Lucx: Lucid Enriched With Context

Kaiyu Wan, Vasu Alagar, Joey Paquet
Department of Computer Science and Software Engineering
Concordia University
Montreal, Quebec H3G 1M8, Canada
E-mail: \{ky_wan, alagar, paquet@cse.concordia.ca\}

Abstract
In this paper, we give an overview of our current work on introducing context as first-class objects in Lucid. The use of contexts as first-class values increases the expressive power of Lucx (Lucid enriched with context). It allows us to use the paradigm for agent communication, for real-time reactive programming and for constraint programming. We include a discussion on context calculus, representation of context aggregations, the syntax and semantic rules of Lucx. The implementation of Lucx in GIPSY, a platform under development for compiling Lucid family of languages, is also given.

Keywords: Intensional Programming, Lucid, Context, Agent Communication, Real-time Reactive Programming

1. Introduction

Context is a rich concept and is hard to define. The meaning of “context” is tacitly understood and used by researchers in diverse disciplines. The notion of context was introduced by McCarthy and later used by Guha [5] as a means of expressing assumptions made by natural language expressions in Artificial Intelligence (AI). Hence, a formula, which is an expression combining a sentence in AI with contexts, can express the exact meaning of the natural language expression. Intensional logic [8] is a branch of mathematical logic which is used to describe precisely context-dependent entities. In Intensional Programming (IP) paradigm, which has its foundations in Intensional Logic, the real meaning of an expression, called intension, is a function from contexts to values, and the value of the intension at any particular context, called the extension, is obtained by applying context operators to the intension. The major distinction between contexts in AI and in intensional programming language is that in the former case they are rich objects that are not completely expressible and in the later case they are implicitly expressible.

Lucid was a dataflow language [12] and evolved into a Multidimensional Intensional Programming Language [1]. It is possible to write an expression in Lucid whose evaluation is context-dependent. However, a context can not be explicitly declared and manipulated in Lucid. This restricts the ability of Lucid to express many requirements and constraints that arise in programming a complex software system. So we have extended Lucid by adding the capability to explicitly manipulate contexts. This is achieved by introducing context as a first-class object in the language. From now on, we call this extended language as Lucx (Lucid extended with contexts).

2. Syntax and Semantic Rules of Lucx

Lucx is a conservative extension of Lucid, with context becoming a first-class object in the language. This way, contexts can be manipulated, assigned values and passed as parameters dynamically. The syntax of Lucx is shown below.

\[
E ::= \text{id} \mid E(E_1, \ldots, E_n) \mid \text{if } E \text{ then } E' \text{ else } E'' \mid \# \mid E @ E' \mid [E_1 : E'_1, \ldots, E_n : E'_n] \mid E \text{ where } Q
\]

\[
Q ::= \text{dimension id} \mid id = E \mid id(id_1, \ldots, id_n) = E \mid Q Q
\]

The difference between Lucx and original Lucid is highlighted in bold in the above syntax rules. The symbols @ and # are context navigation and query operators. The non-terminals \(E\) and \(Q\) respectively refer to expressions and definitions. The abstract semantics of evaluation in Lucx is \(D, \mathcal{P} \vdash E : v\), which means that in the definition environment \(D\), and in the evaluation context \(\mathcal{P}\), expression \(E\)
evaluates to v. The definition environment D retains the definitions of all of the identifiers that appear in a Lucid program. Formally, D is a partial function D : Id → IdEntry, where Id is the set of all possible identifiers and IdEntry has five possible kinds of value such as: Dimensions, Constants, Data Operators, Variables, and Functions[8]. The evaluation context P′ is the result of P ⊢ c, where P is the initial evaluating context, c is the defined context expression, and the symbol ⊢ denotes the overriding function.

A complete operational semantics for Lucid is defined in [8]. The new semantic rules for Lucx are given below.

The evaluation rule for the navigation operator, Ef, which corresponds to the syntactic expression E@E′, evaluates E in context E′, where E′ is a context defined in Section 3.1. Semantically speaking, "#" is a nullary operator, which evaluates to the current context, and "@" is a binary operator, whose left operand is an expression and right operand is a single dimension. This is different from the projection (\cell) operator (see section 3.2), in the sense that the right operand of \cell operator should be a set of dimensions.

