relating to this Component, and *dispatches* messages to the appropriate guard. The problems caused by this are most acute with Components that send requests expecting a response, *and* have other pins to receive messages from. After such a Component has sent a request, it must process messages arriving on other pins until the response comes. The fact that the Component is waiting for a response must somehow be encoded in the Component state.

Too much concurrency? For the most part, concurrency frees us from the burden of specifying the order in which operations will occur, leaving the arrival of input to determine the sequence. Sometimes though we would like more control. For example, suppose a set of Gadgets display the entries in an address book with name, address and telephone number fields. Another Component updates the display by sending new data to each of the displays. In the current system, each display responds by forwarding a new image to the SM, to display the new data. The SM updates the screen after each image is received, resulting in three screen updates. We would prefer the screen to update once, reflecting all three changes in one go. There appears to be no way to control this.

Non-determinism The system described here can exhibit non-determinism, in that program behaviour may depend on the accidental relative timings of messages. However, it is believed that this non-determinism does not affect referential transparency. The non-determinism can be regarded as residing in the implicit world state. Several features of the system described here have been introduced experimentally to provide maximum expressiveness; some of these might have to be restricted to ensure a properly safe system.

Conclusion Though there are some circumstances where processes are difficult to control, the addition of concurrency has greatly improved program expression.

7.1.2 Re-use of Components

There are two ways to re-use a Component. The first and most straightforward way is to use the exact same Component in a new situation. For example, most of the example programs in Chapter 6 make use of the same `button` (see §6.3, §6.5, §6.6, §6.7, §6.8 and §6.4). Giving Components pins has made such re-use easier. A Component is simply *plugged in* to a different *circuit* of Components, with no changes to its definition. Polymorphism is also useful here. The type of messages a Component pin conveys can be polymorphic, and constrained to a particular class of types if necessary. For example, the `mapC show` Component has an input pin that can be connected to a wire carrying messages of any type in the class `Text`. Polymorphism helps make Component definitions more general, so they are applicable to more situations.

The second method of re-use is to use a Component as the basis for a new Component definition. Components can be altered by wrapping them with Components that intercept messages sent in or out. For example, the `pictureStoreButton` in §6.7 is a modification of a `button`. This technique, known as *delegation* [Lie87], is an alternative to using classes and inheritance. It is limited because the intercepting Component has no access to the internal state of the original Component. But it has the advantage that the resulting Component may have a completely different type from the original. This would not be possible using classes and inheritance.

Polymorphism is not the only way to make a Component more general. The range of effects a message can have on a Component can be enlarged if the message is a function to alter part of the Component's state (see §6.3 and §6.7).

We have also seen how to adopt techniques in other systems that make their components more re-usable (see §2.3.1, §2.5.1 and §2.5.2).

Conclusion Specifying communication with pins and wires makes for more re-usable processes.

7.1.3 First-class message values

Encapsulation Being able to pass any first class value unevaluated in a message has many advantages. Defining Components that receive messages containing functions does more than make a Component more re-usable. It enables us to encapsulate state inside a Component without limiting the extent to which it can be

changed from outside. For example, the `boardCell` Gadgets of the Escher program (see §6.7) hold in their state the tile they display. The displayed tile is altered from outside by sending in a tile-changing function.

Dynamic processes Messages can also contain wires or even Components themselves. We have not found a use for passing Components in messages, but passing wire-ends is used in creating connections to the Screen Manager Component. The dynamic creation of Components would be severely limited without this ability — all Components that might communicate with a dynamically created Component would have to be passed at their time of creation the ends of the wires to be used for this communication.

Shared data structures Messages are passed unevaluated. So a single large data structure can be sent to many Components without creating a copy for each one. For example, in the Escher Tile program (see §6.7) a tile design is sent from the `design` Gadget to multiple cells in the `board` Gadget. Each cell actually holds a reference to the same tile in the heap, even though the program appears to make many copies of the tile to send to each cell.

Conclusion The ability to pass any first-class value in a message is one advantage of functional languages we have preserved in our system, and has many uses.

7.1.4 The type of a process

Each Component pin is a separate parameter to the Component function. Why is this important? Consider a radio group of buttons. All the buttons are connected to a radio controller (see §6.7 for a description). Usually the buttons are all of the same type, but what if we wanted to connect some things to the radio group that were not even buttons, but could be turned on or off like a button? For example, a radio group for selecting a font-size may have a small number of fixed sizes, and a box for entering an alternative. Entering a size in the box deselects the other options in the radio group, and selecting one of the fixed options greys-out the entry box. The entry box Component is a different type because it has a pin that emits the size entered, but we can still connect this group to a radio group controller as long as each has the set/notify duplex connection required for a radio group. If

processes were addressed using a single *handle* or *identifer*, it would be more difficult to achieve this effect.

