

**Technical Report TR03-018**

Department of Computer Science  
Univ. of North Carolina at Chapel Hill

**Federating Programs Artfully with DeCo**

Dean Herington and David Stotts

Department of Computer Science  
University of North Carolina  
Chapel Hill, NC 27599-3175

[stotts@cs.unc.edu](mailto:stotts@cs.unc.edu)

May 23, 2003

# Federating Programs Artfully with DeCo

Dean Herington  
University of North Carolina at Chapel Hill  
Campus Box 3175, Sitterson Hall  
Chapel Hill, N.C. 27599-3175 USA  
heringto@cs.unc.edu

David Stotts  
University of North Carolina at Chapel Hill  
Campus Box 3175, Sitterson Hall  
Chapel Hill, N.C. 27599-3175 USA  
stotts@cs.unc.edu

## ABSTRACT

Program federation is assembling a software system from cooperating but independent application programs. We present *DeCo*, a declarative framework for specifying and executing federations. Participating programs and data files are described in the functional language Haskell, extended with operations for large-grain program description and coordination. The declarative expression of a federation in terms of data flow among the component programs captures synchronization requirements implicitly and exploits the inherent concurrency automatically.

DeCo's construction as a domain-specific language embedded in Haskell confers many benefits. High-level features (especially monads, higher-order functions, polymorphism, and type classes) allow the federation language to be simultaneously concise, natural, and powerful. Lightweight concurrency makes the data-flow approach feasible. Type checking ensures the static consistency of the federation, while type inference avoids the need for most type annotations. The full power of Haskell is available where necessary for adding function beyond that of the existing programs being federated.

DeCo is currently being used to federate two existing Fortran models that simulate environmental processes in the Neuse River estuary of North Carolina.

## Keywords

domain-specific embedded language, Haskell, program coordination

## 1. INTRODUCTION

We use the term *program federation* to refer to assembling a software system from cooperating but independent application programs. Combining existing large-scale components has several well-known benefits: reduced cost of construction, better modularity, greater concurrency, and increased

potential for reuse.

Manual programming is typically used to combine and coordinate the components of a program federation. While such a manual process can achieve some of the benefits of program federation, it fails to realize these benefits fully. Moreover, a manual process is unnecessarily tedious and error-prone.

We believe that adopting a more disciplined approach to program federation can produce higher-quality results with less work. The first step in this direction is to describe the components of a federation explicitly and more formally. Providing such *metadata* about the participating programs and data files has benefits at increasing levels of sophistication.

- It sharpens understanding of the components and documents their behavior.
- It enables the consistency of a proposed federation to be checked automatically.
- It allows for automatic mediation between components in the case of mismatch.

The second step is to raise the level of abstraction as much as possible. Put another way, the specification of a federation should be as declarative as possible. A declarative specification suppresses irrelevant detail, uses independent constructs that compose well, and minimizes the context dependence of its parts. The result tends to be a specification that is more concise, more natural, and more understandable, whose components are more interoperable and more reusable.

One good way to realize a highly declarative framework for program federation is to define and implement a domain-specific language embedded in Haskell [6]. Our experimental system *DeCo* (for Declarative Coordination) embodies such an approach. The DeCo system consists of approximately 2150 lines of Haskell (using common extensions to Haskell 98 [9]). It is built using the Glasgow Haskell Compiler and runs on the Linux operating system.

DeCo is targeted at scientific model federations suited to large-grain coordination, that is, where the unit of execution tends to be an entire invocation (possibly repeated many times) of a component program. The constituent models

tend to be existing application programs, usually large and written in imperative programming languages such as Fortran. They communicate, as whole programs, through files and operating system channels, rather than as subroutines via shared variables or message passing. The models deal with many, often large files containing data in a variety of formats.

Despite its bias toward scientific model federations, however, DeCo is suited to large-scale program federation more generally. In particular, it makes no assumption about the language(s) in which the external programs to be federated are written.

The remainder of this paper is structured as follows. Section 2 gives an overview of the architecture of the framework provided by DeCo. Sections 3 and 4 describe the data and control models in more detail. Section 5 treats some practical details of file and directory management. Section 6 discusses our experience to date with DeCo. Section 7 mentions related work. Section 8 concludes.

## 2. OVERVIEW

This section gives an overview of the DeCo architecture by introducing its main components. A fuller account of the semantic model is deferred to later sections.

The preexisting components from which a federation is constructed are application programs and data files. DeCo models these entities with the more abstract notions of *executor* and *stream* to present a simple yet rich semantic model for coordination. These abstract notions are realized as concrete Haskell entities in the Haskell code that constitutes the federation.

### 2.1 Stream

A stream is represented by the abstract datatype

```
newtype Data t
```

An entity of type (`Data t`) represents an aggregation of data of type `t`. The type of a stream is arbitrary, and thus so also is its size.

Note that a stream is not necessarily a sequence of elements. (`Data Char`) denotes a stream consisting of a single character, while (`Data [Char]`) denotes one consisting of a sequence of characters. The use of the term “stream” is intended to indicate that the stream’s contents are read and written in order from the beginning toward the end, not that the stream implicitly contains a sequence of elements. At the same time, the name `Data` is chosen for the abstract type to emphasize that a stream’s contents can be treated as a whole, as will be seen.

