Lecture Notes

4 augusti 1995

Programming with Fudgets[1]

Thomas Hallgren & Magnus Carlsson

Computing Science, Chalmers University of Technology,
S-412 96 Göteborg, Sweden.
E-mail: hallgren, magnus@cs.chalmers.se

1 Introduction

In these notes we present the Fudget Library and the ideas underlying it. The Fudget Library is primarily a toolkit for the construction of Graphical User Interfaces (GUIs) on a high level of abstraction in the lazy functional language Haskell, but it also allows you to construct programs that communicate across the Internet with other programs.

Apart from describing how to use the Fudget Library, we try to describe the underlying ideas in such a way that the reader should be able to use them in his/her own favourite functional language.

The design of the Fudget Library started with the desire to find a good abstraction of GUI building blocks, i.e., an abstraction that makes use of the powerful abstraction mechanisms found in functional languages (higher order functions, polymorphism, etc.) and thereby, hopefully, is better than the abstractions you find in typical GUI toolkits for conventional, imperative languages. We consider an abstraction to be better if it simplifies programming, e.g., by making programs more concise and thereby easier to write, read and maintain.

An additional consideration is that in today's programming language implementations, there usually is a conflict between efficiency and high level of abstraction, so a good abstraction is one that can have a reasonably efficient implementation. If we can not have that, we have lost contact with the real world. To summarise:

> *It is important not to lose contact with the real world, but this does not imply that one must pass around the world explicitly.*

The main abstraction used in the Fudget Library is the *fudget*. A fudget is a process which can, via message passing, communicate with other concurrently running fudgets and with the outside world. A fudget is a first class value of a type that reflects what types of messages the fudget sends and receives. This makes communication type safe. A fudget may have an internal state, which is not visible in the type of the fudget. Fudget programming in this respect resembles object oriented programming, where state information is distributed and hidden within objects rather than centralized and exposed to arbitrary use or misuse. But the encapsulation of state information also makes fudgets easy to compose, like functions in functional languages.

Fudgets are implemented on top of *stream processors*, a simpler kind of process that communicates with its surroundings through an input stream and an output stream of values.

1. From LNCS 925, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 1995. Also at ftp://ftp.cs.chalmers.se/pub/cs-reports/papers/fudgets-springschool.ps.Z

1.1 Overview

We start with a quick recapitulation of some common I/O methods in functional languages (Section 2). In these methods you specify a single-threaded sequence of I/O operations, so the functional program in effect takes the form of a sequential imperative program on the top level.

For reactive programming (Section 2.3), a more attractive program structure is a set of concurrent processes, so we introduce *stream processors* (Section 3). A stream processor is a process that consumes an input stream and produces an output stream. Combinators for serial composition, parallel composition and loops allow programs to be structured as a network of stream processors. Stream processors can be programmed purely functionally.

In addition to communicating with neighbours in a network, in a reactive programming context many stream processors will also need to communicate with external entities through the I/O system. We therefore introduce *fudgets* (Section 4), stream processors which have access to the I/O system in addition to streams for communication with other stream processors or fudgets. Reactive programs can be built as networks of fudgets.

The main use of fudgets is the construction of Graphical User Interfaces (GUIs). The building blocks in GUIs (buttons, menus, sliders, etc.) are reminiscent of physical devices in that they are self-contained units that operate more or less independently and in parallel. The reactive programming model is thus very natural for GUIs. In the Fudget Library, GUI elements are represented as fudgets.[2] Complex user interfaces are built by combining fudgets representing GUI elements and other stream processors (Section 5).

There are two aspects in the design of GUI programs with fudgets: the computational aspect and the visual aspect. The fudget system allows you to worry about them one at a time. Thanks to the automatic layout system you can concentrate on the computational aspect during the initial development stage. You can later add layout information to the program, if the default layout isn't adequate (Section 5.5).

When designing software libraries, e.g., GUI toolkits, there is often a tension between generality and simplicity. Generality is often achieved by using many parameters. Having to give values for a lot of parameters clearly makes library components more difficult to use. In some programming languages there is a mechanism that allows a function parameter to be omitted if the function definition specifies a default value for it. This makes functions easy to use and customizable at the same time. The language that we use (Haskell) does not have such a mechanism, but the scheme used in the fudget library comes pretty close. It uses Haskell's type class system to avoid proliferation of names (Section 5.6).

The second use of fudgets that we cover is the construction of network based client/server programs. A typical server must be able to handle connections from several simultaneous clients, so it is useful to structure a server with a handler process (i.e., a handler fudget) for each client. Programs written in Haskell with the fudget library can communicate with programs written in other languages, but for the case where all programs involved are written in Haskell we show a simple way to make sure that the communication is type safe (Section 6).

2. The word fudget comes from *fu*nctional wi*dget*, where widget comes from *wi*ndow ga*dget*.

1.2 A First Example

As a preview of Graphical User Interface construction with the Fudget Library, Figure 1 shows a small program: a simple counter. The user interface contains a button and a numeric display. When you press the button the number in the display is incremented.

The core of the program is the definition of `counterF`, where two fudgets implementing the two user interface elements and a stream processor implementing the click counter are connected using the serial composition operator `>==<`. Data flow from right to left. The button outputs clicks are fed to the counter. For every click, the counter increments its internal integer state and outputs the new value to the display.

Readers mainly interested in GUI construction may want to skip directly to Section 5 and then go back to the earlier sections to learn more about what stream processors and fudgets really are.

```
module  Main(main) where -- A simple counter

import Fudgets

main :: Dialogue
main = fudlogue (shellF "Counter" counterF)

counterF = intDispF >==< absF countSP >==< incButtonF

incButtonF :: F Click Click
incButtonF = buttonF "Increment"

countSP :: SP Click Int
countSP = putSP startstate $
          mapAccumlSP inc startstate
  where inc n Click = (n+1,n+1)

startstate = 0
```

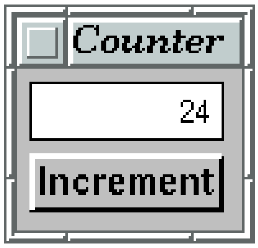


Fig. 1. The Counter Example

1.3 Notation

All programs in these lecture notes are written in the pure functional language Haskell [7]. We deviate from Haskell syntax on two points:

- We write → instead of `->`.
- We write `a+b` instead of `Either a b`, the standard sum type in Haskell, defined as

```
data Either a b = Left a | Right b
```

This proviso aside, the presented examples should compile and run "as is".

To avoid nested bracketing in large expressions we will often use the infix operator `$` defined as

```
f $ x = f x
```

`$` is right associative and has low precedence, so you can write, e.g.,

```
f $ g $ h $ \x → x+1
```

instead of

```
f (g (h (\x → x+1)))
```

Note that in Haskell, functions can be used as infix operators by placing them in backquotes. For example, `parSP` (parallel composition of stream processors) is a function taking two arguments. We will write

```
sp1 `parSP` sp2
```

instead of

```
parSP sp1 sp2
```

2 Input/Output in Functional Languages

In this chapter we give a brief introduction to Input/Output in functional languages. Several models of I/O for lazy functional languages have been developed during the years. Good surveys can be found in [3] and [6]. Here, we present Landin's stream model of I/O and the synchronized streams used in Haskell. We present continuation based I/O and monadic I/O as abstractions from streams and note that they are sequential in nature. We note that for some purposes it is more natural to use a set of concurrent processes than a single-threaded sequence of I/O operations to describe the I/O behaviour of a program.

2.1 Referential Transparency and I/O?

In traditional imperative languages Input/Output operations are usually accomplished by calling some predefined procedures that perform the desired operation as a side effect. In functional languages, however, side effects are usually not wanted, since they break *referential transparency.* As an example, suppose there was a function

```
write :: String → String
```

which would take a string and return it unchanged and, as a side effect, print the string on the terminal. Then the program

```
let s = write "Ha"
in (s,s)
```

would probably not produce the same result as

```
(write "Ha", write "Ha")
```

This means that many nice algebraic laws, such as

$$2*a = a+a$$

$$a+b = b+a$$

no longer hold for arbitrary subexpressions a and b.

Still, this is how I/O works in most strict functional languages like LISP and Standard ML. In a lazy language you don't really want to think about *when* or *if* functions are actually called, so specifying I/O in this way is not very useful.

2.2 Programming Styles for Sequential I/O

In order to maintain referential transparency, I/O in functional languages is *not* thought of as something that happens as a side effect of calling certain functions. Instead the program is thought of as a pure function from some input to some output.

2.2.1 Landin's Stream Based I/O model

Suppose the only I/O operations we want in a program are reading from the computer keyboard and writing to the computer screen (Figure 2). Then the program can be a function

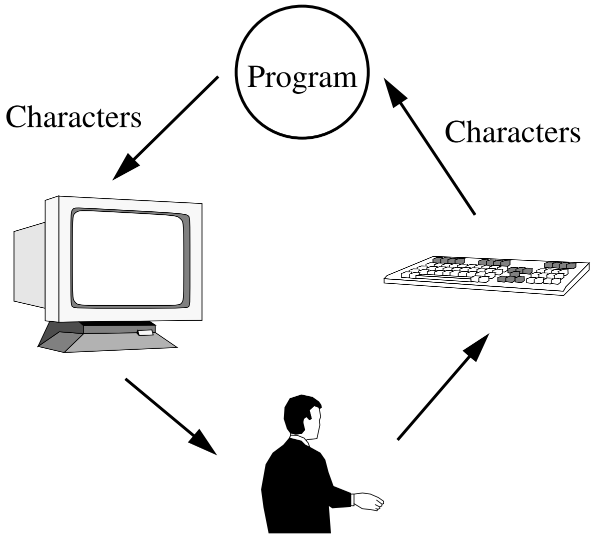


Fig. 2. Landin's Stream I/O model.

from a list of characters (the characters read from the keyboard) to a list of characters (the ones printed on the screen).

As an example, a program to read a sequence of numbers and print their sum could look something like

```
show . sum . map read . words
```

where `words` turns a string into a list of words, and `show` and `read` are overloaded functions that convert to and from string representations of data, respectively.

At first it may seem that thinking of programs as functions from input to output only allows you to write programs with batch behaviour: first read all input, then perform the computation and finally print the result. But thanks to *laziness*, I/O and computations can be interleaved. Input is not demanded until it is needed in the computation of the next output. As an example, a program like

```
map toUpper
```

(where `toUpper` converts lower case letters to upper case letters) reads, processes and outputs one character at a time.

The above described I/O method is Landin's streams model of I/O [8].

2.2.2 I/O Based on Synchronized Streams

I/O in Haskell is also based on streams, but to allow more general I/O operations the elements in the streams are not just characters (Figure 3). The output stream contains requests

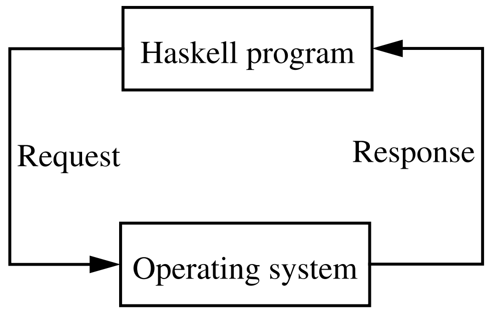


Fig. 3. The I/O model in Haskell.

to the operating system. The input list contains responses to the requests. A program in Haskell is a function of the type

```
type Dialogue = [Response] → [Request]
```

where

```
data Request = ReadFile String
             | WriteFile String String
             | DeleteFile String
             | ReadChan String
             | AppendChan String String
             | ...
data Response = Success
              | Str String
              | Failure IOError
              | ...
data IOError = ...
```

The requests and response streams are synchronized: for each request in the output stream there is a corresponding response in the input stream. A particular request always generates the same kind of response, if the operation succeeds. For example, the response to `WriteFile` *filename newContents* is `Success` and the response to `ReadFile` *filename* is `Str` *contents*, where *contents* is the contents of the file. If an operation fails, the response is `Failure` *ioerror.*

For example, to read some lines of text from a file called "forward" and write them in reverse order to a file called "backward" one could write:

```
main responses =
  ReadFile "forward" :
  (case responses of
     Str contents : responses' →
       let revcontents = (unlines.reverse.lines) contents
       in WriteFile "backward" revcontents :
          [])
```

This program doesn't do any error handling.

2.2.3 I/O in Continuation Passing Style

Dealing with the request and response lists explicitly is a bit clumsy. It is easy to make mistakes, like trying to use a response before the corresponding request has been output, or forgetting to inspect and remove a response from the response list.

