Workflow representation and runtime based on lazy functional streams

Matthew J. Sottile, Geoffrey C. Hulette, Allen D. Malony

University of Oregon
Department of Computer and Information Science

November 15, 2009

Introduction

This talk presents a representation of workflows based on lazy functional streams using Haskell.

We would like to acknowledge Zena Ariola, Dan Keith, and Daniel Mundra, all at UO, for contributions to the development of this work.

This talk will be brief, and will cover the essential findings of our work. The paper has far more detail.

Workflow representation

How well do functional streams represent the relationships between computational activities in a workflow?

Researchers answered this question over a decade ago: there is a correspondence between streams and workflow.

- Streams are infinite lists in functional languages.

We wished to explore the implications of Haskell on this representation:

- Static, strict typing
- Laziness
- A large library of existing list primitives

The journey of a computation

Let's start at the beginning.

A simple function

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

Takes two integers, produces an integer. Nothing terribly special.

Streams

Singletons are boring, so let's consider our function on unbounded lists.

Streams are good for that. How does our simple function look with streams?

A simple stream-based function

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

Streams

These are pretty easy to work with.

A simple stream-based function

```
*Main> let adr = adder [1..] [10..]

*Main> take 15 adr
[11,13,15,17,19,21,23,25,27,29,31,33,35,37,39]
```

Haskell makes working with infinite structures nice due to its lazy evaluation model. You evaluate what you need, as you need it.

This can be modeled in non-lazy languages, but that may require more work. Haskell gives it to us for free.

Pattern for streams

A generic pattern emerges upon examination.

A move towards the generic

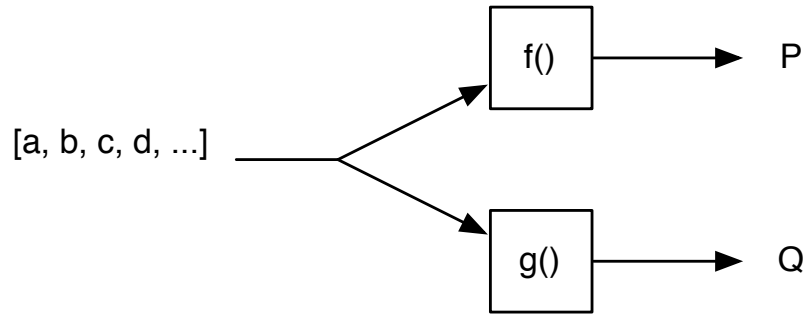
```
streamify :: [a] -> [a] -> (a -> a -> a) -> [a]
streamify (x:xs) (y:ys) f =
    (f x y):(streamify xs ys f)
```

The type signature may be a bit obtuse to non-FP people.

- 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? Is:

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

A stream typeclass

We can experiment with the different stream semantics by defining a simple typeclass representing a stream.

Streamer typeclass

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

Our workflow primitives, like splitting and merging streams, can be defined generically as operating on Streamers.

Pure stream instance

Pure streams are easy! They're just our usual 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: Pure = no side effects.

Effectful stream instance

Effectful streams make this exercise interesting. We need to take a look at these in parts.

Effectful stream instance

```
import Control.Concurrent.MVar

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

Pretty straightforward — we're stashing a pure stream inside a bin that we control effects on.

An `MVar` should be thought of as a slot that can be accessed by multiple threads in a controlled manner.

Effectful stream instance

Making a new stream is easy.

Implementing `newStream`

