A Context Theory for Intensional Programming *

Kaiyu Wan, Vasu Alagar, and Joey Paquet

Department of Computer Science and Software Engineering
Concordia University
Montreal, Quebec H3G 1M8, Canada
{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. It allows us to write programs in Lucx (Lucid enriched with context) in a high level of abstraction which is closer to the problem domain. We include a discussion on context theory, representation of context aggregations, and 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 discussed.
Keywords: Context, Context Theory, Intensional 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. In modelling human-computer interaction [5], context includes the *physical place* of the user, the *time constraints*, and the system's assumption about users interests. In Ubiquitous computing [3], context is understood as both *situated* and *environmental*. In natural language processing, contexts arise as *situations* for interpreting natural language constructs. In imperative programming languages, context introduces index, constants, and pointers. In functional languages, *static* context introduces definitions and constraints, and *dynamic* context processes the executable information for evaluating expressions. In Artificial Intelligence(AI), the notion of *context* was introduced by McCarthy and later used by Guha [4] as a means of expressing assumptions made by natural language expressions. 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 [7] 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. Although the notion of context was implicit in Lucid, an Intensional Programming Language, context cannot be explicitly declared and manipulated in Lucid. By introducing context as a first class object in Lucid, we remove this limitation. The language, thus extended, is called *Lucx*(Lucid extended with *contexts*).

* This work is supported by grants from the Natural Sciences and Engineering Research Council of Canada.

The goal of this paper is to illustrate how context is formally defined and introduced as first class objects in Lucx and as a result, how Lucx can be used for programming diverse application domains. The context theory that we are developing provides a semantic basis for context manipulation in Lucx. The paper is organized as follows: In Section 2 we review briefly contexts in Intensional Programming Paradigm. Section 3 discusses the context theory applied in Lucx. In Section 4 we discuss the syntax and semantics of Lucx. An example of Lucx programming and implementing Lucx are also illustrated. We conclude our work in Section 5.

2 Context in Intensional Programming Paradigm

Intensional Logic, a family of mathematical formal systems that permits expressions whose value depends on *hidden context*, came into being from research in natural language understanding. Basically, intensional logics add *dimensions* to logical expressions, and non-intensional logics can be viewed as *constant* in all possible dimensions, i.e. their valuation does not vary according to their context of utterance. *Intensional operators* are defined to *navigate* in the context space. In order to navigate, some dimension *tags* (or indexes) are required to provide place holders along dimensions. These dimension tags, along with the dimension names they belong to, are used to define the context for evaluating intensional expressions. For example, we can have an expression:

E: the average temperature this month here is greater than $0°C$.

This expression is intensional because the truth value of this expression depends on the context in which it is evaluated. The two intensional natural language operators in this expression are *this month* and *here*, which refer respectively to the time and space dimension. If we evaluate the expression in different cities in Canada and in the months of a particular year, the extension of the expression varies. Hence, we have the following valuation for the expression:

$E' =$

	Ja	Fe	Mr	Ap	Ma	Jn	Jl	Au	Se	Oc	No	De
Montreal	F	F	F	F	T	T	T	T	T	F	F	F
Ottawa	F	F	F	T	T	T	T	T	T	F	F	F
Toronto	F	F	T	T	T	T	T	T	T	T	F	F
Vancouver	F	T	T	T	T	T	T	T	T	T	T	T

The intension of the expression is the above whole table; and the extension of the expression in the time point *Ap* and in the space point *Ottowa* is T.

Intensional programming paradigm has its foundations on intensional logic. It retains two aspects from intensional logic: first, at the syntactic level, are context-switching operators, called *intensional operators*; second, at the semantic level, is the use of *possible worlds semantics* [7].

Lucid was a data-flow language and evolved into a Multidimensional Intensional Programming Language [1]. In extending Lucid with contexts we preserve the intensionality in Lucx. Moreover, contexts exist independent of any objects in the system. That is, one context may be used to evaluate different expressions, at the same time expressions can also be evaluated at different contexts. This feature distinguishes the language Lucx from other imperative languages or functional languages, where index

(for imperative languages) or evaluation environment(for functional language) are always bound to statements or expressions. Because of the separation of the definition of expressions from contexts, Lucx has provided more power of representing problems in different application domains and given more flexibility of programming.

