Math.comput.sci. 2 (2008), 103–122
© 2008 Birkhäuser Verlag Basel/Switzerland
1661-8270/010103-20, *published online* November 4, 2008
DOI 10.1007/s11786-008-0046-6

Mathematics in Computer Science

Integrating Lucid's Declarative Dataflow Paradigm into Object-Orientation

Dominic A. Orchard and Steve Matthews

Abstract. The dataflow language Lucid applies concepts from intensional logic to declarative ISWIM expressions which are intensionalised relative to the dimension of time, thus introducing the notion of an expression's history. Lucian, a language derived from Lucid, embeds dataflow into object-orientation allowing the intensionalisation of objects. Lucian introduces the notion of a declarative intensional object as the history of an object's transformations. This paper discusses the embedding relationships and semantics of conjoining the dataflow and object-oriented paradigms to provide the language Lucian for defining intensional objects.

Mathematics Subject Classification (2000). 68N15, 68N19, 68Q55.

Keywords. Semantics, interoperation, object orientation, dataflow, intensional programming, intensional objects.

1. Introduction

In 1966 Landin introduced the ISWIM family of languages, presenting a general framework for functional, declarative programming [11]. Landin put forward the notion that the development of languages should come from a "well-mapped" space, such as a family of languages, rather than developing a language from the ground up. During the 1980s Bill Wadge and Ed Ashcroft developed Lucid, a dataflow language built upon the ISWIM framework introducing the concept of inten*s*ionality from intensional logic [14]. Lucid intensionalises declarative ISWIM expressions facilitating the definition of computation in terms of a history of an expression's evaluation in an implicitly changing contextual environment. This paper introduces the concept of *declarative intensional objects* expressed in a language that builds upon Lucid to interoperate with object-orientation thus elucidating the history of an object's transformations.

The language presented in this paper, which we have named Lucian[1], is a functional, declarative, dataflow language derived from Lucid and the ISWIM family of languages. Lucian facilitates interoperation of dataflow and object-oriented paradigms, integrating the merits of Lucid and the dataflow paradigm into the prevalent programming paradigm of our age – object-orientation.

Generally language interoperation allows disparate languages to cooperate thus enabling specification of a computation in the form in which it is most naturally expressed and most efficiently computed. This cooperative interoperation can be seen in high-level languages that present an interface to a lower-level language such as the Haskell foreign function interface to C [12]. While the higher-level language may have many desirable features certain concepts may be expressed more succinctly and calculated more efficiently in a lower-level language. The interoperation of dataflow programming with object-orientation seeks to combine their respective expressive powers. While dataflow provides a declarative, implicitly parallel, state-free approach to programming with definitions and functional relationships, object-orientation provides procedural, state-based programming with the abstraction of objects. The motivation behind Lucian is not just to explore an interesting case in the semantics of interoperation, but is also motivated by the goal to ameliorate both dataflow and object-orientation by providing an "escape" from one to the other to allow each paradigm to do what it does best.

The relationship of ISWIM, to Lucid, to Lucian in terms of language constructs is described by the following relation which illustrates the concepts each introduces and presents the etymology of our term *declarative intensional objects*:

$$\begin{array}{ccccc} \mathsf{declarative} & & \mathsf{intensional} & & \mathsf{objects} \\ \text{ISWIM} & \subset & \text{Lucid} & \subset & \text{Lucian} \end{array}$$

Lucid builds upon the concepts put forward by the ISWIM family of languages; Lucian builds upon the concepts put forward by Lucid.

The interoperation of object-orientation with dataflow is described by *embedding* relationships; Lucian embeds dataflow into object-orientation via the concept of *declarative intensional objects*. This conceptual embedding relationship is instantiated in the current proof-of-concept implementation by embedding a Kahn-dataflow subset of single dimension, eager Lucid into the object-oriented language Ruby as is illustrated in Figure 1.

Interoperation is also achieved via a converse embedding relationship in which object-orientation is embedded into dataflow via *object stream filters* c.f. Figure 2. The converse embedding relationship of object-orientation into dataflow is not isomorphic with the former. We delineate the embedding operations in Figures 1 and 2 by naming them *embeds* and *embeds'* respectively.

Section 2 of this paper gives, for the uninitiated, a background to this research, briefly introducing Kahn dataflow, ISWIM, Lucid, intensionality, and object-orientation. The main discussion of the paper takes place in Sections 3 and 4.

[1]The name *Lucian* is a portmanteau of the dataflow language *Lucid* and the artist *Lucian Freud*, known for depicting people and plants in unusual juxtapositions – analogous to juxtaposing streams and objects.

$$\begin{tikzcd}
\text{Object Orientation} \arrow[r, "\textit{instantiated by}"] \arrow[d, "\textit{embeds}"'] & \text{Ruby} \arrow[d, "\textit{embeds}"] \\
\text{Dataflow} \arrow[r, "\textit{instantiated by}"'] & \text{Lucid}
\end{tikzcd}$$

Figure 1. Embedding dataflow into object-orientation. The aim and methodology of Lucian described in an informal commutative diagram.

$$\begin{tikzcd}
\text{Dataflow} \arrow[r, "\textit{instantiated by}"] \arrow[d, "\textit{embeds'}"'] & \text{Lucid} \arrow[d, "\textit{embeds'}"] \\
\text{Object Orientation} \arrow[r, "\textit{instantiated by}"'] & \text{Ruby}
\end{tikzcd}$$

Figure 2. Converse: Embedding object-orientation into dataflow.

