DESIGN AND IMPLEMENTATION OF CONTEXT CALCULUS IN THE GIPSY

Xin Tong

A thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

April 2008

© Xin Tong, 2008

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-40955-8
Our file Notre référence
ISBN: 978-0-494-40955-8

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Abstract

Design and Implementation of Context Calculus in the GIPSY

Xin Tong

The Lucid programming language is a family of intensional programming languages supported by the General Intensional Programming System (GIPSY). Among all the Lucid variants, the notion of *context* is a core concept. After Lucx, a Lucid variant introduced by Wan in her PhD thesis, contexts can be declared explicitly and manipulated directly as first-class values. Lucx also enables a set of *context calculus* operators performed on contexts. Such new concepts have greatly contributed to the evolution of Lucid.

Upon these theoretical foundations, this thesis presents the integration into the GIPSY of Lucx's *context* and *context calculus* as defined in Wan's PhD thesis. We first provide the construction of Lucx's compiler, including building the parser and extending the existing semantic analyzer. Then we present how the concept of context has been abstracted into a new type variant of the GIPSY type system. After that, we specify how the context calculus operators have been implemented within the context type. And finally, we demonstrate our testing strategies on each component of Lucx to justify all the implementation efforts.

Acknowledgments

First, I would like to express my sincere gratitude to my supervisor Dr. Joey Paquet for guiding me into this interesting field, inspiring me with his insightful ideas, providing me everlasting support and encouraging me to overcome all the difficulties in my graduate study.

I also would like to thank my team members for their selfless help and invaluable cooperation.

Especially, I would like to give my special thanks to my family and friends both here and in China. I am forever indebted to my parents who have not only brought me to this world but also educated me how to find the meaning of life. Without their unconditional love, nothing of this would ever happen.

Contents

List of Figures viii

1 Introduction 1

1.1 Problem Statement . . . 1

1.2 Proposed Solution . . . 2

1.3 Contributions . . . 2

1.4 Structure of the Thesis . . . 3

2 Background 4

2.1 Intensional Logic and Lucid . . . 4

2.2 Lucid Programming Languages . . . 5

2.3 GIPL, and Lucx as a SIPL . . . 7

2.3.1 Syntactical Extension . . . 11

2.3.2 Semantic Extension . . . 11

2.4 Overview of GIPSY Framework Architecture . . . 12

2.4.1 Compilation: GIPC . . . 13

2.4.2 Execution: GEE . . . 16

2.5 Related Work . . . 19

2.6 Summary . . . 21

3 Theoretical Basis for Implementing Context Calculus 22

3.1 Context Types . . . 22

3.1.1 Simple Context . . . 23

3.1.2 Context Set . . . 23

3.2 Context Calculus Operators . . . 24

3.3 Tag Set Types . . . 28

3.3.1 Ordered Finite Tag Set 31
3.3.2 Ordered Infinite Tag Set 32
3.3.3 Unordered Finite Tag Set 33
3.3.4 Unordered Infinite Tag Set 33
3.4 Summary 34

4 Implementation **35**
4.1 Integration of Lucx Compiler 35
4.1.1 Lucx Parser 35
4.1.2 Extension of Semantic Analyzer 39
4.2 Design and Implementation of Context Classes 41
4.2.1 An Overview of the GIPSY Type System 42
4.2.2 Design of Context Classes 43
4.3 Implementing Context Calculus in the GIPSY 44
4.3.1 isSubContext 44
4.3.2 difference 45
4.3.3 intersection 46
4.3.4 projection 46
4.3.5 hiding 47
4.3.6 override 48
4.3.7 union 48
4.4 Design and Implementation of Tag Set Classes 49
4.4.1 Design of Tag Set Classes 49
4.4.2 Implementation of Operators on Tag Sets 50
4.4.2.1 Ordered Finite Tag Set 50
4.4.2.2 Ordered Infinite Tag Set 54
4.4.2.3 Unordered Finite Tag Set 57
4.4.2.4 Unordered Infinite Tag Set 57
4.5 Embedding Context and Tag Set Classes into the GIPSY 58
4.6 Summary 59

5 Testing **61**
5.1 Testing Infrastructure for Lucx 61
5.2 Testing for Lucx Parser 61

5.3 Testing for Semantic Analyzer 65
5.4 Unit Testing for Context Class and Tag Set Classes 68
5.4.1 Test Over GIPSYContext Class 68
5.4.2 Test Over Tag Set Classes 70
5.4.2.1 Test Over *isInTagSet* 71
5.4.2.2 Test Over *getPrevious()* and *getNext()* 71
5.5 Summary 71

6 Conclusion **72**
6.1 Extending the GIPC 72
6.2 GIPSY Type System 73
6.3 Context Calculus 73
6.4 Discussions and Limitations 74
6.4.1 Context Set: Motivation and Limitations 74
6.4.2 The Concept of Box 75
6.4.3 The Unordered Infinite Tag Set 75
6.4.4 Test On Context Calculus 76

7 Future Work **77**
7.1 The New Generation Of Execution Engine 77
7.2 Formal Verification of Context Set and Box Theory 77
7.3 More Possibilities Of Tag Set 77

Appendix **83**

A Source Listing For Context Calculus Operators **83**

B Source Listing For Tag Set Types **93**

List of Figures

1	GIPL Syntax	9
2	GIPL Semantics	10
3	New syntactic rules introduced by Lucx SIPL	11
4	New semantic rules introduced by Lucx SIPL	12
5	Architectural overview of the GIPSY	13
6	Compilation/execution of GIPSY programs	14
7	GIPC Structure	36
8	Concrete Lucx Syntax	38
9	GIPSY Type System.	42
10	Context Classes	44
11	Tag Set Classes	50
12	Embed context class into GIPSY	59
13	Testing Infrastructure for Lucx	62
14	Unit Testing for Tag Set and GIPSYContext	69

Chapter 1

Introduction

1.1 Problem Statement

Lucid [43, 12, 4, 2, 3] represents a family of intensional programming languages that has several dialects all sharing a generic counterpart, which we call the Generic Intensional Programming Language (GIPL) [33, 48, 35, 31]. Other Lucid variants are referred to as Specific Intensional Programming Languages (SIPL) [33, 48, 35, 31], and can be translated into the GIPL. The GIPL is an intensional programming language whose semantics was defined according to Kripke's possible worlds semantics [25, 26]. Following this semantics, the notion of context is a core concept, as the evaluation of expressions in intensional programming languages relies on the implicit context of utterance [33, 44]. In earlier versions of Lucid, although the notion of context is pervasive, it could not be manipulated directly as first-class value, meaning that context cannot be assigned to a variable, passed as a parameter, returned as the result of a function, or included in a larger composite value.

To resolve this issue, a new dialect of Lucid, Lucx (Lucid Enriched With Context), was introduced by Wan [45, 44]. Lucx embraced the idea of context as first-class value. It also had a collection of operators defined, coalesced into a well-defined *context calculus*. However, the operational details of integrating Lucx into the GIPSY have not yet been defined, so these latest very important results have not been integrated in our operational system. The main objective of this thesis is the integration of the *context calculus* theory, developed by Wan, into GIPSY.

1.2 Proposed Solution

The classic way of integrating a new Lucid variant into the GIPSY is to provide its parser and derive the translation rules [48] from SIPL to GIPL. The General Eduction Engine (GEE) [33, 31], will then execute the translated source program. This compilation process is described in more details in Chapter 2.

However, there are two main reasons that Lucx cannot follow this route: First, what the translation does is to substitute Lucx context calculus operators with GIPL basic operators. The Lucx-to-GIPL translation rules were given by Wan in her PhD thesis. These translation rules are in fact *simulating* the notion of context using the existing set of operators. Even though these translation rules are correct, such substitutions will to a large degree increase the size of a program, making simple context operators require very elaborated computing to simulate contexts as first class values, thus the evaluation efficiency would be reduced significantly. The second reason is that context is a very important concept in Lucid. The notion of context as first-class value will eventually benefit the evolution of Lucid towards a more complete and general intensional programming language. Thus, it is more reasonable to make the concept of context explicit in GIPSY, as opposed to simulate it through translation.

Following the above reasons, we propose the solution that after constructing Lucx parser, instead of translating it into GIPL, we create a separate component which contains the actual implementation of Lucx's context type and context calculus operators. Then the execution of Lucx programs can be achieved by grafting the resulting context calculus plugin into the GEE [33, 41, 40]. By this means, the efficiency of execution is much higher than the classic translation process and the structure of the engine retains its general properties to the greatest extent.

1.3 Contributions

The major contributions of this thesis are:

- Integration of Lucx's compiler into the GIPSY framework
  - Design of the concrete syntactic rules of Lucx from Wan's thesis

  - Construction of a top-down parser for Lucx and its integration into the GIPSY
  - Extension of the existing Semantic Analyzer by adding type checking for *context* values

- Design of a *context type* and its integration into the GIPSY Type System:
  - Abstraction of a *context* entity into a type shared by all the intensional programming languages supported by the GIPSY
  - Integration of the *context type* into the compile-time and run-time type system
  - Implementation of *context calculus* operators within the context type

- Design of *tag set types* from the definition of context and their integration into the GIPSY Type System:
  - Design of a cohesive group of *tag set types* that enables more flexibility in the definition of Lucid dimensions
  - Implementation of the tag set types and their integration into the GIPSY Type System

1.4 Structure of the Thesis

Chapter 2 presents the background upon which Lucx was developed and implemented. Chapter 3 provides the theoretical basis for the following implementation work, including the definition of context, context calculus and tag set types. Chapter 4 specifies the implementation details of how the Lucx compiler was constructed, how the context type, tag set types and more specifically, context calculus operators were implemented, and how these entities were embedded into the GIPSY Type System. Chapter 5 demonstrates the testing strategies and results of the corresponding modules in Chapter 4. Finally Chapter 6 draws a general conclusion of this thesis and Chapter 7 gives an overview of the future work.

Chapter 2

Background

In this chapter, we present the information of Lucid programming languages and GIPSY framework to provide an overview of the background where the context calculus of Lucx was introduced of and is to be integrated.

2.1 Intensional Logic and Lucid

Intensional logic comes from research in natural language understanding [24, 11, 16]. According to Carnap [6], 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 intensional expression is the set of all its actual truth-values in different possible contexts of utterance, where this expression can be evaluated. 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 have an expression :

E: the average temperature in Montreal in January is greater than $0°C$.

The explicit *context* of this expression is [`place:Montreal`, `month:January`], it adds dimension `place` and `month` to the expression and `Montreal` and `January` are

the place holder tags along those dimensions. Dimension names together with tag values form the context in that expression.

We can extend that intension in both the `place` and `month` dimensions. And the evaluation result of that expression will vary accordingly, as shown in the following: Given the `place` dimension tags $\{Montreal, Ottawa, Quebec\}$, we have a valuation for E:

$E =$

	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	F	F	T	T	T	F	F	F	F
Quebec	F	F	F	F	F	F	T	F	F	F	F	F

Intensional logic and its core concept of *context* is the foundamental basis of intensional programming languages.

2.2 Lucid Programming Languages

Lucid is a multidimensional intensional programming language, whose semantics is based on the possible world semantics of intensional logic. It is a functional language in which expressions and their valuations are allowed to vary in an arbitrary number of dimensions [33]. In its initial form, Lucid was designed to enable the declarative expression of the sequence of values that a procedural program's variables are taking as the program is being executed. In this view, Lucid relied on a unique *time* dimension. Later on, it was realized that the generalization to multidimensionality added much more expressive power to the language. The initial set of basic Lucid operators, *first, next, fby, wvr, asa* and *upon* [33] enabled the sequential/recursive definition of sequences. When Indexical Lucid [13] came into existence, it allowed accessing context properties in multiple dimensions, as well as two basic intensional operators added to allow randomly access data streams. One is intensional navigation (`@.d`), which allows the values of a stream to vary along the dimension `d`. Another is intensional query (`#.d`), which refers to the current position (i.e.tag value) along the dimension `d`. After that, a group of Lucid variants has been derived according to the fundamental intensional logic. Following is a brief list of Lucid dialects:

1. Granular Lucid (GLU), 1996 [21, 22]: First working hybrid intensional-imperative paradigm (C/Fortran and Indexical Lucid), where a GLU program is defined

in two parts: an Indexical Lucid part acting as a *skeleton language*, into which user-defined functions written in the second part (written in C or Fortran) are called. This model was invented as a proof-of-concept to demonstrate the dataflow model of computation attributed to Lucid at the time. Allowing Lucid to call procedures written in a standard procedural language allowed for increased granularity of computation, identified as a flaw when executing fine-grained dataflow programs defined by standard "pure Lucid" programs. In the cases where the nature of the procedures allowed them to be executed in parallel, this model also allowed GLU to be used as a "program parallelization" platform. The tagged-token demand-driven dataflow model (called *eduction*) based on intensional logics used in the evaluation of Lucid programs did also provide a failure-resistant model of distributed/parallel computation. The GLU model is the basis of the design of the GIPSY.

