# Formally Specifying and Proving Operational Aspects of Forensic Lucid in Isabelle

Serguei A. Mokhov and Joey Paquet

Department of Computer Science and Software Engineering
Faculty of Engineering and Computer Science
Concordia University, Montréal, Québec, Canada,
{mokhov,paquet}@cse.concordia.ca

**Abstract.** A Forensic Lucid intensional programming language has been proposed for intensional cyberforensic analysis. In large part, the language is based on various predecessor and codecessor Lucid dialects bound by the higher-order intensional logic (HOIL) that is behind them. This work formally specifies the operational aspects of the Forensic Lucid language and compiles a theory of its constructs using Isabelle, a proof assistant system.

## 1 Introduction

As a part of the Intensional Cyberforensics project, we define a functional-intensional programming/specification language, called Forensic Lucid. The language is under active design and development including its syntax, semantics, the corresponding compiler, run-time, and interactive "development" environments [1,2] that we refer to as General Intensional Programming System (GIPSY) [3]. We approach the problem using Isabelle [4] as a proof assistant.

*Problem Statement.* A lot of intensional dialects have been spawned from the functional intensional programming language called Lucid [5,6,7,8,9,10,11,12]. Lucid (see Section 1.2) itself was invented with a goal for program correctness verification [7,8]. While there were a number of operational semantics rules for compiler and run-time environments developed for all those dialects throughout the years, there was no a complete formal proof set of the rules of the languages. Yet another dialect of Lucid has been created to foster the research on intensional cyberforensics (see Section 1.3), called Forensic Lucid, which, in a large part is a union of the syntax and operational semantics rules from the comprising languages with the forensic extensions. In order to be a credible tool to use, for example, in court, to implement relevant tools for the argumentation, the language ought to have a solid scientific base, a part of which is formalizing the semantics the language and proving correctness of the programs written in it.

*Proposed Solution.* In this work, we propose to begin validation of the Forensic Lucid constructs with the Isabelle prover assistant [4] and extend it to the comprising Lucid dialects as a whole. We proceed bottom-up from "core" Lucid dialects such as GIPL, Lucx, and Indexical Lucid and even their smaller decompositions as well as top-down from Forensic Lucid to arrive to a comprehensive set of proofs covering the dialects.

### 1.1 Intensional Logics and Programming

**Definitions.** Intensional programming (IP) is based on intensional (or multidimensional) logics, which, in turn, are based on natural language understanding aspects (such as time, belief, situation, and direction). IP brings in **dimensions** and **context** to programs (e.g. space and time in physics or chemistry). Intensional logic adds dimensions to logical expressions; thus, a non-intensional logic can be seen as a constant or a snapshot in all possible dimensions. *Intensions are dimensions* at which a certain statement is true or false (or has some other than a Boolean value). *Intensional operators* are operators that allow us to navigate within these dimensions. *Higher-order intensional logic* (HOIL) is the one that couples functional programming as that of Lucid with multidimensional dataflows that the intensional programs can query an alter through an explicitly notion of contexts as first-class values [13,14].

**An Example of Using Temporal Intensional Logic.** Temporal intensional logic is an extension of temporal logic that allows to specify the time in the future or in the past.

(1) $E_1$ := it is raining **here today**

Context: {`place:`**here**, `time:`**today**}

(2) $E_2$ := it was raining **here** *before*(**today**) = *yesterday*

(3) $E_3$ := it is going to rain *at* (altitude **here** + 500 m) *after*(**today**) = *tomorrow*

Let's take $E_1$ from (1) above. Then let us fix **here** to **Montreal** and assume it is a *constant*. In the month of February, 2008, with granularity of day, for every day, we can evaluate $E_1$ to either *true* or *false*:

```
Tags:   1 2 3 4 5 6 7 8 9 ...
Values: F F T T T F F F T ...
```

If one starts varying the **here** dimension (which could even be broken down to $X, Y, Z$), one gets a two-dimensional evaluation of $E_1$:

```
City: /   1 2 3 4 5 6 7 8 9 ...
Montreal  F F T T T F F F T ...
Quebec    F F F F T T T F F ...
Ottawa    F T T T T T F F F ...
```

## 1.2 Lucid

Lucid [5,6,9,7,8] is a dataflow intensional and functional programming language. In fact, it is a family of languages that are built upon intensional logic (which in turn can be understood as a multidimensional generalization of temporal logic) involving context and demand-driven parallel computation model. A program written in some Lucid dialect is an expression that may have subexpressions that need to be evaluated at certain *context*. Given the set of dimension $D = \{dim_i\}$ in which an expression varies, and a corresponding set of indexes or *tags* defined as placeholders over each dimension, the context is represented as a set of $<dim_i : tag_i>$ mappings and each variable in Lucid, called often a *stream*, is evaluated in that defined context that may also evolve using context operators [14,15,16,13]. The generic version of Lucid, GIPL [11], defines two basic operators `@` and `#` to navigate in the contexts (switch and query). The GIPL was the first generic programming language of all intensional languages, defined by the means of only two intensional operators `@` and `#`. It has been proven that other intensional programming languages of the Lucid family can be translated into the GIPL [11]. Please refer to Appendix A for the greater details about Lucid origins, variables as streams, random access to streams, and the basic operators. Since the Lucid family of language thrived around intensional logic that makes the notion of context explicit and central, and recently, a first class value [16,13,14,15] that can be passed around as function parameters or as return values and have a set of operators defined upon. We greatly draw on this notion by formalizing our evidence and the stories as a contextual specification of the incident to be tested for consistency against the incident model specification. In our specification model we require more than just atomic context values – we need a higher-order context hierarchy to specify different level of detail of the incident and being able to navigate into the "depth" of such a context. A similar provision by has already been made by the author [17] and earlier works of Swoboda et al. in [18,19,20,21] that needs some modifications to the expressions of the cyberforensic context.

Some other languages can be referred to as intensional even though they may not refer to themselves as such, and were born after Lucid (Lucid began in 1974). Examples include hardware-description languages (HDLs, appeared in 1977) where the notion of time (often the only "dimension", and usually progresses only forward), e.g. Verilog and VHDL. Another branch of newer languages for the becoming popular is aspect-oriented programming (AOP) languages, that can have a notion of context explicitly, but primarily focused on software engineering aspect of software evolution and maintainability.

## 1.3 Cyberforensic Analysis

Cyberforensic analysis has to do with automated or semi-automated processing of and reasoning about electronic evidence, witnesses, and other details from cybercrime incidents (involving computers, but not limited to them). Analysis is one of the phases in cybercrime investigation, where the others focus on evidence collection, preservation, chain of custody, information extraction that precede the analysis. The phases the follow the analysis are formulation of a report and potential prosecution, typically involving expert witnesses. There are quite a few techniques, tools (hardware and software), and methodologies have been developed for all the briefly mentioned phases of the cybercrime investigation. A lot of attention has been paid to the tool development for evidence collection and preservation; a few tools have been developed to aid "browsing" data in the confiscated storage media, log files, memory, and so on. A lot less number of tools have been developed for case analysis of the data, and the existing commercial packages (e.g. Encase or FTK) are very expensive. Even less so there are case management, event modeling, and event reconstruction, especially with solid formal theoretical base. The first formal approach to the cybercrime investigation was the finite-state automata (FSA) approach by Gladyshev et. al [22,23]. The approach is complex to use and understand for non computer science or equivalent investigators. The aim of Forensic Lucid is to alleviate those difficulties, be sound and complete, expressive and usable, and provide even further usability improvement with the graphic interface that allow data-flow graph-based (DFG) programming that allows translation between DFGs and Lucid code for compilation and is implemented for Indexical Lucid in GIPSY already [24], and requires forensic extensions. While Forensic Lucid is in the design and implementation, its solid base is being established in part with this work. The goal of Forensic Lucid in the cyberforensic analysis is to be able to express in a program form the encoding of the evidence, witness stories, and evidential statements, that can be tested against claims to see if there is a possible sequence or multiple sequences of events that explain a given story. This is designed to aid investigator to avoid ad-hoc conclusions and have them look at the possible explanations the Forensic Lucid program execution would yield and refine the investigation, as was shown in the works [22,23] investigators failed to analyze all the stories and their plausibility before drawing conclusions in the case. We do not recite the cases here due to the length limitations.

## 2 Forensic Lucid

The end goal is to define our Forensic Lucid language where its constructs concisely express cyberforensic evidence, which can be initial state of a case towards what we have actually observed as a final state. The implementing system (i.e. GIPSY) has to backtrace intermediate results in order to provide the corresponding event reconstruction path, if it exists. The result of the expression in its basic form is either *true* or *false*, i.e. "guilty" or "not guilty" given the context per explanation with the backtrace. There can be multiple backtraces, that correspond to the explanation of the evidence (or lack thereof).

### 2.1 Properties

We define Forensic Lucid to model the evidential statements and other expressions representing the evidence and observations as a higher-order context hierarchy. An execution trace of a Forensic Lucid program would expose the possibility of the proposed claim with the events in the middle.

Addition of the context calculus from Lucx for operators on Lucx's context sets (`union`, `intersection`, etc.) are used to address to provide a collection of traces. Forensic Lucid inherits the properties of Lucx, MARFL, Objective Lucid, JOOIP (and their comprising dialects), where the former is for the context calculus, and the latter for the arrays and structural representation of data for modeling the case data structures such as events, observations, and groupings of the related data.

One of the basic requirements is that the complete definition of the operational semantics of Forensic Lucid should be compatible with the basic Lucx and GIPL, i.e. the translation rules or equivalent are to be provided when implementing the language compiler within GIPSY, and such that the GEE can execute it with minimal changes.

```
foo @
{
  [ final observed event , possible initial observed event ],
  [            ],
  [            ]
}
```

**Listing 1.1.** Intensional Storyboard Expression

While the `[...]` notation here may be confusing with respect to the notation of `[dimension:tag]` in Lucid and more specifically in Lucx [13,25], it is in fact a simple syntactical extension to allow higher-level groups of contexts where this syntactical sugar is later translated to the baseline context constructs. The tentative notation of `{[...],...,[...]}` implies a notion similar to the notion of the "context set" in [13,25] except with the syntactical sugar mentioned earlier where we allow syntactical grouping of properties, observations, observation sequences, and evidential statements as our context sets.

