PFL+: A Kernel Scheme for Functions I/O

Andrew Gordon

February 1989
PFL+ : A Kernel Scheme for Functional I/O

Andrew Gordon
University of Cambridge Computer Laboratory,
New Museums Site, Pembroke Street,
Cambridge, CB2 3QG,
United Kingdom.

February 7, 1989

Abstract

In place of the common separation of functional I/O into continuation and stream based schemes, an alternative division between Data Driven and Strictness Driven mechanisms for I/O is proposed. The data driven mechanism determines I/O actions by the Weak Head Normal Forms of programs, while strictness driven I/O is based on suspensions—i.e., I/O actions are triggered when demand arises for the value of a suspension during normal order reduction. The data driven and strictness driven I/O mechanisms are exemplified by the output list and input list, respectively, in Landin's stream based I/O scheme.

PFL+ is a functional I/O scheme, able to express arbitrary I/O actions and both data driven and strictness driven constructs in terms of a small kernel of primitives. PFL+ could be added to any functional language. It is based on Holmström's PFL [5], a parallel functional language with embedded communication and concurrency operators from CCS. PFL+ adds non-strict communication, behaviours with results, and primitives to make suspensions.

Examples are given of how PFL+ can derive from these primitives both stream based I/O and the representation of the file system as a function.

1 Aims and assumptions

Functional I/O need not be ad hoc. The contention of this report is that I/O mechanisms for purely functional languages (e.g., Miranda, Ponder, Haskell) can be expressed by a small kernel of orthogonal primitives.

First some informal definitions. As is conventional [14], reduction of an expression is the process of performing leftmost, outermost (normal order) reductions until a Weak Head Normal Form (WHNF\(^1\)) is reached. Only closed expressions are ever reduced. The denotation of an expression is the value obtained under the non-strict denotational semantics of the lambda calculus [17].

\(^1\)In this context, an expression \(e\) is in WHNF iff \(e\) is a lambda abstraction \(x \mapsto e'\), or \(e\) is of the form \(f\ e_1\ e_2\ \ldots\ e_n\), \(n \geq 0\), and either \(f\) is a data constructor, or \(f\) is a primitive function, where \(f\ e_1\ \ldots\ e_m\) is not reducible for \(m \leq n\).
For a process to communicate externally with other processes and the world outside the computer, any operating system offers a set of I/O actions, such as reading and writing characters, files, network connections, etc. The execution of a program (in any language) by a computer is a pattern of internal computations and external communications—I/O actions. The execution rule defines how a program is executed; it maps a program to a series of internal computations and external I/O actions. What can be observed externally of a program's execution, i.e., the pattern of I/O actions, is the I/O behaviour of the program.

In a functional language, a program is just an expression. An I/O scheme specifies a type for programs, and an execution rule. All programs executed under an I/O scheme A are of the same type A; say that I/O scheme A is of type A. For instance, Landin [9] proposed an I/O scheme of type char 1ist → char 1ist; each program f accepts a stream of input characters as argument, and the result of the application is a stream of characters to be output.

An execution rule defines execution of a functional program—the iterative process of first reducing a program e to WHNF; then, depending on the WHNF, either executing subexpressions of the WHNF or halting. I/O actions can occur either during reduction or after a WHNF is reached.

Two I/O schemes can be related by determining whether one can express the other. Specifically, an I/O scheme A of type A emulates a scheme B of type B iff there is a continuous function f: B → A such that for any B program b: B, the A program f(b) has the same I/O behaviour as b.

Starting with Landin streams many I/O schemes have been proposed for purely functional languages, but they tend to be ad hoc in the sense of being designed to express particular kinds of I/O, without aiming for universal power. An I/O scheme can be universal in two senses, if it:

(U1) expresses any I/O action available to an operating system process;

(U2) emulates any other functional I/O scheme.

The claim of this work is that functional I/O is best understood by decomposing it into a small set of primitives, satisfying both U1 and U2, that preserves referential transparency and is orthogonal—no primitive can be expressed in terms of the others.

This report is an informal description to PFL+, a kernel I/O scheme derived from Holmström's language PFL [5]. PFL is an embedding of CCS [10,11] communication and concurrency primitives in a strict dialect of ML.

The design of PFL+ is based on an analysis (Section 2) of functional I/O that reduces it to two independent mechanisms of strictness driven and data driven I/O. Most I/O schemes are mixtures of the two; I have found no references to purely functional I/O schemes that do not fit this analysis.

A non-strict dialect of PFL (Section 3) is taken as the basis of a universal kernel, because of its powerful set of primitives for communication and concurrency.

PFL can emulate data driven I/O but not strictness driven I/O, so PFL+ (Section 4) is derived by adding new primitives. Section 4 contains examples showing how PFL+ emulates several strictness driven schemes.

It must be admitted that there is yet no proof that PFL+ is a universal kernel satisfying U1 and U2; it is however able to express any I/O scheme founded on data driven and
strictness driven I/O, and arbitrary operating system I/O can be integrated with PFL+ communication.

1.1 Notation

Program fragments are given in a variant of Standard ML syntax, but non-strict semantics is assumed. Lambda abstractions are non-strict and are written as \( p \mapsto e \) where \( e \) is an expression and \( p \) is a pattern, here either an identifier \( x \) or a pair \((x, y)\). The constructors of datatypes (that is, algebraic types or free data types) have non-strict semantics; in particular, lists have non-strict cons ("::"). In place of Standard ML's primitive string type, there is a new primitive type \texttt{char} of characters, with the type \texttt{String} as a synonym for \texttt{char list}.

Denotational equality is written as \( =; \); in assertions of denotational equality, program fragments stand for their denotation (without quasiquotation). For example, \( e = \bot \) asserts that the denotation of \( e \) is \( \bot \), where \( e \) has been defined in a program fragment.

1.2 Referential Transparency

The root of the claim that functional programming makes reasoning about programs easier is that functional languages enjoy a collection of properties informally known as "referential transparency" [16]. In particular, two informal properties are used in this report.

The first is unfoldability of a function application; does \((x \mapsto e_1) e_2\) have the same value as \(e_1\) with each occurrence of \(x\) replaced by \(e_2\), that is, does beta reduction hold? A good test case is whether \((x:\texttt{int} \mapsto x\cdot x)\ e\) has the same value as \(e\cdot e\).

The second is whether reduction is functional; whether the operational behaviour of each object in the language can be explained as a function in the denotational semantics.

Both of these are properties of expression reduction. Normal order reduction of a purely functional language enjoys both. If an execution rule extends the reduction mechanism to cause I/O, these properties must be preserved.

As a notion of referential transparency for I/O schemes, define an I/O scheme to be conformant iff whenever two expressions denote the same value, they have the same I/O behaviour. The motivation for this definition is that formal reasoning techniques concerning the functional meaning—the denotation—of functional programs carry over to I/O behaviours, provided the I/O scheme is conformant.

2 Basic I/O techniques

A common division of functional I/O is between stream and continuation based I/O schemes. For example, Haskell [6] I/O is based on streams, but the standard prelude includes emulation of continuation based I/O. While this division conveniently categorises the programmer's view of I/O, the alternative analysis adopted here distinguishes between two execution mechanisms for I/O. One is strictness driven, used in stream I/O to represent the input stream, and the other is data driven, used in continuation schemes, but also in stream schemes for the output stream. Both mechanisms are present together in most I/O schemes—including Haskell stream based I/O—and separate measures are needed in PFL+ to express the two.
2.1 The Data Driven Mechanism

As a simple example of data driven I/O, consider a scheme, based on Landin streams but omitting the input stream, in which programs return a stream of characters to be output. For example, when executed, the program

\[ ['A', 'n', 'n', 'e'] : String \]

outputs the character 'A', then 'n', 'n', 'e' and halts.

To discover the type and rule of this I/O scheme, recall that streams (\[ String = char list \]) are members of the datatype:

\[
\text{datatype } String = :: \text{ of } char \times String \\
| \text{nil}
\]

where infix :: (cons) is a right associative constructor.

