

A Monad of Imperative Streams

Enno Scholz

Institut für Informatik, Freie Universität Berlin
14195 Berlin, Germany

Abstract

A new approach is presented for performing concurrent I/O in a functional programming language. A new monad St is introduced which generalizes Haskell's IO monad: A value of type $St\ a$ represents an imperative program which, *at certain times during its execution*, will produce a value of type a . In contrast, a value of type $IO\ a$ represents an imperative program which, *at the end of its execution*, will produce a value of type a . Not only may values of type $St\ a$ be used to represent imperative commands, they also serve to create so-called *imperative streams*. The computer's physical input devices are represented as primitive streams. Composite streams may be constructed and interleaved in a non-preemptive way using, for instance, the monad's *bind* operation.

By way of a series of example programs, the usage of imperative streams is illustrated. Moreover, an implementation in terms of the IO monad is given that is suitable for executing the example programs on the Gofer interpreter. An extension of the graphical user interface system PIDGETS with imperative streams is planned.

1 Introduction

This paper presents a new approach to performing concurrent I/O in a functional programming language. With concurrent I/O we mean I/O in the presence of multiple input devices, on which values may become available concurrently. An important application area is graphical user interfaces (GUIs). A GUI application must respond to input which may, at different times, be generated from, for instance, the computer's keyboard, from the mouse, or from the timer. In addition to these *physical input devices*, it has proven to be a useful abstraction to represent GUI elements such as buttons and scrollbars as *virtual I/O devices* (see, for instance, [Finne, Peyton Jones 96]).

The canonical problem when handling input from multiple sources is how to wait for the first data item to become available from either of two input devices, since it cannot be known beforehand on which device the item will become available. According to their solutions to this problem, many existing approaches fall into one of the following three categories:

The *classical streams* approaches, such as [Kahn, MacQueen 77], [Stoy 86], [Turner 87], [Breitinger et al. 95], view an input device as a stream of values becoming available in the course of time. A stream is represented as a list which is evaluated lazily. Each input device is represented as a primitive stream. In order to wait for the first data item to become available from either of two streams, a non-deterministic merge primitive is provided which merges two lists into one, such that the order of data items in the resulting list is the temporal order in which the data items become available on their respective devices. Because non-determinism destroys referential transparency, most approaches restrict it to the system level.

The *abstract streams* approaches represent a stream of values [Carlsson, Hallgren 93], [Noble, Runciman 95], or an object whose value varies discretely [Scholz 96] or continuously [Elliott, Hudak 96] over time, by an object of a primitive data type instead of a by a list. This allows primitive combinators for merging devices to be provided without compromising referential transparency.

In the *explicit concurrency* approaches [Peyton Jones et al. 95], [Grieskamp et al. 95], [Scholz 95] based on the IO monad [Peyton Jones, Wadler 93], input devices are represented as objects of a primitive data type, too. However, a value is read from a device using an imperative read primitive which blocks the thread executing it until a value is available from the device. In order to determine which of two input devices is the next to produce a value, two concurrent threads are spawned, each of which runs in a loop, forever reading the input device, blocking until an input value becomes available, and then writing the value to a common output device.

Our approach, like the explicit concurrency approaches, is based on the well-known framework for imperative functional programming using monads. However, instead of using explicit concurrency in Haskell's IO monad, we introduce a new monad St which is a generalization of the IO monad and replaces it: A value of type St

a represents an imperative program which, *at certain times during its execution*, will produce a value of type a . In contrast, a value of type $IO\ a$ represents an imperative program which, *at the end of its execution*, will produce a value of type a . Values of type $St\ a$ serve to create so-called imperative streams. Input devices are represented as primitive streams. In order to determine which of two streams is the next to produce a value, streams may be interleaved non-preemptively using, for instance, the St monad's *bind* operation.

In contrast to the explicit concurrency approaches, our approach has the property of producing programs which are deterministic with reference to their input. In contrast to the abstract streams approaches, our imperative streams allow arbitrary monad I/O commands to be performed exactly as in the IO monad.

We expect the new monad to be useful for programming reactive systems such as graphical user interfaces and plan to enhance our system PIDGETS [Scholz 96] to work with imperative streams.

The paper is organized as follows: Section 2 gives an informal introduction to the primitives defined on the St monad, illustrating their use with a series of example programs. Section 3 gives an implementation for the St monad by implementing it in terms of the IO monad. Section 4 draws conclusion.

The implementation given in Section 3 may be used to run all the paper's examples using Mark Jones's Gofer interpreter [Jones 94]. Since the standard distribution of Gofer supports only one physical input device, namely, the keyboard, the examples in Section 2 make use of no other physical input devices. However, it is demonstrated how the implementation may easily be extended to cater for multiple input devices.

2 Using imperative streams

A new monad St is presented which generalizes Haskell's IO monad in the following ways:

1. A value of type $IO\ a$ represents an imperative program which, *at the end of its execution*, will produce a value of type a . In contrast, a value of type $St\ a$ represents an imperative program which, *at certain times during its execution*, will produce a value of type a .
2. In the IO monad, only imperative commands are represented as primitive operations on the monad, whereas in the St monad, also input devices are represented as primitive operations.
3. In the IO monad, running *bind* $ma\ f$ means: first execute ma , wait till it produces a value a , then execute $f\ a$, wait till it produces a value b , then produce b . In the St monad, running *bind* $ma\ f$ means: execute ma , while ma is executing, whenever it produces a value a , run $f\ a$ concurrently until ma produces the next value. Whenever the current $f\ a$ produces a value b , produce b .

An *imperative stream* can be seen as a state machine. Every time one of the computer's input devices produces a new value of a given type, the imperative stream can perform a side effect, change its internal state, produce a value, or stop.

It may come as a surprise that imperative streams are not first-class objects in our model; instead, operations creating imperative streams are. A value of type $St\ a$ represents an operation which, when executed, will create a stream that is able to produce values of type a . Note that the operation may perform side effects.

The main program must be of type $St\ ()$. Its semantics are (i) the side effects performed by executing it, and (ii) the side effects performed by the stream it creates. This stream is called the *top-level stream*. The program terminates when the top-level stream stops.

Streams may either be device streams or programmer-defined streams. *Device streams* represent the computer's input devices and produce a value whenever it has become available from the input device. *Programmer-defined streams* may be constructed using a number of primitives defined on the St monad. They can again be distinguished according to whether they are *composite streams* or *leaf streams*. All the streams in the system are organized as a tree whose root is the top-level stream.

