Lucian: Dataflow and Object-orientation

Dominic Orchard

dominic.orchard@cam.ac.uk

Computer Lab, University of Cambridge

BCTCS '09

Language interoperation

- No one language is all things to all people/programs
- Some languages/paradigms are more appropriate for a certain class of program
- The best of both worlds....
  - Language interoperation
  - Multi-paradigm languages
- But how to interoperate and combine paradigms?

Lucian

- Interoperates dataflow and object-orientation
  - Declarative dataflow à la Lucid
  - Interoperated with bject-orientated programming
- What do the two paradigms mean with respect to each other?

LUCID, The dataflow programming language

Wadge, W.W., and Ashcroft, E. A. LUCID, The dataflow programming language. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

- A declarative dataflow language
- Iteration and variable update is declarative
- Lazy, demand driven

Declarative iteration and assignment

Procedural

Multiple scattered statements

```
x = 2
....
x = 0
....
loop:
  .....
  x += 1
  .....
```

Lucid

Single inseparable definition

```
x = 2 fby 0 fby x + 1
```

One interpretation of LUCID: Streams

- Declarative language of infinite streams
- Expressions return streams
- Operators are independent processes, connected with streams
- Programs are *continuous*
- Asynchronous

LUCID operations as stream operations

Constant Streams

$$[\![1]\!] = \langle 1, 1, 1, \ldots \rangle$$

Arithmetic/boolean binary operators applied pointwise

$$[\![\mathtt{x} + \mathtt{y}]\!] = \langle \mathtt{x}_0 + \mathtt{y}_0, \mathtt{x}_1 + \mathtt{y}_1, \ldots \rangle$$

Conditional construct (pointwise)

$$[\![\texttt{if b then x else y}]\!] = \langle \begin{cases} \mathtt{x}_0 & \text{if } \mathtt{b}_0 \\ \mathtt{y}_0 & \text{otherwise} \end{cases}, \ldots \rangle$$

LUCID operations as stream operations

"First" operator - repeat first element (head)

$$[\![\texttt{first x}]\!] = \langle x_0, x_0, \ldots \rangle$$

"Next" operator - skip first element (tail)

$$[\![\texttt{next x}]\!] = \langle x_1, x_2, \ldots \rangle$$

"Followed by" - `fby` - (cons)

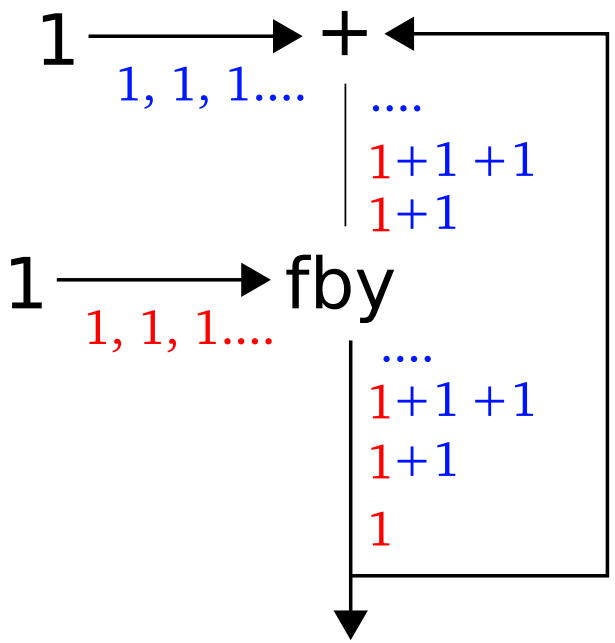
$$[\![\texttt{x fby y}]\!] = \langle x_0, y_0, y_1 \ldots \rangle$$