Fortunately, it is easy to abstract away from the request and response lists. By using the functions `doRequest` and `done` shown in Figure 4, we make sure that we do not use the wrong response at the wrong time.

```
doRequest :: Request → (Response → Dialogue) → Dialogue
doRequest request continuation responses =
  request :
  case responses of
    response : responses → continuation response responses

done:: Dialogue
done[] = []
```

Fig. 4. Abstracting away from the request and response lists.

The example program can now be written like this:

```
main2 =
  doRequest (ReadFile "forward") $ \ (Str contents) →
  let revcontents = (unlines.reverse.lines) contents
  in doRequest (WriteFile "backward" revcontents) $ \ Success →
     done
```

This programming style is called *continuation passing style* (CPS). The function `doRequest` takes a request to perform and a function that defines how the program should *continue* after that.

Using `doRequest` you can define even more convenient functions for various I/O operations. Here is a function for reading a file with error handling:

```
readFile :: String →
            (IOError→Dialogue) →
            (String→Dialogue) →
            Dialogue
readFile filename failcont continuation =
  doRequest (ReadFile filename) $ \ response →
  case response of
    Str contents → continuation contents
    Failure error → failcont error
```

In Haskell there are predefined functions like `readFile` for most I/O requests.

2.2.4 I/O in Monadic Style

An abstraction that has proved to be useful for many purposes is the *monad* [14]. An I/O system based on monads is proposed for version 1.3 of Haskell [4].

Monadic style I/O operations can be implemented on top of synchronized streams in much the same way as CPS style I/O. A simple I/O monad (without error handling) is shown in Figure 5. In monadic style, the above example would look like this:

```
-- The I/O monad type
type IO a = (a → Dialogue) → Dialogue

doIO :: IO () → Dialogue
doIO io = io (\() → done)

returnIO :: a → IO a
returnIO x = \ cont → cont x

bindIO :: IO a → (a → IO b) → IO b
io1 `bindIO` xio2 =
  \ cont → io1 (\x → xio2 x cont)

-- requestIO: performs one request and returns the response
requestIO :: Request → IO Response
requestIO req =
  \ cont resps →
     req : case resps of
             resp1:resps' → cont resp1 resps'

-- Convenient functions for reading and writing files
readFileIO :: String → IO String
readFileIO filename =
  requestIO (ReadFile filename) `bindIO` \(Str contents) →
  returnIO contents

writeFileIO :: String → String → IO ()
writeFileIO filename contents = ...
```

Fig. 5. Monadic I/O (without error handling) on top of synchronized streams.

```
main = doIO mainIO

mainIO =
  readFileIO "forward" `bindIO` \ contents →
  let revcontents = (unlines.reverse.lines) contents
  in writeFileIO "backward" revcontents)
```

2.3 Concurrency and GUI Programming

We have seen above how you can use streams, continuations, or monads, to specify a sequence of I/O operations that the program should perform.

For many purposes, a single sequence of I/O operations is an adequate description of the I/O behaviour of a program. But there are other cases. For programs that interact with several external entities (teletype terminals, other computers on a network, elements in a graphical user interface, robotic sensors/motors, etc.) there is usually no predetermined order in which I/O operations will occur. The program must be prepared to *react* to input from any of the external entities. In this situation it can be more attractive to organize the program as a set of concurrent processes. You define one "handler" process per external entity, to deal with the low level aspects of the interaction with the entity. You then add processes that communicate with the handlers on a higher level. Figure 6 shows a simple program with a graphical user interface structured in this way. The program just shows a button and a numeric display. When you press the button, the number in the display is incremented. The program contains one process per user interface element. They handle the graphical appearance and behaviour of the respective elements. The program also contains a process that does some "useful" work, i.e. counting the button clicks.

To support the above outlined reactive programming model, you need to introduce some kind of process concept in the language. The next chapter describes one way of doing this.

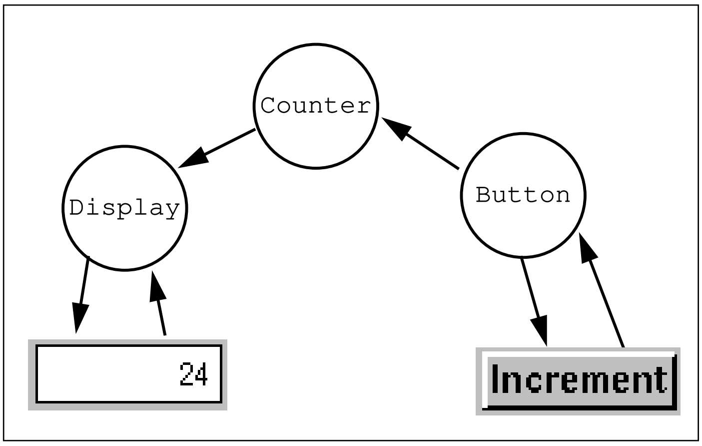


Fig. 6. A program with a graphical user interface, structured according to the reactive programming model.

3 Stream Processors

In this chapter we introduce *stream processors*; a simple but still practical incarnation of the process concept, which can be implemented within a purely functional language. We then define a set of combinators for building networks of stream processors. The stream processors will be first class values, which can be passed around as messages.

First, a *stream* is a potentially infinite sequence of values occurring at different points in time. A stream can be seen as a communication channel, transferring information from one place (a producer) to another (a consumer).

A *stream processor* is a process which consumes some input streams and produces some output streams. A stream processor may have an internal state, i.e., output produced at a certain point in time can depend on all input consumed before that point in time.

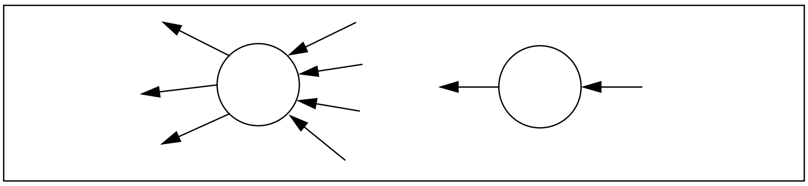


Fig. 7. A general stream processor and a stream processor with a single input stream and a single output stream.

Although stream processors may in general have many input and output streams, in the following we will only consider stream processors with a *single input stream* and a *single output stream* (see Figure 7). This allows us to develop a small set of simple combinators with which it is possible to build complex networks of stream processors. The restriction may seem severe, but the chosen set of combinators allows streams to be merged and split, so a stream processor with many input/output stream can be represented as one with a single input and output stream.

In the following sections, we will present operations used to define atomic stream processors together with a set of combinators for building networks of stream processors. We define three basic compositions: *serial composition*, *parallel composition* and *loops* (circular connections). These are sufficient to describe any network of stream processors.[3] We will also briefly cover an operational semantics of stream processors, the implementation of stream processors in a lazy functional language and some pragmatical aspects.

3.1 The Stream Processor Type

How should stream processors be represented in a lazy functional language? A first attempt is to represent streams as lists,

```
type Stream a = [a]
```

and stream processors as list functions,

```
type SP input output = [input] → [output] -- First attempt
```

With this definition, the type `Dialogue` in Haskell would be equal to `SP Response Request`.

For various reasons, this is not how stream processors are represented in the Fudget library. (We will come back this in Section 3.6.) The Fudget library provides an abstract type for stream processors,

```
data SP input output
```

where `input` and `output` are the types of the elements in the input and output streams, respectively (Figure 8).

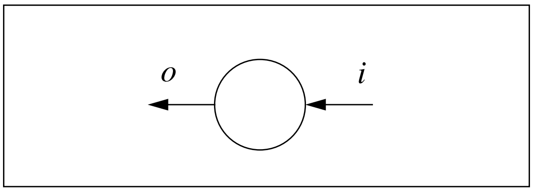


Fig. 8. A stream processor of type `SP` *i o*.

The library also provides the function

```
runSP :: SP i o → [i] → [o]
```

which, when applied to a stream processor from responses to requests, gives us a `Dialogue`.

3.2 Atomic Stream Processors in Continuation Style

The behaviour of an atomic stream processor is described by a sequential program. There are three basic actions a stream processor can take:

- it can put a value in its output stream,
- it can get a value from its input stream,

3. We leave the proof as an exercise for the interested reader.

- it can terminate.

The Fudget library provides the following continuation style operations for these actions:

```
putSP :: output → SP input output → SP input output

getSP :: (input → SP input output) → SP input output

nullSP :: SP input output
```

As an example of how to use these in recursive definitions of stream processors, consider the identity stream processor[4]

```
-- The identity stream processor
idSP :: SP a a
idSP = getSP $ \ x → putSP x idSP
```

and the following stream processor equivalents of the well known list functions:

```
mapSP :: (a → b) → SP a b
mapSP f = getSP $ \ x → putSP (f x) $ mapSP f

filterSP :: (a → Bool) → SP a a
filterSP p = getSP $ \ x → if p x
                             then putSP x $ filterSP p
                             else filterSP p
```

3.3 Stream Processors with Encapsulated State

A stream processor can maintain an internal state. In practice, this can be accomplished by using an accumulating argument in a recursively defined stream processor. As a concrete example, consider `sumSP`, a stream processor that computes the accumulated sum of its input stream:

```
sumSP :: Int → SP Int Int
sumSP acc = getSP $ \ n → putSP (acc+n) $ sumSP (acc+n)
```

In this case, the internal state happens to be a value of the type `Int`, which also happens to be the type of the input and output streams. In general, the type of the state need not be visible in the type of the stream processor.

The Fudget library provides two functions for construction of stream processors with internal state:

```
mapAccumlSP       :: (s → i → (s, o)) → s → SP i o

concatMapAccumlSP :: (s → i → (s, [o])) → s → SP i o
```

Using `mapAccumlSP` we can define `sumSP` without recursion like this:

```
sumSP :: Int → SP Int Int
sumSP = mapAccumlSP (\ acc n → (acc+n,acc+n))
```

Representing state information as one or more accumulating arguments is useful when the behaviour of the stream processor is uniform with respect to the state. If a stream processor reacts differently to input depending on its current state, it can be more convenient to use a set of mutually recursive stream processors that define a finite state automaton. As a simple example, consider a stream processor that outputs every other element in its input stream:

```
passOnSP = getSP $ \ x → putSP x $ skipSP
skipSP = getSP $ \ x → passOnSP
```

4. The infix operator `$` is just function application. More information about notation is in the introduction.

It has two states: the "pass on" state where the next input is passed on to the output, and the "skip" state where the next input is skipped.

3.4 Plumbing: Composing Stream Processors

3.4.1 Serial Composition

The simplest combinator is the one for serial composition,

```
serCompSP :: SP b c → SP a b → SP a c
```

It connects the output stream of one stream processor to the input of another, as illustrated in Figure 9. Streams flow from right to left, just like values in function compositions, $f_1 . f_2$.

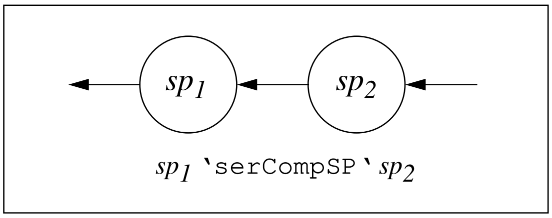


Fig. 9. Serial composition of stream processors.

3.4.2 Parallel Compositions

The combinator for parallel composition in Figure 10 is really the key combinator for

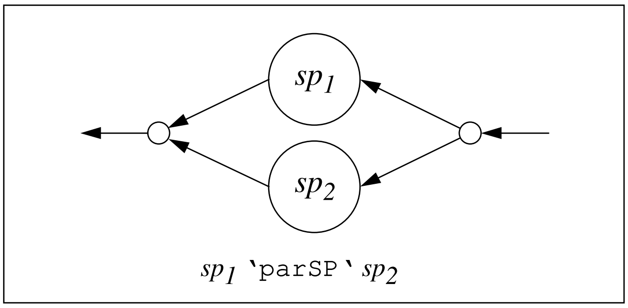


Fig. 10. Parallel composition of stream processors.

stream processors. It allows us to write reactive programs composed by more or less independent, concurrent processes. The idea with parallel composition is that two stream processors should be able to run in parallel, independently of one another. The output streams should be merged in chronological order. We won't be able to achieve exactly this in a functional language, but for stream processors whose behaviour is dominated by I/O operations rather than internal computations we will get close enough for practical purposes.

There is however more than one possible definition of parallel composition. How should values in the input stream be distributed to the two stream processors? How should the output streams be merged? We define two versions:

- Let sp_1 `` `parSP` `` sp_2 denote parallel composition where input values are propagated to both sp_1 and sp_2 and output is merged in chronological order. We will call this version *untagged* or *broadcasting* parallel composition.