The semantics of embedding dataflow into object-orientation (Figure 1) are covered in Section 3. The converse relationship of embedding object-orientation into dataflow (Figure 2) is covered in Section 4. Illustrative examples are presented but are kept short to lessen the dependence of the example's worth on the object-oriented language Ruby which the reader may be unfamiliar with. Section 5 details some possible useful applications of Lucian. Information on the proof-of-concept implementation is given in Section 6 followed by related work, concluding remarks, and further work.

2. Background

2.1. The early years of the dataflow paradigm: Kahn dataflow

The dataflow programming paradigm stems from research carried out over the last 30 years. A prevalent variety of dataflow is based around ideas set forward by Gilles Kahn in his seminal paper: *The semantics of a simple language for parallel*

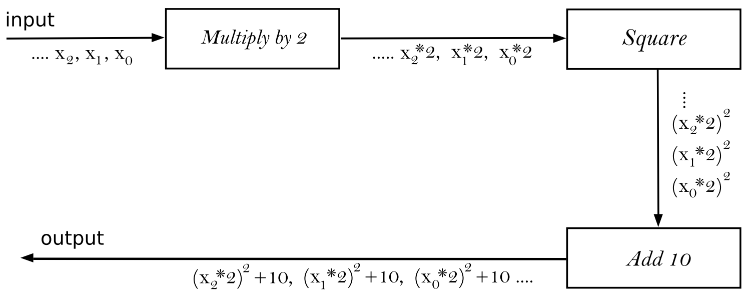

Figure 3. An example process network applying mathematical functions to transform a data stream.

programming [7]. The computational model for Kahn dataflow is based around networks of independent nodes, termed *Kahn process networks*, which simultaneously process data, passing it via directional channels. Nodes are thus functions applied over continuous streams where the output is determined solely by the inputs.

Although a textual representation is often used to describe a process network, a process network may be represented as a directed graph. For example Figure 3 corresponds to a collection of mathematical functions applied to values passed through a Kahn process network.

2.2. Lucid

In 1974 Edward Ashcroft and Bill Wadge began their own exploration of dataflow and over the following ten years conceptualised and introduced Lucid [14]. Initially Ashcroft and Wadge's approach was to "show that conventional, 'mainstream' programming could be done in a purely declarative language, one without assignment or goto statements." – the conventional, mainstream programming being that of procedural programming. Their aim was to ease reasoning and verification of programs by eliminating procedural features such as assignment and goto statements. Procedural features were eliminated by expressing iterative, statement-based procedural programs with declarative expressions. For example a procedural counter in which the value of a variable is updated every iteration of a loop ($i := i + 1$) can be declaratively expressed with the following two definitions:

$$\texttt{first}\, i = 1$$
$$\texttt{next}\, i = i + 1$$

Declarative iteration opened up the possibility that operators such as `first` and `next` could be used to describe the dynamic nature of a program and the contextual values of variables.

Lucid derives from Landin's ISWIM family of languages [11] introducing concepts of *extension* and *intension* from intensional logic to describe the contextual values of declarative ISWIM expressions. At the centre of intensional logic is the concept of *possible world semantics* in which an expression's meaning is inextricably linked to the context of its evaluation. The *extension* of an expression is the value or meaning of an expression in a given context. An expression may have many different contexts and hence many possible extensions. The *intension* of an expression is a mapping from contexts to extensions.

For example, consider the following English natural language expression:

`today it is raining`

The truth of this statement is dependent upon the time and place of its evaluation. Given the intension of the statement and a context, such as 'Monday in London', the extension of the expression can be asserted.

Lucid can be described as an intensional language expressing the intensions of ISWIM expressions. The first instantiation of Lucid was intensional with respect to just one discrete, linearly ordered dimension of contexts, arbitrarily referred to as *time*. Intensional operators in Lucid facilitate context switching relative to the current context to access the extensions of an ISWIM expression. The Lucid operator `next` refers to the extension, or value, of an expression in the context of the next unit of time and `first` refers to the extension of an expression in its first ever instance. Hence the original, single-dimensional Lucid, operating in a discrete dimension of time, becomes a formal, declarative system describing the temporal history of ISWIM expressions.

In Lucid an expression's intension is represented by an infinite stream of *datons* of type: list, string, integer, rational, or symbol. The first daton in a stream is the extension of an expression at the origin of the time dimension. Successive datons in the stream correspond to successive discrete time contexts. Constant expressions remain unchanged over time thus each daton in its representative stream is the same e.g. the constant integer expression 1 is an infinite stream of the integer 1, denotable as $\langle 1, 1, 1, \ldots \rangle$. A daton never exists outside of a stream.

Further to the `first` and `next` operators Lucid provides other operators, also referred to as *filters*, with the ability to manipulate extensions and intensions. An operator called 'followed by', represented by the mnemonic `fby`, is a binary operator expressing the first and following extensions of an expression. Ashcroft and Wadge's declarative iteration in terms of `first` and `next` can be rewritten as:

$$n = 1 \texttt{ fby } (n + 1)\,.$$

In this expression the first extension of n is the integer 1. Successive extensions are recursively defined as $n + 1$. Thus n is the infinite stream of natural numbers $\langle 1, 2, 3, 4, \ldots \rangle$. The `fby` operator is analogous to the concept of induction in which the first parameter of `fby` is the base case and the second parameter is the inductive case.