## 2.2 Transition Function

A transition function determines how the context of evaluation changes during computation. A general issue exists that we have to address is that the transition function $\psi$ is problem-specific. In the FSA approach, the transition function is the labeled graph itself. In the first prototype, we follow the graph to model our Forensic Lucid equivalent. In general, Lucid has already basic operators to navigate and switch from one context to another, which represent the basic transition functions in themselves (the intensional operators such as `@`, `#`, `isEOD`, `first`, `next`, `fby`, `wvr`, `upon`, and `asa` as well as their inverse operators[1]). However, a specific problem being modeled requires more specific transition function than just plain intensional operators. In this case the transition function is a Forensic Lucid function where the matching state transition modeled through a sequence of intensional operators. In fact, the forensic operators are just pre-defined functions that rely on traditional and inverse Lucid operators as well as context switching operators that achieve something similar to the transitions in [22,23]. In fact, the intensional operators of Lucid represent the basic building blocks for $\psi$ and $\Psi^{-1}$.

## 2.3 Primitive Operators

The basic set of the classic intensional operators is extended with the similar operators, but inverted in one of their aspects: either negation of trueness or reverse of direction of navigation. Here we provide an informal definition followed by their formal counterpart of these operators alongside with the classical ones (to remind the reader what they do and enlighten the unaware reader). The reverse operators have a restriction that they must work on the bounded streams at the positive infinity. This is not a stringent limitation as the our contexts of observations and evidence in this work are always finite, so they all have the beginning and the end. What we need is an ability to go back in the stream and, perhaps, negate in it with classical-like operators, but reversed.

The operators are defined below to give a complete picture. The classical operators FIRST, NEXT, FBY, WVR, UPON, and ASA were previously defined in [11] and earlier. The other complimentary, inverse, and negation operators were defined and revised from [26]. In this list of operators, especially the reverse ones, we make an important assumption that the streams we are working with are finite, which is sufficient for our tasks. Thus, our streams of context values can be bound between bod and eod and contain a finite tag set of elements is used as a context type. For summary of the application of the just defined operators' examples, please refer to Appendix B.

Following the steps in [11], we further represent the definition of the operators via `@` and `#`. Again, there is a mix of classical operators that were previously defined in [11], such as `first`, `next`, `fby`, `wvr`, `upon`, and `asa` as well as the new operators from this work. The collection of the translated operators denoted in `monospaced font`, while we provide their equivalence to the original Lucid operators, denoted as SMALL CAPS.

The primitive operators are founding blocks to construct more complex case-specific functions that represent a particular investigation case as well as more complex so-called *forensic operators*.

- A stream of first elements of stream $X$:
  $\text{FIRST}\, X = (x_0, x_0, ..., x_0, ...)$

$$\texttt{first}\ X = X@0 \tag{1}$$

- A stream of second elements of stream $X$:
  $\text{SECOND}\, X = (x_1, x_1, ..., x_1, ...) = \text{FIRST NEXT}\, X$

[1] Defined further.

- A stream of last elements of stream $X$:
  LAST $X = (x_n, x_n, ..., x_n, ...)$
  This definition of the LAST operator relies on the earlier stated assumption that our streams can be explicitly finite for the language we are developing. This affects the follow up operators that rely in that fact just as well. It is also important to note that the LAST operator in our design does not return `eod` all the time on the finite stream due to lack of usefulness for such a value; instead it returns the element of the stream just before the `eod`.

$$\texttt{last}\ X = X@(\#@(\#\texttt{iseod}(\#) - 1)) \tag{2}$$

- A stream of elements one before the last one of stream $X$:
  PRELAST $X = (x_{n-1}, x_{n-1}, ..., x_{n-1}, ...) =$ LAST PREV $X$
- A stream of elements of stream $X$ other than the first:
  NEXT $X = (x_1, x_2, ..., x_{i+1}, ...)$

$$\texttt{next}\ X = X@(\# + 1) \tag{3}$$

- A stream of elements of stream $X$ other than the last:
  PREV $X = (x_{n-1}, ..., x_{i+1}, x_i, x_{i-1}, ...)$

$$\texttt{prev}\ X = X@(\# - 1) \tag{4}$$

- First element of $X$ followed by all of $Y$:
  $X$ FBY $Y = (x_0, y_0, y_1, ..., y_{i-1}, ...)$

$$\begin{aligned} X\ \texttt{fby}\ Y &= \textbf{if}\ \# = 0\ \textbf{then}\ X\ \textbf{else}\ Y@(\# - 1) \\ &= \textbf{if}\ \texttt{isbod}\ X\ \textbf{then}\ X\ \textbf{else}\ \texttt{prev}\ Y \end{aligned} \tag{5}$$

- First element of $X$ preceded by all of $Y$:
  $X$ PBY $Y = (y_0, y_1, ..., y_{i-1}, ..., y_n, x_0)$

$$\begin{aligned} X\ \texttt{pby}\ Y &= \textbf{if}\ \texttt{iseod}\ \#\ \textbf{then}\ X\ \textbf{else}\ Y@(\# + 1) \\ &= \textbf{if}\ \texttt{iseod}\ Y\ \textbf{then}\ X\ \textbf{else}\ \texttt{next}\ Y \end{aligned} \tag{6}$$

- Stream of negated arithmetic values of $X$:
  NEG $X = (-x_0, -x_1, -x_2, ..., -x_{i+1}, ...)$

$$\texttt{neg}\ X = -X \tag{7}$$

- Stream of inverted truth values of $X$:
  NOT $X = (!x_0, !x_1, !x_2, ..., !x_{i+1}, ...)$

$$\texttt{not}\ X = \textbf{if}\ X\ \textbf{then}\ !X\ \textbf{else}\ X \tag{8}$$

- A logical AND stream of truth values of $X$ and $Y$:
  $X$ AND $Y = (x_0 \&\& y_0, x_1 \&\& y_1, x_2 \&\& y_2, ..., x_{i+1} \&\& y_{i+1}, ...)$

$$X\ \texttt{and}\ Y = X \&\& Y \tag{9}$$

- A logical OR stream of truth values of $X$ and $Y$:
  $X$ OR $Y = (x_0 || y_0, x_1 || y_1, x_2 || y_2, ..., x_{i+1} || y_{i+1}, ...)$

$$X\ \texttt{or}\ Y = X || Y \tag{10}$$

– A logical XOR stream of truth values of $X$ and $Y$:
$X$ XOR $Y = (x_0 \oplus y_0, x_1 \oplus y_1, x_2 \oplus y_2, ..., x_{i+1} \oplus y_{i+1}, ...)$

$$X \;\texttt{xor}\; Y = \texttt{not}((X \;\texttt{and}\; Y) \;\texttt{or}\;\texttt{not}\;(X \;\texttt{or}\; Y)) \tag{11}$$

– WVR stands for *whenever*. WVR chooses from its left-hand-side operand only values in the current dimension where the right-hand-side evaluates to *true*.

```
$X$ WVR $Y =$
    **if** FIRST $Y \neq 0$
    **then** $X$ FBY (NEXT $X$ WVR NEXT $Y$)
    **else** (NEXT $X$ WVR NEXT $Y$)
```

$$\begin{aligned} X \;\texttt{wvr}\; Y = X@T \;&\texttt{where} \\ &T = U \;\texttt{fby}\; U@(T+1) \\ &U = \textbf{if}\; Y \;\textbf{then}\; \# \;\textbf{else}\; \texttt{next}\; U \\ \texttt{end}& \end{aligned} \tag{12}$$

– RWVR stands for *retreat whenever*. RWVR chooses from its left-hand-side operand backwards only values in the current dimension where the right-hand-side evaluates to *true*.

```
$X$ RWVR $Y =$
    **if** LAST $Y \neq 0$
    **then** $X$ PBY (PREV $X$ RWVR PREV $Y$)
    **else** (PREV $X$ RWVR PREV $Y$)
```

$$\begin{aligned} X \;\texttt{rwvr}\; Y = X@T \;&\texttt{where} \\ &T = U \;\texttt{pby}\; U@(T-1) \\ &U = \textbf{if}\; Y \;\textbf{then}\; \# \;\textbf{else}\; \texttt{prev}\; U \\ \texttt{end}& \end{aligned} \tag{13}$$

– NWVR stands for *not whenever*. NWVR chooses from its left-hand-side operand only values in the current dimension where the right-hand-side evaluates to *false*.

```
$X$ NWVR $Y = X$ WVR NOT $Y =$
    **if** FIRST $Y == 0$
    **then** $X$ FBY (NEXT $X$ NWVR NEXT $Y$)
    **else** (NEXT $X$ NWVR NEXT $Y$)
```

$$\begin{aligned} X \;\texttt{nwvr}\; Y = X@T \;&\texttt{where} \\ &T = U \;\texttt{fby}\; U@(T+1) \\ &U = \textbf{if}\; Y == 0 \;\textbf{then}\; \# \;\textbf{else}\; \texttt{next}\; U \\ \texttt{end}& \end{aligned} \tag{14}$$

– NRWVR stands for *do not retreat whenever*. NRWVR chooses from its left-hand-side operand backwards only values in the current dimension where the right-hand-side evaluates to *false*.

```
$X$ NRWVR $Y = X$ RWVR NOT $Y =$
    **if** LAST $Y == 0$
    **then** $X$ PBY (PREV $X$ NRWVR PREV $Y$)
    **else** (PREV $X$ NRWVR PREV $Y$)
```

$$X \ \texttt{rnwvr} \ Y = X@T \ \texttt{where} \quad T = U \ \texttt{pby} \ U@(T-1) \quad U = \textbf{if } Y == 0 \textbf{ then } \# \textbf{ else } \texttt{prev} \ U \quad \texttt{end} \tag{15}$$