- Let sp_1 `` `compSP` `` sp_2 denote parallel composition where the values of the input and output streams are elements of a disjoint union. Values in the input stream tagged `Left` or `Right` are untagged and sent to either sp_1 or sp_2, respectively. Likewise, the tag of a value in the output stream indicates from which component it came. We will call this version *tagged* parallel composition.

The types of the two combinators are:

```
parSP :: SP i o → SP i o → SP i o
compSP :: SP i1 o1 → SP i2 o2 → SP (i1+i2) (o1+o2)
```

where we use `a+b` as an abbreviation for `Either a b`, defined as usual in Haskell:

```
data Either a b = Left a | Right b
```

Note that only one of these need to be considered as primitive. The other one can be defined in terms of the primitive one with the help of serial composition and some simple stream processors like `mapSP` and `filterSP`.

Exercise 1. Define `parSP` in terms of `compSP`, and vice versa!

3.4.3 Circular Connections

Serial composition creates a unidirectional communication channel between two stream processors. Parallel composition splits and merges streams but does not allow the composed stream processors to exchange information. So, with these two operators we can not obtain bidirectional communication between stream processors. Therefore, we introduce combinators that construct loops.

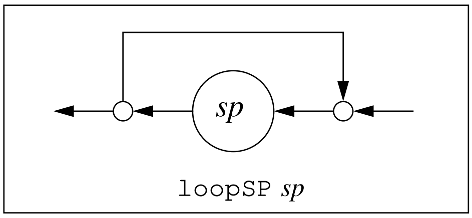


Fig. 11. A simple loop constructor.

The simplest possible loop combinator just connects the output of a stream processor to its input, as illustrated in Figure 11. As with parallel composition, we define two versions of the loop combinator:

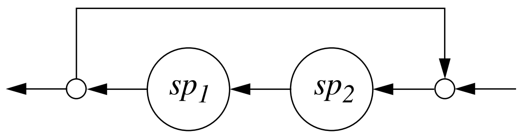
- `loopSP` *sp* – output from *sp* is both looped and propagated to the output.
- `loopLeftSP` *sp* – output from *sp* is required to be in a disjoint union. Values tagged `Left` are looped and values tagged `Right` are output. At the input, values from the loop are tagged `Left` and values from the outside will be tagged `Right`.

The types of these combinators are:

```
loopSP :: SP a a → SP a a
loopLeftSP :: SP (loop+input) (loop+output) → SP input output
```

Each of the two loop combinators can be defined in terms of the other, so only one of them need to be considered primitive.

Using one of the loop combinators, one can now obtain bidirectional communication between two stream processors as shown in Figure 12.




```
loopSP (sp1 `serCompSP` sp2)
```

Fig. 12. Using a loop to obtain bidirectional communication.

As another example, using loops and parallel composition we can create fully connected networks of stream processors. With an expression like

```
loopSP (sp1 `parSP` sp2 `parSP` ... `parSP` spn)
```

we get a broadcasting network. By replacing `` `parSP` `` with `` `compSP` `` and some tagging/untagging, we get a network with point-to-point communication.

3.5 An Operational Semantics for Stream Processors

Here we give an operational semantics for stream processors in the form of a set of rules for rewriting arbitrary stream processor expressions to canonical form. A stream processor is in canonical form if it is built using only the atomic stream processor constructors `nullSP`, `putSP`, and `getSP`.

- Serial composition:

```
nullSP `serCompSP` sp ⇒ nullSP
(e `putSP` sp1)  `serCompSP` sp2 ⇒ e `putSP` (sp1 `serCompSP` sp2)
getSP xsp1  `serCompSP` nullSP ⇒ nullSP
getSP xsp1  `serCompSP` (e `putSP` sp2) ⇒ xsp1 e `serCompSP` sp2
getSP xsp1  `serCompSP` getSP ysp2 ⇒
   getSP (\y → getSP xsp1 `serCompSP` ysp2 y)
```

- Broadcasting parallel composition:

  `` `parSP` `` is meant to be commutative, so we give only the rules for one of two symmetric cases:

```
nullSP `parSP` sp ⇒ sp
(e `putSP` sp1) `parSP` sp2 ⇒ e `putSP` (sp1 `parSP` sp2)
(getSP xsp1) `parSP` (getSP xsp2) ⇒ getSP (\x → xsp1 x `parSP` xsp2 x)
```

- Tagged parallel composition

```
nullSP `compSP` nullSP ⇒ nullSP
(e `putSP` sp1) `compSP` sp2 ⇒ Left e `putSP` (sp1 `compSP` sp2)
sp1 `compSP` (e `putSP` sp2) ⇒ Right e `putSP` (sp1 `compSP` sp2)

getSP xsp1 `compSP` getSP ysp2 ⇒

getSP $ \z → case z of
                Left  x → xsp1 x `compSP` getSP ysp2
                Right y → getSP xsp1 `compSP` ysp2 y
```

The rules for serial compositions are deterministic, but the rules for parallel compositions are not. For example, the expression

```
(a `putSP` nullSP) `parSP` (b `putSP` nullSP)
```

can be reduced to two canonical forms:

```
a `putSP` b `putSP` nullSP
b `putSP` a `putSP` nullSP.
```

This leaves room for both sequential and parallel implementations.

3.6 Implementation of Stream Processors

Using list functions to represent stream processors, `SP i o = [i] -> [o]`, in a sequential language causes some problems. Parallel composition can not be expressed. A reasonable definition would have to look something like this:

```
(sp1 `parSP` sp2) xs = merge (sp1 xs) (sp2 xs)
                       where merge ys zs = ???
```

But what should we replace `???` with so that the first output from the composition is the first output to become available from one of the components? For example, if `sp1` $\bot = \bot$ but `sp2` $\bot$ = 1:$\bot$, then (`sp1 `parSP` sp2`) $\bot$ should be 1:$\bot$. But so should (`sp2 `parSP` sp1`) $\bot$, so `???` must be an expression that chooses the one of `ys` and `zs` which happens to be non-bottom. This can clearly not be done in an ordinary purely functional language. We would need a bottom-avoiding operator, like `amb`, McCarthy's ambivalent operator [9].

So, instead of using lists, we use a data type with constructors corresponding to the actions a stream processor can take (as described in Section 3.2):

```
data SP i o
  = NullSP
  | PutSP o (SP i o)
  | GetSP (i -> SP i o)
```

We call this the *continuation-based representation of stream processors*. It differs from the *list-based representation* in that it makes the consumption of the input stream observable, i.e., a stream processor must evaluate to `GetSP` *sp* each time it wants to read a value from the input stream. It thus comes closer to the operational semantics. It also allows you to make a useful implementation of parallel composition.

With the continuation-based representation, serial composition can be implemented like in Figure 13.

An implementation of broadcasting parallel composition is shown in Figure 14. The implementation of tagged parallel composition is analogous. Note that we arbitrarily choose to inspect the left argument sp_1 first. This means that even if sp_2 could compute and output a value much faster than sp_1, it will not get the chance to do so. But at least, we get the property that if a composition can produce more output without consuming more input, it will do so.

Exercise 2. Implement `runSP :: SP a b -> [a] -> [b]`.

Exercise 3. Implement a combinator `startupSP :: i -> SP i o -> SP i o` that prepends an element to the input stream of a stream processor. Make the implementation independent of the stream processor representation.

Exercise 4. Implement `loopSP` and `loopLeftSP`.

```
sp1 `serCompSP` sp2 =
  case sp1 of
    PutSP y sp1' → PutSP y (sp1' `serCompSP` sp2)
    GetSP xsp1 → xsp1 `serCompSP1` sp2
    NullSP → NullSP

xsp1 `serCompSP1` sp2 =
  case sp2 of
    PutSP y sp2' → xsp1 y `serCompSP` sp2'
    GetSP xsp2 → GetSP (\ x → xsp1 `serCompSP1` xsp2 x)
    NullSP → NullSP
```

Fig. 13. Implementation of serial composition with the continuation-based representation.

```
sp1 `parSP` sp2 =
  case sp1 of
    PutSP y sp1' → PutSP y (sp1' `parSP` sp2)
    GetSP xsp1 → xsp1 `parSP1` sp2
    NullSP → sp2

xsp1 `parSP1` sp2 =
  case sp2 of
    PutSP y sp2' → PutSP y (xsp1 `parSP1` sp2')
    GetSP xsp2 → GetSP (\x→ xsp1 x `parSP` xsp2 x)
    NullSP → GetSP xsp1
```

Fig. 14. Implementation of parallel composition with continuation-based representation.

3.7 More plumbing

3.7.1 Stream processors and Software Re-use

For serious applications programming, it is useful to have libraries of re-usable software components. But in many cases when you find a useful component in a library, you still need to modify it slightly to be able to use it.

A variation of the loop combinators that has turned out to be very useful when re-using stream processors is `loopThroughRightSP`, illustrated in Figure 15. The major difference from `loopSP` and `loopLeftSP` is that the loop does not go straight back from the output to the input of a single stream processor. Instead it goes *through* another stream processor.

A typical situation where `loopThroughRightSP` is useful is when you have a stream processor, sp_{old}, that does almost what you want, but you need it to handle some new kind of messages. You can then define a new stream processor sp_{new} which can pass on old messages directly to sp_{old} and handle the new messages in the appropriate way, on its own or by translating them to messages that sp_{old} understands. (See also section 3.1.1 in [11].)

In the composition `loopThroughRightSP` sp_1 sp_2 all communication with the outside world is handled by sp_1. sp_2 is connected only to sp_1 and is in this sense encapsulated inside sp_1.

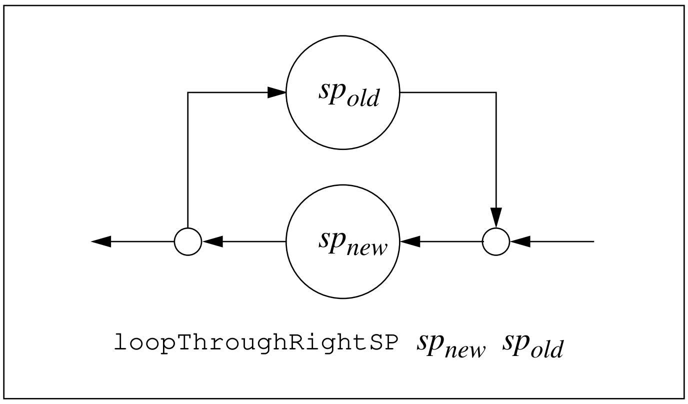


Fig. 15. Encapsulation.

The type of `loopThroughRightSP` is:

```
loopThroughRightSP :: SP (o2+i1) (i2+o1) → SP i2 o2 → SP i1 o1
```

Exercise 5. Implement `loopThroughRightSP` using `loopLeftSP` together with parallel and serial compositions as appropriate.

3.7.2 Handling Multiple Input and Output Streams

Although stream processors have only one input stream, it is easy to construct programs where one stream processor receives input from two or more other stream processors. (The case with several outputs is analogous.) For example, the expression

sp_1 `` `serCompSP` `` (sp_2 `` `compSP` `` sp_3)

allows sp_1 to receive input from both sp_2 and sp_3. For most practical purposes, sp_1 can be regarded as having two input streams, as illustrated in Figure 16. When you use `getSP` in

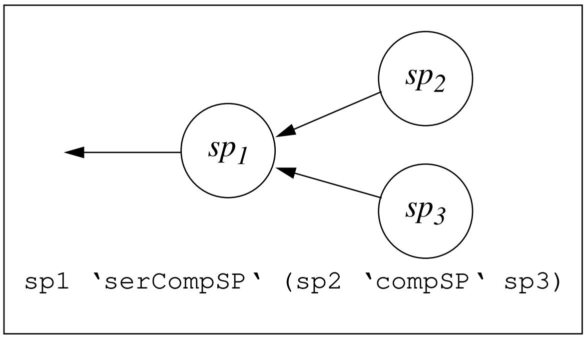


Fig. 16. Handling multiple input streams.

sp_1 to read from the input streams, messages from sp_2 and sp_3 will appear tagged with `Left` and `Right`, respectively. You can not directly read selectively from one of the two input streams, but the Fudget library provides the combinator

```
waitForSP :: (i → Maybe i') → (i' → SP i o) → SP i o
```

which you can use to wait for a selected input. Other input is queued and can be consumed after the selected input has been received. Using `waitForSP` you can define combinators to read from one of two input streams:

```
getLeftSP :: (i1 → SP (i1+i2) o) → SP (i1+i2) o
getLeftSP = waitForSP stripLeft

getRightSP :: (i2 → SP (i1+i2) o) → SP (i1+i2) o
getRightSP = waitForSP stripRight
```

where

```
stripLeft :: a+b → Maybe a
stripLeft (Left x) = Just x
stripLeft (Right _) = Nothing

stripRight :: a+b → Maybe b
stripRight (Left _) = Nothing
stripRight (Right y) = Just y
```

(All of these are provided by the Fudget library.)