3. Context Calculus

In Lucx context is both finite and concrete. This is in contrast to Guha’s notion, wherein contexts are infinite, rich, and generalized objects. Not all contexts studied by Guha can be dealt within our language. However, every context that we can define in Lucx is indeed a context in Guha’s sense, but restricted to well-formed Lucx expressions.

3.1. Context

Informally a context is a reference to a multidimensional stream, making an explicit reference to the dimensions and the tags (indexes) along each dimension. The syntax for context is \{d1 : x1, …, dn : xn\}, where d1, …, dn are dimension names, and xi is the tag for dimension di. Given an expression E and a context c, the Lucid expression E ⊙ c directs the evaluation engine to evaluate E in the context c. According to the semantics, E ⊙ c gives the stream value at the coordinates referenced by c.

Formally, contexts are defined as a subset of a finite union of relations. The relations are defined over dimension and tag sets. Let DIM = \{d1, d2, …, dn\} denote a finite set of dimension names. We associate with each di ∈ DIM a unique enumerable tag set X_i. Let TAG = \{X_1, …, X_n\} denote the set of tags. There exists functions f_dimtotag : DIM → TAG, such that the function f_dimtotag associates with every di ∈ DIM exactly one tag X_i in TAG.

Definition 1 Consider the relations

\[ P_i = \{d_i\} \times f_dimtotag(d_i) \quad 1 \leq i \leq n \]

A context C, given (DIM, f_dimtotag), is a finite subset of \bigcup_{i=1}^{n} P_i. The degree of the context C is \(|\Delta|\), where \(\Delta \subset DIM\) includes the dimensions that appear in C.

A context is written using enumeration syntax. The set enumeration syntax of a context C is C = \{(d, x) | d ∈ Δ, x ∈ f_dimtotag(d)\}. We use the syntax \[d_1 : x_1, …, d_n : x_n\] in Lucx to explicitly denote the aggregation of dimension, tag pairs. Note that the d_i\'s need not be distinct, and

\[ C \subseteq \bigcup_{i=1}^{n} P_i \subseteq DIM \times I \quad I = \bigcup_{x \in TAG} X_i \]

Consequently, every subset of \bigcup_{i=1}^{n} P_i is a context, but not every subset of DIM × I is a context. However, if \(X_1 = X_2 = \ldots = X_n\), every subset of \(DIM \times I\) is a context. This follows from the fact that \(f_dimtotag(d_i) = X_i\), \(i = 1, \ldots, n\) implies that \(\bigcup_{i=1}^{n} P_i = \bigcup_{i=1}^{n} \{(d_i, x) | x \in f_dimtotag(d_i)\} = \{(\cup_{i=1}^{n} d_i), I\} = DIM \times I\). We say a context C is simple (s_context), if \(\langle d_1, x_1, \ldots, d_n, x_n \rangle \in C \Rightarrow d_i \neq d_j\). A simple context C of degree 1 is called a micro (m_context) context.

Example 1:

Let DIM = \{X, Y, Z, U\},

\[ TAG = \{\text{N}, \{\text{blue}, \text{red}\}\}, \]

\[ f_dimtotag(X) = f_dimtotag(Y) = \text{N}, \]

\[ f_dimtotag(U) = f_dimtotag(Z) = \{\text{blue}, \text{red}\} \]

define \(P_1 = X \times \text{N}\), \(P_2 = Y \times \text{N}\),

\[ P_3 = Z \times \{\text{blue}, \text{red}\}, \]

\[ P_4 = U \times \{\text{blue}, \text{red}\}. \]

then \(c_1 = \{X : 1.5, Y : \text{red}\}\) is not a context,

\(c_2 = \{Z : \text{blue}\}\) is a m-context,

\(c_3 = \{X : 3, Y : 2\}\) is a s-context of degree 2,

\(c_4 = \{X : 3, X : 4, Y : 3, Y : 2, U : \text{blue}\}\)

is a context of degree 3.