3 Context Theory in Lucx

Context theory provides a semantic basis for Lucx programs. A context in the theory need not be finite. However, context in Lucx has a *finite* number of dimensions and along each dimension is associated a tag set, which is enumerable. This is in contrast to Guha's notion, wherein contexts are *infinite*, *rich*, and *generalized* objects. We are motivated by Guha's work. However not all contexts studied by Guha can be dealt within our language. On the other hand, 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 Definition

In Intensional Programming context is a reference to the representation of the "possible worlds" relevant to the current discussion. The "possible world" is a multidimensional space enclosing all possible information pertaining to the discussion. Motivated by this, we formalize contexts as *a subset of a finite union of relations*. The relations are defined over *dimension* and *tag* sets. Let $DIM = \{d_1, d_2, \ldots, d_n\}$ denote a finite set of dimension names. We associate with each $d_i \in DIM$ a unique enumerable tag set X_i. Let $TAG = \{X_1, \ldots, X_r\}$ denote the set of tag sets. There exists functions $f_{dimtotag} : DIM \rightarrow TAG$, such that the function $f_{dimtotag}$ associates with every $d_i \in DIM$ exactly one tag X_j 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* $\mid \Delta \mid$*, where* $\Delta \subset DIM$ *includes the dimensions that appear in C.*

A context is written using *enumeration* syntax, as $[d_1 : x_1, \ldots, d_n : x_n]$, where $d_1, \ldots, d_n$ are dimension names, and x_i is the tag for dimension d_i. We say a context C is *simple* (s_context), if $(d_i, x_i), (d_j, x_j) \in C \Rightarrow d_i \neq d_j$. A simple context C of degree 1 is called a *micro* (m_context) context.

Example 1 As an example consider a system in which computations involve pressure, volume, and time, where pressure is observed by different sensors, volume is measured by different devices, and the sampling frequencies are different. The three distinguished dimensions are: (1).*Sampling Time ST* with index $\mathbb{N}$; (2). *Pressure P* with index set $\{s_1, \ldots, s_k\}$, where $s_1, \ldots, s_k$ are named sensory devices; and (3). *Volume V* with index set $\{m_1, \ldots, m_q\}$, where $m_1, \ldots, m_q$ are named measuring devices. A context $c = [ST : 1, P : s_2, V : m_3]$ for one stream σ may be interpreted as a reference

to the tuple $\langle t_1, p_2, v_3 \rangle$, where at the first sampling time t_1 the value of volume measured by m_3 is v_3 and the value of pressure observed by the sensor s_2 is p_2. Supposing $t_1, p_2, v_3 \in \mathbb{R}$, the domain for the entities in the stream σ is $\mathbb{R} \times \mathbb{R} \times \mathbb{R}$. The same context may also be used as a reference to another possible world containing the expression $\sigma' = \frac{p \star v}{t}$. Such a reference will produce the result $\frac{p_2 \star v_3}{t_1}$, which is the result of evaluating the expression σ' with the substitution $[t \mapsto t_1;\ p \mapsto p_2;\ v \mapsto v_3]$. In this case, the domain for the entities in the stream σ' is $\mathbb{R}$.

Several functions on contexts are predefined in [2]. The basic functions *dim* and *tag* are to extract the set of dimensions and their associate tag values from a set of contexts. 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 [9], we have formally defined the following context operators: the *override* $\oplus$ is similar to function override; *difference* $\ominus$, *comparison* $=$, *conjunction* $\sqcap$, and *disjunction* $\sqcup$ are similar to set operators; *projection* $\downarrow$ and *hiding* $\uparrow$ are selection operators; *constructor* $[\_ : \_]$ is used to construct an atomic context; *substitution* / is used to substitute values for selected tags in a context; *choice* | accepts a finite number of contexts and nondeterministically returns one of them; *undirected range* $\rightleftharpoons$ and *directed range* $\rightharpoonup$ produce a set of contexts.

Example 2 illustrates an overall example for some of those operators.

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 $c_1 \oplus c_2 = [X : 2, Y : 4, Z : 5]$, $c_1 \ominus c_2 = [X : 3]$, $c_1 \downarrow D = [Y : 4]$,
$c_1 \sqcap c_2 = [X : 2, Y : 4]$, $c_1 \sqcup c_2 = [X : 2, X : 3, Y : 4, Z : 5]$, $c_2 \uparrow D = [X : 2]$,
$c_2 \rightleftharpoons c_3 = \{[X : 2, Y : 2, Z : 5], [X : 2, Y : 3, Z : 5], [X : 2, Y : 4, Z : 5]\}$,
$c_3 \rightharpoonup c_2 = \{[X : 2, Y : 2, Z : 5], [X : 2, Y : 3, Z : 5], [X : 2, Y : 4, Z : 5]\}$,
$c_2/\langle Y, 3 \rangle = [X : 2, Y : 3, Z : 5]$, $c_2 \rightharpoonup c_3 = \varnothing$

In order to provide a precise meaning for a context expression, we have defined the precedence rules for all the operators in Figure 1[a] (right column) (from the highest precedence to the lowest) and described a set of evaluation rules for context expressions in [9]. 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 in Figure 1[a](left column).

