Formally Specifying Operational Semantics and Language Constructs of Forensic Lucid

Serguei A. Mokhov   Joey Paquet   Mourad Debbabi
Faculty of Engineering and Computer Science
Concordia University, Montréal, Québec, Canada,
{mokhov,paquet,debbabi}@encs.concordia.ca

Abstract: The Forensic Lucid programming language is being developed for intensional cyberforensic case specification and analysis, including the syntax and operational semantics. In significant part, the language is based on its predecessor and codecessor Lucid dialects, such as GIPL, Indexical Lucid, Lucx, Objective Lucid, and JOOIP bound by the intensional higher-order logic that is behind them. This work continues to formally specify the operational semantics of the Forensic Lucid language extending the previous related work.

1 Introduction

We further define a functional-intensional programming language, called Forensic Lucid. This forensic case specification language is under extensive design and development including its syntax, semantics, their formalization and correctness proofs, the corresponding compiler, run-time environment, and interactive development environment. This work further extends our previous developments in the related work [Mok04, Mok07b, Mok07a, MP08a].

1.1 Problem Statement

A lot of Lucid dialects have been spawned from the 30+-year-old functional intensional programming language called Lucid [WA85, Edw95, Zha97, AW76, AW77a, AW77b, FB91, Du94, GP99, Paq99, WAP05a, Agi95]. Lucid itself was invented with a goal for program correctness verification at the time. While there were a number of operational semantics rules for compilers and run-time environments developed for all those dialects throughout the years. In this work we discuss a new dialect of Lucid that has been created to foster the research in *intensional cyberforensics* (i.e. *multidimensional context-oriented cyberforensic specification and analysis*), called Forensic Lucid, which, in large part, is a union of the syntax and operational semantics inference rules from the comprising languages with the forensic extensions based on the finite-state automate approach [Gla05, GP04]. In order to be a credible tool to use in court to implement relevant tools for the argumentation, the language must have a solid scientific base, a part of which is a complete formalizing the syntax and semantics the language.

1.2 Proposed Solution

Based on the previous work to begin mechanized specification and proofs of Forensic Lucid constructs, their equivalence to the comprising dialects for the correctness aspect verification, in Isabelle [PN07], a prover assistant program [MP08a], we further expand and refine the syntax and operational semantics' inference rules extended with the ones of the comprising Lucid dialects. We proceed with the "core" Lucid dialects such as GIPL [Paq99, PMT08], a conservative subset of Lucx [PMT08] to Objective Lucid [MP05, Mok05], JOOIP [WPM08], MARFL [Mok08] and finally Forensic Lucid, to arrive to a comprehensive set of syntactic and semantic rules covering the dialects.

1.3 Organization

We first rehash the notion of intensional logic and programming for the unaware reader. We present the refined syntax and the semantics of the language properly attributing the inherited language constructs and rules, and the extensions of the language.

1.4 Intensional Programming

1.4.1 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 are considered). IP brings in **dimensions** and **context** to programs (eg. 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) [MP08b, MP08a, Ron94] 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 [Wan06, PMT08].

1.4.2 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: $\{\texttt{place}:\textbf{here}, \texttt{time}:\textbf{today}\}$

