Approaches to Functional I/O

Owen Stephens os1v07@ecs.soton.ac.uk
Supervisor: Julian Rathke jr2@ecs.soton.ac.uk

***Abstract*—Functional programming is becoming an increasingly popular programming paradigm; it offers conceptually simple code, a high level of modularity and composability, and powerful abstraction features. However, a long-standing difficulty associated with functional language design is how to correctly integrate I/O functionality in a clean and idiomatic way; on the surface, it would appear that the key features of functional languages are immiscible with I/O. Different functional languages use a variety of approaches to solve this problem; in this paper we investigate the approaches of three well-known functional languages: Haskell, Clean and OCaml, and discuss their differences and relative advantages.**

I. What is Functional Programming?

Functional programming is a programming paradigm in which the basic building blocks are functions, rather than objects or procedures as found in other commonly-used paradigms. Functions differ from procedures in that the former are mathematical objects, which calculate their result based only on their inputs, whereas the latter may employ side-effects to produce their value. Hughes [1] has discussed the benefits of functional programming — essentially, the ability to better decompose problem solutions.

In this paper, we give an introduction to two key properties of functional languages that are fundamental to the implementation strategies employed: *strictness* and *purity*. We discuss the approaches used by three functional languages, OCaml, Haskell and Clean, and show the key issues raised by their use, by implementing a demonstration program in each language.

A. Strictness

Programming languages are often informally classified as either strict or non-strict. The distinction refers to the evaluation strategy of the language; strict languages use the call-by-value[1] evaluation strategy, under which, function arguments are fully evaluated before being substituted in the function body. Conversely, non-strict languages do not evaluate function arguments before substitution in the function body. The arguments are simply substituted[2] into the function body in place of the formal parameters, regardless of their form. Call-by-value corresponds to the applicative order (or leftmost-innermost) reduction strategy of the lambda calculus [2], where the "most nested" *reducible-expression* (redex) is reduced first. Call-by-name corresponds to the normal order (leftmost-outermost) reduction strategy, where the least-nested, leftmost redex is reduced first. Sestoft [3] demonstrates the differences between the different forms of reduction.

Wadler [4] discusses some of the relative pros and cons of non-strict and strict languages. One particular advantage of strict evaluation is that the asymptotic complexity is generally easier to reason about [5]. Plotkin [6] showed that it is possible to simulate call-by-value using call-by-name and vice versa. Wadler et al. [7] discuss some of the potential difficulties in adding laziness to a strict language. Sheard [8] proposes a strict-by-default language that uses explicit, laziness annotations provided by the programmer. An operational optimisation for call-by-name languages is the use of lazy evaluation, known as the call-by-need evaluation strategy. Call-by-need prevents identical sub-expressions being evaluated more than once, *memoizing* the sub-expression's value when it is first evaluated. Hughes [1] discusses the main benefits of lazy evaluation, particularly, the "glue" that lazy functional languages provide to enable decomposition of large problems.

B. Purity

Informally, pure functional languages are those that disallow the unconstrained usage of *side-effects*. Side-effects are any observable interaction that occurs as a result of evaluating an expression, to obtain its value. Examples of side-effects include: I/O, mutable state and concurrency. Side-effects are commonly used in popular strict languages such as OCaml and Scheme to provide I/O functionality. A call to a `print` "function" will return some (usually void) value, in addition to modifying the state of the output stream of the program. Side-effects are particularly problematic in lazy languages; since evaluation of sub-expressions is not guaranteed to occur — if the sub-expression is not required, it will not be evaluated — and has no guaranteed ordering, it can be difficult to ensure that side-effects occur in the correct order. Figure 1 demonstrates the difference in behaviour between OCaml, a strict language and Haskell, a lazy language. A more formal definition of purity [9] is that a pure language will be operationally independent from the argument-passing method employed: an implementation of the language using call-by-value will be equivalent, disregarding divergence[3] and errors, to a call-by-name implementation.

Side-effects reduce composability; given an arbitrary function in an impure language, a programmer cannot guarantee the behaviour of the function — when calling the function, there is no guarantee as to what (if any) side-effects are performed — combining side-effecting functions can at best lead to unexpected behaviour and at worst cause a program error.

Side-effects directly affect the *referential transparency* of an expression. Referential transparency refers to the ability to replace a sub-expression with one of equal value, without changing the value of the outer expression. Originating from Quine [10], the term was introduced to Computer Science by Strachey [11]. Referential transparency is a property that is often cited as beneficial, yet as Søndergaard and Sestoft note, the formal and informal definitions given are not always equivalent [12].

To demonstrate referential transparency, consider the following

[1] Sometimes referred to as pass-by-value.

[2] Some care must be taken in order to avoid variable capture (when changing the context of a variable gives it a new meaning) when making the substitution.

