Codata and Comonads in Haskell [1]

Richard B. Kieburtz
Oregon Graduate Institute
Portland, Oregon, USA
dick@cse.ogi.edu

1 Introduction

Haskell, a wide spectrum, functional programming language, provides means to define and use an extremely rich variety of data including free, polymorphic datatypes, type classes, and data with additional computational structure abstracted by monads. Somewhat less attention has been given to supporting abstract data types, which we shall call *codata* types. Monomorphic, parameterless versions of codata types can be defined by modules, but Haskell's module system is not comparably powerful with the class system.

Through a relatively innocuous and well understood extension, namely rank 2 polymorphic types,

Haskell can provide first-class codata types. This suggestion has precedents. Läufer and Odersky [LO94] have advocated first-class abstract data types as an extension of algebraic data types and have suggested existential type quantification as a means of hiding representations. Their suggestions were first implemented in the *hbc* compiler for Haskell and in an undistributed version of Caml-Light. Subsequently, rank 2 polymorphism has been implemented as a Haskell extension in Hugs versions 1.3 and Hugs98 and in the *ghc* compiler. Simon Peyton Jones has suggested that with the ability to hide a representation type, Haskell's modules permit the declaration of abstract objects (without an inheritance mechanism) [Jon99].

The second topic of the paper is comonads in Haskell. Monads are by now well known and have been found to be extremely useful in programming. Monads serve several purposes:

- they integrate semantic effects into an otherwise purely functional framework [Mog91];
- they provide easily interchangeable semantic interpretations of a common program framework [Wad92];

- they suggest type system extensions that can enforce static encapsulation of state [LP95] and other effects [Kie98].

---

[1] The research reported in this paper was supported by the USAF Materiel Command.

Monads account for computational effects of executing a program segment. Many computational effects can be restricted to a segment of a program in which the effect is visible. To encapsulate an effect, there must be a means to initialize the effect mechanism and to recover a pure value from a program segment that relies upon the effect. The use of a local state variable in a computation is an example of an encapsulatable effect mechanism.

Every monad provides a polymorphic combinator that injects a value into an effect-dependent computation (the *unit* of the monad, called `return` in Haskell). The `bind` combinator extends effect-dependent functions to functions that take effect-producing computations as arguments, so they may be composed.

However, monads do not account as naturally for effects that derive from the context in which a program segment is executed. A program can always project a value from a computation passed to it from its context. Otherwise, the imported computation

would have no interaction with the program. However, not every value can be injected into a computation meaningful in the context. Some types defined within the program may be unknown in the context. This is complementary to the circumstance of a monadic effect, where a computation in the monad can be constructed with the `return` combinator at every type, but the ability to extract a value from a monadic computation is not universal.

Take monadic input/output, for instance. Conceptually, the *IO* monad encapsulates effects in which the Haskell program's evaluation may "change the world". However, the *IO* operations of a program do not so much change the world as they allow a program to respond to a changing world. The *IO* system is provided by the context in which a Haskell program is executed; it is not defined by Haskell's semantics. We note that an *IO* subsystem is not initialized by a Haskell program. The Haskell implementation does not determine the representation of its objects, and its effects cannot be restricted to just those effects manifested by executing a Haskell program.

To further support this point of view, note that when a program reads or writes to a file other than the standard input or output, a file handle is required as an argument of the functions; the basic *IO* operations

to a file are typed as:

```
hGetChar ::  Handle -> IO Char
hPutChar ::  Handle -> Char -> IO ()
```

If the type constructor `IO` is interpreted to designate a computation manifesting a side effect of an *IO* system, these types are unintuitive. It is surprising that the `Handle` type, which is associated only with the file system and whose semantics is said to be "implementation dependent", does not appear under the `IO` type constructor.

A more intuitive typing for these operators might be:

```
hGetChar' ::  IO Handle -> Char
hPutChar' ::  IO Handle -> IO Char -> ()
```

With this typing, it is more natural to associate the effect of the *IO* subsystem with the environment in which a program is run, a concept that seems better aligned with reality than is the view that *IO* is an effect determined by Haskell semantics.

The types suggested here for `hGetChar'` and `hPutChar'` are typical of functions typed in a *comonad*, the mathematical structure dual to a monad. This paper suggests that a useful interpretation of comonads in programming is that they ac-

count for effects originating in the context of a program fragment, effects to which the program fragment may react and which it may further propagate. We shall see several examples in a later section.

2 Codata types and modules

Codata is an appropriate name for objects that have historically been called abstract data types. These objects differ from data in several, fundamental respects:

- The structure of a data object is exposed, while the structure of a codata object is hidden.
- Unlike a data object, a codata object does not have the attributes ordinarily associated with a "value". The `show` function is not defined on a codata object; codata cannot belong to the `Eq` class; there are no derived methods on a codata object.
- A data object has open access, as its structure is visible. A codata object has only the access methods specified by its declaration. These methods comprise a set of typed functions, some of which yield data values when invoked, and

some of which may yield a new codata object.