2. Tensor Lucid, 1999 [33]: Tensor Lucid is a dialect developed by Joey Paquet for plasma physics computations to illustrate advantages and expressiveness of Lucid over an equivalent solution written in Fortran. The objects manipulated in Lucid are scalar fields. More general fields, such as vector or tensor fields, cannot be expressed and manipulated in a natural manner. Tensor Lucid is a generalization of Lucid that enables the expression and manipulation of vector and tensor fields, which are in fact a generalization of scalar fields.

3. GIPL, 1999 [33]: GIPL is a generic form of Lucid, i.e. all Lucid dialects can be translated into GIPL through a set of translation rules. GIPL is in the foundation of the execution semantics of GIPSY because its Abstract Syntax Tree is the only type of Abstract Syntax Tree GEE understands when executing a GIPSY program.(Note that the implementation of Lucx is different than what stated here, we will discuss more about this later in this thesis.)

4. RLucid, 1999 [15]: RLucid programming language is Lucid with a new operator, called *before*. With *before*, merging two streams in a deterministic manner, according to the time of arrival, becomes a possibility. It is a superset of Lustre, which was designed for the programming of real-time kernels. Real-time interfaces can be written in RLucid, thereby making RLucid, in some sense, a general real-time language.

5. JLucid, Objective Lucid, 2003 - 2005 [31]: JLucid brings embedded Java and most of its powers into Indexical Lucid in the GIPSY by allowing intensional languages to manipulate Java methods as first class values. However, it is very natural to have objects with Java and manipulate their members in scientific intensional computation, yet JLucid fails to support Java's capability. Hence, Objective Lucid was designed to address this deficiency.

6. Lucx, 2003 - 2005 [45, 44]: Kaiyu Wan introduced the notion of context as first-class value in Lucid, thus context can be declared and manipulated directly in Lucid programming languages. She also provides a set of well-defined context calculus operators performed on context values to yield new contexts for different applications. Before Lucx, context is always implicit in Lucid. Now Lucx has brought important new feature to Lucid in the sense of *true intension.*

2.3 GIPL, and Lucx as a SIPL

As briefly mentioned in the previous section, GIPL is the generic counterpart of all the Lucid programming languages. Like Indexical Lucid, which it is derived from, it has only two basic intensional operators: `#` and `@` [33]. SIPLs are Lucid dialects with their own attributes and objectives. Theoretically, all SIPLs can be translated into the GIPL. All the SIPL conservatively extend the GIPL syntactically and semantically. The following sections discuss about such extensions of Lucx to GIPL. The syntax and semantics of GIPL are listed in Figure 1 and Figure 2 respectively. We then present the extended syntactic and semantic rules of Lucx to GIPL in Figure 3 and Figure 4.

Following is the description of GIPL semantic rules as presented in [33]:

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

means that under the definition environment $\mathcal{D}$, expression E would evaluate to value v.

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

means that in the definition environment $\mathcal{D}$, and in the evaluation context $\mathcal{P}$ (sometimes also referred to as *point*) , 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 **Id** is the set of all possible identifiers and **IdEntry**, summarized in Table 1, has five possible kinds of value, one for each of the kinds of identifier:

- *Dimensions* define the coordinates in which one can navigate with the `#` and `@` operators. The **IdEntry** is simply (`dim`).

- *Constants* are external entities that provide a single value, whatever the context. Examples are integers and Boolean values. The **IdEntry** is $(\texttt{const}, c)$, where c is the value of the constant.

- *Data operators* are external entities that provide memoryless functions. Examples are the arithmetic and Boolean functions. The constants and data operators are said to define the *basic algebra* of the language. The **IdEntry** is $(\texttt{op}, f)$, where f is the function itself.

- *Variables* carry the multidimensional streams. The **IdEntry** is $(\texttt{var}, E)$, where E is the expression defining the variable. It should be mentioned that this semantics makes the assumption that all variable names are unique. This constraint should be easy to overcome by performing compile-time renaming or using a nesting level environment.

- *Functions* are non-recursive user-defined functions. The **IdEntry** is $(\texttt{func}, id_i, E)$, where the id_i are the formal parameters to the function and E is the body of the function. The semantics for recursive functions could be easily added, but we would have to store the environment, and it would add to the understanding of the situation, especially given that Lucid encourages the use of iteration rather than recursion.

The evaluation context $\mathcal{P}$, which is changed when the `@` operator or a `where` clause is encountered, associates a tag to each relevant dimension. It is therefore a partial function

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

Table 1: Possible identifiers in the definition environment

type	form
dimension	$(\texttt{dim})$
constant	$(\texttt{const}, c)$
operator	$(\texttt{op}, f)$
variable	$(\texttt{var}, E)$
function	$(\texttt{func}, id_i, E)$

$$
\begin{array}{rcl}
E & ::= & id \\
 & | & E(E_1, ..., E_n) \\
 & | & \textbf{if } E \textbf{ then } E' \textbf{ else } E'' \\
 & | & \#E \\
 & | & E@E'E'' \\
 & | & E \texttt{ where } Q \\
Q & ::= & \texttt{dimension } id \\
 & | & id = E \\
 & | & id(id_1, id_2, ..., id_n) = E \\
 & | & QQ
\end{array}
$$

Figure 1: GIPL Syntax

Each type of identifier can only be used in the appropriate situations. Identifiers of type, `op`, `func` and `dim` evaluate to themselves. Constant identifiers (`const`) evaluate to the corresponding constant. Function calls, resolved by the $\mathbf{E_{fct}}$ rule, require the renaming of the formal parameters into the actual parameters (as represented by $E'[id_i \leftarrow E_i]$).

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. The rule for the `where` clause, $\mathbf{E_w}$, which corresponds to the syntactic expression E `where` Q, evaluates E using the definitions (Q) therein.

The additions to the definition environment and context of evaluation made by the $\mathbf{Q}$ rules are local to the current `where` clause. This is represented by the fact that the $\mathbf{E_w}$ rule returns neither $\mathcal{D}$ nor $\mathcal{P}$. The $\mathbf{Q_{dim}}$ rule adds a dimension to the definition environment and, as a convention, adds this dimension to the context of evaluation with tag 0. The $\mathbf{Q_{id}}$ and $\mathbf{Q_{fid}}$ simply add variable and function identifiers along with their definition to the definition environment.

$$\mathbf{E_{cid}} : \frac{\mathcal{D}(id) = (\mathtt{const}, c)}{\mathcal{D}, \mathcal{P} \vdash id : c} \tag{1}$$

$$\mathbf{E_{opid}} : \frac{\mathcal{D}(id) = (\mathtt{op}, f)}{\mathcal{D}, \mathcal{P} \vdash id : id} \tag{2}$$

$$\mathbf{E_{did}} : \frac{\mathcal{D}(id) = (\mathtt{dim})}{\mathcal{D}, \mathcal{P} \vdash id : id} \tag{3}$$

$$\mathbf{E_{fid}} : \frac{\mathcal{D}(id) = (\mathtt{func}, id_i, E)}{\mathcal{D}, \mathcal{P} \vdash id : id} \tag{4}$$

$$\mathbf{E_{vid}} : \frac{\mathcal{D}(id) = (\mathtt{var}, E) \qquad \mathcal{D}, \mathcal{P} \vdash E : v}{\mathcal{D}, \mathcal{P} \vdash id : v} \tag{5}$$

$$\mathbf{E_{op}} : \frac{\mathcal{D}, \mathcal{P} \vdash E : id \qquad \mathcal{D}(id) = (\mathtt{op}, f) \qquad \mathcal{D}, \mathcal{P} \vdash E_i : v_i}{\mathcal{D}, \mathcal{P} \vdash E(E_1, \ldots, E_n) : f(v_1, \ldots, v_n)} \tag{6}$$

$$\mathbf{E_{fct}} : \frac{\mathcal{D}, \mathcal{P} \vdash E : id \qquad \mathcal{D}(id) = (\mathtt{func}, id_i, E') \qquad \mathcal{D}, \mathcal{P} \vdash E'[id_i \leftarrow E_i] : v}{\mathcal{D}, \mathcal{P} \vdash E(E_1, \ldots, E_n) : v} \tag{7}$$

$$\mathbf{E_{c_T}} : \frac{\mathcal{D}, \mathcal{P} \vdash E : true \qquad \mathcal{D}, \mathcal{P} \vdash E' : v'}{\mathcal{D}, \mathcal{P} \vdash \mathtt{if}\ E\ \mathtt{then}\ E'\ \mathtt{else}\ E'' : v'} \tag{8}$$

$$\mathbf{E_{c_F}} : \frac{\mathcal{D}, \mathcal{P} \vdash E : false \qquad \mathcal{D}, \mathcal{P} \vdash E'' : v''}{\mathcal{D}, \mathcal{P} \vdash \mathtt{if}\ E\ \mathtt{then}\ E'\ \mathtt{else}\ E'' : v''} \tag{9}$$

$$\mathbf{E_{tag}} : \frac{\mathcal{D}, \mathcal{P} \vdash E : id \qquad \mathcal{D}(id) = (\mathtt{dim})}{\mathcal{D}, \mathcal{P} \vdash \#E : \mathcal{P}(id)} \tag{10}$$

$$\mathbf{E_{at}} : \frac{\mathcal{D}, \mathcal{P} \vdash E' : id \qquad \mathcal{D}(id) = (\mathtt{dim}) \qquad \mathcal{D}, \mathcal{P} \vdash E'' : v'' \qquad \mathcal{D}, \mathcal{P} \dagger [id \mapsto v''] \vdash E : v}{\mathcal{D}, \mathcal{P} \vdash E\ @E'\ E'' : v} \tag{11}$$

$$\mathbf{E_{w}} : \frac{\mathcal{D}, \mathcal{P} \vdash Q\ :\ \mathcal{D}', \mathcal{P}' \qquad \mathcal{D}', \mathcal{P}' \vdash E : v}{\mathcal{D}, \mathcal{P} \vdash E\ \mathtt{where}\ Q : v} \tag{12}$$

$$\mathbf{Q_{dim}} : \frac{}{\mathcal{D}, \mathcal{P} \vdash \mathtt{dimension}\ id\ :\ \mathcal{D} \dagger [id \mapsto (\mathtt{dim})], \mathcal{P} \dagger [id \mapsto 0]} \tag{13}$$

$$\mathbf{Q_{id}} : \frac{}{\mathcal{D}, \mathcal{P} \vdash id = E\ :\ \mathcal{D} \dagger [id \mapsto (\mathtt{var}, E)], \mathcal{P}} \tag{14}$$

$$\mathbf{QQ} : \frac{\mathcal{D}, \mathcal{P} \vdash Q\ :\ \mathcal{D}', \mathcal{P}' \qquad \mathcal{D}', \mathcal{P}' \vdash Q' : \mathcal{D}'', \mathcal{P}''}{\mathcal{D}, \mathcal{P} \vdash Q\ Q'\ :\ \mathcal{D}'', \mathcal{P}''} \tag{15}$$

$$\tag{16}$$

Figure 2: GIPL Semantics

$$
\begin{array}{rcl}
E & ::= & [E : E, ..., E : E] \\
 & | & \{E, ..., E\} \\
 & | & E@E' \\
 & | & E \; cxtop \; E \\
cxtop & ::= & \textbf{isSubContext} \\
 & | & \textbf{difference} \\
 & | & \textbf{intersection} \\
 & | & \textbf{projection} \\
 & | & \textbf{hiding} \\
 & | & \textbf{override} \\
 & | & \textbf{union}
\end{array}
$$

Figure 3: New syntactic rules introduced by Lucx SIPL

2.3.1 Syntactical Extension

Being a SIPL, Lucx inherits all the generic syntax and semantics of the GIPL. It also extends the GIPL by introducing context as first class value and context calculus operators, as defined in [44]. Figure 3 shows the elements of Lucx syntax to be added to that of the GIPL. In Lucx syntax, the notation $[E : E, ..., E : E]$ is introduced to represent a *context*; the notation $\{E, \ldots, E\}$ is representing a context set; finally, a set of context calculus operators is also presented as the complement of GIPL operators. To explain the notion of context as first class value, let's focus on the `@` expression: In GIPL, it is $E \ @ \ E'E''$, i.e. the notion of context is represented by $E'E''$; while its counterpart in Lucx is $E \ @ \ E'$, because we introduce the $[E : E, ..., E : E]$ notation into Lucx expressions. Thus, from the syntactic perspective of the `@` operator, Lucx' $[E : E, ..., E : E]$ is adding *syntactic sugar* to GIPL by syntactically encapsulating context into square brackets. As explained below, Lucx in fact provides a semantics for such encapsulation by creating a *context* semantic building block.