3.3 Box and Box Operators

A context which is not a micro context or a simple context is called a non-simple context. For example, context $c_4 = [X : 3, X : 4, Y : 3, Y : 2, U : \mathit{blue}]$ is a non-simple context. In general, a non-simple context is equivalent to a set of simple contexts [2]. In several applications we deal with contexts that have the same dimension set $\Delta \subseteq \mathit{DIM}$ and the tags satisfy a predicate formula p. The short hand notation for such a set is $\mathit{Box}[\Delta \mid p]$.

syntax		precedence
$C ::= c$	$C = C$	1. $\downarrow, \uparrow, /$
$\mid C \supseteq C$	$C \subseteq C$	2. $\mid$
$\mid C \mid C$	C/C	3. $\sqcap, \sqcup$
$\mid C \oplus C$	$C \ominus C$	4. $\oplus, \ominus$
$\mid C \sqcap C$	$C \sqcup C$	5. $\rightleftharpoons, \rightharpoonup$
$\mid C \rightleftharpoons C$	$C \rightharpoonup C$	6. $=, \subseteq, \supseteq$
$\mid C \downarrow D$	$C \uparrow D$	

(a) Rules for Context Expression

syntax		precedence
$B ::= b$	$B \mid B$	1. $\downarrow, \uparrow, /$
$\mid B \boxdot B$	$B \boxtimes B$	2. $\mid$
$\mid B \boxplus B$	$B \downarrow D$	3. $\boxdot, \boxplus, \boxtimes$
$\mid B \uparrow D$	$B/\langle d, t\rangle$	

(b) Rules for Box Expression

Fig. 1. Rules for Contex and Box Expressions

Definition 2 *Let $\Delta = \{d_1, \ldots, d_k\}$, where $d_i \in DIM$ $i = 1, \ldots, k$, and p is a predicate formula defined on the tuples of the relation $\Pi_{d \in \Delta} f_{dimtotag}(d)$. The syntax*

$$Box[\Delta \mid p] = \{s \mid s = [d_{i_1} : x_{i_1}, \ldots, d_{i_k} : x_{i_k}]\},$$

where the tuple $(x_1, \ldots, x_k)$, $x_i \in f_{dimtotag}(d_i)$, $i = 1, \ldots k$ satisfy the predicate formula p, introduces a set S of contexts of degree k. For each context $s \in S$ the values in $tag(s)$ satisfy the predicate formula p.

The context operators projection ($\downarrow$), hiding ($\uparrow$), choice ($|$), and substitution ($/$) introduced in Section 3.2 can be naturally lifted to sets of contexts, in particular for *Box*es. As an example : $\uparrow$ and $\downarrow$ can be lifted for Box B: $B \uparrow D = \{c \uparrow D \mid c \in B\}$, $B \downarrow D = \{c \downarrow D \mid c \in B\}$. However not all context operators have natural extensions. Instead, the following three operations $\boxtimes$ (join), $\boxdot$ (intersection), and $\boxplus$ (union) are defined [2] for sets of contexts introduced by *Box*.

Example 3 *:*

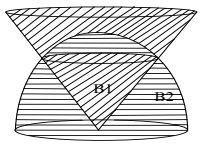
Let $DIM = \{X, Y, Z\}$, $f_{dimtotag}(X) = f_{dimtotag}(Y) = f_{dimtotag}(Z) = \mathbb{N}$,
$B_1 = Box[X, Y \mid x, y \in \mathbb{N} \wedge x + y = 5]$, $B_2 = Box[Y, Z \mid y, z \in \mathbb{N} \wedge y = z^2 \wedge z \leq 3]$.
Then $B_1 = \{[X : 1, Y : 4], [X : 2, Y : 3], [X : 3, Y : 2], [X : 4, Y : 1]\}$
$B_2 = \{[Y : 1, Z : 1], [Y : 4, Z : 2], [Y : 9, Z : 3]\}$.
Hence $B_1 \boxtimes B_2 = Box[X, Y, Z \mid x + y = 5 \wedge (y = z^2 \wedge z \leq 3)]$
$= \{[X : 1, Y : 4, Z : 2], [X : 4, Y : 1, Z : 1]\}$
$B_1 \boxdot B_2 = Box[Y \mid x + y = 5 \wedge (y = z^2 \wedge z \leq 3)] = \{[Y : 1], [Y : 4]\}$
$B_1 \boxplus B_2 = Box[X, Y, Z \mid x + y = 5 \vee (y = z^2 \wedge z \leq 3)] = \{[X : 1, Y : 4, Z : 1..3],$
$[X : 2, Y : 3, Z : 1..3], [X : 3, Y : 2, Z : 1..3], [X : 4, Y : 1, Z : 1..3],$
$[X : 1..3, Y : 1, Z : 1], [X : 2..4, Y : 4, Z : 2], [X : 1..4, Y : 9, Z : 3]\}$

We define these three operators ($\boxtimes$, $\boxplus$, and $\boxdot$) have equal precedence and have semantics analogous to relational algebra operators.

