Workflow representation and runtime based on lazy functional streams

Matthew J. Sottile, Geoffrey C. Hulette, and Allen D. Malony
{matt,ghulette,malony}@cs.uoregon.edu

Department of Computer and Information Science
University of Oregon
Eugene, OR 97403

ABSTRACT

Workflows are a successful model for building both distributed and tightly-coupled programs based on a dataflow-oriented coordination of computations. Multiple programming languages have been proposed to represent workflow-based programs in the past. In this paper, we discuss a representation of workflows based on lazy functional streams implemented in the strongly typed language Haskell. Our intent is to demonstrate that streams are an expressive intermediate representation for higher-level workflow languages. By embedding our stream-based workflow representation in a language such as Haskell, we also gain with minimal effort the strong type system provided by the base language, the rich library of built-in functional primitives, and most recently, rich support for managing concurrency at the language level.

1. INTRODUCTION

Workflows have emerged as a successful programming model for building both tightly-coupled and distributed systems out of existing software components. There exists a large and growing ecosystem of programming languages, runtime systems, and reusable components that support the workflow model. There are several languages, such as WOOL [7] and AGWL [3], that provide a high-level syntax for describing workflows as a set of activities whose inputs and outputs are interconnected by channels of data. It is a challenge to translate this fairly abstract representation down into an executable program. Here, we describe a general-purpose executable representation of workflows, based on lazy functional streams. Our intent is to provide a well defined representation of workflows based on functional streams, allowing this representation to be the target of higher level workflow language compilers.

In particular, we leverage the functional programming language Haskell to encode workflows as a composition of functions on streams. This representation is both concise and powerful, with well-defined semantics. We get a number of features from Haskell for free, including sophisticated static analysis and type checking, an efficient binding to C routines, and portability to a wide range of platforms. Furthermore, recent additions to the Haskell language for multi-threaded applications have proven to be useful for managing side-effects of stream operations while remaining in a purely functional setting.

We will first introduce (or refresh) the reader to the concept of lazy functional streams, and describe how computational activities within workflows can be lifted into the streaming model. A set of stream-based composition and control primitives will be presented in order to support sophisticated structures in workflow-based programs. Finally, we will discuss and demonstrate a few example workflows that can be concisely represented using streams.

WORKS 09, November 15, 2009, Portland Oregon, USA

2. RELATED WORK

There are many approaches to encoding workflows as executable programs.

Kepler [1, 11] is a system for composing scientific workflows. Its execution model is taken from Ptolemy II [8]. Ptolemy II is based on a generalized actor model, where actors represent computations, and have inputs and outputs that are connected via uni-directional channels. The semantics of the workflow are configurable. For example, one workflow may employ the Process Network model [10], where writes to a channel are non-blocking, reads from a channel will block if there is no available data, and an actor will "fire" only in the event that all inputs are available. Another workflow may choose Synchronous Dataflow semantics, where stronger guarantees of progress are available at the cost of more specification [9].

Taverna [15] uses a kind of Lambda calculus as its coordination formalism. The semantics detailed in [16] are strict, such that that activities must process all incoming elements before the next activity sees any output. Though simple to understand, strictness limits pipeline parallelism and expressiveness. Streams are the idea that solves these problems. Taverna is incorporating a notion of streams in their next software version.

Functional reactive programming is similar to our work, in

that it also deals with using functional concepts to compose stream-like data types [5]. In FRP, the basic data type is a "signal," which is a function from some type representing a continuous value that varies with time to a type representing discrete events. Signal functions, then, take a signal as input and produce another signal as output. Programs are written using a minimal set of signal combinators. These combinators are equivalent to Hughes' arrow combinators [6], and many FRP systems have adopted these are their set of primitives. We are currently examining the possibility of a formal connection between arrows and our own workflow representation.

Dataflow languages like Sisal [4], Id [13], and pH [12] are related to modern workflow languages. Dataflow languages view the entire program as combinations of dataflow relationships, all the way down to fine grained operations such as arithmetic. Workflow languages employ many of the same dataflow concepts, but are usually concerned with the coordination of more coarse-grained operations, and the movement of data between them. Workflow languages do not impose this flow-based programming model on the internal implementation of these coarse-grained computations. Of note, Id originally introduced MVars, which are now found in Haskell, and which we use extensively in our representation of effectful streams.

In this paper, we give details on how to construct several workflow primitives. Although we do not make the connection explicitly for each primitive, they are represented in the set of workflow patterns described in [17].

We use Haskell because it has convenient features for our purposes, including simple expression of streams and built-in MVars. Conceptually, however, our representation is language-independent. The stream data structure can be encoded in almost any language. MVars are easy to implement in any language with concurrency and locking.

Yet another approach to executable workflow representation is rule-based workflow execution [2]. In this paradigm, activity firings are triggered by the fulfillment of declarative rules, not unlike the rules in a Makefile. Furthermore, all data is stored into relational database tables, making it easy to examine the provenance of any particular result, and to re-run partial sections of a workflow based on updated rules or activities.