[3] A diverging expression is one that doesn't converge to a value, for example the Omega combinator: `(λx.x x)(λx.x x)`.

```
let l = ["1", "2", read_line ()];;
print_string (List.hd l);;
```

(a) OCaml

```
main = do
  let l = [return "1", return "2", getLine]
  head l >>= putStrLn
```

(b) Haskell

Fig. 1: Example programs demonstrating the difference between strict and lazy languages. Both programs create a list containing two strings and a command that reads a string from the standard input stream; the first element of the list is then printed. The OCaml implementation will block, waiting for user input and then print the string "1", since the print statement is evaluated when the list is created. The Haskell implementation will simply print the string "1", since lazy evaluation means that the list elements are not evaluated[4].

statements, made after a hypothetical race has finished:

$$\text{A} = \text{The winner of the race.} \tag{1}$$

$$\text{B believes that A finished second in the race.} \tag{2}$$

substituting (1) into (2) clearly yields a falsehood. (2) is said to be a *referentially opaque* context. Whereas:

$$\text{A beat all the other competitors.} \tag{3}$$

allows substitution of (1) into (3) without yielding a falsehood. (3) is said to be a referentially transparent context.

There are several advantages of purity:

- If a sub-expression's value is not required in any containing expression, it can safely be removed by the compiler; in an impure language, the expression may not provide a value that is used, but it may perform side-effects that affect the computation. An example is the loop-counter increment in a C-style loop — the value of the incrementation is not used, but the side-effect is required to advance the loop index.
- Automatic function-argument memoization (known as a call-by-need evaluation strategy) is possible, since referential transparency is guaranteed; consider the expression `(f y) + (f y)`. In a pure language, it is safe to transform the expression, by assigning the value of `f y` to a variable `x`: `let x = f y in x + x` (replacing the function call with its value), since the absence of side-effects guarantees the equivalence. The opposite is true for a side-effecting expression; the compiler cannot *guarantee* that the transformation does not alter the resultant value.
- Data-independence: pure expressions may be trivially[5] parallelised - since there are no side-effects the relative ordering of expression-evaluation does not affect the result.

[4]The evaluation of an IO action in Haskell does not actually perform the IO, the action must be executed, usually within a do-block, but the demonstration is still valid.

[5]Although, not necessarily *efficiently*.

II. Why is I/O difficult in Purely Functional Programming Languages?

As noted previously, I/O is usually performed in strict languages through the use of side-effects. Since the order of evaluation is known in a strict language, side-effects can more safely be used — the programmer can guarantee that one side-effect will occur before another. Non-strict languages cannot use side-effects, due to the lack of a predetermined evaluation order. Purely functional languages eschew the use of side-effects, due to the difficulty of correctly controlling their use; performing I/O therefore requires a different approach to those used in impure languages.

Peyton Jones motivates the need for a method of enabling I/O in the purely functional programming language Haskell [13]:

> I/O is the raison d'être of every program. — a program that had no observable effect whatsoever (no input, no output) would not be very useful.

Hudak and Sundaresh surveyed the available I/O models available to early Haskell, describing the difficulties that any model must overcome [14] — referential transparency, efficiency and incremental operation[6].

To summarise, the main question to be answered when implementing I/O functionality in lazy functional languages is that of sequencing; *How does the language ensure that I/O operations occur in the correct relative order, as intended by the programmer?*

III. Case studies

We discuss three popular functional programming languages that employ differing approaches to providing I/O. We observe and discuss the pertinent details of the theory and implementation of I/O in the languages, evaluating their relative merits. The three languages to be studied are Haskell [15], Concurrent Clean [16] and OCaml [17].

A. Haskell

Haskell is a lazy, pure functional programming language. Historically, Haskell has used several approaches to implementing I/O; due to its pure and lazy nature, side-effecting I/O functions could not be used. At the time of its design, there were two main approaches available to be used by Haskell [18]: stream transformations and continuations. [14] discusses streams and continuations and the original Haskell approach to I/O. Peyton Jones and Wadler [19] also discuss the approaches to I/O taken by early Haskell. Monads were introduced to Haskell as a method of implementing I/O, and have been fully integrated into the language. Recently, a new form of structuring I/O named Iteratees has become popular in Haskell. Iteratees are not built in to the language, and are not strictly a method of implementing I/O (they use monads to provide I/O operations) but instead are used to manage I/O more effectively, and solve the problems of "lazy I/O" provided by Haskell.

B. Concurrent Clean

Concurrent Clean (or simply, Clean) is also a lazy, pure functional programming language. Its distinguishing feature

[6]Incremental in the sense that the output a program generates should become available as it is generated, rather than when the program exits.

is the use of a *uniqueness* type-system. The uniqueness type-system is able to distinguish expressions that are not allowed to be shared from those that can. This sharing restriction allows the Clean compiler to ensure that I/O functions are executed in a *single-threaded*, deterministically ordered way.

C. OCaml

OCaml [17] is a strict, impure functional programming language. I/O is channel based, and is performed using side-effecting functions.

IV. Approaches

A. Side-Effects

Strict functional languages nearly always use side-effects to implement I/O operations. One exception is the pure language described by Sheard [8] that uses a strict-by-default evaluation strategy, with explicit "laziness" annotations that allow a programmer to specify that non-strict evaluation should take place. Being pure, the language does not employ side-effects, instead using the monadic approach of Haskell[7].