Several functions on contexts are predefined. The basic functions dim and tag are to extract the set of dimensions and their associate tag values from a set of contexts.

Definition 2 Let M denote a set of m-contexts. We define functions

\[ \text{dim}_m : M \rightarrow DIM, \quad \text{tag}_m : M \rightarrow TAG_m, \]

where \(TAG_m = \bigcup_{m \in M} \text{tag}_m(m)\), such that for \(m = \{x : y\} \in M\), \(\text{dim}_m(m) = x\), and \(\text{tag}_m(m) = y \in f_dimtotag(dim_m(m))\).
Definition 3 Let $S$ denote a set of contexts. We use functions $dim_m$ and $tag_m$ to define the functions $dim$ and $tag$ on a set of contexts.

$$dim : S → \mathbb{P} DIM \quad tag : S → \mathbb{P} TAG,$$

where $TAG = \bigcup_{m∈S} \bigcup_{m∈S} tag_m(m)$ such that for $s ∈ S$, $dim(s) = \{dim_m(m) | m ∈ s\}$, and $tag(s) = \{tag_m(m) | m ∈ s\}$.

Since we are still developing the Lucx language, the set of predefined functions is not exhaustive. Functions on contexts using functions already defined in Lucx can be introduced.

3.2. Context Operators

In our previous papers [2, 10], we have introduced the following context operators: the override $⊕$ is similar to function override; difference $\ominus$, comparison $\equiv$, conjunction $\cap$, and disjunction $\cup$ are similar to set operators; projection $↓$ and hiding $↑$ are selection operators; constructor $\exists$ is used to construct an atomic context; substitution $\Rightarrow$ accepts a finite number of contexts and nondeterministically returns one of them; undirected range $\Rightarrow$ and directed range $\rightarrow$ produce a set of contexts. The formal definitions of these operators can be found in [3].

Example 2 gives an overall example for some of those operators defined above.

Example 2

Let $c_1 = [X : 2, X : 3, Y : 4], c_2 = [X : 2, Y : 4, Z : 5]$,

$$c_3 = \{Y : 2\}, D = \{Y, Z\}.$$

Then $c_1 ⊕ c_2 = [X : 2, Y : 4, Z : 5], \quad c_1 \ominus c_2 = [X : 3]$,

$c_1 ↓ D = \{Y : 4\}, \quad c_1 \cap c_2 = [X : 2, Y : 4]$,

$c_1 \cap c_2 = [X : 2, X : 3, Y : 4, Z : 5], \quad c_2 ↓ D = [X : 2]$,

$c_2 ⇓ c_3 = \{X : 2, Y : 2, Z : 5\}, \quad [X : 2, Y : 3, Z : 5]$,

$c_2 = \{X : 2, Y : 2, Z : 5\}, \quad c_2 \cap c_3 = \emptyset$,

$c_3 \mapsto c_2 = \{X : 2, Y : 2, Z : 5\}, \quad [X : 2, Y : 3, Z : 5]$,

$c_3 ↑ D = [X : 2, Y : 3, Z : 5]$.}

In order to provide a precise meaning for a context expression, we have defined the precedence rules for all the operators in the (right column) table below (from the highest precedence to the lowest) and described a set of evaluation rules for context expressions in [11]. Parentheses will be used to override this precedence when needed. Operators having the same precedence will be applied from left to right. The formal syntax of context expressions is shown below (left column).

Example 3 The evaluation steps of the well-formed context expression $c_3 ↑ D \oplus c_1 \cap c_2$, where $c_1 = [x : 3, y : 4, z : 5]$, $c_2 = [y : 5]$, and $c_3 = [x : 5, y : 6, w : 5]$, $D = \{w\}$, are as follows:

### Syntax and Precedence

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C : = c$</td>
<td>1, ↓, 1/</td>
</tr>
<tr>
<td>$C ⊇ C$</td>
<td>2</td>
</tr>
<tr>
<td>$C \cap C$</td>
<td>3, ∩, ⊃</td>
</tr>
<tr>
<td>$C ⊕ C$</td>
<td>4, ⊕</td>
</tr>
<tr>
<td>$C \div C$</td>
<td>5, $\div$, $\rightarrow$</td>
</tr>
<tr>
<td>$C \mid D$</td>
<td>6, ⊖, ⊖</td>
</tr>
</tbody>
</table>