Wire-ends are *passed in* to a Component definition rather than *returned* from the definition. A function defining a Component can be partially applied to connect *some* of its pins, resulting in a function that describes a Component with less pins. This allows us to connect different pins of a Component at different levels in a program. For example, in the Explode program of §6.6 the `cell`s are first connected to the `referee` by partial application, then the resulting Components connected together by the `grid` combinator.

Conclusion The type of a Component, with its separate input and output pin parameters, allows the use of partial applications in constructing a program.

7.1.5 Functional Screen Management

Screen management in the Gadgets system is handled almost entirely in the functional domain, leaving only primitive operations such as line drawing and clipping outside a rectangle to the imperative world. Our implementation of this system displays its entire screen output in a single X window, but only because that was the most convenient way to display screen output. The system does *not* rely on any particular features of The X Windows System and could easily be adapted to use another interface to the screen, mouse and keyboard. Although the system evaluates under an interpreter and the definition is purely functional, execution speed is sufficient for a wide range of applications such as those in Chapter 6. We have demonstrated that it is possible to program the screen management operations of the system functionally.

Conclusion It is feasible to *push back the frontiers* and include screen management in the functional domain.

7.1.6 User-input push meets lazy evaluation pull

The fundamental problem with programming GUIs in a lazy functional language is that user-input in a GUI *pushes* the program to generate some output, whereas in a lazy evaluation the demand for output *pulls* the program to compute a result. The two types of forces have to meet at some point in the program. In our system, they

meet at the `rx` primitive. A potential lazy evaluation is provided for each input pin of a Component. Each one *pulls* tentatively on its pin until the first input arrives, then `rx` commits to that path of evaluation, discarding the other evaluations.

Is this the best place for *pull* to meet *push*? One alternative is to do without the `rx` primitive, and permanently attach an evaluation to each pin. We can achieve this by spawning a separate process to input from each pin. The disadvantage with this is that each evaluation operates in a separate thread and cannot share state. The receipt of a message cannot affect how the Component responds to subsequent messages.

Conclusion Our use of the `rx` choice primitive represents a good balance between the demands of lazy evaluation and computation triggered by user-input.

7.1.7 State in Components

Private local state in Components helps to improve the modularity of a program by hiding information in the process that uses it. Components can hold state as a parameter of a recursive function. Larger, multi-part states can make the function definition untidy and difficult to maintain. We have suggested several ways of solving this problem in §5.7.2.

Conclusion Components can keep state, but there is room for improvement in the maintainence of state in a Component definition.

7.2 Further Developments

In this section we describe some of our plans for extending Gadget Gofer that were conceived during the development but never made it to the implementation stage. These qualify as *improvements* of the present design rather than radical alternatives to it.

7.2.1 More manager Components

The system was structured with the intention of providing *manager* Components for each different kind of I/O resource. Only one manager has been implemented,

for the screen, keyboard and mouse. Outline plans for a timer manager and file manager were described in §5.10.5.

7.2.2 Shaped windows

Gadget images are rectangles whose sides are parallel to the top and side of the screen. The algorithms and data structures of the Screen Manager are dependent on images having these properties, so to modify it for images of other shapes would require changing most of it. However, it would be possible to create images of various shapes and with transparent holes by allowing images to draw their desired shape within a rectangle. To achieve this, a Gadget would record in the SM display data structure that its image has transparent regions. When the SM redraws such an image, it first redraws the images beneath it, then allows the image to draw its shaped image on top of the images already drawn. Mouse-clicks sent to the shaped Gadget may not land inside its shape, so an extra SM request enables the Gadget to return mouse-clicks that *miss* the shape, to be forwarded to the next image below. One disadvantage with this solution is that parts of the screen display will be painted twice to display such an image.

7.2.3 Graphical Composition

Gadget composition provides for piecing together interfaces, but there is little support for describing the picture in each (non-composed) Gadget. In §5.10.4 we described a library of combinators for creating pictures from graphical elements. We suggest that Gadget layout could be incorporated into this library by making Gadgets a type of graphical element.

7.2.4 Gadget sizing

To keep the Screen Manager simple, we opted for a Gadget layout scheme that operates in one direction only — Gadgets request a particular size of image, they cannot be squeezed or stretched into an alternative size image. We briefly discussed attributing *stretchiness* and *squashiness* to Gadgets in §5.10.3, and again as part of the graphical composition ideas of §5.10.4.