- Data objects are usually finitary (lazily evaluated recursive data structures are an exception to this rule) while codata objects are usually infinitary (finite records are an exception to this rule).

2.1 Streams

There are many familiar examples of codata objects in programming. Infinite streams furnish a particularly simple example. A stream is a sequence whose elements are only accessible in order. A stream codata type is characterized by two access methods:

$$\begin{aligned} \mathit{shd} &:: \mathit{Stream}\ a \to a \\ \mathit{stl} &:: \mathit{Stream}\ a \to \mathit{Stream}\ a \end{aligned}$$

The access methods of any codata object are total functions. Since the *stl* function can be iteratively applied without bound, a stream is seen to be an infinitary object.

Haskell programmers often make use of streams, encoded as lazy lists. The functions *shd* and *stl* are represented by functions $hd :: \mathit{List}\ A \to A$ and $tl :: \mathit{List}\ A \to \mathit{List}\ A$ which can be defined in terms of pattern matches on the structure of a list argument.

However, *hd* and *tl* are partial functions which may raise a pattern match exception if applied to an empty list. A lazy list representation *simulates* a stream, but lazy lists are not isomorphic to streams.

Furthermore, the stream codata type does not imply that an evaluated stream segment is automatically cached, as does the lazy evaluation of a segment of a list. While the caching of values provided by lazy evaluation is often important[2], its benefit is not without cost in time to access a value. If a function over stream data can be computed synchronously and scheduled deterministically, then uncached elaboration of a stream could be more economical.

Other examples of codata in programming are the "objects" of languages such as Java, Smalltalk, Eiffel and C++. Of course, these object-oriented languages provide stateful objects rather than functional objects as their default. They also support inheritance among object classes, which is a notion orthogonal to that of codata. However, the notion of objects restricted to abstract data types has a long history in programming, and was found in early versions of the functional language, ML.

2.2 Modules declare codata types

Module declarations are the mechanism provided in

Haskell to hide representation while exporting access methods. Modules should provide the means to define new codata types, but unfortunately, standard Haskell modules are not equipped with a sufficiently expressive type structure. Suppose, for instance, that we wish to define a Haskell module to export the operators of a polymorphic stream type. We would like to write a module declaration such as:

```
module Stream
 (Stream,
  shd,
  stl,
  mkStream
 )
 where
   data Stream a = S b (b -> a) (b -> b)
```

[2] When a program is written as a set of recursive equations over streams [LLC99], failure to cache evaluated stream segments can cause the time complexity of an algorithm to grow from linear to exponential, if a stream is duplicated in the computation.

```
   shd :: Stream a -> a
   shd (S x f g) = f x

   stl :: Stream a -> Stream a
   stl (S x f g) = S (g x) f g
```

```
mkStream ::
   b -> (b -> a) -> (b -> b) -> Stream a
mkStream x f g = S x f g
```

in which the type variable `b` occurring in the declaration of the data type `Stream a` designates a hidden representation type (this is the *carrier* type of a stream coalgebra). However, this type variable is unbound in the text given above and the data type declaration will not be accepted. The type variable `b` could be bound along with the type parameter `a` of the `Stream` data type declaration, but this would reveal the representation type, as the exported type would be `Stream a b`. If this binding was adopted, the representation type would be visible in the type of each instance of a stream.

A solution that preserves the stream abstraction is furnished by rank 2 polymorphism (universal quantification over types, nested within type expressions) which has been implemented as an extension to standard Haskell in the *ghc* and *hbc* compilers and the *Hugs* interpreter. The syntax we use here is valid in Hugs version 1.3 and Hugs98, when run in Hugs-extensions mode. The keyword `forall` can be used to bind a type variable within a declared type expression. Using this facility, the data type declaration

within the module above can be written as

```
data Stream a =
        forall b. S b (b -> a) (b -> b)
```

In this declaration, the type variable `b` is in scope only in the domain type of the signature of the unexported data constructor, `S`.

The declaration above uses universal quantification over a type variable that occurs in the domain type of a constructor to realize an existential type quantification. The logical equivalence on which this trick relies is $(\forall b.\, P \Rightarrow Q) \equiv ((\exists b.\, P) \Rightarrow Q)$, when b has no free occurrence in Q. The apparently superfluous data constructor, `S`, in the type declaration of `Stream a` effectively represents the leftmost implication in the formula above, when P stands for the type `(b, (b -> a), (b -> b))` and Q stands for the type `Stream a`. In each instance of an expression of the declared datatype, the bound type variable may be independently instantiated to a specific type.