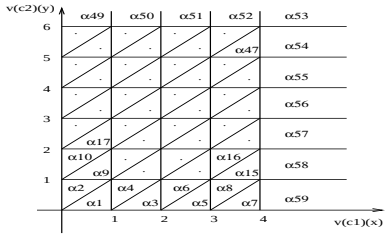
Let B be a box expression and D be a dimension set. A formal syntax for box expression B is defined in Figure 1[b] (left column) and the precedence rules for box operators are defined in Figure 1[b] right column.

3.4 Context Category

Context Regions A *context region* is a finite subset of a multidimensional space generated by a set of dimensions. Boxes can be used to represent different context regions. For example, Figure 2[a] shows two different context regions, which can be represented as follows: $B_1 = Box[X, Y, Z \mid x^2 + z^2 \leq 16 \wedge x = \frac{1}{2}z \wedge z \geq 0]$, $B_2 = Box[X, Y, Z \mid x^2 + y^2 + z^2 \leq 9 \wedge z \geq 0]$. Box B_1 defines a cone, and Box B_2 defines the upper half of hemisphere with the radius 3. If we restrict to integer indices, then the set of contexts defined by Box B_1 consists of all the points with integer coordinates within the cone, and the set of contexts defined by Box B_2 consists of all the points with integer coordinates within the hemisphere.



(a) Different Context Regions



(b) Clock Regions

Fig. 2. Context Category

In timed systems having continuous time model, *clock regions* which are equivalence classes of clock valuations arise when several clocks are used. These clock regions are treated as context regions, and can be represented as boxes [8]. As an example, consider clocks c_1 and c_2 when the maximum duration that can elapse in the clocks are $m_1 = 4$, and $m_2 = 6$. This gives rise to 59 clock regions, as shown in Figure 2[b]. The clock regions corresponding to a set of clocks is represented as a set of *Boxes*. Each *Box* is defined by the dimension set $\Delta = \{c_1, \ldots, c_k\}$, and a constraint on the clock valuations. For example, $Box[\Delta \mid p_1]$, where $\Delta = \{c_1, c_2\}$ and $p_1 = (0 < v(c_1)(x) < 1) \wedge (0 < v(c_2)(y) < v(c_1)(x))$ refers to the region α_1. The tag sets for clocks are reals. For discrete time modelled by multiple clocks the tag sets are integers, and regions become lattice points, vertices of convex regions.

Nested Contexts Nested contexts define the *meta relation* between different contexts. This is similar to Guha's definition of nested context [4]. In his definitions, nested context enables to nest $ist(f, c)$ formulas, which define that formula f is true in context c, and $ist(c_i\ ist(c_j,\ \alpha))$, $ist(c_k\ ist(c_l\ \ldots\ ist(c_j\ \alpha)\ \ldots))$ are all valid. In the former example $ist(c_i\ ist(c_j,\ \alpha))$, context c_i is *outer* or *meta to* context c_j. Guha also provides *entering* and *exiting* actions to migrate the interpretation of formulas from contexts. Similarly, we provide $@^*$ operator to navigate the evaluation of expressions between nested contexts.

Definition 3 *Let $p(x,y)$ and $q(x,y,z)$ be two predicate formulas. If $\forall\, a,b$ for which $p(a,b)$ is true, and if $\exists\, c$ such that $q(a,b,c)$ is true, then we call $p(x,y)$ as a* projection *of the predicate formula $q(x,y,z)$ and write $p(x,y) =\downarrow_z q(x,y,z)$. Conversely, we say $q(x,y,z)$ is a simple extension of $p(x,y)$ and write $p(x,y) \stackrel{z}{\rightarrow} q(x,y,z)$. In general, a predicate formula extended with $s > 1$ arguments can be inductively defined as follows:*

1. $p(x_1,\ldots,x_r) \stackrel{x_{r+1}}{\rightarrow} q(x_1,\ldots,x_r,x_{r+1})$
2. $p(x_1,\ldots,x_r) \stackrel{x_{r+1},\ldots,x_{r+s}}{\longrightarrow} q(x_1,\ldots,x_r,\ldots,x_{r+s})$
   $\stackrel{\Delta}{=} p_0(x_1,\ldots,x_r) \stackrel{x_{r+1}}{\rightarrow} p_1(x_1,\ldots,x_r,x_{r+1}) \stackrel{x_{r+2}}{\rightarrow} p_2(x_1,\ldots,x_{r+2})\ldots$
   $\stackrel{x_{r+s}}{\rightarrow} p_s(x_1,\ldots,x_{r+s})$, *where $p_0 \equiv p, p_s \equiv q$.*