2.3.2 Semantic Extension

As a conservative extension to GIPL, Lucx's semantics extends GIPL by introducing the notion of *context* as a building block into the semantic rules, i.e. *context as first class value*. Still take the `@` expression for example. In GIPL semantic rule (11), we can see that E' and E'' evaluate to two separate semantic entities, which are *dimension* and *tag value*, the combination of these two elements is actually what we refer to as a *context*, i.e. a $< dimension : tag >$ mapping. However, in GIPL, there's

$$\mathbf{E}_{\mathbf{at(cxt)}} \quad : \quad \frac{\mathcal{D},\mathcal{P} \vdash E' : \mathcal{P}' \qquad \mathcal{D},\mathcal{P}\dagger\mathcal{P}' \vdash E : v}{\mathcal{D},\mathcal{P} \vdash E \mathbin{@} E' : v} \tag{17}$$

$$\mathbf{E}_{\mathbf{construction(cxt)}} \quad : \quad \frac{\begin{array}{ll} \mathcal{D},\mathcal{P} \vdash E_{d_j} : id_j & \mathcal{D}(id_j) = (\mathtt{dim}) \\ \mathcal{D},\mathcal{P} \vdash E_{i_j} : v_j & \mathcal{P}' = \mathcal{P}_0\dagger[id_1 \mapsto v_1]\dagger\ldots\dagger[id_n \mapsto v_n] \end{array}}{\mathcal{D},\mathcal{P} \vdash [E_{d_1} : E_{i_1}, E_{d_2} : E_{i_2}, \ldots, E_{d_n} : E_{i_n}] : \mathcal{P}'} \tag{18}$$

$$\mathbf{E}_{\mathbf{construction(cxtset)}} \quad : \quad \frac{\mathcal{D},\mathcal{P} \vdash E_{w:1\ldots m} : \mathcal{P}_m}{\mathcal{D},\mathcal{P} \vdash \{E_1, \ldots, E_m\} : \{\mathcal{P}_1, \ldots, \mathcal{P}_w\}} \tag{19}$$

$$\mathbf{E}_{\mathbf{op(cxt)}} \quad : \quad \frac{\mathcal{D},\mathcal{P} \vdash E : id \qquad \mathcal{D}(id) = (\mathsf{cop}, f) \qquad \mathcal{D},\mathcal{P} \vdash E_i : \mathcal{P}_i}{\mathcal{D},\mathcal{P} \vdash E(E_1, \ldots, E_n) : f(\mathcal{P}_1, \ldots, \mathcal{P}_n)} \tag{20}$$

$$\mathbf{E}_{\mathbf{op(cxtset)}} \quad : \quad \frac{\mathcal{D},\mathcal{P} \vdash E : id \qquad \mathcal{D}(id) = (\mathsf{sop}, f) \qquad \mathcal{D},\mathcal{P} \vdash E_i : \{\mathcal{P}_{i_1}, \ldots, \mathcal{P}_{i_k}\}}{\mathcal{D},\mathcal{P} \vdash E(E_1, \ldots, E_n) : f(\{\mathcal{P}_{1_1}, \ldots, \mathcal{P}_{1_s}\}, \ldots, \{\mathcal{P}_{n_1}, \ldots, \mathcal{P}_{n_m}\})} \tag{21}$$

Figure 4: New semantic rules introduced by Lucx SIPL

no semantic representation of the context *as an entity* that coalesces *dimension* and *tag* together, in other words, context is not first class value in GIPL semantics. On the contrary, in Lucx semantic rule (19), E' evaluates to one encapsulated *context*.

Figure 4 also shows the construction of *simple context* out of the expression $[E_{d_j} : E_{i_j}]$ pairs in rule (19). Rule (20) shows construction of *context set*, which evaluates $\{E_1, \ldots, E_m\}$ as a set of contexts, each element of the set being the result of an evaluation using rule (19). Rule (21) and rule (22) show semantics of context calculus operators on *simple context* and *context set* respectively. These rules were differentiated from the standard GIPL rule $\mathbf{E}_{\mathbf{op}}$ (6) due to the fact that these operators on contexts are meaningless outside of Lucx.

2.4 Overview of GIPSY Framework Architecture

The GIPSY framework consists of three modular sub-systems: The General Intensional Programming Compiler (GIPC) [34, 49, 35]; the General Eduction Engine (GEE) [33, 31], and the Run-time Interactive Programming Environment (RIPE) [9, 39], as shown in Figure 5. GIPC is the component on which all the compilers of the programming languages supported by GIPSY reside. GEE is where a Lucid program is executed using the *eductive* model of computation. And RIPE is a programming environment (i.e. and Integrated Development Environment or IDE) enabling the edition, compilation and execution of GIPSY programs using the GIPC and GEE.

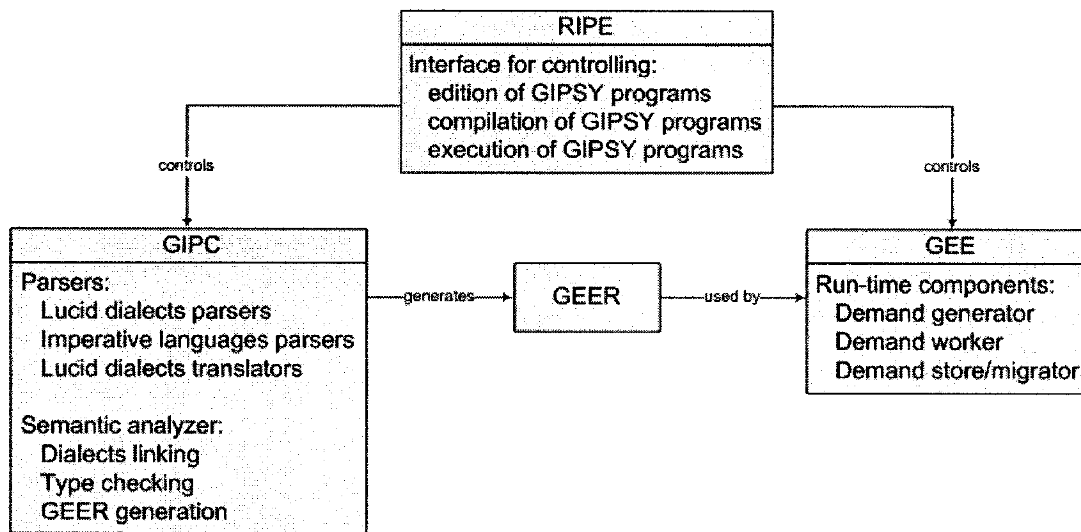

Figure 5: Architectural overview of the GIPSY

In the GIPSY, a *GIPSY Program*, may consist of two parts: the *Lucid part* that defines the intensional data dependencies between variables and, optionally, the *sequential part* that defines the granular sequential computation units (written in a procedural language). The GIPSY program is compiled in a two-stage process, as depicted in Figure 6. First, the intensional (GIPL) part of the GIPSY program is parsed, and then translated in Java data structures, then the resulting Java program is compiled in the standard way, resulting in run-time system resources that we call a GEER (General Eduction Engine Resources). The GIPSY run-time system, the GEE (General Eduction Engine) is an interpreter written in Java that uses the GEER to execute the program using the *eductive* model of computation that can be described as "tagged-token demand-driven dataflow" computing.

2.4.1 Compilation: GIPC

The main component of the GIPC is the GICF (General Intensional Compiler Framework), a *compiler framework* providing a generic infrastructure that enables the compilation of hybrid programs written using various dialects of Lucid, as well as various procedural programming languages, in which the Lucid (intensional) parts are calling procedures written in these various procedural languages. One of the main precepts of this framework design is that of the GIPL being a generic language into which

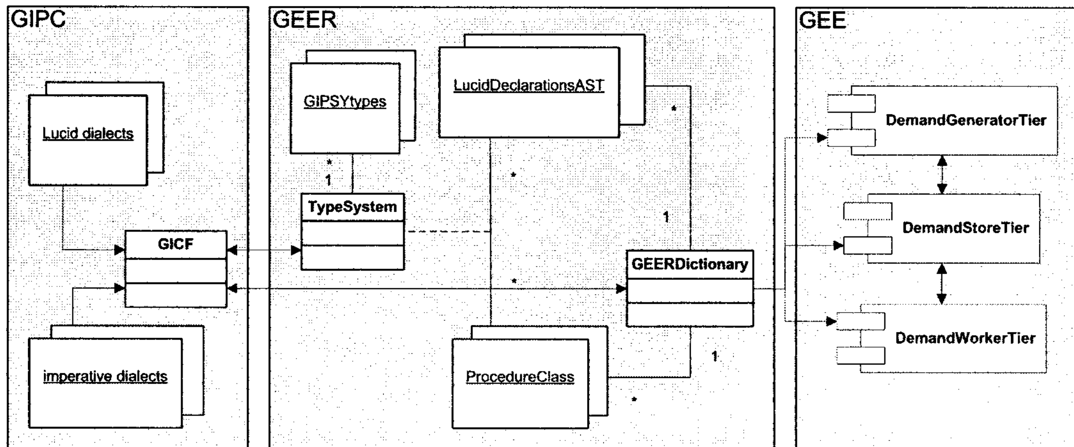

Figure 6: Compilation/execution of GIPSY programs

other Lucid dialects can be translated. This precept has two main goals: (1) to make the intensional part of the compiler framework easy to extend vs Lucid dialects and (2) have a fixed run-time system that will not have to be adapted each time a new Lucid dialect is added.

The compilation process starts with the action of the `Preprocessor` [31, 30] on the incoming GIPSY program's source code stream. The `Preprocessor`'s role is to do preliminary program analysis, processing, and splitting the source GIPSY program into "chunks", each written in a different language and identified by a *language tag*. In a very general view, a GIPSY program is a hybrid program consisting of different languages in one or more source files; then, there has to be an interface between all these code segments. Thus, the `Preprocessor` after some initial parsing (using its own preprocessor syntax) and producing the initial parse tree, constructs a preliminary dictionary of symbols used throughout the program. This is the basis for type matching and semantic analysis applied later on. This is also where the first step of type assignment occurs (for more details, see the description of the type system below), especially on the boundary between typeful and typeless parts of the program, e.g. Java and a specific Lucid dialect. The `Preprocessor` then splits the code segments of the GIPSY program into chunks preparing them to be fed to the respective concrete compilers for those chunks. The chunks are represented through the `CodeSegment` class that the `GIPC` collects. There are four baseline types of segments defined to be used in a GIPSY program. They are:

Table 2: Matching data types between Lucid and Java.

Return Types of Java Methods	Types of Lucid Expressions
`int, byte, long`	`int`
`float`	`float`
`double`	`double`
`boolean`	`bool`
`char, String`	`string`
`void`	`bool::true`
Parameter Types Used in Lucid	**Corresponding Java Types**
`string`	`String`
`float`	`float`
`double`	`double`
`int, dimension`	`int`
`bool`	`boolean`

- `#funcdecl` program segment declares function prototypes written as imperative language functions defined later or externally from this program to be used by the intensional language part. The syntactical form of these prototypes is particular to GIPSY programs and need not resemble the actual function prototype declaration they describe in their particular programming language. They serve as a basis for static and dynamic type assignment and checking within the GIPSY type system with regards to procedural functions called by other parts of the GIPSY program, e.g. the Lucid code segments.

- `#typedecl` segment lists all user-defined data types that can potentially be used by the intensional part; usually objects. These are the types that do not explicitly appear in the matching table in Table 2 describing the basic data types allowed in GIPSY programs.

- `#<IMPERATIVELANG>` segment declares that this is a code segment written in whatever imperative language may be, for example `#JAVA` for Java, `#CPP` for C++, `#FORTRAN` for Fortran, `#PERL` for Perl, `#PYTHON` for Python, etc.

- `#<INTENSIONALLANG>` segment declares that this is a code segment written in whatever intensional language may be, for example `#GIPL`, `#INDEXICALLUCID`, `#LUCX`, `#JLUCID`, `#OBJECTIVELUCID`, `#TENSORLUCID`, etc. as implemented by the GIPSY. An example of a hybrid program is presented in Listing 2.1. The

preamble of the program with the type and function declaration segments are the main source of type information that is used at compile time to annotate the nodes in the tree to help both static and semantic analysis.

Of particular importance in the compiler framework design is the latter code segments (i.e. `#<INTENSIONALLANG>`). Those written in GIPL are parsed and directly compiled into the GEER. For other Lucid dialects, the compiler uses a parser for this specific language, and then relies on translation rules to translate, at the syntax tree level, this specific language's construct into GIPL constructs, then the translated syntax tree is compiled into a GEER. This part of the compiler framework has been designed and implement by Serguei Mokhov in his Master's thesis. In the final compilation phase, the semantic analyzer (including the type checker) makes appropriate semantic checking and final generation of the GEER, which can then be used by the GEE to execute this compiled GIPSY program.

Another important issue that has to be mentioned here is the GIPSY Type System [31]. JLucid [31] enables a GIPSY program to be hybrid, meaning that it can be composed of an intensional part and a imperative part. In pure Lucid, type is implicit and type declarations never appear at the syntactic level; however, in many imperative programming languages, such as Java and C++, typing is explicit. One of the goals of our framework is to support a common run-time environment and co-existence of both the intensional and imperative languages. Thus we have implemented a general type system to keep all the types shared by the programming languages supported by GIPSY.