Filters for arithmetic, parameterised by two streams, apply an operation to extensions from each stream at the same context and are subsequently termed *pointwise operations*.

For example $\langle 1, 1, 1, \ldots \rangle + \langle 1, 2, 3, \ldots \rangle \ = \ \langle 2, 3, 4, \ldots \rangle$.

2.3. Object-orientation

The object-oriented paradigm presents a system of programming in which functions and data are encapsulated into single entities – *objects*. Originating in the 1960s, object-oriented concepts were used in applications such as Sketchpad and languages such as Simula 67. Concepts of object-orientation were explicitly developed further in the 1970s by the language Smalltalk [4].

As a programming paradigm there is no clear definition that encompasses all object-oriented languages, but there are common unifying concepts that bind most languages under this category. These include the concept of objects as entities holding data and actions, inheritance, classes as sets of instantiated objects, methods, message passing, abstraction, polymorphism, and encapsulation [2]. Although these features are not exclusive to object-oriented programming they give an approximate view of the common features of object-oriented languages. The communication of objects via method calling is achieved through "message-passing" in which communication is achieved through the sending and receiving of data.

The use of object-oriented programming proliferated in the 1990s and early 21^{st} century with mainstream languages such as C++, C#, Java, Ruby, PHP, and Python being used extensively in industry and general programming. The language Ruby is used in this paper as an instantiation of an object-oriented language in which to embed Lucid.

3. Embedding dataflow into object-orientation

Lucian embeds dataflow into object-orientation as a form of language interoperation. This embedding is semantically realised by *declarative intensional objects*. An intensional object declares the intension of an object – a mapping of contexts to extensions of the object. An extension of an object is an immutable snapshot of the object and its internal state at a certain time, thus the intensional object as a whole is a stream of such immutable object snapshots representing the historical evaluation of a non-intensional object. This is to be understood in the same manner in which a Lucid stream introduces the notion of the historical evaluation of a non-intensional ISWIM declaration.

For example, if one takes an intensional object and treats it as a stream of object snapshots the *first* value of the stream is the extension of the object just after instantiation. The *next* value of the stream is the extension of the object in the next time step, with possible transformations applied, and so on, as illustrated in Figure 4.

Lucian interoperates with an existing object-oriented language which provides procedural definitions of object classes. The classes are then referenced in the intensional Lucian expressions. Lucian's syntax is modelled after Lucid with some small differences. One such difference in syntax is that Lucian has clearer, ML-style syntax for functions, illustrated in later examples. Additionally Lucian provides

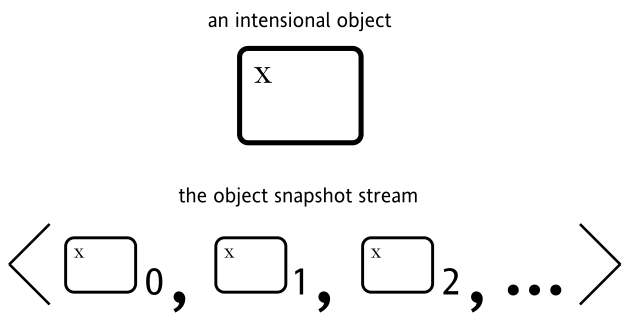

FIGURE 4. An illustration of the notion of an intensional object as an object snapshot stream. The subscript on the object snapshots denotes a time context.

LucianExp	::=	*LucidExp*	Lucid expressions
	|	`new` *var*	instantiation
	|	*var.var*	attribute reference
	|	*var.var* := *var*	attribute update
	|	*var.var*(*Parameters*)	method call
Parameters	::=	*Parameters*, *LucianExp*	method parameters
	|	*LucianExp*	
	|	ϵ	

FIGURE 5. Syntax for handling intensional objects in Lucian as an extension to basic Lucid syntax.

constructs and operators for instantiation, attribute referencing, and method calling on intensional objects. Figure 5 shows a portion of Lucian's grammar defining the syntax for handling intensional objects. The syntax and semantics of the object-oriented constructs in Lucian are now discussed.

Instantiation

Use of the unary `new` operator with the name of an object-oriented class, defined in some local object-oriented code, declares an intensional object that is in an initial state just after instantiation. For instance, if we have a class definition named X in object-oriented code we can instantiate an intensional object of class X with the operation:

$$\texttt{new}\, X$$

This expression evaluates to a new intensional object where, in all contexts, the extension is the object just after instantiation.

Attribute referencing

One can reference an attribute of an object using the common object-oriented "dot" notation for member access:

$$x.\mathit{attribute}$$

As an intensional object embodies the notion of an object's history so an attribute reference represents an object attribute's history. Thus an attribute reference is an intensional *attribute stream.*

Attribute setting can be performed via method calls but Lucian has additional syntax of the form:

$$x.\mathit{attribute} := \mathit{value\_stream}$$

The attribute assignment operator := is a pointwise operation, much like Lucid arithmetic filters, thus the attribute is set from the extensions of *value_stream* in the same contexts as x. The attribute assignment feature is linked with method calling and is visited again in the following treatment of method calls.