3.3. Box and Box Operators

A context which is not a micro context or a simple context is called a non-simple context. Context $c_1$ in Example 1 is a non-simple context. In general, a non-simple context can be naturally lifted to sets of contexts generated by operations $\ominus$ (join), $\ominus$ (interaction), and $\ominus$ (union) are defined in [2] for sets of contexts introduced by $Box$ as well as for sets of contexts generated by $C ← C$ and $C → C$.

Example 4

Let $DIM = \{X, Y, Z\}$

$$f_{dimtotag}(X) = f_{dimtotag}(Y) = f_{dimtotag}(Z) = \mathbb{N},$$

$B_1 = Box[X, Y | x, y ∈ \mathbb{N} ∧ x + y = 5],$

$B_2 = Box[Y, Z | y, z ∈ \mathbb{N} ∧ y = z^2 ∧ z ≤ 3].$

Then $B_1 = \{[X : 1, Y : 4], [X : 2, Y : 3], [X : 3, Y : 2], [X : 4, Y : 1]\}$

$B_2 = \{1, Z : 1, [Y : 4, Z : 2], [Y : 9, Z : 3]\}.$

Hence $B_1 ⊕ B_2 = Box[X, Y, Z | x + y = 5 ∧ y = z^2 ∧ z ≤ 3] = \{[X : 1, Y : 4, Z : 2], [X : 4, Y : 1, Z : 1]\}$
\[ B_1 \Box B_2 = Box[y + z = 5 \land z \leq 3] = \{(Y : 1), (Y : 4)\} \]
\[ B_1 \Join B_2 = Box[x + y = 5 \lor y = z^2 \land z \leq 3] = \{ [X : 1, Y : 4, Z : 1..3], [X : 2, Y : 3, Z : 1..3], [X : 3, Y : 2, Z : 1..3], [X : 4, Y : 1, Z : 1..3], [X : 1..4, Y : 9, Z : 3] \} \]

We define these three operators (\(\square\), \(\Join\), and \(\Box\)) have equal precedence and have semantics analogous to relational algebra operators.

Let B be a box expression and C be a box expression and have semantics analogous to relational algebra operators.

Let B be a box expression and D be a dimension set. A formal syntax for box expression B is defined below (left column) and the precedence rules for box operators are defined in right column.

<table>
<thead>
<tr>
<th>syntax</th>
<th>precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>(B := b)</td>
<td>1. (\downarrow), (\uparrow)</td>
</tr>
<tr>
<td>(B \Box B)</td>
<td>2. (\Box)</td>
</tr>
<tr>
<td>(B \Join B)</td>
<td>3. (\Join), (\Join), (\Box)</td>
</tr>
<tr>
<td>(B \uparrow D)</td>
<td>3. (\Box)</td>
</tr>
</tbody>
</table>

3.4. Mixed Operators

The context operators \(\Box\), \(\Join\), \(\Box\), \(\Downarrow\), and \(\Rightarrow\) can be used as mixed operators when one of the operands is a context variable and the other operand is a Box variable. Their definitions are as follows:

**Definition 5** Let \(b\) be a Box variable, and \(c\) a context variable.

1. \(b \mid c = c \mid b = b \lor c\) nondeterministically;
2. \(b \oplus c = \{c_1 \oplus c \mid c_1 \in b\}\);
3. \(c \circ b\) is not defined;
4. \(b \circ c = \{c_1 \circ c \mid c_1 \in b\}\);
5. \(c \circ b\) is not defined;
6. \(c \sqcup b = c \sqcup c = \{c \sqcup c \mid c_1 \in b\}\);
7. \(c \sqcap b = c \sqcap c = \{c \sqcap c \mid c_1 \in b\}\);
8. \(c \vdash b = \{c \vdash c \mid c_1 \in b\}\);
9. \(b \vdash c = \{c_1 \vdash c \mid c_1 \in b\}\);
10. \(c \rightarrow b = \{c \rightarrow c_1 \mid c_1 \in b\}\);
11. \(b \rightarrow c = \{c \rightarrow c \mid c_1 \in b\}\);