Without the usual syntactic sugar, the program is written:

\[ 'A' :: ('n' :: ('n' :: ('e' :: nil))) : String \]

The type of the scheme is String and the execution rule is this: to execute a program E: String, reduce E to WHNF giving either c::k or nil. In the latter case halt, otherwise reduce c and output the resulting character; then recursively execute the program k.

This output scheme is a simple case of Data Driven I/O, whose distinctive feature is a set of instructions represented as constructors of a datatype (here :: and nil); the arguments to the constructors either qualify the instructions (here c in c::k) or provide sequencing information as embedded continuations (here k in c::k). A continuation represents the rest of the computation after the current instruction; not all continuations need have the same type, but generally each has a type such as t, or bool→t, or String→t, etc., where t is the datatype.

A data driven program can be seen as a piece of abstract syntax in a language whose grammar is defined by the datatype; the execution rule is the dynamic semantics of the language.

A data driven scheme can ask for input by embedding continuations that accept arguments. Take the datatype of the previous scheme, rename :: as write, nil as halt, and add a new constructor read to obtain dd1, the type of a data driven I/O scheme able to do simple character based I/O:

\[
\text{datatype } dd1 = \text{write of } char \times dd1 \\
| \text{read of } char \rightarrow dd1 \\
| \text{halt}
\]

The execution rule is an extension of the previous one: to execute a program E: dd1, reduce E to WHNF. If the answer is halt then halt at once. If the answer is write(cw,kw), reduce and output cw, then recursively execute kw. Otherwise, if the answer is read(kr) then input a character cr, and execute kr(cr).

For example, the following program copies the first character on its input twice to its output, then halts:

\[
\text{val double0: dd1 = read(x \mapsto write(x, write(x, halt)))}
\]
It is important to realise that execution of `double0` is a series of reductions to WHNF, interleaved with I/O actions; in data driven I/O, the values of expressions cause particular I/O actions to occur, but finding the value of an expression during reduction never causes I/O. Imperative I/O actions are kept out of functional reduction.

The normal order reduction mechanism of a language is unaltered if data driven I/O is added, so referential transparency is preserved. The two example schemes are conformant because they depend only on the values of expressions.

### 2.2 Operators with side effects

Applicative order functional languages such as Standard ML or Scheme do I/O by introducing operators with side-effects. As a simplified example, consider Standard ML, with strict semantics, extended by two I/O operators `read`: `unit → char` and `write`: `char → unit`. An equivalent to `double0` above is

```ml
val double1 = let x = read() in write x; write x end
```

Standard ML’s strict semantics entail the reduction of `read()`, then `write x`, twice.

In contrast to continuation based I/O, these operators `read` and `write` violate unfoldability. For instance, executing `(x ⇒ x-x)(read())` causes one `read`, yet `read()-read()` causes two. These operators are not functions.

There are strong pragmatic arguments in favour of this style of I/O for applicative order functional languages, but it is unsuitable for purely functional languages because of the difficulty of mixing side effects with normal order reduction, and of reasoning in the absence of referential transparency.

### 2.3 The Strictness Driven Mechanism

Instead of just returning a stream as output, a Landin stream program takes an argument stream to represent the series of characters that are typed as input. The type of programs is `String → String`. Since the entirety of the input list is not available before the start of program execution, it is initially represented by a suspension, `s1`.

The execution rule for output is the same as before. For input, if demand arises during reduction for the value of a suspension `s1`, reduction pauses until the next input character `c` is obtained, then resumes with the suspension replaced by the value `c :: s2`, where `s2` is a suspension representing the rest of the input. For simplicity, there is no end of input—the input stream is assumed infinite.

For example, the following program of type `String → String` has the same I/O behaviour as `double0` and `double1`,

```ml
fun double2 (x::xs) = x::x::nil
```

When `double2(s1)` is executed, the output rule directs that `x`, bound to the head of `s1`, is reduced. According to the input rule, reduction pauses until the first input character `c` appears, then resumes with `s1` replaced by `c::s2`, yielding `c::c::nil`, causing `c` to be output twice.

**Strictness Driven I/O** is based on suspensions; examples include the many variants on Landin input streams and representations of the file system as a function—as in Ponder [19].
Execution of a suspension is a series of I/O actions followed by returning a result to be the value of the suspension. If an I/O scheme includes a convention for returning a result from a program, executing a suspension is just program execution—known as forcing the suspension. Before being forced a suspension is said to be fresh. The reduction mechanism forces a suspension iff its value is needed to return the WHNF of an expression. Operationally, replacing the suspension by its value is achieved by destructive update, as in lazy evaluation—once a suspension is forced, its value is fixed forever.

Strictness is the denotational analogue of the operational notion of demand for a suspension—hence the "strictness driven mechanism." If an expression $e$ is reduced, and $e$ is denotationally equivalent to $f(s)$ where $s$ is a fresh suspension, then $s$ is forced only if $f$ is strict. After the value $v$ of $s$ is obtained, computation proceeds by reducing $f(v)$. Although the reduction mechanism is an extension of normal order reduction, it can still be defined functionally.

The I/O behaviour of suspension based programs is based on analysis of their strictness, so care must be taken that implementation do not alter the termination properties of programs. This restricts optimisation by automatic program transformers if termination properties are changed [2], or parallel reduction strategies using call-by-speculation. In the latter case, a parallel implementation may have a normal order thread of control, plus several speculative threads evaluating expressions whose values are hoped to be needed later by the normal order thread. Such a parallel implementation can use suspension based I/O if only the normal order thread forces suspensions; speculative threads must die if they discover a fresh suspension—or else the suspension may be forced too early.

Introducing suspensions extends the reduction mechanism, so referential transparency might be violated. Suspensions start off undefined, but when forced they are instantiated to a particular value that they retain forever. Unfoldability is retained; for instance, if $s1 :: \text{int}$ is a fresh suspension, $(x \mapsto x-x) \ s1$ means the same as $s1 \cdot s1$. In each case, the suspension is forced once, and the second reference to it obtains the original value.

The operational forcing of suspensions is modelled denotationally in terms of strictness, but there is limited nondeterminism—witness the expression $s1 \cdot s2 :: \text{int}$, where $s1$ and $s2$ are fresh suspensions. This expression is strict in both suspensions, so reduction can force either $s1$ or $s2$ first and still match the functional semantics. If the order of forcing suspensions affects the value of the expression, nondeterminism is introduced. This nondeterminism can affect conformance. Work is in progress on a denotational characterisation of this nondeterminism.

2.4 Comparison

All functional I/O appears to be founded on combinations of the data driven and strictness driven mechanisms. Strictness driven I/O actions occur during reduction, whereas data driven I/O actions only occur after a WHNF is reached.

I/O schemes are often divided into stream based and continuation based schemes. Landin streams (String $\rightarrow$ String) and the dd1 scheme are typical examples of stream based and continuation based I/O schemes, respectively. Continuation based I/O uses only the data driven mechanism, but stream I/O has data driven output and strictness driven input.

A rigorous analysis of the relative powers of these two mechanisms needs a formalisation of execution rules; work is in progress on a CCS-style operational semantics of execution
rules. Of particular interest is the question of whether either mechanism subsumes the other. The kernel I/O scheme proposed in this report, PFL+, is based on data driven execution, but has constructors to create suspensions to emulate strictness driven execution. Whether an I/O scheme of equivalent power to PFL+ could be based on just one mechanism is the subject of current study.

3 Holmström’s PFL

PFL (Parallel Functional Language) [5] models the world inside a computer as a collection of computing agents that communicate by synchronised, pairwise, value-passing handshakes on a collection of channels. Holmström’s PFL is an embedding of CCS [10,11] notions of concurrency and communication in a strict functional language—PFL is developed here in a non-strict language, with non-strict communication.

Agents have a repertoire of instructions based on CCS primitives: to read or write a value on a channel; to allocate a new channel; to fork into two agents executing in parallel; and to offer communication on several channels simultaneously, committing to at most one.

As presented here, PFL is a data driven I/O scheme embedded in a non-strict dialect of ML; programs denote behaviours, values of the datatype beh. A behaviour specifies the instructions that an agent is to perform.

