Intensional Programming for Agent Communication*

Vasu S. Alagar, Joey Paquet, and Kaiyu Wan

Department of Computer Science
Concordia University
Montreal, Quebec H3G 1M8, Canada
{alagar,paquet,ky_wan}@cs.concordia.ca

Abstract. This article investigates the intensional programming paradigm for agent communication by introducing *context* as a first class object in the intensional programming language *Lucid*. For the language thus extended, a *calculus of contexts* and a *logic of contexts* are provided. The paper gives definitions, syntax, and operators for context, and introduces an operational semantics for evaluating expressions in extended Lucid. It is shown that the extended Lucid language, called Agent Intensional Programming Language(AIPL), has the generality and the expressiveness for being an Agent Communication Language(ACL).
Keywords: Intensional Programming, Context, Lucid, Agent Communication Language, KQML performatives, FIPA

1 Introduction

The goal of this paper is the investigation of Intensional Programming for agent communication by introducing *contexts* as a first class object in the intensional programming language *Lucid* [18]. We provide a *calculus of contexts*, and introduce the *semantics of contexts as values* in the language to add the expressive power required to write non-trivial application programs. We demonstrate that Lucid, extended with contexts, has the generality and the expressibility for being an *Agent Communication Language* (ACL) [6]. We also briefly discuss an implementation framework for agent-based distributed programs written in the extended Lucid.

Intensional programming is a powerful and expressive paradigm based on Intensional Logic. The notion of context is *implicit* in intensional programs, i.e. contexts are not ubiquitous in programs, as in most other declarative or procedural languages. *Intension*, expressed as Lucid programs, can be interpreted to yield values (its *extension*) using demand-driven *eduction* [18]. In this way, intensional programming allows a cleaner and more declarative way of programming without loss of accuracy of interpreting the meaning of programs. Moreover, intensional programming deals with *infinite entities* which can be any ordinary data values such as a stream of numbers, a tree of strings, multidimensional streams, etc. These infinite entities are first class objects in Lucid and functions can be applied to these infinite entities. Information and their computation can be abstracted and expressed declaratively, while providing the support for

* This work is supported by grants from Natural Sciences and Engineering Research Council, Canada

their interpretation in different streams. Such a setting seems quite suitable to hide the internal details of agents while providing them the choice to communicate their internal states, if necessary, for cooperative problem solving in a community of agents. Intensional programming is also suitable for applications which describe the behaviour of systems whose state is changing with time, space, and other physical phenomena or external interaction in multidimensional formats. Agent communication where intensions of agents have to be conveyed is clearly one such application.

The notion of *context* was introduced by McCarthy and later used by Guha [7] 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. The major distinction between contexts in AI and in intensional programming is that in the former case they are *rich objects* that are not *completely expressible* and in the later case they are *implicitly* expressible, i.e. one can write Lucid expressions whose evaluation is context-dependent, but where the context is not explicitly manipulated. In extending Lucid we add the possibility to explicitly manipulate contexts, and introduce contexts as first class objects. That is, contexts can be declared, assigned values, used in expressions, and passed as function parameters. In this paper we give the syntax for declaring contexts, and a partial list of operators for combining contexts into complex expressions. A full discussion on the syntax and semantics of the extended language appears in [1]. The ACL that we introduce in this paper uses context expressions in messages exchanged between communicating agents. The structure of message is similar to the structure of performatives in KQML [5].

The paper is organized as follows: In Section 2 we review briefly the intensional programming paradigm. Section 3 discusses the basic operators of Lucid and illustrates the style of programming and evaluation in Lucid with simple examples. In Section 4 we discuss software agents and communication language for agents as standardized by FIPA [6]. We discuss the extended Lucid language for agent communication as well. The GIPSY [13], which provides a platform for implementation of extended Lucid is briefly discussed in Section 5.

2 Intensional Programming Paradigm

Intensional Logic came into being from research in natural language understanding. According to Carnap, the real meaning of a natural language expression whose truth-value depends on the context in which it is uttered is its *intension*. The *extension* of that expression is its actual truth-value in the different possible contexts of utterance [14], i.e. the different *possible worlds* into which this expression can be evaluated. Hence the statement *"It is snowing"* has meaning in itself (its intension), and its valuation in particular contexts (i.e. its extention) will depend on each particular context of evaluation, which includes the exact time and space when the statement is uttered.

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 placeholders 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 for this month here is greater than $0^\circ C$.