As described earlier, side-effects do not combine well with non-strict languages, due to difficulty of predicting the order in which the side-effects will occur.

Even in strict languages, purity has benefits; the choice to discard purity in order to use side-effecting I/O is probably taken due to the relative ease-of-implementation. Without purity, there are no assurances as to the I/O behaviour of a given function — equational reasoning becomes difficult, at best.

B. Stream-Transformers

Stream transformers map streams of some input type into a stream of some output type. A stream is a lazy list; the tail of the list is not evaluated until it is required. Therefore, a stream can represent an infinite list, and can be used to model I/O interactions of unbounded duration.

The term *stream* was first coined by Landin [20] when describing a correspondence between ALGOL60 for-loop-step evaluations and the Lambda Calculus.

In a lazy language, stream transformers can be represented by a function from lists to lists:

```
type ST in out = [in] -> [out]
```

Non-strictness is critical in the use of stream-transformers: if the list was strictly constructed, the entire list would be evaluated before it was used, which clearly cannot model interleaved I/O — in the case of teletype I/O, the list of user responses would have to be known, before the requests for that input were presented to the user.

Two specialisations of stream transformers used in functional languages are *Landin Streams* and *Synchronised Streams*.

1) Landin Streams

Landin streams are a simple specialisation of stream-transformers, which treat the type of both the input and output streams as Char [21]:

```
type LS = ST Char Char
```

[7]The language is implemented on top of a Haskell interpreter, providing monad-based I/O "for free".

Landin stream I/O was proposed by Landin[20] as a possible method of providing I/O in ALGOL60; output is produced as the values of the output stream are determined, and similarly, inputs occur when the "next" value is demanded from the stream [21]. A program is free to demand many input characters, or append many output characters, without synchronisation between the two streams.

Landin stream I/O is particularly primitive; it does not model all types of I/O that a programmer may wish to perform[8], instead only modelling teletype I/O.

2) Synchronised-Streams

The synchronised-stream mechanism generates a stream of requests and processes the stream of responses, in a one-to-one fashion. Algebraic types are used for both the request and responses, with a data constructor pair for each possible I/O interaction that may be performed:

```
data Req = GetChar
         | PutChar Char
         | ...

data Resp = ReadChar Char
          | Done
          | ...

type SS = ST Resp Req
```

Synchronised streams, also known as Dialogues [22], were the original method of I/O in Haskell [23].

Synchronised stream I/O is an improvement over Landin streams, in that it is able to model all types of I/O that a programmer may wish to perform. In his definition of synchronised streams, Stoye [24] notes that without *requiring* that requests/responses are made/received in pairs, it can be difficult to ensure that the correct synchronisation between input and output is achieved.

As Wadler notes [25], synchronised streams are less modular than monads, it being difficult to compose two functions that both perform I/O. However, synchronised streams do still serve a theoretical purpose, as they can be used to give a denotational semantics to monadic I/O.

Peyton Jones and Wadler [19] point out the difficulties in programming using synchronised streams. They give the following example of an "echo" program that simply copies characters from its input stream to its output stream[9]:

```
echo :: SS
echo responses = GetChar :
        if c == EOF then
            []
        else
            PutChar c :
                echo (drop 2 responses)
    where
       (ReadChar c) = responses !! 0
```

This example shows the fragility of programming using synchronised streams. If three elements were removed from the

[8]For example, using only input/output character lists, there is no way for a program to request that a file is opened for reading.

[9]`x !! n` returns the nth element of list x, and `drop n xs` removes the first n elements from list xs.

response stream, rather than two (the `ReadChar c` and `Done` responses to the `GetChar` and `PutChar c` requests), the program would deadlock, since the recursive call would attempt to evaluate a response element that hadn't been requested. If only one element was removed, the pattern match used to extract the char would fail[10].

C. Continuations

A *continuation* represents the execution state at a given point in the program's execution, that is, a data-structure that *reifies* the control state of a program, including the call-stack and local variables. A history of the origins of continuations is given by Reynolds [26].

If a language supports *first-class* continuations, it provides facilities to capture and invoke continuations within the execution of a program. This allows continuations to be assigned to variables and passed to, or returned from functions. Upon invocation of a continuation, the control is reset as it was when the continuation was captured.

A common use of continuations is the programming style *Continuation Passing Style* or CPS. In CPS, functions do not return values, instead taking as parameter the continuation to invoke (passing the function's result) upon completion.

Continuation I/O operations are written in CPS, for example:

```
putCharCont :: Char -> Cont -> Cont
getCharCont :: (Char -> Cont) -> Cont
doneCont :: Cont
```

The `putCharCont` function takes a character and a continuation, writes the character to the standard output stream and then invokes the continuation. `getCharCont` takes as parameter a function that accepts a character and returns a continuation, reads a character from the standard input stream and then passes the character to the function. `DoneCont` is a termination continuation that performs no action and is used to end the chain of continuations.