7.2.5 Use of more primitive screen drawing commands

Currently, when a Gadget's image is moved, the old and new areas are redrawn using drawing functions. With access to an *area copy* command, this operation could be made more efficient. It is not a simple operation as the image may be partially covered, requiring part copy, part redraw. This issue and a suitable algorithm is discussed in [Mye86].

7.2.6 Scheduling

Two areas related to scheduling have shown need for improvement.

Terminating a Component

We have not been able to create a graphical program in which can *quit* or *close* part of the program (say a certain window). The Screen Manager does not provide a means to remove an image, and although a Component can terminate itself, it is not possible for one Component to terminate another. A Gadget that creates further Gadgets may later want to destroy them. For example, suppose a text-editor Gadget creates menu Gadget and a viewport to display the text, and one of the menu options is to *Quit* the text-editor. Before terminating itself, the text-editor Gadget must destroy the viewport and the menu Gadgets.

One way to achieve this would be to enforce the rule that every Component must have a pin through which another Component can request its termination. On receipt of a `Terminate` message, a Component should send termination messages to all Components that *it* has created, then `end` itself. (This would not be statically checked however.)

Priority scheduling

For the most part, process scheduling based on a count of messages waiting for each process is sufficient, but we have found examples where some form of priority is required (eg. in §6.5). We suggest three types of priority that may be used.

Some programs experience problems of groups of messages *taking over*. For an example, see the description of the *multiple counters* in §6.5. There are three possibilities for using priority to influence scheduling in a program:

Prioritising messages By attaching a priority to each message sent the scheduling decision can be based on the sum of the priorities of messages waiting for each Component, instead of the number of messages waiting. The priority could be attached by the process sending the message.

Prioritising Components If Components are given a priority when created, the scheduling decision can be based on the number of messages waiting multiplied by the priority of the Component. The priority could be decided by the Component's parent.

Prioritising wires If wires are given a priority when created, the scheduling decision can be based on the sum of wire priorities of the messages waiting for a Component. The priority could be decided at the point of wire allocation.

7.2.7 Demand propagated back along wires?

We have already explained how lazy evaluation's demand for values stops at the `rx` primitive. It would be interesting to see if *demand* can be propagated back along wires, so that messages are only transmitted if there is a need for them at the other end of the wire. This would be like synchronous message-passing, but message transmission would be motivated by the receiving instead of the transmitting process.

7.2.8 Making functional GUIs more functional

Despite the formulation of screen management in the functional language, much of this system remains outside of the functional domain. Process creation, scheduling and message passing are all handled by approximately 1850 lines of the imperative language C. The main part of the functional side of the Gadgets system consists of approximately 1900 lines of Gofer, with an additional 1350 lines defining a small set of library Gadgets.

We would prefer a smaller set of simpler primitives that allow us to express the scheduling algorithm and message passing protocols functionally, and therefore more concisely and more securely. The author appreciated the purity of the functional world even more as hundreds of errors in the imperative code passed through the compiler only to generate a run-time error. Mis-directed pointers ran amuck and

variable's contents were not as expected time and time again. By comparison, the functional side of the Gadget Gofer system had a much higher success rate, the type-checking phase spotting most errors.

The reason so much of the system is imperative is that there is no type we can define in Gofer to describe the set of wires in a program and the messages held in them. This is because each wire is a different type. With access to existential types [Aug93, Lau94] we could describe the set of wires and processes functionally. The Gadget simulator in Fudgets, described in §6.10, shows the feasibility of this. The simulator is complicated by the fact that it is a combination of two systems. It remains to be seen how much *purer*, more *concise* and *understandable* a system built from scratch using existential types would be.

Appendix A

Gadget Gofer Manual

This extended version of Gofer provides types and primitive functions for defining concurrent processes. A program consists of multiple processes, evaluating concurrently. A single process is created initially. Further processes are created during evaluation of the program. Processes communicate with one another by message-passing. Processes have *pins* that are joined with *wires* to define the lines of communication in a program. We call processes with pins, *Components*. Pins and wires are *typed* — they convey messages of a particular type. Components can be *composed* to form more complex Components.

A special type of Components, called *Gadgets*, have an image on the screen. Gadgets are composed to build GUIs.

This manual is split into two sections. The definition of basic Components and compositions are explained in §A.1, together with some programming tips. Section A.2 describes Gadgets. Gadget compositions are explained, and some simple library Gadgets introduced.

A.1 Component Gofer

Here is a simple Component that implements a memory cell:

```
memory :: m -> In m -> In t -> Out m -> Component
memory m i t o =
    claim t $
    claim i $
    memory' m where
    memory' h =
        rx [ from i $ \m -> memory' m,
             from t $ \_ -> tx o h $ memory' h ]
```

It can be seen from the type of the `memory` function that, given an initial value to memorise, the input-end of a wire of type `t`, the input-end of a wire of type `m` and the output-end of a wire of type `m`, it returns a Component.