This expression is intensional because the truth value of this expression depends on the context in which it is evaluated. The two intensional operators in this expression are *this month* and *here*, which refer respectively to the time and space dimension. If we "freeze" the space context to the city of Montreal, we will get the yearly temperature at this space context, for an entire particular year (data is freely given by the authors). So along the time dimension throughout a particular year, we have the following valuation for the above expression, with T and F respectively standing for *true* and *false*, where the time dimension tags are the months of the year :

$E =$

Ja	Fe	Mr	Ap	Ma	Jn	Jl	Au	Se	Oc	No	De
F	F	F	F	T	T	T	T	T	F	F	F

So the intension is the expression E itself, and a part of its extension related to this particular year is depicted in the above table. According to Carnap, we are restricting the possible world of intensional evaluation to Montreal, and extending it over the months of a particular year. Furthermore, the intension of E can be evaluated to include the spatial dimension, in contrast with the preceding, where space was made constant to Montreal. Doing so, we extend the possible world of evaluation to the different cities in Canada, and still evaluate throughout the months of a particular year. The extension of the expression varies according to the different cities and months. Hence, we have the following valuation for the same 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 Lucid intensional programming language 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* [16].

3 Lucid

Lucid was invented as a tagged-token dataflow language by William Wadge and Edward Ashcroft [18]. In the original version of Lucid, the basic intensional operators were *first*, *next*, and *fby*. The following is the definition of three popular operators of the original Lucid. [14]:

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

(1) $\underline{\texttt{first}}\ X \stackrel{\text{def}}{=} (x_0, x_0, \ldots, x_0, \ldots)$
(2) $\underline{\texttt{next}}\ X \stackrel{\text{def}}{=} (x_1, x_2, \ldots, x_{i+1}, \ldots)$
(3) $X\ \underline{\texttt{fby}}\ Y \stackrel{\text{def}}{=} (x_0, y_0, y_1, \ldots, y_{i-1}, \ldots)$

Clearly, analogues can be made to list operations, where `first` corresponds to `hd`, `next` corresponds to `tl`, and `fby` corresponds to `cons`. The following example 1 is a simple example of Lucid, which expresses the *infinite sequence* of natural numbers, which is $(0, 1, 2, 3, \ldots)$:

Example 1

```
N
where
    N = 0 fby (N+1);
end
```

Lucid has eventually gone through several generalization steps and has evolved into a multidimensional intensional programming language which enables functions and dimensions as first-class values [15]. To support this, two basic intensional operators are added, which are used respectively for intensional navigation (@) and for querying the current context of evaluation of the program (#). Doing this, the Lucid language went apart from its dataflow nature to the more general intensional programming paradigm (often referred to as *multidimensional indexical paradigm*).

The following example 2 is to extract a value from the stream representing the natural numbers, beginning from the ubiquitous number 42. We arbitrarily pick the third value of the stream, which is assigned tag number two (indexes starting at 0). We also set the stream's variance in the *d* dimension.

Example 2

```
N @.d 2
where
     dimension d;
     N = 42 fby.d (N+1);
end;
```

Intuitively, we can expect the program to return the value 44. To see how the program is evaluated, we rewrite it in terms of the basic @ and # intensional operators. The translation rules used for the rewriting of the program are presented in [14]. It is also interesting to note that Lucid forms a family of languages, and that we have identified a generic form (the one presented in this paper) into which all the other languages can be syntactically translated without loss of meaning.

```
N @.d 2
  where
      dimension d;
      N = if (#.d <= 0)  then 42 else (N+1) @.d (#.d-1);
  end;
```

The implementation technique of evaluation for Lucid programs is an interpreted mode called *eduction*. Eduction can be described as *tagged-token demand-driven dataflow*, in which data elements (tokens) are computed on demand following a dataflow

network defined in Lucid. Data elements flow in the normal flow direction (from producer to consumer) and *demands* flow in the reverse order, both being *tagged* with their current context of evaluation.

Evaluation takes place by generating successive demands for the appropriate values of `N` in different contexts, until the final computation can be effected. The demand for `N @.d 2` generates a demand for `N @.d 1` which in turn generates a demand for `N @.d 0`. The definition of the program explicitly states that the value of `N @.d 0` is 42. Once this is found, the successive addition operations are made on the demand results, as required by the equation `N = 42 fby.d N+1`, giving a final result of 44. For an in-depth description of the syntax and semantics of the language, see Section 4.4.

Lucid has been extended in several ways. Its variants have been used to specify 3D spreadsheets [17], real-time systems using Lustre (a variant of Lucid) [3], database systems [17] and GLU (Granular Lucid) run-time system which illustrates how the multidimensional structure of a problem expressed in Lucid can be harnessed to produce efficient parallel implementations of problems [8]. Currently, we are in the process of implementing the GIPSY (General Intensional Programming System), which is an investigation platform (compiler, run-time environment, etc) for all members of the Lucid family of intensional programming languages [13].

4 Agent Communication in Intensional Programming Language

Software agents, according to Chen et al [4], are personalized, continuously running and semi-autonomous, driven by a set of beliefs, desires, and intentions (BDI). Agent technology is being standardized by FIPA [6] with the goal of seamlessly integrating their architectures and languages with various commercial application systems such as *network management*, *E-commerce*, and *mobile computing* [12]. In such applications agents should have capabilities to exchange complex objects, their intentions, shared plans, specific strategies, business and security policies. An Agent Communication Language (ACL) must be declarative and have a small number of primitives that are necessary to construct the structures required for achieving the above capabilities.