Wadler [25] gives the following continuation I/O implementation for the "echo" program[11]:

```
echoCont :: Cont
echoCont k = getCharCont (\c ->
    if (c == EOF) then
        k
    else
        putCharCont c (echoCont k))

mainCont :: Cont
mainCont = echoCont doneCont
```

Peyton Jones and Wadler [19] show the surprising similarities between continuations and monads, the current method of implementing I/O in Haskell.

Hudak and Sundaresh [14] present a slightly different formulation of continuation I/O, specifically, they include both success and failure continuation parameters to enable failures to be handled by the caller.

Gordon [21] notes that the main advantage of continuation I/O over synchronised stream I/O is that the former cannot suffer from the same synchronisation issues that the later can. In particular, deadlock due to premature response stream evaluation is not possible.

D. Monads

Monads originated in Category Theory, a branch of abstract Mathematics. Moggi [27] was first to observe that monads could be used to structure programs. Wadler [28] [25] and Peyton Jones [19] published several papers, further expanding and elucidating Moggi's ideas. Wadler [29] gives a comprehensive introduction to and motivation for the use of monads in Haskell.

Monads can be used to describe a wide range of structure and behaviour in functional programming languages. They can be used to simulate global state, I/O, exception handling and non-determinism [30]. Peyton Jones outlines the use of monads in modern Haskell [13].

The essential idea of monads in functional programming is that they allow for the construction and sequencing[12] of computations, through the use of just two generalised combinators. The type signatures[13] of the two combinators are:

```
return :: (Monad m) => a -> m a
(>>=)  :: (Monad m) =>
    a -> (a -> m b) -> m b
```

the `return` function takes a value of type `a` and places it into the "default"[14] context of the monad in question, `m`. The operator `(>>=)` pronounced *bind* takes a monad-encapsulated value of type `a`, and a function that takes a value of type `a` and returns a monad-encapsulated value of type `b`, extracting the value from the first argument, and returning the result of passing it to the function. `m a` represents the type of a value of type `a` wrapped within the *computational context* of a monad `m`. For example, the type of the function `getChar` is: `getChar :: IO Char`, that is, getChar is a function that will return an IO computation, which, when executed, will return a value of type Char. Consider the following Haskell expression[15].

```
getChar >>=
    (\c1 -> getChar >>=
        (\c2 -> return (c1,c2)))
```

Fig. 2

This expression sequences two getChar functions, returning a tuple containing the first entered character as its first element, and the second as its second. It has type `IO (Char, Char)`, that is, an IO action that when performed, will

[10] Instead of `ReadChar Char`, the type of `responses !! 0` would be `Done` — the response to the `GetChar` request.

[11] The parameter representing the continuation is commonly named `k`.

[12] Monads do allow for sequencing, and it is that property that the IO monad in Haskell exploits. However, it is not essential for a monad to impose sequencing.

[13] Here, `return` is the name of the function, `::` denotes a type signature, `(Monad m) =>` signifies that the type parameter m has a class constraint within the following type definition (namely, that it must be an instance of a monad) and `a`, `b` and `m` are type parameters.

[14] The default context depends on the monad being used; the default context for `IO` is a computation that when executed simply returns the original value, without performing any I/O.

[15] Brackets are added for visual clarity — parsing is unambiguous in their absence.

return a tuple containing two Chars. Due to the underlying `IO` monad implementation, the I/O will be performed such that the tuple element ordering is always the same as the order-of-character-entry as performed by the user. A possible implementation[16] of the `IO` monad is to model it as a function that transforms "World" values: `data IO a = World -> (a, World)`, known as an implicit *Environment-Passing* scheme, since the `World` value is threaded through the computations by the implementation, not the programmer. The `World` value represents an encoding of the current "state of the world", which is modified by functions such as getChar (where the "state" of the standard input stream would be modified to signify that a character has been read). The data-type `IO a` can therefore be thought of as a function that takes a `World` value, and returns a tuple containing a value of type `a` and a new `World` value. Using this implementation, the bind operator can be used to introduce a so-called *data dependency* between two `IO` computations. A data dependency is introduced whenever an expression contains a sub-expression; the sub-expression must be evaluated before the containing expression, since the value of the expression depends on the value of the sub-expression. For example, consider the following simple expressions:

```
expr = subExpr + 3
subExpr = (1 + 2)
```

there is a data dependency between `expr` and `subExpr`, whereby `expr` cannot be evaluated before `subExpr`.

As a demonstration of the use of data dependencies to ensure correct I/O, an implementation might use the following transformation[17] of the expression in Figure 2.

```
getChar :: World -> (Char, World)
getChar w = (nativeGetChar, mkNewWorld w)

expr :: World -> ((Char, Char), World)
expr w1 = case getChar w1 of
    (char1, w2) -> case getChar w2 of
        (char2, w3) -> ((char1, char2), w3)
```

where `nativeGetChar` is the side-effecting getChar function of the underlying platform[18] and `mkNewWorld` generates a new `World` value. The data dependencies are introduced through the use of the "threaded" `World` values — `w1,w2,w3`. The resulting tuple depends on `w3`, which is generated by the second `getChar` call, which depends on `w2`, as generated by the first `getChar` call. This sequence of data dependencies gives the required sequencing for correct I/O operations, to ensure the first element of the tuple is the first element entered by the user and vice-versa for the second.