Method calls

Method calling also uses the standard "dot" notation for object member access. To call *method* from object x with parameters $\mathit{param}_1,\ \mathit{param}_2,\ \ldots$ the syntax is:

$$x.\mathit{method}(\mathit{param}_1,\ \mathit{param}_2,\ \ldots)$$

Method calls are pointwise operations. When a method is called on an object the method's return value is discarded and is not returned as the extension of the method call expression in the current context. The extension at the context of the method call is in fact the object snapshot representing the new object state after the method call. An intensional object must always consist of a stream of object snapshots thus returning a value from a method and placing it as a daton in the stream would disrupt this continuity. This is a limitation on method calling. If the return result of a method is required then the method must be adapted to store its return result in an attribute to be referenced in an attribute stream. We accept this as a current hindrance but hope to resolve the limitation in the future with a construct for expressing the stream of method return values (see Section 7).

Successive method calls to an intensional object are to be ordered with a special operator, `Then`, which has a similar operation to `fby` but addresses references to an intensional object differently to facilitate object histories. Observe the following:

$$x = \texttt{new}\ \mathrm{X}\ \texttt{Then}\ x.\mathit{method}_1()\ \texttt{Then}\ x.\mathit{method}_2()\ \texttt{Then}\ x \qquad (1)$$

The intensional object x is a stream of object snapshots across time. At $t = 0$ the object has just been instantiated from the class X via the `new` operator. At time $t = 1$ the method named method_1 has been applied to the object and at time $t = 2$ method_2 has been applied. At $t > 2$ the state of the object x remains unchanged. It may be useful for the reader to think of the snapshot stream of x in the following way (where X^1 is an object instantiated from the class X):

$$\left\langle X^1,\ X^1.\mathit{method}_1(),\ X^1.\mathit{method}_1().\mathit{method}_2(),\ X^1.\mathit{method}_1().\mathit{method}_2(),\ \ldots\right\rangle$$

The Lucian-specific operator `Then` is syntactic sugar for a combination of `fby` and `next` operations. Expression (1) can be rewritten as:

$$x = \texttt{new X fby } x.\mathit{method}_1()\ \texttt{fby}\ (\texttt{next}\,x).\mathit{method}_2()\ \texttt{fby}\ (\texttt{next}\,\texttt{next}\,x) \qquad (2)$$

Firstly, x is declared as a newly instantiated object. Then method_1 is applied to this extension. Subsequently method_1 is applied to the `next` extension of x, that is the extension of the object x just after method_1 has been applied. This is followed by another recursive reference, (`next next` x), to the extension of x once method_1 and method_2 have been applied. The operator `Then` abstracts over this use of `fby` and `next` providing a single operation to order object transformations. It is important to note that each extension of x is a distinct object snapshot and not a pointer to a single object.

The operator `Then` can also be used for setting object attributes. For example, the following declares an object x that is instantiated, has $\mathit{attribute}_1$ set to the value of y, then method_1 is called with a parameter, then $\mathit{attribute}_2$ is set:

$$x = (\texttt{new X})\ \texttt{Then}\ (x.\mathit{attribute}_1 := y)\ \texttt{Then}\ x.\mathit{method}_1(z)\ \texttt{Then}\ (x.\mathit{attribute}_2 := w)$$

The example in Figure 6 presents an intensional object `account1` which is defined in terms of its instantiation from the Ruby class `Account` and subsequent method calls transforming the object. The example in Figure 7 develops the example in Figure 6 including more objects and an example of Lucian's improved syntax for functions.

3.1. A formal specification of Lucian

We now discuss a subset of the denotational semantics for Lucian. These formal semantics are not a complete rigorous formal specification of Lucian due to the lack of a full domain theory of objects or formal definitions for the object-oriented language into which the dataflow is embedded. The syntax and semantics for the whole of Lucian, as far as can be defined, are not discussed in this paper but are detailed in a language specification that is currently under preparation. Most of the syntax and semantics are derived from Lucid.

The denotational semantics for Lucian are defined in direct-style denotational semantics with the following semantic function:

$$\mathcal{S} : \mathit{Exp} \rightarrow \big(\mathit{State} \hookrightarrow (\mathit{State} \times \mathit{Stream})\big)$$

State acts as a warehouse of all stream variables in the current scope. An expression is mapped to a function that takes a state and returns a new state – an updated warehouse – and a stream of the intension of the expression.

A subscripted variable such as x_n refers to the n^{th} position in the stream x.

A semantic function S^T is used for the denotational semantics of expressions using the `Then` operator:

$$\mathcal{S}^{\mathcal{T}} : \mathit{Exp} \to \Big(\mathit{Depth} \to \big(\mathit{State} \hookrightarrow (\mathit{State} \times \mathit{Stream})\big)\Big)$$

The additional integer type *Depth* is used to calculate the level of nesting of `Then` expressions so that the correct number of `next` operators can be applied to object variables so that a method call is applied to the latest extension of the intensional object.