In addition to aggregation, two other relations are possible between streams:

- *Source-sink*: Whenever the *source stream* produces a value, the *sink stream* reacts by producing the same value. One source may have multiple sinks, but each sink has exactly one source.
- *Trigger-target*: Whenever the *trigger stream* produces a new value a , the *target stream* destroys itself (and recursively all of its components), executes an operation obtained by applying a fixed trigger function f to a , and then becomes the resulting stream. Each trigger has exactly one target, and vice versa.

A stream may be in one of two states: either *dormant* or *active*. A stream is *dormant* if has never produced a value yet. After it has produced its first value, it is called *active*. An active stream keeps track of the last value it has produced, called its *most recent value*.

1.1 The basics

All Haskell I/O instructions, customarily represented by operations on the *IO* monad, have counterparts defined on the *St* monad. All of these primitives have in common that, when executed, they perform the designated side effect of their counterpart in the *IO* monad and then create a new stream. This stream immediately produces the value returned by the I/O operation and then stops.

```
putString :: String → St ()
newVar   :: a → St (MutableVar RealWorld a)
writeVar :: MutableVar RealWorld a → a → St ()
readVar  :: MutableVar RealWorld a → St a
```

For instance, *putString s* performs the side effect of printing out *s*, then creates a stream which immediately produces the value *()* and then stops. *newVar a* performs the side effect of creating a new mutable variable with initial value *a*, then creates a stream which immediately produces the reference to the new variable and then stops. *readVar v* reads the current content *a* of *v*, then creates a stream which immediately produces the value *a* and then stops.

The computer's input devices, most notably the mouse, the timer, and the keyboard, are represented by device streams. Device streams never perform any side-effects and never terminate. For instance, any time the user enters a character, the keyboard stream will produce that character.

```
keyboard :: St Char
```

When executed, the operation *keyboard* will create a leaf stream *s* which is a sink of the keyboard stream, i.e., whenever, the user enters a character, this will be produced by the keyboard stream and consequently, by *s*. Note the following: *If the keyboard stream is active at the time s is created, the keyboard's most recent value, i.e., the last character the user entered, will immediately be produced by s.*

result a performs no side-effects and creates a stream which immediately produces the value *a* and then stops.

```
result :: a → St a
```

The *St* monad's *bind* operation is one of two ways to build composite streams.

bind aSt f is an operation which, when executed, will do the following: First, it will execute *aSt*, creating a stream *s1*. Second, it will wait until *s1* produces its first value *a*. Third, it will apply *f* to *a*, creating a stream *s2*. Fourth, it will set up *s1* as *s2*'s trigger, with trigger function *f*. Fifth, it will create a stream *s* and set up *s2* as *s*'s sink. Sixth it will return *s*. Because *s1* is *s2*'s trigger, *s2* will be destroyed and replaced by the result of *f a'* whenever *s1* produces a new value *a'*.

```
bind :: St a → (a → St b) → St b
```

The following example *example1* illustrates how streams are combined with *bind*. Like all of the example programs in this paper, the code of *example1* is given together with an *I/O diagram* which illustrates what output is generated

by each character the user enters. In an I/O diagram, the input column contains an entry for every character the user enters. The corresponding entry in the output column shows the characters which the program outputs in response. If the program does not make any output in response, the entry in the output column is "-". In order to show the output which the program produces when it is initialized, the first entry in the input column is always "-". Thus, the I/O diagram below illustrates that, initially, *example1* prints out "Hello world\n", and after that, it echoes every character the user types in.

Example 1	Input	Output
<pre>example1 = do putString ("Hello world!\n") char ← keyboard putString (show char ++ "\n")</pre>	<pre>- a b</pre>	<pre>Hello world\n a\n b\n</pre>

Note that *example1* never terminates: Whenever the keyboard stream produces a new value, it is bound to *char*, and *putString (show char ++ "\n")* is reexecuted.

To understand the behavior of *example1*, let us expand Haskell 1.3's *do* notation [Peterson et al. 96]:

```
example1' :: St ()
example1' = bind (putString ("Hello world!\n"))
            (\_ → bind keyboard
              (\char → putStrLn (show char ++ "\n")))
```

In Fig. 1, *example1* is represented graphically using a so-called *stream diagram*. In a stream diagram, every stream is represented by a box containing the code which was executed to obtain the stream. To this box, a smaller box containing the stream's *stream identifier* is attached unless the stream is a device stream. The identifier of the top-level stream is always "s". A composite stream's components are numbered; the identifier of any programmer-defined stream other than the top-level stream is the name of its parent to which its own number is appended. Solid arrows point from composite streams to component streams; dashed lines point from a source stream to its sink streams; stippled lines point from a trigger stream to its target stream. The stippled arrow is optionally annotated with the name of the variable that is bound each time the trigger produces a value.