When the declaration above is substituted for the datatype declaration in the module `Stream`, the module declaration is accepted by the Hugs98 interpreter and exports the type constructor `Stream` along with the signatures of the three operators `shd, stl` and `mkStream`. The data constructor `S` and the repre-

sentations of the exported functions remain hidden to users of the module. To create an instance of a stream, a program invokes `mkStream` applied to three appropriately typed arguments. For example,

```
let fibs = mkStream (1,1)
                    snd
                    (\(j,k) -> (k,j+k))
in ...
```

specifies the Fibonacci sequence as a stream of integers.

A potential drawback of including the "forall" quantifier in Haskell data types is that a pattern match on a data value constructed in a type whose constructors take arguments of a quantified type might then escape its intended scope (and the scope of the module). A potential escape of scope can easily be detected if type quantification is restricted to declarations of data constructors that are not exported from a module declaration. Encapsulation of a quantified type can be enforced by type checking if the use of explicit type quantification is so restricted.

2.3 A theory of codata

In this section, we provide a brief, intuitive and en-

tirely descriptive summary of the rich theory that underlies codata objects.

Mathematically, codata objects are *structure coalgebras*. Recall that a coalgebra consists of a type, t, together with a tuple of one or more total functions, each of which has t as its domain type. The type, t, is called the *carrier* of a coalgebra and the functions are its access methods.

The "structure" of a coalgebra is specified by the collection of codomain types in the signature of its access methods. There may, of course, be many coalgebras with the same signature. These constitute a class, or *co-variety*, characterized by their common signature.

If the access methods of a coalgebra are specified only up to their signatures, as is the case for the *Stream* codata type discussed in the preceding section, then the coalgebra is said to be *cofree*. A cofree coalgebra is generic in the sense that its theory is also satisfied by every other coalgebra in its class. A coalgebra specification that was not cofree would include additional equations relating its access methods.

An abstract codata declaration specifies an entire class of coalgebras, each with the structure given in the specification module. To form an instance of the coalgebra class it is necessary to give a representa-

tion type and implementations for each of the access methods. Notice that the `mkStream` method for the *Stream* codata type requires three parameters: a value in the representation type that is used to initialize the stream, and two functions that implement the `shd` and `stl` access methods. The representation type can be inferred from the first argument passed to `mkStream`. Thus the parameters of `mkStream` furnish the data required to form a concrete instance of the *Stream* coalgebra class.

When a coalgebra is polymorphic in its representation type and its access methods are prescribed only up to their types, it is said to be *final*. A final *Stream* coalgebra is what remains visible when the representation of a stream is hidden. The importance of final codata types is that they allow programs to be written generically, to operate on any object of the codata type, whereas a program that relies upon a specific representation, such as the lazy-list representation of streams in Haskell, must be modified if it is to operate on other representations of the same class of codata.

A recent series of papers by Bart Jacobs offers a formal theory of objects modeled as coalgebras [Jac95, Jac96a, Jac96b].

3 Comonads

A comonad is a dual structure to a monad. Formally, a comonad consists of:

- a type constructor, W, taking a single argument
- a polymorphic function $co\text{-}eval : Wa \to a$,
- a polymorphic function
  $co\text{-}ext : (Wa \to b) \to Wa \to Wb$,
- three laws:

$$co\text{-}ext\ co\text{-}eval = id_W \qquad (right\text{-}id)$$
$$co\text{-}eval\ .\ co\text{-}ext\ f = f \qquad (left\text{-}id)$$
$$co\text{-}ext\ f\ .\ co\text{-}ext\ g = co\text{-}ext(f\ .\ co\text{-}ext\ g) \qquad (assoc)$$

Monads and comonads are intimately related in category theory, but we shall not go into that aspect in this paper.

Monads have been added to Haskell almost exclusively through its libraries. The only syntactic concession to monads in Haskell that could not be realized through definitions of classes and operators is the `do` syntax. Comonads can similarly be added by declaring a class

```
class Comonad w where
    (=>>)   :: w a -> (w a -> b) -> w b
```

```
(.>>)   :: w a -> b -> w b
coeval  :: w a -> a
```

The infix operator `=>>` is the function *co-ext* with its argument positions reversed to correspond to the "bind" operator of a monad in Haskell. As is the case with the Monad class, it is the programmer's responsibility to assure that the functions supplied in forming an instance of the Comonad class satisfy the three laws required of a comonad. If these laws are not satisfied, then the behavior of the class instance may not be as expected.

