Lazy functional streams for workflow representation and runtime

Matthew J. Sottile[1,2], Geoffrey C. Hulette[1], Allen D. Malony[1]

[1]University of Oregon
Department of Computer and Information Science

[2]Galois, Inc.
Portland, OR, USA

February 18, 2010

UNIVERSITY OF OREGON

galois

Introduction

Our goal: to explore the use of Haskell as a substrate for experimenting with domain specific languages for workflow specification.

Focus on workflow specification from a language perspective, instead of a distributed runtime layer point of view.

This work originated as an effort to simplify workflow specification relative to existing grid workflow systems.

- Priority is expressivity of language with respect to dataflow and workflow semantics.

Workflow model of computation

What is a workflow?

Workflow in computer science gained attention for coordination of computations in grid-based environments.

- Workflow is analogous to dataflow computing.

The focus is on computational activities that have well defined inputs and outputs, and relationships that bind outputs to inputs.

- We use lazy functional streams to provide the abstraction for the output/input flow relationship.

Application area

Our target is not the heroic, x^k core single jobs (where k big).

Our target is large scale analytics problems common in domain sciences (e.g.: biology). The sheer volume of data that can be produced necessitates the use of huge machines to keep up.

Therefore:

- Coordination of small processing tasks and flow of data between them.
- Parallelism possible in many places, both fork-join style and pipeline.

Domain specific languages

DSLs are small languages that are tuned for a specific goal.

Embedding a DSL in another language makes building the DSL easier.

- Many features for free from base language.
- Standard library, type system, etc...

We chose Haskell due to its flexible and strict type system.

Our primary use of the type system is to reason about side-effects and nondeterminism – two features commonly present in workflows, particularly those that contain some degree of parallelism.

Our approach

The first step was to embed our workflow specification language in Haskell to allow the construction of pure Haskell workflow programs.

- All activities are Haskell routines.

The next step was to support the replacement of Haskell computations with external programs written in more conventional languages (such as C or Fortran).

- Potentially in a distributed environment.
- Haskell transitions into a pure coordination-oriented role.
- Computations and data both outside of Haskell – i.e. coordinate *by reference*.

Step 1: Purely Haskell

Let's step through the process of moving pure Haskell functions into a workflow model.

In the process, we will see how we capture the essential pattern of workflow relationships in the abstraction of a stream.

- Streams are infinite lists in functional langauges.
- Laziness makes these easy to deal with.

Our challenge is rectifying necessarily side-effectful aspects of workflows with the purity of Haskell.

The journey of a computation

Let's start at the beginning.

A simple function

```
adder :: Int -> Int -> Int
adder x y = x+y
```

The function named `adder` two integer arguments and produces an integer result. Nothing terribly special. Note that the type declaration is optional; Haskell can infer the types automatically.

Streams

How could we write the same function, but for unbounded lists of integers?

A simple stream-based function

```
adder :: [Int] -> [Int] -> [Int]
adder (x:xs) (y:ys) = (x+y):(adder xs ys)
```

This function will work correctly even (and only) on an infinite list! Haskell can easily manipulate infinite structures because it evaluates them lazily.

When we use the term **stream**, we mean these kinds of lazily-evaluated, potentially infinite lists.

Streams

Streams are easy to work with in Haskell.[1]

Using stream adder

```
ghci> let firstStream = [1..]
ghci> let secondStream = [10..]
ghci> let sums = adder firstStream secondStream
ghci> take 15 sums
[11,13,15,17,19,21,23,25,27,29,31,33,35,37,39]
```

Haskell evaluates only what you need, only when you need it.

Laziness can be achieved in other languages, but Haskell gives it to us for free.

[1] "ghci" in the code indicates that we are in the interactive interpreter.

Laziness

Haskell uses call-by-need evaluation, often referred to as 'lazy evaluation'.

How is this implemented?

- Expressions are turned into "thunks", which represent an unevaluated computation.
- When, and *only when* their values are required are they evaluated.
- Upon evaluation, the thunk is replaced by the resulting value. This leads to automatic memoization.

Pattern for streams

A generic pattern emerges upon examination.

A move towards the generic

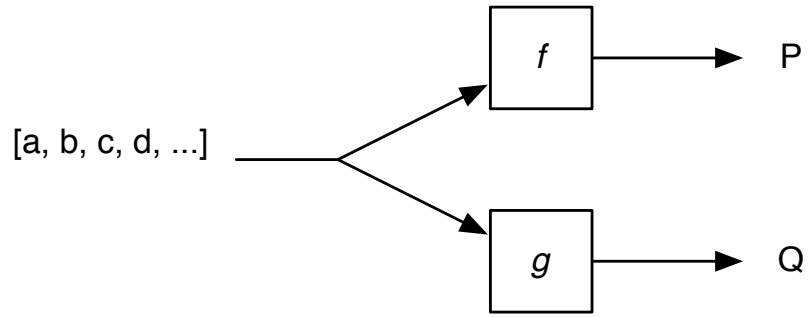
```
streamify :: [a] -> [a] -> (a -> a -> a) -> [a]
streamify (x:xs) (y:ys) f =
    (f x y):(streamify xs ys f)
```

The function `streamify` has a complex type signature. Let's decipher it:

- First two arguments are streams of the same generic type.
- Third argument is a function that takes two arguments of that type, and yields a result of the same type.
- The overall function then yields a stream of the same type.