– ASA stands for *as soon as*. ASA returns the value of its left-hand-side as a first point in that stream as soon as the right-hand-side evaluates to *true*.
$X$ ASA $Y$ = FIRST ($X$ WVR $Y$)

$$X \ \texttt{asa} \ Y = \texttt{first} \ (X \ \texttt{wvr} \ Y) \tag{16}$$

– ALA (other suggested name is RASA) stands for *as late as* (or *reverse of a soon as*). ALA returns the value of its left-hand-side as the last point in that stream when the right-hand-side evaluates to *true* for the last time.
$X$ ALA $Y$ = LAST ($X$ WVR $Y$)

$$X \ \texttt{ala} \ Y = \texttt{last} \ (X \ \texttt{rwvr} \ Y) \tag{17}$$

– NASA stands for *not as soon as*. NASA returns the value of its left-hand-side as a first point in that stream as soon as the right-hand-side evaluates to *false*.
$X$ NASA $Y$ = FIRST ($X$ NWVR $Y$)

$$X \ \texttt{nasa} \ Y = \texttt{first} \ (X \ \texttt{nwvr} \ Y) \tag{18}$$

– NALA (other suggested name is NRASA) stands for *not as late as* (or *reverse of not a soon as*). NALA returns the value of its left-hand-side as the last point in that stream when the right-hand-side evaluates to *false* for the last time.
$X$ NALA $Y$ = LAST ($X$ NWVR $Y$)

$$X \ \texttt{nala} \ Y = \texttt{last} \ (X \ \texttt{nrwvr} \ Y) \tag{19}$$

– UPON stands for *advances upon*. Unlike ASA, UPON switches context of its left-hand-side operand if the right-hand side is *true*.

```
X UPON Y = X FBY (
    if FIRST Y ≠ 0
    then (NEXT X UPON NEXT Y)
    else (X UPON NEXT Y))
```

$$X \ \texttt{upon} \ Y = X@W \ \texttt{where} \quad W = 0 \ \texttt{fby} \ (\textbf{if } Y \textbf{ then } (W+1) \textbf{ else } W) \quad \texttt{end} \tag{20}$$

– RUPON stands for *retreats upon*. RUPON switches context backwards of its left-hand-side operand if the right-hand side is *true*.

```
X RUPON Y = X PBY (
    if LAST Y ≠ 0
    then (PREV X RUPON PREV Y)
    else (X RUPON PREV Y))
```

$$X \ \texttt{rupon} \ Y = X@W \ \texttt{where} \quad W = 0 \ \texttt{pby} \ (\textbf{if } Y \textbf{ then } (W-1) \textbf{ else } W) \quad \texttt{end} \tag{21}$$

– NUPON stands for *not advances upon* or rather *advances otherwise*. NUPON switches context of its left-hand-side operand if the right-hand side is *false*.
$X$ NUPON $Y = X$ UPON NOT $Y = X$ FBY (
    **if** FIRST $Y == 0$
    **then** (NEXT $X$ NUPON NEXT $Y$)
    **else** ($X$ NUPON NEXT $Y$))

$$X \texttt{ nupon } Y = X@W \texttt{ where} \quad W = 0 \texttt{ fby } (\textbf{if } Y == 0 \textbf{ then } (W+1) \textbf{ else } W) \quad \texttt{end} \tag{22}$$

– NRUPON stands for *not retreats upon*. NRUPON switches context backwards of its left-hand-side operand if the right-hand side is *false*.
$X$ NRUPON $Y = X$ RUPON NOT $Y = X$ PBY (
    **if** LAST $Y == 0$
    **then** (PREV $X$ NRUPON PREV $Y$)
    **else** ($X$ NRUPON PREV $Y$))

$$X \texttt{ nrupon } Y = X@W \texttt{ where} \quad W = 0 \texttt{ pby } (\textbf{if } Y == 0 \textbf{ then } (W-1) \textbf{ else } W) \quad \texttt{end} \tag{23}$$

## 2.4 Forensic Operators

The operators presented here are based on the discussion of the combination function and others that form more-than-primitive operations to support the required implementation. The discussed earlier `comb()` operator needs to be realized in the general manner for combining analogies of MPRs, which in our case are higher-level contexts, in the new language's dimension types.

– `combine` corresponds to the *comb* function as originally described by Gladyshev in [22]. It is defined in Listing 1.2. It is a preliminary context-enhanced version.

```
/**
 * Append given e to each element
 * of a given stream e under the
 * context of d.
 *
 * @return the resulting combined stream
 */
combine(s, e, d) =
  if iseod s then eod;
  else (first s fby.d e) fby.d combine(next s, e, d);
  fi
```

**Listing 1.2.** The `combine` Operator

– `product` tentatively corresponds to the cross-product [22] of contexts. It is defined in Listing 1.3.

The translated examples show recursion that we are not prepared to deal with in the current Lucid semantics, and will address that in the future work. The two illustrated operators are the first of the a few more to follow in the final language prototype.

```
/**
 * Append elements of s2 to element of s1
 * in all possible combinations.
 */
product(s1, s2, d) =
  if iseod s2 then eod;
  else combine(s1, first s2) fby.d product(s1, next s2);
  fi
```

**Listing 1.3.** The `product` Operator

## 2.5 Operational Semantics

As previously mentioned, the operational semantics of Forensic Lucid for the large part is viewed as a composition of the semantic rules of Indexical Lucid, Objective Lucid, and Lucx along with the new operators and definitions. Here we list the existing combined semantic definitions to be used the new language, specifically extracts of operational semantics from GIPL [11], and Lucx [13] are in Figure 1, and Figure 3 respectively. The explanation of the rules and the notation are given in great detail in the cited works and are trimmed in this article. For convenience of the reader they are recited here to a degree. The new rules of the operational semantics of Forensic Lucid cover the newly defined operators primarily, including the reverse and logical stream operators as well as forensic-specific operators. We use the same notation as the referenced languages to maintain consistency in defining our rules.

In the implementing system, GIPSY, the GIPL is the generic counterpart of all the Lucid programming languages. Like Indexical Lucid, which it is derived from, it has only the two standard intensional operators: `E @ C` for evaluating an expression `E` in context `C`, and `#d` for determining the position in dimension `d` of the current context of evaluation in the context space [11]. SIPLs are Lucid dialects (Specific Intensional Programming Languages) with their own attributes and objectives. Theoretically, all SIPLs can be translated into the GIPL [11]. All the SIPLs conservatively extend the GIPL syntactically and semantically. The remainder of this section presents a relevant piece of Lucx as a conservative extension to GIPL. The semantics of GIPL is presented in Figure 1. The excerpt of semantic rules of Lucx are then presented as a conservative extension to GIPL and Lucx in Figure 3. Following is the description of the GIPL semantic rules as presented in [11]:

$$\mathscr{D} \vdash E : v$$

tells that under the *definition environment* $\mathscr{D}$, expression $E$ would evaluate to value $v$.

$$\mathscr{D}, \mathscr{P} \vdash E : v$$

specifies that in the definition environment $\mathscr{D}$, and in the *evaluation context* $\mathscr{P}$ (sometimes also referred to as a *point* in the context space), expression $E$ evaluates to $v$. The definition environment $\mathscr{D}$ retains the definitions of all of the identifiers that appear in a Lucid program, as created with the semantic rules 13-16 in Figure 1. It is therefore a partial function

$$\mathscr{D} : \mathbf{Id} \to \mathbf{IdEntry}$$

where **Id** is the set of all possible identifiers and **IdEntry**, has five possible kinds of value, one for each of the kinds of identifier: 1. *Dimensions* define the coordinate pairs, in which one can navigate with the `#` and `@` operators. Their **IdEntry** is simply $(\texttt{dim})$. 2. *Constants* are external entities that provide a single value, regardless of the context of evaluation. Examples are integers and Boolean values. Their **IdEntry** is $(\texttt{const}, c)$, where $c$ is the value of the constant. 3. *Data operators* are external entities that provide memoryless functions. Examples are the arithmetic and Boolean functions. The constants and data operators are said to define the *basic algebra* of the language. Their **IdEntry** is $(\texttt{op}, f)$, where $f$ is the function itself. 4. *Variables* carry the multidimensional streams. Their **IdEntry** is $(\texttt{var}, E)$, where $E$ is the Lucid expression defining the variable. It should be noted that this semantics makes the assumption that all variable names are unique. This constraint is easy to overcome by performing compile-time renaming or using a nesting level environment scope when needed. 5. *Functions* are non-recursive GIPL user-defined functions. Their **IdEntry** is $(\texttt{func}, id_i, E)$, where the $id_i$ are the formal parameters to the function and $E$ is the body of the function. In this paper we do not discuss the semantics of recursive functions.

$$\mathbf{E_{cid}} : \frac{\mathcal{D}(id) = (\texttt{const}, c)}{\mathcal{D}, \mathcal{P} \vdash id : c} \tag{24}$$

$$\mathbf{E_{opid}} : \frac{\mathcal{D}(id) = (\texttt{op}, f)}{\mathcal{D}, \mathcal{P} \vdash id : id} \tag{25}$$

$$\mathbf{E_{did}} : \frac{\mathcal{D}(id) = (\texttt{dim})}{\mathcal{D}, \mathcal{P} \vdash id : id} \tag{26}$$

$$\mathbf{E_{fid}} : \frac{\mathcal{D}(id) = (\texttt{func}, id_i, E)}{\mathcal{D}, \mathcal{P} \vdash id : id} \tag{27}$$

$$\mathbf{E_{vid}} : \frac{\mathcal{D}(id) = (\texttt{var}, E) \qquad \mathcal{D}, \mathcal{P} \vdash E : v}{\mathcal{D}, \mathcal{P} \vdash id : v} \tag{28}$$

$$\mathbf{E_{op}} : \frac{\mathcal{D}, \mathcal{P} \vdash E : id \qquad \mathcal{D}(id) = (\texttt{op}, f) \qquad \mathcal{D}, \mathcal{P} \vdash E_i : v_i}{\mathcal{D}, \mathcal{P} \vdash E(E_1, \ldots, E_n) : f(v_1, \ldots, v_n)} \tag{29}$$

$$\mathbf{E_{fct}} : \frac{\mathcal{D}, \mathcal{P} \vdash E : id \qquad \mathcal{D}(id) = (\texttt{func}, id_i, E') \qquad \mathcal{D}, \mathcal{P} \vdash E'[id_i \leftarrow E_i] : v}{\mathcal{D}, \mathcal{P} \vdash E(E_1, \ldots, E_n) : v} \tag{30}$$

$$\mathbf{E_{c_T}} : \frac{\mathcal{D}, \mathcal{P} \vdash E : true \qquad \mathcal{D}, \mathcal{P} \vdash E' : v'}{\mathcal{D}, \mathcal{P} \vdash \texttt{if}\ E\ \texttt{then}\ E'\ \texttt{else}\ E'' : v'} \tag{31}$$