The combinator `(.>>)` serves to propagate demand to its first argument, which otherwise might not be evaluated by the lazy evaluation rule. It satisfies the equivalence:

$$e_1 \texttt{ .>> } e_2 = \texttt{seq}\, e_1\, e_2$$

3.1 Monads, comonads and functional purity

In Haskell, an expression typed in an effects monad, such as *St* or *IO*, signals the possibility of a side effect occurring when a value of the expression is demanded. Because of the possibility of effects, demands for values of monadically typed expressions are scheduled statically through the use of a special combinator. To

apply a function typed as `a -> m b` to an argument typed as `m a`, where `m` is a type constructor belonging to the `Monad` class, one must use the monad `bind` operator (or the `do` syntax, as shorthand). The `bind` operator forces sequential evaluation, to the extent required by the semantics of a particular monad instance. In the *IO* monad, whose effects may be the most far-reaching, `bind` effectively implements call-by-value function application.

3.1.1 Monads encapsulate effects

It is impossible to write a side-effecting Haskell expression that isn't typed in a monad without using an explicitly unsafe coercion operator, such as `unsafePerformIO`, to circumvent the normal typing rules. We shall call programs that use such coercions "unsafe" Haskell programs; otherwise in our discussion we assume safe Haskell programs.

Note that a function typed as `a -> m b` (where `m` may be a monadic type constructor but `a` is not a monadic type) is pure; applying such a function to an argument of type `a` can have no side effect. The codomain type, `m b`, is a poset of *computations* in the monad named by `m`, but any such computation remains suspended until its evaluation is forced

by a demand for a type `b` value or for the state of the monad. Applications of the `bind` operator force such evaluations; these are the only sites within a safe Haskell program at which demands for potentially side-effecting computations occur.

An impure function is identified with a type `m a -> b`. That is, its domain type is monadic. The typing assertion `unsafePerformIO :: IO a -> a` warns of this function's intrinsic impurity. There are no impure functions in the Haskell standard prelude, apart from `bind`, an operator of the class `Monad`.

An impure function must either be introduced in a library extension as an explicitly unsafe operator, or programmed using one of the existing unsafe coercions (an unsafe programming practice) or it must be programmed using `bind`. For example, the monadic map of a function `f :: a -> b` is

```
mapf :: Monad m => (a -> b) -> m a -> m b
mapf f mx = mx >>= \x -> return (f x)
```

The function `mapf f` is potentially impure, even if `f` is a pure function, because to evaluate an application of `mapf f` forces evaluation of its monadically typed argument.

Haskell's type system assures that an impure function gotten by safe programming, that is, only with

`bind`, has its codomain typed in the same monad as its argument[3]. This ensures that any application of an impure function (which may produce a side effect) is typed in a monad that encapsulates the effect. Thus, no expression that is not typed in a monad can evidence a side-effect in a safe Haskell program.

Typically, *IO* operations of a Haskell program are performed in the `main` module, which has the type `() -> IO()`. This type indicates that it is evaluated only for its effects, which are manifested through the *IO* system. One can think of its linkage to the underlying runtime system as an implicit `bind` operation in the *IO* monad.

3.1.2 Comonads can also encapsulate effects

Projection of a value from an expression typed in the comonad is not a side-effecting computation. The combinator `coeval` provides such a projection at every type. A potentially side-effecting computation constructs a value typed in the comonad. When context-dependent effects are expressed in terms of a comonad, a function typed as `w a -> b`, where `w` is a type constructor that belongs to the class `Comonad`, is pure—application of the function cannot manifest a side effect. Of course, in a lazy language, evaluating

an application may propagate demand for the construction of a value typed as `w a`, which can produce an effect.

The key to making comonadic effects safe, then, is to restrict, via the type system, computations of values of type `w a`. Since comonads are used to encapsulate effects defined in the context of a program, an effects-safe program can allow comonadic effects to be introduced only by expressions formed by applications of the 'cobind' operator, `(=>>)`. However, since there is no Haskell constant of a type `w a`, there isn't any initial argument for an application of `(=>>)`.

To remedy this situation yet assure that effects-capable computations are typed in a comonad, we suggest a top-level program of the form

[3] Strictly speaking, the codomain could be typed in a composite monad which subsumes the structure of the monad to which the domain type belongs.

`* =>> main()`, where '`*`' represents an unspecified constant of type `w ()`. This allows `main` to be typed as `w () -> ()`, and hence to consist of a series of comonadic functions composed with `(=>>)`.

With these restrictions, no function with a type `a -> w b` can be applied, where `a` is not a comonadic

type. Thus every function application is safe for comonadic effects.

The suggestion of a top-level program structured to admit sequential evaluation of expressions that may produce effects is not original. In his seminal 1978 Turing award paper [Bac78], John Backus introduced the functional language FL, but suggested in a later chapter of the paper that programs that interact with their environments would need imperative code at the top level. He suggested a language that he called AST (Applicative State Transitions) which is imperative at the top level, but relies upon FL to declare the functions it uses.

3.2 Some useful comonads

Comonads have not been widely recognized in connection with programming, because attention has been focused primarily on semantic effects of a program itself. However, the effects arising from a program's context can be equally important. Contextual effects are captured by comonads. In some cases we shall give explicitly a datatype construction that could represent the effect functionally. In others, such as IO, it does not seem useful to encode the effect mechanism in a functional representation that we do not consider

to be an accurate operational model.