Effects and streams

Pure functional languages like Haskell avoid side-effects at all costs. This has implications on our workflow representation.



Do f and g have an effect on the shared input? Which is the correct output:

- $P = [(f\ a), (f\ b), (f\ c), \cdots]$, $Q = [(g\ a), (g\ b), (g\ c), \cdots]$
- $P = [(f\ a), (f\ c), \cdots]$, $Q = [(g\ b), (g\ d), \cdots]$

Effects and Haskell

Haskell manages side-effects explicitly in the type system.

Effectful computation

```
haveAnEffect :: Int -> IO Int
haveAnEffect x = do print (show x)
                    return (x+1)
```

The type signature indicates that this function will produce an integer and have a side effect (here, printing to the screen).

The language feature used to implement this is the *IO monad*. Monads are beyond the scope of this talk, but are key in managing both purity and impure language features in Haskell.

A stream typeclass

We can define a generic interface, called a *typeclass*, for streams.

Streamer typeclass

```
class Streamer s where
  newStream :: [a] -> s a
  sEmpty    :: s a -> Bool
  sAdvance  :: s a -> (a, s a)
```

The a is read "alpha", and it stands for any type.

Now we can experiment with different stream semantics by creating different instances of `Streamer`.

Primitives, like splitting and merging streams, are defined generically for all `Streamer` instances.

Pure stream instance

Pure streams are easy! They're just lists in disguise.

Pure stream instance

```
type PStream a = [a]

instance Streamer (PStream a) a where
  newStream l = l
  sAdvance x  = (head x, tail x)
  sEmpty      = null
```

Remember that pure means no side effects.

Effectful stream instance

Effectful streams are more interesting.

Effectful stream instance

```
import Control.Concurrent.MVar

data EStream a = EStream (MVar [a])
```

We are stashing a pure stream (the `[a]`) inside a special container called an *MVar*. An MVar is like a box containing a data value that can be accessed (read and write) by multiple threads in a synchronized manner.

Effectful stream instance

Making a new stream is easy.

Implementing `newStream`

```
newStreamIO :: [a] -> IO (s a)
newStreamIO l = do
  m <- newMVar l
  return (EStream m)
```

Create a new empty MVar, and put a pure stream inside it.

Effectful stream instance

Implementing `sAdvance`

```
sAdvanceIO :: s a -> IO (a, s a)
sAdvance s = do
  let EStream m = s
  r <- modifyMVar m (\i -> return (swap (sAdvance i)))
  return (r, s)
```

The `modifyMVar` atomically takes the value out of the MVar, applies a function to the value, and then puts the result back.

Q: Is the call to `sAdvance` inside `modifyMVar` recursive?
A: No, it calls the pure stream version.

Revisiting streamify (unary case)

Here is a better version of `streamify`, using the generic typeclass.

Improved streamify lifter

```
streamify :: Streamer a -> (a -> a) -> Streamer a
streamify s f = (f x):(streamify xs)
  where
    (x, xs) = sAdvance s
```

Note: this version of `streamify` does not specify a type of `Streamer`. It works for both pure and effectful streams

Higher-arity streamifiers

Lifting functions of multiple arguments is similar.

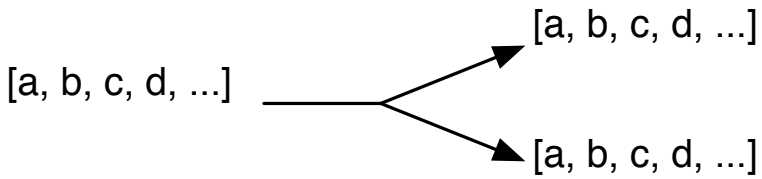
Binary streamify lifter

```
streamify2 :: Streamer a -> Streamer b -> (a -> b -> c)
           -> Streamer c
streamify2 s1 s2 f = (f x y):(streamify2 xs ys)
  where
    (x, xs) = sAdvance s1
    (y, ys) = sAdvance s2
```

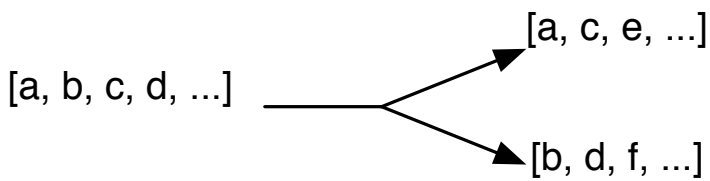
Notice that we can also be more general in the type of the streams. In this case, each of the streams can contain different types.