Classic example: Natural numbers

Example

```
n = 1 fby (n+1)
```

Remember: $[\![\texttt{x fby y}]\!] = \langle x_0, y_0, y_1 \ldots \rangle$

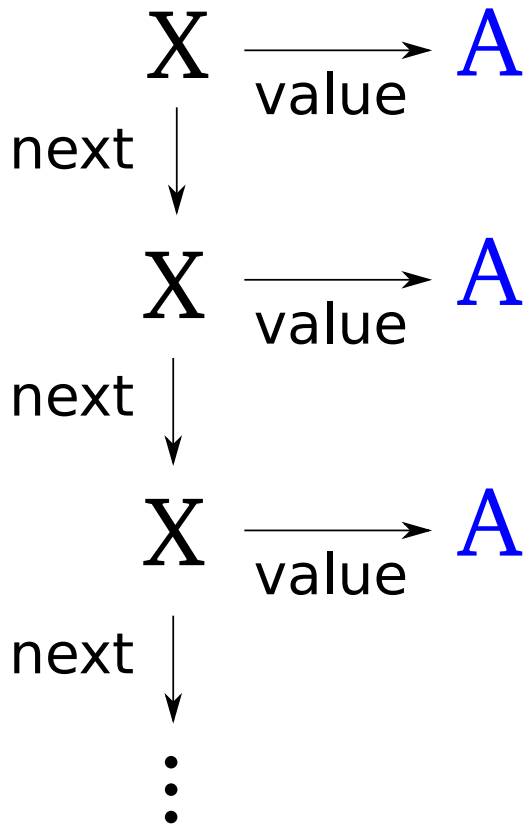


Interoperate Lucid dataflow with objects?

- How are objects and streams related?
- What are objects and streams formally?

Stream data structure

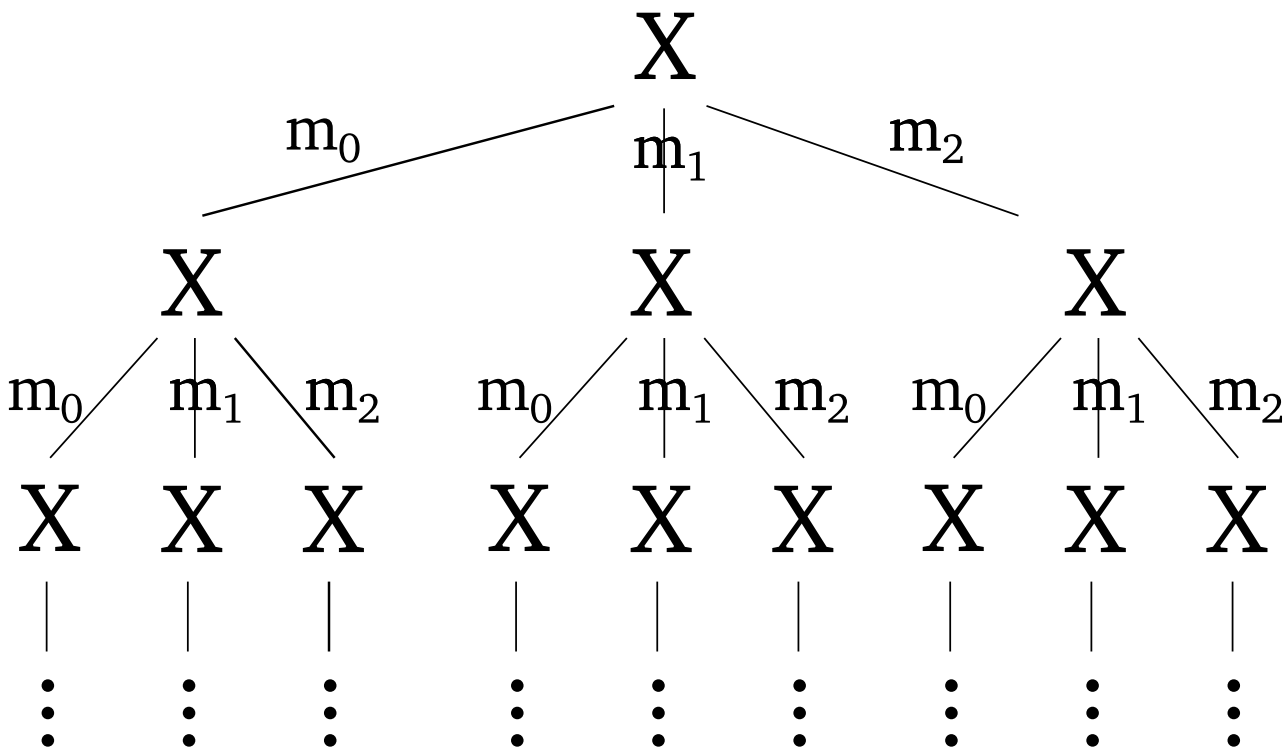
- Blackbox with hidden state X
- Observe a value A
- Transition to next state X
- Keep asking for a next state and observation