3.2.1 State in context

For our first comonad, let's consider a monad of state, where the state exists in a context.

$$\begin{aligned} &\mathit{Co\text{-}St}_s\ a = ((s \to a),\ s) \\ &\mathit{co\text{-}eval}\,(g, x) = g\ x \\ &\mathit{co\text{-}ext}\,f = (\lambda(g, s).(\lambda s'.\ f\ (g, s')),\ s) \end{aligned}$$

The functor map is

$$\mathit{mapCo\text{-}St}\,f = \lambda(g, x).\,(f\,.\,g,\ x)$$

This comonad affords a different view of state than does the monad of state. The *co-eval* operator projects an observable value from the state component. Given a function $f :: \mathit{Co\text{-}St}_s\ a \to b$, an application of the co-extension $\mathit{co\text{-}ext}\,f$ yields a computation in the comonad, equipped with a new function to project an observable value from the state. Whether and when the state component itself is mutated depends upon the context of the program that embodies this comonad, not upon the action of the program.

3.2.2 A comonad of Streams

As a second example, let's consider an abstract view of streams, one in which the representation type is opaque.

$$\begin{aligned} &\mathit{Stream}\ A = \langle \mathit{representation\ hidden} \rangle \\ &\mathit{co\text{-}eval} = \mathit{shd} \\ &\mathit{co\text{-}ext}\ f = \lambda s.\ \mathit{mkStream}\ s\ f\ \mathit{stl} \end{aligned}$$

The co-extension satisfies a pair of equations:

$$\begin{aligned} &\mathit{shd} \circ (\mathit{co\text{-}ext}\ f) = f \\ &\mathit{stl} \circ (\mathit{co\text{-}ext}\ f) = (\mathit{co\text{-}ext}\ f) \circ \mathit{stl} \end{aligned}$$

These equations express a universal property; a homomorphism that holds for every *Stream* object.

The *Stream* comonad map is an instance of the co-extension, namely $\mathit{mapStream}\ f = \mathit{co\text{-}ext}\ (f \circ \mathit{shd})$. Accordingly, $\mathit{mapStream}\ f$ has the properties:

$$\begin{aligned} &\mathit{shd} \circ \mathit{mapStream}\ f = f \circ \mathit{shd} \\ &\mathit{stl} \circ \mathit{mapStream}\ f = \mathit{mapStream}\ f \circ \mathit{stl} \end{aligned}$$

For example, if we reveal the representation used in the Haskell module `Stream` of Section 2, it is easily seen that

$$\mathit{co\text{-}ext}\ h\ (S\ s\ f\ g) = S\ s\ (\lambda s'.\ h\ (S\ s'\ f\ g))\ g$$

satisfies the equations required of the co-extension. In practice, however, one would not wish to reveal the representation of a stream object[4]. Instead, the *Stream* comonad should be defined in the module `Stream` with the functions *co-eval* and *co-ext* as exported functions.

3.2.3 Using streams in programs

Reactive programs produce output events in response to input events. A reactive program is synchronous if each output event is produced in response to an individual input event and is deposited into the program's dynamic execution context before the context changes state or produces another input event. Thus a synchronous reactive program can be characterized by an event-transforming function, $f :: input\ event \to output\ event$. The reactive program is $mapStream\ f :: Stream(input\ event) \to Stream(output\ event)$.

In applications such as programming a simulator for a microprocessor architecture [LLC99] it is necessary to express a stream delayed by some fixed number of elements. When a stream, `s`, is represented by a lazy list, a unit delay is expressed by extending the list at its front with a new initial value, `x_0:s`. To delay an abstract stream, we can write

[4] We've experimented with programming functions on streams in a strict language, Caml-Light, which does not support the rank 2 polymorphic typing used to hide the stream representation in Hugs. To translate examples, it was necessary to use a revealed representation for streams. Programs written in terms of explicit representations are not pretty.

```
mkStream (x_0,s) fst ((pr shd stl).snd)
  where pr f g x = f x, g x
```

A stream can represent the sequence of state valuations of a finite transition system. The *shd* function projects the visible part of the current state value. The state transition function is *stl*.

A stateful function over a stream maps input elements to output elements but the map may depend upon the prior history of it use. The state can be represented by an internal stream. Suppose `Stream a` is the type of inputs, `Stream b` the type of outputs, and `s` the type of the internal state. A stateful stream map can be programmed as

```
mapStreamSt :: (a -> s -> b) ->
               (a -> s -> s) ->
               s -> Stream a -> Stream b
mapStreamSt f1 f2 s0 xs =
  mkStream (xs, s0)
           (\(x,s) -> f1 (shd x) s)
           (\(x,s) -> (stl x, f2 (shd x) s))
```

3.2.4 A comonad of parallel threads

Parallel evaluation may be supported by the context of a Haskell program but is not explicit in Haskell semantics. Parallel evaluation is an appropriate mechanism to represent abstractly with a comonad. It will allow a programmer to specify what segments of a program should be executed in parallel in the event that the program is run in an evaluation context that supports parallelism.