Exercises

6. Implement `waitForSP` described above.
7. Implement serial composition using a tagged parallel composition and a loop.
8. Define a *minimal* set of primitive stream processor combinators. Define the remaining combinators in terms of the minimal set and auxiliary atomic stream processors.

4 Fudgets

We have seen how stream processors can be used to structure a program as a set of concurrent processes.

As outlined in Section 2.3, reactive programs that communicate with several external entities typically contain a handler process for each external entity. The handler processes could be implemented as stream processors, but since they need to communicate both with other stream processors and with the outside world we need a special arrangement to give a stream processor convenient access to the I/O system. The special arrangement is called a *fudget*, and was first presented in [1].

4.1 The Fudget Type

A fudget is a stream processor which has *low level streams* for communication with the input/output system and *high level streams* for communication with other fudgets. Fudgets can be composed with a set of combinators like the ones for plain stream processors presented above. A fudget combinator treats the high level streams like the corresponding stream processor combinator, while the low level streams remain connected directly to the I/O system.

The type of a fudget is

`F` *hi ho*

where *hi* and *ho* are the types of the high level input and output streams, respectively. In order to make types more readable, the types of the low level streams are not parameters of the fudget type. Instead, they are fixed to the request and response types used by the I/O system.

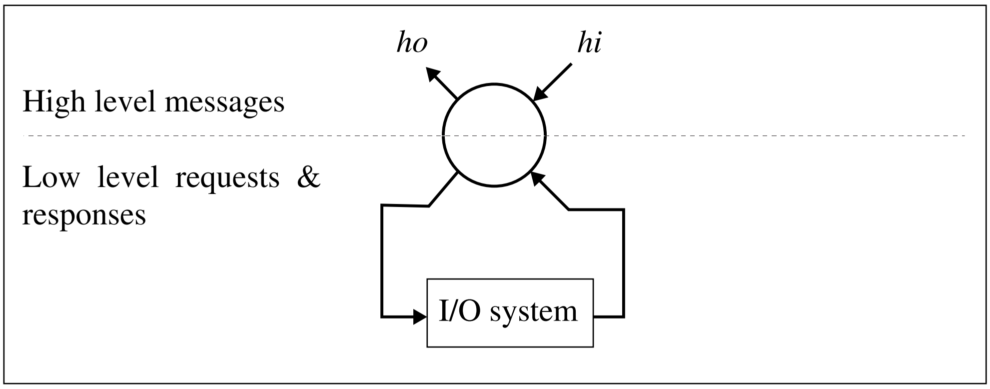


Fig. 17. A fudget of type `F` *hi ho*

Although fudgets have two input and two output streams, they can be represented as stream processors with one input and one output stream. We will return to this in Section 4.4.

On the top level of a fudget program, you use the function

```
fudlogue :: F a b → Dialogue
```

to plug a fudget into the I/O system. `fudlogue` ignores the high level streams of the fudget, so they can be of any type.

4.2 Fudget Plumbing

As mentioned above, there is a set of fudget combinators directly corresponding to the stream processor combinators described in Section 3. Their names are obtained by replacing the `SP` suffix with an `F`. There are also more convenient names for infix use:

```
serCompF, >==< :: F b c → F a b → F a c
compF, >+< :: F i1 o1 → F i2 o2 → F (i1+i2) (o1+o2)
parF, >*< :: F i o → F i o → F i o
listF :: (Eq t) => [(t, F i o)] → F (t, i) (t, o)
loopF :: F a a → F a a
loopLeftF :: F (loop+input) (loop+output) → F input output
loopThroughRightF :: F (o2+i1) (i2+o1) → F i2 o2 → F i1 o1
```

There is one combinator we did not cover in the stream processor section, `listF`, but from the type it should be clear that this is a variation of tagged parallel composition. The argument is a list of tagged fudgets. The elements in the input and output streams are paired with a tag that says which fudget it is to or from.

Figure 18 illustrates serial and parallel composition of fudgets. The low level streams are

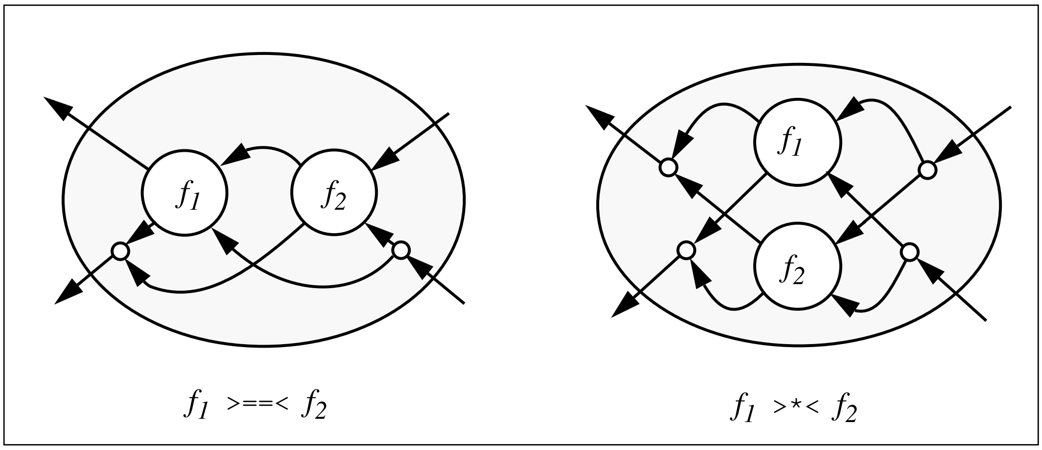


Fig. 18. Serial and parallel composition of fudgets.

treated in the same way in both combinators, i.e., as in parallel stream processor composition, while the high level streams are treated as in the corresponding stream processor combinator. (The tagging of the low level streams is described in Section 4.4.)

4.3 Atomic Fudgets

Fudget programs are built by combining atomic fudgets into more complex ones, using the combinators described above.

In addition to combining fudgets from the library, the application programmer will also need a way to attach his own application specific code. This is done by plugging in *abstract fudgets*.

4.3.1 Abstract Fudgets

An abstract fudget is a fudget that does not use its low level streams. It is simply a stream processor connected to the high level streams of the fudget. The combinator

```
absF :: SP i o → F i o
```

allows you to turn an arbitrary stream processor into an abstract fudget.

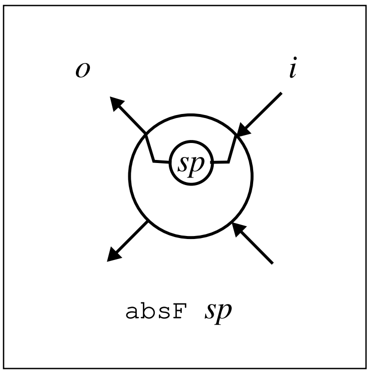


Fig. 19. Abstract Fudgets

Abstract fudgets are often used in serial compositions with other fudgets, i.e., compositions of the form

`absF` *sp* `>==<` *fud*

fud `>==<` `absF` *sp*

are very common. The library provides two operators for these special cases:

```
(>^^=<) :: SP b c → F a b → F a c
sp >^^=< fud = absF sp >==< fud

(>=^^<) :: F b c → SP a b → F a c
fud >=^^< sp = fud >==< absF sp
```

Further, compositions of the form,

`mapSP` *f* `>^^=<` *fud*

fud `>=^^<` `mapSP` *f*

are also common, so the library provides:

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

The implementations of these operators can be very simple, but by making use of the representation of fudgets and stream processors they can be made more efficient. This has been done in the implementation of the Fudget library.

The next section will show some example uses of these combinators.

4.3.2 Fudgets for I/O from the Library

The Fudget system is implemented on top of the usual stream based I/O system in Haskell, but there are extensions that make fudgets more interesting to use:

1. An interface to X Windows. There is a set of extra requests and corresponding responses that allow you to make calls to the Xlib library. The GUI toolkit described in Section 5 uses this extension.
2. An interface to Unix sockets. This allows fudget programs to communicate with other programs, possibly running on different computers. This extension is used for network communication and client/server programming, as shown in Section 6.
3. A mechanism for asynchronous input and timing, i.e., a way for a program to ask for the next available input from any of a set of sources of input. This allows one fudget to wait for input without blocking other fudgets from doing their work. It also allows fudgets to do something even if no input arrives within a certain time.

The library contains fudgets that provide convenient abstraction from the details of the I/O extensions. These fudgets are described in the sections mentioned above. Below, we just take a quick look at some simple fudgets performing I/O.

Standard I/O Fudgets

To read the standard input (usually the keyboard) and write to the standard output or standard error stream (the screen), you can use the fudgets:

```
stdinF  :: F a String
stdoutF :: F String a
stderrF :: F String a
```

The output from `stdinF` is the characters received from the program's standard input channel. For efficiency reasons, you do not get one character at a time, but larger chunks of characters. If you want the input as a stream of lines, you can use

```
inputLinesSP :: SP String String
```

As a simple example here is a fudget that copies text from the keyboard to the screen with all letters converted to upper case:

```
stdoutF >==< (map toUpper >^=< stdinF)
```

It applies `toUpper` to all characters in the strings output by `stdinF` and then feeds the result directly to `stdoutF`.

Here is a fudget that reverses lines:

```
(stdoutF >=^< ((++"\n").reverse)) >==< (inputLinesSP >^^=< stdinF)
```

The precedences and associativities of the combinators are such that we can write these fudgets as:

```
stdoutF >==< map toUpper >^=< stdinF
stdoutF >=^< (++"\n").reverse >==< inputLinesSP >^^=< stdinF
```

The Timer Fudget

The timer fudget generates output after a certain delay and/or at regular time intervals. Its type is

```
data Tick = Tick

timerF :: F (Maybe (Int, Int)) Tick
```

The timer is initially idle. When it receives `Just (interval,delay)` on its input, it starts ticking. The first tick will be output after `delay` milliseconds and then ticks will appear regularly at `interval` milliseconds intervals, unless `interval` is 0, in which case only one tick will be output. Sending `Nothing` to the timer makes it return to the idle state.

As a simple example, here is a fudget that once a second outputs the number of seconds that have elapsed since it was activated:

```
countSP >^^=< timerF >=^^< putSP (Just (1000,1000)) nullSP

  where countSP = mapAccumlSP inc 0
        inc n Tick = (n+1,n+1)
```

4.4 Implementation of Fudgets

4.4.1 Representing Fudgets as Stream Processors

Although fudgets have more than one input and one output stream, they can be represented as stream processors:

```
type F hi ho = SP (Message TResponse hi) (Message TRequest ho)

data Message low high = Low low | High high
```

We use the type `Message` instead of the standard disjoint union `Either` to make programs more readable.

The low level streams carry I/O requests and responses, but when a fudget outputs a request we must be able to send the corresponding response back to the *same* fudget. For this reason, the messages in the low level streams are tagged with a path indicating which fudget in the hierarchy a message is from or to.

```
type TResponse = (Path,Response)
type TRequest  = (Path,Request)

type Path = [Turn]
data Turn = L | R     -- left or right
```

The messages output from atomic fudgets contain an empty path, `[]`. The binary fudget combinators prepend an `L` or and `R` onto the path in output messages to indicate whether the message came from the left or the right subfudget. Non-binary combinators can still use a binary encoding of subfudget positions. On the input side, the path is inspected to find out to which subfudget the message should be propagated.

Figure 20 shows an implementation of tagged parallel composition of fudgets. We have re-used tagged parallel composition of stream processors by adding the appropriate tag adjusting pre- and postprocessors. Other fudget combinators can be implemented using similar techniques.

```
compF, (>+<) :: F i1 o1 → F i2 o2 → F (i1+i2) (o1+o2)

compF f1 f2 = f1 >+< f2

f1 >+< f2 = mapSP post `serCompSP` (f1 `compSP` f2) `serCompSP` mapSP pre

  where

    post msg =
      case msg of
        Left (High ho1) → High (Left ho1)
        Right (High ho2) → High (Right ho2)
        Left (Low (path,req)) → Low (L:path,req)
        Right (Low (path,req)) → Low (R:path,req)

    pre msg =
      case msg of
        High (Left hi1) → Left (High hi1)
        High (Right hi2) → Right (High hi2)
        Low (L:path,resp) → Left (Low (path,resp))
        Low (R:path,resp) → Right (Low (path,resp))
```

Fig. 20. Tagged parallel composition of fudgets.