Definition 4 *We define a relation $\sqsubset$ on a set of boxes $\mathbb{B}$ as follows: for $b_1, b_2 \in \mathbb{B}$, $b_1 = Box[\Delta_1 \mid p_1]$, $b_2 = Box[\Delta_2 \mid p_2]$, $b_1 \sqsubset b_2$ iff $\Delta_1 \subset \Delta_2$ and p_2 is an extension of the logic expression p_1. It is easy to see that $\sqsubset$ is an irreflexive partial order on $\mathbb{B}$. We define a partially order chain $b_1 \sqsubset b_2 \ldots \sqsubset b_k$ to be* nested*, and refer to the boxes in the chain as* nested contexts*.*

We want to study the relationship between nested contexts and sets of expressions obtained by evaluating a given expression E at the boxes in the nested chain.

Definition 5 *Let $B = \{b_1, b_2, \ldots\}$ be a finite set of boxes, $b_i = Box[\Delta_i \mid p_i]$, and E be an expression. Define the relation $\rhd$ on the set $\{E@^*b_1, E@^*b_2, \ldots\}$ as follows:*

$$E@^*b_i \rhd E@^*b_k$$

iff for $E_{ij'} = E@c_{ij'}, c_{ij'} \in b_i$, there exists $E_{kj} = E@c_{kj}, c_{kj} \in b_k$, such that $c_{kj} \uparrow \{\Delta_k - \Delta_i\} = c_{ij'}$, and $E_{kj} = E_{ij'}@(c_{kj} \ominus c_{ij'})$.

Theorem If $b_1 \sqsubset b_2$ then $E@^*b_1 \rhd E@^*b_2$.
Proof Let $b_1 = Box[\Delta_1 \mid p_1]$, $b_2 = Box[\Delta_2 \mid p_2]$, $\Delta_1 = \{X_1, X_2, \ldots, X_k\}$, and $\Delta_2 = \{X_1, X_2, \ldots, X_k, X_{k+1}, \ldots, X_m\}$, $\exists(a_1,\ldots,a_k), p_1(a_1,\ldots,a_k)$ is true,
then $c_{1j'} = [X_1 : a_1, \ldots, X_k : a_k] \in b_1$.
By property $b_1 \sqsubset b_2, \exists\, a_{k+1}, \ldots, a_m$ such that $p_2(a_1,\ldots,a_k,a_{k+1},\ldots,a_m)$ is true.
Hence $c_{2j} = [X_1 : a_1, \ldots, X_k : a_k, X_{k+1} : a_{k+1}, \ldots, X_m : a_m] \in b_2$.
It is easy to verify that $c_{1j'}$ and c_{2j} satisfy $c_{2j} \uparrow \{\Delta_2 - \Delta_1\} = c_{1j'}$
and $E_{1j'} = E@c_{1j'}$, $E_{2j} = E@c_{2j}$ satisfy $E_{2j} = E_{1j'}@(c_{2j} \ominus c_{1j'})$.
Hence it follows that $E@^*b_1 \rhd E@^*b_2$.
In general if $b_1 \sqsubset b_2 \ldots \sqsubset b_k$ is a chain of nested contexts, we get a corresponding chain of *cascading expressions*:

$$E@^*b_1 \rhd E@^*b_2 \ldots \rhd E@^*b_k.$$

This rule gives the base for the reasoning and reducing rules for constraint programming solver mentioned in [9].

Context Dependent Expressions Contexts can be passed as parameters to functions.

Definition 6 *Let* $B_1 = Box[\Delta_1 \mid p_1]$, *and* $B_2 = Box[\Delta_2 \mid p_2]$ *define the context regions in the space generated by the dimensions* $\Delta_1 \cup \Delta_2$. *The* context-dependent expression E *is defined differently in different regions:*

$$\lambda \cdot E = \begin{cases} E_1 & in\ B_1 \ominus B_2 \\ E_2 & in\ B_2 \ominus B_1 \\ E_3 & in\ B_1 \sqcap B_2 \end{cases}$$

$\mu \cdot E$ is defined to indicate corresponding context regions, namely,
$\mu \cdot E = \{B_1 \ominus B_2, B_2 \ominus B_1, B_1 \sqcap B_2\}$

An application of Definition 6 is to use contexts as parameters in a function definition. Let $f : X \times Y \times Z \times C \rightarrow W$, where C is a set of contexts; and $f(x,y,z,c), x \in X, y \in Y, z \in Z, c \in C$, be defined such that for different context values, the function's definitions are different. For example, function $f(x,y,z,c)$ is defined according to different context regions shown in Figure 2[a]. Hence $\mu \cdot f = \{B_1 \ominus B_2, B_2 \ominus B_1, B_1 \sqcap B_2\}$, and $\lambda \cdot f = \{2x^3 + y - 6, x + y^2, z^3 + y\}$. The evaluation of functions f varies depending on the actual context value given as input when f is called.