Fine-grained parallel execution is commonly expressed in terms of parallel threads. A thread executes concurrently with other threads without synchronizing its activity with them. Synchronization can be programmed explicitly by inserting the operator *fence*, to specify that the results of evaluating an expression with a cluster of threads are to be combined before further computation is enabled. The value computed from the *fenced* expression is then returned to a single continuation thread in the context of the expression.

Parallel threads can be expressed in terms of a comonad,

$$Par\, a = \langle hidden\ representation \rangle$$

$$co\text{-}eval = fence$$
$$co\text{-}ext\, f = mapPar\, f\, .\, mkThread$$
where $f :: Par\, a \to b$

The polymorphic operator $mkThread :: Par\ a \to Par(Par\ a)$ starts a new thread. $mapPar :: (a \to b) \to Par\, a \to Par\, b$ is the comonad map for Par. The effect of $mapPar\, f$ allows an application of f to a suitably typed argument to be evaluated in the presence of parallel threads of computation. These operations are related by:

$$fence \circ mkThread = id$$

Keep in mind that satisfaction of the above formula is not a trivial requirement. A multi-threaded evaluation model that cannot ensure the validity of this condition does not have the properties of a comonad for a functional language. This requirement can be considered as a correctness condition for a multi-threaded implementation. Without it, the operational semantics of programs may be nondeterministic.

Parallel evaluation of pairs If we designate the two-threaded, parallel evaluation of an expression e

by $(e \| e)$, the semantics of *fence* can be expressed by

$$fence = \lambda(x \| y).\, x \sqcup y$$

To specify the parallel evaluation of two copies of an expression, we write $mkThread\ e$, which is equivalent to $(e \| e)$.

A suitable domain for these semantics is a partially-ordered set in which the values, or computable elements are the prime elements of posets of approximating elements. An element p is prime with respect to a poset X if $\forall x, y \in X. p \leq x \sqcup y \Rightarrow p \leq x \vee p \leq y$. Then $Par\ Q = (\{X \subset Q \mid X \text{ has a sup}\}, \subseteq)$ and $co\text{-}eval\ X = sup\ X$ and $f\ X \equiv \bigsqcup\{f\ Z \mid Z \subseteq X\}$.

For a program to invoke parallel evaluation to speed up its own computation on a platform that supports parallel threads, it must be possible to evaluate the components of a pair in parallel. To specify parallel evaluation of the components of a pair of expressions,(e_1, e_2) , we write $mkThread\ (e_1, e_2)$. If the immediate context of this parallel expression is *fence*, then an admissible strategy for its evaluation is to calculate $(e_1, \bot) \| (\bot, e_2)$. The presence of $\bot$ as a component of a pair indicates that a thread need not invest any energy in evaluating that component. Evaluation of $fence\ (mkThread\ (e_1, e_2))$ is equivalent

to $(e_1, \bot) \| (\bot, e_2) = (e_1, e_2)$.

The *Par* comonad does not coexist comfortably with some other comonads, such as *IO*, in which the sequentialized behaviors of a series of interactions of a program with its environment can be observed. The type system can be used to identify safe circumstances in which to invoke parallel evaluation. A function typed as $\tau_1 \to \tau_2$, where neither τ_1 nor τ_2 contains an occurrence of the type constructor *IO*, can be re-written to specify parallel evaluation of the function body without concern that multiple threads might interfere by invoking *IO* actions. The parallel version acquires the type $Par(\tau_1) \to \tau_2$.

3.2.5 The OI comonad

We have argued previously that input-output operations fit more naturally in a comonad than in a monad, because *IO* effects are derived from the context of a program, not from the program itself. These effects cannot be expressed in the semantics of the programming language because they are manifested differently on various computational platforms. To avoid confusion with the *IO* monad in Haskell, we shall use *OI* as the name of the comonad. We can express *OI* as a comonad with:

$$
\begin{aligned}
&OI\,A = \langle \mathit{hidden\ representation} \rangle \\
&\mathit{co\text{-}eval} = \mathit{synchOI} \\
&\mathit{co\text{-}ext}\ f = (\mathit{mapOI}\ f)\ .\ \mathit{enableOI} \\
&\text{where } f\ ::\ OI\,A \to B
\end{aligned}
$$

The operation $\mathit{synchOI}$ commits all IO activity whose effects may be observed by a program in execution or visible to its human operator, by flushing buffers and transmitting data. These operations are not directly controlled by an application program but they may be requested by a running application through its API.