Under Lucian's denotational semantics the Lucid followed-by operator `fby` is redefined as:

$$\begin{aligned}
\mathcal{S}|[\mathit{Exp}_1 \; \texttt{fby} \; \mathit{Exp}_2|] \; s &= (s,\ \beta) \\
\text{where}\ \ (s, x) &= \mathcal{S}|[\mathit{Exp}_1|] \; s \\
(s, y) &= \mathcal{S}|[\mathit{Exp}_2|] \; s \\
\beta &= \ \langle x_0,\ y_0,\ y_1,\ y_2,\ \ldots\rangle
\end{aligned}$$

The operator `Then` is defined similarly:

$$\begin{aligned}
\mathcal{S}|[\mathit{Exp}_1 \; \texttt{Then} \; \mathit{Exp}_2|] \; s &= \mathcal{S}^{\mathcal{T}}|[\mathit{Exp}_1 \; \texttt{Then} \; \mathit{Exp}_2|] \; s \; 0 \\
\mathcal{S}^{\mathcal{T}}|[\mathit{Exp}_1 \; \texttt{Then} \; \mathit{Exp}_2|] \; s \; d &= (s,\ \beta) \\
\text{where}\ \ (s, x) &= \mathcal{S}^{\mathcal{T}}|[\mathit{Exp}_1|] \; s \; d \\
(s, y) &= \mathcal{S}^{\mathcal{T}}|[\mathit{Exp}_2|] \; s \; (d+1) \\
\beta &= \ \langle x_0,\ y_0,\ y_1,\ y_2,\ \ldots\rangle
\end{aligned}$$

In terms of the abstract syntax tree for Lucian the semantics of variable references are affected by belonging to a subtree originating from a `Then` operator. The following rules define how `next` operators are applied to an object variable for accessing the latest snapshot at a particular depth.

$$\begin{aligned}
\mathcal{S}^{\mathcal{T}}|[\mathit{var}|] \; s \; 1 \; &= (s,\ s[\mathit{var}]) \\
\mathcal{S}^{\mathcal{T}}|[\mathit{var}|] \; s \; d \; &= \mathcal{S}^{\mathcal{T}}|[\texttt{next} \; \mathit{var}|] \; s \; (d-1)
\end{aligned}$$

All other expressions under the semantic function $\mathcal{S}^{\mathcal{T}}$ are defined in the same way as their semantic definitions under $\mathcal{S}$ with additional passing of the *depth* parameter to sub-expressions.

4. Embedding object-orientation into dataflow

Lucian embeds dataflow into object-orientation via declarative intensional objects as described in the previous section. A converse embedding relationship is also provided by Lucian which embeds object-orientation into dataflow c.f. Figure 2. It is important to note that the relationship of embedding object-orientation

```
class Account
        attr_reader :balance

        def initialize
                @balance = 0
        end

        def deposit(x)
                @balance+=x
        end

        def withdraw(x)
                @balance-=x
        end
end

<%
        account1 = new Account Then
                        account1.deposit(10) Then
                        account1.withdraw(8) Then
                        account1.withdraw(5) Then
                        account1.deposit(2) Then
                        account1
%>
```

FIGURE 6. The `Account` class is instantiated as an intensional object named `account1` in the dataflow code. Various method calls are performed. The attribute stream `account1.balance` has values $\langle 0, 10, 2, -3, -1, -1, \ldots \rangle$.

into dataflow is *not* a mathematical inverse of embedding dataflow into object-orientation. Hence one cannot embed via one embedding relationship and retract via the other.

In early instantiations of Smalltalk designers Alan Kay and Adele Goldberg conceptualised objects as processes that transform input and output message streams linked to procedure calls [5]. This conceptual view from SmallTalk-72 is remarkably similar to the concept of filters in dataflow alluding to an object-filter "duality" where objects can be described as filters and filters described as objects. Lucian explicitly realises this duality by embedding object-orientation into dataflow via *object stream filters* in which objects defined in object-oriented code are used as pointwise filters in dataflow. Thus the return stream and parameter streams of the object filter embody Kay's concept of input and output message streams from an object.

The advantages of representing filters with objects are two-fold: *coarse-grained parallelism* can be achieved, and state can be handled.

```
class Account ...

class Bank ....

<%
        fun withdraw account amount =
                if ((account.balance)>=amount)
                        then        account.withdraw(amount)
                        else        account.withdraw(0)
                end

        fun deposit account amount =
                account.deposit(amount)

        account1 = new Account Then
                       (deposit(account, 10)) Then
                       (withdraw(account, 8)) Then
                       (withdraw(account, 5)) Then
                       (deposit(account, 2)) Then account1

        account2 = new Account

        bank = new Bank Then
                bank.addAccount(account1) Then
                bank.addAccount(account2) Then bank
%>
```

FIGURE 7. Building on the previous example, this example uses user-defined dataflow functions. The attribute stream `account1.balance` now has values $\langle 0, 10, 2, 0, 2, 2, \ldots\rangle$. A further account is defined and a `Bank` intensional object holds all accounts.

Declarative dataflow is free from side-effects thus expressions can be evaluated in any order, or in parallel, hence dataflow is *implicitly parallel.* Furthermore the notion of streams and filters can be modelled simply as a network of nodes/processes joined by channels as in Kahn process networks. The nodes of the process network are atomic and thus define the granularity of potential parallelism. In Lucid and Lucian the built-in filters are the atomic units and are thus the minimum level of granularity. If a Lucid program is transcribed into a process network every filter is transformed into a process therefore the granularity is very fine with many nodes. Being able to specify a filter in procedural object-oriented code reduces the granularity as fine-grained computations can be expressed procedurally, encapsulated, and hidden within an object which is seen as an atomic filter. Therefore the resolution of the granularity is reduced hence *coarse-grained parallelism* is provided.