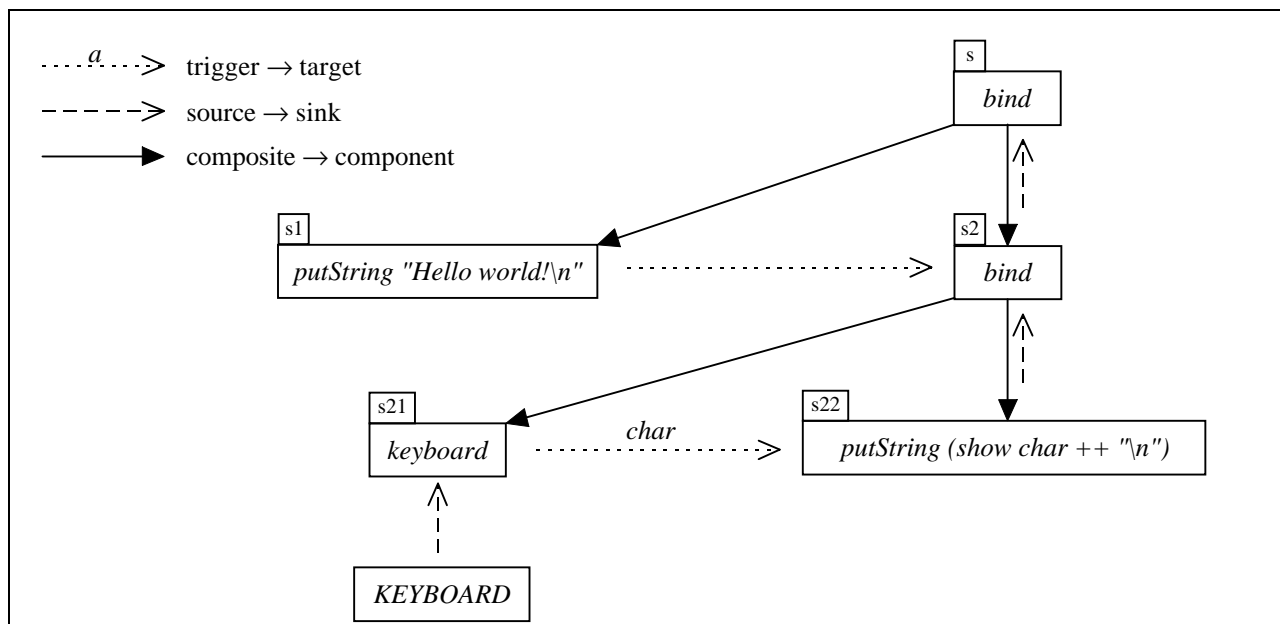


Fig. 1: Stream diagram for *example1*

When executed, *example1* first creates *s1*, executing *putString ("Hello world!\n")*. Since *s1* produces its first value () without delay, *bind keyboard (\char → putString (show char ++ "\n"))* may be executed immediately, which causes *keyboard* to be executed, and *s21* to be created. Now a delay occurs until the first value of the keyboard stream is available, i.e., until the user has entered a character. When the user has entered the first character, say, 'a', *putString (show 'a' ++ "\n")* is executed, creating *s22*. Then *s21* is set up as *s22*'s trigger with trigger function $(\text{char} \rightarrow \text{putString}(\text{show char} ++ "\n"))$ and *s2* is created and set up as *s22*'s sink. Then *s1* is set up as *s2*'s trigger and *s* is created and set up as *s1*'s sink. From now on, whenever the user enters a new character, *s22* is destroyed and replaced by a new version obtained by executing $(\text{char} \rightarrow \text{putString}(\text{show char} ++ "\n"))$ applied to the new character, which causes the character to be echoed as a side-effect. Because the keyboard never stops, *s21* will never stop, *s22* will be destroyed regularly, to be replaced by a new version that immediately produces (), which is propagated via *s2* to *s*, which will thus never stop.

In the sequel, we will make use of stream diagrams for larger programs than *example1*. That is why it is worthwhile to introduce a slightly more condensed notation by compressing composite streams created with *bind* whose left component streams stop immediately. Fig. 2 shows the compressed stream diagram for *example1*.

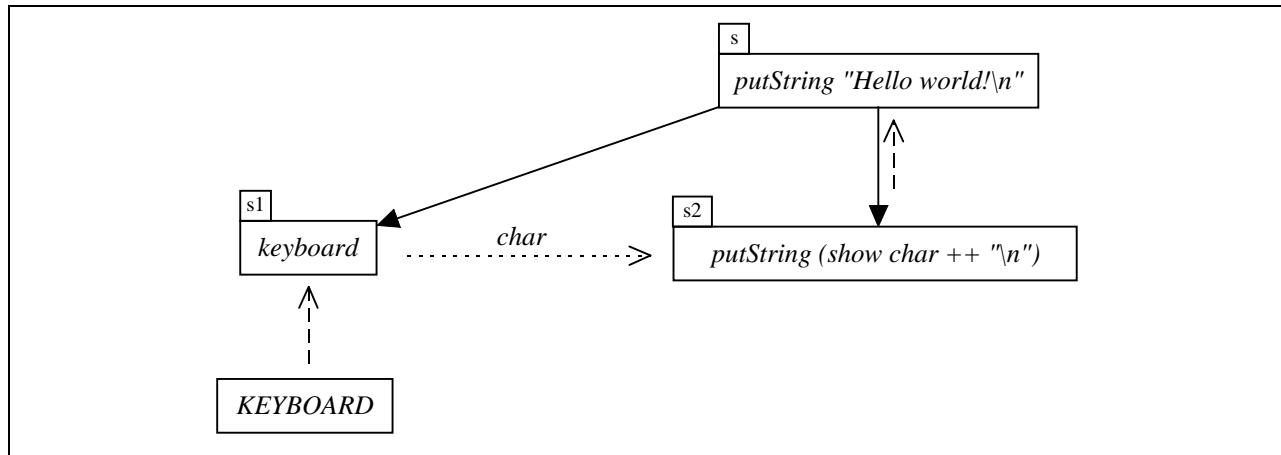


Fig. 2: Compressed stream diagram for *example1*

1.2 Interleaving side effects

One source stream may have multiple sinks, each of which may trigger a different target, causing different side effects to be performed when the targets are replaced. Thus, the question arises in which order a source's sinks are notified of the fact that a new value has been produced, enabling them to react to this event by triggering their target and thus causing side-effects.

Consider the following example program:

Example 2	Input	Output
<pre> example2 = do keyboard putString "Hi\n" keyboard putString "Ho\n" </pre>	<pre> - a b </pre>	<pre> - Hi\nHo\n Hi\nHo\n </pre>

It has the stream diagram given in Fig. 3.

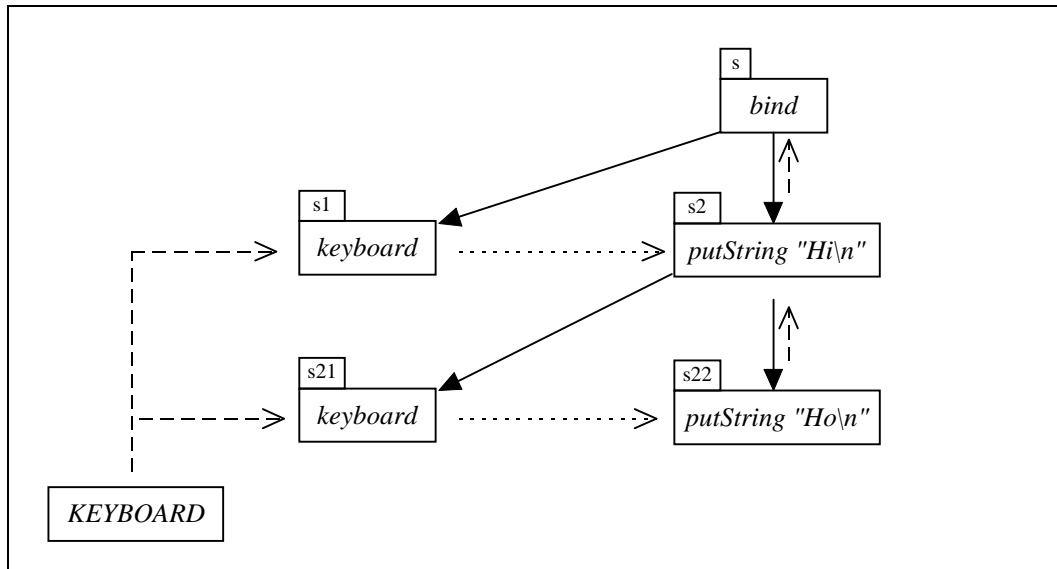


Fig. 3: Stream diagram for *example2*

When the keyboard stream produces a new character, the question is which of $s1$ or $s21$ are notified first, yielding the following alternatives:

- $s21$ is notified first, causing it to trigger $s22$, causing "Ho\n" to be printed. $s1$ is notified second, causing it to trigger $s2$, causing "Hi" to be printed, causing a new $s21$ to be created. Since the keyboard stream is active, its most recent value will be produced by the new $s21$ immediately, enabling the new $s22$ to be created without delay, causing "Ho\n" to be printed. This alternative means that *example2* would output "Ho\nHi\nHo\n" every time the user enters a character.
- $s1$ is notified first, causing it to trigger $s2$, causing "Hi" to be printed, causing a new $s21$ to be created. Since the keyboard stream is active, its most recent value will be produced by the new $s21$ immediately, enabling the new $s22$ to be created without delay, causing "Ho\n" to be printed. Now the old $s21$ should be notified, but this does not happen because it no longer exists. This means that *example2* would output "Hi\nHo\n" every time the user enters a character.

As is obvious from the above I/O diagram, the second alternative is correct. More generally, values are propagated according to the following two rules:

1. A source's sinks are notified in the lexical order of their stream identifiers.
2. Destroyed sinks are not notified

1.3 Using the *map* operation

In any monad, the following law holds (cf. [Wadler 92]):

$$\text{map } f \text{ ma} = \text{bind } \text{ma } (\lambda a \rightarrow \text{result } (f a)).$$

For the St monad, according to the definitions of *bind* and *result*, it can be deduced that executing $\text{map } f \text{ aSt}$ will create a stream s' which has the following relation to the stream s which would be created by executing aSt : s' is dormant while s is dormant, it performs the same side effects as s , and at any time s produces the value a , s' produces the value $f a$.

This is illustrated by the following program *example3*. For every character the user enters, it prints out whether the character is a 'b'.

Example 3	Input	Output
<pre>example3 = do boolean ← map (== 'b') keyboard putString (show boolean ++ "\n")</pre>	<pre>- a b c</pre>	<pre>- False\n True\n False\n</pre>

In the sequel, it will be useful to monitor the state transitions of a given stream. Therefore, let us encapsulate this in an abstraction *monitor* which prints out the new observable state of a stream each time it makes a state transition.

```
monitor :: Text a ⇒ St a → St ()
monitor ma =
  do a ← ma
    putString (show a ++ "\n")
```

Using *monitor*, *example3* may be rephrased as follows:

```
example3' =
  monitor (map (== 'b') keyboard)
```

1.4 Defining streams with internal state

In addition to their observable state, streams can have an internal state that is stored in mutable variables. This is illustrated by the following abstraction *history* which serves to record a stream's past observable states.

```
history :: St a → St [a]
history aSt =
  do asVar ← newVar []
    a ← aSt
    as ← readVar asVar
    writeVar asVar (as ++ [a])
    result (as ++ [a])
```

In the implementation of *history*, note that *newVar* and *aSt* are executed only once, while *readVar*, *writeVar*, and *result* are executed once for every new value produced by the stream created when executing *aSt*.

The following example *example4* illustrates the use of *history*.

Example 4	Input	Output
<pre>example4 = monitor (history keyboard)</pre>	<pre>- a b c</pre>	<pre>- a\n ab\n abc\n</pre>

Since the problem explained in the following subsection will make further use of *history*, let us be a little more precise about its semantics: the stream *s'* created by executing *history aSt* at a given time *t* will have the following relationship to the stream *s* obtained by executing *aSt* at *t*: Whenever *s* produces a value *a_n*, *s'* will produce a value which is the list *[a₁, ..., a_n]* of values *s* has ever produced.

1.5 Making multiple references to a stream

So far, we have made a great effort to distinguish between a value of type *St a*, which denotes a command for creating a stream, and the stream itself. The following example *example5* illustrates the reason for this.

Example 5	Input	Output
<pre>example5 = do keys1 ← history keyboard keys2 ← history keyboard putString (show (keys1, keys2) ++ "\n")</pre>	<pre>- a b c</pre>	<pre>- ("a", "a")\n ("ab", "b")\n ("abc", "c")\n</pre>

At first, it may come as a surprise that *keys1* and *keys2* are not bound to the same value for every character the user enters. However this behavior is easily explained using *example5*'s stream diagram, which is given in Fig. 4.

The stream diagram shows that, whenever the keyboard stream produces a new value, it is propagated to *s11*, triggering *s12*. When *s12* is replaced, it immediately produces a value which is propagated to *s1*, causing *s2* to be triggered, causing *s21*, *s211*, *s212*, and *s22* to be destroyed. Along with *s21*, its variable *asVar* is destroyed. Sure enough, a new version of *s21* will eventually be created, but it creates a new variable named *asVar*, which initially stores the empty list. When the new *s212* is created, it will immediately produce the singleton list containing the new character. This list will be bound to *keys2* when *s22* is created anew, and thus, the behavior illustrated by *example5*'s I/O diagram is explained.