4.1 KQML and FIPA Languages

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) [5] and the FIPA [6] communication language. The FIPA language includes the basic concepts of KQML, yet they have slightly different semantics. We summarize below the major points of contrasts between KQML and FIPA ACL, from the work of Labrou, Finin, and Peng [11].

KQML has a *predefined* set of *reserved performatives*. It is neither a minimal required set nor a closed set. That is, an agent may use only those primitives that it needs in a communication, and a community of agents may agree either to use the union of the sets of primitives required by each one of them or use some additional performatives with a consensus on the semantics and protocols for using them. In the latter case, it is not clear as to how the agents will construct the additional performatives and how a

semantics can be dynamically worked out. As an example of the former case, a KQML message representing a query about the price of a share of IBM stock might be encoded as follows [5]:

```
( askone
 :content (PRICE IBM ?price)
 :receiver STOCK-SERVER
 :language LPROLOG
 :ontology NYSE-TICKS )
```

Fig. 1. The `ask-one` performative of KQML

In this message, the KQML performative is `ask-one`, the *content* of the message is `PRICE IBM ?price`, the *ontology* assumed by the query is identified by the token `NYSE-TICKS`, the *receiver* of the message is to be a server identified as `STOCK-SERVER` and the *query* is written in a language called `LPROLOG`. KQML also provides a small number of performatives that the agents can use to define meta data. A semantics of KQML in a style similar to Hoare logic is given in [9], [10].

The syntax of the FIPA ACL resembles KQML, however its semantics is formally given by a quantified multi-modal logic [19]. The communication primitives in FIPA ACL are called *communicative acts* (CA), yet they are the same as KQML primitives. The semantics of the FIPA ACL is given in the formal language SL, which provides the modal operators for beliefs (B), desires (D), intentions (persistent goals PG), and uncertain goals (U). Actions of objects, object descriptions, and propositions can be described in the language. Each formula in SL defines a constraint that the sender of the message must satisfy in order for the sender to conform to the FIPA ACL standard.

In order to achieve cooperation and interoperability, both KQML and FIPA ACL need to predefine a set of performatives, which is neither a minimal required set nor a closed one. This creates a big problem for maintaining and extending the agents to face the fast evolution of performatives. However, if we design the communication language from a higher level and in a more abstract way in which the performatives become *first class objects*, we will be able to create additional performatives as contextual expressions. In the AIPL, which we discuss next, we define contexts as first class objects and encapsulate performatives in them. We define operators on contexts, that can be used to create new contexts from existing contexts. Informally, when an agent A sends a communicative act $CA\ x$ to an agent B, we view x as a collection (may be a sequence) of objects, where each object is bound to some description on its interpretation, evaluation criteria, temporal properties, constraints, and any other information that can be encoded in the language. We view this collection as a context.

4.2 Contexts in AIPL

The approach of using intensional programming for agent communication is to make a conservative extension of Lucid by introducing context as a first class object in Lucid [1]. In our approach, the name of a performative is considered as an expression,

and the rest of the performative constitute a *context* which can be understood as a *communication context*; each field except the name in the message is 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.

Definitions of Contexts in AIPL In extended Lucid contexts are defined as *a subset of a finite union of relations*. Let $DIM = \{d_1, d_2, \ldots, d_n\}$ denote a finite set of dimension names. With each dimension, a unique domain is associated. A domain is a set, finite or infinite, of values. For instance, a domain may be $\mathbf{N}$, the set of natural numbers, or $\mathbf{R}$, the set of real numbers, or any arbitrary set of named objects. Let $DOM = \{D_1, D_2, \ldots, D_m\}$ denote a finite set of domains. There exists a function $f_{dimtodom} : DIM \rightarrow DOM$, which maps each $d_i \in DIM$ to a unique domain $f_{dimtodom}(d_i)$ in DOM.

Definition 2 *Consider the relations*

$$P_i = \{d_i\} \times f_{dimtodom}(d_i) \quad 1 \leq i \leq n$$

A context C, given $(DIM, f_{dimtodom})$, is a finite subset of $\bigcup_{i=1,n} P_i$. The degree *of the context is* $\mid DIM \mid$.