Filters defined by objects in the object-oriented language can manage local internal state and can be used to deal with any global state-transforming operations, such as input and output. State, whether it be global or local, is handled only by the object-oriented code thus the dataflow code remains state-free. The ability of the object-oriented code to fully manipulate state provides the potential for the implicit parallelism of dataflow to be invalidated by side-effects in object filters. We acknowledge this and can do little to prevent it but argue that Lucian helps to control the careful usage of state. The interoperation of object-orientation with dataflow reduces the need for state in the object-oriented code. Additionally via *object stream filters* Lucian provides a way for state to be handled procedurally which is hidden from the dataflow code, delegating the task to the object-oriented paradigm as opposed to attempting to add the functionality for state manipulation to the dataflow.

The action of nesting object filters into the dataflow component of Lucian is syntactically expressed using a caret '`^`' operator with a class name from the embedded object-oriented code and a set of parameters. Figure 8 shows an example usage that uses an object as a filter for pretty-printing the Fibonacci sequence of numbers. This example demonstrates internal state management within the `Pretty_Print_Fib` filter object via a counter that is incremented for each input daton received. The example also demonstrates global state effects by printing to the standard output in the object-oriented code. All state transformations are left to the procedural code, with the dataflow code remaining state-free.

Some preparation of the class describing the object filter is required in the current implementation, including subclassing it from a `Node` superclass which provides interoperation via buffers to the inputs and outputs of the filter.

5. Applications

Aside from using declarative intensional objects to help formalise object-oriented programming some interesting applications of intensional objects arise. One such use is to manipulate object snapshot streams to facilitate *versioned objects* where changes to objects can be "rolled-back" or undone. Versioned objects have particular use in user interfaces where undo and redo functionalities are required, or even branching of versioned objects.

An interesting feature of dataflow languages such as Lucian and Lucid is their ability to express dynamical systems, specifically in *initial value problem* form. Analysis of dynamical systems aims to understand how a variable or system of variables changes over time, thus analysis captures the intension of the variables in the context of time.

An *initial value problem* is a system of ordinary differential equations defined in terms of an initial value and the first order derivative for each variable in the system of equations [10]. For example, variable y has a first order derivative in the form:

$$\dot{y}(t) = F\big(y(t)\big)\,. \tag{3}$$

```
<%
        class Pretty_Print_Fib < Node

                def initialize(x, y)
                        super(x, y)
                        @counter = 0
                end

                # Formatting expressed in Ruby

                def pretty_print(a)
                        # Local counter state
                        @counter+=1
                        print "Fibonacci number "+@counter.to_s+" = "+a.to_s+"\n"
                end

                # Implementing the 'run' method from the Node class
                # Using input_buffers and output_buffer attributes
                # added by the Node class

                def run
                        a = @input_buffers[0].read
                        pretty_print(a)
                        @output_buffer.write(a)
                end
        end
%>

// Lucian code

// Use the caret operator to embed the Pretty_Print_Fib class
// from Ruby and use as a dataflow filter to print the fib sequence

^Pretty_Print_Fib(fib) where
                            fib = (1 fby (1 fby (fib + (next fib))))
                        end
```

An example execution of this program:

```
$ lucian pretty_fib.lcr; ruby pretty_fib.rb;
Fibonacci number 1 = 1
Fibonacci number 2 = 1
Fibonacci number 3 = 2
Fibonacci number 4 = 3
Fibonacci number 5 = 5
....
```

FIGURE 8. An example of embedding an object from Ruby as a filter into Lucian's dataflow code to handle local and global state, pretty printing the Fibonacci sequence. The Ruby code is enclosed by <% and %>.

And an initial condition on the function y is specified:

$$y(t_0) = y_0 \quad \text{where} \quad y_0 \in \mathit{dom}(F)\,. \tag{4}$$

The definition of $y(t)$ is typically unknown and F is usually independent of time. Such a system is naturally expressible in Lucid or Lucian in the form:

$$y = y_0 \texttt{ fby } y + F(y)\,.$$

Discrete small-step approximations may be used, as in the Euler technique, to calculate a numerical approximation of the solution, where h is the small step value:

$$y = y_0 \texttt{ fby } y + h * F(y)\,.$$

This simple definition of y captures the intension of the variable, evaluating to the infinite time series of the approximate solution of y. Whole systems of equations can be defined simply in this way, with the ability to apply more accurate approximation techniques easily, such as Runge–Kutta:

$$\begin{aligned}
&y = y_0 \texttt{ fby } y + \mathit{ydot}\\
&\quad\quad \text{where}\\
&\quad\quad\quad k_1 = F(y)\\
&\quad\quad\quad k_2 = F(y + k_1 * h/2)\\
&\quad\quad\quad k_3 = F(y + k_2 * h/2)\\
&\quad\quad\quad k_4 = F(y + k_3 * h)\\
&\quad\quad\quad \mathit{ydot} = y + (h/6) * (k_1 + 2 * k_2 + 2 * k_3 + k_4)
\end{aligned}$$

One particular usage of dynamical systems is in biology in modelling biochemical reactions within cells.

The ever expanding field of systems biology seeks to understand biological systems at all levels of abstraction, aiming to understand the relationships and interactions between all aspects of a system in order to understand emergent behaviours at an organismal level [8]. In the case of understanding cells dynamical systems are used to model the biochemical reactions that take place within cells. While dynamical systems models are informative their relationship with other aspects of a cell is still relatively unmodelled in systems biology. To link biochemical reactions to higher behaviours requires models of other processes and structures within the cell, such as membranes, receptors, or flagella. Hybrid modelling techniques are appearing in which abstract models of cell components are integrated with dynamical models of the biochemical reactions.