Although type is implicit at the syntactical level in Lucid, at the semantical level, necessary type checking needs to refer to types. Also at runtime, the evaluation of expressions needs to resort to types. The uniqueness of our type system is that it not only supports the conventional types such as `integer` and `string`, but also types related to intensionality, such as `context`, which will be discussed more in detail later in this thesis.

2.4.2 Execution: GEE

The GIPSY recently adopted a multi-tier architecture, in which the system is executed using tiers that have various roles in the execution of GIPSY programs. Tiers can be residing on various GIPSY computation nodes (i.e. computers hosting a number of

```
/**
 * Language-mix GIPSY program.
 * @author Serguei Mokhov
 */
#typedecl

myclass;

#funcdecl

myclass foo(int,double);
float bar(int,int):"ftp://newton.cs.concordia.ca/cool.class":baz;
int f1();

#JAVA
myclass foo(int a, double b)
{
   return new myclass(new Integer((int)(b + a)));
}

class myclass
{
   public myclass(Integer a)
   {
      System.out.println(a);
   }
}

#CPP
#include <iostream>

int f1(void)
{
   cout << "hello";
   return 0;
}

#OBJECTIVELUCID

A + bar(B, C)
where
   A = foo(B, C).intValue();
   B = f1();
   C = 2.0;
end;

/*
 * in theory we could write more than one intensional chunk,
 * then those chunks would evaluate as separate possibly
 * totally independent expressions in parallel that happened
 * to use the same set of imperative functions.
 */

// EOF
```

Listing 2.1: Example of a hybrid GIPSY program.

GIPSY tiers), thus achieving distributed computing through the use of the Demand Migration Framework for communication between tiers [41, 40, 36].

Demand Generator Tier Using the *eductive* mode of computation, the Demand Generator Tier (DGT) generates demands according to the program declarations and definitions stored in one of the instances of GEER that it hosts. The demands generated by the Demand Generator Tier instance can be further processed by other Demand Generator Tiers instances or Demand Worker Tier instances, the demands being migrated across tier instances through a Demand Store Tier instance. Each DGT instance hosts a set of GEER instances that corresponds to the Lucid programs it can process demands for. A demand-driven mechanism allows the Demand Generator Tier to issue system demands requesting for additional GEER instances to be added to its GEER Pool, thus enabling DST instances to process demands for additional programs as they are executed on the GIPSY instances they belong to. The three different execution tiers are describe briefly below:

Demand Store Tier The Demand Store Tier (DST) acts as a middleware between tiers in order to migrate demands between them. In addition to the migration of the demands and values across different tiers, the Demand Store Tier provides persistent storage of demands and their resulting values, thus achieving better processing performances by not having to re-compute the value of every demand every time it is re-generated after having been processed. From this latter perspective, it is equivalent to the historical notion of *warehouse* in the eductive model of computation. A centralized communication point or warehouse is likely to become an execution bottleneck. In order to avoid that, the Demand Store Tier uses a peer-to-peer architecture and mechanism to connect all Demand Store Tier instances in a given GIPSY instance. This allows any demand or its resulting value to be stored on any DST instance, but yet allows abstract querying for a specific demand value on any of the DST instances. If the demanded value is not found on the DST instance receiving the demand, it will contact its DST instance peers using a peer-to-peer mechanism. This mechanism allows to see the Demand Store abstractly as a single store that is, behind the scenes, a distributed one.

Demand Worker Tier The Demand Worker Tier (DWT) processes *procedural demands* i.e. demands for the execution of functions defined in a procedural language, which are only present in the case where hybrid intensional programs are being executed. The DGT and DWT duo is an evolution of the generator-worker architecture adopted in GLU [21, 22]. It is through the operation of the DWT that increased granularity of computation is achieved. Similarly to the DGT, each DWT instance hosts a set of compiled procedures (Procedure Classes) that corresponds to the procedural demands it can process. A demand-driven mechanism allows the Demand Worker Tier to issue system demands requesting for additional Procedure Classes to be added to its Procedure Class Pool, thus achieving increasing capacities over time, on demand.

2.5 Related Work

Context-awareness According to [8], a context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the users and applications themselves. And context-awareness is defined as: A system is context-aware if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user's task. Context-awareness now has a wide application domains such as: peer-to-peer mobile networks [17]; body sensor networks [10] and so on. In their research and implementation, the notion of context is more related to the applications that can sense their physical environment. More specifically, three concepts of context are: (1) where you are; (2) who you are with; and (3) what resources are nearby [46]. Our definition of context is at a more abstract and formal level, and our applications would cover scientific programming [33] as well.

Context-oriented Programming Context-dependent behavior is becoming increasingly important for a wide range of application domains, from pervasive computing [1] to common business applications. Context-oriented Programming (COP) is a new method of programming which aims to alleviate pervasive problems by incorporating context as a first-class construct of a programming language, much in the same way that variables, classes, and functions form the first-class constructs of many modern languages [23].

In [23], they proposed that a context-oriented program is one with many 'gaps' or 'open-terms', and the context-filling operation completes such a program by dynamically selecting portions of code from a repository of candidates to fill these gaps. This selection process is based on the execution context of the program and a description of the open-term's requisites. In practical terms COP allows the authoring of program skeletons, which are pieces of software that contain a number of open terms. Open terms are simply gaps in a program, which specify a goal and context. Actual code to replace them is selected dynamically from a number of stubs in a context-dependent fashion. This idea is referred to as context-filling. Context-filling is the distinguishing feature of COP, the result being a separation of code into a context-free skeleton and context-dependent stubs. Their basic idea is first, leave gaps in a program where context-dependent statements are needed; then at run time, fill those gaps with proper context in the database repository thus the execution of the program would respond to the context and yield proper result.

In [19], they present their idea towards COP as a new programming technique to enable context-dependent computation. They claim that Context-oriented Programming brings a similar degree of dynamicity to the notion of behavioral variations that object-oriented programming brought to ad-hoc polymorphism. In support of this claim, they carry out their first language extension called ContextL [7] to argue that the dynamic representation of layers and their scoped activation and deactivation in arbitrary places of the code are the essential ingredients for COP. In their implementation, context is represented by "layer". They define the context-dependent attributes or behaviors into different layers and by attaching or detaching those layered definitions, the original plain object could have polymorphic behaviors. For example, assume that there's a `Person` class, and an object of `Person` could be an *employee* or *student* and so on. Their approach is to define the employee attributes in `Employee` layered class and if the end user wants to display the employee's attributes, he can just activate the employee layer and attach it to the original `Person` class.

Although their work and ours share the idea of "Context as first class value", there is a major difference: They treat context as an individual concept in programming, however, it is never represented at an abstract level, in other words, context has never been abstracted into a "type". For example, in [23], they use a notion of context-dependent "stub" to represent contexts in different application domains. They are

stored in databases and retrieved at runtime according to different requirements. In [19], contexts are represented by context-dependent layers, which can be context-dependent classes or functions. And there are certain base plain classes, the idea of context-oriented programming is implemented by attaching or detaching those layers to the objects of the base class at runtime according to different specifications. Thus in a word, what they refer to as *context* can be viewed as particular instances of `Context` type.

Context Calculus There were similar computations on context, such as in [18, 37]. They developed a typed calculus for contexts i.e., lambda terms with "holes". In addition to ordinary lambda terms, the calculus contains labeled holes, hole abstraction and context application for manipulating first-class contexts. They also defined a type system that precisely specifies the variable-capturing nature of contexts and that keeps track of bound variable renaming. The fundamental theory behind their context calculus is lambda calculus [5, 47], while our context calculus is based on the set theory [27, 14]. And their effort is to find a formal specification for a programming language, and ours is to perform computation on the context entity.

2.6 Summary

This Chapter has provided the background for implementing Lucx in the GIPSY framework. It gives a review of Lucid programming languages, and then introduces Lucx into the family as an extension to GIPL. Except these, it also gives an overview of the GIPSY framework, emphasizing on GIPC compiler layer and the type system, which are highly related to the actual implementation of Lucx.

Chapter 3

Theoretical Basis for Implementing Context Calculus

In this chapter, we present the formal definitions of context and context calculus operators according to Kaiyu Wan's PhD thesis [44]. We derived the notion of *tag set types* according to the concept of context. These definitions form the theoretical basis for our implementation.

3.1 Context Types

Definition 1 Context: *A context c is a finite subset of the relation:* $c \subset \{(d, x) | d \in DIM \wedge x \in T\}$*, where DIM is the set of all possible dimensions, and T is the set of all possible tags.*

Tags are the indices to mark positions in dimensions, more details about tags and tag sets will be discussed later in this chapter.

According to [44], she provides two types of context entities under discussion, which are *simple context* and a *context set*. Simple context is actually what we refer to as `context` in previous Lucid programming languages. Wan extends `context` by introducing *context set* also as first-class context value. Thus here we make `context` a more general form as the parent of both *simple context* and *context set*. In the following subsections, we first provide the definitions of simple context and context set, followed by their syntax and examples.

3.1.1 Simple Context

A *simple context* is a collection of <*dimension* : *tag*> pairs, where there are no two such pairs having the same dimension component. Conceptually, a simple context represents a point in the context space. A simple context having only one pair of <*dimension* : *tag*> is called a *micro context*. It is the building block for all the context types. As shown in Chapter 2, Figure 3, the syntax of simple context is:

$[E : E, ..., E : E]$

Example 1 • `[d:1,e:2]`

3.1.2 Context Set

Although context set has been introduced as a type of context, and also has a set of context calculus operators defined on them, the operational semantic rules of context set have not been integrated into Lucid yet. There is potential need for including context set in our system, thus in the following sections, we still present the definition, syntax and operators of context set as in [44]. See also Chapter 6 and Chapter 7 for limitations and future work on context set.

A context set is a set of simple contexts. Context sets are also often named *non-simple contexts*. Context sets represent regions of the context space, which can be seen as a set of points, considering that the context space is discrete. Formally speaking, a non-simple context is a set of <d : x> mappings that are not defined by a function. As shown in Chapter 2, Figure 3, the syntax of context set is:

$\{E, ..., E\}$, where $E \rightarrow [E : E, ..., E : E]$

Example 2 • `{[x:3,y:4,z:5],[x:3,y:1,z:5]}`

In [44], she also defined a concept of `Box`, which is a set of contexts, all of which have the same dimension set and the tags corresponding to the dimensions in each context satisfy a given constraint. It is a special kind of context set. More issues about box will be discussed in Chapter 6.

3.2 Context Calculus Operators

In the following section, we provide the formal definition for the context calculus operators on simple context and context set. The operators are `isSubContext`, `difference`, `intersection`, `projection`, `hiding`, `override`, and `union`.

Definition 2 *isSubContext*

- *If C_1 and C_2 are simple contexts and every micro context of C_1 is also a micro context of C_2, then C_1 **isSubContext** C_2 returns true.*

  Note that an empty simple context is the sub-context of any simple context. Also note that as the concept of subset in set theory, C_1 could be the proper subset of C_2, or C_1 could be equal to C_2.

- *If S_1 and S_2 are context sets, then S_1 **isSubContext** S_2 returns true if every simple context of S_1 is also a simple context of S_2.*

  Note that an empty context set is the sub-context of any context set. Also note that as the concept of subset in set theory, S_1 could be the proper subset of S_2, or S_1 could be equal to S_2.

Example 3 *Example for **isSubContext** on both simple context and context set.*

- `[d:1,e:2]` ***isSubContext*** `[d:1,e:2,f:3]` $=$ *true*
- `[d:1,e:2]` ***isSubContext*** `[d:1,e:2]` $=$ *true*
- $\emptyset$ ***isSubContext*** `[d:1,e:2]` $=$ *true*
- `{[d:1,e:2],[f:3]}` ***isSubContext*** `{[d:1,e:2],[f:3],[g:4]}` $=$ *true*
- `{[d:1,e:2],[f:3]}` ***isSubContext*** `{[d:1,e:2],[f:3]}` $=$ *true.*

Definition 3 *difference:*

- *If C_1 and C_2 are simple contexts, then C_1 **difference** C_2 returns a simple context that is the collection of all micro contexts which are members of C_1, but not members of C_2: C_1 **difference** $C_2 = \{m_i | m_i \in C_1 \wedge m_i \notin C_2\}$.*

  *Note that if C_1 **isSubContext** C_2 is true, then the returned simple context should be the empty context. Also note that it is valid to "differentiate" two*

simple contexts that have no common micro context; the returned simple context is simply C_1.

- *If* S_1 *and* S_2 *are context sets, this operator returns a context set* S, *where every simple context* $C \in S$ *is computed as* C_1 ***difference*** C_2: $S = S_1$ ***difference*** $S_2 = \{C | C = C_1$ ***difference*** $C_2 \wedge C \neq \emptyset \wedge C_1 \in S_1 \wedge C_2 \in S_2\} \vee S = \emptyset$.