3. LAZY FUNCTIONAL STREAMS

The fundamental primitive in our model is that of a *lazy functional stream*. There are two important parts to this label – the notion of a functional stream and the concept of lazy evaluation. We will start by discussing functional streams, and will visit the role of laziness in Section 3.4. Stream functions can be viewed as higher-order versions of regular functions. Instead of operating on a single value, they operate on a sequence of values. For example, consider this simple function to compute the square of an integer:

```
square :: Int -> Int
square x = x*x
```

The stream version of this function is similar, but it consumes values from a potentially unbounded list of inputs, and produces a potentially unbounded list of outputs. Both versions, however, employ exactly the same logic. The streaming version of `square` (using Haskell's list as a simple kind of stream data structure) looks like this:

```
squarestream :: [Int] -> [Int]
squarestream (x:xs) = (x*x):(squarestream xs)
```

In the stream model, the function is defined to take a list of inputs (`x` followed by the remainder of the list, `xs`), and produce a list containing the value `x` squared followed by the recursive application of the function to the remaining elements of the input list. Notice the lack of a recursive base case handling the empty list – this is not a function on finite lists, but rather on streams with unbounded length. We can improve this function by making the acquisition of the head and tail explicit, allowing us to maintain the stream abstraction while giving us the flexibility change the underlying stream representation:

```
squarestream :: [Int] -> [Int]
squarestream s = (x*x):(squarestream xs)
  where x  = head s
        xs = tail s
```

Abstracting this pattern to make a higher-order function representing a stream computation based on a singleton function is straightforward:

```
applyToIntStream :: [Int] -> (Int -> Int) -> [Int]
applyToIntStream s f = (f x):
                       (applyToIntStream xs f)
  where x  = head s
        xs = tail s
```

In this case, any function that takes a single integer and produces an integer can be applied to an unbounded stream. We can abstract the pattern even further by making the arguments polymorphic. This function is a general application of a singleton function to each element of a stream:

```
applyToStream :: [a] -> (a -> b) -> [b]
applyToStream s f = (f x):(applyToStream xs f)
  where x  = head s
        xs = tail s
```

Functional programmers will note that our `applyToStream` function is equivalent to the familiar `map` function over lists. We create our own version here because, as we show in Section 3.1, we want to operate on more complex stream data structures.

Our `applyToStream` function is a useful workflow abstraction because a workflow program is a the composition of computations that work with single elements. The computations

are executed in a context where a sequence of elements will be passed through them. A workflow programmer writes a program by composing primitive functions. The workflow system provides a mechanism to move data elements through the composition. The stream-based workflow abstraction neatly reconciles this per-element notion of computational activities with the sequenced nature of the workflows.

Along with a method of applying functions to streams, we will want a set of combinators for composing stream functions together. For example, to sequence two functions `f` and `g`, such that `f` produces values for `g` to consume, we can use function composition (for which Haskell conveniently provides a built-in operator). We give examples of other useful combinators in Section 3.3.

Note that this kind of composition can be used effectively to build reusable sub-workflows. Once two or more activities are composed, there effectively exists a new activity where the inputs and outputs are exactly the free inputs and outputs of the underlying stream functions.

Thus far, we have limited our stream representation to Haskell lists. While lists capture the essential interface of a stream, they are lacking some important features related to workflows. We will now consider what properties a more robust stream representation should have if it is to serve as the basic data structure in a real-world workflow system.

3.1 Effectful streams

The stream model that we have described so far assumes that the functions are side-effect free. Indeed, Haskell explicitly disallows side-effects in this sort of purely functional code. This is a serious limitation on our ability to represent real-world workflows. In particular, it is often the case that we would like the act of reading from a stream to alter the state of that stream, such that each consumer sees the effects of all the other consumers. For example, in the purely functional stream model, if we have a single stream that is split and handed to two subworkflows, the subworkflows see the original stream as if the other subworkflow did not exist. One subworkflow cannot tell if a value intended to be consumed by one and only one subworkflow has been consumed already or not. This is a consequence of the pass by value nature of most functional languages. To support this "consume once and only once" behavior, we would require the consumption to have an *effect* on the stream. To support this while remaining in the functional setting, we must take advantage of concurrency constructs provided by Haskell in the form of MVars.

An MVar is a concept originally introduced in Id [14, 13] and used in later derivative languages such as Sisal [4] and pH [12]. An MVar should be thought of as a slot holding a value that can be read from and modified. A common use of MVars is to hold shared counters. If, instead of holding a singleton value, we create an MVar that contains a pure stream, we can define an operation that atomically takes the head of the stream, returns it to an accessor, and replaces the contents of the MVar with the remainder of the stream. If we do this, then the act of reading the head of a stream can have an effect on the stream such that any other consumer from the stream will see the state of the stream based on the actions of other consumers.

The choice of pure versus effectful streams is based largely on the requirements of the workflow — in some cases one is preferable to the other. Ideally, we would like to make this choice transparent to computational activities within the workflow, and supporting primitives provided by the workflow representation. To achieve this, we employ Haskell's type class facility to define a generic stream type, along with two instantiations of the class. The first instance provides pure functional streams, and the second effectful streams.