A context is written using *enumeration* syntax. The set enumeration syntax of a context C is

$$C = \{(d_i, x_j) \mid d_i \in DIM, x_j \in f_{dimtodom}(d_i)\}$$

and the syntax used in extended Lucid is

$$[d_{i_1} : x_{j_1}, \ldots, d_{i_k} : x_{j_k}].$$

If C is a context over $(DIM, f_{dimtodom})$, it is true that

$$C \subseteq \bigcup_{i=1,n} P_i \subset DIM \times D, \quad D = \bigcup_{i=1}^{m} D_i$$

Consequently, every subset of $\bigcup_{i=1,n} P_i$ is a context, but not every subset of $DIM \times D$ is a context. However, if $D_1 = D_2 \ldots, = D_n$, every subset of $DIM \times D$ is a context. We say a context C is *simple* (s_context), if $[x_i, y_i], [x_j : y_j] \in C \Rightarrow x_i \neq x_j$. A simple context C of degree 1 is called a *micro* (m_context) context.

Example 3 *Let $DIM = \{X, Y, Z, U\}$, $D = \{\mathbf{N}, \mathbf{R}, \mathbf{Q}\}$, $f_{dimtodom}(X) = \mathbf{N}$, $f_{dimtodom}(U) = \mathbf{N}$, $f_{dimtodom}(Y) = \mathbf{R}$, and $f_{dimtodom}(Z) = \mathbf{Q}$.*

1. $C_1 = [X : 1.5, Y : 2]$ *is not a valid context.*
2. $C_2 = [Z : \frac{4}{5}]$ *is a m_context.*
3. $C_3 = [X : 3, Y : \frac{3}{2}]$ *is a s_context.*
4. $C_4 = [X : 3, X : 4, Y : 3, Y : 2.35, Z : \frac{16}{17}]$ *is a context.*

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

Definition 3 *Let M denote a set of m_contexts. We define functions*

$$dim_m : M \rightarrow DIM \quad tag_m : M \rightarrow TAG_m,$$

where $TAG_m = \bigcup_{m \in M} tag_m(m)$, such that for $m = [x, y] \in M$, $dim_m(m) = x$, and $tag_m(m) = y \in f_{dimtodom}(dim_m(m))$.

Definition 4 *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.*

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

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

Example 4 *Consider the contexts introduced in Example 3. An application of dim and tag functions to these contexts produces the following results:*

1. *dim and tag are not defined for context C_1.*
2. *$dim_m(C_2) = Z$, $tag_m(C_2) = \frac{4}{5}$.*
3. *$dim(C_3) = \{X, Y\}$, $tag(C_3) = \{3, \frac{3}{2}\}$.*
4. *$dim(C_4) = \{X, Y, Z\}$, $tag(C_4) = \{3, 4, 2.35, \frac{16}{17}\}$.*

In general, a set of contexts may include contexts of different degrees. We use the syntax $Box[\Delta \mid p]$ to introduce a finite set of contexts in which all contexts are defined over $\Delta \subseteq DIM$ and have the same degree $\mid \Delta \mid$.

Definition 5 *Let $\Delta = \{d_{i_1}, \ldots, d_{i_k}\}$, where $d_{i_r} \in DIM$ $r = 1, \ldots, k$, and p is a k-ary predicate defined on the tuples of the relation $\Pi_{d \in \Delta} f_{dimtodom}(d)$. 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 $(x_{i_1}, \ldots, x_{i_k})$, $x_{i_r} \in f_{dimtodom}(d_{i_r})$, $r = 1, \ldots k$ satisfy the predicate p introduces a set S of contexts of degree k. For each context $s \in S$ the values in $tag(s)$ satisfy the predicate p.

Example 5 *Let $prime(x)$ be the predicate that is true when $x \in \mathbf{N}$ is true.*

1. *The declaration $Box[X \mid prime(x)]$, where $f_{dimtodom}(X) = \{2, 3, 4, \ldots, 118\}$ introduces the set of m_contexts $\{m = [X : x] \mid prime(x) \wedge x \in \mathbf{N} \wedge 2 \leq x \leq 118\}$*
2. *The set of contexts defined by*

   $$Box[X, U \mid \tfrac{x}{4} + \tfrac{u}{5} \leq 1, x \in, u \in U],$$

   $f_{dimtodom}(X) = f_{dimtodom}(U) = \mathbf{N}$ is given by
   $\{[X : 0, U : 0], [X : 0, U : 1], [X : 0, U : 2], [X : 0, U : 3],$
   $[X : 0, U : 4], [X : 0, U : 5], [X : 1, U : 0], [X : 1, U : 1],$
   $[X : 1, U : 2], [X : 1, U : 3], [X : 2, U : 0], [X : 2, U : 1],$
   $[X : 2, U : 2], [X : 3, U : 0], [X : 3, U : 1], [X : 4, U : 0]\}$

4.3 Context Calculus

We provide a set of operators which can be applied on contexts to produce many kinds of contexts according to the requirements of different applications. These operators include: *constructor* $[\_ : \_]$, *override* $\oplus$, *difference* $\ominus$, *choice* $|$, *conjunction* $\sqcap$, *disjunction* $\sqcup$, *undirected range* $\rightleftharpoons$, *directed range* $\rightharpoonup$, *projection* $\downarrow$, *hiding* $\uparrow$, *substitution* $/$, and *comparison* $=, \supseteq, \subseteq$. The language allows user defined functions on contexts. The definitions, properties, and examples of these operators are discussed in [1]. The following are the definitions and examples of some of them.