Consider an alternative expression (which is not valid Haskell):

```
(nativeGetChar, nativeGetChar)
```

In a lazy language such as Haskell, the order of evaluation is not specified and it is therefore not possible to ensure that the first entered character will necessarily be the first element of the resulting tuple — if the evaluation order is such that the second `nativeGetChar` is evaluated first, the resulting tuple's value ordering will not respect the order-of-character-entry.

The obvious advantage of monadic programming, particularly using Haskell's `do` notation is that of syntactic neatness — there is no "cluttering" of continuations or streams to be passed around in the program, the monad encapsulates the state it requires, effectively "hiding the plumbing".

A potential drawback of monads, is that they force the use of a single environment, to be passed between all I/O operations within a program, effectively sequentialising all I/O operations. Effects encapsulated within the `IO` monad are difficult to reason about, a topic discussed by both Swierstra [32] and Gibbons and Hinze [33].

E. Uniqueness Types

Clean uses a Uniqueness type-system, which is used to confine the copying of expressions. The type-system includes annotations to mark types as either unique or non-unique — expressions used in a context requiring a unique type may have at most one reference to them. Clean's I/O system uses functions that take a unique valued world as argument and return a unique world, in a similar style to the implicit world-passing of Haskell's IO monad.

Uniqueness types offer two main benefits: they permit safe *destructive updates* [34] for efficiency and also enforce the sequencing of I/O operations. The type-system uses a technique called *sharing-analysis* to determine whether there are multiple references to a given expression.

If the compiler can guarantee that an object is used in a unique way, then it is free to employ a destructive update, rather than a full copy and update, giving an increase in performance. Referential transparency is maintained when using a destructive update of a unique object; conceptually, the function is returning a different object, it just happens that the argument no longer exists in its original form. The same function implemented without a destructive update will have the same observable value, and could be used as a less efficient replacement.

Uniqueness types can be used to allow the programmer to use side-effecting functions, in a way that does not violate referential transparency. As an example, consider a function `getChar :: File -> Char`, which may be used to obtain a single character, given a file. Referential transparency dictates that given the same file variable, the function must return the same character. However, this is not much use — the programmer should be able to read all the characters in a file, not just the first. Using uniqueness types, we can mark the function such that it takes a unique file, and returns a pair containing a character, and a new, unique file: `getChar :: *File -> (Char, *File)`[19]. The uniqueness constraint means that the `getChar` function may never be called more than once with the same argument - to do so would mean that the argument must have been duplicated, losing its uniqueness property. Referential transparency trivially holds, since the type-system ensures that the same function is never passed the same expression as an arguments. The net effect is that the programmer is forced to thread the unique values through the program[20], sequencing the order of effects. The required

[16] This is a simplified model, and as such does not take account of the concurrency features of real-world implementations.

[17] Indeed, this is similar to that of the popular GHC compiler [31].

[18] For example, if the compiler targeted C directly, the call would be to the `getc` function contained in the stdio header.

[19] `*File` signifies a unique `File` type.

[20] This style of value threading is known as an explicit *environment passing* style.

sequencing of I/O operations is thus guaranteed by the type-system.

Uniqueness types offer a finer level of granularity, when compared to the IO monad of Haskell. Using uniqueness types, a programmer may specify that a single file is unique, rather than the entire "world". The IO monad implementation in Haskell does not allow such granularity - a file cannot be separated from the world that contains it. This allows Clean to safely perform mutually exclusive I/O operations in parallel, using multiple environments [35], whereas Haskell cannot[21].

One problem introduced by uniqueness types is that the type-system's complexity is increased as it must enforce the uniqueness restrictions. Ensuring uniqueness of arguments and function results is undecidable in the general case [35], however the use of a uniqueness type-system makes the problem tractable. De Vries et al. present a simplified uniqueness type system [36], compared to that of Clean, allowing a simpler type-system implementation.

Butterfield and Strong [37] note that the uniqueness type-system statically ensures that it is not possible to incorrectly attempt to interact with a closed file handle in Clean; since the Clean `closeHandle` call does not return a new handle, the programmer must attempt to reuse an old handle, thus violating the required uniqueness property. There is no such guarantee in Haskell — a programmer can attempt to read or write using a closed Handle, which will cause a runtime exception to be thrown.

F. Iteratees

A common idiom within functional programming is the notion of "folding" a data-structure. Folding is accomplished by iterating a function over each element in the data-structure, which accumulates a single return value. Common implementations of the higher-order-function[22] fold take a function to be iterated, an initial value and a data-structure to iterate over; its type signature for operating on lists would be:

```
fold :: (a -> b -> a) -> a -> [b] -> a.
```

For example, summing the elements of a list can be simulated using a fold: `sum list = fold (+) 0 list`.

Within the fold idiom, there are two main components; we call the object being iterated over the data-structure an *iteratee* and the controller of the iteration the *enumerator*. In the previous example, `(+)` is the iteratee and `fold` is the enumerator. Enumerators can be thought of as iteratee transformers — they pass values to the iteratee, to transform its internal state.