All communication between agents is mediated by unidirectional channels. Each channel appears as two values in PFL+: one of type $\alpha$ outChan is the “writing side” of the channel, and the other of type $\alpha$ inChan is the “reading side,” for a particular type $\alpha$. When an agent allocates a new channel, it obtains the two sides of the channel as a pair:

\[
\text{type } \alpha \text{ Chan } = \alpha \text{ inChan } \times \alpha \text{ outChan}
\]

In Holmström’s PFL, a channel appears as just one value (of type analogous to $\alpha$ Chan) used by both read and write operations. Most functions make use of just one side of a channel, so decomposing Chan into inChan and outChan in this presentation lets the type checker catch the error of reading or writing the wrong side of a channel.

By convention an identifier standing for an inChan starts with a lower case letter, while one standing for an outChan begins in upper case.

3.1 Behaviours

The behaviour type has six constructors:

\[
\text{datatype beh } = \text{NIL}
\]

\[
| \mkChan \text{ of } \alpha \text{ Chan } \to \text{ beh}
| \text{ ?? } \text{ of } \alpha \text{ inChan } \times (\alpha \to \text{ beh})
| \text{ !! } \text{ of } \alpha \text{ outChan } \times \alpha \times \text{ beh}
| \| \text{ of } \text{ beh } \times \text{ beh}
| + \text{ of } \text{ beh } \times \text{ beh};
\]

The simplest constructor is NIL: beh; an agent executing NIL performs no action.

The mkChan constructor denotes a behaviour that allocates a new channel then passes it to the continuation. For example, if an agent reduces a program to the expression,

\[
\mkChan(k: \text{ int Chan } \to \text{ beh}),
\]
the agent allocates a new channel pair \((c: \text{int}, \text{inChan}, C: \text{int}, \text{outChan})\), then executes the continuation \(k\) applied to \((c, C)\), that is the program \(k(c, C)\).

### 3.2 Communication constructors

The behaviour \(??(c, x \mapsto P)\), an application of the read constructor ?? to a channel \(c\) and a continuation \(x \mapsto P\), directs an agent to offer to read from channel \(c\). The agent waits until a parallel agent offers to write a value \(v\) on \(c\), accepts \(v\) and proceeds to execute \((x \mapsto P)\) \(v\).

Similarly, the behaviour \(!!(c, e, P)\), formed with the write constructor !!, directs an agent to offer to write \(e\) on the channel \(c\). The agent waits until a parallel agent offers to read on \(c\), transfers \(e\), and proceeds to execute \(P\).

The act of reading or writing is a communication, a synchronised handshake. If more than one agent offers to write a value on \(c\), a read from \(c\) obtains only one and the other writing agents must wait for further reads, and vice versa.

Following Holmström, it is convenient to introduce new syntax for these constructors; write \(c?x.P\) for \(??(c, x \mapsto P)\) and \(C!e.P\) for \(!!(C, e, P)\).

Holmström's PFL is embedded in a strict dialect of ML, but here the dialect is assumed to have non-strict semantics, so a choice arises concerning the strictness of value passing. When a value is written on a channel, must it first be reduced to WHNF or is it written unreduced? For instance, if \(\bot\) is the undefined, divergent value, is an agent executing \(C!\bot.P\) able to communicate the value \(\bot\) to another agent then execute \(P\), or does it try to compute \(\bot\) and so behave equivalently to \(\text{NIL}\)?

There are two distinct phenomena here: synchronised communication between agents and reduction to WHNF. Rather than combine both and have strict communication, it is better to be orthogonal by making them separate, i.e., make communication non-strict, and have another way to call for reduction to WHNF. A simple way to do the latter is to introduce a new beh constructor,

\[
\text{val reduce: } \alpha \times \text{beh} \rightarrow \text{beh}
\]

The rule to execute \(\text{reduce}(e, P)\) is: reduce \(e\) to WHNF, then execute \(P\). If \(e=\bot\), reducing \(e\) to WHNF never halts, so \(\text{reduce}(e, P)\) has behaviour equivalent to \(\text{NIL}\). A value \(e\) of any type \(\alpha\) can be reduced, provided that in the denotational semantics, \(e \neq \bot\) iff \(e\) has a WHNF.

### 3.3 Parallelism and choice constructors

Behaviours can be composed in parallel by the infix constructor ||. For example, an agent executing the behaviour

\[
\text{mkChan}((c, C) \mapsto (C!42.\text{NIL} || c?x.(k x)))
\]

allocates a new channel, binds it to \(c\) and \(C\), then splits into two parallel agents. One offers to write 42 on \(C\), while the other offers to read from \(c\). Since two agents are concurrently offering to read and write on the same channel, a handshake occurs, and 42 is bound to \(x\). The first of the agents terminates in \(\text{NIL}\), while the other proceeds to execute the continuation \(k\) applied to 42.
The behaviour $P+Q$, formed by the infix choice constructor $+$, can either perform a communication of $P$, then act as $P$, or perform a communication of $Q$ and act as $Q$, but not both. For example, an agent executing the value of $C1x.P + D1y.Q$ offers to write concurrently on both $C$ and $D$. If another agent handshakes on $C$, then $P$ is executed, and the offer on $D$ is withdrawn; conversely, if another agent handshakes on $D$, then $Q$ is executed and the offer on $C$ withdrawn.

For implementation efficiency, it is required that the behaviour arguments to $+$ be either the result of another $+$, or a read or a write. If this constraint is relaxed, implementing the choice between two behaviours can involve starting then killing arbitrarily many processes, e.g.,

$$(P1 || \ldots || Pn) + (Q1 || \ldots || Qm)$$

This constraint is expressed in the type system of Holmström’s PFL, but is omitted from this presentation for simplicity. It appears that the constraint is not a problem for practical programming [12].

### 3.4 Formulation of behaviours

The PFL beh type is given here as a datatype, in contrast to Holmström’s original formulation as an abstract type with a set of constructor functions, but no selector or destructor functions (which are implicitly available for any datatype by pattern matching). He has two reasons for using an abstract type:

(i) the absence of selectors and destructors stresses the extensional nature of behaviours—no program need ever examine the (intensional) structure of behaviours;

(ii) the datatype beh is illegal in ML because the type variable $\alpha$ occurs in the types of three of the constructors but not as a parameter of beh.

The type beh is presented here as a datatype to stress that a datatype underlies each data driven I/O scheme—the datatype makes a natural implementation of Holmström’s abstract type and acts as an abstract syntax.

The problem with a non-parameter variable in a constructor is that in the Hindley-Milner type system all type variables are universally quantified. The type system can be broken if a type variable appearing in the type of a constructor is not a parameter of the datatype and is treated as if it were universally quantified. Rather than being universally quantified, non-parameter variables in type constructors should be existentially quantified [4]. This is supported in Hope+ [Personal communication from Nigel Perry, 1989].

For clarity in the rest of this report, non-parameter type variables in datatypes are explicitly existentially quantified; although few existing functional languages support existential types, such datatypes can be recast as abstract types for practical implementation.

### 4 Towards PFL+

PFL is an I/O scheme of type beh; PFL+ is an I/O scheme of type $\alpha$ Beh. This section develops $\alpha$ Beh from beh, and shows how PFL+ can emulate both data driven and strictness driven I/O.
4.1 Behaviours with results

An agent executing a behaviour often finishes by returning a result to another agent. In PFL this might be achieved by writing the result on a channel. For syntactic convenience in PFL+, behaviours with results are represented by a new datatype \( \alpha \) Beh. A program of type \( \alpha \) Beh is a behaviour that can return a result of type \( \alpha \); the new datatype has constructors corresponding to all those of beh, plus new ones to deal with results. The core is:

\[
\text{datatype } \alpha \text{ Beh } = \text{NIL} \\
| \text{mkChan of } \exists \beta. \beta \text{ Chan } \rightarrow \alpha \text{ Beh} \\
| \text{?? of } \exists \beta. \beta \text{ inChan } \times (\beta \rightarrow \alpha \text{ Beh}) \\
| \text{!! of } \exists \beta. \beta \text{ outChan } \times \beta \times \alpha \text{ Beh} \\
| \parallel \text{ of } \alpha \text{ Beh } \times \alpha \text{ Beh} \\
| + \text{ of } \alpha \text{ Beh } \times \alpha \text{ Beh} \\
| \text{reduce of } \exists \beta. \beta \times \alpha \text{ Beh} \\
| \text{Ret of } \alpha \\
| \triangleright \text{ of } \exists \beta. \beta \text{ Beh } \times (\beta \rightarrow \alpha \text{ Beh})
\]

The new constructor, Ret, defines the result of a behaviour. An agent executing (\text{Ret 42}): \text{int Beh} returns the value 42 as its result. Ret is equivalent to a write to an implicit result channel that is carried along by the agents executing a particular behaviour. The write returning the result is non-strict like any other communication. The implicit result channel is read once only to determine the result of the behaviour, if a behaviour gives rise to multiple parallel agents, only one Ret succeeds. For example, the result of executing Ret false || Ret true is either true or false, depending on execution order. Note that parallel agents are not terminated by one of them executing Ret, though only one Ret can be executed.