When a request reaches the top level of a fudget program, the path should be detached before the request is output to the I/O system and then attached to the response before it is sent back into the fudget hierarchy. This is taken care of in `fudlogue`. A simple version of `fudlogue` is shown in Figure 21. However, to handle asynchronous input you need more than this (see Section 4.4.3).

```
fudlogue :: F a b → Dialogue
fudlogue mainF = runSP (loopThroughRightSP routeSP (lowSP mainF))

routeSP =
    getLeftSP $ \ (path,request) →
    putSP (Right request) $
    getRightSP $ \ response →
    putSP (Left (path,response)) $
    routeSP

lowSP :: SP (Message li hi) (Message lo ho) → SP li lo
lowSP fud = filterLowSP `serCompSP` fud `serCompSP` mapSP Low

filterLowSP = mapFilterSP stripLow

stripLow (Low  low) = Just low
stripLow (High _  ) = Nothing
```

Fig. 21. A simple version of fudlogue. It does not handle asynchronous input.

4.4.2 Writing synchronous atomic fudgets

With the above fudget representation, an atomic fudget which repeatedly accepts a Haskell I/O request, perform it and outputs the response can be implemented as show in Figure 22.

```
requestF :: F Request Response
requestF = getHighSP $ \ req →
           putSP (Low ([],req)) $
           getLowSP (_,response) $ \ resp →
           putSP (High response) $
           requestF
```

Fig. 22. An atomic fudget for synchronous I/O.

The combinators `getHighSP` and `getLowSP` waits for high and low level messages, respectively. They are defined in terms of `waitForSP` (Section 3.7.2).

Some requests should be avoided, because when we evaluate their responses, the program will block. For example, we should not use `ReadChan stdin`, because its response is a lazy list representing the character streams from the standard input.

Files are usually OK to read, so the fudget `readFileF` could be useful:

```
readFileF :: F String (IOError + String)
readFileF = post >^=< requestF >=^< ReadFile
  where post (Str s) = Right s
        post (Failure f) = Left f
```

4.4.3 Handling Asynchronous Input

The version of `fudlogue` in Figure 21 will suffice for programs where the individual fudgets do not block in their I/O requests. If we want to react on input from many sources (e.g. sockets, standard input, the window system, timeout events), this implementation will not be enough. Instead, the library version of `fudlogue` maintains a table which maps file descriptors and window identifiers to paths. It then issues a request that waits for input on any of these descriptors, or a timeout (using the UNIX `select` system call). When input becomes available on a descriptor, `fudlogue` finds the path to the responsible fudget via the table, and sends the input to it.

If the stream model for Haskell I/O is used, the request and response data types need to be extended in order to implement this, something which has been done in the Chalmers Haskell B Compiler.[5] If monadic I/O is used (or rather the C monad, as in Glasgow Haskell [12]), there is no need to change the run-time system.

4.4.4 Fudgets in other I/O models

If we implement fudgets on top of monadic I/O, we might want to perform any monadic I/O operation in a fudget, without the old-fashioned coding in request and response values. What we need then is a function `ioF`:

```
ioF:: IO a → (a → F b c) → F b c
```

which will take an I/O operation, perform it, and pass the result to the continuation fudget. This can be implemented by modifying the fudget type to be

```
data F hi ho = F (SP hi (IO (F hi ho) + ho))
```

The idea is that if a fudget outputs the value `Left ioOp`, we should perform the I/O operation `ioOp` which will yield the continuation fudget. `fudlogue` would then have the following type and implementation:

```
fudlogue :: F a b → IO ()
fudlogue (F f) = case f of
  NullSP → returnIO ()
  GetSP xf → returnIO ()
  PutSP o f' → case o of
    Left ioOp → ioOp `bindIO` fudlogue
    Right _ → fudlogue (F f')
```

Exercises

9. Implement `ioF` and the fudget combinators for the suggested fudget type suitable for monadic I/O.
10. Implement fudgets on top of Clean's I/O system [13]. One approach is to implement monadic I/O first.

5. Or one could have the Haskell program talk to another process which implements the necessary extensions.

5 Fudgets for Graphical User Interfaces

The Fudget concept and the Fudget library was first conceived and designed as a tool for the construction of Graphical User Interfaces (GUIs) in a lazy functional language. Although the Fudget library now supports other kinds of I/O, the biggest part of the library still relates to GUI programming.

In the Fudget library, each GUI element is represented as a fudget. The library provides fudgets for many common basic building blocks, like buttons, popup menus, text boxes, etc. The fudget combinators introduced in Section 4 allow you to combine building blocks into complete user interfaces.

This section starts with a couple of programming examples. They illustrate the basic principles of how to create complete programs from GUI elements and application specific stream processors. After the examples follows a brief presentation some common GUI fudgets from the library. We then describe combinators for layout and a scheme for parameter passing with default values.

5.1 The "Hello, World!" Example

We begin with a simple program that just displays a message in a window (see Figure 23). This example illustrates what the main program should look like, as well as some other practical details.

The Fudgets library contains a fudget[6] for static messages,

```
labelF :: String → F a b
```

It just shows the argument string. It does not use its high level streams.

In the example program we have put the display in a top-level window created with the fudget,

```
shellF:: String → F a b → F a b
```

which given a window title and a fudget, creates a shell window containing the graphical user interface defined by the argument fudget. The fudgets for GUI elements, like `labelF`, can not be used directly on the top level in a program, but must appear inside a shell window.

This illustrates the typical structure of a fudget program. In the `main` function, which in Haskell should have the type `Dialogue`, we call the function `fudlogue`,

```
fudlogue:: F a b → Dialogue
```

```
module Main where -- The "Hello, World!" program
import Fudgets

main = fudlogue (shellF "Hello" helloF)

helloF = labelF "Hello, World!"
```

Fig. 23. The "Hello World" program.

6. To be precise, `labelF` is a function returning a fudget, but for convenience, we will often say "a fudget" when we mean "a function returning a fudget".

which sets up the communication with the window system, gathers commands sent from all fudgets in the program and sends them to the window system, and distributes events coming from the window system to the appropriate fudgets.

Additional things to note with this program is that you do not need to specify the size and placement of the GUI elements. The fudget system automatically picks a suitable size for the label and the size of the shell is adapted to that.

Useful programs of course contain more than one GUI element. The next example will contain two elements!

5.2 The Counter Example

This program is a simple counter. Its user interface consists of a button and a numeric display (see Figure 1). When you press the button, the number in the display is incremented.

In this program we use two basic building blocks,

```
intDispF :: F Int a
```

which is a fudget that displays the numbers it receives on the input, and

```
buttonF :: String → F Click Click
```

which implements command buttons. It outputs clicks,

```
data Click = Click
```

when you press the button. Feeding clicks to its input has the same effect as clicking on the button with the mouse.

When combining fudgets for GUI elements, there are two considerations:

1. The data flow aspect: how should they communicate, i.e., should one use a serial, parallel, or some other combinator?
2. The visual aspect: how should the GUI elements be placed on the screen?

These are quite separate concerns, and fortunately the fudget system allows us to worry about them one at a time. Thanks to the automatic layout system, you can concentrate on the data flow aspect during the initial development stage. Later on, if you are not happy with

```
module Main(main) where -- A simple counter

import Fudgets

main :: Dialogue
main = fudlogue (shellF "Counter" counterF)

counterF = intDispF >==< absF countSP >==< incButtonF

incButtonF :: F Click Click
incButtonF = buttonF "Increment"

countSP :: SP Click Int
countSP = putSP startstate $
          mapAccumlSP inc startstate
  where inc n Click = (n+1,n+1)

startstate = 0
```

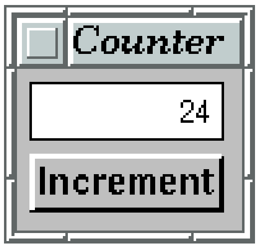


Fig. 24. The Counter Example

the default layout, you can add layout information in one of the ways described in Section 5.5.

The data flow in this program, illustrated as a circuit diagram in Figure 25, is simple and is implemented by one program line in the definition of `counterF`:

```
intDispF >==< absF countSP >==< incButtonF
```

`countSP` contains the application specific code in this program. It starts by outputting the initial state, so that it will be visible in the display when the program starts. It then uses `mapAccumlSP` (Section 3.3) to maintain an internal counter that is incremented for each click received from the button.

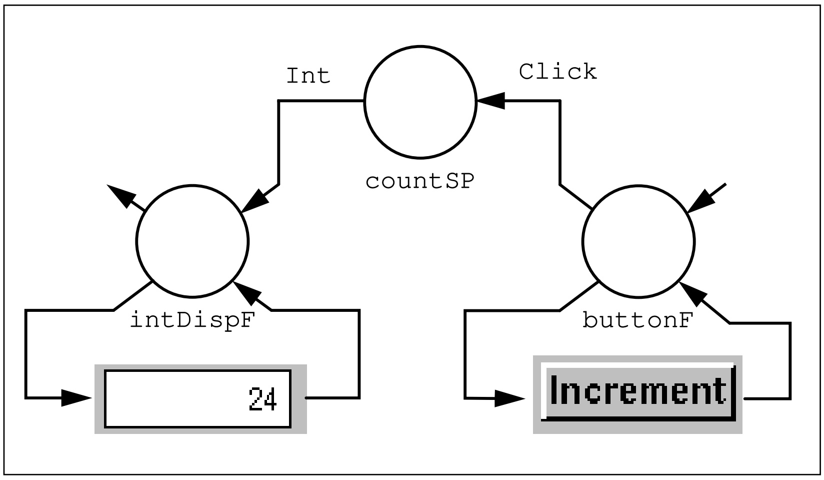


Fig. 25. Circuit diagram for the counter

5.3 Extending the Counter

What if we want an extended counter that can be incremented, decremented and reset?

Starting from the simple counter program above, we obviously have to add two new buttons, but we will also have to change `countSP`. It will now have to deal with three different input messages that have the effect of incrementing, decrementing and resetting the current value of the counter.

There are different ways you can go about this. One way would be to define a data type for the kind of messages we need,

```
data ButtonMsg = Inc | Dec | Reset
```

and then interpret these inside `countSP`. Another way would be to define a general state maintaining stream processor, which receives state modifying functions on the input and delivers the new state on the output after a state modifier has been applied. Here is such a stream processor:

```
stateSP :: state → SP (state→state) state
stateSP state = putSP state $ mapAccumlSP modify state
  where modify state f = (state',state')
          where state' = f state
```

Using `stateSP` instead of `countSP`, all that is left to do to complete the extended counter is to define buttons that output the appropriate state modifying functions,

```
incButtonF   = fButtonF (+1)       "Increment"
decButtonF   = fButtonF (+(-1))    "Decrement"
resetButtonF = fButtonF (const 0) "Reset"

fButtonF f lbl = const f >^=< buttonF lbl
```

and put everything together. The resulting program is shown in Figure 26.

```
module  Main(main) where -- An extended counter

import Fudgets

main :: Dialogue
main = fudlogue (shellF "Counter" counterF)

counterF = intDispF >==<
           absF (stateSP startstate) >==<
           buttonsF

buttonsF :: F a (Int→Int)
buttonsF = incButtonF >*< decButtonF >*< resetButtonF

incButtonF   = fButtonF (+1)       "Increment"
decButtonF   = fButtonF (+(-1))    "Decrement"
resetButtonF = fButtonF (const 0) "Reset"

fButtonF f lbl = const f >^=< buttonF lbl

stateSP :: state → SP (state→state) state
stateSP state = putSP state $
                mapAccumlSP modify state
  where modify state f = (state',state')
          where state' = f state

startstate = 0
```

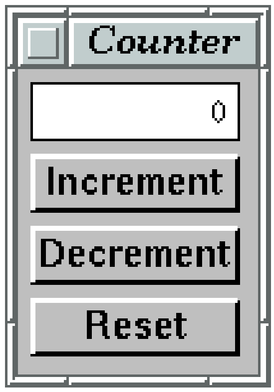


Fig. 26. The Extended Counter Example

Exercises

11. Draw the circuit diagram for the extended counter.
12. Extend the extended counter to a pocket calculator. (Don't worry about the layout of the buttons at this point.)

5.4 More GUI elements

In this section we present some common GUI elements provided by the Fudget Library. For more information, consult the reference manual, which is available via WWW [5].

5.4.1 Buttons

We have already seen `buttonF` in the examples above. It provides *command buttons*, i.e., buttons that you press to trigger some action. The Fudget library also provides toggle buttons and radio groups (Figure 27). Pressing these buttons causes a change that have a lasting visual effect (and probably also some other lasting effect). A toggle button changes between two states (on and off) each time you press it. A radio group allows you to activate one of several mutually exclusive alternatives. The types of these fudgets are

```
toggleButtonF :: String → F Bool Bool
radioGroupF   :: (Eq alt) => [(alt,String)] → alt → F alt alt
```

The input messages can be used to change the setting under program control.