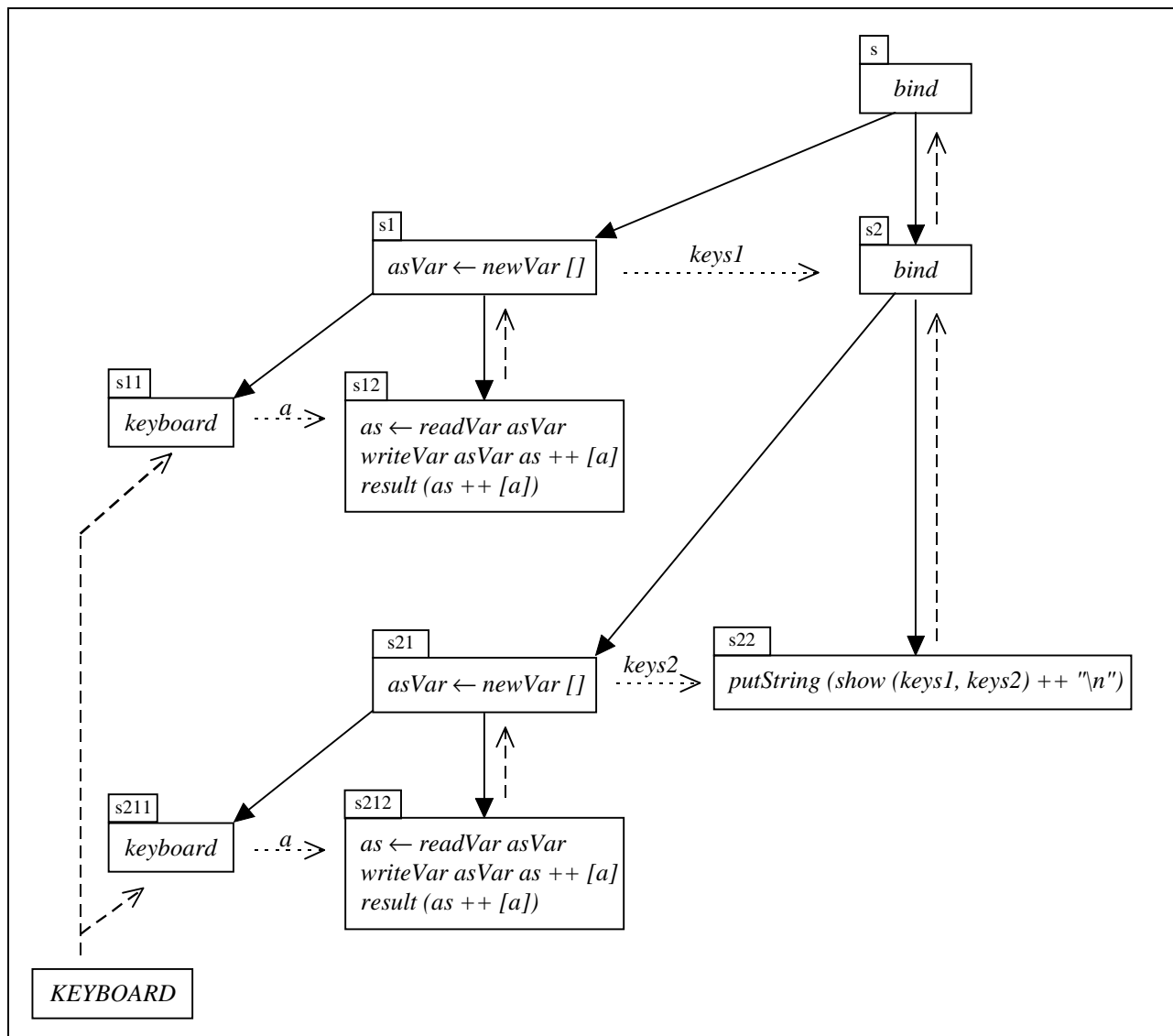


Fig. 4: Stream diagram for *example5*

In order to be able to refer to one stream in various places in a program, the primitive *start* exists. *start aSt* is an operation that, when executed, will do the following: First, it will execute *aSt*, creating a stream *s1*. Second, it will create an operation *aSt'* which is defined as follows: When *aSt'* is executed, it will create a stream *s1'* and set it up as a sink of *s1*. *s1'* has the special property that it will stop if *s1* stops. Third, it will create a stream *s* which immediately produces the value *aSt'*. Fourth, it will return *s*.

```
start :: St a → St (St a)
```

Using *start*, we can rewrite *example5* such that, for every character the user types in, *keys1* and *keys2* are bound to the same values:

Example 6	Input	Output
<pre>example6 = do aSt ← start (history keyboard) keys1 ← aSt keys2 ← aSt putString (show (keys1, keys2) ++ "\n")</pre>	<pre>- a b c</pre>	<pre>- ("a", "a")\n ("ab", "ab")\n ("abc", "abc")\n</pre>