A behaviour can be “called” using the sequential composition constructor \( \triangleright \). The behaviour \( P \triangleright x \mapsto Q \) (parsed \( P \triangleright (x \mapsto Q) \)), executes as \( P \), but after Ret is executed, the result is bound to \( x \) and \( Q \) started in parallel. For example, an agent given Ret 4 \( \triangleright x \mapsto \text{Ret}(x+3) \) first executes Ret 4 to give result 4 which is bound to \( x \) for the execution of Ret(x+3).

Since Ret is non-strict, the behaviour Ret \( \perp \triangleright x \mapsto \text{Ret} 3 \) causes Ret \( \perp \) to be executed, returning the value \( \perp \), then Ret 3 to be executed returning 3 as the result of the whole behaviour. If Ret were strict, Ret \( \perp \) would be equivalent to \( \text{NIL} \), so the whole behaviour would be equivalent to \( \text{NIL} \), and would never return a result.

A strict version of Ret is useful if a result is to be reduced before being returned:

\[
\text{fun RetS x = reduce}(x, \text{Ret} x)
\]

RetS \( x \) has the same behaviour as Ret \( x \) unless \( x = \perp \), when the former acts as \( \text{NIL} \).

A PFL+ procedure is a function from an argument to a behaviour delivering a result:

\[
\text{type } (\alpha, \beta) \text{ Proc } = \alpha \rightarrow \beta \text{ Beh}
\]

If \text{getFile}: \((\text{String, String}) \text{ Proc}) \) is a procedure to read files from a file system, the expression \text{getFile} "Anne" \( \triangleright x \mapsto P \) is a procedure call of get, binding the contents of file "Anne" to \( x \) for the execution of \( P \).
4.2 Linking lists and channels

As Holmström shows in his paper, PFL can express concepts like semaphores and locks, and hence enables the expression of explicit concurrency from a purely functional language. The purpose in this section is to examine how Landin’s I/O scheme can be emulated.

To emulate Landin streams in PFL+, two functions are needed:

val ListToChan : α outChan → α list → unit Beh
val ChanToList : α inChan → (α list → β Beh) → β Beh

The intention is that ListToChan C xs is a behaviour that writes each member of the list xs in order on channel C, and terminates by returning (): unit if xs is finite; and that ChanToList(c, ys → P) is a behaviour equivalent to P with ys bound to (a suspension of) the list of values that are written to c.

Given these two functions it is possible to define LSify, a function that takes a channel for input, i, and a channel for output, O, and a Landin stream function f:

val LSify : α inChan → β outChan → (α list → β list) → unit Beh
fun LSify i O f = ChanToList i (xs ⇝ ListToChan O (f xs))

The list xs is bound to the series of values that appear on the channel i; and the members of the list f xs are written in order on the channel O.

The problem now is to define ListToChan and ChanToList using the PFL+ primitives. The former is straightforward:

fun ListToChan C (x::xs) = C!(ListToChan C xs)
| ListToChan C [] = RET ()

Can ChanToList be programmed using the PFL primitives? No, because I/O actions occur during expression reduction in strictness driven I/O, but PFL is entirely data driven, so actions can only occur between reductions of expressions.

4.3 Suspensions

To express strictness driven schemes such as Landin input streams, a constructor is needed to create suspensions. The program defining a suspension is naturally just a PFL+ behaviour, α Beh; once the suspension is forced the behaviour is executed and its result replaces the suspension.

The new constructor to create suspensions is suspend:

val suspend: α Beh × (α → β Beh) → β Beh

The behaviour of suspend(P, x ⇝ Q) is to make P into a suspension, bind it to x, then execute Q. Only if Q makes strict use of x is the suspension forced, i.e., P executed. The result of P then becomes the value of the suspension so that all uses of x obtain the same value. The behaviour P is executed at most once.

The types of the constructors suspend and are the same so it is worth stressing the difference between their semantics. Consider an agent executing P > x ⇝ Q; it executes P then binds the result to x before executing Q. Behaviour P is executed exactly once before Q, whether or not Q makes use of x, in contrast to suspend(P, x ⇝ Q).
If \( P: \alpha \text{ Beh} \), then write \( \langle P \rangle : \alpha \) for a fresh suspension of the behaviour \( P \). Expression reduction is normal order reduction, except that if \( \langle P \rangle \) is the next redex in normal order, reduction pauses while \( P \) is executed, and its result replaces \( \langle P \rangle \). A suspension is forced only when it is the next normal order redex. A suspension is a unique object; if it is copied and then forced, all copies are forced together. The notation \( \langle P \rangle \) is not part of the functional language; it denotes a fresh suspension allocated by suspend.

The functional semantics of how suspensions affect reduction is based on the following idea. Suppose an expression \( e_1 \) containing a single suspension can be written as \( f(\langle P \rangle) \). If \( f(\bot) = e_2 \neq \bot \) then \( e_1 \) reduces to \( e_2 \), but if \( f(\bot) = \bot \), i.e., \( f \) is strict, then \( e_1 \) reduces as \( P \triangleright x \mapsto f \ x \).

As an example, the table below shows the execution of the program:

\[
\text{suspend}(i?x.\text{Ret} \ x, y \mapsto \text{RetS} \ (y+1)): \text{int} \ \text{Beh}
\]

The left column shows states of the computation, and the middle column indicates which action occurs at each step; actions can be internal reductions or external communications. Each step is numbered in the right column. This is an informal execution model; work is in progress on a Labelled Transition System semantics in the spirit of CCS.

| suspend \((i?x.\text{Ret} \ x, y \mapsto \text{RetS} \ (y+1))\) | Make susp | 1 |
| RetS \((i?x.\text{Ret} \ x)+1\) | Force susp | 2 |
| \(i?x.\text{Ret} \ x\) | \(i?43\) | 2.1 |
| Ret 43 | Resume | 2.2 |
| RetS \((43+1)\) | \(\rightarrow\) | 3 |
| Ret 44 | Terminate | 4 |

Recall that execution is of the form: reduce to WHNF, then obey the execution rule for the constructor. At step 1, the expression \( \text{suspend}(\ldots) \) is in WHNF, so the execution rule is to make a suspension out of \( i?x.\text{Ret} \ x \) and then execute \( y \mapsto \text{RetS}(y+1) \) applied to the new suspension. The reduction in step 2 pauses before WHNF, because the value of the suspension is needed. Steps 2.1 and 2.2 show the forcing of the suspension. In step 2.1, it is assumed an external agent has written the value 43 on channel i. Step 3 shows the suspension replaced by 43; the \( \rightarrow \) shows that step 4 is obtained from step 3 by a normal order reduction. Execution of the program terminates at step 4 with the result 44.

### 4.4 PFL cannot emulate suspend

Operational intuition suggests that strictness driven suspensions cannot be emulated in a data driven I/O scheme. If \( \text{suspend} \) could be emulated there must be a function \( \text{susp} \), using only the \( \alpha \ \text{Beh} \) constructors defined in Section 4.1,

\[
\text{val susp: } \alpha \ \text{Beh} \times (\alpha \rightarrow \beta \ \text{Beh}) \rightarrow \beta \ \text{Beh}
\]

\[
\text{fun susp}(P,k) = \ldots
\]

such that \( \text{susp}(P,k) \) acts as \( \text{suspend}(P,k) \) for all \( P, k \). What is \( \text{susp} \) to do with \( P \) and \( k \)? It must only execute \( P \) if \( k \) is strict, but there is no computable operation on \( k \) to determine its strictness.