Primitives

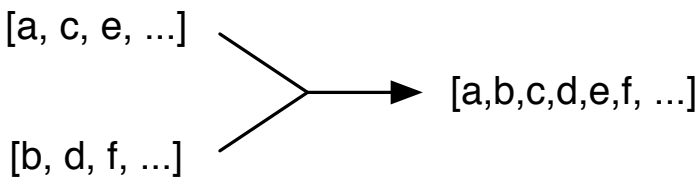
Duplicating split



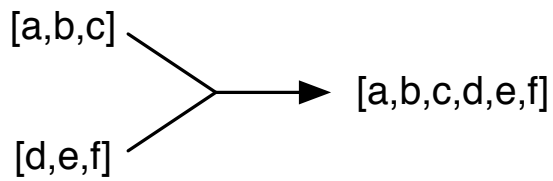
Alternating split



Alternating merge



Biased merge



We can build primitives that manipulate streamers. These are described in more detail in our paper.

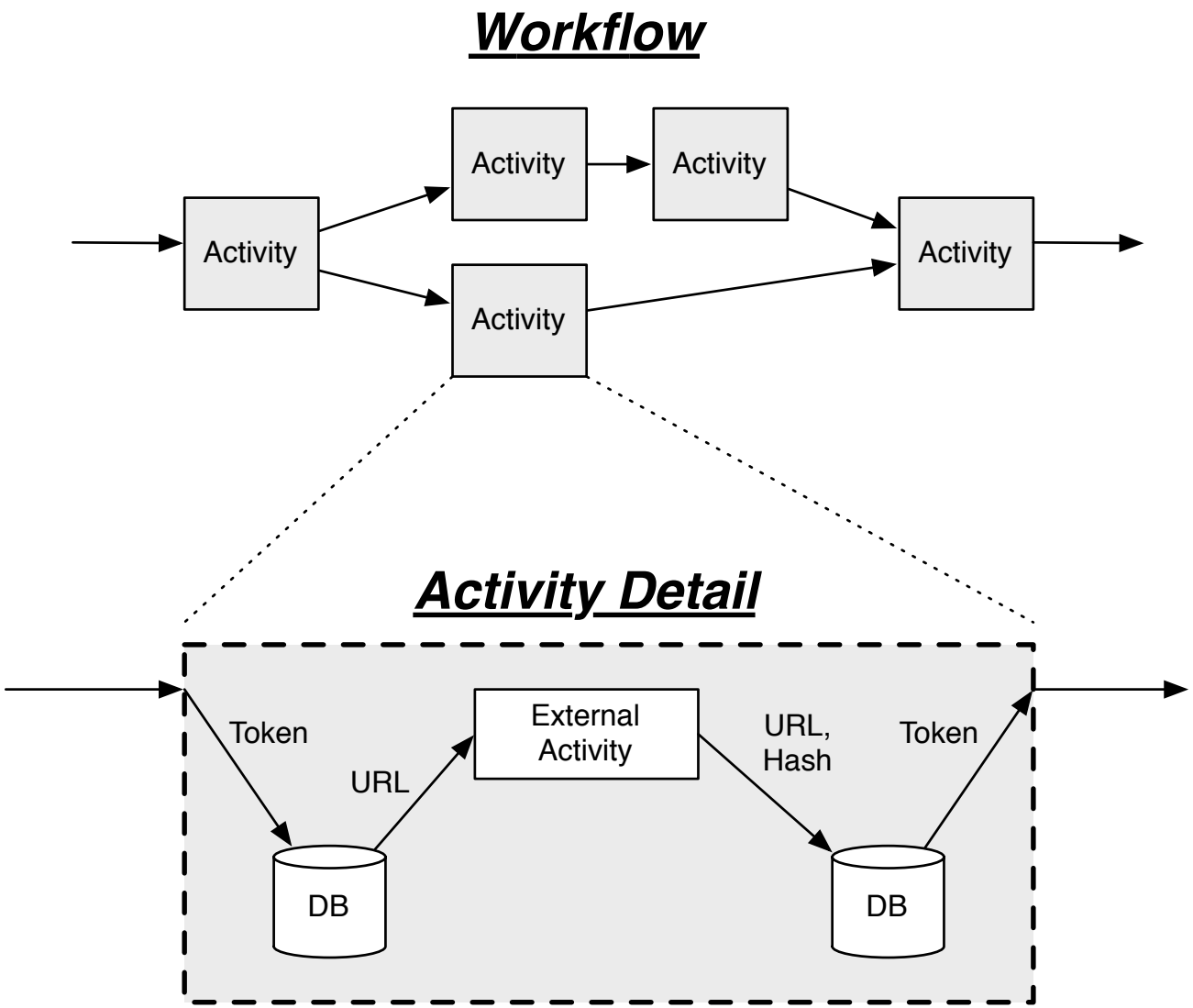
When we allow side-effects on the streams, we can also build conveniently non-deterministic primitives too.