The combinator $\mathit{mapOI} :: (A \to B) \to OI\,A \to OI\,B$ has the operational effect of hiding all currently open files and buffers of the IO system from the function (program) it receives as an argument. Thus, that function could be a candidate for multi-threaded, parallel evaluation.

The polymorphic combinator $\mathit{enableOI} :: OI\,A \to OI(OI\,A)$ has the effect of copying pointers to currently accessible IO resources, in effect, duplicating the current IO environment. The operational effect of $\mathit{co\text{-}ext}\ f$ allows a function f, which accepts arguments that may depend upon the current IO environment, to propagate the IO environment as an implicit parameter along with its result.

An OI comonad in Haskell To experiment with comonadic input-output, several of the *IO* primitives of Haskell have been repackaged in a module that imports the Haskell *IO* module and exports a set of functions that produce the same behaviors but whose types are compatible with the `OI` comonad. Some of these primitives are shown in the module declaration below.

```
module OI_comonad
  (OI,
   coOpenFile,
   coGetChar,
   coPutChar,
   stdGetChar,
   stdPutChar,
   coClose)
where
import IO
import IOExts

import Comonad

data OI a = D a | forall b. StdOI b

stdOI :: OI a     -- this constant provides an
stdOI = StdOI ()  -- argument to which OI oper-
                  -- ations can be applied
```

```
coOpenFile :: String -> IOMode -> OI () -> Handle
coOpenFile s m stdIO =
     unsafePerformIO (openFile s m)

coGetChar :: OI Handle -> Char
coGetChar =
     \(D h) ->  unsafePerformIO (hGetChar h)

coPutChar :: Char -> OI Handle -> ()
coPutChar  =
     \c (D h) -> unsafePerformIO (hPutChar h c)

stdGetChar :: OI () -> Char
stdGetChar stdOI = unsafePerformIO getChar

stdPutChar :: OI Char -> ()
stdPutChar = \(D c) -> unsafePerformIO (putChar c)

coClose :: OI Handle -> ()
coClose = \(D h) -> unsafePerformIO (hClose h)

instance Comonad OI where
  w =>> f = D (f w)
  coeval (D a) = a
```

A simple program using the `OI` comonad is:

```
module Bar
```

```
where
import Comonad
import OI

echo :: String -> OI a -> ()
echo s t =
   coeval (t =>>
           coOpenFile s ReadMode =>>
           coGetChar =>>
           stdPutChar)
```

in which the second argument of `echo` is a token whose type permits the sequence of (`=>>`) operations to be invoked. When this module has been loaded, the expression `echo ``alpha.txt'' stdOI` will then open a file of the given name, if one is present, read a single character from the file and print it to the standard output file.

The `OI` package should not be taken to indicate how comonadic `IO` operations should be implemented in a compiler, but only to show some operations and their types. The coercion `unsafePerformIO` effectively performs a *synchIO* operation on each *IO* transaction. If the `OI` module were implemented directly from system primitives, rather than in terms of Haskell's monadic *IO* primitives, it would not be necessary to design operators that synchronize the *IO*

system to this degree.

Another concession to expediency is definition of a constant, `stdOI`, with the polymorphic type `OI a`. This was needed to provide an initial argument to the `(=>>)` operator. However, the presence of `stdOI` as a constant makes the encapsulation of *IO* operations by the `OI` comonad unsafe, as it allows the application of an operation of type `OI a -> b` to produce a side-effecting computation whose type, `b`, does not indicate that it may have such an effect. As mentioned earlier, a top-level program should provide a unique, initial application of `(=>>)` to enforce that *IO* effects are associated only with expressions typed in the `OI` comonad.

3.2.6 The COM comonad

This comonad is a bit more imaginative than are the others discussed in this paper, yet it seems consistent with the design goals of dynamically linked, component-based programming. A software component is an object characterized by a finite set of access methods whose types are given in a signature. The representation of a component is hidden. This allows any object with the same signature at its interface to be substituted for a given component.

Dynamic linking permits a component to be imported from the evaluation context of a program in execution, giving the program access to the methods of the components it imports. Dually, a program might export some objects, along with the signatures of their visible access methods, as new components. The mechanisms that support importation and exportation of components should be polymorphic; furthermore they should be related in ways that make the use of components uniform. If sufficiently well behaved, these mechanisms can constitute the data of an instance of a comonad.

A comonad of components is described as:

$$
\begin{aligned}
&COM\,a = \langle \mathit{hidden\ representation} \rangle \\
&\mathit{co\text{-}eval} = \mathit{getInterface} \\
&\mathit{co\text{-}ext}\,f = (\mathit{mapCOM}\,f)\,.\,\mathit{buildInterface} \\
&\text{where } f :: COM\,a \to b
\end{aligned}
$$

The method *getInterface* is supported by every *COM* object and it always succeeds, yielding the typed interface of the object[5].