Definition 6 *Constructor operator constructs a* $m\_context$ *for a given dimension* d*, and domain* $f_{dimtodom(d)}$*:*

$$[\_ : \_] : d \times f_{dimtodom}(d) \rightarrow M,$$

$[d : t] = m \in M$*. Using the set notation and the definitions for contexts, we construct contexts.*

Definition 7 *Override operator takes two contexts* $c_1, c_2 \in G$*, and returns a context* $c \in G$*, which is the result of the conflict-free union of* c_1 *and* c_2*, as defined below:*

$$\_ \oplus \_ : G \times G \rightarrow G,$$

$$c = c_1 \oplus c_2 = \{ m \mid (m \in c_1 \wedge \neg m \in c_2) \vee m \in c_2\}$$

Example 6 *Override operator: Let* $c_1 = [d : 1]$, $c_2 = [e : 2]$, $c_3 = [e : 5]$, $c_4 = [d : 2 , e : 5 , f : 4]$, $c_5 = [d : 2 , d : 3 , f : 4]$,
Then
$c_1 \oplus c_2 = [d : 1 , e : 2]$,
$c_2 \oplus c_3 = [e : 5]$,
$c_3 \oplus c_2 = [e : 2]$,
$c_4 \oplus (c_1 \oplus c_2) = [d : 1 , e : 2 , f : 4]$,
$(c_4 \oplus c_1) \oplus c_2 = [d : 1 , e : 2 , f : 4]$,
$c_5 \oplus (c_1 \oplus c_2) = [d : 1 , d : 2 , e : 2 , f : 4]$,
$(c_5 \oplus c_1) \oplus c_2 = [d : 1 , d : 2 , e : 2 , f : 4]$.

Definition 8 *Difference operator is similar to the set difference operator:*

$$\_ \ominus \_ : G \times G \rightarrow G,$$

$$c = c_1 \ominus c_2 = \{ m \mid m \in c_1 \wedge \neg m \in c_2 \}$$

Example 7 *Difference operator: Let* $c_1 = [d : 1, d : 2, e : 3]$,
$c_2 = [d : 1, e : 4]$, $c_3 = [d : 1]$,
Then
$(c_3 \ominus c_2) \ominus c_1 = \varnothing$
$c_3 \ominus (c_2 \ominus c_1) = [d : 1]$.

Definition 9 *Choice operator accepts a finite number of $c_1, \ldots, c_k$ of contexts and nondeterministically returns one of the c_is. The definition $c = c_1 \mid c_2 \mid \ldots, \mid c_k$ implies that c is one of the c_i, where $1 \leq i \leq k$:*

$$\_ \mid \_ : G \times G \times \ldots \times G \rightarrow G,$$

Example 8 *Choice operator:*
Let $c_1 = [\, e : 2,\ d : 1\,],\ c_2 = [\, d : 1\,],\ c_3 = [d : 3\,], c_4 = c_1 \mid c_2 \mid c_3$,
Then $c_4 = c_1 = [\, e : 2,\ d : 1\,]$ *or*
$c_4 = c_2 = [\, d : 1\,]$ *or*
$c_4 = c_3 = [\, d : 3\,]$.

Definition 10 *Hiding operator enables a set of dimensions D to be applied on a context $c \in G$, and the result removes all the m_contexts in c whose dimensions are in D:*

$$\_ \uparrow \_ : G \times D \rightarrow G,$$

$$c \uparrow D = \{(\, d,\ t) \in c \wedge \neg\, d \in D\}$$

Example 9 *Hiding operator:*
Let $c_1 = [\, d : 1,\ e : 4,\ f : 3\,],\ c_2 = [\, d : 3\,],\ c_3 = [\, f : 3\,],\ D = \{\, d,\ e\,\}$
Then $c_1 \uparrow D = [\, f : 3\,]$
$c_2 \uparrow D = \varnothing$
$c_3 \uparrow D = [\, f : 3\,]$

In order to provide a precise meaning for a context expression, we define the precedence rules for all the operators. The precedence rules for the operators are shown in Figure 2 (from the highest precedence to the lowest). Parentheses will be used to override this precedence when needed. Operators having the same precedence will be applied from left to right.

1. $\downarrow, \uparrow, /$
2. $\mid$
3. $\sqcap, \sqcup$
4. $\oplus, \ominus$
5. $\rightleftharpoons, \rightharpoonup$
6. $=, \subseteq, \supseteq$

Fig. 2. Precedence Rules for Operators

As an illustration, consider the context expression $c_1 \mid c_2 \oplus c_3 \uparrow D$. Applying the precedence rules, this expression is equivalent to $(c_1 \oplus (c_3 \uparrow D)) \mid (c_2 \oplus (c_3 \uparrow D))$.