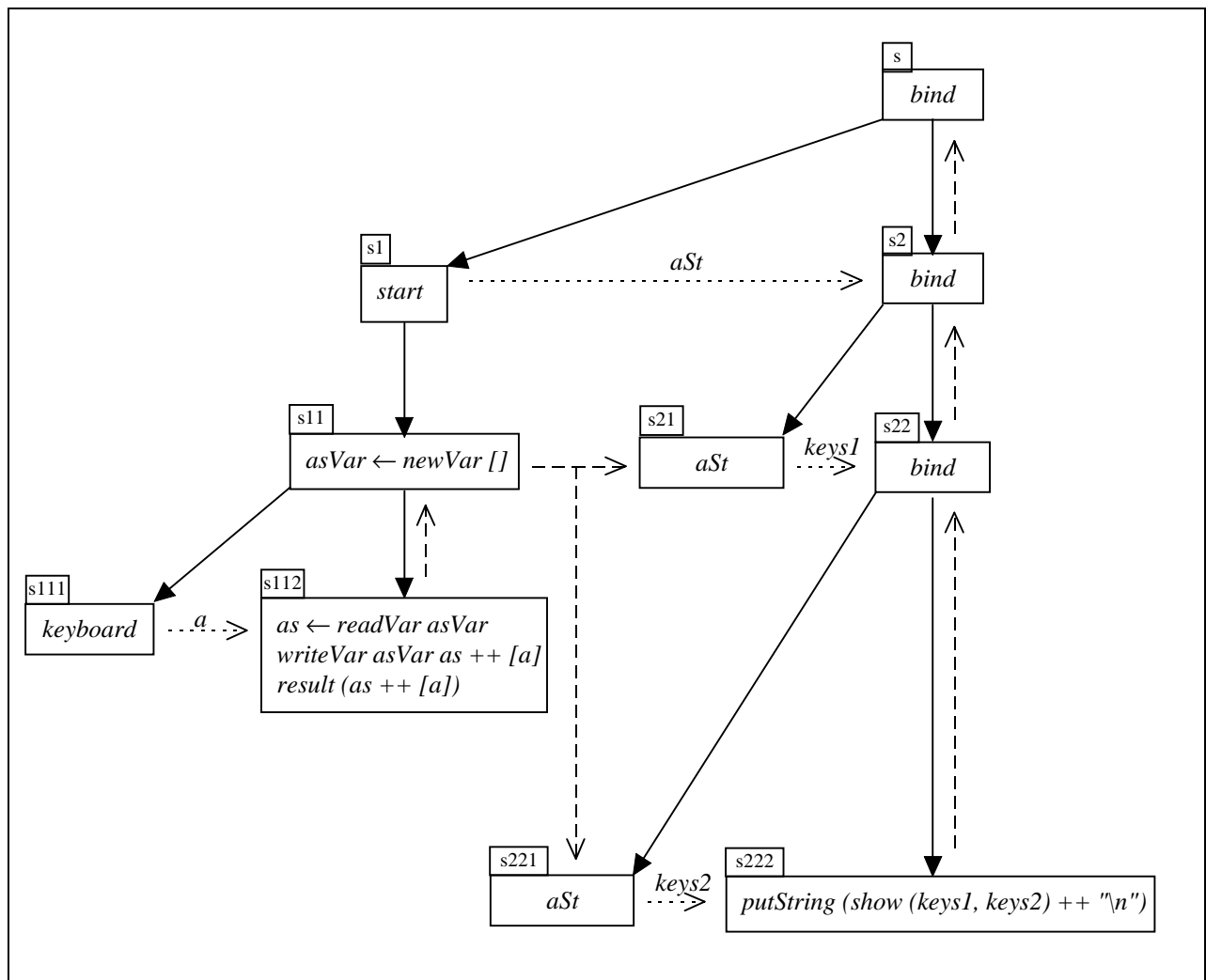


Fig. 5: Stream diagram for *example6*

The above stream diagram shows how both occurrences of aSt refer to the stream created when executing *history keyboard* only once.

From now on, we will no longer distinguish between values of type $St\ a$ and streams unless there is a possibility for confusion.

1.6 Testing whether a stream is dormant

As mentioned in the definition of *history* above, if *history*'s argument stream is dormant, the stream returned is dormant, too. However, a dormant stream's list of past values is well defined, namely, to be the empty list. Thus, a primitive for testing whether a stream is dormant is needed. Here it is:

isDormant :: $St\ a \rightarrow St\ Bool$

isDormant aSt denotes an operation which, when executed, will do the following: First, aSt is executed, returning a stream s . Second, a stream s' is created and returned, whose behaviour is defined as follows: If s is initially active, s' immediately produces the value *False* and stops. If s is initially dormant, then s' immediately produces the value *True*. Should s ever become active later on, s' will produce the value *False* and stop.

Using *isDormant*, we can rewrite *history* such that *history aSt* will never be dormant but rather, while aSt is dormant, the observable state of *history aSt* will be the empty list.

```

history' aSt =                -- incorrect
  do  $b \leftarrow isDormant\ aSt$ 
    if  $b$  then result []
    else history aSt

```

example7 seems to demonstrate this:

Example 7	Input	Output
<pre> <i>example7</i> = monitor (<i>history' keyboard</i>) </pre>	<pre> - a b c </pre>	<pre> \n a\n ab\n abc\n </pre>

However, our definition of *history'* is not correct yet. For instance, *example8* yields the following undesirable output:

Example 8	Input	Output
<pre> <i>example8</i> = monitor (<i>history' (putString "Hello\n")</i>) </pre>	<pre> - </pre>	<pre> Hello\nHello\n </pre>

Because aSt occurs twice in the definition of *history'*, it is executed twice!

Again, this problem can be remedied by using the primitive *start*. Here is the correct definition of *history*:

```

history'' aSt =                -- correct
  do  $aSt' \leftarrow start\ aSt$ 
     $b \leftarrow isDormant\ aSt'$ 
    if  $b$  then result []
    else history aSt'

```

1.7 The art of not producing a value

With the operations on the monad St that were introduced so far, only streams can be constructed that produce one value initially, and one for every time the user presses a key. What if we want to write an abstraction

```
lines :: St String
```

which creates a stream that only produces a new value whenever the user has completed one line by pressing carriage return? The following operation nil makes this possible.

```
nil :: St a
```

Executing nil creates a stream which immediately stops, without producing any values.

This is how it is used in the definition of $lines$:

```
lines =
  do prefixStringVar ← newVar []
     char ← keyboard
     prefixString ← readVar prefixStringVar
     if char == '\n' then
       do writeVar prefixStringVar []
          result prefixString
     else
       do writeVar prefixStringVar (prefixString ++ [char])
          nil
```

In a mutable variable, $lines$ stores as its internal state the sequence $prefixString$ of characters typed in but not yet terminated by a carriage return. Whenever the user presses a key, $keyboard$ produces a value and the character typed in is bound to $char$. However, $lines$ produces a value only if this character is carriage return, in which case the new state is the current content of the character buffer. If the character is not carriage return, $lines$ only updates its internal state, but does not produce a value.

Example 9	Input	Output
$example9 =$ $monitor\ lines$	- H i \n y o u \n	- - - Hi\n - - - you\n

1.8 Customizing a stream's input devices

With the operations on monad St that were defined so far, streams could be constructed which are reactive to the computer's input devices. However, in the St monad, it is possible to customize a stream's perception of the computer's input devices. For instance, the primitive $withKeyboard$ changes a stream's perception of what the keyboard stream is.

```
withKeyboard :: St Char → St a → St a
```

withKeyboard keyboard' aSt is an operation which, when executed, does the following. First, *keyboard'* is executed, creating a stream *sI*. Second, *aSt* is executed in a special way causing all streams created with *keyboard* to be set up as sinks of *sI* instead of being set up as sinks of the keyboard stream.

For example, the following example program runs *lines*, but fools it into thinking that the user types only uppercase characters:

Example 10	Input	Output
<pre>example10 = do let keyboard' = map toUpper keyboard monitor (withKeyboard keyboard' lines)</pre>	<pre>- H i \n y o u \n</pre>	<pre>- - - HI\n - - - YOU\n</pre>

1.9 Two ACTIVEVRML-like combinators

We have now completed our survey of the primitives defined on the stream monad's primitives. To further illustrate their versatility, let us define two abstractions *snap* and *until* which bear some resemblance to ACTIVEVRML's *snapshot* and *until* [Elliott 96], [Elliott, Hudak 96].