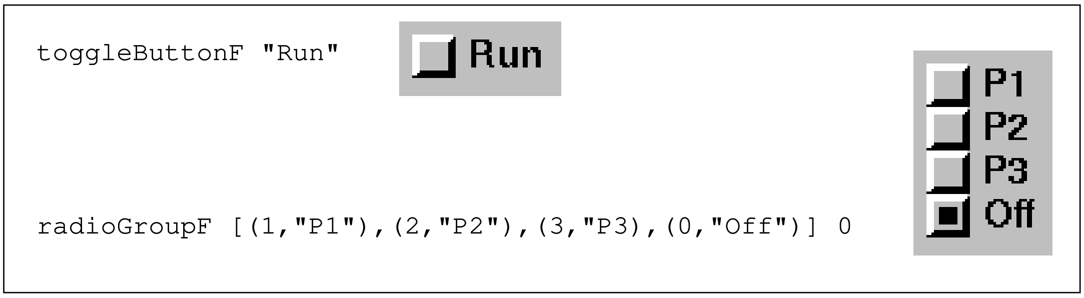


Fig. 27. Toggle buttons and radio groups

5.4.2 Menus and Scrollable Lists

Menus serve much the same purpose as buttons, but they save screen space by appearing only when activated. The fudget `menuF name alts`, where

```
menuF :: String → [(alt,String)] → F alt alt,
```

provides pull-down menus. `name` is the constantly visible name you press to activate the menu and `alts` is the list of menu alternatives.

The fudget

```
popupMenuF :: [(alt,String)] → F i o → F (alt+i) (alt+o)
```

provides pop-up menus, i.e., menus that are activated when a certain mouse button (the third by default) is pressed over the screen area occupied by the argument fudget. The menu fudget and the argument fudget are put in a tagged parallel composition.

As an example, suppose we wanted a compact version of the extended counter in Section 5.3. We could then replace the three buttons with a pop-up menu attached to the display (Figure 28).

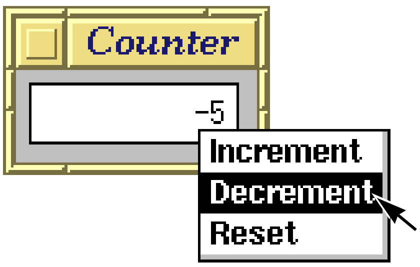


Fig. 28. A Compact Extended Counter

When the number of alternatives is large, or when they change dynamically, you can use a scrollable list instead of a menu. The function

```
pickListF :: (a→String) → F [a] a
```

(shown in Figure 29) takes a show function and returns a fudget that displays lists of alternatives received on the high level input. When an alternative is selected, by clicking on it, it will appear in the output stream.

Fig. 29. `pickListF`

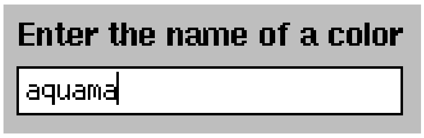


Fig. 30. `stringF`

Exercises

13. Implement the compact extended counter. *Hint*: a handy combinator is `serCompLeftToRightF`,

    ```
    serCompLeftToRightF :: F (i+a) (a+o) → F i o
    ```

    which turns a tagged parallel composition into a serial composition.

5.4.3 Entering values

Choosing an alternative from a list is usually easier than typing something, e.g., the name of a colour, on the keyboard. But when there is no predefined set of alternatives, you can use fudgets that allow the user to enter values from the keyboard. The library provides

```
stringF :: InF String String
intF    :: InF Int Int
```

for entering strings and integers (see Figure 30). For entering other types of values, you can use `stringF` and attach the appropriate printer and parser functions. The type `InF` is defined as

```
type InF a b = F a (InputMsg b)
data InputMsg b = ...
```

The input fudgets have two kinds of output messages: one that is output whenever the current value changes and one that is output when the user indicates, e.g., by pressing the `Return` or `Enter` key, that the value is complete. You can use

```
stripInputMsg :: InputMsg a → a
inputDoneSP :: SP (InputMsg a) a
```

as postprocessors to filter out the messages you are interested in. Use

```
stripInputMsg >^=< stringF
```

if you are interested in all changes made to a string, and

```
inputDoneSP >^^=< stringF
```

if you only want a message when the user has completed a string.

5.4.4 Displaying and editing text

The library provides the fudgets

```
moreF :: F [String] a
moreFileF, moreFileShellF:: F String a
```

which can display longer text.[7] The input to `moreF` is lines of text to be displayed. The other two fudgets display the contents of file names received on the high level input. `moreFileShellF` in addition appears in its own shell window with a title reflecting the name of the file being displayed.

There also is a text editor fudget:

```
editorF :: F EditCmd EditEvt
```

which supports cut/paste editing with the mouse as well as a small subset of the keystrokes used in GNU emacs. It also has an undo/redo mechanism.

5.4.5 Scroll Bars

GUI elements that potentially can become very large, like `pickListF`, `moreF` and `editorF` have scroll bars attached by default. There are also combinators to explicitly add scroll bars:

```
scrollF, vScrollF, hScrollF :: F a b → F a b
```

The `v` and `h` version give only vertical and horizontal scroll bars, respectively. The argument fudget can be any combination of GUI elements.

5.5 Layout

When developing fudget programs, normally we don't have to think about the actual layout of the GUI fudgets if we don't want to. For example, the fudget

```
shellF "Buttons" (buttonF "A Button" >+< buttonF "Another Button")
```

will get some default layout which might look like Figure 30. Sooner or later, we will want

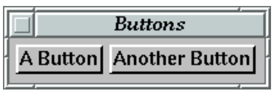


Fig. 30.

to have control over the layout, though. The GUI library lets us do this two different ways:

1. *Fudget Combinator Layout*. This method is based on variants of the fudget combinators `>+<`, `>==<`, and `listF`. It is a quick way of adding layout control to a program. However, the layout possibilities are limited by the structure of the fudget program.
2. *Name Layout*. Here, the layout is specified separately from the fudget structure. GUI fudgets are given names, and these are used to specify layout at one place inside each `shellF`.

Before describing these, we will present the layout combinators that both of them use.

7. The names comes from the fact that they serve the same purpose as the UNIX program `more`.

5.5.1 Boxes, Placers and Spacers

Layout is done hierarchically. Each GUI fudget will reside in a *box*, which will have a certain size and position when the layout is complete. A list of boxes can be put inside a single box by a *placer*, which also defines how the boxes should be placed in relation to each other inside the larger box. The effects of some placers are illustrated in Figure 31. The parameter

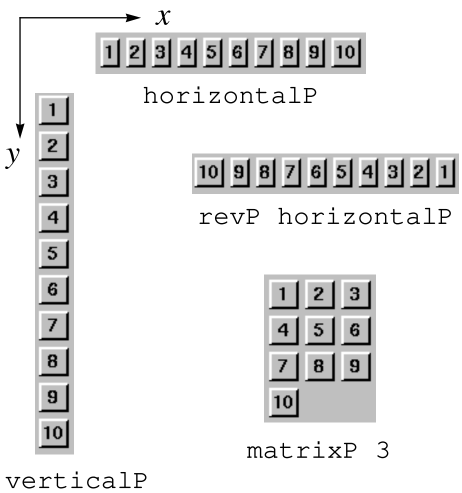


Fig. 31. Different placers.

to `matrixP` specifies the number of columns the matrix should have. The types of the placers are

```
horizontalP :: Placer
verticalP :: Placer
matrixP :: Int → Placer
revP :: Placer → Placer
```

The effect of applying `revP` is as if the list of boxes were reversed. Another higher order placer is `flipP`, which transforms a placer into a mirror symmetric placer, with respect to the line $x = y$:

```
flipP :: Placer → Placer
```

Hence, we can define `verticalP` as

```
verticalP = flipP horizontalP
```

So, placers are used to specify the layout of a group of boxes. In contrast, *spacers* are used to wrap a box around a single box. Spacers could be used to determine how a box should be aligned if it is given too much space, or to add extra space around a box. Examples of

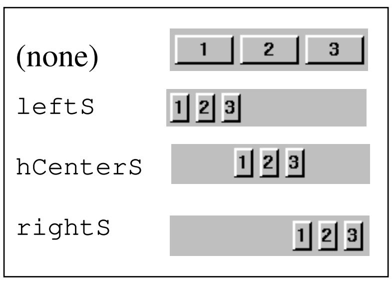


Fig. 32. Spacers for alignment.

spacers that deal with alignment can be seen in Figure 32. On top, the box (placed with horizontalP) has to fill up all the available space. The lower three boxes have been placed inside a box, which consumes the extra space. The spacers used are derived from the spacer hAlignS, whose argument tells the ratio between the space to the left and the right side of the box:

```
hAlignS :: RealFrac a => a → Spacer
leftS = hAlignS 0
hCenterS = hAlignS 0.5
rightS = hAlignS 1
```

There is a corresponding spacer to flipP, namely flipS. It too flips the *x* and *y* coordinates, and let us define some useful vertical spacers:

```
flipS :: Spacer → Spacer
vAlignS a = flipS (hAlignS a)
topS = flipS leftS
vCenterS = flipS hCenterS
bottomS = flipS rightS
```

By compS, we can compose spacers, and define a spacer that centers both horizontally and vertically:

```
compS :: Spacer → Spacer → Spacer
centerS = vCenterS `compS` hCenterS
```

To add extra space to the left and right of a box, we use hMarginS left right, where

```
hMarginS :: Distance → Distance → Spacer
type Distance = Int
```

Distances are given in number of pixels.[8] From hMarginS, we can derive marginS, which adds equally much space on all sides of a box:

```
vMarginS above below = flipS (hMarginS above below)
marginS s = vMarginS s s `compS` hMarginS s s
```

Spacers can be applied to fudgets by means of spacerF:

```
spacerF :: Spacer → F a b → F a b
```

spacerF f will apply the spacer to all boxes inside f.[9] We can also modify a placer by wrapping a spacer around the box that the placer assembles:

```
spacerP :: Spacer → Placer → Placer
```

For example, spacerP leftS horizontalP gives a horizontal placer which will left adjust its boxes.

5.5.2 Combinator Layout

Combinator layout is good when flexible layout is not a major issue in your program. As an example, we could specify that the two buttons in Figure 30 should have vertical layout by saying that the first button should be above the second:

```
shellF "Buttons"
     ((buttonF "A Button",Above) >+#< buttonF "Another Button")
```

8. This is easy to implement, but makes programs somewhat device dependent.
9. It will not apply it recursively to the boxes inside the boxes, however.

Here, we have replaced `>+<` with `>+#<` which takes an extra layout argument:

```
(>+#<) :: (F a b,Orientation) → F c d → F (a+c) (b+d)
data Orientation = Above | Below | RightOf | LeftOf
```

The result can be seen in Figure 33. In a similar way, the first button could be placed below,

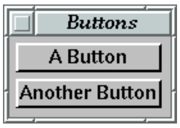


Fig. 33.

to the right of, or to the left of the second button, by using the corresponding constructor of type `Orientation`.

The same trick can be used on serial composition by using `>==#<`:

```
(>==#<) :: (F a b,Orientation) → F c a → F c b
```

If we want to specify the layout for the fudgets inside a `listF`, we use `listLF` instead:

```
listLF :: (Eq a) => Placer → [(a, F b c)] → F (a, b) (a, c)
```

The first argument to `listLF` is a placer, specifying the layout.

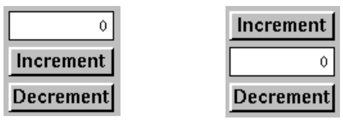
Suppose we have the following fudget, where we have used combinator layout:

```
top = (intDispF >=^^< acc,Above) >==#< buttons

buttons = snd >^=< listLF verticalP (number 1 buttonlist)
buttonlist = [const f >^=< buttonF s | (f,s) <- list]
       where list = [((+1),   "Increment"),
                     ((+(-1)),"Decrement")]

acc = ac 0 where ac n = putSP n $ getSP $ \f → ac (f n)
```

It will have the layout shown in Figure 31a. Now, suppose we want the number display to



a b

Fig. 34.

appear between the buttons, as in Figure 31b. We can not do that with combinator layout without restructuring the program, because the buttons reside in a `listLF`, whose placer will box them together. By using Name Layout, we get around the problem.