$$\mathbf{E_{c_F}} : \frac{\mathcal{D}, \mathcal{P} \vdash E : false \qquad \mathcal{D}, \mathcal{P} \vdash E'' : v''}{\mathcal{D}, \mathcal{P} \vdash \texttt{if}\ E\ \texttt{then}\ E'\ \texttt{else}\ E'' : v''} \tag{32}$$

$$\mathbf{E_{tag}} : \frac{\mathcal{D}, \mathcal{P} \vdash E : id \qquad \mathcal{D}(id) = (\texttt{dim})}{\mathcal{D}, \mathcal{P} \vdash \#E : \mathcal{P}(id)} \tag{33}$$

$$\mathbf{E_{at}} : \frac{\mathcal{D}, \mathcal{P} \vdash E' : id \qquad \mathcal{D}(id) = (\texttt{dim}) \qquad \mathcal{D}, \mathcal{P} \vdash E'' : v'' \qquad \mathcal{D}, \mathcal{P}\dagger[id \mapsto v''] \vdash E : v}{\mathcal{D}, \mathcal{P} \vdash E\ @E'\ E'' : v} \tag{34}$$

$$\mathbf{E_{w}} : \frac{\mathcal{D}, \mathcal{P} \vdash Q : \mathcal{D}', \mathcal{P}' \qquad \mathcal{D}', \mathcal{P}' \vdash E : v}{\mathcal{D}, \mathcal{P} \vdash E\ \texttt{where}\ Q : v} \tag{35}$$

$$\mathbf{Q_{dim}} : \frac{}{\mathcal{D}, \mathcal{P} \vdash \texttt{dimension}\ id : \mathcal{D}\dagger[id \mapsto (\texttt{dim})], \mathcal{P}\dagger[id \mapsto 0]} \tag{36}$$

$$\mathbf{Q_{id}} : \frac{}{\mathcal{D}, \mathcal{P} \vdash id = E : \mathcal{D}\dagger[id \mapsto (\texttt{var}, E)], \mathcal{P}} \tag{37}$$

$$\mathbf{Q_{fid}} : \frac{}{\mathcal{D}, \mathcal{P} \vdash id(id_1, \ldots, id_n) = E : \mathcal{D}\dagger[id \mapsto (\texttt{func}, id_i, E)], \mathcal{P}} \tag{38}$$

$$\mathbf{QQ} : \frac{\mathcal{D}, \mathcal{P} \vdash Q : \mathcal{D}', \mathcal{P}' \qquad \mathcal{D}', \mathcal{P}' \vdash Q' : \mathcal{D}'', \mathcal{P}''}{\mathcal{D}, \mathcal{P} \vdash Q\ Q' : \mathcal{D}'', \mathcal{P}''} \tag{39}$$

**Fig. 1.** GIPL Semantics

$$\mathbf{E_{E.did}} : \frac{\mathcal{D}(E.id) = (\texttt{dim})}{\mathcal{D}, \mathcal{P} \vdash E.id : id.id} \tag{40}$$

**Fig. 2.** Higher-Order Context Dot Operator

The evaluation context $\mathcal{P}$, which is changed when the `@` operator is evaluated, or a dimension is declared in a `where` clause, associates a *tag* (i.e. an index) to each relevant dimension. It is, therefore, a partial function

$$\mathcal{P} : \mathbf{Id} \rightarrow \mathbf{N}$$

Each type of identifiers can only be used in the appropriate situations. Identifiers of type `op`, `func`, and `dim` evaluate to themselves (Figure 1, rules 25,26,27). Constant identifiers (`const`) evaluate to the corresponding constant (Figure 1, rule 24). Function calls, resolved by the $\mathbf{E_{fct}}$ rule (Figure 1, rule 30), require the renaming of the formal parameters into the actual parameters (as represented by $E'[id_i \leftarrow E_i]$). The function $\mathscr{P}' = \mathscr{P} \dagger [id \mapsto v'']$ specifies that $\mathscr{P}'(x)$ is $v''$ if $x = id$, and $\mathscr{P}(x)$ otherwise. The rule for the `where` clause, $\mathbf{E_w}$ (Figure 1, rule 35), which corresponds to the syntactic expression $E$ `where` $Q$, evaluates $E$ using the definitions $Q$ therein. The additions to the definition environment $\mathscr{D}$ and context of evaluation $\mathscr{P}$ made by the $\mathbf{Q}$ rules (Figure 1, rules 36,37,38) are local to the current `where` clause. This is represented by the fact that the $\mathbf{E_w}$ rule returns neither $\mathscr{D}$ nor $\mathscr{P}$. The $\mathbf{Q_{dim}}$ rule adds a dimension to the definition environment and, as a convention, adds this dimension to the context of evaluation with tag 0 (Figure 1, rule 36). The $\mathbf{Q_{id}}$ and $\mathbf{Q_{fid}}$ simply add variable and function identifiers along with their definition to the definition environment (Figure 1, rules 37,38).

As a conservative extension to GIPL, Lucx's semantics introduces the notion of *context* as a building block into the semantic rules, i.e. *context as a first-class value*, as described by the rules in Figure 3. In Lucx, semantic rule 42 (Figure 3) creates a context as a semantic item and returns it as a context $\mathscr{P}$ that can then be used by rule 43 to navigate to this context by making it override the current context. GIPL's semantic rule 29 is still valid for the definition of the context operators, where the actual parameters evaluate to values $v_i$ that are contexts $\mathscr{P}_i$. The semantic rule 41 expresses that the # symbol evaluates to the current context. When used as a parameter to the context calculus operators, this allows for the generation of contexts relative to the current context of evaluation.

$$\mathbf{E_{\#(cxt)}} : \frac{}{\mathscr{D}, \mathscr{P} \vdash \# : \mathscr{P}} \tag{41}$$

$$\mathbf{E_{construction(cxt)}} : \frac{\begin{array}{ll} \mathscr{D}, \mathscr{P} \vdash E_{d_j} : id_j & \mathscr{D}(id_j) = (\texttt{dim}) \\ \mathscr{D}, \mathscr{P} \vdash E_{i_j} : v_j & \mathscr{P}' = \mathscr{P}_0 \dagger [id_1 \mapsto v_1] \dagger \ldots \dagger [id_n \mapsto v_n] \end{array}}{\mathscr{D}, \mathscr{P} \vdash [E_{d_1} : E_{i_1}, E_{d_2} : E_{i_2}, \ldots, E_{d_n} : E_{i_n}] : \mathscr{P}'} \tag{42}$$

$$\mathbf{E_{at(cxt)}} : \frac{\mathscr{D}, \mathscr{P} \vdash E' : \mathscr{P}' \qquad \mathscr{D}, \mathscr{P} \dagger \mathscr{P}' \vdash E : v}{\mathscr{D}, \mathscr{P} \vdash E \ @ \ E' : v} \tag{43}$$

**Fig. 3.** Conservative Semantic Rules Introduced by Lucx

## 3 Conclusion

While the list of Isabelle's proofs is incomplete at the time of the writing of this manuscript some formalization in Isabelle took place, and the work on them is currently on-going.

### 3.1 Results

Due to a non-standard nature of the Lucid language (as opposed to standard imperative languages), it takes some time to understand the full scope of some of its details and model them. This complicates a way to model its operators, expressions, overall meaning in Isabelle. This fact resulted in several trials and attempts to approach the language, from fairly complex to fairly basic – plain integers and pipelined processing and basic index support. They are not fully complete, but some of the basic properties are modeled and proven; please refer to the Isabelle sources for details (once completed it is planned to be released as a part of the Archive of Formal Proofs at [27]).

– The `IntegerLucid` Isabelle file is the most developed out of all as far as definition and exploitation of intensional operators of classical Lucid concerned. It is called "integer" because all the streams and dimensions and all operators around them play with integers, natural numbers, and in rarer cases Booleans. There are no identifiers in there. The Isabelle file contains three theories: `OriginalLucidOperators`, `LucidOperators`, and `IntegerLucid`. The first models classical Lucid operators as pipelined dataflows. The second adds up some index support and proves equivalence to the first definitions. The latter provides new definitions of the intensional operators through `@` and `#`, defines meaning functions, propositions, and lemmas from [11]. Integer Lucid proves the example for $N$ @.d $2 = 44$ for the `at()`.

– The `BasicLucid` theory is currently the second one derived to support Lucid definitions. It is an extension of IntegerLucid by adding identifiers. `ASA` and `UPON` are in this theory.
– The `LucidSemanticRules` theory is meant to have the meaning of complete semantic rules and proven, but it only has a definition of a Hoare tuple [28] and a meaning function for it.
– The `CommonLucidTypes` theory is used by all (most) theories and defines some common types used by most [29].
– `ForensicLucid.thy`, `GIPL.thy`, `IndexicalLucid.thy`, `JLucid.thy`, `JOOIP.thy`, `Lucx.thy`, `ObjectiveLucid.thy` are the theories under current development with some results from the above. The completed work will have a complete list of the files publicly available and submitted to the AfP [27].

### 3.2 Future Work

The near-future work will consist primarily of the following items:

– Complete semantics of all the mentioned Lucid dialects and their formalization with Isabelle.
– Augment the language specification to include the Depmster-Shafer theory [30,31] of evidence to allow weights for claims, credibility, belief, and plausibility parameters.
– Prove semantic rules involving intensional data warehouse.
– Implementation of the Forensic Lucid compiler, run-time and interactive development environments.

## 4 Acknowledgments

This research and development work was funded in part by NSERC and the Faculty of Engineering and Computer Science of Concordia University, Montreal, Canada. Thanks to Drs. Mourad Debbabi, Patrice Chalin, Peter Grogono on valuable suggestions used in this work.