A stream’s contents can exist in one of three forms: as a Haskell *value*, as the contents of a *file*, or as the contents of a *channel* (the data read from or written to an operating system file descriptor). By making `Data` an abstract type, the framework allows the three forms of stream to be used interchangeably while accommodating different sorts of data connection between processes in an efficient manner.

### 2.2 Executor

An executor is represented by a Haskell function with abstract result type

```
newtype EX t
```

Having a result type of (`EX t`) allows a function to manipulate streams.

The most important kind of executor is one that serves as a proxy in the federation program for an existing external application program. Such an executor invokes a DeCo-supplied utility to start a subordinate process, having set up input and output connections appropriately. On the other hand, an executor may also be implemented purely in Haskell, creating no subordinate process. Such an executor might be used to transform stream contents from one set of types to another, for example. However an executor is implemented, it is used in the same way. Moreover, since an executor is simply a Haskell function, it is first-class: it may be higher-order, it may be partially applied, *etc.*

A stream *connection* represents a unidirectional data flow between executors. The executor providing the stream (and hence defining the contents of the stream) is termed its *producer*. The executor using the stream (and hence relying on the contents of the stream) is termed its *consumer*.

### 2.3 Metadata

Federation metadata—descriptions of the federation’s components—are expressed in Haskell. This fact is crucial to two of DeCo’s capabilities. First, it means that Haskell type checking ensures the static consistency of a federation. Second, it allows DeCo to mediate stream connections between executors where there exist discrepancies.

Expressing metadata in Haskell allows metadata to enjoy the characteristics of Haskell code, including its functional nature and its rich set of features for abstraction. In particular, the metadata for an application program are expressed as an executor, in other words, as a stylized Haskell function.

The use of Haskell-expressed metadata integrates well with the domain-specific embedded language approach used by DeCo. In fact, as will be seen, it is difficult (and fortunately pointless) to distinguish sharply between metadata and federation code.

### 2.4 Federation

The expression of the federation itself is also given in Haskell. A federation is cast simply as an executor that interconnects preexisting external components, using streams, other executors, and metadata.

The abstract datatype (`EX t`) is an extension of Haskell’s built-in type (`IO t`). Hence, an executor, whose result type is (`EX t`), may perform such side effects as part of its execution.

```
io :: IO a -> EX a
```

This ability is crucial for executors that serve as proxies for application programs, as they need to create processes, files, and directories.

A federation is executed with

```
runEX :: [String] -> EX a -> IO a
```

The function `runEX` is passed a list of option strings and an executor. The executor is executed and its result returned.

The design of the executor abstraction—in particular, that an executor can serve as a proxy for an application program—has another advantage. A federation expressed as a Haskell program can be used as an application program in a yet larger federation. Thus, federations are appropriately compositional.

### 3. DATA MODEL

To allow for widely varying data storage formats while providing maximum flexibility for data streams, DeCo defines a two-level typing scheme.

- The (abstract or high-level) type of a stream, expressed as a Haskell type, captures the high-level semantics of the stream data.
- The (concrete or low-level) representation of a stream, encoded separately, specifies the storage format of stream data in a particular context.

This two-level scheme separates the essential type of data from its packaging as a stream. The separation is akin to the difference between abstract and concrete syntax in a programming language.

The two-level approach to stream typing provides several key benefits.

- DeCo’s notion of stream compatibility is simple and clear, being defined in terms of Haskell types.
- Stream compatibility is very general, as it is defined to ignore matters of representation.
- DeCo can automatically mediate between a stream producer and its consumer when the produced and consumed representations differ.

Recall that a stream has the Haskell abstract type `(Data t)`, for some `t`. The type parameter `t` exactly encodes the high-level type of the stream. As a result, Haskell’s type checking ensures that streams are used in a type-safe manner. Moreover, Haskell’s type inferencing relieves the federation programmer in most cases from the need to declare stream types explicitly.

At the level of data storage, the contents of a stream are deemed to consist of a sequence of bytes. The low-level representation of a stream is cast as a translation between

```
-- universal representation for a 'Data' repr
data ReprRep = ReprRep [String] [ReprRep]
  deriving (Show)

-- declare a 'Data' repr
class (Show r) => Repr r where
  reprRep :: r -> ReprRep

-- equality of two 'Data' reprs
reprEq :: (Repr r1, Repr r2) => r1 -> r2 -> Bool
reprEq r1 r2 = reprRep r1 == reprRep r2

-- associate a representation and a type
class (Repr r) => ReprType r t where
  putData :: r -> t -> DW () -- encode function
  gmbData :: r -> DR (Maybe t) -- decode function

-- decode function that fails at end-of-stream
getData :: (ReprType r t) => r -> DR t
```

Figure 1: Expressing Representations for Types

a stream of bytes and a Haskell value of the appropriate type. A decoding function translates a stream of bytes to a Haskell value. An encoding function translates a Haskell value to a stream of bytes.

An important property of (low-level) representations is that there may be more than one of them for a given (high-level) type. The alternative representations for a type are expressed as distinct Haskell types that are instances of a type class `Repr r`. A representation type `r` is associated with a Haskell type `t` by an instance declaration for `ReprType r t`. In this way, both new representations and new associations between representations and types can easily be defined. Figure 1 defines the essential mechanism for expressing data representations. (`DW` and `DR`, not shown, are monads for “data writing” and “data reading” that manage the book-keeping for stream data access.)