There is an informal domain theoretic argument that such a \( \text{susp} \) is not continuous, i.e., is uncomputable.

Consider two functions, \( f \) and \( g \), of type \( \text{bool} \rightarrow \text{bool} \ \text{Beh} \),

```plaintext
12
```
fun f x = if x then Ret true else Ret true
fun g x = Ret true

Clearly, f is just a strict version of g, so f \sqsubseteq g, where \sqsubseteq is the partial order in the domain of continuous functions in the denotational semantics.

Define try as a function making use of susp:

val try: (bool \to bool Beh) \to bool Beh
fun try f = susp(i?x. Ret x, f)

Let X=i?x. Ret true and Y=Ret true. If susp has the same semantics as suspend, it must be that try f executes as X and try g executes as Y. For susp to be monotonic, there must be behaviour values X'=try f and Y'=try g such that X' \sqsubseteq Y', and X' executes as X, and Y' executes as Y. X can perform the action i?x, so X' must too. But since X' \sqsubseteq Y', the value Y' must contain a constructor for the read i?x and be able to perform i?x, so Y' can not have the same behaviour as Y.

By this contradiction susp cannot be monotonic nor continuous, so suspend cannot be emulated by the existing constructors. This functional argument backs the operational intuition that suspensions cannot be emulated by the data driven mechanism, suggesting that in general the data driven mechanism cannot emulate strictness driven I/O.

4.5 A function to execute behaviours?

An alternative to the suspend constructor, is to extend PFL with a primitive function to make the suspension of a behaviour:

val execute: \alpha Beh \to \alpha

The expression execute(P) returns a fresh suspension (P); strict use of execute(P) causes (P) to be forced. Rather than allowing suspensions to be introduced only by the suspend constructor, execute can introduce suspensions anywhere.

For example, the expression execute(i?x. Ret x) would be reduced by reading a value v from channel i and then reducing v to WHNF. The execute function can define a function ListToChan that given a channel cx, returns the stream of values occuring on cx; with ListOfChan, it is easy to define ChanToList:

val ListOfChan: \alpha inChan \to \alpha list
fun ListOfChan cx = execute(cx?x. Ret(x::ListOfChan cx))
fun ChanToList cx k = k (ListOfChan cx)

Unfortunately execute is not a safe extension to a functional language because it is not unfoldable—witness the expression

(x \mapsto x-x)(execute(i?x. Ret x))

which unfolds to

execute(i?x. Ret x) - execute(i?x. Ret x)
The first executes by reading one value from channel i, then returning 0; while the second reads two values from i, and returns their difference. The first makes one suspension, the second two.

Nor is execute a function; the subexpression \(\text{execute}(i?x. \text{Ret } x)\) can take on two different values in the expression above.

There are two imperative actions on suspensions: allocation and forcing. Forcing is defined functionally in terms of strictness, and allocation must also be defined functionally. Using the data driven style, suspend does so by separating allocation from reduction, but the operator execute would extend reduction to include allocation—and in so doing would violate referential transparency.

### 4.6 Linking lists and channels (resumed)

Given suspend and behaviours with results, ChanToList is defined recursively:

```
fun ChanToList cx k =
  suspend(cx?x. ChanToList cx (xs → Ret(x::xs)), k)
```

Intuitively, an agent given ChanToList \(i \ k\), creates a new suspension, standing for the input stream, then executes

\(k \ (i?x.\text{ChanToList }i \ (xs → Ret(x::xs)))\)

If \(k\) is strict in its argument, the suspension is forced as follows. A value \(c\) is read from \(i\), then the agent executes:

\(\text{ChanToList }i \ (xs → Ret(c::xs))\)

So recursively a second suspension is obtained, and finally

\(\text{Ret}(c::(i?x.\text{ChanToList }i \ (xs → Ret(x::xs))))\)

yields the result of the first suspension. As much of the suspended stream is read from \(i\) as execution requires.

In Section 4.2, LSify is defined in terms of ChanToList and ListToChan. Execution of the program LSify \(i \ 0 \ \text{id}\) is traced in the following table, where \(\text{id}\) is the identity function—the simplest Landin stream program. For brevity ChanToList is abbreviated to C2L, ListToChan to L2C and suspend to susp.

| LSify i 0 id | → | 1 |
| C2L i (xs → L2C 0 xs) | → | 2 |
| susp(i?x.C2L i (xs → Ret(x::xs)), xs → L2C 0 xs) | Make susp | 3 |
| L2C 0 (i?x.C2L i (xs → Ret(x::xs))) | Force susp | 4 |
| i?x.C2L i (xs → Ret(x::xs)) | i?100 | 4.1 |
| C2L i (xs → Ret(100::xs)) | → | 4.2 |
| susp(i?x.C2L i (xs → Ret(x::xs)), xs → Ret(100::xs)) | Make susp | 4.3 |
| Ret(100::(i?x.C2L i (xs → Ret(x::xs)))) | Resume | 4.4 |
| L2C 0 (100::(i?x.C2L i (xs → Ret(x::xs)))) | → | 5 |
| 0!100.(L2C 0 (i?x.C2L i (xs → Ret(x::xs)))) | 0!100 | 6 |
| L2C 0 (i?x.C2L i (xs → Ret(x::xs))) | → | 7 |
Steps 1 and 2 show the reduction to the susp constructor. In step 3 the suspension (of the input stream) is created and passed to L2C 0. In step 4 L2C pattern matching tests whether or not the suspension is [], forcing it—steps 4.1 to 4.4. A value, assumed to be 100, is read from 1 and consed onto a fresh suspension representing the rest of the input to be the result of the suspension. The reduction started in 4 reaches WHNF in step 6, a write of 100 to 0. Execution proceeds at step 7, exactly the same as 4, showing that steps 4 to 6 repeat indefinitely, copying each value from 1 onto 0.

The example of Landin streams shows how PFL+ can express an I/O scheme in terms of simple primitives for communication (!,?) and suspensions (suspend).

4.7 Representing I/O devices

To satisfy U1—to be an I/O scheme able to express any I/O action available to an operating system process—PFL+ must provide an interface to operating system I/O abstractions, such as input and output streams, the file system, asynchronous interrupts or signals, network interfaces, and so on.

4.7.1 Standard input and output

Communication with the operating system is best mediated by ordinary PFL+ channels. For example, many operating systems, such as UNIX, provide each process with buffered input and output character streams. PFL+ represents such streams by two predefined channels:

val stdin: char inChan
val stdout: char outChan

PFL+ programs can do I/O by reading from stdin and writing to stdout. The semantics of the interface is described in terms of special, predefined agents that can examine the state of the operating system’s buffers, and communicate on the predefined channels. A special agent SI is presumed to be writing characters from the operating system’s input buffer to stdin, while another, SO, reads characters from stdout and adds them to the operating system’s output buffer. If ever there are characters in the input buffer, SI offers them on stdin; and provided the output buffer is not full, SO accepts output on stdout.

Output to the outside world must be strict; for instance, the operating system expects 8 bits representing a character, not a pointer to an unreduced expression. One could decree that a write to Stdout be strict, but that would make Stdout different from other outChans. Better is for SO to reduce each character it reads from stdout using reduce before adding it to the operating system’s output buffer. So output is strict, but not the handshake on Stdout; an agent executing Stdout!⊥.P can proceed to P after the write, but after reading ⊥, SO diverges, preventing future output by other agents.

Many programs need to poll their input to discover whether any data is available. Polling could be represented by defining a new channel

val stdinPoll: bool inChan

and redefining SI to handshake Booleans on stdinPoll indicating whether the input buffer is non-empty. Such redefinition of the input interface is ad hoc because it makes input from
the operating system on stdin different from reading data from a PFL process that outputs on an ordinary PFL channel.