<table>
<thead>
<tr>
<th>syntax</th>
<th>precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>(M := B \mid C)</td>
<td>1. (\downarrow), (\uparrow)</td>
</tr>
<tr>
<td>(B \Downarrow C)</td>
<td>2. (\Downarrow)</td>
</tr>
<tr>
<td>(B \Rightarrow C)</td>
<td>3. (\Rightarrow), (\Rightarrow)</td>
</tr>
<tr>
<td>(B \rightarrow C)</td>
<td>5. (\rightarrow)</td>
</tr>
<tr>
<td>(B \sqcup C)</td>
<td>6. (\sqcup), (\sqcup), (\sqcup)</td>
</tr>
<tr>
<td>(B \sqcap C)</td>
<td>7. (\sqcap), (\sqcap), (\sqcap)</td>
</tr>
</tbody>
</table>

A mixed expression involves Box variables, context variables, and mixed operators. Mixed expressions are evaluated by a strict application of precedence rules shown in the above table (right column) and the mixed mode evaluation rules given in Definition 5. Let B be a box expression and C be a context expression. A formal syntax for mixed expression M is defined in the above table (left column).

4. Applications of Lucx

In this section, we give an overview of our current work on applying Lucx to problem solving in different application domains.

4.1. Agent Communication Language

Lucx can be used as an Agent Communication Language (ACL) [2]. An ACL is a high-level language to describe messages exchanged between agents in agent-based systems. The language has the constructs to express the description of complex objects, intentions of agents, shared plans, specific strategies, business and security policies. An ACL must support interoperability in an agent community while providing the freedom for an agent to hide or reveal its internal details to other agents. The two existing ACLs are Knowledge Query and Manipulation Language (KQML) and the FIPA communication language [6]. KQML has a predefined set of reserved performatives. It is neither a minimal required set nor a closed set. The communication primitives in FIPA ACL are called communicative acts (CA), yet they are the same as KQML primitives.

Due to the static nature of the predefined CAs in FIPA and performatives in KQML, it is not possible to express the dynamic aspects in agent’s states and requirements. Thus, interoperability is not fully achieved. In using Lucx as ACL this problem is remedied. The performatives are expressed as context expressions, and context is first class object in Lucx, hence we are able to dynamically manipulate performatives. The name of a performative is considered as an expression, and the rest of the performative constitutes a context which can be understood as a communication context, with each field except the name in the message being a micro context. The communication context will be evaluated by the receiver, by evaluating the expression at the context obtained by combining the micro contexts. In some cases, the receiver may combine the communication context with its local context to generate a new context.

The syntax of a message in Lucx from agent A is of the form \(E_A, E'_A\), where \(E_A\) is the message name and \(E'_A\) is a context expression. In an implementation \(E_A\) corresponds to a function. The context \(E'_A\) includes all the information that agent A wants to convey in an interaction to another agent. A response from agent B to agent A will be of the form \(E_B, E'_B\), where \(E'_B\) will include the reference to the query for which this is a response in addition to the contexts in which the response should be understood.

}\]
The operational semantics of Lucx is the basis for query evaluation. The query from agent $A$ ($E_A, E'_A$) to agent $B$ is evaluated as follows:

1. agent $B$ obtains the context $F_B = E'_A \oplus L_B$, where $L_B$ is the local context for $B$.
2. agent $B$ evaluates $E_A \oplus F_B$.
3. agent $B$ constructs the new context $E''_B$ that includes the evaluated result and information suggesting the context in which it should be interpreted by agent $A$, and
4. sends the response $\langle E_B, E''_B \rangle$ to agent $A$.