3.2 Stream type class

In our discussion above, we start with a purely functional lazy stream and move on to describe streams where the act of consuming values from them has a side-effect. Both models of streams are valid and have their place, and often we want to define functions for composing stream-based functions that apply to both types of stream. One way to achieve this in Haskell is to use a type class representing a generic stream, from which we can define instances for both pure and effectful streams. The key is to make operations such as *head* and *tail* explicit instead of relying on built-in Haskell operators. This gives the implementors of instances of the type class freedom to control precisely what happens when streams are operated on — a very important consideration when using streams to represent channels in distributed systems or other contexts out of control of the Haskell language runtime. We provide a single operation, *advance*, that given a stream returns a pair representing the first element available on the stream and the remainder of the stream.

Our type class is defined as follows:[1]

```
class Streamer a b | a -> b where
  newStream :: [b] -> a
  sAdvance  :: a -> (b, a)
  sAdvanceN :: Int -> a -> ([b],a)
  sEmpty    :: a -> Bool
```

How do we read this? First, we are defining a type class called `Streamer` with generic type parameters `a` and `b`. We use the functional dependency notation to denote that the type `b` is uniquely determined by the type `a`. We will see what this means shortly. Within our type class, we define four functions that an instance of the type class must provide.

- `newStream`: This function takes a list of elements and returns a stream containing them. We must use this to lift regular Haskell lists into our stream type so that the following operators may apply to them.
- `sAdvance`: Returns a pair containing the element at the front of the, and the remainder of the stream after this element has been removed.

[1]Our definitions take advantage of extensions to the Haskell language provided by the Glasgow Haskell Compiler.

- `sAdvanceN`: Given that streams are potentially infinite structures, we must use this function to take a finite number of elements from the stream. `sAdvanceN` is equivalent to iterating `sAdvance` to create a list of elements.
- `sEmpty`: Used to test whether or not a stream contains elements that can be accessed. Whether or not an empty stream can later acquire additional elements depends on how the it is implemented.

The simplest instance of our type class is that of a purely functional stream.

```
type PStream a = [a]

instance Streamer (PStream a) a where
  newStream l   = l
  sAdvance  s   = (head s, tail s)
  sAdvanceN i s = (take i s, drop i s)
  sEmpty        = null
```

As we can see, a `PStream` (or *pure stream*) is nothing more than a Haskell list in disguise. The type `PStream a` is defined as a list of elements of type `a`, and the instance functions are nothing more than aliases for the built in list primitives. We can now see why the functional dependency notation was necessary in the definition of the type class. In our instance of `Streamer`, we see that the two parameter types are `PStream a` and `a`. In the type class definition, we saw that the second type was to be uniquely determined by the first, and that is shown here in this instance. A `PStream` is a list of elements of type `a`. Making both the stream type and element type explicit and related is necessary for the Haskell type checker to validate that the member functions that require both can be properly type checked (such as `sAdvance`).

Our type class becomes more interesting when we consider streams that are not simply traditional functional lists. Consider the following scenario. Say we wish to have streams that can be passed to multiple threads of execution with the requirement that consumption of each element of the stream may occur once and only once. In other words, operations on the stream must have a *side-effect* on the stream. The notion of "effectful" streams motivates our second instance of the type class.

```
type EStream a = MVar (PStream a)

swap :: (a,b) -> (b,a)
swap (x,y) = (y,x)

instance Streamer (EStream a) a where
  newStream l = unsafePerformIO $ newMVar l

  sAdvanceN n s = unsafePerformIO $
    do r <- modifyMVar s
            (\i -> do return $ swap $ sAdvanceN n i)
       return $ (r, s)

sEmpty s = unsafePerformIO $
  withMVar s (\i -> do return $ sEmpty i)

sAdvance s = unsafePerformIO $
  do r <- modifyMVar s
          (\i -> do return $ swap $ sAdvance i)
     return $ (r, s)
```

As we can see, instead of providing a thin layer above the built-in list primitives for purely functional lists, each function operates on the contents of the MVar. The function `modifyMVar` takes two arguments: the MVar and a function to apply when modifying its contents. It first reads the contents of the MVar, applies the provided function, and then places the updated value back into the MVar. The function that `modifyMVar` takes as an argument is provided the contents of the MVar, and then returns a pair with the new value to be placed in the MVar as the first element, and the value to return to the caller of `modifyMVar` as the second. The function `withMVar` operates similarly, but does not change the content of the MVar — it simply extracts it, applies the provided function, restores the value into the MVar, and returns the result of the function that was applied. Our use of the `swap` helper function is necessary because our `sAdvance` operations define the tuple in the opposite order. We chose this ordering to better match the relationship of the front and back of a list.