First, let us define an auxiliary function on top of which *snap* and *until* may be defined. *snapWhen p aSt* is dormant until *aSt* produces a value *a* such that *p a* evaluates to *True*, then it produces *a* and stops.

```

snapWhen :: (a → Bool) → St a → St a
snapWhen p aSt =
  do isFirstTimeVar ← newVar True
     a ← aSt
     isFirstTime ← readVar isFirstTimeVar
     if isFirstTime && p a then
       do writeVar isFirstTimeVar False
          result a
     else
       nil

```

snap is just an instantiation of *snapWhen*: *snap aSt* is dormant until *aSt* produces its first value, then it produces that value itself and stops.

```

snapWhen :: St a → St a
snapWhen = snap (const True)

```

until bSt aSt1 aSt2 behaves like *aSt1* until *bSt* produces *True*. After that, it behaves like *aSt2*.

```

until :: St Bool → St a → St a → St a
until bSt aSt1 aSt2 =
  do b ← isDormant (snapWhen (== True) bSt)
     if b then aSt1
     else aSt2

```

example11 illustrates the use of *until* and *snap*. Any time the user enters a character which is not a digit, the character is printed *n* times in a row. Initially *n* is 1. Any time the user enters a series of digits, these are taken to be a new value for *n*. Initially, and whenever the user has entered a new value for *n*, this is printed out.

Example 11	Input	Output
<pre> example11 = do let parseInt digits = foldl (\sum char → 10 * sum + ord char - 48) 0 digits let loop n = do putStrLn ("n = " ++ show n ++ "\n") until (map isDigit keyboard) (do char ← keyboard putStrLn (copy n char ++ "\n")) (do digitsSt ← start (history keyboard) until (map (not . isDigit) keyboard) (result ()) (do digits ← snap digitsSt loop (parseInt (init digits))) loop 1 </pre>	<pre> - a 1 2 b c </pre>	<pre> n = 1\n a\n - - n = 12\nbbbbbbbbbbbbbb\n cccccccccccccc\n </pre>

In the implementation, note especially how a stream *digitSt* recording the history of the keyboard is started when the user enters the first digit, how a snapshot *digits* is made of *digitSt* when the user enters the first non-digit afterwards, and how all but the last element in *digits* is parsed to become the new value of *n*.

3 Implementing imperative streams

A stream producing values of type *a* is implemented by a function of type *Devices* → *IO (Maybe a, Bool)*, called a *stream function*.

```
type Stream a = Devices → IO (Maybe a, Bool)
```

Every time a new value *a* becomes available on one of the computer's input devices, the stream function is applied to a record of type *Devices* which for each input device indicates whether it has terminated, and what the new value is, if there is one. The resulting pair's first element indicates whether the stream produced a value in response (*Just a*) or not (*Nothing*), the second element indicates whether the stream has stopped.

In order to make the implementation of the *St* monad implementable in the standard version of Gofer, we restrict the physical input devices to be only the keyboard. However, let us prepare the machinery needed to cater for multiple input devices by defining a record type *Devices* with selectors and update functions which, for the moment, contains only the possible states of the keyboard.

```
type Devices = (Maybe Char, Bool)
```

```
selectKeyboard :: Devices → (Maybe Char, Bool)
```

```
selectKeyboard = id
```

```
updateKeyboard :: (Maybe Char, Bool) → Devices → Devices
```

```
updateKeyboard = const
```

Type *St a* denotes an operation for creating a stream. We define it as a restricted type synonym in Gofer (cf. [Jones 94]).

```
type St a = IO (Stream a) in result_St, bind_St, nil, isDormant, start, io, keyboard, withKeyboard, eval
```

This allows us to declare *St* to be an instance of type class monad, enabling the *do* notation for *St*.

```
instance Monad St where result = result_St
```

```
bind = bind_St
```

Every value *ioa* of type *IO a* has a counterpart of type *St a*, i.e., an operation which executes $a \leftarrow ioa$ and creates a stream which immediately returns *a* and stops. This means that the stream function returns (*Just a, True*).

```

io :: IO a → St a
io ioa =
  do isFirstTimeVar ← newVar True
     result (\devices → do isFirstTime ← readVar isFirstTimeVar
                           if isFirstTime then
                             do a ← ioa
                                writeVar isFirstTimeVar False
                                result (Just a, True)
                           else
                             result (Nothing, True))

```

The *St* monad's *result* can conveniently be implemented in terms of *io*.

```

result_St :: a → St a
result_St a = io (result a)

```

The *bind* operation first creates a stream function *aStream* representing the trigger stream. The source stream, which may be replaced when triggered, is stored in a mutable variable *bStreamMaybeVar*; initially, it is not present.

Whenever the stream function created by *bind* is invoked, it first invokes *aStream*. If this produces a new value *a*, a new *bStream* is created; it replaces *bStreamMaybeVar*'s current content. Then the current *bStream*, no matter whether it is new or not, is invoked, if it exists. The stream function created by *bind* stops if both *aStream* and *bStream* have stopped.

```

bind_St :: St a → (a → St b) → St b
bind_St aSt f = do aStream ← aSt
                  bStreamMaybeVar ← newVar Nothing
                  result (\devices → do (aMaybe, aHasTerminated) ← aStream devices
                                         case aMaybe of
                                           Just a →
                                             do bStream ← f a
                                                writeVar bStreamMaybeVar (Just bStream)
                                           Nothing →
                                             result ()
                  bStreamMaybe ← readVar bStreamMaybeVar
                  case bStreamMaybe of
                    Just bStream →
                      do (bMaybe, bHasTerminated) ← bStream devices
                         result (bMaybe, aHasTerminated && bHasTerminated)
                    Nothing →
                      result (Nothing, aHasTerminated))

```

The stream function created by *start* uses a mutable variable *isFirstTimeVar* analogously to *io*. Whenever invoked, it first invokes the stream function *aStream* created by executing *aSt*, then makes the result available in a mutable variable *aVar*. The first time it is invoked it returns an operation of type *St a* which, whenever invoked will return a stream function which, whenever invoked, returns the current result of *aStream* presently stored in *aVar*.

```

start :: St a → St (St a)
start aSt = do aVar ← newVar undefined
              aStream ← aSt
              isFirstTimeVar ← newVar True
              result (\devices → do (aMaybe, aHasStopped) ← aStream devices
                                     writeVar aVar (aMaybe, aHasStopped)
                                     isFirstTime ← readVar isFirstTimeVar
                                     if isFirstTime then
                                       do writeVar isFirstTimeVar False
                                          result (Just (result (\_ → readVar aVar)), aHasStopped)
                                     else
                                       result (Nothing, aHasStopped))

```

nil creates a stream function which returns *Nothing* and stops immediately.

```

nil :: St a
nil = result (\_ → result (Nothing, True))