## References

1. Mokhov, S.: Intensional Forensics – the Use of Intensional Logic in Cyberforensics. Technical report, Concordia Institute for Information Systems Engineering, Concordia University, Montreal, Canada (January 2007) ENGR6991 Technical Report.
2. Mokhov, S.: Intensional Cyberforensics – a PhD Proposal. Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada (December 2007)
3. The GIPSY Research and Development Group: The General Intensional Programming System (GIPSY) project. Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada (2002-2008) `http://newton.cs.concordia.ca/~gipsy/`, last viewed April 2008.
4. Paulson, L.C., Nipkow, T.: Isabelle: A generic proof assistant. University of Cambridge and Technical University of Munich (2007) `http://isabelle.in.tum.de/`, last viewed: December 2007.
5. Wadge, W., Ashcroft, E.: Lucid, the Dataflow Programming Language. Academic Press, London (1985)
6. Edward Ashcroft and Anthony Faustini and Raganswamy Jagannathan and William Wadge: Multidimensional, Declarative Programming. Oxford University Press, London (1995)
7. Ashcroft, E.A., Wadge, W.W.: Lucid - A Formal System for Writing and Proving Programs. Volume 5., SIAM J. Comput. no. 3 (1976)
8. Ashcroft, E.A., Wadge, W.W.: Erratum: Lucid - A Formal System for Writing and Proving Programs. Volume 6(1):200., SIAM J. Comput. (1977)
9. Ashcroft, E.A., Wadge, W.W.: Lucid, a nonprocedural language with iteration. Communication of the ACM **20**(7) (July 1977) 519–526
10. Gagné, J.R., Plaice, J.: Demand-Driven Real-Time Computing, World Scientific (September 1999)
11. Paquet, J.: Scientific Intensional Programming. PhD thesis, Department of Computer Science, Laval University, Sainte-Foy, Canada (1999)
12. Wan, K., Alagar, V., Paquet, J.: A Context theory for Intensional Programming. In: Workshop on Context Representation and Reasoning (CRR05), Paris, France. (July 2005)
13. Wan, K.: Lucx: Lucid Enriched with Context. PhD thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada (2006)

14. Paquet, J., Mokhov, S.A., Tong, X.: Design and implementation of context calculus in the GIPSY environment. In: Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC), Turku, Finland, IEEE Computer Society (July 2008) 1278–1283
15. Tong, X.: Design and implementation of context calculus in the GIPSY. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada (April 2008)
16. Wan, K., Alagar, V., Paquet, J.: Lucx: Lucid Enriched with Context. In: Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005), Las Vegas, USA, CSREA Press (June 2005) 48–14
17. Mokhov, S.A.: Towards syntax and semantics of hierarchical contexts in multimedia processing applications using MARFL. In: Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC), Turku, Finland, IEEE Computer Society (July 2008) 1288–1294
18. Swoboda, P.: A Formalisation and Implementation of Distributed Intensional Programming. PhD thesis, The University of New South Wales, Sydney, Australia (2004)
19. Swoboda, P., Wadge, W.W.: Vmake, ISE, and IRCS: General tools for the intensionalization of software systems. In Gergatsoulis, M., Rondogiannis, P., eds.: Intensional Programming II, World-Scientific (2000)
20. Swoboda, P., Plaice, J.: A new approach to distributed context-aware computing. In Ferscha, A., Hoertner, H., Kotsis, G., eds.: Advances in Pervasive Computing, Austrian Computer Society (2004) ISBN 3-85403-176-9.
21. Swoboda, P., Plaice, J.: An active functional intensional database. In Galindo, F., ed.: Advances in Pervasive Computing, Springer (2004) 56–65 LNCS 3180.
22. Gladyshev, P., Patel, A.: Finite state machine approach to digital event reconstruction. In: Digital Investigation Journal. Volume 2. (2004)
23. Gladyshev, P.: Finite state machine analysis of a blackmail investigation. In: International Journal of Digital Evidence, Technical and Security Risk Services, Sprint 2005, Volume 4, Issue 1 (2005)
24. Ding, Y.M.: Bi-directional translation between data-flow graphs and Lucid programs in the GIPSY environment. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada (2004)
25. Tong, X., Paquet, J., Mokhov, S.A.: Context Calculus in the GIPSY. Unpublished (2007)
26. Mokhov, S.A., Paquet, J., Debbabi, M.: Designing a language for intensional cyberforensic analysis. Unpublished (2007)
27. Klein, G., Nipkow, T., Paulson, L.C.: The archive of formal proofs. SourceForge.net (2008) `http://afp.sourceforge.net/`, last viewed: April 2008.
28. Moeller, A.: Program Verification with Hoare Logic. Technical report, University of Aarhus (2004) `http://www.brics.dk/~amoeller/talks/hoare.pdf`.
29. Mokhov, S.A., Paquet, J., Tong, X.: Hybrid intensional-imperative type system for intensional logic support in GIPSY. Submitted for publication at LPAR'08 (2008)
30. Shafer, G.: The Mathematical Theory of Evidence. Princeton University Press (1976)
31. Haenni, R., Kohlas, J., Lehmann, N.: Probabilistic argumentation systems. Technical report, Institute of Informatics, University of Fribourg, Fribourg, Switzerland (October 1999)
32. Kahn, G.: The semantics of a simple language for parallel processing. In: Proceedings of the IFIP Congress '74, Amsterdam, Elsevier North-Holland (1974) 471–475
33. Kahn, G., MacQueen, D.B.: Coroutines and networks of parallel processes. In: Proceedings of the IFIP Congress '77, Amsterdam, Elsevier North-Holland (1977) 993–998
34. Landin, P.J.: The next 700 programming languages. Communications of the ACM **9**(3) (1966) 157–166

# Appendix

## A Lucid Axioms, Theorems, and Proofs

Here we present and extend the notion of the formalisms from Paquet [11] and extend them on to the present work.

### A.1 Streaming and Basic Operators

The origins of Lucid date back to 1974. At that time, Ashcroft and Wadge were working on a purely declarative language, in which iterative algorithms could be expressed naturally, which eventually resulted in [9]. Their work fits into the broad area of research into program semantics and verification. It would later turn out that their work is also relevant to the dataflow networks and coroutines of Kahn and MacQueen [32,33]. In the original Lucid (whose operators are in $\texttt{THIS FONT}$), streams were defined in a pipelined manner, with two separate definitions: one for the initial element, and another one for the subsequent elements. For example, the equations

$$\begin{aligned} &\texttt{FIRST}\, X = 0 \\ &\texttt{NEXT}\, X \;\; = X + 1 \end{aligned}$$

define variable $X$ to be a stream, such that

$$x_0 = 0$$
$$x_{i+1} = x_i + 1$$

In other words,

$$\begin{aligned} 0 &= (0,0,0,...,0,...) \\ X &= (x_0, x_1, \ldots, x_i, \ldots) = (0, 1, \ldots, i, \ldots) \end{aligned}$$

Similarly, the equations

$$\begin{aligned} &\texttt{FIRST}\, X = X \\ &\texttt{NEXT}\, Y \;\; = Y + \texttt{NEXT}\, X \end{aligned}$$

define variable $Y$ to be the running sum of $X$, i.e.

$$y_0 = x_0$$
$$y_{i+1} = y_i + x_{i+1}$$

In other words,

$$Y = (y_0, y_1, \ldots, y_i, \ldots) = \left(0, 1, \ldots, \frac{i(i+1)}{2}, \ldots\right)$$

It soon became clear that a "new" operator at the time, `fby` (*followed by*) can be used to define such typical situations. Hence, the above two variables could be defined as follows:

$$\begin{aligned} X &= 0\; \texttt{FBY}\; X + 1 \\ Y &= X\; \texttt{FBY}\; Y + \texttt{NEXT}\, X \end{aligned}$$

As a result, we can summarize the three basic operators of the original Lucid.

---

**Definition 1** *If* $X = (x_0, x_1, \ldots, x_i, \ldots)$ *and* $Y = (y_0, y_1, \ldots, y_i, \ldots)$*, then*

$$\begin{aligned} &(1)\;\; \texttt{FIRST}\, X \overset{\text{def}}{=} (x_0, x_0, \ldots, x_0, \ldots) \\ &(2)\;\; \texttt{NEXT}\, X \;\; \overset{\text{def}}{=} (x_1, x_2, \ldots, x_{i+1}, \ldots) \\ &(3)\; X\; \texttt{FBY}\; Y \;\; \overset{\text{def}}{=} (x_0, y_0, y_1, \ldots, y_{i-1}, \ldots) \end{aligned}$$

---

Here parallels can be drawn to the list operations, where `first` corresponds to `head`, `next` corresponds to `tail`, and `fby` corresponds to `cons`. When these operators are combined with Landin's ISWIM [34] (*If You See What I Mean*), essentially typed $\lambda$-calculus with syntactic sugar, it becomes possible to define complete Lucid programs. The following three derived operators have turned out to be very useful (we will use them later in the text):

---

**Definition 2**

$$\begin{aligned} &(1)\; X\; \texttt{WVR}\; Y \;\; \overset{\text{def}}{=} \; \mathit{if}\; \texttt{FIRST}\, Y\; \mathit{then}\, X\; \texttt{FBY}\; (\, \texttt{NEXT}\, X\; \texttt{WVR}\; \texttt{NEXT}\, Y) \\ &\qquad\qquad\qquad\qquad\qquad\quad \mathit{else} \qquad\;\; (\, \texttt{NEXT}\, X\; \texttt{WVR}\; \texttt{NEXT}\, Y) \\ &(2)\; X\; \texttt{ASA}\; Y \;\; \overset{\text{def}}{=} \; \texttt{FIRST}\; (X\; \texttt{WVR}\; Y) \\ &(3)\; X\; \texttt{UPON}\; Y \overset{\text{def}}{=} X\; \texttt{FBY}\; (\mathit{if}\; \texttt{FIRST}\, Y\; \mathit{then}\, (\, \texttt{NEXT}\, X\; \texttt{UPON}\; \texttt{NEXT}\, Y) \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\;\; \mathit{else}\, ( \qquad X\; \texttt{UPON}\; \texttt{NEXT}\, Y)) \end{aligned}$$

Where wvr stands for *whenever*, asa stands for *as soon as* and upon stands for *advances upon*.

## A.2 Random Access to Streams

With the original Lucid operators, one could only define programs with pipelined dataflows, i.e. in which the $(i+1)$-th element in a stream is only computed once the *i*-th element has been computed. This situation is potentially wasteful of resources, since the *i*-th element might not necessarily be required. More importantly, it only allows sequential access into streams.