(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 **Sydney** 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 ...
Sydney    F F T T T F F F T ...
Moscow    F F F F T T T F F ...
Ottawa    F T T T T T F F F ...
```

1.4.3 Lucid Summary

Lucid [WA85, Edw95, AW77b, AW76, AW77a] 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 [TPM07, WAP05b, Wan06]. The generic version of Lucid, GIPL [Paq99], defines two basic operators @ and # to navigate in the contexts (switch and query). The GIPL is the first[1] 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 [Paq99]. 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 [WAP05b, Wan06, PMT08] 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. Luckily, such a proposition by has already been made [Mok08] and needs some modifications to the expressions of the cyberforensic context.

[1]The second being Lucx.

JLucid, Objective Lucid, and JOOIP. JLucid [Mok05, GMP05] was a first attempt on intensional arrays and "free Java functions" in the GIPSY. The approach used the Lucid language as the driving main computation, where Java methods were peripheral and could be invoked from the Lucid part, but not the other way around. This was the first instance of hybrid programming within the GIPSY. The semantics of this approach was not completely defined, plus, it was only one-sided view (Lucid-to-Java) of the problem. JLucid did not support objects of any kind, but introduced the wrapper class idea for the free Java methods and served as a precursor to Objective Lucid.

Objective Lucid [Mok05, MP05] is an extension of the JLucid language that inherits all of the JLucid's features and introduced Java objects to be available for use by Lucid. Objective Lucid expanded the notion of the Java object (a collection of members of different types) to the array (a collection of members of the same type) and first introduced the dot-notation in the syntax and operational semantics in GIPSY. Like in JLucid, Objective Lucid's focus was on the Lucid part being the "main" program and did not allow Java to call intensional functions or use intensional constructs from within a Java class. Objective Lucid was the first in GIPSY to introduce the more complete operational semantics of the hybrid OO intensional language.

JOOIP [WPM08] greatly complements Objective Lucid by allowing Java to call the intensional language constructs closing the gap and making JOOIP a complete hybrid OO intensional programming language within the GIPSY environment. JOOIP's semantics further refines in a greater detail the operational semantics rules of Lucid and Objective Lucid in the attempt to make them complete.

MARFL. While not of strictly Lucid family or GIPSY, MARFL [Mok08] was nearly entirely influenced by Lucid, and is based on overloaded `@` and `#` operators as well as allows to navigate into the depth of the higher-order contextual space using the dot operator. The latter was indirectly (re-invented in part) influenced by iHTML and libintense [Swo04, SW00].

1.5 General Intensional Programming System (GIPSY)

The GIPSY [VP05, PK00, The08, Lu04, PGW04, Mok05, WP05, PW05] is a platform implemented primarily in Java to investigate properties of the Lucid family of languages and beyond. It executes Lucid programs following a demand-driven distributed generator-worker architecture, and is designed as a modular collection of frameworks where components related to the development (RIPE[2]), compilation (GIPC[3]), and execution (GEE[4]) of Lucid programs are separated allowing easy extension, addition, and replacement of the components. This is a proposed testing and investigation platform for our *Forensic Lucid* language.

[2]Run-time Integrated Programming Environment, implemented in `gipsy.RIPE`

[3]General Intensional Programming Compiler, implemented in `gipsy.GIPC`

[4]General Eduction Engine, implemented in `gipsy.GEE`

2 Forensic Lucid Requirements and Design

This section lists concepts and considerations in the design of the Forensic Lucid language. The language has been studied through a case study [MPD07, MP08a] and another one is under development [MP08b]. The end goal is to define our Forensic Lucid language where its constructs concisely express cyberforensic evidence, which can be initial state of the case (e.g. initial printer state when purchased from the manufacturer as in [GP04]), towards what we have actually observed as a final state (e.g. when an investigator finds the printer with two queue entries $(B_{deleted}, B_{deleted})$). The implementing system also back-traces intermediate results 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 Features

We define Forensic Lucid to model the evidential statements and other expressions representing the evidence and observations as a higher-order context. An execution trace of a Forensic Lucid program would expose the possibility of the proposed claim with the events in the middle between the final observed event to the beginning of the events. Forensic Lucid aggregates the features of multiple Lucid dialects mentioned earlier needed for these tasks along with its own extensions.

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, 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 syntax, and 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. The most difficult aspect here is, of course, the semantics of Forensic Lucid (luckily, the bulk of it is an aggregation of the semantic rules of the languages we inherit from).

2.2 Forward Tracing vs. Back-tracing

Naturally, the GEE makes demands in the demand-driven evaluation in the order the tree of an intentional program is traversed. Tracing of the demand requests in this case will be “forward tracing”. Such tracing is less useful than the mentioned back-tracing when demands are resolved, when dealing with the back-tracing in forensic investigation in an attempt to reconstruct events from the final state observations. Back-tracing is also naturally present when demands are computed and return results. The latter may not be sufficient in the forensic evaluation, so a set of reverse operators to `next`, `fby`, `asa`, etc. is needed. The development of such operators is discussed further in the syntax and semantics sections.

2.3 Context

We need to provide an ability to encode the stories told by the evidence and witnesses. This will constitute the context of evaluation. The return value of the evaluation would be a collection of backtraces, which contain the "paths of truth". If a given trace contains all truths values, it's an explanation of a story. If there is no such a path, i.e. the trace, there is no enough supporting evidence of the entire claim to be true.

The context for this task for simplicity of the prototype language can be expressed as integers or strings, to which we attribute some meaning or description. The contexts are finite and can be navigated through in both directions of the index, potentially allowing negative tags in our tag sets of dimensions. Alternatively, our contexts can be a finite set of symbolic labels and their values that can internally be enumerated. This approach will be naturally more appropriate for humans and we have a machinery to so in Lucx's implementation in GIPSY [Ton08, PMT08].

We define streams of observations as our context, that can be a simple context or a context set. In fact, in Forensic Lucid we are defining higher-level dimensions and lower-level dimensions. The highest-level one is the *evidential statement*, which is a finite unordered set of observation sequences. The *observation sequence* is a finite *ordered* set of observations. The *observation* is an "eyewitness" of a particular property along with the duration of the observation. As in the FSA [Gla05, GP04], the observations are a tuples of (P, min, opt) in their generic form. The observations in this form, specifically, the property P can be exploded further into Lucx's context set and further into an atomic simple context [Wan06, TPM07]. Context switching between different observations is done naturally with the Lucid @ context switching operator. Consider some conceptual expression of a storyboard in Listing 1 where anything in `[ ... ]` represents a story, i.e. the context of evaluation. `foo` can be evaluated at multiple contexts (stories), producing a collection of final results (e.g. *true* or *false*) for each story as well as a collection of traces.

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

Listing 1: Intensional Storyboard Expression

While the `[...]` notation here may be confusing with respect to `[dimension:tag]` in

Lucid and more specifically in Lucx [Wan06, TPM07], 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 [Wan06, TPM07] except with the syntactical sugar mentioned earlier where we allow syntactical grouping of properties, observations, observation sequences, and evidential statements as our context sets.

The generic observation sequence can be expanded [GP04] into the context stream using the *min* and *opt* values, where they will translate into index values. Thus, $obs = (A,3,0)(B,2,0)$ expands the property labels A and B into a finite stream of five indexed elements: $AAABB$. Thus, a Forensic Lucid fragment in Listing 2 would return the third A of the $AAABB$ context stream in the observation portion of o. Therefore, possible evaluations to check for the properties can be as shown in Figure 1.