A decoding function returns a type wrapped by `Maybe` to signal stream exhaustion: `Just v` indicates that a value `v` of the desired type was extracted from the stream at its current position and properly decoded, while `Nothing` indicates that the end of the stream was encountered instead. This behavior allows for the very convenient use of end-of-stream to delimit a sequence of items.

The function `getData` is an alternate decoding function used when end-of-stream is not permitted. It is defined in terms of `gmbData` and implicitly evokes an error if the stream contents have been exhausted.

We show two examples of representation definitions. Figure 2 shows a representation for unsigned integers that applies to types `Integer` and `Int`. Figure 3 gives an example of a compound representation. It shows the definition of `TailSeq r`, which is a representation for a sequence of items of like type and representation, where the end of the stream marks the end of the sequence. Such a sequence is called a *tail sequence* because the sequence uses up the “tail” of the stream. The parameter `r` gives the representation for each

```

data Endian = LE | BE    -- little- or big-endian
  deriving (Show)

-- unsigned integer repr (length is in bits)
data UInt = UInt Endian Int
  deriving (Show)

instance Repr UInt where
  reprRep (UInt e l) =
    ReprRep ["UInt", show e, show l] []

instance FollowableRepr UInt

instance ReprType UInt Integer where
  putData (UInt e l) x =
    putFixed (encodeBinary l) e l x
  gmbData (UInt e l) =
    gmbFixed decodeBinary e l

instance ReprType UInt Int where
  putData (UInt e l) x =
    putFixed (encodeBinary l . toInteger) e l x
  gmbData (UInt e l) =
    gmbFixed (fmap fromInteger . decodeBinary) e l

-- types for utility functions
encodeBinary :: Int -> Integer -> DW [Byte]
decodeBinary :: [Byte] -> DR Integer
putFixed :: (t -> DW [Byte]) -> Endian -> Int
           -> t -> DW ()
gmbFixed :: ([Byte] -> DR t) -> Endian -> Int
           -> DR (Maybe t)

```

**Figure 2: Unsigned Integer Representation**

sequence item. The most typical use of the representation, that for a list, is also shown.

There is a subtlety concerning `TailSeq` that illustrates the power of Haskell’s type system and its use in DeCo. Because a tail sequence uses up the remainder of a stream, it is not possible for any other data to follow a tail sequence in a stream. DeCo enforces this constraint *at compilation time* by embedding it in the Haskell type system. A representation is declared as *followable* if it can be followed by more data in a stream.

```
class (Repr r) => FollowableRepr r
```

All common representations, with the notable exception of `TailSeq r`, are declared to be followable. Where a representation occurs in a stream followed by other data, the representation is required to be followable. Hence, the representation

```
TailSeq (UInt BE 16)
```

is valid (and represents a tail sequence of 16-bit unsigned integers in big-endian order), whereas the representation

```

data (FollowableRepr r) => TailSeq r = TailSeq r
  deriving (Show)

instance (FollowableRepr r) => Repr (TailSeq r)
  where reprRep (TailSeq r) =
        ReprRep ["TailSeq"] [reprRep r]

instance (FollowableRepr r, ReprType r t) =>
  ReprType (TailSeq r) [t] where
  putData (TailSeq r) xs = mapM_ (putData r) xs
  gmbData (TailSeq r) = do xs <- many
    return (Just xs)

  where
  many = do mx <- gmbData r
    case mx of Nothing -> return []
               Just x -> do xs <- many
    return (x:xs)

```

**Figure 3: Tail Sequence Representation**

```
TailSeq (TailSeq (UInt BE 16))
```

is invalid, resulting in the following compilation error message:

```
No instance for (FollowableRepr (TailSeq UInt))
```

DeCo predefines a number of other representations, including those for common scalar types, counted sequences, and tuple types. As can be seen from the examples shown, it is straightforward to extend this set as needed.

## 4. CONTROL MODEL

A federation is essentially expressed as a directed data-flow graph, where the nodes are executor invocations (*executions*) and the edges are stream connections among them. Federation control flow—that is, the temporal sequence of executions—is derived automatically from the data flow. The characteristics of the stream connections among executions imply the appropriate synchronization among those executions and allow DeCo to realize the inherent concurrency of the federation automatically. The complexities of this data-flow machinery are hidden from the federation programmer by the abstract type `(EX t)`. We will have more to say about this when we return to executors in Section 4.2; first we treat streams in greater detail.

### 4.1 Stream design

Control flow complexities are also hidden by the abstract type `(Data t)`. In certain situations, it is essential that a stream be accessed incrementally. That is, it must not be necessary for later portions of a stream to be generated by its producer before earlier portions of that stream may be consumed by another concurrent execution. For example, if a stream connecting two executors could be of unbounded size, it may not be acceptable for the consumer to wait for the entire stream to be produced before starting to consume it. Similarly, if the stream connecting two executors is not of unbounded size but rather subject to unbounded delay

during its production, it may not be acceptable for the consumer to wait for the entire stream.