The `memory` Component stores messages received on its first pin, overwriting any previously stored value. A message received on its second pin triggers the `memory` to emit its contents on the third pin.

A.1.1 Continuation passing style

One of the benefits of lazy functional languages is that the order of evaluation is not important. However, where I/O is concerned, there are some cases where the order is important. Within a Component definition we use the continuation passing style (CPS) to restrict evaluation to a particular order. Each I/O function in a Component's definition takes an extra parameter that is *the rest of the program*. Instead of returning a value directly, the function instead passes it on to the continuation function. The `$`s save us from having to write many nested brackets: for example, we write `f $ g $ h` instead of `f (g h)`.

A.1.2 I/O with no side-effects?

There is an extra value being passed through the continuations of a Component definition, hidden by some abstraction. This value represents the state of the I/O world (the contents of all the wires in the world of Components). The primitives that perform I/O manipulate this world value to indicate any I/O that is to occur.

A.1.3 Types and Functions

A Component is a state-transforming process with state `ComponentState`:

```
type Component = Process ComponentState
type Process s = s -> s
```

The exact representation of ComponentState does not concern the programmer. Suffice it to say that the state includes the *world* value used to perform I/O.

Wires, input pins and output pins have types:

```
data Wire m
data In m
```

```
data Out m
```

where `m` is the type of message they convey. Again, the exact representation is not important.

Functions `ip` and `op`, applied to a wire, return the ends that connect to an input and output pin respectively:

```
ip :: Wire m -> In m
op :: Wire m -> Out m
```

Often, an input and output pin on a Component are paired because they are used together. A shorthand is provided for this:

```
type InOut a b = (In a, Out b)
```

An `InOut` can be viewed as a two-way pin. A *duplex* wire that connects two pairs of `InOut` pins has the type:

```
type Duplex a b = (InOut a b, InOut b a)
```

A.1.4 Primitive Component functions

Primitive functions support the definition of Components. Most of the primitives used within the definition of a Component have one or more *continuation* parameters.

`launch :: Process s -> ()` starts a Component Gofer program running. It is the only primitive of the system used outside of a Component. The library function `go` (defined in terms of `launch`) is used to start a *Gadget* program.

`wire :: (Wire b -> Process s) -> Process s` supplies a new wire to the continuation, for use in connecting Component pins together.

`spawn :: Process a -> Process s -> Process s` creates a Component.

`claim :: In b -> Process s -> Process s` is used to claim a wire that a Component will receive messages from. If you forget to claim a wire then the Component may not evaluate or a run-time error may occur. You can see in the `memory` example above that the first thing the Component does is to claim the two wires it will be receiving from.

`disown :: In b -> Process s -> Process s` is the opposite of `claim`. After a wire has been disowned, it can be claimed by another Component.

`rx :: [Guarded s] -> Process s` receives a message sent to the Component. It is applied to a list of *guards* (see `from` below), one for each input pin that the Component has claimed. The guard that matches the next message supplies the continuation to use.

`from :: In b -> (b -> Process s) -> Guarded s` applied to an input pin and a continuation this primitive returns a guard (see `rx` above). The continuation is applied to the message when it is received. In the `memory` example above, `rx` is used together with two `from` guards, one for the input pin, and one for the trigger pin.

`tx :: Out b -> b -> Process s -> Process s` transmits a message on a pin. Messages *queue* in wires, so the `tx` primitive does not wait for the message to be received before continuing.

`end :: Process s` terminates the Component that evaluates it. It is not possible to terminate another process.

A.1.5 Functions

`sequence :: [Component -> Component] -> Component -> Component` performs a list of actions. For example, the following expression emits the numbers `1` to `5` in separate messages via pin `o`:

```
... sequence (map (tx o) [1..5]) $ ...
```

`accumulate :: [(a -> Component)] -> ([a] -> Component) -> Component` performs a list of actions that pass on a value to their continuation, collecting the values passed on, then passes all the values in a list to the rest of the program. For example, this expression evaluates five lots of `wire` collecting the five wires that result:

```
... accumulate (copy 5 wire) $ \ws -> ...
```

`wires :: Int -> ([Wire a] -> Component) -> Component` obtains a specified number of wires (all of the same type) in a list.

`duplex :: (Duplex a b -> Component) -> Component` supplies a pair of wires arranged back-to-back (ie. one part of the pair is the input of the first wire with the output of the second wire, and vice-versa).