Originally proposed by Kiselyov [38] as a solution to the problems caused by lazy I/O in Haskell, iteratees are an application of the fold operation with possible early-termination, to enable safe, predictable I/O. Lazy I/O enables a file to be read, on demand, in a lazy fashion. The `readFile` function provides lazy I/O [39]:

```
readFile :: FilePath -> IO String
```

as more of the string returned by `readFile` is required, side-effecting file operations are performed. This has many problematic corner cases, as outlined by Kiselyov [40], but essentially the problem is that a pure expression returned by a lazy I/O operation may cause side-effects upon evaluation, breaking referential transparency.

We give a simplified[23] outline of the data-types involved in iteratees:

```
data Stream a = Chunk [a]
              | EOF

data Iteratee a b = Stream a -> Iteratee a b
                  | Yield b

type Enumeratee a b = Iteratee a b ->
                      Iteratee a b
```

that is, an iteratee is either waiting for more input, or has yielded a result value. `Enumeratee` is a type synonym for an iteratee transformer, which are usually implemented by performing some action (for example reading a line from a file) to obtain input, and passing it to the iteratee.

The main improvements of iteratees, over lazy I/O are:

Improved modularity: producer and consumer code is clearly separated, by design, improving composability — multiple enumerators or iteratees can be easily composed.

Guaranteed performance: with lazy I/O, no guarantee is made about when a file handle will be closed, leading to possible runtime resource exhaustion. Iteratees guarantee that an opened file handle is closed when the file's contents have been read (or earlier, if the iteratee yields a value) and that a fixed amount of data is read at once, giving predictable, constant memory usage.

Safety: The implementation of lazy I/O silently discards any errors that occur when using the lazily-read data, simply truncating data. Iteratees explicitly handle exceptions[24]. Since lazy I/O returns a pure value, which when evaluated may read more data from a file, exceptions can be thrown in pure code, where they cannot be handled [43].

V. Demonstration Code

To demonstrate the application of the different methods of I/O in the three example languages, we present an example program, translated into each language. The program implements a simple remote service, which sums over a list of files that each contain a single integer. We only include pertinent snippets with commentary here, due to lack of space; the full source code is made available online [44].

The following code shows a benefit of Clean's uniqueness type system:

```
readInts :: [*File] -> [Int]
readInts []      = []
readInts [f : fs]
    # (ok, i, f) = freadi f
    # is = readInts fs
    | not ok = is
    | otherwise = [i:is]
```

[21] A *commutative* monad allows for arbitrary ordering, but there is no way to express that the commutative property holds for a given IO (or other monad) operation, in Haskell.

[22] Higher-order-functions are functions that either take as parameters, or return, functions.

[23] Real implementations, such as [41] [42] use more complicated definitions that include error handling and monadic actions (our definition doesn't include the ability to actually perform I/O).

[24] Which may become cumbersome, in instances where no recovery is desired or necessary.

`readInts` takes a list of unique files, but since it does not return a list of files, the type-system guarantees that any future attempted use of the file variables is an error. `freadi` is a side-effecting function, with type

```
freadi :: *File -> (Bool, Int, *File)
```

but since the file variables are guaranteed to be unique, there is no chance of another function attempting to read from the same file variable (leading to possible differences in side-effect ordering, dependent on evaluation order). If Clean provided a built in parallel operator, we would be able to take advantage of Clean's environment partitioning — we could read each file, in parallel, unlike in Haskell, where we are forced to use a single (implicit) environment.

The following code shows the drawback of an explicit environment-passing scheme:

```
doLoop :: TCP_Listener *World ->
    (TCP_Listener, *World)
doLoop listener w
# ((_, {sChannel, rChannel}),
    listener, w) = receive listener w
# (bytemsg, rChannel, w) =
    receive rChannel w
# strMsg = removeNL (toString bytemsg)
# (result, w) =
    tryProcess (tryReadCommand strMsg) w
# (sChannel, w) =
    send (toByteSeq result) sChannel w
# w = closeChannel sChannel w
# w = closeRChannel rChannel w
= doLoop listener w
```

Throughout the function, the `World` parameter, `w` has to be manually passed to each function call that affects the world, to ensure the correct sequencing. This puts a burden on the programmer, and visually distracts from the intent of the code — not particularly Clean, even modulo line-wrapping!

The OCaml directory enumeration code demonstrates OCaml's iim

```
let enumDir(dir : string) : string list =
  let fileList = ref [] in
  try
    let dirHandle = opendir dir in
    while true do
      let file = readdir dirHandle in
        fileList := file :: !fileList ;
    done;
    !fileList
  with
  | End_of_file  -> !fileList ;
  | Unix_error _ -> [];;
```

`fileList` is a reference to a list that is destructively updated on each iteration of the loop[25]. The `readdir` function attempts to read the "next" entry in the directory, throwing an exception once there are no further entries. This iterator-like function is very imperative in appearance and use, but has the advantage (similar to non-strict evaluation) that the entire directory is enumerated, only if it is requested.