  Note that if for every C_1 *and* C_2, C_1 ***difference*** $C_2 = \emptyset$, *then* S_1 ***difference*** $S_2 = \emptyset$. *However, if there's at least one pair of* C_1 *and* C_2 *where* C_1 ***difference*** $C_2 \neq \emptyset$, *the result doesn't contain empty context. This also applies to the following definitions on context set.*

Example 4 *Example for* ***difference*** *on both simple context and context set.*

- `[d:1,e:2]` ***difference*** `[d:1,f:3]` = `[e:2]`
- `[d:1,e:2]` ***difference*** `[d:1,e:2,f:3]` $= \emptyset$
- `[d:1,e:2]` ***difference*** `[g:4,h:5]` = `[d:1,e:2]`
- `{[d:1,e:2,f:3],[g:4,h:5]}` ***difference*** `{[g:4,h:5],[e:2]}` = `{[d:1,e:2,f:3],[d:1,f:3],[g:4,h:5]]`

Definition 4 ***intersection***

- *If* C_1 *and* C_2 *are simple contexts, then* C_1 ***intersection*** C_2 *returns a new simple context, which is the collection of those micro contexts that belong to both* C_1 *and* C_2: C_1 ***intersection*** $C_2 = \{m_i | m_i \in C_1 \wedge m_i \in C_2\}$.

  Note that if C_1 *and* C_2 *have no common micro contexts, the result is an empty simple context.*

- *If* S_1 *and* S_2 *are context sets, then the resulting intersection set* $S = S_1$ ***intersection*** $S_2 = \{C | C = C_1$ ***intersection*** $C_2 \wedge C \neq \emptyset \wedge C_1 \in S_1 \wedge C_2 \in S_2\} \vee S = \emptyset$

  Note that if for every C_1 *and* C_2, C_1 ***intersection*** $C_2 = \emptyset$, *then* S_1 ***intersection*** $S_2 = \emptyset$. *However, if there's at least one pair of* C_1 *and* C_2 *where* C_1 ***intersection*** $C_2 \neq \emptyset$, *the result doesn't contain empty context.*

Example 5 *Example for* ***intersection*** *on both simple context and context set:*

- [d:1,e:2] ***intersection*** [d:1] = [d:1]

- [d:1,e:2] ***intersection*** [g:4,h:5] $= \emptyset$

- {[d:1,e:2,f:3],[g:4,h:5]} ***intersection*** {[g:4,h:5],[e:2]} =
  {[e:2],[g:4,h:5]}

Definition 5 ***projection:***

- *If C is a simple context and D is a set of dimensions, this operator filters only those micro contexts in C that have their dimensions in set D: C* ***projection*** *$D = \{m | m \in C \wedge dim(m) \in D\}$.*

  Note that if there's no micro context having the same dimension as in the dimension set, the result would be an empty simple context. ***dim(m)*** *returns the dimension of micro context* ***m****.*

- *The projection of a dimension set onto a context set is a context set, which is a collection of all the simple contexts* ***project*** *the dimension set. If S is a context set, D is a dimension set; S' = S* ***projection*** *D = $\{n | n$ = C* ***projection*** *$D \wedge n \neq \emptyset \wedge C \in S\} \vee S' = \emptyset$.*

Example 6 *Example of* ***projection*** *on both simple context and context set:*

- [d:1,e:2,f:3] ***projection*** {d,f} = [d:1,f:3]

- {[d:1,e:2,f:3],[g:4,h:5],[f:4]} ***projection*** {e,f,h} =
  {[e:2,f:3],[h:5],[f:4]}

Definition 6 ***hiding:***

- *If C is a simple context and D is a dimension set, this operator is to remove all the micro contexts in C whose dimensions are in D: C* ***hiding*** *$D = \{m | m \in C \wedge dim(m) \notin D\}$.*

  Note that if ***D*** *contains all the dimensions appeared in* ***C****, the result is an empty simple context. Also note that C* ***projection*** *$D \bigcup C$* ***hiding*** *$D = C$.*

- *For context set S, and dimension set D, the* ***hiding*** *operator constructs a context set S' where S' is obtained by* ***hiding*** *each simple context in S on the dimension set D: $S' = S$* ***hiding*** *$D = \{n | n = C$* ***hiding*** *$D \wedge n \neq \emptyset \wedge C \in S\} \vee S' = \emptyset$.*

Example 7 *Example for* ***hiding*** *on both simple context and context set:*

- `[d:1,e:2,f:3]` ***hiding*** `{d,e}` = `[f:3]`
- `[d:1,e:2,f:3]` ***hiding*** `{g,h}` = `[d:1,e:2,f:3]`
- `[d:1,e:2,f:3]` ***hiding*** `{d,e,f}` $= \emptyset$
- `{[d:1,e:2,f:3],[g:4,h:5],[e:3]}` ***hiding*** `{d,e}` = `{[f:3],[g:4,h:5]}`

Definition 7 ***override:***

- *If C_1 and C_2 are simple contexts, then C_1* ***override*** *C_2 returns a new simple context C, which is the result of the conflict-free union of C_1 and C_2, as defined below: $C = C_1$* ***override*** *$C_2 = \{m|(m \in C_1 \wedge dim(m) \notin dim(C_2)) \vee m \in C_2\}$.*
- *For every pair of context sets S_1, S_2, this operator returns a set of context S, where every context $C \in S$ is computed as C_1* ***override*** *C_2; $C_1 \in S_1$, $C_2 \in S_2$: $S = S_1$* ***override*** *$S_2 = \{C|C = C_1$* ***override*** *$C_2|C_1 \in S_1 \wedge C_2 \in S_2 \wedge C \neq \emptyset\} \vee S = \emptyset$.*

Example 8 *Example of* ***override*** *on both simple context and context set:*

- `[d:1,e:2,f:3]` ***override*** `[e:3]` = `[d:1,e:3,f:3]`
- `[d:1,e:2,f:3]` ***override*** `[e:3,g:4]` = `[d:1,e:3,f:3,g:4]`
- `{[d:1,e:2],[f:3],[g:4,h:5]}` ***override*** `{[d:3],[h:1]}` =
  `{[d:3,e:2],[d:1,e:2,h:1],[f:3,d:3],`
  `[f:3,h:1],[g:4,h:5,d:3],[g:4,h:1]}`

Definition 8 ***union:***

- *If C_1 and C_2 are simple contexts, then C_1* ***union*** *C_2 returns a new simple context C, for every micro context m in C: m is an element of C_1 or m is an element of C_2: C_1* ***union*** *$C_2 = \{m|m \in C_1 \vee (m \in C_2 \wedge m \notin C_1)\}$. Note that if there is at least one pair of micro contexts in C_1 and C_2 sharing the same dimension and these two micro contexts are not equal then the result is a non-simple context, which can be translated into context set: For a non-simple context C, we construct the set $Y = \{y_d = C$* ***projection*** *$\{d\}|d \in$*

*$dim(C)\}$. Denoting the elements of set Y as $y_1, \ldots, y_p$, we construct the set $S(C)$ of simple contexts: $S(C) = \{m_1$ **override** m_2 **override** $\ldots$ **override** $m_p | m_1 \in y1 \wedge m_2 \in y_2 \wedge \ldots m_p \in y_p\}$, The non-simple context is viewed as the set $S(C)$. It is easy to see that $S(C) = \{s \in S | dim(s) = dim(C) \wedge s \subset C\}$*

- *As described earlier for the **union** operator performing on simple contexts, the result could be a non-simple context. If we simply compute union for each pair of simple context inside both context sets, the result may be a set of sets, in other words,* higher-order sets. *Due to unnecessary semantic complexities, we should avoid the occurrence of such sets, thus we define the **union** of two context sets as following to eliminate the possibility of having a higher-order set. If C_1 and C_2 are context sets, then $C = C_1$ **union** C_2 is computed as follows: $D_1 = \{dim(m) \wedge m \in C_1\}, D_2 = \{dim(m) \wedge m \in C_2\}, D_3 = D_1 \bigcap D_2$.*

  1. *Compute $X_1 : X_1 = \{m_i \bigcup (m_j$ **hiding** $D_3) \wedge m_i \in C1 \wedge m_j \in C_2\}$*
  2. *Compute $X_2 : X_2 = \{m_j \bigcup (m_i$ **hiding** $D_3) \wedge m_i \in C1 \wedge m_j \in C_2\}$*
  3. *The result is: $C = X_1 \bigcup X_2$*

Example 9 *Example of **union** on both simple context and context set:*

- `[d:1,e:2]` ***union*** `[f:3,g:4]` = `[d:1,e:2,f:3,g:4]`
- `[d:1,e:2]` ***union*** `[d:3,f:4]` =
  `[d:1,d:3,e:2,f:4]` $\Leftrightarrow$ `{[d:1,e:2,f:4],[d:3,e:2,f:4]}`
- `{[d:1,e:2],[g:4,h:5]}` ***union*** `{[g:4,h:5],[e:3]}` =
  `{[d:1,e:2],[g:4,h:5],[g:4,h:5,d:1],[e:3,d:1],[e:3]}`

3.3 Tag Set Types

A context is essentially a relation between dimensions and tags, the latter being indices used to refer to points defined over these dimensions. In Lucx, such a relation is represented using a collection of `<dimension:tag>` pairs [44]. In such a pair, the current position of the dimension is marked by the tag value, while properties of the tags, such as what are valid tags in this dimension, are bound to the dimension

they index. When a context is declared, a semantic check should be performed to determine whether a tag is valid in the dimension it is used. Therefore, we introduce the notion of *tag set*:

Definition 9 *A tag set is a collection of all possible tags attached to a dimension.*

In earlier versions of Lucid programming languages, tag set was assumed to be the ordered infinite set of natural numbers, and was never explicitly declared as such. However, as we explore more domains of application, natural numbers can no longer represent tag values sufficiently.

Assume that we want to construct an application of `Student Course Management System`: Some requirements related to our concern are: a student can take courses in different fields, such as computer science, art, business and so on. And in a particular field, eg. computer science, there are also different branches, such as database, programming language study, software engineering and so on.

If we set our focus to a particular school, eg. Concordia, and we define programming language courses offered in computer science as our dimension, then the tags inside this dimension are finite.

If we simply define *course* as our dimension without any restriction, meaning that it could contain any course that had been taught in the past, is being given at present or will be offered in the future, anywhere around the world then the tag set would be infinite.

Now we go back to the dimension *programming language courses*(plc), assume that there are 3 courses offered under this category: Introduction To Programming Languages(COMP001), Programming in C++(COMP002) and Advanced Programming Practice(COMP003). They must be taken following the order as the way they are listed, because the previous one is the prerequisite course for the next one. Thus we have the program as shown in Listing 3.1: We use identifier *isFinished* to represent the state of a student who takes programming language courses. "0" represents that he has finished all the courses offered in this area, "1" otherwise. The context notation `[plc : COMP002]` denotes that he is currently taking COMP002. The tag set is explicitly defined as "ordered finite" with the prerequisite order. *iseod* is an intensional operator which returns true if the current tag is the end of its dimension. *next* and # are also intensional operators returning the next tag and current tag of a dimension respectively.

```
isFinished @ [plc : COMP002]
where
  dimension plc : ordered finite {COMP001, COMP002, COMP003};
  isFinished = if(iseod(next(#plc))) then 0 else 1;
end
```

Listing 3.1: An Example For Ordered Finite Tag Set

As discussed earlier, it is also possible for a student to take courses in different fields, thus we can define a new dimension *course field*(cf) to represent all types of courses offered in Concordia. However, there's no order restricting the student to take which kind of courses first, so tags inside the tag set are unordered. Listing 3.2 shows unordered tag set. *pickField* returns the choice of which field of courses he would take currently. He's free to choose any type, there's no order to restrict him, this program just simulates that he wants to choose "ComputerScience".

```
    pickField
      where
        dimension cf : unordered finite {Art, Business, ComputerScience,
            Engineering, Science}
        pickField = ComputerScience;
      end
```

Listing 3.2: An Example For Unordered Finite Tag Set

So to sum up, it is clear that the properties of natural numbers set–ordered and infinite–are not sufficient to include all the possibilities of tag set types. Additionally, the tag value can actually be of `string` or other types, not only `int`, as shown in our `plc` dimension. Therefore, it is necessary to introduce the keywords "ordered/unordered" and "finite/infinite" to determine the types of tag set associated with dimensions upon declaration. Note that more keywords might also be included in the future, here we only present those to the scope of our knowledge and the current application. Following are the definitions for those keywords when they are used to determine the type of a tag set. As tag sets are in fact sets, we define the following terms as of set theory [27, 14]:

Definition 10 Ordered Set: *A set on which a relation R satisfies the following three properties:*

1. *Reflexive: For any $a \in S$, we have aRa*

2. *Antisymmetric: If aRb and bRa, then $a = b$*

3. *Transitive: If aRb and bRc, then aRc*

Definition 11 Unordered Set: *A set which is not ordered is called an unordered set.*

Definition 12 Finite Set: *A set I is called finite and more strictly, inductive, if there exists a positive integer n such that I contains just n members. The empty set $\emptyset$ is also called finite.*

Definition 13 Infinite Set: *A set, which is not finite is called an infinite set.*

Out of backward compatibility with previous versions of Lucid, we assume that the default tag set is the set of natural numbers, and its order is as with the order of natural numbers. If other tag sets are to be applied, the programmer must specify them by explicitly specifying and/or enumerating the tag set and its order, as discussed further in this thesis.