In the effectful stream instance, the operations are implemented via the `unsafePerformIO` function. In Haskell, many imperative features take place within what is known as the *IO monad*. Monads are the Haskell mechanism for containing and controlling side effects, most often related to IO. Unfortunately, use of features (such as MVars) that reside within the IO monad requires much of the code to then be implemented within the IO monad as well. If we required workflow programs based on effectful streams to reside in the IO monad, we would violate the transparency we are hoping to achieve with the type class — users of effectful streams would program differently than those using pure streams. Instead of forcing all code using effectful streams to reside within the IO monad, we encapsulate the effectful code necessary to manipulate the MVars within the `unsafePerformIO` call that allows us to briefly enter the IO monad, perform our effectful operations, and escape back to the pure world.

While convenient, use of `unsafePerformIO` can have consequences (hence the choice of the Haskell designers to label this operation as "unsafe"). In general, `unsafePerformIO` makes it difficult to reason about the relative ordering of IO events. This would be a disaster for most programs, but in the case of most workflow systems the relative ordering of effects is only defined up to the constraints imposed by the data flow dependencies anyway. We do, however, lean a bit on the implementation provided by the Glasgow Haskell Compiler (GHC), in particular the way it memoizes functions. We are currently exploring ways to reconcile our stream types with monadic effects, without losing the polymorphic and lazy characteristics.

3.3 Stream primitives

Given the type class and instances for different types of streams, we would like to provide a set of primitives that can be used to build interesting workflows. As we demonstrated earlier in our explanation of *applyToStream*, we can provide generic helper functions that can promote functions on primitive types to act on streams containing elements of those types. We currently provide a set of functions called *lifters* for this purpose. For example, consider the following lifter for functions of two arguments:

```
lifter2 :: (Streamer a b, Streamer a' b',
            Streamer c d)
        => (b -> b' -> d)
        -> a -> a' -> c
lifter2 f s1 s2 = newStream (appf2 s1 s2)
  where appf2 s1' s2' =
          let (fr1, r1) = sAdvance s1'
              (fr2, r2) = sAdvance s2' in
            (f fr1 fr2):(appf2 r1 r2)
```

Examination of the type signature shows that the lifter is provided with a function on two arguments (of potentially different types) that produces an output of a third type. The lifter then takes two streams containing elements of types that match those of the function specified in the first argument. Finally, what is produced is a stream that contains the output of the lifted function applied to the two input streams. Our library of workflow primitives contains similarly structured lifters for functions of various numbers of arguments.

Once we have functions lifted to operate on streams, we then can use other primitives to connect them together in interesting ways.

Splitters

Given a stream, we may want to split it into a set of streams that can be passed into distinct sub-workflows (see Figure 1). Of course, there are different ways that this can be achieved, leading to different workflow semantics. Consider the following function for splitting a single stream into two streams.

```
split :: Streamer a b => a -> (a,a)
split s = (left,right)
  where left  = lifter id s
        right = lifter id s
```

Here a single stream input is passed into the *split* primitive, and a pair of streams are created, each of which is the result of the identity function lifted and applied to the original stream. An issue arises if we consider what this construct means with respect to operations on streams. If an element is consumed from the left stream, does that have an impact on the contents of the right stream? Even though we have split the stream, we should notice that the two resulting streams are defined based on the original. So, if applied to effectful streams, effects of operations by one stream would be visible to the other. This primitive is therefore useful

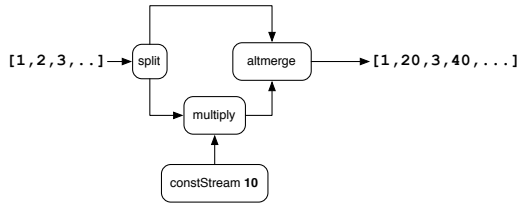

Figure 1: Effectful split

if we want to allow the two sub-workflows to represent concurrently executing sub-programs that are consuming values from a stream in a first-come, first-served manner.

If this is not the desired semantics, and we wish to effectively duplicate the stream being split, we have an alternative primitive supporting that.

```
dup :: Streamer a b => a -> (a,a)
dup s =
  let (l1,l2) = unzip (dupList s)
  in (newStream l1, newStream l2)
  where
    dupList st = let (hd,tl) = sAdvance st
                 in (hd,hd) : (dupList tl)
```

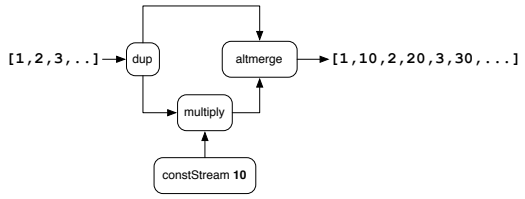

Figure 2: Effectful dup

With this primitive, every element of the input stream is duplicated and present in both output streams. It is interesting to note that with pure streams, both *dup* and *split* have identical behavior. When applied to effectful streams though, their behaviors are distinct.