4.4 Syntax and Semantics of Extended Lucid

The abstract syntax of the extended Lucid is defined in Figure 3. The operator @ is the navigation operator, which evaluates an expression E in context E', where E' is an expression evaluating to a context. The operator # is the context query operator, operating on the current evaluation context. The non-terminals E and Q respectively refer to *expressions* and *definitions*. The only change applied to the syntax of the language in order to achieve contexts as first class objects comes in the syntactic rules presented in bold. The older syntax for the @ operator was of the form: E @ $E'E''$ where, semantically speaking, E' evaluated to a dimension, and E'' evaluated to a dimension tag (as depicted in its semantic rule presented in Figure 5). In fact, the $E'E''$ part of this syntactic construct is representing a $m\_context$, even though E' and E'' were evaluated as separate semantic entities, and not to a context. In contrast, the E' part of the new E @ E' semantically evaluates to a $m\_context$, thus introducing contexts as first class objects. The syntactic construct $[E_1 : E'_1, \ \ldots, \ E_n : E'_n]$ is representing how $s\_contexts$ are syntactically introduced in the language. The E' part of the E @ E' rule shall be eventually evaluating to something of this form, as is reflected in the $\mathbf{E_{at(c)}}$ and $\mathbf{E_{context}}$ semantic rules. As for the operational semantics of Lucid, the general form of evaluating in Lucid is as following:

$$
\begin{aligned}
E ::= & \ \mathit{id} \\
& \mid E(E_1, \ldots, E_n) \\
& \mid \texttt{if } E \texttt{ then } E' \texttt{ else } E'' \\
& \mid \#E \\
& \mid \mathbf{E} \ @ \ \mathbf{E'} \\
& \mid [\mathbf{E_1} : \mathbf{E'_1}, \ \ldots, \ \mathbf{E_n} : \mathbf{E'_n}] \\
& \mid E \texttt{ where } Q \\
Q ::= & \texttt{ dimension } \mathit{id} \\
& \mid \mathit{id} = E \\
& \mid \mathit{id}(\mathit{id}_1, \ldots, \mathit{id}_n) = E \\
& \mid Q \ Q
\end{aligned}
$$

Fig. 3. Abstract syntax for the Extended Lucid

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

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. It is therefore a partial function

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

where $\mathbf{Id}$ is the set of all possible identifiers and $\mathbf{IdEntry}$ has five possible kinds of value such as: *Dimensions*, *Constants*, *Data Operators*, *Variables*, and *Functions* [14].

The evaluation context $\mathcal{P}$, associates a tag to each relevant dimension. It is therefore a partial function:

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

The complete operational semantics is defined in Figure 4 [14]. The rule for the navigation operator is $\mathbf{E_{at(c)}}$, which corresponds to the syntactic expression $E@E'$, evaluates E in context E'. The function $\mathcal{P}' = \mathcal{P} \dagger [id \mapsto v'']$ means that $\mathcal{P}'(x)$ is v'' if $x = id$, and $\mathcal{P}(x)$ otherwise. For example, the evaluation of the expression $E@E_1 \oplus E_2 \ominus E_3$ is done in the following order:

- compute $E' = E_1 \oplus E_2$
- compute $E'' = E' \ominus E_3$
- evaluate $E@E''$

4.5 Message Structure and Evaluation in AIPL

The syntax of a message in AIPL is $\langle E, E' \rangle$, where E is the message name and E' is a context. The message name in a Communicative Act CA of FIPA ACL or the name of a performative in KQML is captured in AIPL by E. In an implementation E corresponds to a function. The context E' includes all the information that an agent wants to convey in an interaction to another agent. Thus, a query from an agent A to an agent B is of the form $\langle E_A, E'_A \rangle$. A response from agent B to agent A will be of the form $\langle E_B, E''_B \rangle$, 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.

Query Evaluation The operational semantics in extended Lucid is the basis for query evaluation in AIPL. The query from agent A $\langle E_A, E'_A \rangle$ to agent B is evaluated as follows:

- agent B obtains the context $F_B = E'_A \oplus L_B$, where L_B is the local context for B.
- agent B evaluates $E_A@F_B$
- 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
- sends the response $\langle E_B, E''_B \rangle$ to agent A.