```

The first time the stream function returned by *isDormant aSt* is invoked, it invokes the stream function *aStream* created by executing *aSt* and produces a boolean indicating whether *aStream* produced a value or not. If *aStream* did not produce a value, the stream function returned by *isDormant aSt* is prepared to produce the value *False* as soon as *aStream* produces its first value.

```

isDormant :: St a → St Bool
isDormant aSt = do aStream ← aSt
                  stateVar ← newVar (True, True)
                  result (\devices → do (aMaybe, aHasStopped) ← aStream devices
                                         (isDormant, isFirstTime) ← readVar stateVar
                                         case aMaybe of
                                           Nothing | isFirstTime →
                                             do writeVar stateVar (isDormant, False)
                                                result (Just True, aHasStopped)
                                           Just a | isDormant →
                                             do writeVar stateVar (False, False)
                                                result (Just False, aHasStopped)
                                           _ →
                                             result (Nothing, aHasStopped))

```

At any time, the streams associated with a given device will make the same state transitions as the device itself.

```

keyboard :: St Char
keyboard = result (\devices → result (selectKeyboard devices))

```

In *withKeyboard*, the state of the keyboard which *aStream* sees is determined by the result of *keyboardStream*.

```
withKeyboard :: St Char → St a → St a
withKeyboard keyboardSt aSt =
  do keyboardStream ← keyboardSt
     aStream ← aSt
     result (\devices → do (charMaybe, keyboardHasStopped) ← keyboardStream devices
                          aStream (updateKeyboard (charMaybe, keyboardHasStopped) devices))
```

The implementation of *withKeyboard* reveals why, in the definition of *Devices*, we had to account for the possibility of one of the devices terminating: Although physical devices won't terminate, a stream's devices may be customized to be arbitrary streams, which may of course terminate.

To run a value of type *St a*, the top-level stream is created which is first applied to *(Nothing, False)* and then to *(Just char, False)* for every character *char* read in. The program runs until the top-level stream stops.

```
eval :: St () → IO ()
eval aSt =
  do aStream ← aSt
     (\_ hasTerminated) ← aStream (Nothing, False)
     if hasTerminated then
       result ()
     else
       do let loop = do char ← getch
                    (\_ hasTerminated) ← aStream (Just char, False)
                    if hasTerminated then
                      result ()
                    else
                      loop
          loop
```

Note that, if Gofer were suitably extended with a primitive for querying, for instance, the X event queue, it would be trivial to introduce mouse pointer, mouse button, and timer devices by just extending the *Devices* record, writing function such as *mousePointer* and *withMousePointer*, and passing a suitable device structure to the top-level stream.

4 Conclusion

We have presented a new monad for performing concurrent I/O in a functional programming language. The new monad represents an integration of two well-known ideas: To represent streams of values of an arbitrary type using a primitive type constructor, and to represent imperative commands returning a value of an arbitrary type using a primitive monad: Our monad *St* encapsulates reactive objects which may, in response to values becoming available on the computer's input devices, both produce values and execute imperative commands. Moreover, a reactive object's perception of the computer's I/O devices may be customized, which is a property we consider to be of great importance for building reusable user interface components.

We have given an implementation suitable for running all the paper's examples using the Gofer interpreter. Shortly, we plan to explore the suitability of the *St* monad for interactive graphics programming in the context of our GUI system PIDGETS.

References

- [Breitinger et al. 95] S. Breitinger, R. Loogen, Y. Ortega-Mallen: *Towards a Declarative Language for Concurrent and Parallel Programming*, Glasgow FP Workshop 1995
- [Carlsson, Hallgren 93] M. Carlsson, T. Hallgren: *FUDGETS - A Graphical User Interface in a Lazy Functional Language*, Conference on Functional Programming Languages and Computer Architectures, 1993
- [Elliott 96] C. Elliott: *A Brief Introduction to ActiveVRML*, Technical Report MSR-TR-96-05, Microsoft Research, 1996
- [Elliott, Hudak 96] C. Elliott, P. Hudak: *The Essence of ActiveVRML*, Yale University, July 1996
- [Finne, Peyton Jones 96] S. Finne, S.L. Peyton Jones: *Composing the User Interface with HAGGIS*, Summer School on Advanced Functional Programming, Olympia, WA, Aug 25-30, Springer-Verlag LNCS, 1996
- [Grieskamp et al. 96] W. Grieskamp, T. Frauenstein, P. Pepper, M. Südholt: *Functional Programming of Communicating Agents and its Application to Graphical User Interfaces*, 2nd International Conference on Perspectives in System Informatics, Novosibirsk, Springer-Verlag LNCS 1996
- [Jones 94] M. Jones: *Gofer 2.21/2.28/2.30 Release Notes*, available by anonymous ftp from <ftp.cs.yale.edu>, 1994
- [Kahn, MacQueen 77] G. Kahn, D.B. MacQueen: *Coroutines and Networks of Parallel Processes*, IFIP 1977
- [Noble, Runciman 95] R. Noble, C. Runciman: *GADGETS: Lazy Functional Components for Graphical User Interfaces*, PLILP 1995
- [Peterson et al. 96] J. Peterson et al: *Haskell 1.3: A non-strict, purely functional language*, Technical Report YALEU/DCS-1106, Yale University 1996
- [Peyton Jones, Wadler 93] S. Peyton Jones, P. Wadler: *Imperative functional programming*, ACM Conference on the Principles of Programming Languages, ACM Press, 1993
- [Peyton Jones et al. 95] S. Peyton Jones, A. Gordon, S. Finne: *Concurrent Haskell*, ACM Symposium on the Principles of Programming Languages, ACM Press, 1996
- [Scholz 95] E. Scholz: *Four Concurrency Primitives for Haskell*, Haskell Workshop Proceedings, Yale University Research Report YALEU/DCS/RR-1075, 1995
- [Scholz 96] E. Scholz: *PIDGETS - Unifying Pictures and Widgets in a Constraint-Based Framework for Concurrent Functional GUI Programming*, 8th International Symposium on Programming Languages: Implementations, Logics, and Programs, Aachen, Germany, Springer-Verlag 1996
- [Stoye 86] W. Stoye: *Message-Based Functional Operating Systems*, Science of Computer Programming 6, pp 291-311, North-Holland 1996
- [Turner 87] D. Turner: *Functional Programming and Communicating Processes*, Conference on Parallel Languages and Architectures Europe, Springer-Verlag 1987
- [Wadler 92] P. Wadler: *The Essence of Functional Programming*, 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1992