5.5.3 Name Layout

To separate layout from fudget structure, we put unique names on each box (usually corresponding to a simple GUI fudget) whose layout we want to control, by using `nameF`:

```
type LName = String
nameF :: LName → F a b → F a b
```

The layout of the boxes which have been named in this way is specified using the type `NameLayout`. Here are the basic functions for constructing `NameLayout` values:

```
leafNL :: LName → NameLayout
placeNL :: Placer → [NameLayout] → NameLayout
spaceNL :: Spacer → NameLayout → NameLayout
```

The desired layout in Figure 31b has the buttons in a row, so we will use `verticalP`. To apply the layout to named boxes, we use `nameLayoutF`:

```
nameLayoutF :: NameLayout → F a b → F a b
```

The names used for the boxes are "bound" by `nameLayoutF`, by corresponding occurrences of `leafNL`. Our example becomes

```
top = nameLayoutF layout $ nameF dispN textF >=^^< acc >==< buttons

buttons = snd >^=< listF (number 1 buttonlist)
buttonlist = [const f >^=< nameF n (buttonF s) | (n,f,s) <- list]
      where list = [(incN,(+1),    "Increment"),
                    (decN,((+(-1)),"Decrement")]

acc = ac 0 where ac n = putSP n $ getSP $ \f → ac (f n)
-- only layout below
layout = listNL verticalP (map leafNL [incN, dispN, decN])
incN = "inc"
decN = "dec"
dispN = "disp"
```

Now, we can muck around with the layout of the two buttons and the display as much as we want, without changing the rest of the program. The actual strings used for names are not important, as long as they are unique within the part of the fudget structure where they are in scope. So instead we could write

```
(incN:decN:dispN:_) = map show [1..]
```

5.5.4 The placer fudget (the middle way)

Actually, there is a third way of doing layout, which is somewhere in between Fudget Combinator Layout and Name Layout. The fudget

```
placerF :: Placer → F a b → F a b
```

will apply the placer to all boxes in the argument fudget. The order of the boxes is left to right, with respect to the combinators `listF`, `dynListF`, `>==<` and `>+<`. Actually, when there is no layout specified in a shell fudget with more than one box in it, an implicit placer fudget is applied to obtain one box, which the shell fudget can handle.

With `placerF`, we can derive the combinators used for Fudget Combinator Layout:

```
listLF placer f = placerF placer (listF f)
place2F (><) (f1,al) f2 = placerF (placer al) (f1 >< f2) where
   placer LeftOf = horizontalP
   placer RightOf = revP horizontalP
   placer Above = verticalP
   placer Below = revP verticalP
(>+#<) = place2F (>+<)
(>==#<) = place2F (>==<)
```

If we take a look at the layout in Figure 31a again, we see that if we write a placer that permutes the first and the second box (cf. `revP`), we could get the desired layout in b. However, such a layout system would be sensitive for changes in the fudget structure (e.g., if we change f >+< g to g >+< f, we have to change the placer. If we use Name Layout, this change does not affect the layout.

Exercises

14. Augment the pocket calculator in exercise 12 with proper layout of the buttons.

5.6 Parameters for Customization

There are many aspects of GUI fudgets that one might want to modify, e.g. the font or the foreground or background colours for `displayF`. The simple GUI fudgets have some hopefully reasonable default values for these aspects, but sooner or later, we will want to change them. A simple way of doing this would be to have a data type with constructors for each parameter that has a default value. In the case of `displayF`, it might be

```
data DisplayFParams = Font FontName |
                      ForegroundColor ColorName |
                      BackgroundColor ColorName
```

Then, one could have the display fudget take a list of display parameters as a first argument:

```
displayF :: [DisplayFParams] → F String a
```

Whenever we are happy with the default values, we just use an empty parameter list, and all is fine.

However, suppose we want to do the same trick with the button fudget. We want to be able to customise font and colours for foreground and background, like the display fudget, and in addition we want to specify a "hotkey" that could be used instead of clicking the button:

```
data ButtonFParams =  Font FontName |
                      ForegroundColor ColorName |
                      BackgroundColor ColorName |
                      HotKey (ModState,Key)
```

Now, we are in trouble if we want to customise a button and a display in the same module, because in a given scope in Haskell, no two constructor names should be equal. Of course, we could rename the constructors when importing them, but this is tedious. We could also have different constructor names to start with (`ButtonFFont`, `ButtonFForegroundColor` etc.), which is just as tedious.

Our current solution[10] is to not use constructors directly, but to use overloaded functions instead. We will define a class for each kind of default parameter. Then, each customizable fudget will have instances for all parameters that it accepts. This entails some more work when defining customizable fudgets, but the fudgets become easier to use, which we feel more than justifies the extra work.

10. The basics of this design are due to John Hughes.

5.6.1 A Mechanism for Default Values

Let us return to the display fudget example, and show how to make it customizable. First, we define classes for the customizable parameters:

```
type Customiser a = a → a

class HasFont a where
   setFont :: FontName → Customiser a

class HasForegroundColor a where
   setForegroundColor :: ColorName → Customiser a

class HasBackgroundColor a where
   setBackgroundColor :: ColorName → Customiser a
```

Then, we define a data type for the parameter list to `displayF`:

```
data DisplayF = Pars [DisplayFParams]
```

and add the instance declarations

```
instance HasFont DisplayF where
   setFont p (Pars ps) = Pars (Font p:ps)

instance HasForegroundColor DisplayF where
   setForegroundColor p (Pars ps) = Pars (ForegroundColor p:ps)

instance HasBackgroundColor DisplayF where
   setBackgroundColor p (Pars ps) = Pars (BackgroundColor p:ps)
```

The type of `displayF` will be

```
displayF :: (Customiser DisplayF) → F String a
```

We put these declarations inside the module defining `displayF`, making `DisplayF` abstract. When we later use `displayF`, the only thing we need to know about `DisplayF` is its instances, which tell us that we can set font and colours. For example:

```
myDisplayF = displayF (setFont "fixed" . setBackgroundColor "green")
```

If we want to have `buttonF` customizable the same way, we define the additional class:

```
class HasKeyEquiv a  where
   setKeyEquiv :: (ModState,Key) → Customiser a
```

The button module defines

```
data ButtonF = Pars [ButtonFParams]
```

and makes it abstract, as well as defining instances for font, colours and hotkeys.[11] We can now customise both the display fudget and the button fudget, if we want:

```
myFudget = displayF setMyFont >+< buttonF (setMyFont.setMyKey) "Quit"
   where setMyFont = setFont "fixed"
         setMyKey = setKeyEquiv ([Meta],"q")
```

11. Note that the instance declarations for font and colours will look exactly the same as for the display parameters!

If we do not want to change any default values, we use `standard`, which doesn't modify anything:

```
standard :: Customiser a
standard p = p

standardDisplayF = displayF standard
```

5.6.2 The Customizable GUI Fudgets

The GUI fudget library is designed so that when you start writing a fudget program, there should be as few distracting parameters as possible. Default values will be chosen for colour, fonts, layout, etc. But a customizable fudget must inevitably have an additional argument, even if it is `standard`. We use short and natural names for the standard versions of GUI fudgets, without customization argument. So we have

```
buttonF :: String → F Click Click
buttonF = buttonF' standard
buttonF' :: Customiser ButtonF → String → F Click Click

displayF :: F String a
displayF = displayF' standard
displayF' :: Customiser DisplayF → F String a
```

and so on.[12] This way, a programmer can start using the toolkit without having to worry about the customization concept. Later, when the need for customization comes, just add an apostrophe and the parameter.

Most parameters can in fact be changed *dynamically*, if needed. Therefore, each customizable fudget comes in a third variant, which is the most expressive:

```
type CF p a b = F (Customiser p + a) b
buttonF'' :: Customiser ButtonF → String → CF ButtonF Click Click
displayF'' :: Customiser DisplayF → CF DisplayF String a
```

etc.

Exercises

15. Use different colours for different kinds of buttons in the pocket calculator from exercise 12 and 14.

6 Client/Server Programming & Typed Sockets

In this section, we will see how fudgets can be suitable for other kind of I/O than graphical user interfaces. We will write client/server applications, where a fudget program acts as a server on one computer. The clients are also fudget programs, and they can be run from other computers if desired.

The server is an example of a fudget program which may not have the need for a graphical user interface. However, the server should be capable of handling many clients simultaneously. One way of organising the server is to have a *client handler* for each connected

12. One could also have the apostrophe on the standard versions, something that sounds attractive since apostrophes usually stand for omitted things (in this case the customizer). But then a programmer must learn which fudgets are customizable (and thus need an apostrophe), even if she isn't interested in customization.

client. Each client handler communicates with its client via a connection (a socket), but it will probably also need to interact with other parts of the server. This is a situation where fudgets come in handy. The server will dynamically create fudgets as client handlers for each new client that connects.

We will also see how the type system of Haskell can be used to associate the address (a host name and a port number) of a server with the type of the messages that the server can send and receive. If the client is also written in Haskell, and imports the same specification of the typed address as the server, we know that the client and the server will agree on the types of the messages, or the compiler will catch a type error.

6.1 Fudgets for Internet Stream Sockets

The type of sockets that we consider here are Internet stream sockets. They provide a reliable, two-way connection, similar to pipes, between any two hosts on the Internet. They are used in Unix tools like telnet, ftp, finger, mail, Usenet and World Wide Web.

6.1.1 Clients

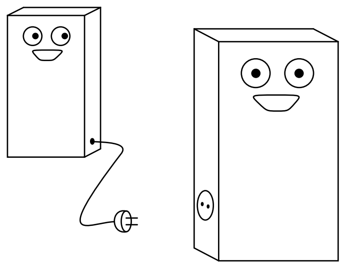


Fig. 35. A client about to connect to a server.

To be able to communicate with a server, a client must know where the server is located. The location is determined by the name of the host (a computer on the network) and a port number. A typical host name is `www.cs.chalmers.se`. The port number distinguishes different servers running on the same host. Standard services have standard port numbers. For example, WWW servers are usually located on port 80.

The Fudget library uses the following types:

```
type Host = String
type Port = Int
```

The simple fudget

```
socketTransceiverF :: Host → Port → F String String
```

allows a client to connect to a server and communicate with it (Figure 35).[13] Chunks of characters appear in the output stream as soon as they are received from the server (c.f. `stdinF` in Section 4.3.2).

The simplest possible client you can write is perhaps a telnet client:

```
telnetF host port = stdoutF >==< socketTransceiverF host port
                      >==< stdinF
```

13. The library also provides combinators that gives more control over error handling and the opening and closing of connections.

This simple program doesn't do the option negotiations required by the standard telnet protocol [RFC854,855], so it doesn't work well when connected to the standard telnet server (on port 23). However, you can use it to talk to many other standard servers, e.g., mail and news servers.

6.1.2 Servers

Whereas clients actively connect to a specific server, servers passively wait for clients to connect. When a client connects, a new communication channel is established, but the server typically continues to accept connections from other clients as well (Figure 36).

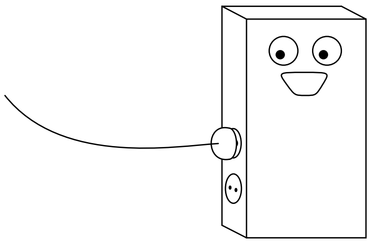


Fig. 36. A communication socket is created.

A simple fudget to create servers is

```
simpleSocketServerF :: Port → F (Int,String) (Int,String)
```

The server allows clients to connect to the argument port on the host where the server is running. A client is assigned a unique number when it connects to the server. The messages to and from `simpleSocketServerF` are strings tagged with a client number. Empty strings in the input and output streams mean that a connection should be closed or has been closed, respectively.

This simple server fudget does not directly support a program structure with one handler fudget per client. A better combinator is shown below.

Exercise 16. Write a chat client and a chat server. The chat client allows a user to exchange messages with other users running the chat client. A message entered by a user is sent to all other users.

6.2 Typed Sockets

Many Internet protocols use messages that are human readable text. When implementing these, the natural type to use for messages is `String`. However, when we write both clients and severs in Haskell, we may want to use an appropriate data type for messages sent between clients and server, as you would do if the client and server were fudgets in the same program. In this section we show how to abstract away from the actual representation of messages on the network.

We introduce two abstract types for typed port numbers and typed server addresses. These types will be parameterised on the type of messages that we can transmit and receive on the sockets. First, we have the typed port numbers:

```
data TPort c s
```

The client program needs to know the typed address of the server:

```
data TServerAddress c s
```

In these types, `c` and `s` stand for the type of messages that the client and server transmit, respectively.

To make a typed port, we apply the function `tPort` on a port number:

```
tPort :: (Text c, Text s) => Port → TPort c s
```

The context `Text` in the signature tells us that not all types can be used as message types. Values will be converted into text strings before transmitted as a message on the socket. This is clearly not very efficient, but it is a simple way to implement a machine independent protocol.