Step 2: Haskell becomes coordinator

The next step is to migrate away from a pure-Haskell model to one in which:

- Data resides outside Haskell (possibly on disk or in a database).
- Computational tasks take place outside Haskell, possibly in a different machine than that which manages the coordination.

External program binding



Coordinating non-Haskell programs

Haskell acts as coordinator of computations.

External data are represented in Haskell as tokens.

Tokens

```
type Hash = String
type Token = (FilePath,Hash)
```

Tokens are resolved into a concrete URI, which is passed to the external program. The result is packaged up as a new token.

Hashes (e.g., MD5 sum) are used to uniquely identify data and to memoize computations for performance gains.

The data hash may be put to other uses as well, e.g. to track provenance.

Persistent external programs

We wish to avoid high overhead for starting and stopping external activities if they will be called repeatedly.

External process interaction

```
do
  (Just stdIn, Just stdOut, Just stdErr, h) <-
    createProcess (proc path args)
    {std_in  = CreatePipe,
     std_out = CreatePipe,
     std_err = CreatePipe}

  -- code interacting via pipes w/ proc ...

  terminateProcess h
```

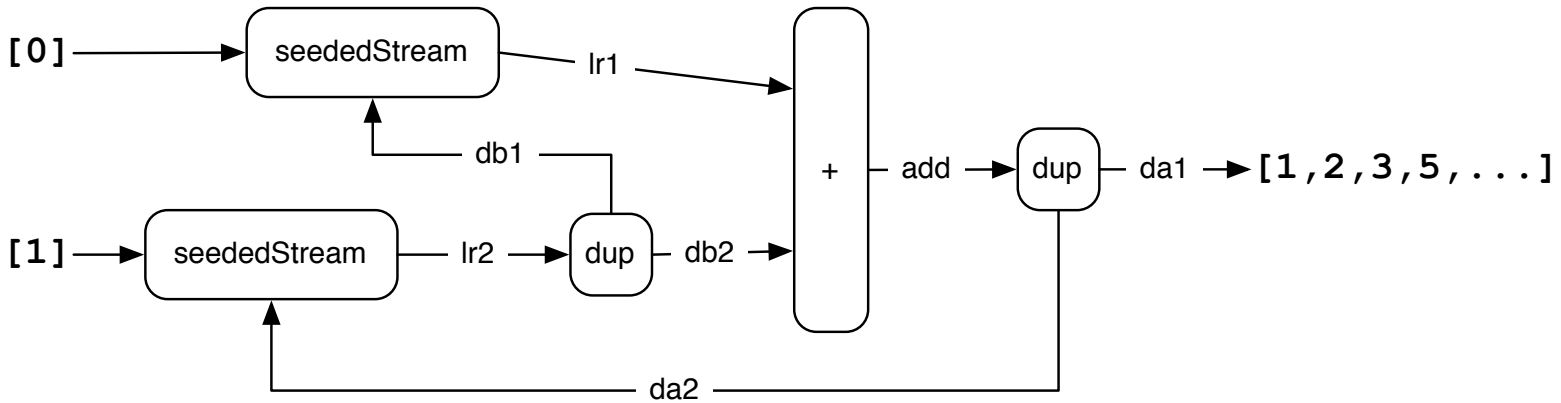
Processes can be created and interacted with via pipes for as long as we would like. Programs written (or wrapped) to support this style of interaction can avoid start/stop overhead.

Examples

Now we will show some workflows entirely based on Haskell to illustrate the primitives.

- Fibonacci number generation
- Prime number generation (*time permitting*)

Example: Fibonacci sequence



This very simple Haskell workflow computes a stream of Fibonacci numbers.

This is a conveniently simple example to illustrate the code.