Given contexts as input, context-dependent functions can be used to produce a new context as a result to achieve adaptation in context-aware system [9].

Dependent Context We define context dependency analogous to the functional dependency in relational data models.

Definition 7 *Let* $\Delta = \{X_1, \ldots, X_k\}$ *be a dimension set. If there exists a function* $\phi_{ij} : f_{dimtotag}(X_i) \rightarrow f_{dimtotag}(X_j)$, *we say* ϕ_{ij} *is a functional dependency in the set* Δ.
In general, a functional dependency exists in Δ, *if* $A \subset \Delta$, $B \subset \Delta$, $A \cap B = \varnothing$, *and there exists a function :*

$\phi_{AB} : \Pi_{X_i \in A} f_{dimtotag}(X_i) \rightarrow \Pi_{X_j \in B} f_{dimtotag}(X_j)$.
For a given functional dependency ϕ_{ij} *in* Δ, *we define dependent contexts as the set of contexts:*

$$S_\Delta = \{c \mid dim(c) = \Delta' \wedge (\{X_i, X_j\} \subseteq \Delta' \subseteq \Delta)\}$$

As an example, $c = [X_i : a, X_j : \phi_{ij}(a)] \in S_\Delta, a \in f_{dimtotag}(X_i)$. This definition is easy to generalize for the general functional dependency ϕ_{AB}.

Dependent context effectively reduces the possible worlds that are relevant to evaluate expressions. As an example, let a function ϕ: $V \rightarrow ST$ be defined as follows: $\phi(m_1) = \phi(m_2) = 1$; $\phi(m_3) = \phi(m_4) = 2$. Hence context of this form $[P : s_1, V : m_3, ST : 1]$ need not be considered for evaluating expressions in the example.

Moreover, dependent contexts also help to represent context sets compactly. In [9], we proved the following: starting with a set S_Δ of contexts, whose dimensions are subsets of Δ and a finite set of functional dependencies on Δ, we can represent S_Δ as an expression $S_\Delta = S_{\Delta_k} \boxtimes S'_{\Delta_k}$, where there is no dependency in S_{Δ_k} and $S'_{\Delta_k} = b_1 \boxtimes b_2 \ldots \boxtimes b_k$, each b_i is a Box representing one dependency. Since a box has a compact representation, the representation for S_Δ given above is a compact way to

manage the contexts in this S_Δ. The substitutions for tags in $b_1, \ldots, b_k$ are subject to dependencies. However, those tags corresponding to dimension in S_{Δ_k} can be done freely. This is analogous to *substitution principle* in functional languages.

4 Intensional Language 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.

Syntax and Semantics of Lucx The syntax of Lucx is shown below.

$$
\begin{array}{ll}
E ::= id & S ::= \{\mathbf{E_1}, \ldots, \mathbf{E_m}\} \\
\quad | \; E(E_1, \ldots, E_n) & \quad | \; \mathbf{Box}[\mathbf{E} \mid \mathbf{E}'] \\
\quad | \; \texttt{if } E \texttt{ then } E' \texttt{ else } E'' & Q ::= \texttt{dimension } id \\
\quad | \; \# & \quad | \; id = E \\
\quad | \; \mathbf{E} \; @ \; \mathbf{E}' & \quad | \; id(id_1, \ldots, id_n) = E \\
\quad | \; [\mathbf{E_1} : \mathbf{E'_1}, \ \ldots, \ \mathbf{E_n} : \mathbf{E'_n}] & \quad | \; Q \; Q \\
\quad | \; \langle \mathbf{E_1}, \ldots, \mathbf{E_n} \rangle \mathbf{E} & \\
\quad | \; \mathbf{select}(\mathbf{E}, \mathbf{E}') & \\
\quad | \; \mathbf{E} \; @^* \; \mathbf{S} & \\
\quad | \; E \texttt{ where } Q &
\end{array}
$$

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 $\mathcal{D}, \mathcal{P}' \vdash E : v$, which means that in the definition environment $\mathcal{D}$, and in the evaluation context $\mathcal{P}'$, expression E evaluates to v. The definition environment $\mathcal{D}$ retains the definitions of all of the identifiers that appear in a Lucid program. Formally, $\mathcal{D}$ is a partial function $\mathcal{D} : \mathbf{Id} \rightarrow \mathbf{IdEntry}$, 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*[7]. The evaluation context $\mathcal{P}'$, is the result of $\mathcal{P} \dagger c$, where $\mathcal{P}$ is the initial evaluating context, c is the defined context expression, and the symbol $\dagger$ denotes the overriding function.