A final splitter of some interest is the alternating splitter, *altsplit*. Like the *dup* splitter, the resulting streams are independent. Unlike *dup*, they are not identical. This splitter represents a binary round robin splitter where the odd elements of the input stream are placed in the first output stream, and the even numbered elements end up in the second.

```
altsplit :: (Streamer a b) => a -> (a,a)
altsplit s =
  let (l1,l2) = unzip(altList s)
  in (newStream l1, newStream l2)
  where
    altList st = let (h1,t1) = sAdvance st in
                 let (h2,t2) = sAdvance t1
                 in (h1,h2):(altList t2)
```

Mergers

Another common pattern in workflows is to integrate multiple streams into a single stream. A few primitives are provided to help with this task. Say we have two streams joining together and we wish to produce a stream of their values serialized in a specific way. One choice is to alternate between the two streams.

```
altmerge :: (Streamer a b) => a -> a -> a
altmerge xs ys = newStream (alt xs ys)
  where alt l r = let (lf,lr) = sAdvance l
                  in lf : (alt r lr)
```

In another case, we may want to give priority to one stream over the other. If we assign a "side" to the two streams, we can call this primitive *selectLtR* for "select left then right". If the left stream has values available, they are emitted on the output stream. Otherwise, values are taken from the right stream.

```
selectLtR :: Streamer a b => a -> a -> a
selectLtR xs ys = newStream (choose xs ys)
  where choose l r = if sEmpty l
                     then let (rf,rr) = sAdvance r
                          in (rf : choose l rr)
                     else let (lf,lr) = sAdvance l
                          in (lf : choose lr r)
```

We also may not want to serialize the streams, but instead emit a stream where the values from the streams being merged are packaged together in a single aggregate entity. We can use a simple zipper-style primitive for this.

```
zipper :: (Streamer a b, Streamer a' b',
           Streamer c (b,b'))
       => a -> a' -> c
zipper xs ys = newStream (zipped xs ys)
  where zipped l r = let (lf,lr) = sAdvance l in
                     let (rf,rr) = sAdvance r
                     in (lf,rf) : (zipped lr rr)
```

Seeders

Often we want the ability to inject special values into workflows for the purposes of parameterizing activities or performing tasks like parameter studies. These are easily represented. For example, if we wish to produce a stream containing an infinite number of instances of some fixed constant, we can say:

```
constantStream :: (Streamer a b) => b -> a
constantStream c = newStream (repeat c)
```

We may also wish to initialize a stream between two activities with a finite number of values waiting to be consumed[2]. This is useful for seeding workflows with some initial values necessary to bootstrap them.

```
seededStream :: (Streamer a b) => [b] -> (a -> a)
seededStream l s = selectLtR (newStream l) s
```

Counting streams are also easily represented, and can be used when building generic activities that require a counter (such as an activity reading a set of numbered data files from a directory).

```
counterStream :: (Enum b, Streamer a b) => b -> a
counterStream start = newStream ([start..])
```

3.4 The virtue of being lazy

We rely on Haskell's lazy evaluation for our stream representation. In particular, many streams are naturally infinite structures. Lazy evaluation allows us to manipulate these kinds of infinite streams without having to fully evaluate them. For example, consider the lifter operations, such as *lifter2*, shown earlier. The recursive definition of the stream in terms of a list where the tail is defined as recursive application of the function leads to an infinite list. Clearly we cannot evaluate the list in its entirety. In a lazy model, evaluation of the elements will occur only when they are required – when the elements are actually read from the outputs of the workflow and evaluated. Both of the example workflows shown in Section 5 deal with infinite streams. The `sTake` function that is part of the stream type class is a convenient way to examine a finite number of elements from these infinite structures.

4. CONCRETE COMPUTATION BINDINGS

Given a stream representation of the workflow in Haskell, we are still faced with the problem of binding the workflow activities to concrete computations. There are two sub-problems: associating computations defined outside the Haskell runtime system (e.g. C functions) with our stream functions, and encoding and moving real data through the workflow.

4.1 Mapping external computations to activities

Our workflow representation is intended to coordinate the execution of real-world computations. To do so, the computations must be bound to workflow activities. As such, we require a binding mechanism that maps the Haskell representation of an activity to the intended implementation. There are at least two options for this task. The first is the Haskell foreign function interface (FFI), used to bind

[2]Strictly speaking, this activity can be seeded with an infinite list, but that would cause the second stream to never be consumed from. That would not be very useful.

Haskell functions directly to implementations in other languages. The other is Haskell's interprocess communication facilities, where a computation is invoked via a Haskell wrapper which reads and writes data from IPC channels.

In practice, we tend to use the FFI for routines that are coded in C, since we can maintain useful type information. For other codes, such as those written Matlab, Python, or other higher-level languages, we use IPC.

4.2 Data representation

All production workflow systems must strive to avoid unnecessary movement and replication of data. To this end, we use a token scheme to represent data within stream-based workflows. Tokens are conceptually orthogonal to our workflow representation, but are an important part of a real runtime-system implementation.

For reasons of efficiency, our Haskell code performs no interpretation of the data contained within a token – we simply pass the tokens between activities as dictated by the data- and control-flow semantics of the workflow. An activity implementer is therefore free to use whatever data encoding they choose, and is not forced to implement a (possibly costly) mapping of their data types into a Haskell representation.

