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


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

<table>
<thead>
<tr>
<th>Procedural</th>
<th>Lucid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple scattered statements</td>
<td>Single inseparable definition</td>
</tr>
</tbody>
</table>

\[
x = 2 \\
\ldots \\
x = 0 \\
\ldots \\
\text{loop:} \\
\ldots \\
x += 1 \\
\ldots \\
\]

\[
x = 2 \text{ fby} 0 \text{ fby} x + 1 \\
\]
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

\[
[x + y] = \langle x_0 + y_0, x_1 + y_1, \ldots \rangle
\]

Conditional construct (pointwise)

\[
[\text{if } b \text{ then } x \text{ else } y] = \langle \begin{cases} x_0 & \text{if } b_0 \\ y_0 & \text{otherwise} \end{cases}, \ldots \rangle
\]
LUCID operations as stream operations

“First” operator - repeat first element (head)

\[[\text{first } x] = \langle x_0, x_0, \ldots \rangle\]

“Next” operator - skip first element (tail)

\[[\text{next } x] = \langle x_1, x_2, \ldots \rangle\]

“Followed by” - \(\text{fby}\) - (cons)

\[[x \text{ fby } y] = \langle x_0, y_0, y_1 \ldots \rangle\]
Classic example: Natural numbers

Example

\[ n = 1 \text{ fby } (n+1) \]

Remember: \([x \text{ fby } y] = \langle x_0, y_0, y_1, \ldots \rangle\)
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.
Lucian: Dataflow and Object-orientation

Dominic Orchard

Introduction

Lucid Stream interpretation

Streams and Objects

Lucian Declaration Attributes Pointwise methods

Conclusions

Application

Further Work

Single path of object tree = stream

- Can describe path (history) of an object’s transformations using Lucid dataflow equations
• Underlying computational device: objects
• Lucid streams embedded into functional objects
  • \textit{attribute} = \textit{value}
  • \textit{method} = \textit{next}

\textbf{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

```python
class BankAccount:
    balance = 0
credit(x) = ...
debit(x) = ...
```

Lucian

```python
o = new BankAccount ...
.....
```
Dataflow Object: Declaration

- Describe an object's life in a single definition

Example

\[
\text{new class}
\]

\[
\text{new class} \quad <= \quad \text{methods}
\]

\[
[\text{new class}] = \langle \text{BA}_0, \text{BA}_0, \text{BA}_0, \ldots \rangle
\]

where \(\text{BA}_0\) 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
```

```
[b a] = ⟨BA₀, BA₀.c(2), BA₀.c(2).d(6), BA₀.c(2).d(6), . . .⟩
```

where BA₀ is a newly instantiated BankAccount object.
Dataflow Object: Attribute reference

Example

\[ ba.balanace \]

\[ [ba.balanace] = \langle ba_0.balance, ba_1.balanace, ba_2.balanace, \ldots \rangle \]

- Attribute at each object snapshot
Dataflow Object: **Pointwise methods**

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

**Example**

\[ \text{ba}' = \text{ba}.\text{credit}(\text{param}) \]

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

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

\[
\text{ba} = \text{new BankAccount} \ll\ll \text{credit}(10) \\
\text{fby debit}(2) \\
\text{fby debit}(20) \\
\text{fby id}
\]

\[
\text{balance} = \text{ba.balance} \\
\text{overdrawn?} = \text{ba.balance} < 0
\]

\[
\text{[[balance]]} = \langle 0, 10, 8, -12, -12, \ldots \rangle \\
\text{[[overdrawn?]]} = \langle \text{false, false, false, true, true, \ldots} \rangle
\]
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
Lucian: Dataflow and Object-orientation

Dominic Orchard

Introduction

Lucid

Stream interpretation

Streams and Objects

Lucian

Declaration
Attributes
Pointwise methods

Conclusions

Application

Further Work

eod