```
// Give me observed property at index 2 in the observation sequence obs
o @.obs 2
where
  // Higher-level dimension in the form of (P,min,opt)
  observation o;
  // Equivalent to writing = { A, A, A, B, B };
  observation sequence obs = (A,3,0)(B,2,0);
  where
    // Properties A and B are arrays of computations
    // or any Expressions
    A = [c1,c2,c3,c4];
    B = E;
    ...
  end;
end;
```

Listing 2: Observation Sequence With Duration

The property values of A and B can be anything that context calculus allows. The dimension type `observation sequence` is a finite ordered context tag set [PMT08] that allows an integral "duration" of a given tag property. This may seem like we allow duplicate tag values that are unsound in the classical Lucid semantics; however, we find our way around little further in the text with the implicit index tag. The semantics of the arrays of computations is not a part of either GIPL or Lucx; however, the arrays are provided by JLucid and Objective Lucid. We need the notion of the arrays to evaluate multiple computations at the same context. Having an array of computations is conceptually equivalent of running an a Lucid program under the same context for each array element in a separate instance of the evaluation engine and then the results of those expressions are gathered in one ordered storage within the originating program. Arrays in Forensic Lucid are needed to represent a set of results, or *explanations* of evidential statements, as well as denote some properties of observations. We will explore the notion of arrays in Forensic Lucid much greater detail in the near future work. In the FSA approach computations c_i correspond to the state q and event i that enable transition. For Forensic Lucid, we can have c_i as theoretically any Lucid expression E.

In Figure 1 we are illustrating a possibility to query for the sub-dimension indices by raw

```
Observed property (context): A A A B B
       Sub-dimension index: 0 1 2 3 4

o @.obs 0 = A
o @.obs 1 = A
o @.obs 2 = A
o @.obs 3 = B
o @.obs 4 = B

To get the duration/index position:

o @.obs A = 0 1 2
o @.obs B = 3 4
```

Figure 1: Handling Duration of an Observed Property in the Context

property where it persists that produces a finite stream valid indices that can be used in subsequent expressions, or, alternatively by supplying the index we can get the corresponding raw property at that index. The latter feature is still under investigation of whether it is safe to expose it to Forensic Lucid programmers or make it implicit at all times at the implementation level. This is needed to remedy the problem of "duplicate tags": as previously mentioned, observations form the context and allow durations. This means multiple duplicate dimension tags with implied subdimension indexes should be allowed as the semantics of a traditional Lucid approaches do not allow duplicate dimension tags. It should be noted however, that the combination of the tag and its index in the stream is still unique and can be folded into the traditional Lucid semantics.

2.4 Concrete Forensic Lucid Syntax

The concrete syntax of the Forensic Lucid language is presented in Figure 2. It is influenced by the productions from Lucx [WAP05b, Wan06], JLucid and Objective Lucid [Mok05, GMP05, MP05], and Indexical Lucid [Paq99]. Some of the syntactical definitions can be, perhaps, implemented as a collection of macros. The `evidential statement`, `observation sequence`, and `observation` dimension types can be translated into `dimension` by some translation rules flattening them into simple contexts and context sets. The GIPSY compiler framework (GIPC) allows for the introduction of such semantic translation rules to define new language variants. We will use this feature as much as possible, though some of our syntactic constructs may have some underlying semantic details that cannot be translated into generic Lucid primitives, in which case we need to expand the existing semantics.

```
(01)                 E  ::=   id
(02)                    |    E(E,...,E)                      #LUCX
(03)                    |    E[E,...,E](E,...,E)             #GIPL
(04)                    |    if E then E else E fi
(05)                    |    # E
(06)                    |    E @ E  E                        #GIPL
(07)                    |    E @ E                           #LUCX
(08)                    |    E where Q end;
(09)                    |    [E:E,...,E:E]                   #LUCX
(10)                    |    E bin-op E                      #INDEXICAL
(11)                    |    un-op E                         #INDEXICAL
(12)                    |    E i-bin-op E                    #INDEXICAL
(13)                    |    i-un-op E                       #INDEXICAL
(14)                    |    bounds
(15)                    |    embed(URI, METHOD, E, E, ...)   #JLUCID
(16)                    |    E[E,...,E]                      #JLUCID
(17)                    |    [E,...,E]                       #JLUCID
(18)                    |    E.id                            #OBJECTIVE
(19)                    |    E.id(E,...,E)                   #OBJECTIVE

(20)                 Q  ::=   dimension id,...,id;
(21)                    |    evidential statement id,...,id [ = ES ];
(22)                    |    observation sequence id,...,id [ = OS ];
(23)                    |    observation id,...,id [ = O ];
(24)                    |    id = E;
(25)                    |    id(id,....,id) = E;             #LUCX
(26)                    |    id[id,...,id](id,....,id) = E;  #GIPL
(27)                    |    E.id = E;                       #OBJECTIVE
(28)                    |    id.id,...,id(id,...,id) = E;    #OBJECTIVE
(29)                    |    QQ

(30)                 ES ::=   { OS,...,OS } # evidential statement
(31)                 OS ::=   { O,...,O }   # observation sequence
(32)                  O ::=   ( E, E, E )   # (property, min, opt)
                        |    $             # no-observation (Ct, 0, infinitum)
                        |    \0( E )       # zero-observation (P, 0, 0), where P = E

(33)             bin-op ::= arith-op | logical-op | bitwise-op
(34)              un-op ::= + | -

(35)           arith-op ::=  +  | - | *  | /  | %  | ^
(36)         logical-op ::=  <  | > | >= | <= | == | in | && | "||" | !
(37)         bitwise-op ::= "|" | & | ~  | !| | !&

(38)           i-bin-op ::= @ | i-bin-op-forw | i-bin-op-back | i-logic-bitwise-op | i-forensic-op