A better solution is to define an agent that adds a polling capability to any inChan. What is needed is an agent that reads a series of values on a channel a, and writes the series out on channel B together with polling information on Poll. The agent is a one-character lookahead buffer with two states: empty until a character appears on a, and full until the character has been written to B.

```haskell
fun IBuffEmpty(a,B,Poll) = a?x.IBuffFull(x,a,B,Poll) + Poll!false.IBuffEmpty(a,B,Poll)
and IBuffFull(x,a,B,Poll) = B!x.IBuffEmpty(a,B,Poll) + Poll!true.IBuffFull(x,a,B,Poll)
```

There is a subtle problem here in the semantics of choice +. When empty, the buffer has behaviour:

```haskell
a?x.IBuffFull(x,a,B,Poll) + Poll!false.IBuffEmpty(a,B,Poll)
```

The intention is that an empty buffer replies false to queries on Poll until a character can be read on a—the action a?x should have priority over Poll!false. Like CCS's choice operator, PFL's + gives neither action priority over the other. An agent executing IBuffEmpty is unconstrained from indefinitely handshaking Poll!false even if input is offered on a.

The solution is to introduce a new constructor \( \rightarrow \) for priority choice, with semantics that \( P \rightarrow Q \) acts as \( P+Q \), except that \( P \) has priority over \( Q \). Camilleri [8] has recently added such an operator to CCS, and developed its equational theory. IBuffEmpty is rewritten with \( \rightarrow \) replacing \( + \).

Given IBuffEmpty, the following function adds the polling capability to a channel:

```haskell
val pollify: \( \alpha \text{ inChan} \rightarrow (\alpha \text{ inChan} \times \text{ bool inChan} \rightarrow \beta \text{ Beh}) \rightarrow \beta \text{ Beh}
fun pollify(a,k) = mkChan((b,B)\(\rightarrow \)
mkChan((poll,Poll)\(\rightarrow \)
            IBuffEmpty(a,B,Poll) || k (b,poll))))
```

There is now no need to redefine input to give a predefined polling channel. Suppose \( P \) is a behaviour expression that makes use of stdin and stdinPoll (i.e., has free variables stdin and stdinPoll); then pollify(stdin, (stdin,stdinPoll)\(\rightarrow \)P) is a behaviour expression with the same meaning but relying only on the original definition of stdin.

### 4.7.2 Server interfaces

Often an operating system provides services to user processes using the client-server model: user client agents send request messages to the server agent, which does some task then returns an answer to the client agent. Many operating system facilities can be cast in this mould, e.g., file servers, network name servers, time servers and authentication servers.

In conventional operating systems, a server is located at an address which all clients can determine; analogously, a PFL server is an agent that waits for requests on a particular channel in scope to all client agents. As Holmström explains in his PFL paper, the interesting question concerns how the server is to communicate its answer back to the client.
Channel passing

Holmström suggests that the request contains a channel, on which the server writes the answer. Consider the interface to a file server. There are three kinds of request, represented by the datatype FSreq: reading (FSgetFile), writing (FSputFile) and deletion (FSdelFile). Each request contains an α outChan for the reply, where α is its type, e.g., a String for a file value, or a bool to indicate success. The interface is simply the channel FileServer:

\[
\text{datatype } \text{FSreq} = \text{FSgetFile of String } \times (\text{String outChan}) \\
\hspace{1cm} \mid \text{FSputFile of String } \times \text{String } \times (\text{bool outChan}) \\
\hspace{1cm} \mid \text{FSdelFile of String } \times (\text{bool outChan})
\]

\text{val FileServer: } \text{FSreq outChan}

Again, if FileServer is an interface to a file service provided by the operating system, its semantics is given in terms of a predefined agent, FS, that reads requests from FileServer, asks the operating system to act on the request, then returns the result on the channel embedded in the request. Depending on the operating system, FS may or may not accept concurrent requests.

For example, the following procedure of type (String \times \text{String, bool}) \text{Proc} yields a behaviour to copy a file from old to new:

\text{fun copy1(old,new) =}
\hspace{1cm} \text{mkChan}((a1,A1): \text{String Chan } \mapsto \hspace{1cm} (\ast \text{ answer for the read request } \ast) \\
\hspace{2cm} \text{mkChan}((a2,A2): \text{bool Chan } \mapsto \hspace{1cm} (\ast \text{ answer for the write request } \ast) \\
\hspace{3cm} \text{FileServer!FSgetFile(old,A1) . a1?fval .} \\
\hspace{3cm} \text{FileServer!FSputFile(new,fval,A2) . a2?ok .} \\
\hspace{3cm} \text{Ret ok})

This style of interface is not limited to services provided by the operating system. If agents in PFL+ are persistent, then a file server or database can be programmed using PFL+ agents, and the service made available on a channel like FileServer.

Behaviour passing

In the previous scheme, the channel embedded in the request effectively represents a continuation to which the answer of the request is to be passed. Since behaviours are values in PFL+, an alternative to passing channel continuations is to pass behaviours. In the example of a file server, behaviours are passed in place of channels:

\[
\text{datatype } \alpha \text{ FSreq} = \text{FSgetFile of String } \times (\text{String } \to \alpha \text{ Beh}) \\
\hspace{1cm} \mid \text{FSputFile of String } \times \text{String } \times (\text{bool } \to \alpha \text{ Beh}) \\
\hspace{1cm} \mid \text{FSdelFile of String } \times (\text{bool } \to \alpha \text{ Beh})
\]

\text{val FileServer: } \alpha \text{ FSreq outChan}

The copy program rewritten for this interface looks like this:

\text{fun copy2(old,new) =}
\hspace{1cm} \text{FileServer! FSgetFile(old, fval } \mapsto \\
\hspace{2cm} \text{FileServer! FSputFile(new, fval, b } \mapsto \text{ Ret b).NIL} \\
\hspace{1cm} .\text{NIL}
Procedures

The client-server model is reminiscent of procedure calls in imperative languages, and both
the channel passing and behaviour passing solutions have a flavour of saving a return
address and jumping into the (server) procedure. Indeed, in single machine operating
systems such servers are often presented as system calls, while in distributed systems they
appear as remote procedure calls (RPC). So it is natural to represent the interface as a
collection of PFL+ procedures; for instance, the file server is represented by three:

val getFile: (String, String) Proc
val putFile: (String × String, bool) Proc
val delFile: (String, bool) Proc

Given this higher level interface, the copy program is much simpler to read:

fun copy3(old,new) = getFile old \triangleright fval \mapsto putFile (new, fval)

The procedural interface does not need to be a new primitive, but it can be emulated
in terms of either the channel or behaviour passing interface. As an example, here is how
the putFile procedure is defined from the channel passing interface:

fun putFile (fname, fval) =
  mkChan((a,A)\mapsto
    FileServer!FSputFile(fname, fval, A).NIL \parallel a?ok.Ret ok)

In this section three representations of the client-server model have been defined in
PFL+: by explicit channel or behaviour passing or by procedures. All three are equivalent
in the sense that each could define the other, but the third is higher level and so more
suited to applications.

A final decision on the interface awaits experience with an implementation of PFL+,
but the interface will probably be based channel passing, with the procedural interface
available in a library.

4.8 Memoising procedures

Some I/O schemes, e.g., Fairbairn’s Ponder [19], represent reading from the file system as
applications of a function read: String \rightarrow String. The function read is referentially
transparent in the sense that if name1=name2 then read name1 = read name2, i.e., once
read has returned the value of a file, it retains it so that later reads return the same
value. Intuitively, read takes a snapshot of each file it reads, but there is no guarantee that
snapshots of distinct files are consistent.

To express read in PFL+, what is needed is a combinator memoise that turns a pro-
cedure into a function, e.g., from a procedure get: String \rightarrow String Beh produce a
function read: String \rightarrow String,

val memoise : (α \rightarrow β Beh) \rightarrow ((α \rightarrow β) \rightarrow γ Beh) \rightarrow γ Beh

So the behaviour memoise(get, read \mapsto k read) defines a read function from the get
procedure for use by continuation k.

It turns out that memoise can in principle be defined in terms of suspend if the type
α has an equality operator and is recursively enumerable; so a read function is definable
because String has an equality, and the infinite list of all finite strings exists. Though correct, the definition is extremely inefficient and hence is impractical.