A.1.6 Component Compositions

A Component can be defined directly (like `memory`), or it can be defined as a composition of other Components. The Components of a composition will remain separate as far as scheduling is concerned, but will appear to the rest of the program to be a single Component. For example, two memories could be composed to form a `dual_memory` that can store two values, both triggered by a single message.

```
dual_memory :: m -> n -> In m -> In n -> In t -> Out m -> Out n -> Component
dual_memory s1 s2 i1 i2 t o1 o2 =
    wire $ \a ->
    wire $ \b ->
    spawn (tee t (op a) (op b)) $
    spawn (memory s1 i1 (ip a) o1) $
    spawn (memory s2 i2 (ip b) o2) $
    end
```

The `tee` Component duplicates messages received on its input on two outputs. Notice that the `dual_memory` process exists only long enough to wire up and spawn the three Components it is composed of, before it ends.

A.1.7 Components with Lists of Pins

A Component may have a list of pins, the number of which is unknown at the time of definition. For example, the `tag` Component has a list of input pins. Messages arriving on one of the inputs are tagged with a number unique to the pin and output:

```
tag :: [In a] -> Out (a,Int) -> Component
tag is o =
    sequence (map claim is) $
    let fs = zip is [1..] in
    tag' where
    tag' = rx [froms fs $ \m n -> tx o (m,n) $ tag']
```

A.1.8 Creating New Pins on a Component as it Operates

For example, suppose a Component `comp` is to create a new Component `child`, connecting its single output to a new input on itself:

```
comp :: Component
comp =
    wire $ \w ->
    spawn (child (op w)) $
    claim (ip w) $
    rx [from (ip w) $ \m -> end]
```

A.1.9 Programming Server Components

Imagine `client1` has a duplex connection to a server, sending messages of type `ServerRequest` and getting back messages of type `ServerReponse`. One of the requests adds a new connection to the server. `client1` creates a new Component `client2`, attaching wires to the pins that will connect to the server, then passes the other ends of the wires to the server through it's own connection. The server accepts these new wires as a new connection.

```
data ServerRequest = NewConnection (InOut ServerRequest ServerResponse) | ...
data ServerReponse = ...

client1 :: InOut ServerResponse ServerRequest -> Component
client1 (i,o) =
    duplex $ \(a,b) ->
    spawn (client2 a) $
    tx o (NewConnection b) $ ...
```

A.2 Gadgets

A.2.1 The relationship between Gadgets and Components

A Gadget is a special type of Component. It has a connection to the screen manager Component that gives it an image on the screen. Gadgets can also be composed, but as well as specifying the connections between Gadgets in a composition, their relative layout on the screen is also specified.

A.2.2 The Type of a Gadget

Like a Component, a Gadget is a type of process:

```
type Gadget = Process GadgetState
```

The Gadget state contains the ends of the wires that connect the Gadget to the Screen Manager. The programmer does not deal directly with the Gadget state, but uses library functions (described below) to access it.

A.2.3 Launching a Gadget Program

Gadget programs require an *environment* to be set up before they can execute. A Gadget program is launched using the `go` function:

```
go :: [(Gadget, String)] -> ()
```

`go` is applied to a list of Gadgets paired with a title for the window they will be displayed in. All the Gadgets in the list are created and displayed in windows.

A.2.4 Library Gadgets

Buttons, icons, windows and text displayers are all Gadgets. For example, here is a button:

Figure A.1: A Button.

The button is a Gadget with a single output on which a message is sent whenever the button is clicked. The type of a simple button is:

```
button :: Out m -> m -> Process s -> Process s
```

The *attributes* of a Gadget (for example its size and the picture displayed in it) assume default values for a `button` There is also a `button'` function in which some or all of the attributes can be set. For example, a button with size 40 by 20 is defined:

```
button' (width 40.height 20) o m
```

where `o` is an output pin and `m` is the message output when the button is clicked. The *attribute modifiers*, `width` and `height` can be applied to all primed (`'`) Gadgets. Some other common attribute modifiers are:

- `border`, applied to an integer to set the size of a border around a Gadget;
- `picture`, applied to a drawing function to set the picture displayed in a Gadget;
- `pictureIn`, applied to the input-pin end of a wire through which the picture in a Gadget can be changed at a later stage by sending a drawing function;
- `fgcol` and `bgcol`, applied to a colour to set the foreground or background colour of the Gadget;

In addition to these, buttons have their own private set of attribute modifiers:

- `buttonMomentary` makes the button a *momentary push* type, which presses in when clicked and pops out when released;
- `buttonPonPoff` makes the button a *push on, push off* type, which presses in when clicked and pops out when clicked again;
- `buttonPon` makes the button a *latching* type, which presses in when clicked and must be popped out using `buttonSet`. This is used for radio-buttons.
- `buttonSet`, applied to the input-pin end of a wire through which the state of a button can be set by sending a boolean value (True for pushed in, False for popped out);
- `buttonNotify`, applied to the output-pin end of a wire through which the state of a button is sent whenever the button changes state;

Other Gadgets in the library are:

- `editor` a text entry box;
- `paper` a box in which lines can be drawn with the mouse;
- `bargraph` a simulation of a LED bargraph display;
- `wall` an area on which windows can be placed.