Example: Fibonacci sequence

Fibonacci workflow

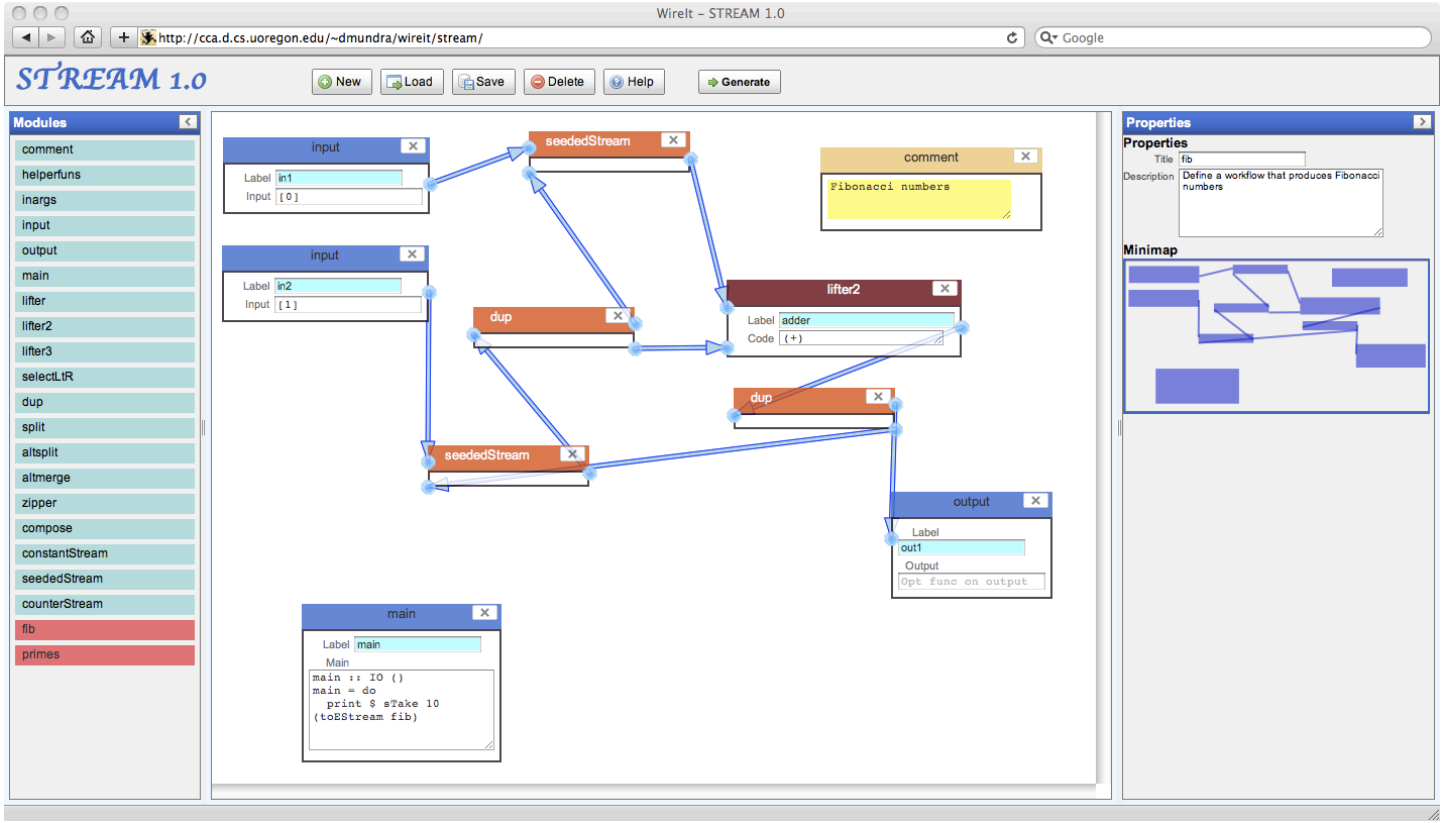
```
import WF.Primitives
import WF.Types

adder :: (Num a, Streamer s) => s a -> s a -> s a
adder = lifter2 (+)

fib :: (Num a, Streamer s) => s 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

main :: IO ()
main = do
  let fibs = fib :: PStream Int
  let (f,_) = sAdvanceN 10 fibs
  print $ f
```

Example: GUI Builder



Example workflow generated using a *Wire-It* web-based GUI. This GUI generates Haskell code like on the previous slide.

GUI users need not know Haskell to create workflows.

Example: Fibonacci sequence

Demo: the difference between effectful and pure streams.