In practice, we find ourselves using two different strategies for encoding data with tokens. In many cases, the token's payload is exactly some primitive data type defined by Haskell (e.g. 32-bit integers) where little or no mapping is required between Haskell and the host language. For more complex data (e.g. images), we use the digest of some suitable hash function on the data, such as MD5, as the payload. The hash digest is passed to the host language and used as a vector to the real data, often by way of a shared relational database or the filesystem. Once the data has been recovered, it is passed to the activity routine. If the routine produces new data with a hashed representation, the process is revered – the hash function is used to produce the digest, which is stored along with the result if it does not already exist, and the digest is passed back into Haskell.

Note that our token strategy still allows us to leverage Haskell's sophisticated type system to verify workflow connections. In the case where a token encodes a primitive data type (e.g. `Int`), the corresponding token type will simply wrap the primitive one (e.g. `Token Int`). In the case where we are using a hashed representation, we must define a new Haskell datatype to represent the data as a hash value. For example:

```
data Image = Image Digest

foo :: Token Image -> Token Image
foo img = externalRoutine img

sFoo :: (Streamer a (Token Image)) => a -> a
sFoo imgs = lifter foo imgs
```

If some other type of stream is passed to `sFoo`, the type system will catch the error.

We are currently exploring other possible benefits of the hash token strategy such as provenance tracking, memoization of functional activities, and distributed execution.

5. DEMONSTRATION

We have established the stream representation as a Haskell type class and defined a small number of primitive stream operators. We now examine how these elements can be used to compose some actual workflows.

5.1 Fibonacci generator

In this example, we show a workflow that produces an infinite sequence of Fibonacci numbers. Clearly, there are more concise (and efficient) ways of performing the same computation – we use this example here because it gives a clear example of some interesting control flow constructs. The only "domain" operation in this case is addition, all the other activities are drawn from our set of primitives. See Figure 3 for a graphical representation of the workflow.

```
-- Function to lift primitive addition operator
-- to a stream model.
adder :: (Num b, Streamer a b) => a -> a -> a
adder = lifter2 (+)

-- Define a workflow to produce Fibonacci numbers
fib :: (Num b, Streamer a b) => a
fib = let lr1 = seededStream [0] db1
          lr2 = seededStream [1] da2
          (db1,db2) = dup lr2
          add = adder lr1 db2
          (da1,da2) = dup add
      in da1
```

First, we define our activity `adder` by lifting the binary addition function to a stream context. The result is a workflow activity that operates on streams of numbers. The second function we define is `fib`, which represents the workflow itself. As noted previously, the result is a function that produces a stream no differently than other primitive activities like *constantStream* or *counterStream*. This means that the entire workflow can be called from within a larger workflow, which demonstrates how our workflow representation supports workflow nesting.

The first control flow construct to notice is the loop. Notice how `lr2` is defined in terms of `da2`, which itself uses `lr2` (by way of `add` and `db2`). In our system, loops correspond to recursive definitions of this kind. Haskell makes defining this kind of recursion easy; in particular, it does not insist that symbols be defined in order of use. You are free to use a symbol in an expression, and define it further down in the same `let` block before the corresponding `in` keyword.

The second control flow construct is the delay. Notice how `lr2` is fed by the result of the workflow, and then moves its last item into `lr1` for the next sequence element. This kind of order-based sequencing is quite common in scientific workflows, and is easy to express in our system.

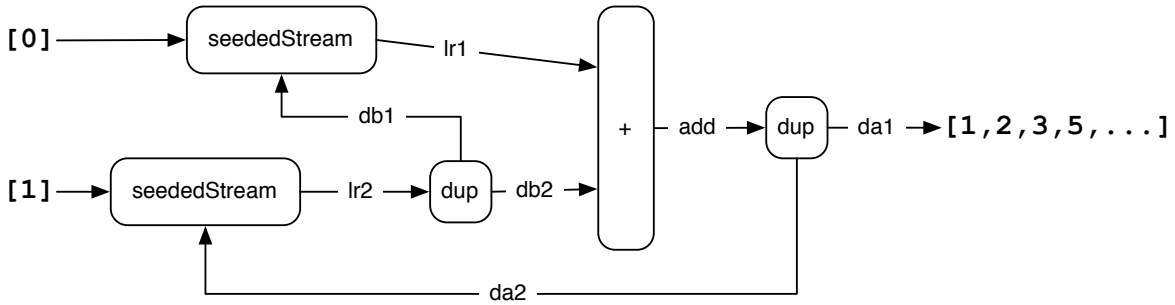

Figure 3: Fibonacci workflow

5.2 Prime Sieve

This example demonstrates a workflow inspired by the well-known Sieve of Eratosthenes, an algorithm for finding prime numbers. The algorithm is intuitive – start with a list of consecutive integers, and a list of known primes that is initially empty. Iterate through each number in the list in ascending order, and if it is divisible by any of the numbers in the list of primes encountered so far, ignore it. Otherwise, add it to the list of primes.

Implementing this algorithm as a workflow results in two feedback loops. First, the list of primes must be duplicated and fed back into the list of known primes. Additionally, when a new prime is found, the known list must be expanded to include both the new prime number and the old list of primes that was used to generate it. So, the old known-prime list must also be fed back into the loop.