For example, the query in Figure 1 is represented in AIPL as the expression E @ E', $E' = E_1 \oplus E_2 \oplus E_3 \oplus E_4$.

```
E @ [ E1 ⊕ E2 ⊕ E3 ⊕ E4 ]
where
E = "ask-one";
E1 = [ content : (PRICE IBM ?price)];
E2 = [ receiver: STOCK-SERVER ];
E3 = [ language: LPROLOG ];
E4 = [ ontology: NYSE-TICKS ];
end
```

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

$$\mathbf{E_{opid}}: \frac{\mathcal{D}(id) = (\texttt{op}, f)}{\mathcal{D}, \mathcal{P} \vdash id : id} \qquad \mathbf{E_{fid}}: \frac{\mathcal{D}(id) = (\texttt{func}, id_i, E)}{\mathcal{D}, \mathcal{P} \vdash id : id}$$

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

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

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

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

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

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

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

$$\mathbf{E_{context}}: \frac{\mathcal{D}, \mathcal{P} \vdash E_{d_j} : id_j \qquad \mathcal{D}(id_j) = (\texttt{dim}) \qquad \mathcal{D}, \mathcal{P} \vdash E_{i_j} : v_j \qquad v = \mathcal{P}\dagger[id_j \mapsto v_j]}{\mathcal{D}, \mathcal{P} \vdash [E_{d_1} : E_{i_1}, E_{d_2} : E_{i_2}, \ldots, E_{d_n} : E_{i_n}] : v}$$

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

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

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

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

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

Fig. 4. Semantic rules for Lucid

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

Fig. 5. Semantic rule for for the old @ operation

The implementation will assure that the local context of B is sufficient to evaluate the query and respond to A within an acceptable time delay. This is an important issue because we want the agents to be reactive (responds within acceptable time limits) while the eduction is allowed to continue. The choice operator helps in achieving such a goal. For example, the query:

```
E @ [ E1 ⊕ E2 ⊕ E3 | E4 ⊕ E5 ]
where
E = "ask-one";
E1 = [ content : (PRICE IBM ?price)];
E2 = [ receiver: STOCK-SERVER ];
E3 = [ language: LPROLOG ];
E4 = [ language: STANDARD_PROLOG ];
E5 = [ ontology: NYSE-TICKS ];
end
```

gives the receiver, depending on its local context, choose either LPROLOG or STANDARD_PROLOG to ensure timeliness. The fields in the performative in Figure 1 can not be dynamically changed in either FIPA or KQML. In our language, we form the context expression $E'' = E' \uparrow \{language\} \oplus [language : Java]$ to dynamically replace the language requirement and construct a new query.

In general, an interaction between agents will be a *conversation*, which can be expressed as a sequence, possibly infinite, of messages. That is, a conversation is

$$\langle (\alpha_1, \beta_1);\ \ldots;\ (\alpha_k, \beta_k) \rangle,$$

where $\alpha_i = \langle E_{iA}, E'_{iA} \rangle$, and $\beta_i = \langle E_{iB}, E'_{iB} \rangle$. A conversation is evaluated by evaluating each pair (α_i, β_i) in the sequence according to the above semantics. A conversation among an agent group, a finite set of simultaneously interacting agents, is handled by combining the evaluation mechanisms described above.

In Lucid, a conversation can be represented as *dimension streams* whose values are dimensions. An agent can convey a *plan* to another agent by annotating its messages with different data structures such as a stream of numbers, two-dimensional tables, tree of strings, and multidimensional objects. When an expression is evaluated by the compiler, if the compiler meets the @, the compiler first interprets all the contexts into a context using the operators provided by the expression, then use this context to evaluate the expression. If the expression is reducible to the original form of Lucid, the expression can directly be evaluated by the original compiler for Lucid, which has been already implemented by GIPSY [13]. The interface that we plan to build in GIPSY will handle the case when the expression is not directly reducible to standard Lucid form.

5 Conclusion

The Agent Communication Language AIPL that we have introduced in this paper has a number of advantages:

– In KQML and FIPA, performatives, other than the primitive performatives defined in the language, can be agreed upon by the community of agents involved in a collaboration. That is, interoperability is proved. However, performatives are only static status and not first class objects in the language. As a consequence, performatives can not be changed dynamically, nor can they be used as a vehicle to communicate local state information of agents. In AIPL, by making context as first class objects, we have removed the above limitations. In addition, we can define functions on contexts and they can be used as parameters in programs. Thus, we have enhanced both *interoperability* and *flexibility* in agent communication.
– AIPL is declarative and has a formal semantics.
– AIPL uses multidimensional streams of objects, which can be used to represent plans and conversations in multiple streams.
– *Multiple formats of communication* can be supported since intensional programming language deals with any kind of ordinary data type. Even the multimedia streams between agents become feasible.

In our ongoing research, we are formalizing the semantics for plans and conversations. We are developing a logic of contexts and proof rules for reasoning about programs written in AIPL. Different variants of the Lucid family of languages have been implemented for various purposes and application domains over the years. Lately, we have undertaken the development of the GIPSY (General Intensional Programming System) that is an integrated programming language investigation platform allowing the automated generation of compiler components for the different variants of the Lucid family of languages [13, 20, 21]. The GIPSY is designed as a framework in order to reach for maximal flexibility and generality of application.