A complete operational semantics for Lucid is defined in [7]. The new semantic rules for Lucx are given below.

$$\mathbf{E_{at(c)}} : \frac{\mathcal{D}, \mathcal{P} \vdash E' : \mathcal{P}' \qquad \mathcal{D}, \mathcal{P}' \vdash E \; : \; v}{\mathcal{D}, \mathcal{P} \vdash E \; @E' : v}$$

$$\mathbf{E_{\#}} : \frac{}{\mathcal{D}, \mathcal{P} \vdash \# : \mathcal{P}}$$

$$\mathbf{E_{.}} : \frac{\mathcal{D}, \mathcal{P} \vdash E_2 : id_2 \qquad \mathcal{D}(id_2) = (\texttt{dim})}{\mathcal{D}, \mathcal{P} \vdash E_1.E_2 : tag(E_1 \downarrow \{id_2\})}$$

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

$$\mathbf{E_{select}} : \frac{E = [\mathtt{d} : \mathtt{v}'] \qquad E' = \langle \mathtt{E_1}, \ldots, \mathtt{E_n} \rangle \mathtt{d} \qquad \mathcal{P}' = \mathcal{P} \; \dagger \; [d \; \mapsto \; v'] \qquad \mathcal{D}, \mathcal{P}' \vdash E' : v}{\mathcal{D}, \mathcal{P} \vdash select(E, E') : v}$$

$$\mathbf{E_{at(s)}}: \frac{\mathcal{D},\mathcal{P}\vdash S\ :\ \{\mathcal{P}_1,\ldots,\mathcal{P}_m\} \qquad \mathcal{D},\mathcal{P}_{i:1..m}\vdash E\ :\ v_i}{\mathcal{D},\mathcal{P}\vdash E\ @^* S : \{v_1,\ldots,v_m\}}$$

$$\mathbf{E_{set}}: \frac{\mathcal{D},\mathcal{P}\vdash E_{w:1..m} : \mathcal{P}_m}{\mathcal{D},\mathcal{P}\vdash \{E_1,\ldots,E_m\} : \{\mathcal{P}_1,\ldots,\mathcal{P}_w\}}$$

$$\mathbf{E_{box}}: \frac{\begin{array}{ll}\mathcal{D},\mathcal{P}\vdash E : \Delta & \Delta = \{v'_1,\ldots,v'_n\} = dim(\mathcal{P}_1) = \ldots = dim(\mathcal{P}_m)\\ \mathcal{D}(v'_{k:1..n}) = (\texttt{dim}) & \mathcal{D},\mathcal{P}\vdash E' : f_p(tag(\mathcal{P}_{i:1..m})) = true\end{array}}{\mathcal{D},\mathcal{P}\vdash Box[E\mid E'] : \{\mathcal{P}_1,\ldots,\mathcal{P}_m\}}$$

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

The evaluation rule for the navigation operator, $\mathbf{E_{at_c}}$, which corresponds to the syntactic expression $\mathbf{E}\ @\ \mathbf{E}'$, evaluates E in context E', where E' is a context defined in Section 3.1. The evaluation rule for the set navigation operator $\mathbf{E_{at_s}}$, which corresponds to the syntactic expression $\mathbf{E}\ @^*\ \mathbf{S}$, evaluates E in a set of contexts S. Hence, the evaluation result should be a collection of results of evaluating E at each element of S. Semantically speaking, the symbol # is a nullary operator, which evaluates to the current evaluation context $\mathcal{P}$. And the symbol . is a binary operator, whose left operand is an expression and the right operand is a single dimension. The semantic rule $\mathbf{E_{tupid}}$ evaluates a tuple as a finite stream whose dimension is explicitly indicated as E in the corresponding syntax rule $\langle\mathbf{E_1},\ldots,\mathbf{E_n}\rangle\mathbf{E}$. Accordingly, the semantic rule $\mathbf{E_{select}}$ picks up one element indexed by E from the tuple E'. The semantic rule $\mathbf{E_{box}}$ evaluates a Box to a set of contexts according to the definition in Section 3.3. $f_p(tag(\mathcal{P}_i)_{i=1..m})$ is a boolean function. The rule $\mathbf{E_{set}}$ evaluates $\{E_1,\ldots,E_m\}$ to a set of contexts.

Examples of Lucx Programs We give two examples.