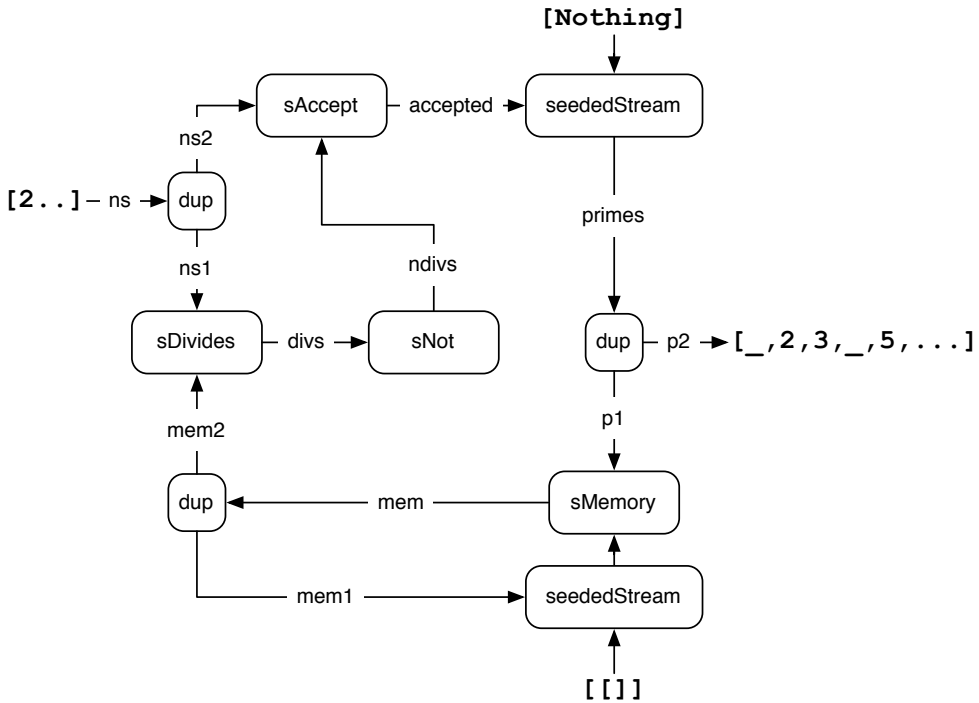
Fibonacci workflow

```
ghci> let fibs = fib :: PStream Int
ghci> let (f1,_) = sAdvanceN 10 fibs
ghci> f1
[1,2,3,5,8,13,21,34,55,89]
ghci> let (f2,_) = sAdvanceN 10 fibs
ghci> f2
[1,2,3,5,8,13,21,34,55,89]

ghci> let fibs = fib :: EStream Int
ghci> let (f1,_) = sAdvanceN 10 fibs
ghci> f1
[1,2,3,5,8,13,21,34,55,89]
ghci> let (f2,_) = sAdvanceN 10 fibs
ghci> f2
[144,233,377,610,987,1597,2584,4181,6765,10946]
```

For the pure `PStream`, the first `sAdvanceN` does not effect the second. For `EStreams`, it does.

Example: Prime sequence



More interesting than Fib due to the feedback loop necessary to remember primes we've already seen, and the notion of "bubbles".

Example: Prime sequence

Let's start by defining some functions outside the workflow context as our activities.

Primes

```
divides :: Int -> Int -> Bool
divides x y = x `mod` y == 0

dividesByN :: [Int] -> Int -> Bool
dividesByN ys x = or (map (divides x) ys)

accept :: a -> Bool -> Maybe a
accept k True  = Just k
accept _ False = Nothing
```

Example: Prime sequence

Use the lifters to bring them into the world of streams.

Primes

```
sDivides :: (Streamer s) => s [Int] -> s Int -> s Bool
sDivides = lifter2 dividesByN

sAccept :: (Streamer s) => s Int -> s Bool -> s (Maybe Int)
sAccept = lifter2 accept

sNot :: (Streamer s) => s Bool -> s Bool
sNot = lifter not
```

These are now proper "boxes" to be composed.

Example: Prime sequence

We need that feedback loop for memorizing what primes we've seen so far.

Primes

```
memorize :: [a] -> Maybe a -> [a]
memorize mems (Just x)  = x:mems
memorize mems (Nothing) = mems

sMemory :: (Streamer s) => s [Int] -> s (Maybe Int) -> s [Int]
sMemory = lifter2 memorize
```

Note the use of the `Maybe` type. In the stream world, I call streams that use Maybe ones that support "bubbles".

Example: Prime sequence

Bubbles are useful, as they allow the activities to proceed and do work even though they may be acting on the case where nothing interesting is arriving into them.

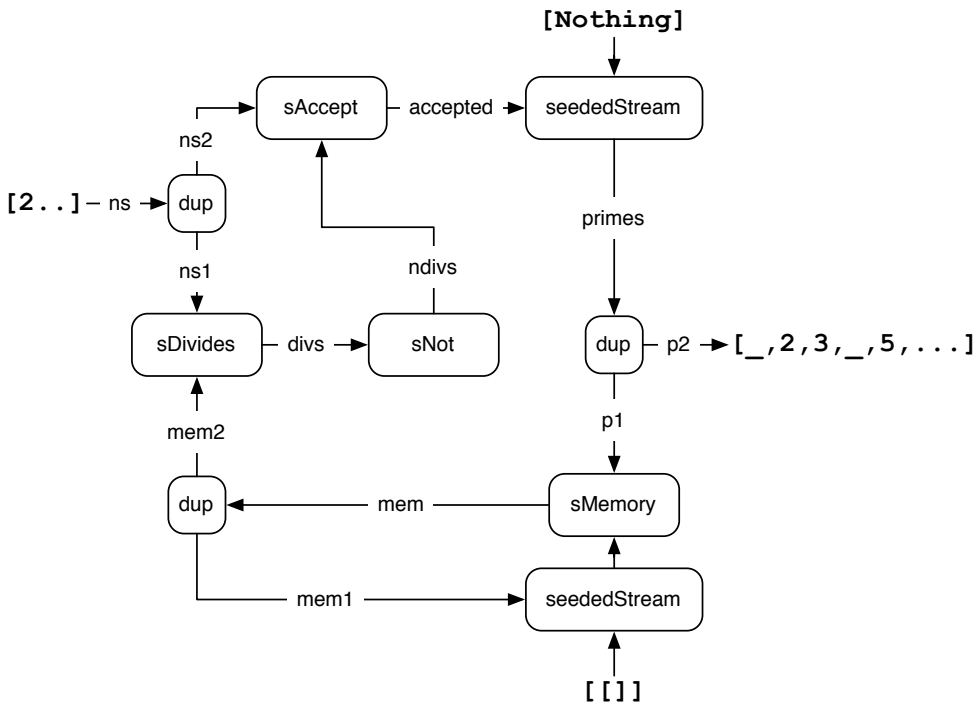
In practice though, we may want to not see them. So, we introduce a new primitive that yields a bubble-free stream.