Functional objects

- Blackbox with hidden state (private data/methods)
- Observe attributes
- Immutable
- Method calls return new object i.e. transform/transition to next object state
- Turn imperative OO into pseudo-functional OO using deep cloning

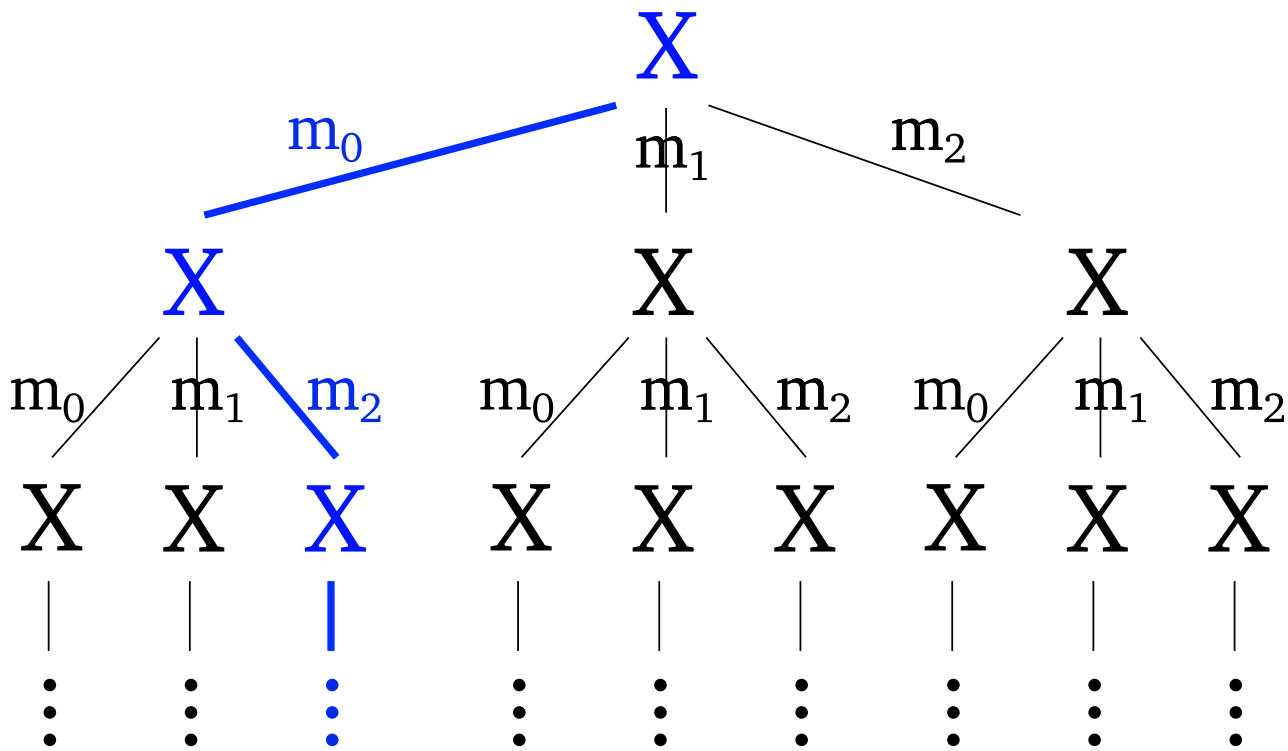
Functional objects - Infinite tree



Objects and Streams

- Objects generalise streams
  - Objects have more transformations and observations
- Therefore, can embed Lucid streams into OO.