[25]In OCaml, `!` is the dereference operator and `:=` is the assignment operator.

The following code shows the use of Haskell's lazy I/O:

```
doSum :: String -> IO Integer
doSum dir = do
  contents <- liftM (map (dir </>)) $
    enumDir dir
  fNames <- filterM doesFileExist contents
  lines <- mapM readFile fNames
  return . sum . map read $ lines
```

`readFile` lazily reads the whole file, returning a pure string value. This is not ideal however, since being pure, there is no restriction on the sequencing of consuming the lazy string. This can lead to unexpected results, in the presence of errors (the file contents can be silently truncated). This solution also causes an exhaustion of file handles, if the directory that is being enumerated contains many files[26]. This is because `readFile` strictly opens the files, but lazily reads the data; the Clean implementation also suffers from the same problem, with the same cause. To prevent the issue, a refactoring would be required: instead of opening all the files, then reading a line from each file, we would have to open and read a line from each file in turn. In this case, the refactoring is simple and easy to make, but this is not always the case. Lazy I/O allows the processing of large files, without requiring large amounts of RAM (to store the file in memory, whilst it's being processed), but causes resource exhaustion problems as discussed.

To combat the problems introduced by lazy I/O we implemented an iteratee-based `doSum` function:

```
doSum :: String -> IO Integer
doSum dir = do
  contents <- liftM (map (dir </>)) $
    enumDir dir
  fNames <- filterM doesFileExist contents
  let enums = concatEnums $
    map ET.enumFile fNames
  lines <- run_ $ enums ==<< EL.consume
  return . sum . map (read . unpack) $ lines
```

We use a composed list of enumerators that each read a single file, and an iteratee that creates a list of the values it has consumed. Each file is opened in turn, the iteratee is fed its contents, then the file is closed. Other than using a different function to convert the lines to integers and assigning the enumerators to a variable, the remaining code is the same as the lazy I/O example.

We have shown pertinent snippets of each of our implementations. The use of Clean's uniqueness types forces us to manually thread the environment state through our program, unlike Haskell, which implicitly passes the world environment, "hiding the plumbing" — albeit conservatively over-sequentialising the I/O actions, which could otherwise be performed concurrently. We demonstrated that OCaml allows us to use destructive updates and side-effecting I/O, due to its strict nature; however, we propose that giving up on referential transparency, and consequently, equational reasoning and simple composability, is too great a loss for a language.

VI. Conclusion

Functional programming languages approach the difficulty of providing I/O, whilst maintaining the benefits of a functional programming language in variety of ways. We have explored

[26]Around 1050 files on the first author's Linux machine.

the usage of side-effecting "functions" in a strict language, OCaml, monads and iteratees in the lazy, pure language Haskell, and uniqueness types in the lazy, pure language Clean.

The obvious requirement of an I/O mechanism is that it must correctly enforce the required sequencing of I/O interactions. However, I/O is an integral part of any program, and thus, should not be difficult for a programmer to use. Stream transformers are fragile to use, continuations are powerful but somewhat clutter the syntax of functions. Monads and uniqueness types both present a trade-off, do we accept the over-sequentialisation imposed by monads, or the visual disorder of explicit environment passing? We believe that a compromise is still to be found; I/O is not a particularly active area of research, but new approaches are still being discovered, iteratees being a case in point.

REFERENCES

[1] J. Hughes, "Why functional programming matters," *Comput. J.*, vol. 32, pp. 98–107, April 1989.

[2] H. Barendregt, *The Lambda Calculus, Its Syntax and Semantics (Studies in Logic and the Foundations of Mathematics, Volume 103). Revised Edition*. North Holland, 1985.

[3] P. Sestoft, "Demonstrating Lambda Calculus Reduction," in *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones, number 2566 in Lecture Notes in Computer Science*. Springer-Verlag, 2001, pp. 420–435.

[4] P. Wadler, "Lazy versus strict," *ACM Comput. Surv.*, vol. 28, pp. 318–320, June 1996.

[5] C. Okasaki, *Purely Functional Data Structures*. Cambridge University Press, 1999.

[6] G. D. Plotkin, "Call-by-Name, Call-by-Value and the lambda-Calculus," *Theor. Comput. Sci.*, vol. 1, no. 2, pp. 125–159, 1975.

[7] P. Wadler, W. Taha, and D. Macqueen, "How to Add Laziness to a Strict Language Without Even Being Odd," in *Workshop on Standard ML, Baltimore.*, 1998.

[8] T. Sheard, "A Pure Language with Default Strict Evaluation Order and Explicit Laziness," August 2003.

[9] A. Sabry, "What is a Purely Functional Language?" *Journal of Functional Programming*, vol. 8, pp. 1–22, 1998.

[10] W. V. O. Quine, *Word and Object (Studies in Communication)*. The MIT Press, 1964.

[11] C. Strachey, "Fundamental Concepts in Programming Languages," *Higher Order Symbol. Comput.*, vol. 13, pp. 11–49, April 2000.