See Figure 4 for a graphical representation of the workflow. The input to the workflow is an infinite sequence of integers starting at 2, represented by `ns`. The workflow maps this input into a stream of type `Maybe Int`, such that each found prime is represented by `Just` n, where n is prime, and all other integers by `Nothing`. In a moment, we will see that this represents a more general pattern of "workflow bubbles."

In the workflow, the `sMemory` activity is responsible for keeping track of the list of primes. It takes two inputs – the current known list of primes, and a `Maybe Int`. If the second input is `Just` n, it appends n to the current list, and outputs it. If it is `Nothing`, it just outputs the current list. Notice how the list that it outputs is fed back into itself; the loop effectively acts as a memory bank, using the stream as storage for the memorized list.

```
-- "Remember" Just elements and "forget" Nothings
memorize :: [a] -> Maybe a -> [a]
memorize mems (Just x)  = x:mems
memorize mems (Nothing) = mems

-- Stream version of memorize
sMemory::(Streamer a [Int],Streamer b (Maybe Int),
          Streamer c [Int]) => a -> b -> c
sMemory = lifter2 memorize
```

The main input to the workflow, the sequence of integers, is duplicated and fed to `sDivides`, which checks if the integer in question is divisible by any of a list of known primes. The boolean result is negated by `sNot`, and then sent along with a copy of the corresponding integer to `sAccept`. This activity checks the integer and the boolean, and outputs `Just` n if the boolean is true, and `Nothing` otherwise. This output is our list of primes.

```
-- Prime Sieve of Eratosthenes
primes :: (Streamer a Int) => a
primes =
    let ns = newStream [2..]
        empty = newStream [[]]
        (ns1,ns2) = dup ns
        mem = sMemory (selectLtR empty mem1) p1
        (mem1,mem2) = dup mem
        divs = sDivides mem2 ns1
        ndivs = sNot divs
        accepted = sAccept ns2 ndivs
        primes = seededStream [Nothing] accepted
        (p1,p2) = dup primes
    in sDebubble p2
```

Notice that the `Maybe Int` list of primes is fed back to the memory sub-workflow, and is also copied to an activity we call the "debubbler". This activity removes the `Nothing` elements from the stream, and unwraps the remaining integers from the `Just` data type, leaving a stream of nothing but prime integers.

```
sDebubble :: (Streamer a (Maybe b), Streamer c b)
          => a -> c
sDebubble s = newStream (db s)
  where db st = let (sf,sr) = sAdvance st
                in if (isNothing sf) then (db sr)
                   else (fromJust sf) : (db sr)
```

The debubbler is an interesting primitive that we include with the library of stream primitives as it represents a useful, generic operation. At times, we may wish to allow an active "null" datum to traverse streams between activities.

Instead of resorting to magic values, like zeros on an integer stream or empty strings on a string stream, the Maybe type allows us to extend *any* type of stream with the notion of a null, `Nothing` datum. The debubbler is necessary when we wish to convert these streams that support bubbles to those that do not. Our workflow bubbles were inspired by similar techniques encountered in a variety of pipelined systems.

6. CONCLUSIONS AND FUTURE WORK

We have demonstrated that lazy functional streams are a clean and useful representation for workflows, along with a concrete implementation of the types and operators necessary for building stream-based workflows in Haskell. We have also shown how Haskell's concurrency control constructs can be used to implement different stream semantics to reflect different ways that side-effects can be handled. Finally, we have demonstrated manually constructed flow-based programs that use our set of workflow primitives for programs with interesting flow-structures such as feedback loops and stream "bubbles."

The intent of this work is to provide a language-neutral intermediate representation for higher-level workflow languages to target. The goal of this is to cleanly decouple the workflow description as seen by the user from the runtime support and coordination layer that actually orchestrates computations and data movement. Our current activity is focused on understanding how our work in this paper is related to, and possibly representable by structures such as Haskell Arrows. We are also actively investigating a monadic implementation of our system so that we can replace the undesirable use of `unsafePerformIO` within the effectful stream type class instance with a cleaner, but transparent use of the IO monad. Finally, we will release in the near future a version of the toolkit implementing the work described in this paper as an open source package for public consumption.

7. REFERENCES

[1] Ilkay Altintas, Chad Berkley, Efrat Jaeger, Matthew Jones, Bertram Ludäscher, and Steve Mock. Kepler: an extensible system for design and execution of scientific workflows. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management 2004*, pages 423–424, 2004.

[2] John Conery, Julian Catchen, and Michael Lynch. Rule-based workflow management for bioinformatics. *The VLDB Journal*, 14(3):318–329, 2005.

[3] Thomas Fahringer, Sabri Pllana, and Alex Villazon. A-GWL: Abstract Grid Workflow Language. In *4th International Conference on Computational Science (ICCS 2004)*, Lecture Notes in Computer Science, pages 42–49. Springer Berlin / Heidelberg, June 2004.