1. The example models the problem of heat transfer in a solid. There is a metal rod which initially has temperature 0 and whose left-hand end touches a heat source with temperature 100. As the heat is transfered, the temperature at the various points of the rod changes. That is, the temperature depends on the time point and the spatial position on the rod. The following equations illustrate the temperature of the rod as a function of time and space (where k is a small constant related to the physical properties of the rod):

$$\texttt{Temp}_{\texttt{t+1,s+1}} = \texttt{k} \times \texttt{Temp}_{\texttt{t,s}} - (1 - 2\times\texttt{k}) \times \texttt{Temp}_{\texttt{t,s+1}} + \texttt{k} \times \texttt{Temp}_{\texttt{t,s+2}}$$
$$\texttt{Temp}_{\texttt{t,0}} = 100$$
$$\texttt{Temp}_{\texttt{0,s+1}} = 0$$

The Lucx program that models the above equations and queries the temperature at the space 10 at time 10 is the following:

```
Temp @ [Time : 10, Space : 10]
where
  Temp @ [Time : t + 1, Space : s + 1] = k × Temp @ [Time : t, Space : s]
                −(1 − 2 × k) × Temp @ [Time : t, Space : s + 1]
                +k × Temp @ [Time : t, Space : s + 2]
```

```
  Temp@[Time : t, Space : 0] = 100
  Temp@[Time : 0, Space : s + 1] = 0
end
```

2. Consider the problem of finding the solution in positive integers that satisfy the following constraints:

$x^3 + y^3 + z^3 + u^3 = 100$

$x < u$

$x + y = z$

The Lucx program is given below:

```
Eval.B1, B2, B3 (x', y', z', u') = N
where
  N = merge ( merge( merge(x, y), z), u)
      @ B1 ⊠ B2 ⊠ B3;
  where
    merge(x, y) = if (x <= y) then x else y;
    B1 = Box [ X, Y, Z, U | x^3 + y^3 + z^3 + u^3 = 100,
         x ∈ X, y ∈ Y , z ∈ Z , u ∈ U ];
    B2 = Box [ X, U | x < u, x ∈ X , u ∈ U ];
    B3 = Box [ X, Y, Z | x + y = z, x ∈ X ,
         y ∈ Y , z ∈ Z ];
  end
end
```

Implementing Lucx in GIPSY The GIPSY is an Intensional Programming investigation platform under development which allows the automated generation of compiler components for the different variants of the Lucid family of languages [6]. Currently, the compiler for Indexical Lucid, a variant of Lucid, has been implemented successfully in the GIPSY. Lucx is a conservative extension of Lucid. We will provide the Lucx parser and Lucx AST(Abstract Syntax Tree) translator as a Lucx front end to GIPSY. Lucx parser can be automatically generated using the *JavaCC* tool as the Indexical Lucid parser being obtained. In [9], we provide the translation rules for translating Lucx operators into Indexical Lucid operators. Combined with the translation rules for Indexical Lucid operators provided in [7], we achieve a two-pass Lucx AST translator. Once these two models are integrated into GIPSY, the programs written in Lucx will be compiled and run in GIPSY.

5 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.

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 us to use the paradigm for agent communication by allowing the sharing and manipulation of multidimensional contextual information among agents [2]. In addition, the use of the paradigm for real-time reactive programming is shown in [8]. We are developing larger application programs that arise in constraint programming and in context-aware systems [9].

References

1. E. Ashcroft, A. Faustini, R. Jagannathan, W. Wadge. *Multidimensional, Declarative Programming.* Oxford University Press, London, 1995.
2. V. S. Alagar, J. Paquet, K. Wan. *Intensional Programming for Agent Communication.* Proceedings of DALT'04, New York, July 2004, post proceeding will appear in Lecture Notes in Computer Science, Springer-Verlag.
3. A.K.Dey. *Understanding and Using Context.* Personal and Ubiquitous Computing Journal 5(1).pp.4-7.2001.
4. R. V. Guha. *Contexts: A Formalization and Some Applications.* Ph.d thesis, Stanford University, February 10,1995.
5. Cheverst, K., N. Davies, K. Mitchell and C. Efstratiou. *Using Context as a Crystal Ball: Rewards and Pitfalls.* Personal Technologies Journal, Vol. 3 No5, pp. 8-11, 2001.
6. J. Paquet, P. Kropf. *The GIPSY Architecture*. DCW 2000, 144-153.
7. Joey Paquet. *Intensional Scientific Programming* Ph.D. Thesis, Département d'Informatique, Universite Laval, Quebec, Canada, 1999
8. K. Wan, V.S. Alagar, J. Paquet. *Real Time Reactive Programming Enriched with Context.* ICTAC2004, Guiyang, China, September 2004, Lecture Notes in Computer Science,3407,Page 387-402, Springer-Verlag.
9. K. Wan. *Lucx: An Intensional Programming Language Enriched With Contexts*, Ph.d thesis(under preparation), Department of Computer Science, Concordia University, Montreal, Canada, 2005.