Although memoise could be provided as a new primitive in PFL, the alternative is to introduce parameterised suspensions, or nonces. A nonce is a suspension that receives an argument when forced. A nonce is defined by a PFL procedure, \( x \mapsto P \colon \alpha \to \beta \) Beh, with meta notation \( (x \mapsto P) : \alpha \to \beta \) for a fresh nonce. A nonce \((x \mapsto P)\) is forced the first time an application \((x \mapsto P)\ y\) is the next normal order redex, i.e., when strict use is made of the nonce. To force the nonce the behaviour \((x \mapsto P)\ y\) is executed, to yield a result \(z\). This becomes the result of the application, and other copies of the nonce are replaced by the non-strict constant function \(x \mapsto z\).

The new constructor to create parameterised suspensions is nonce:

```plaintext
val nonce: (\alpha \to \beta \) \times ((\alpha \to \beta) \to \gamma \) \to \gamma \)
```

Emulation of suspend by nonce is both possible and practical:

```plaintext
fun suspend(P,k) = nonce(() \mapsto P, f \mapsto k (f))
```

Consider execution of the following program (with channel \(b1\ colon int inChan\),

```plaintext
nonce(c \mapsto c?x.Ret x, f \mapsto RetS(f \cdot b1))
```

The nonce is created and applied to the continuation:

```plaintext
RetS((c \mapsto c?x.Ret x) \cdot b1)
```

Reduction needs the value of the nonce, so it is forced with argument \(b1\):

```plaintext
b1?x.Ret x
```

A value, say \(77\), is read from \(b1\), and returned as the result of the procedure. Reduction continues with the nonce replaced by \(x \mapsto 77\),

```plaintext
RetS((x \mapsto 77) \cdot b1)
```

This reduces to Ret \(77\) terminating execution. Later uses of the nonce in the program obtain the constant function \(x \mapsto 77\) as its value, no matter what argument is supplied.

Parameterised suspensions are named nonces because they are intended to be used just once for a particular purpose, for the nonce. Although nonces are very low level they can express powerful high level combinators such as memoise.

The key idea behind the definition of memoise in terms of nonce is to represent the function produced by memoise (e.g., read) by its graph (i.e., an association list of argument-result pairs). The graph starts off completely undefined, but as the function is used on different arguments it becomes defined for those arguments.

The association list cannot be an ordinary list because it must be able to "grow" as it is used—its value depends on which arguments are supplied to the memoised procedure. Instead each cell of the list is represented as a triple of argument, result, and the rest of the list—the latter being a nonce. Each cell has type \((arg, res)Graph\).²

²This is a recursive type, not part of ML's Hindley-Milner type system, but present in, e.g., Ponder. It can be expressed as an ML type using a dummy constructor:

```plaintext
datatype (arg,res)Graph = LIFT of arg \times res \times (arg \to (arg,res)Graph)
```

19
type (arg,res)Graph = arg × res × (arg → (arg,res)Graph)

Each cell in the graph of a memoised procedure \( k \) is a triple consisting of a pair \((a,r)\) corresponding to the execution of \( k(a) \) returning \( r \)—and a link function—either a constant function returning the next cell, or a fresh nonce. If the nonce is ever applied to an argument \( a \), behaviour \( k(a) \) is executed to obtain a result \( r \), and the nonce is replaced by the constant function \( x \mapsto (a,r,(j)) \), where \( (j) \) is another fresh nonce.

To apply the memoised procedure to \( a \), the graph is searched sequentially for a triple matching \( a \), at each step supplying \( a \) to the link function. If argument \( a \) is not already in the association list, the nonce at the end of the list will be forced and \( k(a) \) executed, returning a triple containing \( a \); if \( a \) is already in the list, the result of the previous execution will be found by the search and returned. The function \( \text{lookup} \) supplies arguments to the graph of a memoised procedure:

```haskell
fun lookup (g1: arg → (arg,res)Graph) (x1: arg) =
  case g1 x1 of (x2,r2,g2) ↦
    if x1=x2 then r2 else lookup g2 x1;
```

In \( \text{memoise} \) below, a function is obtained from procedure \( p \) and passed to continuation \( k \). An association list \( L \) is defined using nonce and a procedure \( \text{app}(p,a) \) that first applies \( p \) to an argument \( a \) giving \( r \), then obtains a fresh nonce \( f \) by a recursive call to \( \text{memoise} \), and finally returns \( (a,r,f) \) as its result.

```haskell
fun memoise (p, k) =
  nonce(a ↦ \text{app}(p,a), L ↦
    k (\text{lookup} L))
where
  \text{app}(p,a) =
    get a ↦ r ↦
      \text{memoise}(p, f ↦
        \text{Ret}(a,r,f))

  (* make fresh assoc. list L *)
  (* make L into a function with lookup *)

  (* call p on arg. a yielding res. r *)
  (* get a fresh nonce *)
  (* return new triple *)
```

For a simple example, assume \( \text{getPrice}: \text{String} → \text{int} \). \( \text{Beh} \) is a procedure querying a beer price database. The following program is a simple functional database query:

```haskell
\text{memoise}(\text{getPrice}, \text{Price} ↦ \text{RetS}(\text{Price "McEwans" + Price "Lorimers"}))
```

The program reduces to a nonce constructor:

```haskell
\text{nonce}(a ↦ \text{app}(\text{getPrice},a), L ↦
  (\text{Price} ↦ \text{RetS}(\text{Price "McEwans" + Price "Lorimers")}) (\text{lookup} L))
```

The nonce creates a fresh association list and binds it to \( L \). For syntactic convenience, the two references to the nonce are combined using a \( \text{where} \) clause:

```haskell
\text{RetS}(\text{Price "McEwans" + Price "Lorimers" where}
  \text{Price} = \text{lookup} (a ↦ \text{app}(\text{getPrice},a)))
```

There are two applications of the nonce, and both are needed to obtain a WHNF. Suppose the left hand application is reduced first, and execution of \( \text{getPrice "McEwans"} \) yields 90 pence; after the first application the expression is:
datatype α Beh = NIL
  | mkChan of ∃β.β Chan → α Beh
  | ?? of ∃β.β inChan × (β → α Beh)
  | !! of ∃β.β outChan × β × α Beh
  | || of α Beh × α Beh
  | + of α Beh × α Beh
  | ++ of α Beh × α Beh
  | reduce of ∃β.β × α Beh
  | ▷ of ∃β.β Beh × (β → α Beh)
  | Ret of α
  | RetS of α
  | suspend of ∃β.β Beh × (β → α Beh)
  | nonce of ∃β.∃γ. (β → γ Beh) × ((β → γ) → α Beh)

Figure 1: Constructors of α Beh in PFL↓

RetS(90 + Price "Lorimers" where
  Price = lookup
  (a ↦ ("McEwans", 90,
    (a ↦ app(getPrice,a)))))

If the price of "Lorimers" is found to be 93, the expression reduces to:

RetS(90 + 93 where
  Price = lookup
  (a ↦ ("McEwans", 90,
    a ↦ ("Lorimers", 93,
      (a ↦ app(getPrice,a)))))

Hence the result of the query is 183. Notice how the association list starts undefined, but
is defined further as reduction proceeds, always terminated by a fresh nonce.

The memoise function works for procedures; it is simple to define an fmemoise function
to memoise functions:

fun fmemoise (f,k) = memoise(x ↦ Ret(f x), f' ↦ k f')

The memoised functions are strict because of the association list search, unlike Hughes' lazy memo-functions [7].

There are many other applications of nonces, e.g., Redelmeier dialogues [15] and single
assignment variables [8].