**Example 5** A query from agent joe about the price of a share of IBM stock which was encoded as “ask-one” performative in [2] is represented in Lucx as the expression $E \circ E' = E_1 \oplus E_2 \oplus E_3 \oplus E_4 \oplus E_5 \oplus E_6$.

where

$E = \text{"ask-one";}$

$E_1 = \text{\{sender: joe\};}$

$E_2 = \text{\{content: \{PRICEIBM\', price\};}$

$E_3 = \text{\{receiver: STOCK \- SERVER\};}$

$E_4 = \text{\{reply \- with: IBM \- STOCK\};}$

$E_5 = \text{\{language: LPROLOG\};}$

$E_6 = \text{\{ontology: NYSE \- TICKS\};}$

end

Lucx can be used as a declarative language for Real-time Reactive Programming [10]. Synchronous dataflow languages Lustre [4], and RLucid [9], based on Lucid, have been shown to be useful for reactive programming. SIGNAL is a dataflow language in which *signals*, as streams, model timed sequences of typed values. In all these approaches time is discrete, and streams implicitly have the time dimension, although clocks associated with dimensions may be different. Lucx is more expressive than the above languages: time constraints involving multiple clock evaluations can be represented by *Box* expressions and both discrete and continuous time models can be programmed [10].

Lucx can faithfully represent the formal model of a timed system specified by a variant of the Timed Input Output Automaton, which we refer to as an extended state machine (ESM). In Lucx, the representation of events, states, state transitions, functions, contexts and boxes of timed systems are all streams. In the formal model we assume that one or more clocks may be used and constraints on state transitions are specified in guard-action paradigm. The guard $g$ on a transition from state $s_i$ to $s_j$ is of the form $p \land tc$, where $p$, a predicate on the variable in state $s_i$, serves as a precondition for enabling the transition and $tc$ is the time constraint predicate $\text{lower} \leq t < \text{upper}$. The action $a$ is a predicate on the variable in the post state $s_j$. The static aspects of a state machine specification are represented as follows:

1. State transitions are modelled as a 2-dimensional stream $\text{tf}$, which has dimensions $\text{STATE}_{\text{from}}$ and $\text{STATE}_{\text{to}}$ with state names as tags. The evaluation $\text{tf} @ [\text{STATE}_{\text{from}} : s_i, \text{STATE}_{\text{to}} : s_j]$ is the tuple $(m, e)$, where $m$ is the transition number and $e$ is the event triggering the transition from $s_i$ to $s_j$ in this example.

2. A precondition is modelled as a 1-dimensional stream $\text{pre}$, with dimension $\text{TRAN}$ and tag $\mathbb{N}$. The evaluation $\text{pre} @ [\text{TRAN} : k]$ is a tuple $(p_k, v)$ giving the predicate $p_k$ for variable $v$.

3. A postcondition is modelled similarly, as a stream $\text{post}$, with dimension $\text{TRAN}$ and tag $\mathbb{N}$. The evaluation $\text{post} @ [\text{TRAN} : k]$ is a tuple $(a_k, v)$, giving the postcondition for variable $v$.

4. A time constraint is modelled as a 1-dimensional stream $\text{tc}$, which has one dimension $\text{TRAN}$ with tag $\mathbb{N}$. The evaluation $\text{tc} @ [\text{TRAN} : k]$ is a tuple of integers $(\text{times}_k, \text{lower}_k, \text{upper}_k)$ corresponding to the constraint $\text{lower} \leq t < \text{upper}$ for transition $k$.

The dynamic behaviour of the state machine is the set of traces produced according to the state transition semantics. For each state $s_i$, let $E(s_i)$ denote the set of events that are
possible in \( s_i \).

\[
(s_t, v_i) \wedge e \in E(s_t) \wedge p((v_i)_t) \wedge \text{te}(t) \\
\frac{}{(s_t, v_i) \xrightarrow{e} (s_j, v_j) \wedge v'_j = a((v_j)_t)}
\]

We represent each trace of a machine by a stream of tuples \((s, v_i)\) in the program, where \( s \in S \), a finite set of states in the formal model, and \( v \) is the active variable. An element of the trace is computed by applying the state transition semantics to the element that was generated at the previous step. If event \( e \) occurs at time \( t \), and is admissible for the current element in the trace, the transition happens instantaneously; if it is not admissible in this state, transition does not happen, but time is allowed to progress.