At the same time, it is useful to treat the contents of a stream in its entirety, as a single Haskell datum. Stream processing—especially data transformation—is greatly simplified if the entire contents of a stream can be directly pattern matched, passed among functions, mapped over, *etc.* Eliminating the need for explicit, incremental manipulation of stream contents allows for a much more declarative treatment of stream data.

Fortunately, these two seemingly contradictory views of a stream can be reconciled by exploiting Haskell’s ability to read lazily. Access by an `EX` action to a stream’s contents is made with either `readFile` or `hGetContents`, as appropriate. The resulting string is, therefore, read lazily from the stream. In this way, DeCo is able to provide a fully declarative treatment of streams.

Recall from Section 2.1 that a stream may exist in one of three forms, as a value, file, or channel. Although a stream usually can and should be manipulated without regard to its form, this is not always possible. A federation must commit to the form of a stream where it originates and where it terminates, including at the interface to an external program. For this purpose DeCo provides types with which to describe the forms of a stream.

```
data File t      = forall r. ReprType r t =>
                  File FilePath r

data Channel t = forall r. ReprType r t =>
                  Channel Fdx r

data Value t     = Value t ()
```

- A `(File path repr :: File t)` represents a file with pathname `path` that contains (or will contain) data of type `t` in representation `repr`.
- A `(Channel fdx repr :: Channel t)` represents a channel with extended file descriptor `fdx` that contains (or will contain) data of type `t` in representation `repr`.
- A `(Value val fin :: Value t)` represents a value `val` with `finish` value `fin`. The finish value enables detection of “run-on” streams in the context of lazy reading of stream contents. Evaluating the finish value for a value stream strictly reads any remaining bytes corresponding to the value `val`, then raises an error if the stream contents continue past that point.

Thus, a stream, as represented by a `Data` value, carries a form and, except for the value form, a representation. A stream of a given type (`Data t` for some `t`) can be used wherever a stream of that type is needed, despite any difference in form and/or representation between the stream and the requirements of its use site. DeCo performs any mediation necessary to account for differences in form and/or representation. This automatic stream mediation is one of the main benefits of DeCo. It greatly improves the modularity and reusability of executors.

## 4.2 Executor design

We now come to the most complex part of DeCo, the design of an executor. In order to support the data-flow model of execution, each executor is implemented by a concurrent Haskell thread [10]. When an executor cannot immediately complete access to one of its input or output streams, that executor’s thread will pend. (This could occur because an input stream was not yet available, because the next byte of an input stream was not yet available, or because the buffer associated with an output stream was full.) However, other executors’ threads can continue executing. In fact, executor threads will generally schedule themselves according to the availability of stream data.

So far, so good; however, there is an additional complication. In order to effect an efficient connection between a stream’s producer and its consumer, it is necessary to know not only the form of the actual stream generated by the producer but also the form of stream desired by its consumer. But a new stream is created as the result of invoking its producer’s `EX` action, which necessarily completes before any consumer of the stream exists. Hence, we try to defer creating a stream connection until at least one consumer of the stream has become known. More precisely, we normally require an executor’s thread to wait until each of its output streams has been requested by at least one consumer before it commits to the actual forms for its output streams. Permitting an executor to commit to the forms for its output streams we call *triggering* that executor.

But what if one of an executor’s output streams is requested by no other executor? This situation will lead eventually to temporary deadlock, that is, no executors being able to proceed. When we detect temporary deadlock we forcibly trigger all current executors waiting to be triggered, and execution proceeds.

We can now explain the general structure of an executor. The *body* of an executor is the portion executed by a separate thread.

An executor performs these actions, generally in the order presented.

1. It registers a *request* for each stream it will consume. The request specifies the form of stream needed by the executor. The response to the request is a computation to be used in the body to receive the actual stream in the requested form.
2. It registers a *promise* for each stream it will produce. The promise specifies the form of stream that will be produced by the executor. The response to the promise is the `Data` object for the new stream along with a computation to be used in the body to fulfill the promise.
3. It registers the body. The body has its own structure, described next.
4. Finally, it returns its results, which include all the produced streams returned by its promises.

```

1  text = TailSeq (ASCII 8)
2
3  example :: Data [Int] -> Data String
4      -> EX (Data [Double], Data String)
5  example fileIn chanIn = executor $ do
6      fileIn' <- wantFile Nothing
7          (Just (TailSeq (UInt BE 32)))
8          fileIn
9  chanIn' <- wantChannel (Just text) chanIn
10 (fileOut, fileOut') <-
11     makeFile (Just "example.out")
12     (Just (TailSeq ws_sp))
13 (chanOut, chanOut') <-
14     makeChannelWant (Just text)
15 body $ \ bh -> do
16     File pathIn _ <- fileIn' Nothing Nothing
17     inChan <- chanIn' Nothing
18     awaitTrigger bh
19     (outChan, readyChan) <- chanOut' Nothing
20     process <- startProgram "example" False
21         False [pathIn] Nothing Nothing
22         [(0, inChan), (1, outChan)]
23     status <- awaitTermination process
24     checkStatus status
25     fileOut' Nothing Nothing
26     readyChan
27     return (fileOut, chanOut)

```

**Figure 4: Example Executor Definition**

The body of an executor performs these actions, in the order presented.