(39)      i-bin-op-forw ::= fby   |  upon  | asa   | wvr
                        |  nfby  | nupon  | nasa  | nwvr

(40)      i-bin-op-back ::= pby   | rupon  | ala   | rwvr
                        |  npby  | nrupon | nala  | nrwvr

(41) i-logic-bitwise-op ::= and   | or     | xor
                        |  nand  | nor    | nxor
                        |  band  | bor    | bxor

(42)            i-un-op ::= i-bin-un-forw | i-bin-un-back | #

(43)      i-bin-un-forw ::= first  | next  | iseod
                        |  second | nnext | neg | not

(44)      i-bin-un-back ::= last    | prev  | isbod
                        |  prelast | nprev

(45)      i-forensic-op ::= combine | product | psi | invpsi

(46)             bounds ::= eod | bod | +inf | -inf
```

Figure 2: Concrete Forensic Lucid Syntax

2.5 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 ψ 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). 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.

A question arises a of how to explicitly model the transition function ψ and its backtrace Ψ^{-1} in the new language. A possible approach is to use predefined macros in Lucid syntax [MPD07]. 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. Once modeled, it would be the GEE actually execution ψ within GIPSY. In fact, the intensional operators of Lucid represent the basic building blocks for ψ and Ψ^{-1}. We provide a first implementation of Ψ^{-1} in [MPD07].

```
alice_claim @ es
where
  evidential statement es = [ printer, manuf, alice ];

  observation sequence printer = F;
  observation sequence manuf = [Oempty, $];
  observation sequence alice = [Oalice, F];

  observation F = (``B_deleted'', 1, 0);
  observation Oalice = (P_alice, 0, +inf);
  observation Oempty = (``empty'', 1, 0);

  // No ``add_A''
  P_alice = unordered {``add_B'', ``take''};

  invpsiacme(F, es);
end;
```

Listing 3: Developing the Pinter Case “main”

2.6 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 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 have been defined so far in [MP08a], we only summarize their definitions through the `@` and `#` operators in Figure 3.

2.7 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.

- **`combine`** corresponds to the *comb* function described earlier. It is defined in Listing 4. 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);
```

Listing 4: The **`combine`** Operator

- **`product`** tentatively corresponds to the cross-product of context, translated from that of the LISP example and added with context. It is defined in Listing 5.

```
/**
 * 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)
```

Listing 5: The **`product`** Operator

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.

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

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

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

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

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

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

$$\begin{aligned} X\ \mathtt{wvr}\ Y = {} & X@T\ \mathtt{where} \\ & \quad T = U\ \mathtt{fby}\ U@(T+1) \\ & \quad U = \textbf{if}\ Y\ \textbf{then}\ \#\ \textbf{else}\ \mathtt{next}\ U \\ & \mathtt{end} \end{aligned} \tag{7}$$

$$\begin{aligned} X\ \mathtt{rwvr}\ Y = {} & X@T\ \mathtt{where} \\ & \quad T = U\ \mathtt{pby}\ U@(T-1) \\ & \quad U = \textbf{if}\ Y\ \textbf{then}\ \#\ \textbf{else}\ \mathtt{prev}\ U \\ & \mathtt{end} \end{aligned} \tag{8}$$

$$\begin{aligned} X\ \mathtt{nwvr}\ Y = {} & X@T\ \mathtt{where} \\ & \quad T = U\ \mathtt{fby}\ U@(T+1) \\ & \quad U = \textbf{if}\ Y == 0\ \textbf{then}\ \#\ \textbf{else}\ \mathtt{next}\ U \\ & \mathtt{end} \end{aligned} \tag{9}$$

$$\begin{aligned} X\ \mathtt{rnwvr}\ Y = {} & X@T\ \mathtt{where} \\ & \quad T = U\ \mathtt{pby}\ U@(T-1) \\ & \quad U = \textbf{if}\ Y == 0\ \textbf{then}\ \#\ \textbf{else}\ \mathtt{prev}\ U \\ & \mathtt{end} \end{aligned} \tag{10}$$

$$X\ \mathtt{asa}\ Y = \mathtt{first}\ (X\ \mathtt{wvr}\ Y) \tag{11}$$

$$X\ \mathtt{nasa}\ Y = \mathtt{first}\ (X\ \mathtt{nwvr}\ Y) \tag{12}$$

$$X\ \mathtt{ala}\ Y = \mathtt{last}\ (X\ \mathtt{rwvr}\ Y) \tag{13}$$

$$X\ \mathtt{nala}\ Y = \mathtt{last}\ (X\ \mathtt{nrwvr}\ Y) \tag{14}$$

$$\begin{aligned} X\ \mathtt{upon}\ Y = {} & X@W\ \mathtt{where} \\ & \quad W = 0\ \mathtt{fby}\ (\textbf{if}\ Y\ \textbf{then}\ (W+1)\ \textbf{else}\ W) \\ & \mathtt{end} \end{aligned} \tag{15}$$

$$\begin{aligned} X\ \mathtt{rupon}\ Y = {} & X@W\ \mathtt{where} \\ & \quad W = 0\ \mathtt{pby}\ (\textbf{if}\ Y\ \textbf{then}\ (W-1)\ \textbf{else}\ W) \\ & \mathtt{end} \end{aligned} \tag{16}$$