[12] H. Søndergaard and P. Sestoft, "Referential transparency, definiteness and unfoldability," *Acta Inf.*, vol. 27, pp. 505–517, January 1990.

[13] S. P. Jones, "Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell," in *Engineering theories of software construction*, 2002, pp. 47–96.

[14] P. Hudak and R. S. Sundaresh, "On the Expressiveness of Purely Functional I/O Systems," Yale University, Tech. Rep., 1989.

[15] Haskell — HaskellWiki. [Online]. Available: http://www.haskell.org/haskellwiki/Haskell [Accessed: 09/02/2011]

[16] Clean. [Online]. Available: http://wiki.clean.cs.ru.nl/Clean [Accessed: 09/02/2011]

[17] Objective Caml. [Online]. Available: http://caml.inria.fr/ocaml/ [Accessed: 09/02/2011]

[18] P. Hudak *et al.*, "A history of Haskell: being lazy with class," in *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, 2007.

[19] S. L. Peyton Jones and P. Wadler, "Imperative functional programming," in *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '93, 1993, pp. 71–84.

[20] P. Landin, "A correspondence between ALGOL 60 and Church's Lambda-notations: Part II," *Communications of the ACM*, vol. 8, no. 3, pp. 158–167, 1965.

[21] A. Gordon, *Functional Programming and Input/Output*. Cambridge Univ Press, 1994.

[22] J. O'Donnell, "Dialogues: A basis for constructing programming environments," in *Proceedings of the ACM SIGPLAN 85 symposium on Language issues in programming environments*. ACM, 1985, pp. 19–27.

[23] P. Hudak *et al.*, "Report on the programming language Haskell: a non-strict, purely functional language version 1.2," *SIGPLAN Not.*, vol. 27, pp. 1–164, May 1992.

[24] W. Stoye, "The implementation of functional languages using custom hardware," Computer Laboratory, University of Cambridge, Tech. Rep., Dec. 1985.

[25] P. Wadler, "How to declare an imperative," *ACM Comput. Surv.*, vol. 29, pp. 240–263, September 1997.

[26] J. Reynolds, "The discoveries of continuations," *Lisp and symbolic computation*, vol. 6, no. 3, pp. 233–247, 1993.

[27] E. Moggi, "Notions of Computation and Monads," *Information and Computation*, vol. 93, pp. 55–92, 1989.

[28] P. Wadler, "Comprehending Monads," in *Mathematical Structures in Computer Science*, 1992, pp. 61–78.

[29] P. Wadler, "The essence of functional programming," in *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1992, pp. 1–14.

[30] P. Wadler, "Monads for Functional Programming," in *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, 1995, pp. 24–52.

[31] GHC/Base.lhs. [Online]. Available: http://www.haskell.org/ghc/docs/7.0.3/html/libraries/base-4.3.1.0/src/GHC-Base.html#bindIO [Accessed: 29/04/2011]

[32] W. Swierstra, "A Functional Specification of Effects," Ph.D. dissertation, University of Nottingham, 2009.

[33] J. Gibbons and R. Hinze, "Just do it: Simple monadic equational reasoning," March 2011, submitted for publication. [Online]. Available: http://www.comlab.ox.ac.uk/jeremy.gibbons/publications/mr.pdf [Accessed: 30/04/2011]

[34] S. Smetsers *et al.*, "Guaranteeing Safe Destructive Updates through a Type System with Uniqueness Information for Graphs," 1994.

[35] P. Achten and R. Plasmeijer, "The Ins and Outs of Clean I/O," 1995.

[36] E. D. Vries, R. Plasmeijer, and D. M. Abrahamson, "Uniqueness Typing Simplified."

[37] A. Butterfield and G. Strong, "Proving Correctness of Programs with IO - A Paradigm Comparison," in *Selected Papers from the 13th International Workshop on Implementation of Functional Languages*, ser. IFL '02, 2002, pp. 72–87.

[38] O. Kiselyov. Streams and Iteratees. [Online]. Available: http://okmij.org/ftp/Streams.html#iteratee [Accessed: 01/05/2011]

[39] B. O'Sullivan, J. Goerzen, and D. Stewart, *Real World Haskell*. O'Reilly Media, 2008.

[40] O. Kiselyov. Lazy vs Correct I/O. [Online]. Available: http://okmij.org/ftp/Haskell/Iteratee/Lazy-vs-correct.txt [Accessed: 01/05/2011]

[41] O. Kiselyov and J. W. Lato. HackageDB: iteratee-0.8.2.0. [Online]. Available: http://hackage.haskell.org/package/iteratee [Accessed: 01/05/2011]

[42] J. Millikin. HackageDB: enumerator-0.4.10. [Online]. Available: http://hackage.haskell.org/package/enumerator [Accessed: 01/05/2011]

[43] A. Reid, "Handling exceptions in Haskell," in *submitted to Practial Applications of Declarative Languages (PADL'99)*, 1999.

[44] Owen Stephens IRP Code. [Online]. Available: http://owenstephens.co.uk/files/code.html [Accessed: 09/05/11]