In general, if there are several state machines, the program will have a stream \( P \) with the dimensions \( \{\text{Time}, \text{Machine}\} \). The evaluation \( P[@\text{Time} : i] \) is a state stream \( M_i \) corresponding the \( i \)-th state machine \( M_i \) in the model. The following function \( \text{progress}(P, t, e) \) implements the operational semantics of the ESM to compute the status of the system following the occurrence of event \( e \) at time \( t \):

**Example 6** \( \text{progress}(\text{Time, Machine})(P, t, e) = \)

\[
P[@\text{Time} : i] \xrightarrow{\text{by} \cdot \text{Machine}} \\
\text{if } (\text{IsAdmissible}(P[@\text{Time} \# \text{Machine}, e, t])) \\
\text{then NextState}(P, e, t) \\
\text{else } P[@\text{Time} : i] \# \text{.Machine};
\]

In GIPSY environment [7] Lucx programs may call external functions written in Java, the target language of the compiler. Hence, \text{IsAdmissible} and \text{NextState} functions will be implemented in Java.

## 5. Implementing Lucx in GIPSY

The Generic Intensional Programming Language (GIPL), consisting only of \( @ \) and \( \# \) operators, is one of the members of the Lucid family of languages. As its name implies, it is actually a generic language, i.e. all other languages of the family can be translated into it (syntactically and semantically speaking). All the other members of the Lucid family of languages are called Specific Intensional Programming Languages (SIPL), such as Indexical Lucid and Lucx. The GIPSY is an IP investigation platform under development which allows the automated generation of compiler components for the different variants of the Lucid family of languages, as well as an execution environment allowing for these programs to be executed in multi-threaded or distributed mode [7].

Using concrete language specifications, the GIPSY allows for the automated generation of compiler components for all SIPLs. These specifications take the form of (1) specification of the syntax of the language and (2) specification of the translation rules translating the syntactic constructs into the generic GIPL constructs. For example, in [8], translation rules are provided to translate Indexical Lucid operators such as \text{first}, \text{next}, \text{by} into GIPL.

One of the main precepts of the GIPSY is that the General Execution Engine (GEE) can execute programs written in any IP language that is part of the Lucid family of languages. This is achievable through the generic nature of the GIPL, i.e. all SIPLs are translated into the GIPL, and the GEE then executes the generic version of the program. This is similar to Java’s byte code. For each SIPL, the GIPC (General Intensional Programming Compiler) generates and uses a specific parser (SIPL parser). That parser is automatically generated using the JavaCC tool, with a grammatical definition provided by the user. This parser generates an abstract syntax tree (SIPL AST), which has to be translated into its generic counterpart using an AST translator (SIPL-GIPL AST translator). That translator is also automatically generated by the GIPC using the translation rules of this SIPL as input.

Currently, the compiler for Indexical Lucid has been implemented successfully in the GIPSY. The parser for Indexical Lucid is generated using JavaCC, and the SIPL-GIPL AST translator uses the translation rules translating Indexical Lucid operators into GIPL. According to the design of the GIPSY, for a new SIPL, a front end that provides the SIPL parser and SIPL-GIPL AST translator is required to implement the compiler for this SIPL in the GIPSY.

Lucx is a conservative extension of Lucid and is thus a SIPL. We will provide the Lucx parser and Lucx-GIPL AST translator as a Lucx front end to GIPC. Lucx parser can also be automatically generated using the JavaCC tool. In section 5.1, we provide the translation rules for translating Lucx operators into Indexical Lucid operators. Combined with the translation rules for Indexical Lucid operators provided in [8], we achieve a two-pass Lucx-GIPL AST translator. Once these two models are integrated into GIPSY, the programs written in Lucx will be compiled and run in GIPSY.

### 5.1. Translation Rules for Context Operators

The translation rules for context operators are shown below. Because we are still developing these rules, the list is not exhaustive.