$$\begin{aligned} X\ \mathtt{nupon}\ Y = {} & X@W\ \mathtt{where} \\ & \quad W = 0\ \mathtt{fby}\ (\textbf{if}\ Y == 0\ \textbf{then}\ (W+1)\ \textbf{else}\ W) \\ & \mathtt{end} \end{aligned} \tag{17}$$

$$\begin{aligned} X\ \mathtt{nrupon}\ Y = {} & X@W\ \mathtt{where} \\ & \quad W = 0\ \mathtt{pby}\ (\textbf{if}\ Y == 0\ \textbf{then}\ (W-1)\ \textbf{else}\ W) \\ & \mathtt{end} \end{aligned} \tag{18}$$

$$\mathtt{neg}\ X = -X \tag{19}$$

$$\mathtt{not}\ X = \textbf{if}\ X\ \textbf{then}\ !X\ \textbf{else}\ X \tag{20}$$

$$X\ \mathtt{and}\ Y = X \&\& Y \tag{21}$$

$$X\ \mathtt{or}\ Y = X || Y \tag{22}$$

$$X\ \mathtt{xor}\ Y = \mathtt{not}((X\ \mathtt{and}\ Y)\ \mathtt{or}\ \mathtt{not}\ (X\ \mathtt{or}\ Y)) \tag{23}$$

Figure 3: Operators Translated to GIPL-Compatible Definitions

2.8 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 [Paq99], Objective Lucid [Mok05], and Lucx [Wan06] are in Figure 4, Figure 5, and Figure 7 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. The Objective Lucid semantic rules were affected and refined by some of the semantic rules of JOOIP [WPM08]. The new rules of the operational semantics of Forensic Lucid cover the 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 [Paq99]. SIPLs are Lucid dialects (Specific Intensional Programming Languages) with their own attributes and objectives. Theoretically, all SIPLs can be translated into the GIPL [Paq99]. 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 4. The excerpt of semantic rules of Lucx are then presented as a conservative extension to GIPL and Lucx in Figure 7. Following is the description of the GIPL semantic rules as presented in [Paq99]:

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

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 4. It is therefore a partial function

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

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 `(dim)`. 2. *Constants* are external entities that provide a single value, regardless of the context of evaluation.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 4: GIPL Semantics

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 $(\texttt{op}, f)$,

$$\mathbf{E_{c-vid}} : \frac{\mathcal{D},\mathcal{P} \vdash E : id \quad \mathcal{D},\mathcal{P} \vdash E' : id' \quad \mathcal{D}(id) = (\texttt{class},\ \texttt{cid},\ \underline{\texttt{cdef}}) \quad \mathcal{D}(id') = (\texttt{classv},\ \texttt{cid.cvid},\ \underline{\texttt{vdef}}) \quad \mathcal{D},\mathcal{P} \vdash <\texttt{cid.cvid}>: v}{\mathcal{D},\mathcal{P} \vdash E.E' : v} \tag{40}$$

$$\mathbf{E_{c-fct}} : \frac{\mathcal{D},\mathcal{P} \vdash E : id \quad \mathcal{D},\mathcal{P} \vdash E' : id' \quad \mathcal{D},\mathcal{P} \vdash E_1,\ldots,E_n : v_1,\ldots,v_n \quad \mathcal{D}(id) = (\texttt{class},\ \texttt{cid},\ \underline{\texttt{cdef}}) \quad \mathcal{D}(id') = (\texttt{classf},\ \texttt{cid.cfid},\ \underline{\texttt{fdef}}) \quad \mathcal{D},\mathcal{P} \vdash <\texttt{cid.cfid}(v_1,\ldots,v_n)>: v}{\mathcal{D},\mathcal{P} \vdash E.E'(E_1,\ldots,E_n) : v} \tag{41}$$

$$\mathbf{E_{ffid}} : \frac{\mathcal{D},\mathcal{P} \vdash E : id \quad \mathcal{D},\mathcal{P} \vdash E_1,\ldots,E_n : v_1,\ldots,v_n \quad \mathcal{D}(id) = (\texttt{freefun},\ \texttt{ffid},\ \underline{\texttt{ffdef}}) \quad \mathcal{D},\mathcal{P} \vdash <\texttt{ffid}(v_1,\ldots,v_n)>: v}{\mathcal{D},\mathcal{P} \vdash E(E_1,\ldots,E_n) : v} \tag{42}$$

$$\mathbf{\#JAVA_{objid}} : \frac{\underline{\texttt{cdef}} = \texttt{Class cid } \{\ldots\}}{\mathcal{D},\mathcal{P} \vdash \underline{\texttt{cdef}} : \mathcal{D}\dagger[\texttt{cid} \mapsto (\texttt{class, cid, } \underline{\texttt{cdef}})],\ \mathcal{P}} \tag{43}$$

$$\mathbf{\#JAVA_{objvid}} : \frac{\underline{\texttt{cdef}} = \texttt{Class cid } \{\ldots\underline{\texttt{vdef}}\ldots\} \quad \underline{\texttt{vdef}} = \texttt{public } type \texttt{ vid;}}{\mathcal{D},\mathcal{P} \vdash \underline{\texttt{cdef}} : \mathcal{D}\dagger[\texttt{cid.vid} \mapsto (\texttt{classv, cid.vid, } \underline{\texttt{vdef}})],\mathcal{P}} \tag{44}$$