After introducing the keywords above, we can have four types of tag sets: *ordered finite tag set*, *ordered infinite tag set*, *unordered finite tag set* and *unordered infinite tag set*. The limitation of implementing *unordered infinite tag set* will be discussed later.

In the following sections, we provide definitions for those types, syntax of their expressions, when these expressions are applied and examples of them.

Here we use $\mathbb{Z}$ to denote the set of all integers; S to denote the tag set. We define $l, u, p, e \in \mathbb{Z}$ as integers to denote the lower boundary (l), upper boundary (u), step (p) and any element (e) of the tag set when describing it syntactically. Also note that $prev(e)$ returns the element previous to the current element under discussion.

3.3.1 Ordered Finite Tag Set

For this type, tags inside the tag set are ordered and finite. In our approach, this tag set can be expressed by listing all the tag values, or giving the lower and upper boundaries when there are too many elements to be listed. If it is the latter case, for now, we only consider subsets of integers.

Expression Type 1 `dimension` *id:* `ordered finite` $\{E, \ldots, E\}$

- *All the tag values inside the tag set are enumerated and their order is implicitly defined as the order in which they are enumerated.*

Expression Type 2 `dimension` *id:* `ordered finite` $\{l$ `to` $u\}$

- $S \subset \mathbb{Z} = \{e | e - prev(e) = 1 \wedge l \leq e \leq u\}$

Expression Type 3 `dimension` *id:* `ordered finite` $\{l$ `to` u `step` $p\}$

- $S \subset \mathbb{Z} = \{e | e - prev(e) = p \wedge l \leq e \leq u \wedge p > 0\}$
- $S \subset \mathbb{Z} = \{e | e - prev(e) = p \wedge u \leq e \leq l \wedge p < 0\}$

Example 10 *The following examples correspond to the syntactic expressions listed above, respectively.*

- *dimension d : ordered finite {rat, bull, tiger, rabbit}*
- *dimension d : ordered finite {1 to 100}*
- *dimension d : ordered finite {2 to 100 step 2}*

3.3.2 Ordered Infinite Tag Set

For this type, tags inside the tag set are ordered and infinite. For now, we only consider subsets of integers. Note, in what follows `INF-` and `INF+` stand for minus infinity ($-\infty$) and plus infinity ($+\infty$) respectively.

Expression Type 4 `dimension` *id:* `ordered infinite` $\{l$ `to INF+`$\}$

- $S \subset \mathbb{Z} = \{e | e - prev(e) = 1 \wedge l \leq e\}$

Expression Type 5 `dimension` *id:* `ordered infinite` $\{l$ `to INF+ step` $p\}$

- $S \subset \mathbb{Z} = \{e | e - prev(e) = p \wedge l \leq e \wedge p > 0\}$

Expression Type 6 `dimension` *id:* `ordered infinite` $\{$`INF- to` $u\}$

- $S \subset \mathbb{Z} = \{e | e - prev(e) = 1 \wedge e \leq u\}$

Expression Type 7 `dimension` *id:* `ordered infinite` $\{$`INF- to` u `step` $p\}$

- $S \subset \mathbb{Z} = \{e | e - prev(e) = p \wedge e \leq u \wedge p > 0\}$

Expression Type 8 `dimension` *id:* `ordered infinite` {`INF- to INF+`}

- *This represents the whole stream of integers, from minus infinity to plus infinity.*

Note that the default tag set is $\mathbb{N}^{+}$, which is also within this type. Either by leaving the tag set declaration part empty or specifying {`0 to INF+`}, they both refer to the set of natural numbers.

Example 11 *The following examples correspond to the syntactic expressions listed above, respectively.*

- *dimension d : ordered infinite {2 to INF+}*
- *dimension d : ordered infinite {2 to INF+ step 2}*
- *dimension d : ordered infinite {INF- to 100}*
- *dimension d : ordered infinite {INF- to 100 step 2}*
- *dimension d : ordered infinite {INF- to INF+}*

3.3.3 Unordered Finite Tag Set

Tags of this type are unordered and finite.

Expression Type 9 `dimension` *id:* `unordered finite` $\{E, \ldots, E\}$

Example 12 *The following example corresponds to the syntactical expression above.*

- *dimension d: unordered finite {red, yellow, blue}*

3.3.4 Unordered Infinite Tag Set

Tags of this type are unordered and infinite. Please refer to Chapter 6 for more details about the limitation in defining and implementing this type.

3.4 Summary

In this chapter, we first introduce the definition of context, which is a relation between dimension and tag, then we categorize it into simple context and context set. After that, a set of context calculus operators is specified. And in order to clearly define context, we also introduce the notion of tag set and its types. All these definitions and theories form the basis of implementing context and context calculus of Lucx in the GIPSY.

Chapter 4

Implementation

After providing the formal definitions of context, context calculus and tag set types, now we present the design and implementation of integrating Lucx into the GIPSY framework. As described in Chapter 1, the integration of Lucx mainly consists of two major tasks: First, the construction of Lucx compiler in the GIPC. Second, the implementation of context and context calculus.

4.1 Integration of Lucx Compiler

As described in Chapter 2, the design of GIPC provides a generic and dynamic infrastructure that enables the compilation of hybrid programs composed of Lucid part and sequential part. Figure 7 shows the structure of GIPC. As one of the SIPLs, Lucx programs fit into the Lucid part, in other words, intensional part of the hybrid GIPSY program. Its compiler should be embedded into the SIPL parser unit, where all the parsers for SIPLs reside. Thus after the preliminary analysis by `Preprocessor`, which splits a GIPSY program into chunks, Lucx code segment can be parsed by its own parser and the initial AST is generated.

4.1.1 Lucx Parser

We use JavaCC(Java Compiler Compiler) [42] to generate parser for Lucx. Here we first give a brief overview of JavaCC in order to better explain the syntax specifications and so on.

JavaCC along with the built-up JJTree, is the tool the GIPSY project is relying

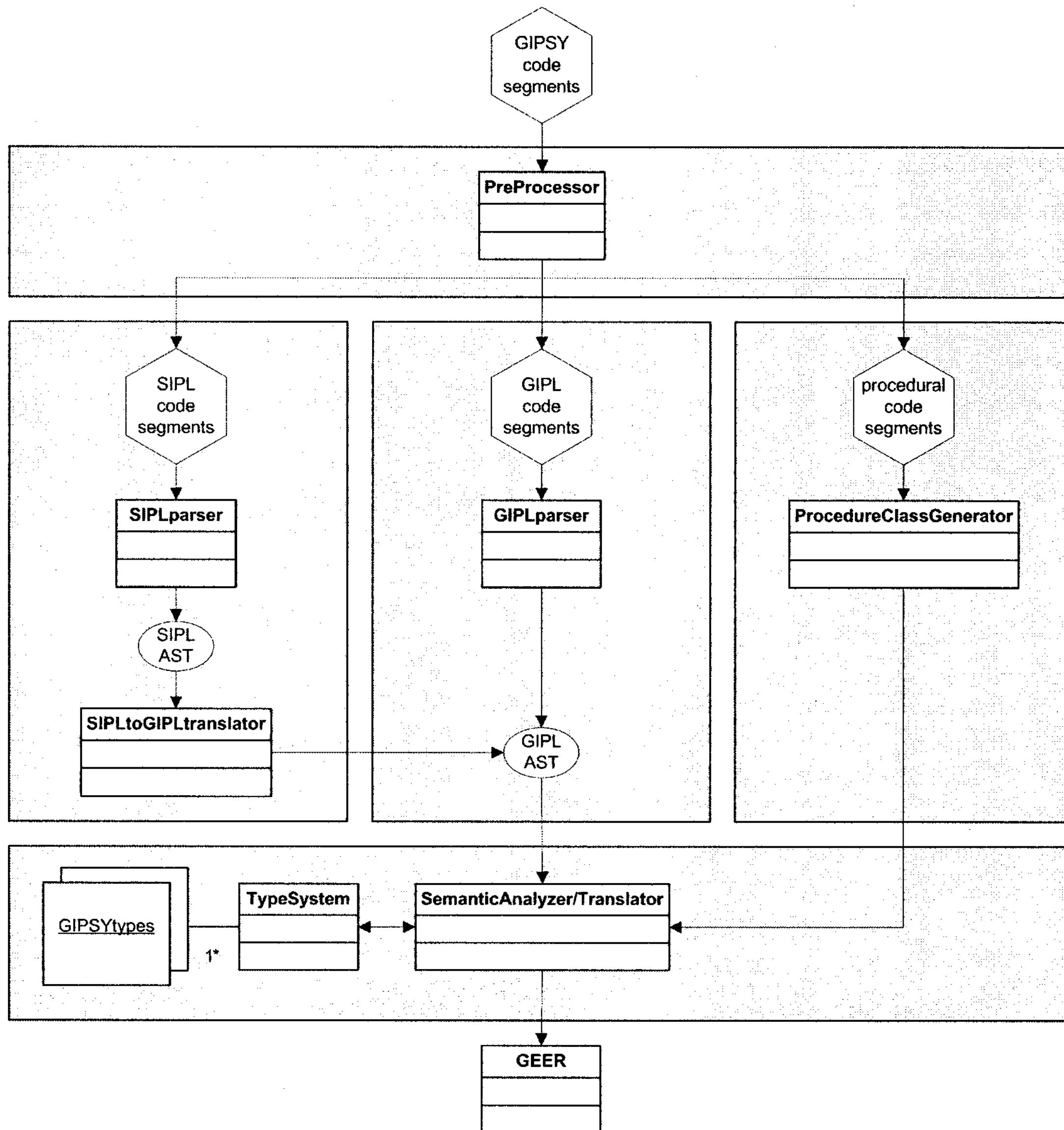

Figure 7: GIPC Structure

on since the first implementation [35] to create Java-language parsers and ASTs for source grammar files. The Java Compiler Compiler tool implements the same idea for Java, as do `lex/yacc` [28] (or `flex/bison`) for C – reading a source grammar they produce a parser that complies with this grammar and gives you a handle on the root of the abstract syntax tree. The GIPL, Indexical Lucid, JLucid, Objective Lucid, `PreprocessorParser`, and `DFGGenerator` parsers are generated with the JavaCC/JJTree parser generation tools. JavaCC requires a grammar specification of the target programming language, written in the syntax of JavaCC. Thus we first provide the concrete syntax of Lucx with the expressions for context, context calculus operators and tag set, as shown in Figure 8.

Elimination of Left Recursion JavaCC is a LL(K) [28] parser generator, although left recursion could be eliminated by setting a particular LOOKAHEAD integer value, the efficiency of parsing a source program would then be decreased, since the parser has to 'look ahead' for more than one token when there's an ambiguity in the grammar. Thus we modified the grammar to remove the left recursion[32] to make Lucx grammar LL(1). Following is some examples of removing left recursion:

Example 13 *An Example of eliminating left recursion*
tagset ::= `ordered finite` *(*{ `INTEGER to INTEGER` }
| { `INTEGER to INTEGER step INTEGER` } *)*
| `ordered infinite` *(*{ `INTEGER to INF+` }
| { `INTEGER to INF+ step INTEGER` }
| { `INF- to INTEGER` }
| { `INF- to INTEGER step INTEGER` }
| { `INF- to INF+` }*)*

After modification, the grammar changed into:
tagset ::= `ordered finite` { `INTEGER to INTEGER` *(*ϵ | `step INTEGER`*)* }
| `ordered infinite` *(*{ `INTEGER to INF+` *(* ϵ | `step INTEGER` *)* }
| { `INF- to`*(*`INTEGER` *(*ϵ | `step INTEGER` *)* | `INF+` *)* }*)*

Building up the Lucx Syntax Tree Although JavaCC is a top-down parser, JJTree constructs the parse tree from the bottom up. To do this it uses a stack where it pushes nodes after they have been created. When it finds a parent for them, it pops

```
            E  ::=  (if E then E else E) E1 | # E E1 | Term E1
         Term  ::=  factor Term1
        Term1  ::=  ε | (* | / | % | and) Term Term1
           E1  ::=  ε |(+ | - | or) Term E1
                |   ( < | > | >= | <= | == | != ) E E1
                |   @ E E1
                |   where Q E1
                |   (fby | asa | upon
                |   isSubContext | difference | intersection
                |   projection | hiding | override | union) E1
                |   Tail E1
         Tail  ::=  ε | [E(,E)*](E(,E)*)
       factor  ::=  ID | INTEGER | FLOAT | STRING | ( E )
                |   unary
                |   context
      context  ::=  context _ simple | context _ set
context_simple ::=  [micro_context (, micro_context)* ]
 micro_context ::=  E : E
   context_set ::=  { context_simple (, context_simple)*}
         unary ::=  (first | next | prev | iseod) E
             Q ::=  (dimension ID (, ID)* tagset ;)(Q)* | (ID QTail = E ;)(Q)*
         QTail ::=  ε | [ ID (,ID)* ](ID (,ID)*)
        tagset ::=  ordered finite ({ E (, E)*}
                |   { INTEGER to INTEGER }
                |   { INTEGER to INTEGER step INTEGER } )
                |   ordered infinite ({ INTEGER to INF+ }
                |   { INTEGER to INF+ step INTEGER }
                |   { INF- to INTEGER }
                |   { INF- to INTEGER step INTEGER }
                |   { INF- to INF+ })
                |   unordered finite { E(, E)*}
```

Figure 8: Concrete Lucx Syntax

the children from the stack and adds them to the parent, and finally pushes the new parent node back. The stack is open, which means that you have access to it from within grammar actions: you can push, pop and otherwise manipulate its contents however you feel appropriate.