4.9 Semantics of Beh

The full repertoire of constructors of α Beh is shown in Figure 1. Such a wide range of
constructors is convenient for programming, but the semantics is clarified if α Beh can be
derived from a smaller kernel of constructors. PFL's beh type can do so when extended
with new constructors reduce' and nonce'.
datatype beh = NIL'
   | mkChan' of α Chan → beh
   | ??' of α inChan × (α → beh)
   | !!' of α outChan × α × beh
   | ||' of beh × beh
   | +' of beh × beh
   | ->' of beh × beh
   | reduce' of ∃α.α × beh
   | nonce' of ∃α.∃β.(α → β outChan→beh)×((α→β)→beh);

Figure 2: Constructors of beh in PFL+

val em: α Beh → α outChan → beh
fun em NIL R = NIL'
   | em (mkChan k) R = mkChan(C ↦ em (k C) R)
   | em (??(i,k)) R = i?'x. em (k x) R
   | em (!(0,v,P)) R = 0!'v. em P R
   | em (P || Q) R = (em P R) ||' (em Q R)
   | em (P + Q) R = (em P R) +' (em Q R)
   | em (P -> Q) R = (em P R) ->' (em Q R)
   | em (Ret v) R = R! 'v. NIL'
   | em (P ∨ k) R = mkChan((c,C) ↦ (em P C) ||' (c?'x. em (k x) R))
   | em (reduce(v,P)) R = reduce'(v, em P R)
   | em (nonce(k1,k2)) R = nonce'(v ↦ S ↦ em (k1 v) S,
                                     f ↦ em (k2 f) R)
   | em (RetS v) R = em (reduce(v, Ret v)) R
   | em' (suspend(P,k)) R = em (nonce(() ↦ P, f ↦ k (f()))) R;

Figure 3: Emulation of α Beh by beh
Figure 2 shows the extended version of beh for PFL+; the names of all constructors end in \^ to distinguish them from their counterparts in \( \alpha \) Beh.

The suspensions and nonces created in \( \alpha \) Beh are emulated by a new kind of nonce that expects a result channel and an argument when forced. It is just the same as a nonce in \( \alpha \) Beh except that the result channel is explicit. For example, when executed

\[
\text{nonce}' (x \mapsto R \mapsto P, f \mapsto Q): \text{beh}'
\]

creates a nonce \((x \mapsto R \mapsto P)\), binds it to \( f \) then executes \( Q \). If \( Q \) makes strict use of an application \( v \), then reduction pauses while \( P \) is executed with \( v \) bound to \( x \), and a new result channel bound to \( R \). As soon as a result \( r \) is read from the result channel, execution of \( Q \) continues with \( f \) bound to \( x \mapsto r \).

This is rather an unintuitive operator, but \text{nonce}' and \text{reduce}' are the only new constructors needed to extend PFL into PFL+.

The function \text{em}: \( \alpha \) Beh \( \rightarrow \alpha \) outChan \( \rightarrow \) beh in Figure 3 shows how \( \alpha \) Beh is emulated in beh. A behaviour with a result, \( \alpha \) Beh, is represented as a function \( \alpha \) outChan \( \rightarrow \) beh. The outChan returns the result of the behaviour, and is written to in the emulation of \text{ret} to return the result. In the emulation of \( P \triangleright k \), a new channel \((c,C)\) is allocated, and \text{em P C} executed in parallel with an agent that reads the result \( x \) of \( P \) from \( c \), then executes the emulation of \( k \) \( c \).

The function \text{em} shows how PFL+'s I/O scheme is emulated by a simple extension of PFL's beh scheme (Figure 2). PFL+'s beh is the kernel from which \( \alpha \) Beh, and a great many data driven and strictness driven I/O schemes can be emulated, e.g., Wadge histons [20], and Stoye synchronised streams [18].

5 Related work

Two recent functional languages, Hope+C [13] and Haskell [6], have I/O schemes with a well developed operating system interface, potentially satisfying U1. In the analysis of this report, Hope+C is based on data driven I/O, whereas Haskell I/O is based on a Landin stream style interface, with feedback from the output stream to the input stream in the style of Stoye [18]—a mixture of data driven and strictness driven execution. The aims behind these languages' I/O schemes and PFL+ differ; Hope+C and Haskell are intended as production functional languages able to perform any I/O action, whereas the design of PFL+ neglects the details of a portable operating system interface, to concentrate on finding a small set of primitives able to emulate arbitrary language constructs for I/O (U2). As illustration, in Hope+C and Haskell there are primitives to make input from the operating system available as a stream, while in PFL+ such a stream is defined using a primitive for communication with the operating system (!!), and a primitive for constructing strictness driven streams (suspend).

There is a perhaps confusing difference in terminology between Hope+C and PFL+. Hope+C is based upon a datatype Continuation, corresponding to the type Beh in PFL+. The primitives \text{mkChan}, \text{ret}, etc., of Beh are called "constructors" in PFL+ (because they are constructors of a datatype) but Hope+C primitives such as \text{OpenChannel}, \text{GetChar}, etc., are called "continuations". The confusion is that in PFL+, "continuation" refers to a value embedded in a Beh constructor to be later executed, like \( k \) in \text{mkchan}(k), or \( P \) in \( P \ | | \)
A term almost equivalent in Hope+C is "continuation function," referring to a function from a status value to a Continuation value.

An interesting style of I/O is proposed for the functional database language DL [1]. DL I/O is split into two levels. The top level, based on continuations, contains a set of primitive operators to perform I/O actions, and a set of combining forms. The continuation based methodology is not tied to DL, but can be applied to languages with different I/O schemes, e.g., Hope+C, Nebula and Miranda.

The bottom level implements the top level in terms of non-functional side-effecting *atoms*, similar to ML or Scheme I/O. For example,

```plaintext
read : String → String      read a file
write : String → String → bool write a file
```

DL's reduction mechanism causes side-effects when these atoms are reduced; e.g., the expression `read "Anne"` reduces to the contents of the file named "Anne", with the side-effect of a read from the file system. The non-functional nature of atoms is hidden from the top level (and the end user) by imposing a continuation based discipline that forbids direct use of atoms. For instance, the two atoms above are represented in the top level by the combinators:

```plaintext
fun Read name k = k (read name)
fun Write name value k = k (write name value)
```

The bottom level of DL is an interesting mix of normal order reduction and side effects, but lacks referential transparency; the top level is referentially transparent and potentially satisfies U1, because any I/O action can be incorporated as a continuation based combinator, like `Read` and `Write`. It is not clear whether the upper level satisfies U2.

### 6 Summary and future work

There are two orthogonal mechanisms for functional I/O, data driven and strictness driven I/O. The former expresses ordering of I/O actions in data structure while the latter uses strictness information. The strictness driven mechanism causes I/O actions to occur during reduction, whereas they can occur only once a WHNF is reached in the data driven mechanism.

Holmström's embedding of CCS primitives in ML is extended to yield PFL+, an I/O scheme capable of emulating either class of I/O mechanisms. Relative to the original PFL, PFL+ adds polarised channels, non-strict communication, behaviours with results and the notion of a procedure. PFL can already emulate data driven I/O, and the new PFL+ constructs suspend and nonce introduce suspensions and parameterised suspensions, respectively, to emulate strictness driven I/O. Arbitrary interfaces to the operating system can be represented by communication on predefined PFL+ channels.

PFL+ appears to have universal expressive power in the sense that it can express any operating system interface (U1) or functional I/O construct (U2). It is not clear that any other I/O scheme satisfies U2. More than other I/O schemes, PFL+ emphasises the difference between data driven and strictness driven mechanisms for I/O, unifies I/O and interprocess communication on channels, and from a small orthogonal kernel derives many constructs usually taken as primitive.

This is a preliminary and informal report of work in progress on PFL+. There are three streams of future work:
(i) PFL+ must be finalised. New constructors are needed for timeouts and interrupts.

(ii) PFL+ must be furnished with a formal semantics. The operational semantics can be given as a transition system in the style of CCS. The challenge is to match the operational semantics to a functional denotational semantics.

(iii) PFL+ must be implemented. There is a crude implementation on top of Poly/ML, but a full implementation will probably be done using Ponder.

7 Acknowledgements

I wish to thank my supervisors, Alan Mycroft and Larry Paulson, for their help and encouragement. My thanks go to Peter Breuer and Nigel Perry for discussions of functional I/O. Several other people read drafts of the report, and I am grateful for their comments; thanks to Juanito Camilleri, Joe Dixon, and Jon Fairbairn. This work is supported by an SERC Studentship.

References