$$\mathbf{\#JAVA_{objfid}} : \frac{\underline{\texttt{cdef}} = \texttt{Class cid } \{\ldots\underline{\texttt{fdef}}\ldots\} \quad \underline{\texttt{fdef}} = \texttt{public} frttype\ \texttt{fid}(fargtype_1\ farg_{id_1},\ldots,fargtype_n\ farg_{id_n})}{\mathcal{D},\mathcal{P} \vdash \underline{\texttt{cdef}} : \mathcal{D}\dagger[\texttt{cid.fid} \mapsto (\texttt{classf},\ \texttt{cid.fid},\ \underline{\texttt{fdef}})],\mathcal{P}} \tag{45}$$

$$\mathbf{\#JAVA_{ffid}} : \frac{\underline{\texttt{ffdef}} = frttype\ \texttt{ffid}(fargtype_1\ farg_{id_1},\ldots,fargtype_n\ farg_{id_n})}{\mathcal{D},\mathcal{P} \vdash \underline{\texttt{ffdef}} : \mathcal{D}\dagger[\texttt{ffid} \mapsto (\texttt{freefun},\ \texttt{ffid},\ \underline{\texttt{ffdef}})],\mathcal{P}} \tag{46}$$

Figure 5: Extract of Operational Semantics of Objective Lucid

$$\mathbf{E_{E.did}} : \frac{\mathcal{D}(E.id) = (\texttt{dim})}{\mathcal{D},\mathcal{P} \vdash E.id : id.id} \tag{47}$$

Figure 6: Higher-Order Context Dot Operator of MARFL

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.

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

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

Each type of identifiers can only be used in the appropriate situations. Identifiers of type `op`, `func`, and `dim` evaluate to themselves (Figure 4, rules 25,26,27). Constant

$$\mathbf{E}_{\#(\mathbf{cxt})} \quad : \quad \frac{}{\mathscr{D},\mathscr{P} \vdash \# : \mathscr{P}} \tag{48}$$

$$\mathbf{E}_{\mathbf{construction(cxt)}} \quad : \quad \frac{\begin{array}{ll}\mathscr{D},\mathscr{P} \vdash E_{d_j} : id_j & \mathscr{D}(id_j) = (\texttt{dim}) \\ \mathscr{D},\mathscr{P} \vdash E_{i_j} : v_j & \mathscr{P}' = \mathscr{P}_0 \dagger [id_1 \mapsto v_1] \dagger \ldots \dagger [id_n \mapsto v_n]\end{array}}{\mathscr{D},\mathscr{P} \vdash [E_{d_1} : E_{i_1}, E_{d_2} : E_{i_2}, \ldots, E_{d_n} : E_{i_n}] : \mathscr{P}'} \tag{49}$$

$$\mathbf{E}_{\mathbf{at(cxt)}} \quad : \quad \frac{\mathscr{D},\mathscr{P} \vdash E' : \mathscr{P}' \qquad \mathscr{D},\mathscr{P} \dagger \mathscr{P}' \vdash E : v}{\mathscr{D},\mathscr{P} \vdash E \;@\; E' : v} \tag{50}$$

$$\mathbf{E}_{.} \quad : \quad \frac{\mathscr{D},\mathscr{P} \vdash E_2 : id_2 \qquad \mathscr{D}(id_2) = (\texttt{dim})}{\mathscr{D},\mathscr{P} \vdash E_1.E_2 : tag(E_1 \downarrow \{id_2\})} \tag{51}$$

$$\mathbf{E}_{\mathbf{tuple}} \quad : \quad \frac{\mathscr{D},\mathscr{P} \vdash E : id \qquad \mathscr{D} \dagger [id \mapsto (\texttt{dim})] \qquad \mathscr{P} \dagger [id \mapsto 0] \qquad \mathscr{D},\mathscr{P} \vdash E_i : v_i}{\mathscr{D},\mathscr{P} \vdash \langle E_1, E_2, \ldots, E_n \rangle E : v_1 \; \mathit{fby.id} \; v_2 \; \mathit{fby.id} \; \ldots \; v_n \; \mathit{fby.id} \; \texttt{eod}} \tag{52}$$

$$\mathbf{E}_{\mathbf{select}} \quad : \quad \frac{E = [\texttt{d} : \texttt{v}'] \qquad E' = \langle \texttt{E}_1, \ldots, \texttt{E}_n \rangle \texttt{d} \mathscr{P}' = \mathscr{P} \dagger [d \mapsto v'] \qquad \mathscr{D},\mathscr{P}' \vdash E' : v}{\mathscr{D},\mathscr{P} \vdash select(E, E') : v} \tag{53}$$

$$\mathbf{E}_{\mathbf{at(s)}} \quad : \quad \frac{\mathscr{D},\mathscr{P} \vdash \mathscr{C} : \{\mathscr{P}_1, \ldots, \mathscr{P}_2\} \qquad \mathscr{D},\mathscr{P}_{i:1\ldots m} \vdash E : v_i}{\mathscr{D},\mathscr{P} \vdash E \;@C : \{v_1, \ldots, v_m\}} \tag{54}$$

$$\mathbf{C}_{\mathbf{box}} \quad : \quad \frac{\begin{array}{ll}\mathscr{D},\mathscr{P} \vdash E_{d_i} : id_i & \mathscr{D}(id_i) = (\texttt{dim}) \\ \{E_1, \ldots, E_n\} = dim(\mathscr{P}_1) = \ldots = dim(\mathscr{P}_m) & \\ E' = \texttt{f}_p(\texttt{tag}(\mathscr{P}_1), \ldots, \texttt{tag}(\mathscr{P}_m)) & \mathscr{D},\mathscr{P} \vdash E' : true\end{array}}{\mathscr{D},\mathscr{P} \vdash Box[E_1, \ldots, E_n | E'] : \{\mathscr{P}_1, \ldots, \mathscr{P}_m\}} \tag{55}$$