[4] Jean-Luc Gaudiot, Tom DeBoni, John Feo, Wim Bohm, Walid Najjar, and Patrick Miller. The sisal model of functional programming and its implementation. In *Proceedings of the Second Aizu International Symposium on Parallel Algorithms/Architecture Synthesis*, pages 112–123, Los Alamitos, CA, USA, March 1997. IEEE Computer Society.

[5] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming. *Lecture Notes in Computer Science*, 2638:159–187, 2003.

[6] J. Hughes. Generalising monads to arrows. *Science of computer programming*, 37(1-3):67–111, 2000.

[7] Geoffrey C. Hulette, Matthew J. Sottile, and Allen D. Malony. Wool: A workflow programming language. *IEEE International Conference on eScience*, 0:71–78, 2008.

[8] Christopher Hylands, Edward Lee, Jie Liu, Xiaojun Liu, Stephen Neuendorffer, Yuhong Xiong, Yang Zhao, and Haiyang Zheng. Overview of the Ptolemy project. Technical report, University of California Berkeley, 2003.

[9] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. In *Proceedings of the IEEE*, volume 75, pages 1235–1245, September 1987.

[10] Edward A. Lee and Thomas Parks. Dataflow process networks. In *Proceedings of the IEEE*, pages 773–799, 1995.

[11] B. Ludascher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger-Frank, M. Jones, E. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice & Experience*, 2005.

[12] Jan-Willem Maessen, Arvind, Shail Aditya, Lennart Augustsson, and Rishiyur S. Nikhil. Semantics of pH: A parallel dialect of Haskell. In *In Proceedings from the Haskell Workshop (at FPCA 95)*, pages 35–49, 1995.

[13] Rishiyur S. Nikhil. *ID Language Reference Manual (Version 90.1)*. MIT Computational Structures Group, Cambridge, MA, July 1991.

[14] Rishiyur S. Nikhil. An overview of the parallel language Id (a foundation for pH, a parallel dialect of Haskell). Technical report, Digital Equipment Corporation, Cambridge Research Laboratory, 1994.

[15] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R. Pocock, Anil Wipat, and Peter Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, November 2004.

[16] Daniele Turi, Paolo Missier, Carole Goble, David De Roure, and Tom Oinn. Taverna workflows: Syntax and semantics. In *E-SCIENCE '07: Proceedings of the Third IEEE International Conference on e-Science and Grid Computing*, pages 441–448, Washington, DC, USA, 2007. IEEE Computer Society.

[17] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

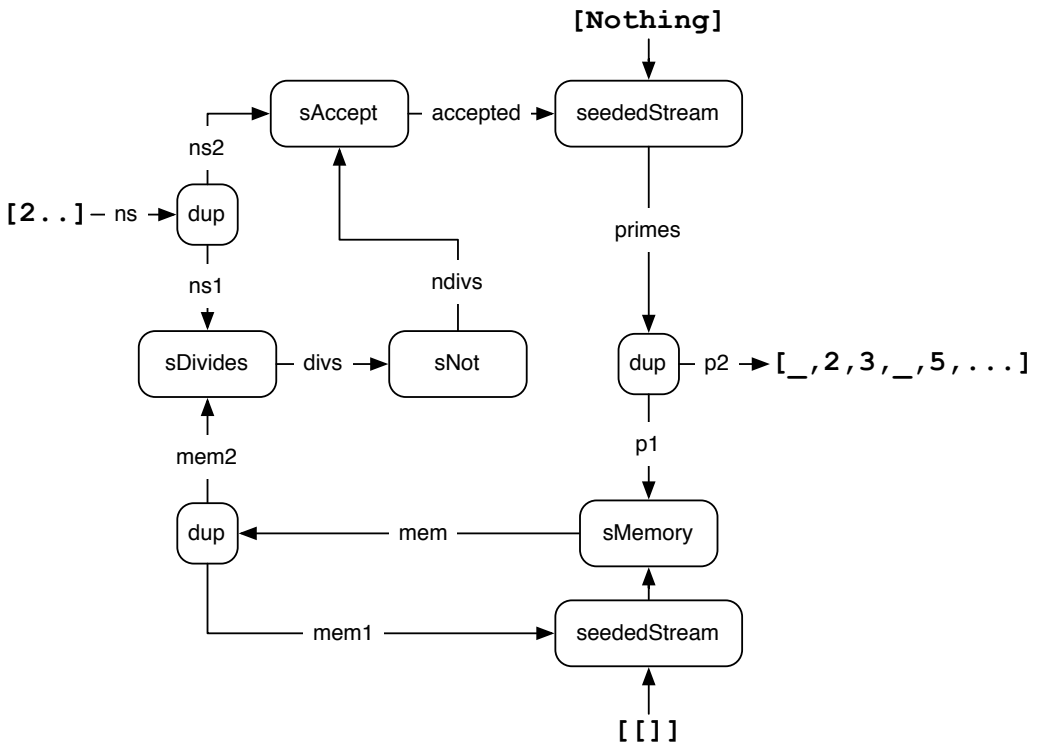

Figure 4: Prime sieve workflow with bubbles shown in the output (indicated as _ elements)