By taking a different approach, it is possible to have random access into streams, using an index # corresponding to the current position, the current context of evaluation. No longer are we manipulating infinite extensions (streams), rather we are defining computation according to a context (here a single integer). We have set out on the road to intensional programming. We redefine all ORIGINAL Lucid operators in terms of the operators # and @:

**Definition 3**

$$
\begin{array}{ll}
(1)\ \# & \overset{\text{def}}{=} 0\ \texttt{FBY}\ (\#+1) \\
(2)\ X\ @\ Y & \overset{\text{def}}{=} \mathit{if}\, Y=0\ \mathit{then}\ \texttt{FIRST}\, X \\
 & \quad \mathit{else}\,(\ \texttt{NEXT}\, X)\ @\,(Y-1)
\end{array}
$$

Further, we give definitions for the original operators using these two baseline operators. In so doing, we will use the following axioms.

---

**Axiom 1** *Let $i \geq 0$.*

$$
\begin{array}{l}
(1)\ [c]_i = c \\
(2)\ [X + c]_i = [X]_i + c \\
(3)\ [\,\texttt{FIRST}\, X]_i = [X]_0 \\
(4)\ [\,\texttt{NEXT}\, X]_i = [X]_{i+1} \\
(5)\ [X\ \texttt{FBY}\ Y]_0 = [X]_0 \\
(6)\ [X\ \texttt{FBY}\ Y]_{i+1} = [Y]_i \\
(7)\ \textit{if true then}\ [X]_i\ \textit{else}\ [Y]_i = [X]_i \\
(8)\ \textit{if false then}\ [X]_i\ \textit{else}\ [Y]_i = [Y]_i \\
(9)\ [\textit{if}\ C\ \textit{then}\ X\ \textit{else}\ Y]_i = \textit{if}\ [C]_i\ \textit{then}\ [X]_i\ \textit{else}\ [Y]_i
\end{array}
$$

---

Prior giving the re-definitions of the standard Lucid operators, we show some basic properties of @ and #. We will use throughout the discussion here $[X]_i$ instead of $x_i$, as it allows for greater readibility. Furthermore, we will, as is standard, write $X = Y$ whenever we have

$$(\forall i : i \geq 0 : [X]_i = [Y]_i)$$

---

**Proposition 1.** *Let $i \geq 0$.*

$$
\begin{array}{l}
(1)\ [\#]_i = i \\
(2)\ [X\ @\ Y]_i = [X]_{[Y]_i}
\end{array}
$$

---

*Proof*
(1) Proof by induction over $i$.
**Base step** ($i = 0$).

$$
\begin{array}{rll}
[\#]_0 & = [0\ \texttt{FBY}\ (\# + 1)]_0 & \text{Defn. 3.1} \\
& = [0]_0 & \text{Axiom 1.5} \\
& = 0 & \text{Axiom 1.1}
\end{array}
$$

**Induction step** ($i = k + 1$). Suppose $(\forall i : i \leq k : [\#]_i = i)$.

$$
\begin{array}{rll}
[\#]_{k+1} & = [0\ \texttt{FBY}\ (\# + 1)]_{k+1} & \text{Defn. 3.1} \\
& = [\# + 1]_k & \text{Axiom 1.6} \\
& = [\#]_k + 1 & \text{Axiom 1.2} \\
& = k + 1 & \text{Ind. Hyp.}
\end{array}
$$

Hence $(\forall i : i \geq 0 : [\#]_i = i)$.

(2) Let $i \geq 0$. We will prove by induction over $y_i$ that $y_i \geq 0 \Rightarrow [X\ @\ Y]_i = [X]_{[Y]_i}$.
**Base step** ($y_i = 0$).

$$
\begin{array}{rll}
[X\ @\ Y]_i & = [\texttt{if}\ Y = 0\ \texttt{then}\ \texttt{FIRST}\ X\ \texttt{else}\ (\,\texttt{NEXT}\ X)\ @\ (Y - 1)]_i & \text{Defn. 3.2} \\
& = \texttt{if}\ [Y = 0]_i\ \texttt{then}\ [\,\texttt{FIRST}\ X]_i\ \texttt{else}\ [(\,\texttt{NEXT}\ X)\ @\ (Y - 1)]_i & \text{Axiom 1.9} \\
& = \texttt{if}\ [Y]_i = 0\ \texttt{then}\ [\,\texttt{FIRST}\ X]_i\ \texttt{else}\ [(\,\texttt{NEXT}\ X)\ @\ (Y - 1)]_i & \text{Axiom 1.2} \\
& = [\,\texttt{FIRST}\ X]_i & \text{Axiom 1.7} \\
& = [X]_0 & \text{Axiom 1.3} \\
& = [X]_{[Y]_i} & \text{Hypothesis}
\end{array}
$$

**Induction step** ($y_i = k+1$). Suppose $(\forall i : i \le k : [\#]_i = i)$.

$$\begin{aligned}
[X \mathbin{@} Y]_i &= [\texttt{if } Y = 0 \texttt{ then } \textsf{FIRST}\, X \texttt{ else } (\textsf{NEXT}\, X) \mathbin{@} (Y-1)]_i && \text{Defn. 3.2}\\
&= \texttt{if } [Y=0]_i \texttt{ then } [\textsf{FIRST}\, X]_i \texttt{ else } [(\textsf{NEXT}\, X) \mathbin{@} (Y-1)]_i && \text{Axiom 1.9}\\
&= \texttt{if } [Y]_i = 0 \texttt{ then } [\textsf{FIRST}\, X]_i \texttt{ else } [(\textsf{NEXT}\, X) \mathbin{@} (Y-1)]_i && \text{Axiom 1.2}\\
&= [(\textsf{NEXT}\, X) \mathbin{@} (Y-1)]_i && \text{Axiom 1.8}\\
&= [\textsf{NEXT}\, X]_{[Y-1]_i} && \text{Ind. Hyp.}\\
&= [\textsf{NEXT}\, X]_{[Y]_i - 1} && \text{Axiom 1.2}\\
&= [X]_{[Y]_i - 1 + 1} && \text{Axiom 1.4}\\
&= [X]_{[Y]_i} && \text{Arith.}
\end{aligned}$$

Hence $(\forall i : i \ge 0 : [Y]_i \ge 0 \Rightarrow ([X \mathbin{@} Y]_i = [X]_{[Y]_i}))$. □

---

**Definition 4**

$$\begin{array}{lll}
(1) & \texttt{first}\, X & \overset{\text{def}}{=} X \mathbin{@} 0\\
(2) & \texttt{next}\, X & \overset{\text{def}}{=} X \mathbin{@} (\#+1)\\
(3) & X \texttt{ fby } Y & \overset{\text{def}}{=} \textit{if } \# = 0 \textit{ then } X \textit{ else } Y \mathbin{@} (\#-1)\\
(4) & X \texttt{ wvr } Y & \overset{\text{def}}{=} X \mathbin{@} T\\
 & & \quad \textit{where}\\
(4.1) & & \qquad T = U \texttt{ fby } U \mathbin{@} (T+1)\\
(4.2) & & \qquad U = \textit{if } Y \textit{ then } \# \textit{ else } \texttt{next}\, U\\
 & & \quad \textit{end}\\
(5) & X \texttt{ asa } Y & \overset{\text{def}}{=} \texttt{first}\, (X \texttt{ wvr } Y)\\
(6) & X \texttt{ upon } Y & \overset{\text{def}}{=} X \mathbin{@} W\\
 & & \quad \textit{where}\\
(6.1) & & \qquad W = 0 \texttt{ fby } \textit{if } Y \textit{ then } (W+1) \textit{ else } W\\
 & & \quad \textit{end}
\end{array}$$

---

The advantage of these new definitions is that they do not use any form of recursive function definitions. Rather, all of the definitions are iterative, and in practice, more easily implemented in an efficient manner. We prove below that the `new` definitions are equivalent to the `OLD` ones.

---

**Proposition 2.** $\texttt{first}\, X = \textsf{FIRST}\, X$.

---

*Proof* Let $i \ge 0$. Then

$$\begin{aligned}
[\texttt{first}\, X]_i &= [X \mathbin{@} 0]_i && \text{Defn. 4.1}\\
&= [X]_{[0]_i} && \text{Prop. 1.2}\\
&= [X]_0 && \text{Axiom 1.1}\\
&= [\textsf{FIRST}\, X]_i && \text{Axiom 1.3}
\end{aligned}$$

Hence $\texttt{first}\, X = \textsf{FIRST}\, X$. □

**Proposition 3.** $\texttt{next}\,X = \texttt{NEXT}\,X$.

*Proof* Let $i \geq 0$. Then

$$\begin{aligned}
[\texttt{next}\,X]_i &= \big[X \mathbin{\texttt{@}} (\texttt{\#}+1)\big]_i && \text{Defn. 4.2}\\
&= [X]_{[\texttt{\#}+1]_i} && \text{Prop. 1.2}\\
&= [X]_{[\texttt{\#}]_i+1} && \text{Axiom 1.2}\\
&= [X]_{i+1} && \text{Prop. 1.1}\\
&= [\,\texttt{NEXT}\,X]_i && \text{Axiom 1.4}
\end{aligned}$$

Hence $\texttt{next}\,X = \texttt{NEXT}\,X$. □

**Proposition 4.** $X \mathbin{\texttt{fby}} Y = X \mathbin{\texttt{FBY}} Y$.

*Proof* Proof by induction over $i$.