$$\mathbf{C}_{\mathbf{set}} \quad : \quad \frac{\mathscr{D},\mathscr{P} \vdash E_{w:1\ldots m} : \mathscr{P}_m}{\mathscr{D},\mathscr{P} \vdash \{E_1, \ldots, E_m\} : \{\mathscr{P}_1, \ldots, \mathscr{P}_w\}} \tag{56}$$

$$\mathbf{C}_{\mathbf{op}} \quad : \quad \frac{\mathscr{D},\mathscr{P} \vdash E : id \qquad \mathscr{D}(id) = (\texttt{cop}, f) \qquad \mathscr{D},\mathscr{P} \vdash C_i : v_i}{\mathscr{D},\mathscr{P} \vdash E(C_1, \ldots, C_n) : f(v_1, \ldots, v_n)} \tag{57}$$

$$\mathbf{C}_{\mathbf{sop}} \quad : \quad \frac{\mathscr{D},\mathscr{P} \vdash E : id \qquad \mathscr{D}(id) = (\texttt{sop}, f) \qquad \mathscr{D},\mathscr{P} \vdash C_i : \{v_{i_1}, \ldots, v_{i_k}\}}{\mathscr{D},\mathscr{P} \vdash E(C_1, \ldots, C_n) : f(\{v_{1_1}, \ldots, v_{1_s}\}, \ldots, \{v_{n_1}, \ldots, v_{n_m}\})} \tag{58}$$

Figure 7: Extract of Operational Semantics of Lucx

identifiers (`const`) evaluate to the corresponding constant (Figure 4, rule 24). Function calls, resolved by the $\mathbf{E_{fct}}$ rule (Figure 4, 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 4, 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 4, 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 4, 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 4, 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 7. In Lucx, semantic rule 49 (Figure 7) creates a context as a semantic item and returns it as a context $\mathscr{P}$ that can then be used by rule 50 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 48 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.

3 Conclusion

Through the series of discussions, definitions of the syntax, semantics, and some examples of the Forensic Lucid we believe we are on the right track to show the benefits of the intensional approach to the cyberforensic, which is promising to be more usable and can be improved even further by providing a graphical DFG editor for the investigators. As far as implementing system concerned it has advantages of parallelizing the computation and introduce the notion of context that is absent in the FSA approach of Gladyshev et al. [Gla05, GP04]. We took a lot of advantages of the existing concepts, syntax and semantic rules and constructs from the intensional programming and the Lucid family of languages that are in place, whose theory for the most part are said to be correct. We are gearing towards completion of the design of the Forensic Lucid language and its compiler and run-time environment based on the GIPSY system and its multi-tier architecture [Paq08].

The proposed practical approach in the cyberforensics field can also be used in a normal investigation process involving crimes not necessarily associated with information technology.

3.1 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.
- Implementation of the Forensic Lucid compiler, run-time and interactive development environments.

4 Acknowledgments

This work was funded in part by NSERC and the Faculty of Engineering and Computer Science, Concordia University. Thanks to many of the GIPSY project team members for their valuable contributions, suggestions, and reviews, including Dr. Peter Grogono, Xin Tong, Aihua Wu, Emil Vassev, and Amir Pourteymour.

References

[Agi95] I. Agi. GLU for multidimensional signal processing. In *ISLIP'95: The Eighth International Symposium on Languages for Intensional Programming, Sydney, Australia*, 1995.

[AW76] Edward A. Ashcroft and William W. Wadge. Lucid - A Formal System for Writing and Proving Programs. volume 5. SIAM J. Comput. no. 3, 1976.

[AW77a] Edward A. Ashcroft and William W. Wadge. Erratum: Lucid - A Formal System for Writing and Proving Programs. volume 6(1):200. SIAM J. Comput., 1977.

[AW77b] Edward A. Ashcroft and William W. Wadge. Lucid, a nonprocedural language with iteration. *Communication of the ACM*, 20(7):519–526, July 1977.

[Du94] Weichang Du. Object-oriented Implementation of Intensional Language. In *Proceedings of the 7th International Symposium on Lucid and Intensional Programming*, pages 37–45. SRI International, Menlo Park, California, USA, September 1994.

[Edw95] Edward Ashcroft and Anthony Faustini and Raganswamy Jagannathan and William Wadge. *Multidimensional, Declarative Programming*. Oxford University Press, London, 1995.

[FB91] B. Freeman-Benson. Lobjcid: Objects in Lucid. In *Proceedings of the 1991 Symposium on Lucid and Intensional Programming*, pages 80–87. SRI International, Menlo Park, California, USA, April 1991.

[Gla05] Pavel Gladyshev. 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.

[GMP05] Peter Grogono, Serguei Mokhov, and Joey Paquet. Towards JLucid, Lucid with Embedded Java Functions in the GIPSY. In *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005), Las Vegas, USA*, pages 15–21. CSREA Press, June 2005.

[GP99] Jean-Raymond Gagné and John Plaice. Demand-Driven Real-Time Computing. World Scientific, September 1999.

[GP04] Pavel Gladyshev and Ahmed Patel. Finite State Machine Approach to Digital Event Reconstruction. In *Digital Investigation Journal*, volume 2, 2004.

[Lu04] Bo Lu. *Developing the Distributed Component of a Framework for Processing Intensional Programming Languages*. PhD thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, March 2004.