Lucian's natural ability to laconically express systems of dynamical equations is furthered by its ability to interoperate these time-series-like computations with intensional objects which can model more abstract aspects and behaviours. Therefore Lucian provides a basis with which to build hybrid models of complex systems that unifies differential equations with other abstract models expressible as intensional objects. Figure 9 presents a fragment of a hypothetical model of a

```
// Reaction Rates
r1 = 3.2*x*a
r2 = x*y
r3 = x
r4 = z
r5 = y

// First order derivatives
xdot = (r1 - r2) - r3
ydot = r4 - r5
zdot = r3 - r4

// Dynamics
x = 2.5 fby (x+h*xdot)
y = 2.5 fby (y+h*ydot)
z = 2.5 fby (z+h*zdot)
// Receptor concentration integrated into the dynamics
a = 1 fby (a + h*(receptor.concentration - (next receptor.concentration)))

// Abstract cell components
flagella = (new Flagella) fby flagella.tumble(x)
receptor = (new Receptor) fby receptor.sense()
```

Figure 9. A hypothetical hybrid model of a cell, modelled with a dynamical system of a equations and abstract models of cell components represented in objects (object-oriented code not shown).

cell written in Lucian that combines dynamical equations with abstract models. The model uses a Hopf bifurcation system modulated by a receptor, sensing the environment of a cell, which controls the "tumbling" of a flagellum to propel the cell. The intension of the variables x, y, and z in the differential equations, the intension of the cell components, and the relationships between the two types of model are captured succinctly within Lucian.

6. Implementation of Lucian

A proof-of-concept implementation of Lucian is written in Haskell. Lucian programs are compiled by translating Lucian code into a target object-oriented language, determined by the language of the object classes used for intensional objects or filters, which in this case is Ruby. Interoperation with the popular Java and C# languages is planned. Input to the Lucian translator is in the form of source code that has dataflow and object-oriented code suitably delimitered with `<%` and `%>` markers as can be seen in Figure 8.

The translated program forms a concurrent Kahn process network in which independent parallel processes communicate via unbounded buffers within the

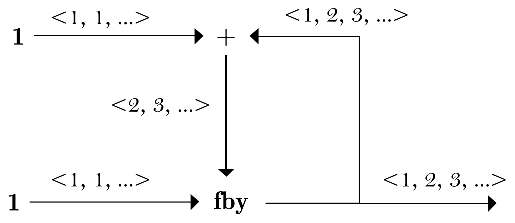

Figure 10. A Kahn process network for the natural numbers illustrating the concurrent implementations of Lucian programs.

object-oriented language; filters are translated into processes, and streams are translated into unbounded buffers.

For example a simple expression declaring the stream of natural numbers: `n = 1 fby n+1` is translated into a Kahn-style process network by the Lucian translator as in Figure 10. Processes are defined for the nodes `1`, `+`, and `fby`, which pass values between each other with concurrent buffers. The intermediate streams are shown next to the edges in the process graph in Figure 10 representing the flow of datons in the concurrent buffers joining processes.

The Lucian translation process allows an easily implementable and controllable integration process between the two languages (Lucian and Ruby) in which the Lucian dataflow code is translated into the object-oriented language. If translation was made into a third language a higher formal understanding of the object-oriented language would be required in order to translate both the object-oriented and dataflow code into the third language. A complete treatment of the formal semantics of an object-oriented language is rare as many object-oriented languages are partially or totally undefined – true for Java and Ruby.

The scheme of translating Lucian into the object-oriented language works well although as dataflow code increases in complexity the speed of execution rapidly decreases due to large overheads in Ruby's concurrency constructs. It is hoped that these implementational issues will be resolved soon, although the current implementation serves the purpose of being an initial proof-of-concept translator.

6.1. Related work

The language GLU (Granular Lucid) explores the concept of integrating dataflow with the procedural paradigm. GLU (pronounced *glue*) combines the intensional dataflow programming of Lucid with the procedural language C, allowing functions written in C to be used as intensional filters within Lucid [6]. GLU's C filters allow coarse-grained parallelism in Lucid programs. GLU thus provides a similar functionality to the embedding of object-orientation into dataflow in Lucian via object stream filters. We feel Lucian's object-based filters provide a more natural model of a filter than a C procedure as an object filter encapsulates internal state and implements method calls via message passing.

The OLucid language extends Lucid to allow classes defined in the object-oriented language Java to be instantiated as *datons* in Lucid streams [15]. OLucid has the benefit of allowing Lucid to operate with complex data structures that are useful for real-world applications, rather than just the standard set of Lucid datons. While there are some similarities between Lucian and OLucid the semantics of objects in OLucid differ from Lucian's semantics. OLucid allows streams where each daton is an individual object whereas Lucian presents a single intensional object whose extensions are immutable snapshots. While OLucid is practical and has a strong parallel eductive implementation the intensionality of the dataflow language is compromised by the object constructs which do not preserve the intensionality of Lucid. We feel Lucian presents a more coherent, intensionality preserving, semantic model for interoperating objects with Lucid via declarative intensional objects and object stream filters.

The notion of intensional objects presented in this paper differs significantly from the intensional objects of Kropf and Plaice which are used in a intensional warehouse system for intensional versioning [9]. The use of objects in a distributed environment is discussed by Kropf and Plaice in which objects can have their implementations adapted based on their context. This differs from Lucian's intensional objects which are a direct intensional version of non-intensional objects.