**Base step** ($i = 0$).

$$\begin{aligned}
[X \mathbin{\texttt{fby}} Y]_0 &= [\texttt{if}\ \texttt{\#} = 0\ \texttt{then}\ X\ \texttt{else}\ Y \mathbin{\texttt{@}} (\texttt{\#}-1)]_0 && \text{Defn. 4.3}\\
&= \texttt{if}\ [\texttt{\#} = 0]_0\ \texttt{then}\ [X]_0\ \texttt{else}\ [Y \mathbin{\texttt{@}} (\texttt{\#}-1)]_0 && \text{Axiom 1.9}\\
&= \texttt{if}\ [\texttt{\#}]_0 = 0\ \texttt{then}\ [X]_0\ \texttt{else}\ [Y \mathbin{\texttt{@}} (\texttt{\#}-1)]_0 && \text{Defn. 1.2}\\
&= \texttt{if}\ 0 = 0\ \texttt{then}\ [X]_0\ \texttt{else}\ [Y \mathbin{\texttt{@}} (\texttt{\#}-1)]_0 && \text{Prop. 1.1}\\
&= [X]_0 && \text{Axiom 1.7}\\
&= [X \mathbin{\texttt{FBY}} Y]_0 && \text{Axiom 1.5}
\end{aligned}$$

**Induction step** ($i = k+1$).

$$\begin{aligned}
[X \mathbin{\texttt{fby}} Y]_{k+1} &= \big[\texttt{if}\ \texttt{\#} = 0\ \texttt{then}\ X\ \texttt{else}\ Y \mathbin{\texttt{@}} (\texttt{\#}-1)\big]_{k+1} && \text{Defn. 4.3}\\
&= \texttt{if}\ [\texttt{\#} = 0]_{k+1}\ \texttt{then}\ [X]_{k+1}\ \texttt{else}\ [Y \mathbin{\texttt{@}} (\texttt{\#}-1)]_{k+1} && \text{Axiom 1.9}\\
&= \texttt{if}\ [\texttt{\#}]_{k+1} = 0\ \texttt{then}\ [X]_{k+1}\ \texttt{else}\ [Y \mathbin{\texttt{@}} (\texttt{\#}-1)]_{k+1} && \text{Axiom 1.1}\\
&= \texttt{if}\ k+1 = 0\ \texttt{then}\ [X]_{k+1}\ \texttt{else}\ [Y \mathbin{\texttt{@}} (\texttt{\#}-1)]_{k+1} && \text{Prop. 1.1}\\
&= \big[Y \mathbin{\texttt{@}} (\texttt{\#}-1)\big]_{k+1} && \text{Axiom 1.8}\\
&= \big[Y\big]_{[\texttt{\#}-1]_{k+1}} && \text{Prop. 1.2}\\
&= \big[Y\big]_{[\texttt{\#}]_{k+1}-1} && \text{Axiom 1.2}\\
&= \big[Y\big]_k && \text{Prop. 1.1}\\
&= [X \mathbin{\texttt{FBY}} Y]_{k+1} && \text{Axiom 1.6}
\end{aligned}$$

Hence $(\forall i : i \geq 0 : [X \mathbin{\texttt{fby}} Y]_i = [X \mathbin{\texttt{FBY}} Y]_i)$. Hence $\texttt{fby} = \texttt{FBY}$. □

The proof for `wvr` is more complicated, as it requires relating an iterative definition to a recursive definition. We will therefore need four lemmas that refer to variables $T$ and $U$ in the text in Definitions 4.4.1 and 4.4.2. In addition, we must define the *rank* of a Boolean stream. Finally, we will have to introduce another set of axioms, that allow us to compare two entire streams, as opposed to particular elements in the two streams.

**Axiom 2** *Let $i \geq 0$.*

$$\begin{array}{l}
(1)\ X^0 = X \\
(2)\ [X^i]_0 = [X]_i \\
(3)\ \texttt{FIRST}\, X^i = [X]_i \\
(4)\ \texttt{NEXT}\, X^i = X^{i+1} \\
(5)\ \texttt{NEXT}\,(X\ \texttt{FBY}\ Y) = Y \\
(6)\ (\ \texttt{FIRST}\, X)\ \texttt{FBY}\ Y = X\ \texttt{FBY}\ Y \\
(7)\ \mathit{if\ true\ then}\, X\ \mathit{else}\, Y = X \\
(8)\ \mathit{if\, false\ then}\, X\ \mathit{else}\, Y = Y
\end{array}$$

**Definition 5** *Let $Y$ be a Boolean stream.*

$$\begin{array}{ll}
(1)\ \mathit{rank}(-1,Y) & \overset{\text{def}}{=} -1 \\
(2)\ \mathit{rank}(i+1,Y) & \overset{\text{def}}{=} \min\{k : k > \mathit{rank}(i,Y) : [Y]_k = \mathit{true}\}
\end{array}$$

Further, we write $r_i$ for $\texttt{rank}(i,Y)$.

**Lemma 1.** $(\forall i : i \geq -1 : (\forall j : r_i < j \leq r_{i+1} : X^j\ \texttt{WVR}\ Y^j = X^{r_{i+1}}\ \texttt{WVR}\ Y^{r_{i+1}}))$.

*Proof* Let $i \geq -1$. Proof by downwards induction over $j$. Note that $r_i < r_{i+1}$.

**Base step** $(j = r_{i+1})$.

$$X^{r_{i+1}}\ \texttt{WVR}\ Y^{r_{i+1}} = X^{r_{i+1}}\ \texttt{WVR}\ Y^{r_{i+1}} \qquad \text{Identity}$$

**Induction step** $(j = k-1,\ j > r_i)$.

$$\begin{array}{rll}
X^{k-1}\ \texttt{WVR}\ Y^{k-1} & = \texttt{if}\ \texttt{FIRST}\ Y^{k-1}\ \texttt{then}\ X^{k-1}\ \texttt{FBY}\ X^k\ \texttt{WVR}\ Y^k & \text{Defn. 2.1} \\
 & \qquad\qquad\qquad\quad \texttt{else}\ X^k\ \texttt{WVR}\ Y^k & \\
 & = \texttt{if}\ [Y]_{k-1}\ \texttt{then}\ X^{k-1}\ \texttt{FBY}\ X^k\ \texttt{WVR}\ Y^k & \text{Axiom 2.3} \\
 & \qquad\qquad \texttt{else}\ X^k\ \texttt{WVR}\ Y^k & \\
 & = X^k\ \texttt{WVR}\ Y^k & \text{Axiom 2.8} \\
 & = X^{r_{i+1}}\ \texttt{WVR}\ Y^{r_{i+1}} & \text{Ind. Hyp.}
\end{array}$$

Hence, $(\forall i : i \geq -1 : (\forall j : r_i < j \leq r_{i+1} : X^j\ \texttt{WVR}\ Y^j = X^{r_{i+1}}\ \texttt{WVR}\ Y^{r_{i+1}}))$. □

**Lemma 2.** $(\forall i : i \geq 0 : (X\ \texttt{WVR}\ Y)^i = X^{r_i}\ \texttt{WVR}\ Y^{r_i})$.

*Proof* Proof by induction over $i$.

**Base step** $(i = 0)$.

$$\begin{array}{rll}
(X\ \texttt{WVR}\ Y)^0 & = X\ \texttt{WVR}\ Y & \text{Axiom 2.1} \\
 & = X^0\ \texttt{WVR}\ Y^0 & \text{Axiom 2.1} \\
 & = X^{r_0}\ \texttt{WVR}\ Y^{r_0} & \text{Lemma 1}
\end{array}$$

**Induction step** ($i = k + 1$).

$$\begin{aligned}
(X \;\texttt{WVR}\; Y)^{k+1} &= \texttt{NEXT}\,((X \;\texttt{WVR}\; Y)^{k}) && \text{Axiom 2.4}\\
&= \texttt{NEXT}\,(X^{r_k} \;\texttt{WVR}\; Y^{r_k}) && \text{Ind. Hyp.}\\
&= \texttt{NEXT}\,(\texttt{if FIRST}\; Y^{r_k} \;\texttt{then}\; X^{r_k} \;\texttt{FBY}\; X^{r_k+1} \;\texttt{WVR}\; Y^{r_k+1} && \text{Defn. 2.1}\\
&\qquad\qquad \texttt{else}\; X^{r_k+1} \;\texttt{WVR}\; Y^{r_k+1})\\
&= \texttt{NEXT}\,(\texttt{if}\; [Y]_{r_k} \;\texttt{then}\; X^{r_k} \;\texttt{FBY}\; X^{r_k+1} \;\texttt{WVR}\; Y^{r_k+1} && \text{Axiom 2.3}\\
&\qquad\qquad \texttt{else}\; X^{r_k+1} \;\texttt{WVR}\; Y^{r_k+1})\\
&= \texttt{NEXT}\,(X^{r_k} \;\texttt{FBY}\; X^{r_k+1} \;\texttt{WVR}\; Y^{r_k+1}) && \text{Axiom 2.7}\\
&= X^{r_k+1} \;\texttt{WVR}\; Y^{r_k+1} && \text{Axiom 2.5}\\
&= X^{r_{k+1}} \;\texttt{WVR}\; Y^{r_{k+1}} && \text{Lemma 1}
\end{aligned}$$

Hence, $(\forall i : i \geq 0 : (X \;\texttt{WVR}\; Y)^{i} = X^{r_i} \;\texttt{WVR}\; Y^{r_i})$. $\square$

---

**Lemma 3.** $(\forall i : i \geq -1 : (\forall j : r_i < j \leq r_{i+1} : [U]_j = r_{i+1}))$.

---

*Proof* Let $i \geq -1$. Proof by downwards induction over $j$. Note that $r_i < r_{i+1}$.

**Base step** ($j = r_{i+1}$).

$$\begin{aligned}
[U]_{r_{i+1}} &= [\texttt{if}\; Y \;\texttt{then \# else next}\; U]_{r_{i+1}} && \text{Defn. 4.4.2}\\
&= \texttt{if}\; [Y]_{r_{i+1}} \;\texttt{then}\; [\texttt{\#}]_{r_{i+1}} \;\texttt{else}\; [\texttt{next}\; U]_{r_{i+1}} && \text{Axiom 1.9}\\
&= [\texttt{\#}]_{r_{i+1}} && \text{Axiom 1.7}\\
&= r_{i+1} && \text{Prop. 1.1}
\end{aligned}$$

**Induction step** ($j = k - 1,\; j > r_i$).

$$\begin{aligned}
[U]_{k-1} &= [\texttt{if}\; Y \;\texttt{then \# else next}\; U]_{k-1} && \text{Defn. 4.4.2}\\
&= \texttt{if}\; [Y]_{k-1} \;\texttt{then}\; [\texttt{\#}]_{k-1} \;\texttt{else}\; [\texttt{next}\; U]_{k-1} && \text{Axiom 1.9}\\
&= [\texttt{next}\; U]_{k-1} && \text{Axiom 1.8}\\
&= [U]_{k} && \text{Axiom 1.4}\\
&= r_{i+1} && \text{Ind. Hyp.}
\end{aligned}$$

Hence, $(\forall i : i \geq -1 : (\forall j : r_{i-1} < j < r_i : [U]_j = r_{i+1}))$. $\square$

**Lemma 4.** $(\forall i : i \geq 0 : [T]_i = r_i)$.

*Proof* Proof by induction over $i$.