The types of these Gadgets can be discovered from within the Gofer interpreter (eg. type "`:i editor`"), and their use is fairly intuitive.

A.2.5 Gadget Composition

Like Components, Gadgets may be composed, but their screen layout must be specified as well as connections between them. Layout is defined using *layout combinators* such as `<->` and `<|>`. For example, two buttons may be composed horizontally with:

```
twobut :: Out m -> m -> Gadget
twobut o m = button o m <-> button o m
```

We are using the fact that more than one process may output on the same wire to make both buttons send their message from the same output on the composition. In a Gadget composition we do not have to `spawn` the Gadgets as we do for Components, as the layout combinators create them.

Components can also be connected into a Gadget composition. For example, we can create a button with two outputs by connecting a `tee` Component to it:

```
doublebutton :: Out m -> Out m -> m -> Gadget
doublebutton a b m =
    wire $ \w ->
    spawn (tee (ip w) a b) $
    button (op w) m
```

There are a selection of layout combinators, some binary operators and some for lists, some centering and others justifying to one side:

```
<<> :: Gadget -> Gadget -> Gadget
<>> :: Gadget -> Gadget -> Gadget
<^> :: Gadget -> Gadget -> Gadget
<.> :: Gadget -> Gadget -> Gadget
top :: [Gadget] -> Gadget
beside :: [Gadget] -> Gadget
bottom :: [Gadget] -> Gadget
left :: [Gadget] -> Gadget
above :: [Gadget] -> Gadget
right :: [Gadget] -> Gadget
```

A.2.6 Defining a Gadget Directly

Normally the programmer should not have to define any Gadgets directly, as the library Gadgets should provide a rich enough base on which to build any interface. However, if a Gadget is required that cannot be built from library Gadgets, it is simple to define one directly. Three library functions are needed:

`initGadget` is applied to a size (a pair of width and height) drawing function and continuation to initialise the size and picture of a new Gadget;

`fromSM` is equivalent to `from` applied to the input wire from the Screen Manager. It is used to receive `SMResponse` messages.

`txSM` is equivalent to `tx` applied to the output wire to the Screen Manager. It is used to send `SMRequest` messages.

For an explanation of drawing functions, `SMRequest`s and `SMReponse`s, see Chapter 5, *"Screen Management"*.

Bibliography

[Ahl93] C. Ahlberg. GUIT — A Graphical User Interface Builder for the Fudgets Library. In *Proceedings of the PMG Winter Meeting*, January 1993.

[Ary89] Kavi Arya. Processes in a Functional Animation System. In *FPCA '89 Conference Proceedings*, pages 382–395, 1989.

[Aug93] Lennart Augustsson. *Haskell B user's manual*, July 1993. distributed with the Haskell B compiler.

[AWV93] J. Armstrong, M. Williams, and R. Virding. *Concurrent Programming in Erlang*. Prentice-Hall, 1993.

[BWW86] J. Backus, J. Williams, and E. Wimmers. FL Language Manual. Technical report, IBM Almaden Research Centre, 1986.

[CH93] M. Carlsson and T. Hallgren. FUDGETS - A Graphical User Interface in a Lazy Functional Language. In *Functional Programming & Computer Architecture*, March 1993.

[DH93] D. J. Duke and M. D. Harrison. Abstract interaction objects. Technical report, HCI group, Dept. of Computer Science, University of York, 1993.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[Dix87] Alan Dix. Giving control back to the user. In H J Bullinger and B Shackel, editors, *Proceedings of IFIP INTERACT'87: Human-Computer Interaction*, 2. Design and Evaluation Methods: 2.3 Goals and Guidelines for Design, pages 377–382. North-Holland, 1987.

[Dwe87] A. Dwelly. Synchronising the I/O Behaviour of Functional Programs with Feedback. *Information Processing Letters 28 (1988) 45–51*, December 1987.

[Dwe89] Andrew Dwelly. Functions and Dynamic User Interfaces. *Proceedings of the Conference on Functional Programming Languages and Computer Architecture, Imperial College, London*, pages 371–381, 1989.