1. **Primitive Functions**: \( \text{dim}_m(m) \), which returns the dimension of a \( m \)-context \( m \); \( \text{tag}_m(m) \), returning the tag of a \( m \)-context \( m \); \( \text{dim}(s) \), returning the set of dimensions of a \( s \)-context \( s \); and \( \text{tag}(s) \), returning the set of tags of a \( s \)-context \( s \). Those primitive functions should be hard coded, and hence written in Java and incorporated into the GEE. Set operators, such as \( \cap, \in \) we use below, do not exist in Lucid, however for simplicity, these operators are assumed to be hard coded in Java as well.
2. **Override Function:** Override($E, E'$) ($\oplus$), $E$, and $E'$ are s contexts and treated as a m context stream respectively.

\[
\text{Override}(E, E') = \begin{cases} 
\text{if}(\text{dim}(E) \cap \text{dim}(E') = \text{NULL}) & \text{then E fby E'} \\
\text{else E'} \text{ fby append}(E, E') 
\end{cases}
\]

where

\[
\text{append}(E, E') = \begin{cases} 
\text{if}((\text{dim}_m(\text{first E}) \in \text{dim}(E')) = \text{false}) & \text{then first E fby append}(\text{next E}, E') \\
\text{else append}(\text{next E}, E') 
\end{cases}
\]

3. **Difference Function:** Difference($E, E'$) ($\ominus$)

\[
\text{Difference}(E, E') = \begin{cases} 
\text{if}(\text{compare}(\text{first E}, E') = \text{true}) & \text{then Difference}(\text{next E}, E') \\
\text{else first E fby Difference}(\text{next E}, E') 
\end{cases}
\]

where

\[
\text{compare}(m, E') = \text{if}(m == (\text{first E'})) \\
\text{then true} \\
\text{else false} \text{ fby compare}(m, \text{next E'})
\]

4. **Intersection Function:** Intersection($E, E'$) ($\cap$)

\[
\text{Intersection}(E, E') = \begin{cases} 
\text{if}(\text{compare}(\text{first E}, E') = \text{true}) & \text{then first E fby Intersection}(\text{next E}, E') \\
\text{else Intersection}(\text{next E}, E') 
\end{cases}
\]

5. **Union Function:** Union($E, E'$) ($\cup$)

\[
\text{Union}(E, E') = E' \text{ fby Difference}(E, E')
\]

6. **Projection Function:** Projection($E, D$) ($\downarrow$). $E$ is a m context stream and D is a dimension stream.

\[
\text{Projection}(E, D) = \begin{cases} 
\text{if}(\text{compare}(\text{tag}_m(\text{first E}), D) = \text{true}) & \text{then first E fby Project}(\text{next E}, D) \\
\text{else Project}(\text{next E}, D) 
\end{cases}
\]

7. **Hiding Function:** Hiding($E, D$) ($\uparrow$)

\[
\text{Hiding}(E, D) = \begin{cases} 
\text{if}(\text{compare}(\text{tag}_m(\text{first E}), D) = \text{true}) & \text{then Hiding}(\text{next E}, D) \\
\text{else first E fby Hiding}(\text{next E}, D) 
\end{cases}
\]

6. **Conclusion**

The notion of context is the cornerstone of the intensional programming paradigm. The previous versions of Lucid were merely using the notion of context of evaluation. They provided a single operator for the navigation in the context of evaluation, but did not provide a mechanism to represent and manipulate contexts as first class values. Lucx allows the definition (infinite) streams of contexts, thus offering the mechanism to program non-terminating continuous systems, such as hybrid systems.

The use of contexts as first class values increases the expressive power of the language by an order of magnitude. It allows the definition of aggregate contexts, which are a key feature to achieve efficiency of evaluation through granularization of the manipulated data. It also allows to use the paradigm for agent communication by allowing the sharing and manipulation of multidimensional contextual information among agents. It allows the use of the paradigm for real-time reactive programming as well. We are now working on the use of contexts as first class values to provide increased possibilities for constraint programming [11].

The next step is to integrate the language Lucx in GIPSY. These new possibilities of the language will enable us to broaden the scope of our investigations on the applicability of the Intensional Programming Paradigm.

**References**