Each node is associated with a node scope. User actions within this scope can access the node under construction by using the special identifier `jjtThis` to refer to the node. This identifier is implicitly declared to be of the correct type for the node, so any fields and methods that the node has can be easily accessed.

A scope is the expansion unit immediately preceding the node decoration. This can be a parenthesized expression. When the production signature is decorated (perhaps implicitly with the default node), the scope is the entire right hand side of the production including its declaration block.

By specifying grammar expansions and node scopes for each Lucx non-terminal, an Lucx AST is built. Listing 4.1 is an example of building the node `E`. It first lists all the grammar expansions on the right hand side of `E`, then in the node scope section, it specifies the tree building operations.

4.1.2 Extension of Semantic Analyzer

In [48], Wu provided the implementation of Semantic Analyzer with well-established semantic analysis. It was designed to recognize only GIPL, as other SIPLs were all supposed to be translated into GIPL. However, because of Lucx's uniqueness in context and context calculus, the translation route cannot be applied to Lucx. Thus we adapt the Semantic Analyzer with the minimum changes to make it aware of the concepts of context and context calculus.

The Semantic Analyzer has a mechanism of traversing the AST generated by the front-end layer in a top-down, depth first manner. It uses the Dictionary, which keeps all the identifiers and their attributes to do type checking, rank analysis, and function elimination [48]. Finally, the filled Dictionary, which is essentially the representation of compiled AST, will be fed to GEE as the resource for final execution.

Type Checking for Lucx Except the common type checkings as within the other Lucid programming languages, Lucx has its unique type checking schema because of the notion of context and context calculus operators. Note that here in the Semantic

```
void E() : {}
{
        try
        {
                (ID() <ASSIGN> E()E1() ) #ASSIGN E1()   //Grammar rules

                |(<IF> E() <THEN> E() <ELSE> E() <FI>) #IF E1()

                |(<WHEN> E()) #HASH E1()

                |Term() E1()

                |(( <PLUS> #POSI | <MINUS> #NEGE ) Term() )
                {
                 // Operation on the nodes.
                 SimpleNode midNode1 = (SimpleNode)jjtree.popNode();
                 SimpleNode midNode2 = (SimpleNode)jjtree.popNode();
                 SimpleNode node = sign(midNode1, midNode2);
                 jjtree.pushNode(node);
                }
                 E1()
        }
        catch(ParseException e)
        {
                countErrors();
                System.err.println("Lucx Parser: " + e.toString());
        }
}
```

Listing 4.1: An example of building up JJTNode

Analyzer at compile time, we only deal with simple expressions because we could have encountered some complex expressions, initiating other intensional commands requiring even distributed evaluation, which cannot be determined at compile time. In those cases, the semantic analysis should be delayed to the runtime execution engine. For example, if we have an expression of $[\ d : (c \ @\ [f : 2])]$, which is a simple context whose tag value requires the result of evaluating another intensional expression. The tag type cannot be determined at compile time, thus we defer type checking for these complex expressions to the engine side. The following is the list of type checking that should be performed on Lucx programs.

- Tag Value Validity: to check if the tag value in a given context is valid for the tag set declared.
- Operands Validity: to check if the operands, in other words, context objects are

valid for the computing context calculus operator.

For the first checking, we define a method called `checkTagValueInContext(SimpleNode pTagValue)` inside the SemanticAnalyzer class, which takes a tag value node in a micro context as parameter. This method makes a reference to the tree, finding the dimension identifier in the micro context, looking it up in the Dictionary, and gets the tag set declared in that dimension. Then we instantiate an object of the tag set with all its attributes. After that, we call the set membership method defined in the tag set class to do the actual semantic checking. The `checkTagValueInContext(SimpleNode pTagValue)` is called in the main `check()` method, which traverses the tree and does type checkings. Thus every time the `check()` method traverses the tree and meets `MICRO_CONTEXT` node, it first checks the validity of the dimension identifier, then if the tag value node is a simple expression, for example, a constant, in other words, string or numeric literal, `checkTagValueInContext(SimpleNode pTagValue)` is called.

For the second checking, when we start traversing the tree and meet the context calculus operator nodes, it checks to see if the two operands are of the same type, for example, the operands of `difference` should be both `Simple Context` or `Context Set` and also note that certain operators such as `projection` and `hiding` require the operands to be `context` and `DimensionSet`.

The current engine cannot deal with user defined functions directly, thus we have a mechanism of function elimination [48]. Because Lucx semantics is an extension to the GIPL semantics, the function in Lucx would also follow this routine. However, in the current function elimination mechanism, recursive functions cannot be managed. Our team is working on design a new generation of runtime execution engine to actually support all the functions instead of flattening them.

4.2 Design and Implementation of Context Classes

Because of the consideration of execution efficiency and the importance of context, we decide to define context as a data type, with all the context calculus operators implemented inside it. The data type is represented by a Java class as any other types inside the type system. Now we present the original design of GIPSY Type System by Serguei Mokhov.

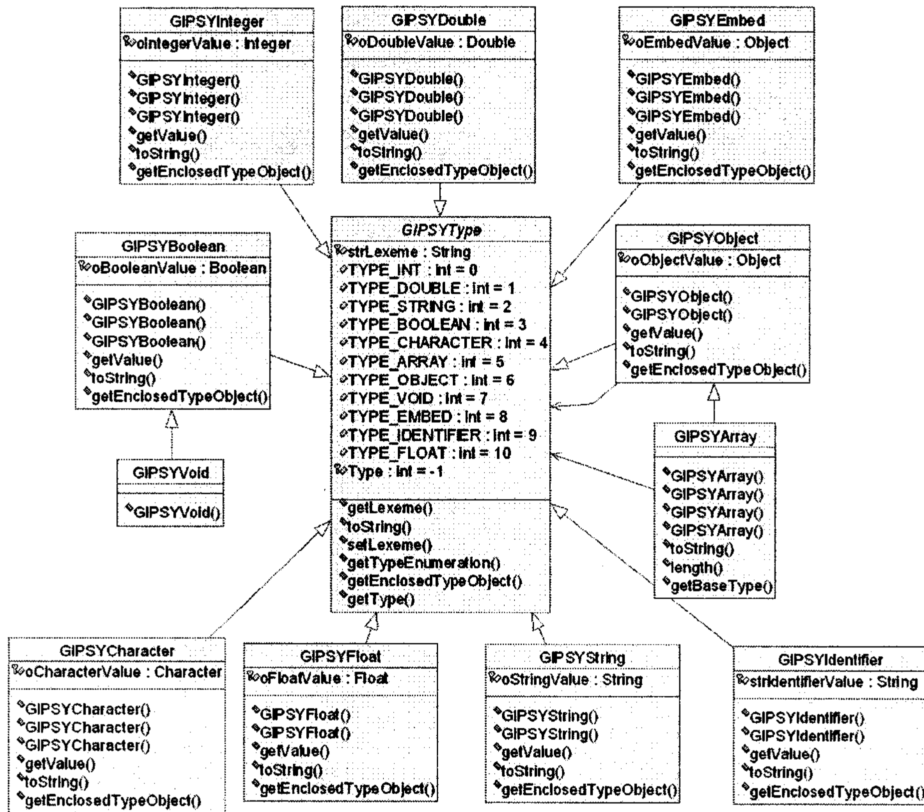

Figure 9: GIPSY Type System.

4.2.1 An Overview of the GIPSY Type System

According to [31] The structure of GIPSY Type System before introducing `context` is shown as Figure 9

Each class is prefixed with `GIPSY` to avoid possible confusion with similar definitions in the `java.lang` package. Primitive types, such as `Long`, `Float`, etc. are wrapped around the corresponding Java object wrapper classes. Every class keeps a lexeme (a lexical representation) of the corresponding type in a GIPSY program and overrides `toString()` to show the lexeme and the contained value. These types are extensively used by the `Preprocessor`, imperative and intensional (for constants) compilers, the `SequentialThreadGenerator`, and `SemanticAnalyzer` for the general type of GIPSY program processing, and by the GEE's `Executor` [31]. All the GIPSY data types extend the generic GIPSYType.

4.2.2 Design of Context Classes

As presented in Chapter 3, there are two types of context: `Simple Context` and `Context Set`. Note that as described in Chapter 3, the semantics of `Context Set` has not been proved in Lucid and thus integreted in GIPL semantics. However, we still embed it as a type variant because of potential future need, see Chapter 6 for more details about this issue.

We define the abstract class `Context` as the parent class for both `Simple Context` and `Context Set`. Inside the parent class, there's an attribute *oSet*, which keeps all the `Micro Context` elements for `Simple Context` and `Simple Context` elements for `Context Set`. It also has a generic method *size()* to return the size of *oSet*, which represents the size of the `Context`. Note that if the *size()*==0, it represents an empty context. There's also a set of abstract signatures for context calculus operators. `add()`, `remove()` and `getChild()` are operators used to manipulate the components.

`MicroContext` is the atomic component of `SimpleContext`. It consists of an object of `Dimension` and an object of `GIPSYType` to represent the tag value.

`SimpleContext` is the child of `Context`, it has the actual implementation of all the context calculus operators on simple context. It is also the component of `Context Set`.

`Context Set` also inherits `Context` and overwrites the context calculus operators. It is the composition of `SimpleContext`.

According to the inheritance hierarchy and component and composition relationship among `MicroContext`, `SimpleContext` and `ContextSet`, we apply *composite design pattern* [29, 38, 20] here, where `SimpleContext` is the leaf, `Context` is the component and `Context Set` is the composition. Figure 10 shows the structure of context classes. The composition `Context Set` can add or remove component dynamically. According to the definitions of context calculus operators on context set, most of them rely on implementation of their counterparts on simple context. Thus by applying composite pattern, the implementations of them can be achieved by calling the actual implementation of context calculus operators on `SimpleContext`.

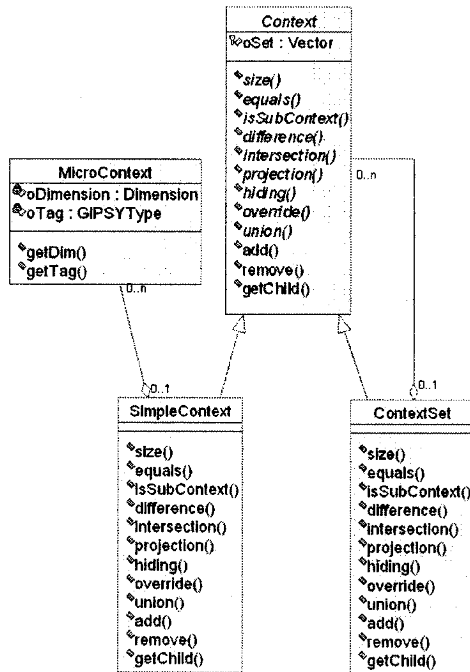

Figure 10: Context Classes

4.3 Implementing Context Calculus in the GIPSY

After designing context classes, now we have a solid container of the context calculus operators. In this section, we provide the algorithms for implementing those operators, which are *isSubContext*, *difference*, *intersection*, *hiding*, *projection*, *override* and *union*. Those operators apply on both Simple Context and Context Set. Context Set is the composition of Simple Context, as specified in Chapter 3, the definitions of these operators on Context Sets are mostly performing the computation on each pair of Simple Context components. In this section, we don't list all the algorithms for implementing context calculus operators on Context Set, please refer to appendix for more details.