The de-bubbler

```
sDebubble :: (Streamer s) => s (Maybe a) -> s a
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)
```

Example: Prime sequence

Composing everything together, we get the following flow diagram.



Code shown next slide.

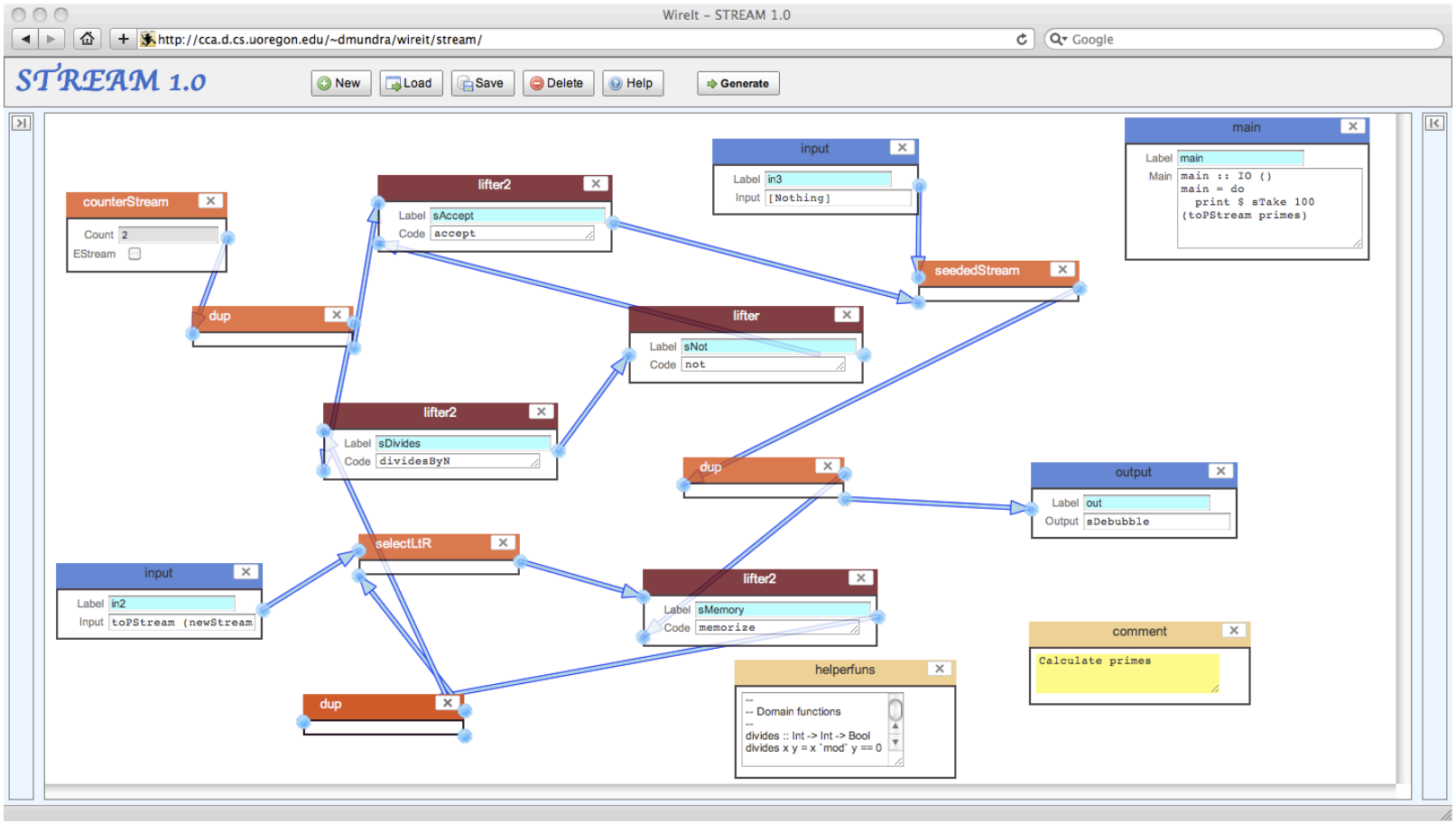
Example: Prime sequence

Putting it all together, we can then build a workflow that corresponds to the diagram.