[FJ95] Sigbjørn Finne and Simon Peyton Jones. Composing Haggis. In *Proceedings of the Fifth Eurographics Workshop on Programming Paradigms in Computer Graphics*, Maastrict, Netherlands, September 1995.

[Fou95] S. P. Foubister. *Graphical Application and Visualisation of Lazy Functional Computation.* Dphil, Dept. of Computer Science, University of York, May 1995.

[FR91] S. Foubister and C. Runciman. After Escher...Patterning Graphical Interaction in the Functional Style. In *ICYCS '91*, pages 151–155, 1991.

[GR93] Emden R. Gansner and John H. Reppy. A Multithreaded Higher-order User Interface Toolkit. *User Interface Software: Sotware Trends*, 1:61–80, 1993.

[HC93] Ian Holyer and David Carter. Concurrency in a Purely Declarative Style. In *1993 Glasgow Workshop on Functional Programming.* Springer Verlag Workshops in Computing Series, 1993.

[HC95] Ian Holyer and David Carter. Deterministic Concurrency. Technical Report CSTR-95-015, Department of Computer Science, University of Bristol, June 1995.

[HD88] K. Hanna and Neil Daeche. Computational Logic: An Algebraic Approach. Technical report, Faculty of Information Technology, The University of Kent at Canterbury, May 1988.

[Hen82a] Peter Henderson. Functional Geometry. *Conference record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 179–187, 1982.

[Hen82b] Peter Henderson. Purely Functional Operating Systems. In J. Darlington, Peter Henderson, and David A. Turner, editors, *Functional Programming and its Applications*, pages 177–192. Cambridge University Press, 1982.

[HHJW94] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip Wadler. Type Classes in Haskell. In *European Symposium on Programming*, volume 788 of *LNCS*. Springer-Verlag, April 1994.

[HM89] S. Hayes and L. McLoughlin. Imperative Effects from a Pure Functional Language. Technical report, Hewlett-Packard Laboratories Memo HPL-ISC-TM-89-120, August 1989.

[HM95] Thomas Hallgren and Carlsson Magnus. Programming with Fudgets. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 137–182. Springer, May 1995.

[Hoa80] C. A. R. Hoare, editor. *Functional Programming Application and Implementation*. Series in Computer Science. Prentice-Hall International, 1980.

[HS88] P. Hudak and R. S. Sundaresh. On the Expressiveness of Purely Functional I/O Systems. Technical report, Yale University Research Report YALEU/DCS/RR-665, Dept. of Computer Science, December 1988.

[Hug90] R. J. M. Hughes. Why functional programming matters. In D. A. Turner, editor, *Research Topics in Functional Programming*, pages 17–42. Addison-Welsey, Reading, MA, 1990.

[JH93] Mark Jones and Paul Hudak. Implicit and Explicit Parallel Programming in Haskell. Technical Report YALEU/DCS/RR-982, Department of Computer Science, Yale University, August 1993.

[JL91] S. L. Peyton Jones and J. Launchbury. Unboxed Values as First Class Citizens in a Non-Strict Functional Language. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, pages 636–666. Springer-Verlag Lecture Notes in Computer Science 523, August 1991.

[Joh87] Thomas Johnsson. Attribute Grammars as a Functional Programming Paradigm. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer*

Science, pages 154–173. Springer-Verlag, New York–Heidelberg–Berlin, September 1987. Portland.

[Jon92] S. L. Peyton Jones. UK Research in Functional Programming. *SERC Bulletin*, 4(11):24–25, 1992.

[Jon93] Mark Jones. The GOFER Functional programming environment. Technical report, Yale University, 1993.

[Jon94] Mark P. Jones. The Implementation of the Gofer Functional Programming System. Technical Report YALEU/DCS/RR-1030, Department of Computer Science, Yale University, New Haven, Connecticut, USA, May 1994, May 1994.

[JP93] SL. Peyton Jones and Wadler P. Imperative Functional Programming. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages, Charleston, South Carolina*, pages 71–84, January 1993.

[Kar81] Kent Karlsson. Nebula: A functional operating system. Programming Methodology Group Memo LPM11, Laboratory for Programming Methodology, Chalmers University of Technology, Goteborg, 1981.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[Lau94] Konstantin Laufer. Combining Type Classes and Existential Types. In *Proceedings of the Latin American Informatics Conference*, Mexico, 1994.

[Lie87] H Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 214–223, Orlando, FL USA, December 1987. ACM Press , New York, NY , USA. Published as SIGPLAN Notices, volume 22, number 12.

[LJ94] J. Launchbury and S. Peyton Jones. Lazy Functional State Threads. In *Programming Languages Design and Implementation*, Orlando, 1994. ACM Press.