1. It awaits availability of each input stream by performing the computation received earlier with each input stream request. When each such computation returns, information about the associated input stream is made available to the body.
2. It waits to be triggered. Under normal circumstances, the body of an executor is not triggered until each of the executor's output streams has been requested at least once.
3. It supplies the necessary information for each output stream by performing the computation received earlier with each output stream promise.

Note that actions in the first of the two lists above should usually not pend, whereas actions in the second list may pend.

Consider the example definition of an executor in Figure 4. This example consists of correct and complete DeCo code. However, we will not describe every detail therein.

Executor `example` consumes two input streams. The first input, `fileIn`, of type `(Data [Int])`, is requested as a file of arbitrary pathname containing a tail sequence of 32-bit integers in big-endian order (lines 6–8). The second input, `chanIn`, of type `(Data String)`, is requested as a channel

containing plain text (line 9 and line 1). The executor produces two output streams. The first output, `fileOut`, of type `(Data [Double])`, is a file named `"example.out"` containing a tail sequence of whitespace-delimited double-precision values (lines 10–12). The second output, `chanOut`, of type `(Data String)`, is a channel to be created by DeCo containing plain text (lines 13–14). The body is established (lines 15–26) and the results are returned (line 27).

In the executor's body, the input file is awaited; on receipt, `pathIn` is bound to its pathname (line 16). The input channel is awaited; on receipt, `inChan` is bound to it (line 17). Having received all its inputs, the executor then awaits its trigger (line 18). After triggering, the output channel and a readying computation are received from DeCo (line 19). The program `"example"` is started, with the input file pathname as its sole argument and the input and output channels attached to standard input and output (lines 20–22). Program termination is awaited (line 23) and the termination status is checked (line 24). The output file (line 25) and output channel (line 26) are readied.

The example just described shows how executor metadata are provided procedurally and that wide variation is accommodated in a stylized format. Although an executor definition is somewhat complicated due to the staging required to support data-dependent scheduling of executor bodies, Haskell's type checking is of considerable help in managing this complexity. Note also how there is little that is extraneous in the definition.

Due to space considerations we omit the remaining details of executor definition. Fortunately, a typical federation requires only a small number of executor definitions, and they are rather formulaic. More often used are the higher-level operations described next.

### 4.3 Federation construction

This section defines some higher-level operations that are useful for constructing federations. The design of DeCo makes it easy to add new operations similar to those described here.

The first group below includes basic functions for producing and consuming streams. The stream producers, whose names begin with `"from"`, never pend. (The result `Data` value, which is a sort of stream handle, is produced immediately. Subsequent access to the stream contents may pend, of course.) The stream consumers, whose names begin with `"to"`, may pend, waiting for the contents of the stream to be available.

```
fromValue :: Value t -> EX (Data t)
```

`fromValue` makes a stream from a value stream specifier. This function does not pend.

```
toValue :: Data t -> EX (Value t)
```

`toValue` makes a value stream specifier from a stream. This function pends until reading of the stream can begin.

```
fromVal :: t -> EX (Data t)
```

`fromVal` makes a stream from a Haskell value, supplying a finish value whose evaluation simply succeeds. This function does not pend.

```
toVal :: Data t -> EX t
```

`toVal` extracts the Haskell value from a stream, first evaluating the enclosed finish value. This function pends until the entire stream has been read.

```
fromFile :: File t -> EX (Data t)
```

`fromFile` makes a file stream from a file specifier, assuming the file is ready. This function does not pend.

```
toFile :: File t -> Data t -> EX ()
```

`toFile` arranges for the contents of a stream to be written to a file. This function pends until the entire file has been written.

```
fromChannel :: Channel t -> EX (Data t)
```

`fromChannel` makes a channel stream from a channel specifier, assuming the channel specifier's extended file descriptor is ready for reading. This function does not pend.

```
toChannel :: Channel t -> Data t -> EX ()
```

`toChannel` arranges for the contents of a stream to be written to a channel. This function does not pend. The stream contents are written to the channel as the contents become available, and the channel is closed on completion of writing.

Sometimes a stream connection does not naturally involve a file (that is, the stream is neither produced nor consumed as a file), yet it is useful to record the contents of the stream in a file. Perhaps the contents need to be accessed again later, either during federation execution or after the federation completes. Perhaps serialization is needed—that is, the stream consumer must wait to begin execution until the stream producer has completed execution. These requirements can be met with the following function.

```
viaFile :: File t -> Data t -> EX (Data t)
```

`viaFile` arranges to write the contents of a stream to a file and returns a file stream for that file.

```
viaFile file stream
```

```
class From c where
  from :: c t -> EX (Data t)
```

```
instance From Value where
  from = fromValue
```

```
instance From File where
  from = fromFile
```

```
instance From Channel where
  from = fromChannel
```

```
class To c where
  to :: c t -> Data t -> EX ()
```

```
instance To File where
  to = toFile
```

```
instance To Channel where
  to = toChannel
```

```
via = viaFile
```

### Figure 5: Implementing the `from`, `to`, and `via` Shorthands

is equivalent to

```
do toFile file stream
  fromFile file
```

except that in the former, waiting for the entire file to be written is delayed until the result file stream is demanded, whereas in the latter the waiting occurs as part of the construct (due to the use of `toFile`).