[5]The reader may notice a similarity between the *COM* comonad and Microsoft's *Common Object Module* standard. While there are similarities, there are also significant differences. Most significantly, a Microsoft COM object is not typed with the signature of an interface, as we have assumed

for objects of the COM comonad. Instead, an MS COM object can be queried for an interface known to the client program, which may be one of several supported by the object. If the object possesses an interface matching the query, a pointer to that interface is returned; otherwise the query signals its failure.

The function $buildInterface$ is polymorphic for objects typed in the COM comonad, i.e.

$$buildInterface :: COM\ a \to COM\ (COM\ a)$$

and may also be defined at some non-COM types. When $buildInterface$ is applied to an instance of an object, it constructs a COM interface, including procedures for marshalling and unmarshalling parameters of the types declared in the object's method signature. Marshalling is only defined on types for which the respresentations of values are finitary. Marshalling is a strict operation.

The operations $getInterface$ and $buildInterface$ are related by:

$$getInterface \circ buildInterface = id$$

The signature of an object of type $COM\ a$ contains only the method $getInterface$. The result returned by this method is an interface whose type is a signature. The action of $buildInterface$ is uniform at all signa-

ture types. At other than *COM* types, the action of *buildInterface* may be specific to the type of its argument.

A function of type $COM\ a \to b$ is able to access a *COM* object passed to it as an argument but unable to dynamically create *COM* objects out of the values it returns. Such a function is transformed by the *co-ext* combinator of this comonad into a fully *COM*-capable function that wraps its results in *COM* interfaces.

4 Conclusions

This paper suggests that Haskell could have codata types declared via its module system if it incorporated a previously suggested extension to its type system: restricted rank 2 polymorphism. This extension presently exists in Hugs98. The codata type most familiar to Haskell programmers is probably the type (constructor) of streams, simulated in Haskell programs by unbounded lists, lazily evaluated. With an abstract codata type, however, its representation can be hidden. Also, the evaluated segments of a codata object are not cached by defaut.

Although streams are the only types of codata given as examples in this paper, other examples are

not difficult to imagine. In particular, abstract data types, as they are usually construed in a functional programming language, fit the definition of codata. Representation hiding is an essential aspect of abstract data types and should be supported in Haskell. Restricted rank 2 polymorphism would suffice.

A second suggestion is to provide comonads in Haskell by including a Comonad class in the standard prelude, and providing syntax for a top-level program to apply the co-extension of a function typed in a comonad to an implicit argument. This extension would support comonads whose operational semantics are defined in an execution context.

We have argued that comonads provide a natural abstraction of the effects that arise in the context of a program, such as input/output, component interfaces and parallel evaluation. There are other examples of comonads which have not been discussed in this paper because they are less compelling as potential extensions of Haskell. These examples include concurrent communicating processes, exceptions and explicit continuations.

Acknowledgments
The author has been influenced by discussions of the material presented here with many persons, including Jeremy Gibbons, John Hughes, Mark Jones, John

Launchbury, Lennart Augustsson, Eugenio Moggi, and to Ross Paterson, who pointed out some deficiencies in an earlier version of the paper.

References

[Bac78] John Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[Jac95] Bart Jacobs. Mongruences and cofree algebras. In *AMAST'95*, volume 936 of *Lecture Notes in Computer Science*, pages 245–260. Springer Verlag, July 1995.

[Jac96a] Bart Jacobs. Coalgebraic specifications and models of deterministic hybrid systems. In *AMAST'96*, volume 1101 of *Lecture Notes in Computer Science*, pages 520–535. Springer Verlag, July 1996.

[Jac96b] Bart Jacobs. Inheritance and cofree constructions. In *ECOOP'96*, volume 1098 of *Lecture Notes in Computer Science*, pages 210–231. Springer Verlag, July 1996.

[Jon99] Simon Peyton Jones. *Explicit quantification in Haskell.* URL: research.microsoft.com/users/simonpj/Haskell/quantification.html, 1999.

[Kie98] Richard B. Kieburtz. Taming effects with monadic typing. In *Proc. of 1998 ACM/SIGPLAN International Conference on Functional Programming*, pages 51–62. ACM Press, September 1998.

[LLC99] John Launchbury, Jeff Lewis, and Byron Cook. On embedding a microarchitectural design language within Haskell. In *Proc. of 1999 ACM/SIGPLAN International Conference on Functional Programming*, page (to appear). ACM Press, September 1999.

[LO94] Konstantin Läufer and Martin Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems*, 16(5):1411–1430, September 1994.

[LP95] John Launchbury and Simon Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, pages 293–351, 1995.

[Mog91] Eugenio Moggi. Notions of computations

and monads. *Information and Computation*, 93(1):55–92, July 1991.

[Wad92] Philip Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–14. ACM Press, January 1992.