Being a functional language, Lucid programs can be evaluated in parallel or distributed execution mode. In such case, in order to augment the granularity of parallelism, GIPSY programs can be written as hybrid programs, allowing Java functions to be called by the Lucid part of the program. Interestingly, these Java functions can actually be the implementation of software agents. Then the Lucid part becomes a declarative specification describing the relationships between agents, implicitly describing how these agents are collaborating in a distributed execution. The AIPL described in this paper is then used as a formal ACL in order to achieve transparent contextual communication between agents. The semantics of the calculus of contexts being intrinsic to each agent through the eduction engine embedded in each node, there is no need to write agents that embed a parser and semantic analyzer and translator for the ACL primitives that are exchanged between agents at run time.

Based on this system, communication between different categories of agents such as *interface agent*, *middle agent*, *task agent*, and *security agent* [2] can be used as case studies for AIPL. We will also investigate the use of AIPL for mobile agents communication and multimedia communication between agents.

References

1. V.S.Alagar, Joey Paquet, Kaiyu Wan. *Contexts in Intensional Programming.* (in preparation) Technical Report, Department of Computer Science, Concordia University, Montreal, Canada, April 2004.

2. V.S.Alagar, J.Holliday, P.V.Thiyagarajan, B.Zhou. Agent Types and Their Formal Descriptions. Technical Report, Department of Computer Engineering, Santa Clara University, Santa Clara, CA, U.S.A., May 2002.
3. P.Caspi, D,Pilaud, N.Halbwachs, J.A.Plaice. *LUSTRE: A declarative language for programming synchronous systems.* P.O.P.L. 1987
4. Q.Chen, M.Hsu, U.Dayal, M.Griss. *Multi-Agent Cooperation, Dynamic Workflow and XML for E-Commerce Automation.* Proceedings Autonomous Agents 2000, June, Barcelona, Spain, June 2000.
5. Tim Finin, Richard Fritzson, Don McKay, Robin McEntire. *KQML as an Agent Communication Language.* Proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM'94), ACM Press, November 1994.
6. FIPA Semantic Language Specification. *FIPA Specification repository*, FIPA-specification identifier XC00008G, September 2000 Foundation for Intelligent Physical Agents, Geneva, Switzerland.
7. R. V. Guha. *Contexts: A Formalization and Some Applications.* Ph.d thesis, Stanford University, February 10,1995.
8. R. Jagannathan, C. Dodd, and I. Agi. *GLU: A High-Level System for Granular Data-Parallel Programming.* Concurrency: Practice and Experience, vol. 9,1997.
9. Y. Labrou. *Semantics for an Agent Communication Language.* Doctoral Dissertation, Computer Science and Electrical Engineering Department, University of Baltimore, Baltimore County, 1996.
10. Y. Labrou, T. Finin. *Semantics for an Agent Communication Language.* Agent Theories, Architectures, and Languages IV, M. Woodridge, J.P. Muller, and M. Tambe, eds., Lecture Notes in Artificial Intelligence, Springer-Verlag, Berlin, 1998.
11. Y. Labrou, T. Finin, and Y. Peng. *Agent Communication Languages: The Current Landscape.* IEEE Journal on Intelligent Agents, Amrch/April 1999, pp. 45-52.
12. A. Lingnau, O. Drobink. *AN INFRASTRUCTURE FOR MOBILE AGENTS: REQUIREMENTS AND ARCHITECTURE.* Johann Wolfgang Goethe University, Frankfurt am Main, Germany, 1995.
13. Joey Paquet, Peter Kropf. *The GIPSY Architecture.* DCW 2000: 144-153
14. Joey Paquet. *Intensional Scientific Programming.* Ph.D. Thesis, Departement d'Informatique, Universite Laval, Quebec, Canada, 1999
15. Joey Paquet and John Plaice. *Dimensions and Functions as Values.* Proceedings of the Eleventh International Symposium on Lucid and Intensional Programming, Sun Microsystems, Palo Alto, California, USA, May 1998.
16. John Plaice and Joey Paquet. *Introduction to Intensional Programming.* In Intensional Programming I, pages 1-14. World Scientific, Singapore, 1996
17. P.Rondogiammis, William Wadge. *Intensional Programming Languages.* Proceedings of the First Panhellenic Conference on New Information Technologies, 1998
18. W.W.Wadge, E.A.Ashcroft. *Lucid, the dataflow programming language.* Academic Press, 1985
19. Michael Wooldridge. *Verifiable Semantics for Agent Communication Languages.* Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS'98).
20. Ai Hua Wu, Joey Paquet and Peter Grogono. *Design of a compiler framework in the GIPSY system.* In Parallel and Distributed Computing and Systems - PDCS 2003, Marina Del Rey, California, USA, 2003.
21. Ai Hua Wu and Joey Paquet. *Translator generation in the general intensional programming complier.* In Eighth International Conference on Computer Supported Cooperative Work in Design (CSCW2003). XiaMen, P.R. of China. May 24-28, 2004.