Convenient shorthands are provided for some of the most common of the operations described above. Use of Haskell type classes (see Figure 5) provides an overloaded function `from` that can be used in place of `fromValue`, `fromFile`, and `fromChannel`, and an overloaded function `to` that can be used in place of `toFile` and `toChannel`, plus a simple synonym `via` for `viaFile`.

For the remaining operations described in this section, we include implementations as further illustrations of executor definition.

Figure 6 gives the definition of `programFilter`, which can be used to define an executor that serves as a proxy for a simple Unix filter program. The external program is defined by its pathname (`path`) and arguments (`args`). The executor hooks its input stream (`input`) to the program's standard input (using representation `ir`) and the program's standard output (using representation `or`) to its output stream.

The last two operations are examples of higher-order operations; they take executors as arguments.

```

programFilter :: (ReprType ir String,
                 ReprType or String)
              => String -> [String] -> ir -> or
              -> Data String -> EX (Data String)
programFilter path args ir or input = executor $ do
  ichan <- wantChannel (Just ir) input
  (output, ochan) <- makeChannelWant (Just or)
  body $ \ bh -> do
    ich <- ichan Nothing
    awaitTrigger bh
    (och, oready) <- ochan Nothing
    proc <- startProgram path True True args
              Nothing Nothing
              [(0, ich), (1, och)]
    oready
    awaitTermination proc >>= checkStatus
  return output

```

Figure 6: programFilter

```

mapEX :: (Data a -> EX (Data b)) -> Data [a]
      -> EX (Data [b])
mapEX forEach input = executor $ do
  inValue <- wantValue input
  (output, outValue) <- makeValue Nothing
  body $ \ bh -> do
    Value inVal inFin <- inValue
    awaitTrigger bh
    outValues <- mapM each inVal
    let (outVals, outFins) =
          unzip [ (val, fin)
                 | Value val fin <- outValues ]
        outFin = inFin 'seq' mergeFins outFins
    outValue (Just (Value outVals outFin))
  return output
where each a = fromVal a >>= forEach >>= toValue

```

Figure 7: mapEX

Figure 7 shows `mapEX`, which is similar to Haskell’s `mapM` function but specialized to the `EX` monad.

Similarly, Figure 8 shows `foldEX`, which is similar to Haskell’s `foldM` function but specialized to the `EX` monad.

## 5. FILES AND DIRECTORIES

During the execution of a federation, three directories are maintained by DeCo.

- The *top directory* is the directory that was current when the federation began execution. The top directory remains fixed for the duration of federation execution. It serves as a reference point within the invoker’s directory environment.
- The *run directory* is a new directory created when the federation begins execution. It is the root of a directory subtree that serves as a repository for new files created by the federation. The run directory name and location are under control of the invoker of the federa-

```

foldEX :: (Data a -> Data b -> EX (Data a))
        -> Data a -> Data [b] -> EX (Data a)
foldEX forEach initial elems = executor $ do
  elemsValue <- wantValue elems
  (output, outValue) <- makeValue Nothing
  body $ \ bh -> do
    Value elemsVal elemsFin <- elemsValue
    awaitTrigger bh
    Value finalVal finalFin <-
      foldM each initial elemsVal >>= toValue
    let outFin = elemsFin 'seq' finalFin
        outValue (Just (Value finalVal outFin))
    return output
  where
    each input elem = fromVal elem >>= forEach input

```

Figure 8: foldEX

tion. The run directory remains fixed for the duration of federation execution.

- The *current directory* is a directory within the subtree rooted at the run directory that associates a portion of that subtree with the currently executing federation code. The current directory starts out equal to the run directory but may vary under control of federation code. It defines a “directory scope” during execution; in particular, it provides the default initial directory for external program invocations.

The operating system maintains a “current working directory” for a process that may vary during process execution. Varying the current working directory during federation execution, however, would lead to unpredictable results, because executors are implemented as concurrent Haskell threads. Instead, DeCo leaves the actual current working directory unchanged and provides a virtual one (the current directory introduced above) that works properly in the presence of multiple threads.

Federation code manages the current directory with `inDir`.

```
inDir :: FilePath -> EX a -> EX a
```

(`inDir dir act`) creates a directory named `dir` in the current directory, then performs `act` with `dir` as the current directory. In other words, `inDir` opens a new, temporary, directory scope for the execution of a subordinate action. Note that, although the current directory reverts after the subordinate action completes, the file subtree rooted at the newly created directory persists.

DeCo’s interpretation of pathnames is extended to provide access to the top, run, and current directories, as shown in the following table. The remainder of a pathname with the indicated initial character is interpreted relative to the corresponding directory.

@	top
#	run
\$	current

In addition, a pathname beginning with a / character is interpreted as usual, whereas a pathname beginning with a character other than these four (@ # \$ /) is interpreted relative to the top directory (as if it were preceded by @).

## 6. DISCUSSION

In this section we give preliminary assessments of our experience using and implementing DeCo.

### 6.1 Case study experience

We have applied DeCo to a realistically complex case study in order to evaluate its effectiveness. The case study involves the federation of two existing environmental models for aspects of the Neuse River estuary in eastern North Carolina. The first models estuary water quality through time, given initial concentrations, inflow rates, and outflow rates of water constituents, plus meteorological data for the modeled time period. The second models chemical processes in the sediment underlying the river, computing fluxes of constituents between water and sediment.