[Mok04] Serguei A. Mokhov. Lucid, the Intensional Programming Language and its Semantics in PVS. Semantics of Programming Languages Course Project Report, April 2004.

[Mok05] Serguei A. Mokhov. Towards Hybrid Intensional Programming with JLucid, Objective Lucid, and General Imperative Compiler Framework in the GIPSY. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, October 2005. ISBN 0494102934.

[Mok07a] Serguei Mokhov. *Intensional Cyberforensics – a PhD Proposal*. Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, December 2007.

[Mok07b] Serguei Mokhov. 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.

[Mok08] Serguei A. Mokhov. 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)*, pages 1288–1294, Turku, Finland, July 2008. IEEE Computer Society.

[MP05] Serguei Mokhov and Joey Paquet. Objective Lucid – First Step in Object-Oriented Intensional Programming in the GIPSY. In *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005), Las Vegas, USA*, pages 22–28. CSREA Press, June 2005.

[MP08a] Serguei A. Mokhov and Joey Paquet. Formally Specifying and Proving Operational Aspects of Forensic Lucid in Isabelle. Technical Report 2008-1-Ait Mohamed, Department of Electrical and Computer Engineering, Concordia University, August 2008. In Theorem Proving in Higher Order Logics (TPHOLs2008): Emerging Trends Proceedings.

[MP08b] Serguei A. Mokhov and Joey Paquet. Using the General Intensional Programming System (GIPSY) for Evaluation of Higher-Order Intensional Logic (HOIL) Expressions. Submitted for publication at SAC'09, 2008.

[MPD07] Serguei A. Mokhov, Joey Paquet, and Mourad Debbabi. Designing a Language for Intensional Cyberforensic Analysis. Unpublished, 2007.

[Paq99] Joey Paquet. *Scientific Intensional Programming*. PhD thesis, Department of Computer Science, Laval University, Sainte-Foy, Canada, 1999.

[Paq08] Joey Paquet. A Multi-Tier Architecture for the Distributed Eductive Execution of Hybrid Intensional Programs. Submitted for publication at SAC'09, 2008.

[PGW04] Joey Paquet, Peter Grogono, and Ai Hua Wu. Towards a Framework for the General Intensional Programming Compiler in the GIPSY. In *Proceedings of the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004). Vancouver, Canada.* ACM, October 2004.

[PK00] Joey Paquet and Peter Kropf. The GIPSY Architecture. In *Proceedings of Distributed Computing on the Web, Quebec City, Canada*, 2000.

[PMT08] Joey Paquet, Serguei A. Mokhov, and Xin Tong. Design and Implementation of Context Calculus in the GIPSY Environment. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*, pages 1278–1283, Turku, Finland, July 2008. IEEE Computer Society.

[PN07] Lawrence C. Paulson and Tobias Nipkow. Isabelle: A Generic Proof Assistant. University of Cambridge and Technical University of Munich, 2007. `http://isabelle.in.tum.de/`, last viewed: December 2007.

[PW05] Joey Paquet and Ai Hua Wu. GIPSY – A Platform for the Investigation on Intensional Programming Languages. In *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005), Las Vegas, USA*, pages 8–14. CSREA Press, June 2005.

[Ron94] Panagiotis Rondogiannis. *Higher-Order Functional Languages and Intensional Logic*. PhD thesis, Department of Computer Science, University of Victoria, Victoria, Canada, 1994.

[SW00] Paul Swoboda and William W. Wadge. Vmake, ISE, and IRCS: General Tools for the Intensionalization of Software Systems. In M. Gergatsoulis and P. Rondogiannis, editors, *Intensional Programming II*. World-Scientific, 2000.

[Swo04] Paul Swoboda. *A Formalisation and Implementation of Distributed Intensional Programming*. PhD thesis, The University of New South Wales, Sydney, Australia, 2004.

[The08] 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.

[Ton08] Xin Tong. 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.

[TPM07] Xin Tong, Joey Paquet, and Serguei A. Mokhov. Context Calculus in the GIPSY. Unpublished, 2007.

[VP05] Emil Vassev and Joey Paquet. A Generic Framework for Migrating Demands in the GIPSY's Demand-Driven Execution Engine. In *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005), Las Vegas, USA*, pages 29–35. CSREA Press, June 2005.

[WA85] William Wadge and Edward Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, London, 1985.

[Wan06] Kaiyu Wan. *Lucx: Lucid Enriched with Context*. PhD thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2006.

[WAP05a] Kaiyu Wan, Vasu Alagar, and Joey Paquet. A Context theory for Intensional Programming. In *Workshop on Context Representation and Reasoning (CRR05), Paris, France*, July 2005.

[WAP05b] Kaiyu Wan, Vasu Alagar, and Joey Paquet. Lucx: Lucid Enriched with Context. In *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005), Las Vegas, USA*, pages 48–14. CSREA Press, June 2005.

[WP05] Ai Hua Wu and Joey Paquet. Object-Oriented Intensional Programming in the GIPSY: Preliminary Investigations. In *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005), Las Vegas, USA*, pages 43–47. CSREA Press, June 2005.

[WPM08] Aihua Wu, Joey Paquet, and Serguei A. Mokhov. Object-Oriented Intensional Programming: Intensional Classes Using Java and Lucid. Submitted for publication to SAC'09, 2008.

[Zha97] Q. Zhao. Implementation of an object-oriented intensional programming system. Master's thesis, Department of Computer Science, University of New Brunswick, Canada, 1997.