Single path of object tree = stream



- Can describe path (history) of an object's transformations using Lucid dataflow equations

Lucian

- Underlying computational device: objects
- Lucid streams embedded into functional objects
  - *attribute* = *value*
  - *method* = *next*
- **Dataflow objects:** describe object transformations in a single definition
- Single definition is a stream of object "snapshots"

Dataflow objects

Procedural

Multiple scattered statements

```
x = new X
....
x.m1(...)
.....
x.m2(...)
.....
```

Lucian

Single inseparable definition

```
x = new X <<= m1(...)
          fby m2(...)
```

Interoperation

- Has no class definitions of its own.
- Instantiate objects defined in some OO language.

Some OO language

```
class BankAccount:

  balance = 0

  credit(x) = ...
  debit(x) = ...
```

Lucian

```
o = new BankAccount ...
.....
```

Dataflow Object : Declaration

- Describe an object's life in a single definition

Example

`new` *class*

`new` *class* `<<=` *methods*

$$[\![\texttt{new}\ \textit{class}]\!] = \langle \mathrm{BA}_0, \mathrm{BA}_0, \mathrm{BA}_0, \ldots \rangle$$

where BA_0 is a newly instantiated `BankAccount` object.

Dataflow Object : Declaration

- Give list of method calls after <<=

Example

```
ba = new BankAccount <<= c(2) fby d(6) fby id}
```

$$[\![\texttt{ba}]\!] = \langle \text{BA}_0, \text{BA}_0.c(2), \text{BA}_0.c(2).d(6), \text{BA}_0.c(2).d(6), \ldots \rangle$$

where BA_0 is a newly instantiated `BankAccount` object.

Dataflow Object : Attribute reference

Example

`ba.balanace`

$$[\![\texttt{ba.balance}]\!] = \langle \text{ba}_0.\mathit{balance}, \text{ba}_1.\mathit{balance}, \text{ba}_2.\mathit{balance}, \ldots \rangle$$

- Attribute at each object snapshot

Dataflow Object : Pointwise methods

- Apply a method pointwise as opposed to giving a stream of method calls

Example

```
ba' = ba.credit(param)
```

$$[\![\text{ba'}]\!] = \langle \text{ba}_0.c(p_0), \text{ba}_1.c(p_1), \ldots \rangle$$

- No longer a dataflow object, just a stream of object snapshots

Example

```
ba = new BankAccount <<= credit(10)
                         fby debit(2)
                         fby debit(20)
                         fby id

balance = ba.balance
overdrawn? = ba.balance < 0
```

$$[\![\texttt{balance}]\!] = \langle 0, 10, 8, -12, -12, \ldots \rangle$$

$$[\![\texttt{overdrawn?}]\!] = \langle \mathit{false}, \mathit{false}, \mathit{false}, \mathit{true}, \mathit{true}, \ldots \rangle$$

Conclusions

- Lucian is an OO-dataflow language extending Lucid
- Has a declarative *dataflow* view of objects
  - Define object and its transformations in a single declaration
- Lucian embeds streams into objects i.e. Lucid interpreted with objects

Application

- Equational reasoning on object behaviours
- Object versioning, and "rollback"
- Reactive, continuous programming with objects
- Hybrid programming: combining dynamic, reactive programs with discrete units of functionality: objects
- Benefits of IO and libraries from OO into Lucid
- Coarse grained parallelism

Further Work

- Coalgebraic semantics for reasoning
- Invariant equations on object behaviour for further reasoning
- Typing rules for when a stream of snapshots is "historical" or not
- Real world OO is not well behaved: global side effects, aliasing
- Current implementation is old and broken in places
- Parallel implementation on stream processors

eod