Other notable uses of intensional programming have been in the domain of real-time systems such as the language Lustre [3] and its derivatives which have become well established in critical real-time systems programming in avionics and control systems.

7. Conclusions and further work

This paper has presented the Lucian dataflow language, derived from Lucid, which interoperates dataflow and object-orientation via two different embedding relationships. Lucian has successfully demonstrated that the paradigms can be succinctly combined providing a useful tool for the programmer. The interoperation of dataflow and object-orientation in Lucian is not a simple foreign-function interface, but is such that it resolves the semantics of objects, streams, and intensionality relative to each other.

The embedding of dataflow into object-orientation is realised via *declarative intensional objects* in which streams of object snapshots define the intension of an object. This presents a formal declarative method for defining object-oriented programs in terms of histories of computations on objects.

The converse embedding relationship of embedding object-orientation into dataflow is realised through *object stream filters* in which objects defined in an object-oriented language are used as filters in dataflow. This facilitates handling local state and input/output from a dataflow program promoting practical applications. Functionality defined procedurally in filters also reduces the granularity of implicit parallelism by expressing finely granular operations procedurally in an atomic, encapsulated object stream filter.

While the Lucian translator is still in development it is usable with many operable sample programs. This includes some larger programs as alluded to in Section 5 that show the benefits of mixing declarative dataflow programming with object-orientation. Specifically we have given details of using Lucian to define hybrid models for the purpose of systems biology modelling and simulation in which models of dynamical systems are unified with other system components abstractly modelled with objects.

We accept that the current inability of intensional object method calls to return a value directly into a stream is a loss in expressivity. The return value of a function must be explicitly stored in an object attribute to be retrieved. A possible future solution may be to offer syntax to reference the return value of a method. For example suffixing the method call with an exclamation mark:

$$x\_method\_return = x.method(param_1,\ param_2,\ \ldots)!$$

This expression would not actually call the method on x but would evaluate to the stream of datons from any calls to *method*. This problem requires further experimentation and exploration to provide the most expressive and natural solution in line with the semantics of intensional objects.

While operational and denotational semantics for Lucian have been formally defined the definitions are incomplete as the object domains are unknown and the semantics of the underlying object-oriented languages are undefined. We would like to see further work in strengthening the formal specification of Lucian, perhaps using Abadi and Cardelli's imperative object calculus [1] combined with stream calculus [13].

Lucian has allowed for an exploration of the issues involved in creating a multi-paradigm language that interoperates languages of different paradigms via more than just a simple data-marshalling foreign function interface. Lucian has given a sufficient example of how dataflow and object-orientation can be conjoined, and raises questions in the area of the semantics of combined paradigm languages. It is our hope that further work will develop the formalism of multi-paradigm languages and that concepts of *intensionality* and dataflow will find their way into mainstream programming to improve the expressivity of techniques, languages, and tools available to the problem solver.

Acknowledgements

We would like to thank Bill Wadge for his input and Andrew Davey, Daniel James, and Sara Kalvala for their reading and advice in refining the ideas of this paper. Additional thanks to the anonymous reviewers for their comments and advice.

References

[1] M. Abadi and L. Cardelli. An Imperative Object Calculus. In *TAPSOFT '95: Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 471–485, UK, 1995. Springer-Verlag.

[2] D. J. Armstrong. The quarks of object-oriented development. *Commun. ACM*, 49(2):123–128, 2006.

[3] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A declarative language for real-time programming. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 178–188, New York, NY, USA, 1987. ACM Press.

[4] A. Goldberg and D. Robson. *Smalltalk-80: The language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[5] A. Goldberg and A. Kay. *Smalltalk-72 Instruction Manual*. Learning Research Group, Xeroz Palo Alto Research Center, 1976.

[6] R. Jagannathan, C. Dodd, and I. Agi. GLU: A high-level system for granular data-parallel programming. *Concurrency: Practice and Experience*, 9(1):63–83, 1997.

[7] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

[8] H. Kitano. Systems Biology: A Brief Overview. *Science*, 295(5560):1662–1664, 2002.

[9] P. Kropf and J. Plaice. Intensional Objects. In *12th International Symposium on Languages for Intensional Programming* 1999, pages 180–187, National Centre for Scientific Research, Athens, Greece, June 1999.

[10] J. D. Lambert. *Numerical methods for ordinary differential systems: The initial value problem*. Wiley, 1991.

[11] P. J. Landin. The Next 700 Programming Languages. *Communications of the ACM*, 9(3):157–166, March 1966.

[12] S. Peyton Jones, T. Nordin, and A. Reid. Green Card: A foreign-language interface for Haskell. *Proceedings of the Haskell Workshop*, June 1997.

[13] J. J. M. M. Rutten. A tutorial on coinductive stream calculus and signal flow graphs. *Theor. Comput. Sci.*, 343(3):443–481, 2005.

[14] W. W. Wadge and E. A. Ashcroft. *LUCID, the dataflow programming language*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[15] Q. Zhao. Implementation of an object-oriented intensional programming system. Master's thesis, University of New Brunswick, Fredericton, N.B., Canada, 1998.

Dominic A. Orchard and Steve Matthews
Department of Computer Science
University of Warwick
Coventry, CV47AL
United Kingdom
e-mail: `dom.orchard@gmail.com`
`sgm@dcs.warwick.ac.uk`

Received: November 30, 2007.

Accepted: April 30, 2008.