Given a typed port, we can form a typed server address by specifying a computer as a host name:

```
tServerAddress :: TPort c s → Host → TServerAddress c s
```

For example, suppose we want to write a server that will run on the host `animal`, listening on port 8888. The server should accept integer messages, and will send strings to the clients. This can be specified by

```
thePort :: TPort Int String
thePort = tPort 8888
theServerAddr = tServerAddress thePort "animal"
```

A typed server address can be used in the client program to open a socket to the server by means of `tSocketTransceiverF`:

```
tSocketTransceiverF ::
        (Text c, Text s) => TServerAddress c s → F c (Maybe s)
```

Again, the `Text` context appears, since this is where the actual conversion from and to text strings occurs. `tSocketTransceiverF` will output an incoming message `m` as `Just m`, and if the connection is closed by the other side, it will output `Nothing`.

In the server, we will wait for connections, and create client handlers when new clients connect. This is accomplished with `tSocketServerF`:

```
tSocketServerF :: (Text c, Text s) => TPort c s →
        (F s (Maybe c) → F a (Maybe b)) → F (Int,a) (Int,Maybe b)
```

So `tSocketServerF` takes two arguments, the first one is the port number to listen on for new clients. The second argument is the client handler function. Whenever a new client connects, a socket transceiver fudget is created and supplied to the client handler function, which yields a client handler fudget. The client handler is then spawned inside `tSocketServerF`. From the outside of `tSocketServerF`, the different client handlers are distinguished by unique integer tags. When a client handler emits `Nothing`, `tSocketServerF` will interpret this as the end of a connection, and kill the handler.

The idea is that the client handlers should use the transceiver argument for the communication with the client. Complex handlers can be written with `loopThroughRightF`, if desired. In many cases though, the supplied socket transceiver is good enough as a client handler directly. A simple socket server can therefore be defined by:

```
simpleTSocketServerF :: TPort c s → F (Int,s) (Int,Maybe c)
simpleTSocketServerF port = tSocketServerF port id
```

6.3 Avoiding Type Errors Between Client and Server

By using the following style for developing a client and a server, we can detect when the client and the server disagree on the message types:

First, we define a typed port to be used by both the client and the server. We put this definition in a module of its own. Suppose that the client sends integers to the server, which in turn can send strings:

```
module MyPort where
myPort :: TPort Int String
myPort = tPort 9000
```

We have picked an arbitrary port number. Now, if the client is as follows:

```
module Main where -- Client
import MyPort
...
main = fudlogue (... tSocketTransceiverF myPort ...)
```

and the server

```
module Main where -- Server
import MyPort
...
main = fudlogue (... tSocketServerF myPort ... )
```

then the compiler can check that we don't try to send messages of the wrong type. Of course, this is not foolproof. There is always the problem of having inconsistent compiled versions of the client and the server, for example. Or one could use different port declarations in the client and the server.

Now, what happens if we forget to put a type signature on `myPort`? Is it not possible then that we get inconsistent message types, since the client and the server could instantiate `myPort` to different types? The answer is no, and this is because of a subtle property of Haskell, namely the monomorphism restriction. A consequence of this restriction is that the type of `myPort` can not contain any type variables. If we forget the type signature, this would be the case, and the compiler would complain. It is possible to circumvent the restriction by explicitly expressing the context in the type signature, though. When defining typed ports, it defeats the purpose, of course:

```
module MyPort where
myPort :: Text a => TPort a String -- No no, don't do this!
myPort = tPort 9000
```

6.4 Example: Calendar

Outside the lunch room in our department, there is a whiteboard where the week's activities are registered. We will look at an electronic version of this calendar, where people can get a view like this on their workstation (Figure 37).

The entries in the calendar can be edited by everyone. When that happens, all calendar clients should be updated immediately.

The calendar consists of a server maintaining a database, and the clients, running on the workstations.

6.4.1 The Calendar Server

The server's job is to maintain a database with all the entries on the whiteboard, to receive update messages from clients and then update the other connected clients. The server con-

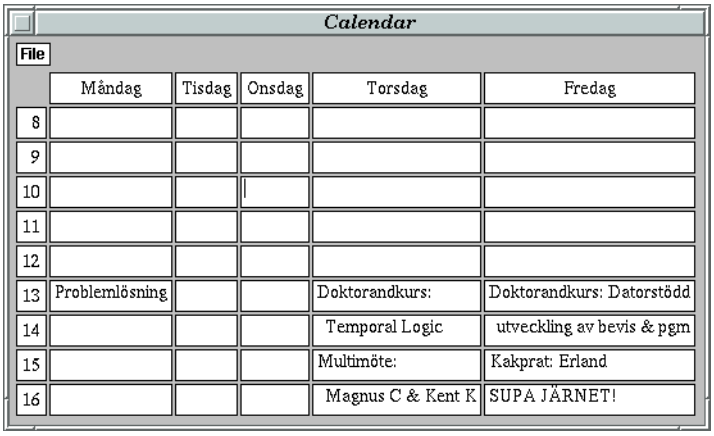


Fig. 37. The calendar client.

sists of the stream processor `databaseSP`, and a `tSocketServerF`, where the output from the stream processor goes to `tSocketServerF`, and vice versa (Figure 38).

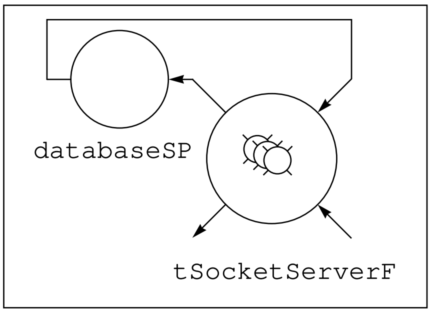


Fig. 38. The structure of `server`. The small fudgets are client handlers created inside the socket server.

`databaseSP` maintains two values: the client list `cl`, which is a list of the tags of the connected clients, and the database `db`, organised as a list of (key,value) pairs. This database is sent to newly connected clients. When a user changes an entry in her client, it will send that entry to the server, which will update the database and use the client list to broadcast the new entry to all the other connected clients. When a client disconnects, it is removed from the client list. The client handlers (`clienthandler`) initially announce themselves with `NewHandler`, then they apply `HandlerMsg` to incoming messages.

Here is a complete listing of the server:

```
module Main where -- Server
import Fudgets
import MyPort(myPort) -- also used in the client

main = fudlogue (server myPort)

data HandlerMsg a = NewHandler | HandlerMsg a
server port = loopF (databaseSP [] [] >^^=<
                    tSocketServerF port clienthandler) where
  clienthandler transceiver =
      -- New client - announce myself,
      -- convert Just a → Just (HandlerMsg a)
    put1SP (Just NewHandler) (mapSP (mapMaybe HandlerMsg))
    >^^=< transceiver
  databaseSP cl db =
```

```
getSP $ \(i,e) →
let clbuti = filter (/= i) cl
in case e of
        -- A message from client number i:
      Just handlermsg -> case handlermsg of
           -- A new client, send the database to it,
           -- and add to client list:
         NewHandler -> putSP [(i,d) | d <- db] $
                       databaseSP (i:cl) db
           -- Update entry in the database...14
         HandlerMsg s -> let db' = replace s db in
           -- ... and tell the other clients
                         putSP [(i',s) | i' <- clbuti] $
                         databaseSP cl db'
        -- A client disconnected, remove it from
        -- the client list:
      Nothing -> databaseSP clbuti db
```

`replace` and `mapMaybe` are defined in the Fudget library:

```
replace :: (Eq a) => (a,b) → [(a,b)] → [(a,b)]
replace p [] = [p]
replace (t, v) ((t', v') : ls') | t == t' = (t, v) : ls'
replace p (l : ls') = l : replace p ls'

mapMaybe:: (a → b) → Maybe a → Maybe b
mapMaybe f Nothing = Nothing
mapMaybe f (Just x) = Just (f x)
```

The type of the (key,value) pairs in the database is the same as the type of the messages received and sent, and is defined in the module `MyPort`:

```
module MyPort where
import Fudgets
type SymTPort a = TPort a a
myPort :: SymTPort ((String,Int),String)
  --          e.g. (("Torsdag",13),"Doktorandkurs:")
port = tPort 8888
```

Exercises

17. Implement the calendar client.

7 Conclusions

Stream processors and fudgets make it possible to structure programs as networks of concurrent processes in purely functional languages, like Haskell. This is a useful program structure when a program interacts with several external entities and all the time has to be prepared to react to input from any external source. We have shown two concrete examples of this. In Graphical User Interface programming the external entities that the program interacts with are the GUI components. In Client/Server programming the server usually interacts with several clients.

14. Unfortunately, the update will not take place until a new client connects, resulting in a space leak. It can be eliminated by inserting `seq (force db') $` after `let db' = replace s db in`.

Rather than being a new mechanism for I/O, the Fudget concept is an abstraction that can be implemented on top of many existing I/O systems, e.g., stream based I/O, monadic I/O, or the I/O model used in Clean. Although fudgets can be implemented on top of sequential I/O models, fudgets give the feeling of programming in a parallel language.

The Fudget combinators allow programs to be built in a hierarchical way. The basic building blocks are fudgets. Complete programs are fudgets too. This makes it easy to use existing applications as components when writing new, larger applications.

Polymorphism and higher order functions are valuable features of functional languages that allow libraries of re-usable software components to be both flexible and type safe. The Fudget library clearly benefits from this.

7.1 More Information on Fudgets

There are lots of things we didn't write about (because of time and space constraints). For example, these notes don't say much about the implementation of the Fudget GUI toolkit or how to write new GUI elements. We haven't said anything about parallel implementations of stream processors. We have not shown any large programming examples.

The WWW home page for Fudgets is located at URL

```
http://www.cs.chalmers.se/Fudgets/
```

There you can find pointers to more information on fudgets. You can also run demo programs and browse the hypertext version of the Fudget Library Reference Manual.

The Fudget library is distributed free of charge by anonymous ftp from `ftp.cs.chalmers.se` (for more info, see the Fudgets home page).

7.2 Acknowledgements

Thanks to Johan Jeuring, Andrew Moran and Jan Sparud for comments on these notes. Jan Sparud also implemented the first version of Name Layout, described in Section 5.5.3. John Hughes came up with the idea of default parameters (Section 5.6).

References

[1] M. Carlsson & T. Hallgren, Fudgets - A Graphical User Interface in a Lazy Functional Language, in *FPCA 93' - Conference on Functional Programming Languages and Computer Architecture*, pages 321--330, June 1993.

[2] M. Carlsson & T. Hallgren, The Fudget distribution, See ftp://ftp.cs.chalmers.se/pub/haskell/chalmers/

[3] A. D. Gordon, *Functional Programming and Input/Output*, Cambridge University Press, 1994. ISBN 0-521-47103-6.

[4] A. D. Gordon et al, *Haskell 1.3 Monadic I/O Definition*. At http://www.cl.cam.ac.uk/users/adg/io.html

[5] T. Hallgren & M. Carlsson, *The Fudgets Home Page*. At http://www.cs.chalmers.se/Fudgets/

[6] P. Hudak & R. S. Sundaresh. *On the expressiveness of purely functional I/O systems*. Research Report YALEU/DCS/RR-665, Yale University Department of Computer Science, March 1989.

[7] Paul Hudak et al., *Report on the Programming Language Haskell: A Non-Strict, Purely Functional Language*, March 1992. Version 1.2. Also in Sigplan Notices, May 1992.

[8] P.J. Landin. A correspondence between ALGOL 60 and Church's lambda-notation: Parts I and II. *Communications of the ACM*, 8(2,3):89-101, 158-165, February and March 1965.

[9] J. McCarthy. A basis for a mathematical theory of computations. In P. Brattort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, 1963.

[10] R. Milner, *Communication and concurrency*, Prentice-Hall International, 1989. ISBN 0-13-114984-9.

[11] R. Noble & C. Runciman, *Functional Languages and Graphical User Interfaces – a review and a case study*, Technical report YCS-94-223, Dept. of Comp. Sci., Univ. of York, Heslington, York, Y01 5DD, England, 1994.
At ftp://minster.york.ac.uk/reports/YCS-94-223.ps.Z

[12] Simon L. Peyton Jones, Cordelia V. Hall, Kevin Hammond, Will Partain, and Philip Wadler "The Glasgow Haskell compiler: a technical overview" In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, July 93.
At ftp://ftp.dcs.gla.ac.uk/pub/glasgow-fp/papers/grasp-jfit.ps.Z

[13] R. Plasmeijer, *Cleans' Home Page*. At http://www.cs.kun.nl/~clean/

[14] P. Wadler, "Monads for functional programming". In *Lecture Notes on Advanced Functional Programming Techniques* (i.e., this volume), LNCS, Springer-Verlag 1995.