**Base step** $(i = 0)$.

$$\begin{aligned}
[T]_0 &= [U \;\texttt{fby}\; U \;\texttt{@}\; (T+1)]_0 && \text{Defn. 4.4.1}\\
&= [U]_0 && \text{Axiom 1.5}\\
&= r_0 && \text{Lemma 3}
\end{aligned}$$

**Induction step** $(i = k+1)$.

$$\begin{aligned}
[T]_{k+1} &= [U \;\texttt{fby}\; U \;\texttt{@}\; (T+1)]_{k+1} && \text{Defn. 4.4.1}\\
&= [U \;\texttt{@}\; (T+1)]_k && \text{Axiom 1.6}\\
&= [U]_{[T+1]_k} && \text{Prop. 1.2}\\
&= [U]_{[T]_k+1} && \text{Axiom 1.2}\\
&= [U]_{r_k+1} && \text{Ind. Hyp.}\\
&= r_{k+1} && \text{Lemma 3}
\end{aligned}$$

Hence, $(\forall i : i \geq 0 : [T]_i = r_i)$. □

**Proposition 5.** $X \;\texttt{wvr}\; Y = X \;\texttt{WVR}\; Y$.

*Proof*

$$\begin{aligned}
[X \;\texttt{wvr}\; Y]_i &= [X \;\texttt{@}\; T]_i && \text{Defn. 4.4}\\
&= [X]_{[T]_i} && \text{Prop. 1.2}\\
&= [X]_{r_i} && \text{Lemma 4}\\
&= [X^{r_i}]_0 && \text{Axiom 1.2}\\
&= [X^{r_i} \;\texttt{FBY}\; X^{r_i+1} \;\texttt{WVR}\; Y^{r_i+1}]_0 && \text{Axiom 1.6}\\
&= [\texttt{if}\; [Y]_{r_i} \;\texttt{then}\; X^{r_i} \;\texttt{FBY}\; X^{r_i+1} \;\texttt{WVR}\; Y^{r_i+1} && \text{Axiom 2.7}\\
&\qquad\quad \texttt{else}\; X^{r_i+1} \;\texttt{WVR}\; Y^{r_i+1}]_0 &&\\
&= [\texttt{if}\; \texttt{FIRST}\; Y^{r_i} \;\texttt{then}\; X^{r_i} \;\texttt{FBY}\; X^{r_i+1} \;\texttt{WVR}\; Y^{r_i+1} && \text{Axiom 2.3}\\
&\qquad\qquad\quad \texttt{else}\; X^{r_i+1} \;\texttt{WVR}\; Y^{r_i+1}]_0 &&\\
&= [X^{r_i} \;\texttt{WVR}\; Y^{r_i}]_0 && \text{Defn. 2.1}\\
&= [(X \;\texttt{WVR}\; Y)^i]_0 && \text{Lemma 2}\\
&= [X \;\texttt{WVR}\; Y]_i && \text{Axiom 2.2}
\end{aligned}$$

Hence $X \;\texttt{wvr}\; Y = X \;\texttt{WVR}\; Y$. □

**Proposition 6.** $X \;\texttt{asa}\; Y = X \;\texttt{ASA}\; Y$.

*Proof*

$$\begin{aligned}
X \;\texttt{asa}\; Y &= \texttt{first}\; (X \;\texttt{wvr}\; Y) && \text{Defn. 4.5}\\
&= \texttt{first}\; (X \;\texttt{WVR}\; Y) && \text{Prop. 5}\\
&= \texttt{FIRST}\; (X \;\texttt{WVR}\; Y) && \text{Prop. 2}\\
&= X \;\texttt{ASA}\; Y && \text{Defn. 2.2}
\end{aligned}$$

Hence $X \;\texttt{asa}\; Y = X \;\texttt{ASA}\; Y$. □

**Lemma 5.** $(\forall i : i \geq 0 : (X \texttt{ UPON } Y)^i = X^{[W]_i} \texttt{ UPON } Y^i)$

*Proof* Proof by induction over $i$.

**Base step** $(i = 0)$.

$$\begin{aligned}
(X \texttt{ UPON } Y)^0 &= X \texttt{ UPON } Y && \text{Axiom 2.1}\\
&= X^0 \texttt{ UPON } Y^0 && \text{Axiom 2.1}\\
&= X^{[0 \texttt{ fby } \ldots]_0} \texttt{ UPON } Y^0 && \text{Defn. 2.3}\\
&= X^{[W]_0} \texttt{ UPON } Y^0 && \text{Defn. 4.6.1}
\end{aligned}$$

**Induction step** $(i = k+1)$.

$$\begin{aligned}
(X \texttt{ UPON } Y)^{k+1} &= \texttt{NEXT } \left((X \texttt{ UPON } Y)^k\right) && \text{Axiom 2.4}\\
&= \texttt{NEXT } \left(X^{[W]_k} \texttt{ UPON } Y^k\right) && \text{Ind. Hyp.}\\
&= \texttt{if } (\ \texttt{FIRST } Y^k) && \text{Defn. 2.3 and}\\
&\quad \texttt{then } (X^{[W]_k+1} \texttt{ UPON } Y^{k+1}) && \text{Axiom 2.5}\\
&\quad \texttt{else } (X^{[W]_k} \texttt{ UPON } Y^{k+1}) && \\
&= \texttt{if } [Y]_k && \text{Axiom 2.4}\\
&\quad \texttt{then } (X^{[W]_{k+1}} \texttt{ UPON } Y^{k+1}) && \text{Defn. 4.6.1}\\
&\quad \texttt{else } (X^{[W]_k} \texttt{ UPON } Y^{k+1}) && \\
&= \left(X^{(\texttt{if } [Y]_k \texttt{ then } [W]_{k+1} \texttt{ else } [W]_k)}\right) && \text{Substit.}\\
&\quad \texttt{UPON } Y^{k+1} && \\
&= X^{[W]_{k+1}} \texttt{ UPON } Y^{k+1} && \text{Defn. 4.6.1}
\end{aligned}$$

Hence, $(\forall i : i \geq 0 : (X \texttt{ UPON } Y)^i = X^{[W]_i} \texttt{ UPON } Y^i)$ □

**Proposition 7.** $X \texttt{ upon } Y = X \texttt{ UPON } Y$.

*Proof* Let $i \geq 0$. Then

$$\begin{aligned}
[X \texttt{ upon } Y]_i &= [X \texttt{ @ } W]_i && \text{Defn. 4.6}\\
&= [X]_{[W]_i} && \text{Prop. 1.2}\\
&= [X^{[W]_i}]_0 && \text{Axiom 2.2}\\
&= [X^{[W]_i} \texttt{ FBY } \ldots]_0 && \text{Axiom 1.5}\\
&= [X^{[W]_i} \texttt{ UPON } Y^i]_0 && \text{Defn. 2.3}\\
&= [X \texttt{ UPON } Y]_i && \text{Lemma 5}
\end{aligned}$$

Hence $X \texttt{ upon } Y = X \texttt{ UPON } Y$. □

Now that the corresponding definitions are shown to be equivalent, we can generalize and head off in the negative direction as well:

**Definition 6**

$$\begin{aligned}
&(1)\ \texttt{prev } X \overset{\text{def}}{=} X \texttt{ @ } (\# - 1)\\
&(2)\ X \texttt{ fby } Y \overset{\text{def}}{=} \mathit{if}\ \# \leq 0\ \mathit{then}\, X\ \mathit{else}\, Y \texttt{ @ } (\# - 1)
\end{aligned}$$

## B Summary of the Operators' Examples

Here we illustrate a few basic examples of application of the Forensic Lucid operators (both, classical Lucid and the newly introduced operators). Assume we have two bounded (between `bod` and `eod`) streams *X* and *Y* of ten elements. The *X* stream is just an ordered sequence of natural numbers between 1 and 10. If queried for values below 1 an beginning-of-data (`bod`) marker would be returned; similarly if queried beyond 10, the end-of-data marker (`eod`) is returned. The *Y* stream is a sequence of ten truth values (can be replaced with 0 for "false" and 1 for "true"). The operators applied to these streams may return bounded or unbounded streams of the same or different length than the original depending on the definition of a particular operator. Also assume the current dimension index is 0. The resulting table showing the application of the classical and the new operators is in Table 1.

| stream/index | -1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| X | bod | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | eod | eod |
| Y | bod | T | F | F | T | F | F | T | T | F | T | eod | eod |
| X FIRST Y | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | | |
| X LAST Y | | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | | |
| X NEXT Y | | | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | eod | eod |
| X PREV Y | | bod | | | | | | | | | | | |
| X FBY Y | | 1 | T | F | F | T | F | F | T | T | F | T | eod |
| X PBY Y | | T | F | F | T | F | F | T | T | F | T | 1 | eod |
| X WVR Y | | 1 | | | 4 | | | 7 | 8 | | 10 | | |
| X RWVR Y | | 10 | | | 8 | | | 7 | 4 | | 1 | | |
| X NWVR Y | | | 2 | 3 | | 5 | 6 | | | 9 | | | |
| X NRWVR Y | | | 9 | 6 | | 5 | 3 | | | 2 | | | |
| X ASA Y | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | | |
| X NASA Y | | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | | |
| X ALA Y | | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | | |
| X NALA Y | | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | | |
| X UPON Y | | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 4 | 5 | 5 | eod | |
| X RUPON Y | | 10 | 9 | 9 | 8 | 7 | 7 | 7 | 6 | 6 | 6 | bod | |
| X NUPON Y | | 1 | 1 | 2 | 3 | 3 | 4 | 5 | 5 | 5 | 6 | 6 | eod |
| X NRUPON Y | | 10 | 10 | 9 | 9 | 9 | 8 | 7 | 7 | 6 | 5 | 5 | bod |
| NEG X | | -1 | -2 | -3 | -4 | -5 | -6 | -7 | -8 | -9 | -10 | eod | eod |
| NOT Y | | F | T | T | F | T | T | F | F | T | F | eod | eod |
| X AND Y | | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | eod | eod |
| X OR Y | | 1 | 2 | 3 | 5 | 5 | 6 | 7 | 9 | 9 | 11 | eod | eod |
| X XOR Y | | 0 | 2 | 3 | 5 | 5 | 6 | 6 | 9 | 9 | 11 | eod | eod |

**Table 1.** Example of Application of Forensic Lucid Operators to Bounded Streams