The water model is a single program of approximately 9300 lines. The sediment model consists of two programs, whose total size is approximately 4700 lines. Both models are written in Fortran 77. Together the two models read and write dozens of files during their execution.

The goal of combining these two models is to obtain a more precise simulation of the physical, chemical, and biological processes occurring in the Neuse River estuary. When run separately, each model makes simple assumptions about the other's medium: the water model about the sediment, and the sediment model about the water. In the federation, each model provides a more sophisticated simulation of its medium for the other model.

Few modifications needed to be made to the Fortran models to get them to run under control of DeCo. For the water model, a very small effort (approximately 30 lines added or changed) was required by the maintainer of the program to add the capability to read constituent fluxes at program initiation. For the sediment model, the two programs needed to be enhanced to run in batch mode, as they were originally written to read input parameters from the keyboard. (These changes involved many more lines, but they were mostly the same changes repeated dozens of times.)

The challenging part of federating these models was resolving mismatches of four types:

- Spatial mismatch. The water model divides the studied portion of the Neuse River estuary into 59 *segments*, whereas the sediment model divides the same portion of the estuary into 4 *regions*. This mismatch was resolved by interposing executors to average data passed from the water model to the sediment model and replicate data passed in the other direction.
- Temporal mismatch. The water model runs with much smaller time steps than the sediment model. This mismatch was resolved by interposing an executor to average the data passed from the water model to the sediment model.

- Units mismatch. Some parameters common to the models are represented in different units ( $g/m^2/d$  versus  $mmol/m^2/d$ , for example). Simple scaling by existing executors resolved these mismatches.
- Data format mismatch. Some parameters common to the models are represented in different formats in their respective files. DeCo representation specifications handled these mismatches.

The use of DeCo in the Neuse River case study has been both pleasant and effective. The separate specification of stream type and stream representation copes with varying data formats while allowing streams to be treated abstractly. The abstraction of the form of a stream's content (as file, channel, or value) successfully supports the data-flow approach while allowing (in cooperation with lazy stream reading) the contents of a stream to be treated as a whole. Synchronization of component programs according to their mutual data flows makes for a highly declarative expression of control flow. Simple but declarative DeCo features reduce the burden of file and directory management to a minimum. Overall, these features enhance interoperability and reusability for federation components and make DeCo effective at automating program federation.

Based on the Neuse River case study alone, a conclusion cannot yet be drawn concerning conciseness and efficiency of program federation using DeCo. The Neuse River program federation consists of approximately 850 lines of code. Although this may seem like a large amount of code for coordinating the multiple executions of three Fortran programs, it should be noted that these programs are executed repeatedly in alternation, that their data formats are rather complex, and that considerable data manipulation is required to combine the programs. The very high code density of the program federation suggests that its size is not excessive for the problem.

A current shortcoming in the Glasgow Haskell Compiler's runtime system has prevented us from precisely comparing the time spent executing the federation itself to the time spent in the federation's subordinate program executions. However, coarse wall-clock timing during execution of the case study shows that the executions of the subordinate Fortran programs heavily dominate. Although more precise measurement is desirable, DeCo performance is clearly not a limiting factor in the Neuse River case study.

### 6.2 Use of Haskell

The use of Haskell has been very positive, both as a base for the program federation language and as a base for the implementation of DeCo.

As the base for the federation language, Haskell offers a highly declarative foundation that greatly eases construction of a declarative domain-specific language. Haskell's abstraction capabilities—especially monads, higher-order functions, polymorphism, and type classes—allow the federation language to be both simple and powerful. Haskell's strong typing provides static consistency checking for federation programs, while its capable type inferencing greatly reduces the number of type annotations needed. Haskell's power and

expressiveness make it easy for a federation programmer to provide the “connective tissue” that is inevitably required in federating existing components.

As the base for implementation of DeCo, Haskell has also been very successful. First, all of the advantages cited above concerning its use as a base for the federation language pertain to its use as a base for framework implementation. More specifically, however, the power, simplicity, and efficiency of Haskell’s concurrency (threads) support made design of DeCo not only feasible but also elegant and relatively easy. The availability of an interface to Posix capabilities made programming the external interactions of DeCo quite comfortable. Finally, the expert and willing assistance of the volunteers who build, maintain, and use the Glasgow Haskell Compiler system was invaluable.

## 7. RELATED WORK

Three strains of prior research are most relevant to our work on DeCo: coordination languages, domain-specific embedded languages, and functional shells and scripting languages.

*Coordination languages* (also called *configuration languages* or *module interconnection languages*) aim to provide a framework in which to express an application as an aggregation of components. At this high level, DeCo’s goal is the same. However, there are significant differences in emphasis. Coordination languages tend to focus on issues of distribution and finer-grained parallelism, whereas DeCo focuses on expressing components abstractly to facilitate adaptation, composability, and reuse. In addition, coordination languages are usually deliberately distinct from the computation languages in which the federated components are written, whereas DeCo exploits the fact that Haskell can be used not only for coordination but also for as much computation as is useful for a given federation (for data adaptation, for example). Examples of coordination languages include Polyolith [11], Strand and PCL [4], and Linda [2].