[Mye86] Brad A. Myers. A Complete and Efficient Implementation of Covered Windows. *Computer*, 19(9):57–67, September 1986.

[O'D85] J. O'Donnell. Dialogues: A Basis for Constructing Programming Environments. *SIGPLAN Notices*, 20(7):19–27, 1985.

[Oka95] Chris Okasaki. Simple and Efficient Purely Functional Queues and Deques. *Journal of Functional Programming*, October 1995.

[Per92] N. Perry. Towards a Concurrent Object / Process Oriented Functional Language. In *Proceedings of the 15th Australian Computer Science Conference*, pages 715–730. Australian Computer Science Communications 14, January 1992.

[PJGF96] Simon Peyton Jones, Andrew Gordon, and Sigbjørn Finne. Concurrent Haskell. In *Symposium on Principles of Programming Languages*, January 1996.

[PvE93] M. J. Plasmeijer and M. C. J. D. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison Wesley, 1993.

[PvE95] Rinus Plasmeijer and Marko van Eekelen. Concurrent Clean Language Report. Technical report, University of Nijmegen, April 1995.

[RS93] A. Reid and S. Singh. Implementing Fudgets with Standard Widget Sets. In *Glasgow functional programming workshop*, pages 222–235. Springer-Verlag, 1993.

[Ser95] Pascal Serrarens. BriX — A Deterministic Concurrent Functional X Windows System. Technical report, Department of Computer Science, University of Bristol, June 1995.

[SG86] Robert. W. Scheifler and Jim Gettys. The X Window System. *ACM Transactions on Graphics*, 5(2), 1986.

[Sin89] Duncan Sinclair. Graphical User Interfaces for Functional Languages. Technical report, Dept. of Computer Science, University of Glasgow, May 1989.

[Sin91] S. Singh. Using XView / X11 from Miranda. In *Proceedings of the Fourth Annual Glasgow Workshop on Functional Programming, 13-15th August*, pages 592–607, Skye, August 1991. Springer-Verlag.

[Sin92] Duncan C. Sinclair. Graphical user interfaces for Haskell. In J. Launchbury and Patrick M. Samson, editors, *Glasgow functional programming workshop*, pages 252–257. Springer-Verlag, 1992.

[Sto85a] W. Stoye. A new scheme for writing functional operating systems. In Lennart Augustsson, John Hughes, Thomas Johnsson, and Kent Karlsson, editors, *Proceedings of the workshop on implementation of functional languages*, pages 357–389, Chalmers University of Technology, February 1985.

[Sto85b] William Stoye. The Implementation of Functional Languages using Custom Hardware. Technical Report 81, University of Cambridge Computing Laboratory, December 1985.

[Sto86] W. Stoye. Message-based Functional Operating Systems. *Science of Computer Programming*, 6(3):291–311, 1986.

[Tho86] S. J. Thompson. Writing Interactive Programs in Miranda. Technical Report 40, Computing Lab., University of Kent, August 1986.

[Tho87] S. J. Thompson. Interactive Functional Programs: a method and a formal semantics. Technical Report 48, Computing Lab., University of Kent, November 1987.

[Tur87] David Turner. Functional programming and communicating processes. In *PARLE '87*, number 259 in LNCS, pages 54–74. Springer-Verlag, 1987.

[vEHN+93] Marko van Eekelen, Halbe Huitema, Eric N¨ocker, Sjaak Smetsers, and Rinus Plasmeijer. Concurrent Clean language manual. Technical report, University of Nijmegen, February 1993.

[VTS95] Ton Vullinghs, Daniel Tuinman, and Wolfram Schulte. Lightweight GUIs for Functional Programming. In Manuel Hermenegildo and

S. Doaitse Swierstra, editors, *PLILP'95: Seventh International Symposium on Programming Languages, Implementations, Logics and Programs*, volume 982 of *Lecture Notes in Computer Science*, pages 341–356. Springer-Verlag, September 1995.

[Wad90] P. Wadler. Comprehending Monads. In *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 61–78, 1990.

[Wad92] Philip Wadler. The Essence of Functional Programming. In *Principles of Programming Languages*, January 1992.

[Wal95] Malcolm Wallace. *Functional Programming and Embedded Systems.* Dphil, Dept. of Computer Science, University of York, January 1995.

[WR94] Malcolm Wallace and Colin Runciman. Type-checked message passing between functional processes. In *Proceedings of the Seventh Annual Glasgow Workshop on Functional Programming*, pages 245–254. Springer-Verlag, September 1994.

[Yal93] Yale. Yale Haskell X Interface (Preliminary Version). Technical report, Yale University, 1993.