Primes

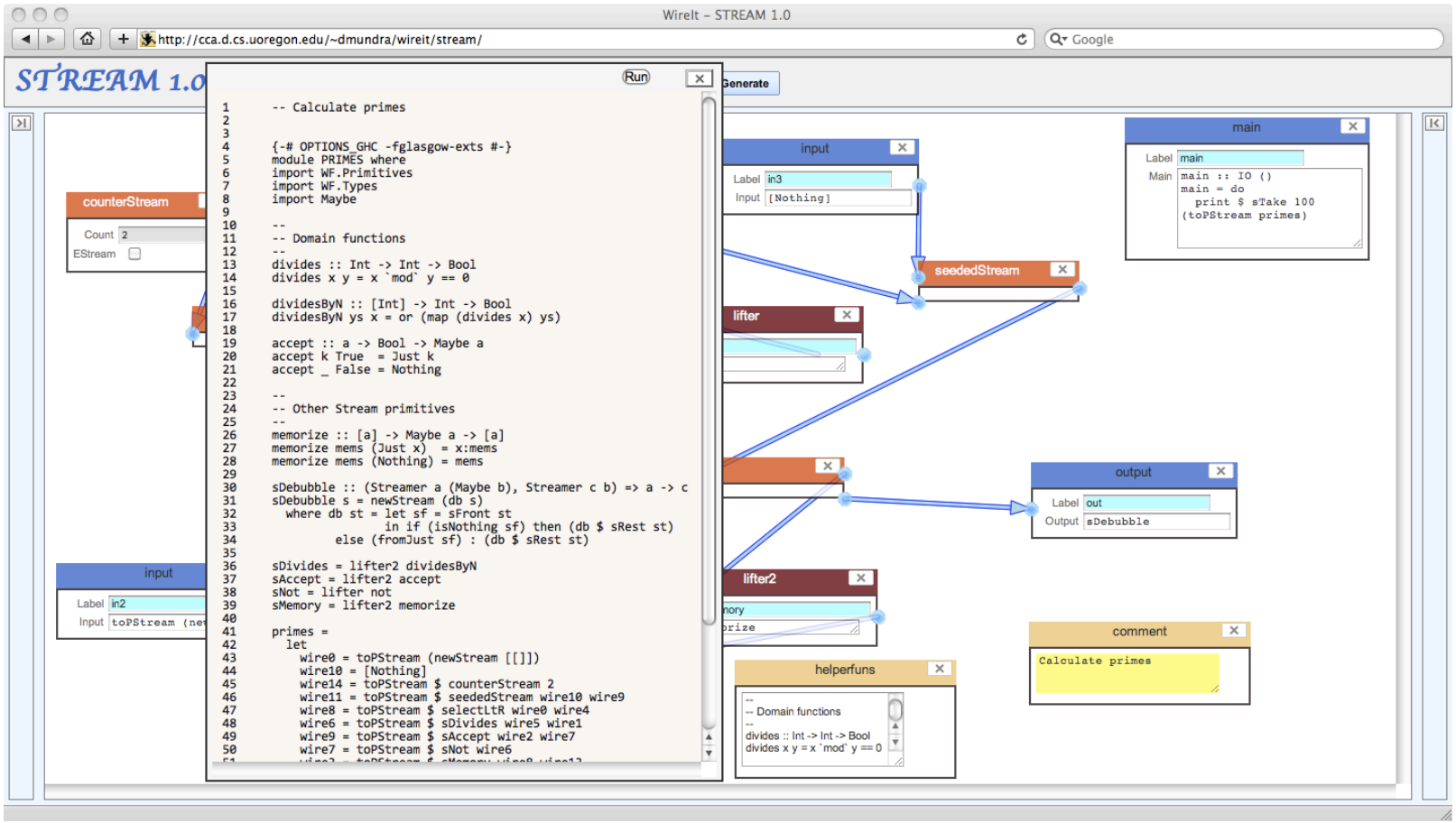
```
primes :: (Streamer s) => s Int
primes = let ns = newStream [2..]
             empty = newStream [[]]
             (ns1,ns2) = dup ns
             mem = sMemory (selectLtR empty mem1) p1
             (mem1,mem2) = dup mem
             divs = sDivides mem2 ns1
             ndivs = sNot divs
     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

Example: GUI Primes



Example workflow generated using a *Wire-It* web-based GUI.

Example: GUI Primes



Showing autogenerated Haskell in GUI.

Concluding remarks

We learned that:

- Haskell provides a very clean platform to implement stream-based workflow representations.
- Type classes are convenient for building generic primitives to embed a DSL in Haskell.
- Token-passing allows us to use this representation for coordination and composition, with a clean binding to the outside world.

Work remains to more cleanly deal with effects, in particular, the interaction of lazy stream semantics and strict evaluation of IO actions. This is the subject of ongoing work.

Thank you! Questions?