A *domain-specific language (DSL)* is a language tailored to a particular application domain. A *domain-specific embedded language (DSEL)* is a DSL built as an extension to an existing *base* language. DeCo is constructed as a DSEL based on Haskell whose domain is program federation. As such it is part of a recent trend toward basing DSELS on Haskell [6]. Examples of other application domains (and representative DSELS) for which Haskell-based DSELS have been built include web programming (HaXml [15] and WASH/CGI [13]), hardware description (Lava [1] and Hawk [7]), animation (Fran [3]), and robotics (Frob [8]).

Shells and scripting languages share with coordination languages the high-level aim of facilitating aggregation of existing computational components, though their style is that of a more traditional programming language. Although one tends to think of a *shell language* as interactive and a *scripting language* as batch-oriented, the two notions are essentially similar, and both can be used in both ways. DeCo can be viewed as a scripting language for federations of applications. DeCo is not intended particularly for interactive use, but it can be used that way, and could easily be extended to be more convenient for such use. Shells and scripting languages are numerous; however, many fewer are especially

functional in nature as is DeCo. Examples of functional shells and scripting languages include *Es* [5], *scsh* [12], and the shell included in Famke [14], a prototype of a strongly typed operating system.

## 8. CONCLUSIONS AND FUTURE WORK

We have presented DeCo, a framework for large-scale program federation. Specifications in DeCo are concise, convenient, and highly declarative.

With DeCo, data are treated abstractly as *streams*, for which the data type, data representation, and form (as value, file, or operating system channel) are specified separately. The type gives the high-level semantics of the stream and is checked by Haskell. The representation gives the low-level encoding of the stream. The set of representations is easily extended to handle formats particular to a federation. The various forms of stream content facilitate reuse of streams in different contexts. DeCo automatically resolves mismatches between the actual and desired representations and/or forms of a stream.

External programs and Haskell functions are treated similarly and abstractly as *executors*. Haskell type checking ensures that executors are combined properly. Control flow is derived from the data flow among executors rather than being specified explicitly. Hence, DeCo automatically synchronizes the execution of external programs and realizes the inherent concurrency of a federation. DeCo also provides constructs to simplify the bookkeeping aspects of program federation, such as file and directory management.

An important characteristic that distinguishes DeCo from other coordination frameworks is that its basic entities—streams and executors—are composable. That is, larger, more complex entities can be built simply and predictably from smaller entities. Along with the abstract nature of streams and executors, this composability makes federation components more flexible, more interoperable, and more reusable.

DeCo has been applied to a realistically complex case study, a federation of existing environmental models for the Neuse River of North Carolina. The experience has shown that federation specification in DeCo can be comfortable and effective. However, more application experience is needed to substantiate such a conclusion on a usefully wide range of model federations. We plan, therefore, to apply DeCo to a number of federations with varying characteristics.

DeCo’s advantages would have been difficult or impossible to achieve in combination were it not based on a high-level, declarative language like Haskell and a robust implementation such as the Glasgow Haskell Compiler. Our experience with DeCo reinforces our belief that functional languages in general—and Haskell in particular—are highly suitable bases on which to build domain-specific embedded languages.

## 9. ACKNOWLEDGMENTS

This work was supported by the U.S. Environmental Protection Agency, under grant R82-795901-3.

## 10. REFERENCES

- [1] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *International Conference on Functional Programming*, pages 174–184, 1998.
- [2] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.
- [3] C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, volume 32(8), pages 263–273, 1997.
- [4] I. Foster. Compositional parallel programming languages. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(4):454–476, 1996.
- [5] P. Haahr and B. Rakitzis. Es: A shell with higher-order functions. In *USENIX Winter*, pages 51–60, 1993.
- [6] P. Hudak. Modular domain specific languages and tools. In P. Devanbu and J. Poulin, editors, *Proceedings: Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.
- [7] J. Matthews, B. Cook, and J. Launchbury. Microprocessor specification in Hawk. In *International Conference on Computer Languages*, pages 90–101, 1998.
- [8] J. Peterson, P. Hudak, and C. Elliott. Lambda in motion: Controlling robots with Haskell. *Lecture Notes in Computer Science*, 1551:91–105, 1999.
- [9] S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Apr. 2003.
- [10] S. Peyton Jones et al. Control.Concurrent. <http://www.haskell.org/ghc/docs/latest/html/base/Control.Concurrent.html>.
- [11] J. M. Purtilo. The POLYLITH software bus. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(1):151–174, 1994.
- [12] O. Shivers. A Scheme shell. Technical Report MIT/LCS/TR-635, Massachusetts Institute of Technology, 1994.
- [13] P. Thiemann. WASH/CGI: Server-side web scripting with sessions and typed, compositional forms. In *Practical Aspects of Declarative Languages*, pages 192–208, 2002.
- [14] A. van Weelden and R. Plasmeijer. Towards a strongly typed functional operating system. In *Selected Papers Proceedings 14th International Workshop on the Implementation of Functional Languages, IFL 2002*, Madrid, Spain, 2002.
- [15] M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation? In P. Lee, editor, *Proc. international conference on functional programming 1999*, pages 148–259, New York, NY, 1999. ACM Press.