```
do store <- newEmptyMVar
   putMVar store l
   return store
```

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

Effectful stream instance

Implementing `sAdvance`

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

`modifyMVar` wraps the act of taking the value out of the MVar, applying a function to the contents, and putting the result back into the MVar.

Revisiting streamify (unary case)

We can define our function to lift singleton functions to the stream model based on the type classes.

A simple stream-based function

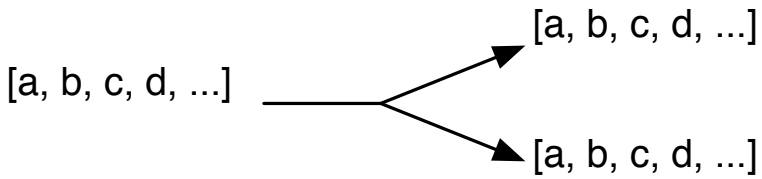
```
streamify :: Stream a -> (a -> a) -> Stream a
streamify s f = (f x):(streamify xs)
  where
    (x, xs) = sAdvance s
```

Where `sAdvance` corresponds to advancing the stream, popping off the next value and the state of the stream containing the rest.

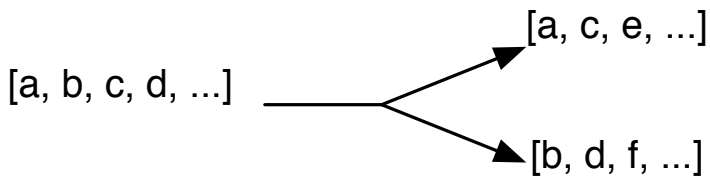
Critical observation: this function doesn't specify which instantiation of Streamer it works on. It functions for both!

Primitives

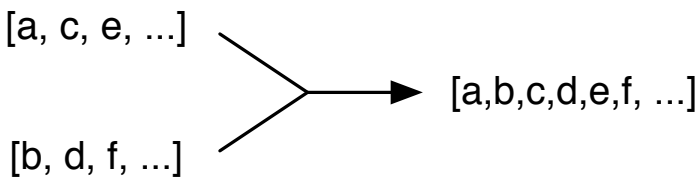
Duplicating split



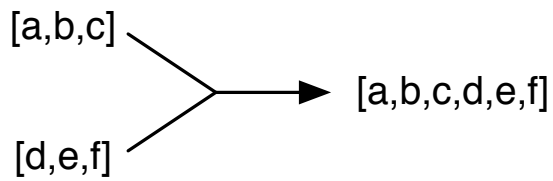
Alternating split



Alternating merge



Biased merge



We can build primitives out of 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.

Coordinating non-Haskell programs

Realistically, Haskell acts as coordinator of "real" computations.

Workflow representation based on coordinating flow of tokens.

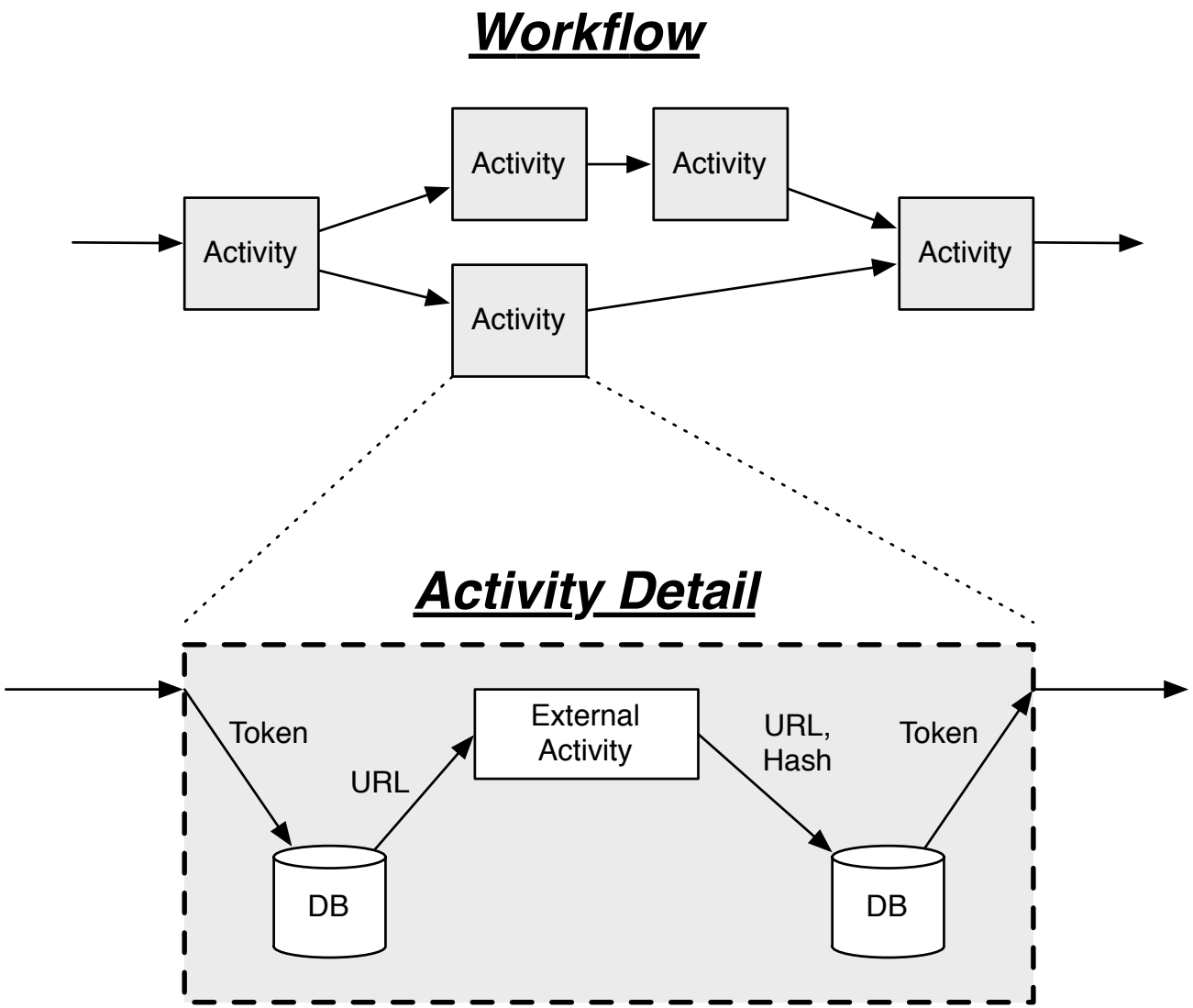
Tokens

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

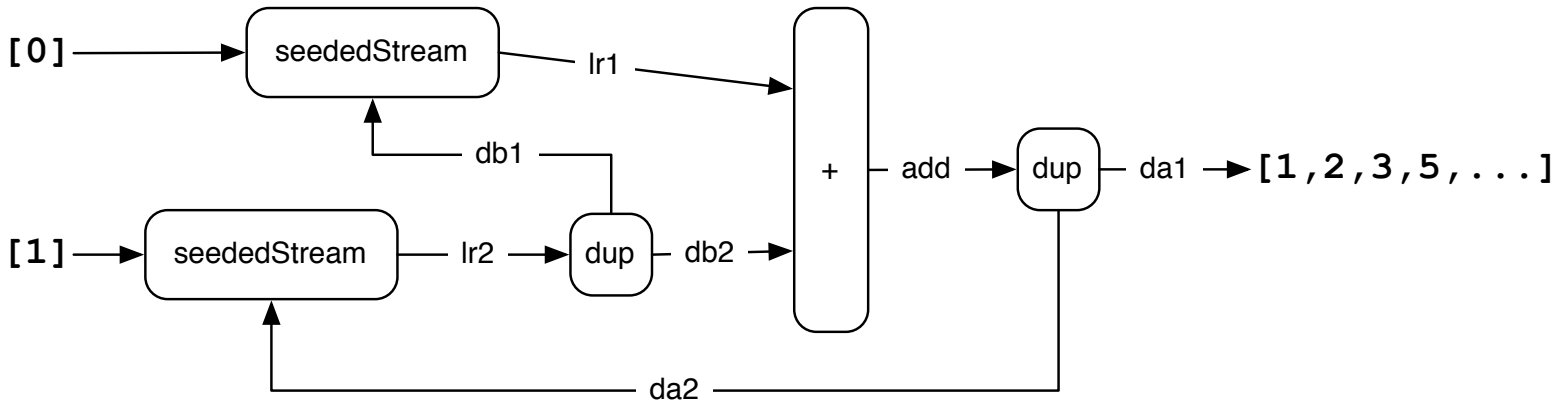
Binding to external programs by resolving token into concrete URI for data, invoking external computation, and building new token representing results.

Hashes (e.g., MD5 sum) are used to uniquely identify data for tracking provenance and memoizing computations across runs to eliminate redundant work.

External program binding



Example: Fibonacci sequence



Above is a flow diagram for a simple native Haskell workflow that computes Fibonacci numbers.

This is a conveniently simple example to illustrate the code.

Example: Fibonacci sequence

Fibonacci workflow

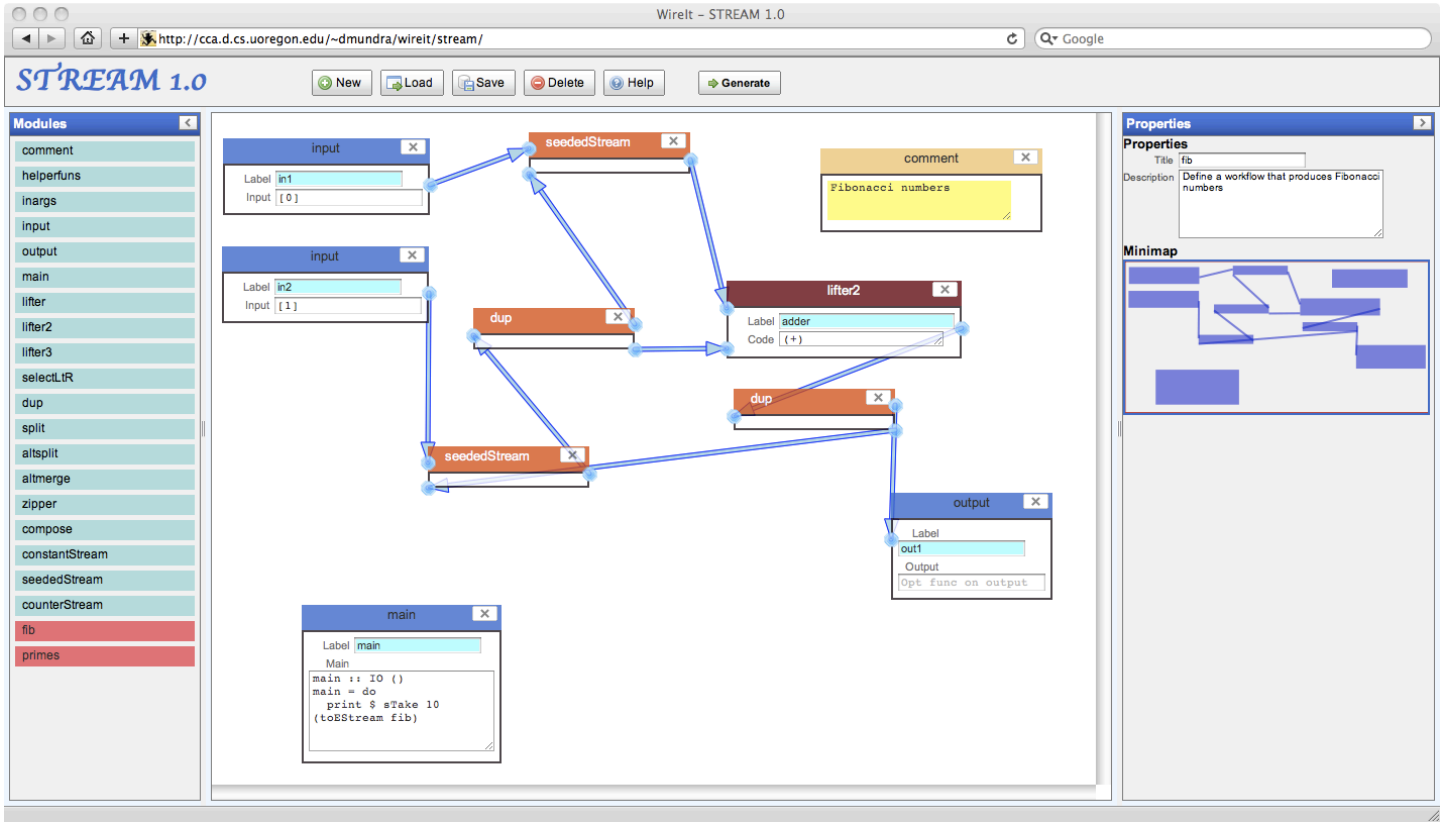
```
import WF.Primitives
import WF.Types

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

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

main :: IO ()
main = do
  print $ sTake 10 (fib :: EStream Int)
```

Example: GUI Builder



Example workflow generated using a *Wire-It* web-based GUI. This GUI generates Haskell code like on the previous slide. Users of GUI are insulated from seeing the underlying Haskell.

Example: Fibonacci sequence

This is a good demonstration of our effectful vs pure streams.

Fibonacci workflow

```
*Main> let f = fib :: PStream Int
*Main> sTake 10 f
[1,2,3,5,8,13,21,34,55,89]
*Main> sTake 10 f
[1,2,3,5,8,13,21,34,55,89]

*Main> let f = fib :: EStream Int
*Main> sTake 10 f
[1,2,3,5,8,13,21,34,55,89]
*Main> sTake 10 f
[144,233,377,610,987,1597,2584,4181,6765,10946]
```

We see that in the pure case, the act of taking has no effect on the stream. When we instead use effectful streams, the act of taking from the stream is observable by subsequent operations.

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.
- 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 the proper "Haskell way" (aka, monadic code). This is part of our current work.

Thank you! Questions?