4.3.1 isSubContext

This method takes an object of Simple Context as parameters and returns a boolean value. If `this` context is an empty Simple Context, then the result is always true

because empty context is the sub context of any context. And if the parameter contains less Micro Context elements than `this` context, the result is always false because no subset contains more elements than the superset. Otherwise, if every Micro Context in `this` context has a match in the parameter, the result returns true; otherwise, false.

```
boolean isSubContext(SimpleContext pC)
{
  if(this.size() == 0)
    return true;
  if(pC.size() < this.size()) //It includes the case where pC.size() ==
      0
    return false;
  else{
     boolean flag;
     for(int i = 0; i < this.size(); i++){
       flag=false;
       for(int j = 0; j < pC.size(); j++){
         if(pC.micro_context(j) == this.micro_context(i)){
           flag=true;
           break;
         }
       }
       if(flag == false)
         break;
     }
     return flag;
  }
}
```

Listing 4.2: Algorithm for implementing *isSubContext* on simple context

4.3.2 difference

This method takes an object of Simple Context as parameter and returns a new computed Simple Context object. The result is basically Micro Contexts in `this` Simple Context but not in the parameter. We use an object of Simple Context to keep the result of computation. Initially, it is a copy of `this` Simple Context. Here we invoke the *clone()* method in Java, because later on we would change the content of the result object. All the Java objects are reference variables, if we simply use the assign operator '=', the two variables(the first parameter and the result) will make reference to the same memory location, thus either one changes, the other would be

affected. That's not safe for both sides. So we use *clone()* to make a copy of `this` Simple Context. Then it compares each Micro Context in `this` Simple Context with that of the parameter, if a match is found, the current Micro Context is removed from the result Simple Context. Note that if every Micro Context in `this` Simple Context is also inside the parameter, the result is an empty Simple Context.

```
SimpleContext difference(SimpleContext pC){
  SimpleContext result=this.clone();
    if(this.size() == 0)
     return result;
    if(pC.size() == 0)
     return result; //return the result immediately without going into
         the loop
    for(int i = 0; i < this.size(); i++){
     for(int j = 0; j < pC.size(); j++){
      if(pC.micro_context(j)==this.micro_context(i)){
       result.remove(this.micro_context(i));
      }
     }
    }
  return result;
}
```

Listing 4.3: Algorithm for implementing *difference* on simple context

Recall that the *difference* set of two context sets is computed by 'differentiating' every pair of simple contexts in both of the context sets. If all the resulting simple contexts are empty context, then the result is an empty context set, but if there's at least one non-empty simple context, the result doesn't contain empty simple context. Here we give the algorithm as an example for all the other operators, as they are implemented in the similar way.

4.3.3 intersection

This method takes an object of Simple Context as parameter and returns a new computed Simple Context object. By definition, the complementary set of *intersection* is *difference*, thus we design the algorithm as shown in the following listing.

4.3.4 projection

This method takes a set of Dimension objects as parameter and returns a new computed object of Simple Context. It compares the Dimension objects inside the set

```
ContextSet difference(ContextSet pS){
  ContextSet result=new ContextSet();
  for(int i = 0; i < this.size(); i++){
    for(int j = 0; j < pS.size(); j++){
      SimpleContext tempResult=difference(this.simple_context(i), pS.
          simple_context(j));
      if(tempResult.size() != 0) //if all the tempResult.size()==0, the
          result is an empty context
        result.add(tempResult);
    }
  }
  return result;
}
```

Listing 4.4: Algorithm for implementing *difference* on context set

```
SimpleContext intersection(SimpleContext pC){
  return this.difference(this.difference(pC));
}
```

Listing 4.5: Algorithm for implementing *intersection* on simple context

with those in the context object, if they are equal, the Micro Context would be added to the result Simple Context.

```
SimpleContext projection(Vector dimSet){
  SimpleContext result = new SimpleContext;
  for(int i = 0; i < dimSet.size(); i++){
    for(int j = 0; j < this.size(); j++){
      if(this.micro_context(j).dimension == dimSet.dimension(i))
        result.add(this.micro_context(j));
    }
  }
  return result;
}
```

Listing 4.6: Algorithm for implementing *projection* on simple context

4.3.5 hiding

This method takes an Object of Simple Context and a set of Dimension objects as parameter and returns a new computed object of Simple Context. According to the definition, the complementary set of *hiding* is *projection*, thus we implement *hiding* as follows.

```
SimpleContext hiding(SimpleContext c, DimensionSet dimSet){
  return(difference(c, projection(c, dimSet)));
}
```

Listing 4.7: Algorithm for implementing *hiding* on simple context

4.3.6 override

This method takes an object of Simple Context as parameter and returns a new computed object of Simple Context. It's composed of three parts: micro contexts in the parameter whose dimensions are common in both simple contexts, micro context in `this` Simple Context and in the parameter whose dimensions are unique.

```
SimpleContext override(SimpleContext pC){
  SimpleContext result = new SimpleContext();
  boolean flag=false;
  SimpleContext uniqueMCInC2=pC.clone();//micro contexts whose
      dimensions are unique in pC
  for(int i = 0; i < this.size(); i++){
    for(int j = 0; j < pC.size(); j++){
      if(this.micro_context(i).dimension == pC.micro_context(j).
          dimension)
      {
        flag=true;
        result.add(pC.micro_context(j));
        uniqueMCInC2.remove(pC.micro_context(j));
      }
    }
    if(flag == false)
      result.add(this.micro_context(i)); //Add the micro contexts whose
          dimensions are unique in this
    flag=false;
  }
  for(int k = 0; k < uniqueMCInC2.size(); k++)
  {
    result.add(uniqueMCInC2.micro_context(k));
  }
  return result;
}
```

Listing 4.8: Algorithm for implementing *override* on simple context

4.3.7 union

This method takes an object of Simple Context as parameter and returns a new computed object of Simple Context or Context Set. There's possibility of involving

elimination of non-simple context, for example [f:1, e:1, d:2] *union* [e:2, d:1, t:4]. If there's no Dimension object in common, then the result is simply the combination of two simple contexts. However, if there is, the result should be an object of Context Set. We define a set of helping methods to translate the non-simple context into Context Set. *union* on context sets are different than the other context calculus operators on context set because it has a mechanism of eliminating higher-order set. Because of the lengths of the algorithms, we list them in appendix.

4.4 Design and Implementation of Tag Set Classes

After we derive the notion of tag set from the definition of context, it should also be a type variant inside the type system, because the instantiation of context objects depend on the tag set types. Thus in the following subsections, we present the design and implementation of tag set classes inside the GIPSY.

4.4.1 Design of Tag Set Classes

As stated above, there are four kinds of tag sets. They are organized as shown in Figure 11. The `TagSet` class is an abstract class and it's the parent of all of tag set classes. It has a data field *iExpressionType* to distinguish different expressions under the same tag set type(eg. There are three kinds of expressions under `Ordered Finite Tag Set`). It also has two abstract method signatures of *equals()* and *isInTagSet()*. The latter is the set membership method that must be implemented over all types of tag sets.

There is also a group of interfaces for keeping the type information, for example, the class `OrderedFiniteTagSet` should implement the `IOrdered` and `IFinite` interfaces. Such mechanism also provides the facility of adding and defining proper operators inside proper tag set classes. Such as *getNext(poTag)*, which takes a tag object as parameter and returns the next tag value in the dimension, should be valid only for ordered sets. Then only the tag set classes implement the `IOrdered` interface should give the concrete implementation for this method. And since the length of finite set can be determined, in `IFinite` interface, a signature of *size()* is specified.

All the concrete tag set type classes inherits the `TagSet` class and have their own data fields and operators defined as the following subsection.

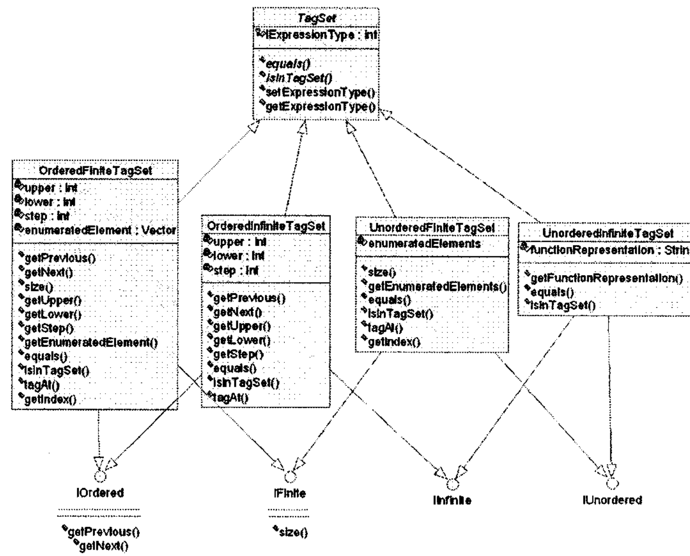

Figure 11: Tag Set Classes

4.4.2 Implementation of Operators on Tag Sets

A tag set is attached to a dimension, it specifies all the possible tags in that dimension. When a context is declared as `<dimension:tag>` pairs, certain checks should be performed to determine if the tag value is valid for the dimension. Thus it is necessary to define set membership method over all tag set types. Here we also provide equality method for each type of tag sets. And because the nature of each type of tag set varies, there are different operators allowed on certain tag sets. For example, it is reasonable to perform `getNext()` or `getPrevious()`, namely tag set index switching operators, on ordered tag set, however, the performance of such operators on unordered tag set would be invalid since there's no fixed order in an unordered sequence. As described earlier in Chapter 2, there are four types of tag set, which are `ordered finite tag set`, `ordered infinite tag set`, `unordered finite tag set` and `unordered infinite tag set`. Here we still follow this category while specifying the implementations.

4.4.2.1 Ordered Finite Tag Set

There are three kinds of expressions for this tag set type, we use *iExpressionType* == 0 to represent expression as `dimension d :ordered finite {rat, bull, tiger,`

`rabbit`}; *iExpressionType* == 1 to represent expression as `dimension d : ordered finite` {`1 to 100`}; and *iExpressionType* == 2 to represent expression as `dimension d : ordered finite` {`2 to 100 step 2`}. We provide set membership method, equality method and the applicable tag set index switching method for each of them. Also note that for the set membership method and index switching methods, since an actual tag value parameter could be any kind of object of GIPSY types, we make it GIPSYType [31]. And because this methods are implemented inside the tag set type class, we use `this` to represent the tag set itself.

Set Membership

- If *iExpressionType* == 0, *isInTagSet()* returns true if and only if the given parameter is equal to one of the tag values inside the tag set as enumerated.
- If *iExpressionType* == 1, *isInTagSet()* returns true if and only if the given parameter is greater than or equal to the lower boundary and smaller than or equal to the upper boundary.
- If *iExpressionType* == 2, *isInTagSet()* returns true if and only if the given parameter *para* is greater than or equal to the lower boundary l, and smaller than or equal to the upper boundary, if the step is positive; or smaller than or equal to the lower boundary and greater than or equal to the upper boundary if the step is negative; and that $((para - l) \texttt{ mod } p) = 0$ in both cases, where p is the step specified.

The algorithm is giving in Listing 4.9

Equality

- If *iExpressionType* == 0, *equals()* returns true if and only if all the enumerated elements in both tag sets are equal. Note that because we use `Vector` in Java to contain the enumerated elements, we call the *equals* method of `Vector` to implement our equality method. Since the *equals* method in `Vector` is implemented based on the order that the elements are listed, it is appropriate for our requirement.

```
boolean isInTagSet(GIPSYType poTag)
{
  boolean result = false;
  switch(this.iExpressionType){
    case : 0 {
      for(int i = 0; i < this.size(); i++){
        if(poTag == this.tagAt(i))
         result = true;
      }
      break;
    }
    case : 1 {
      if(poTag >= lower && poTag <= upper)
        result = true;
      break;
    }
    case : 2 {
      if(step > 0){
        if(poTag >= lower && poTag <= upper && (poTag-lower) mod step
            ==0)
          result = true;
      }
      else {
        if(poTag <= lower && poTag >= upper && (poTag-lower) mod step
            ==0)
      result = true;
      }

      break;
    }
    return result;
  }
}
```

Listing 4.9: Algorithm for implementing `isInTagSet` on ordered finite tag set

- If *iExpressionType* == 1, *equals()* returns true if and only if the lower and upper boundaries of the two tag sets are equal respectively.
- If *iExpressionType* == 2, *equals()* returns true if and only if the lower and upper boundaries and the steps of two tag sets are equal respectively.

The algorithm is listed in Listing 4.10.

```
boolean equals(Object otherObject)
{
  boolean result = false;
  if(otherObject instanceof OrderedFiniteTagSet){
   switch(this.iExpressionType){
    case : 0 {
      if((OrderedFiniteTagSet)otherObject.expressionType == 0 &&
      (OrderedFiniteTagSet)otherObject.getEnumeratedElements().equals(
          this.getEnumeratedElements()))
        result = true;
      break;
    }
    case : 1 {
      if((OrderedFiniteTagSet)otherObject.expressionType == 1 &&
      (OrderedFiniteTagSet)otherObject.getLower()==this.getLowewr() &&
      (OrderedFiniteTagSet)otherObject.getUpper()==this.getUpper())
        result = true;
      break;
    }
    case : 2 {
      if((OrderedFiniteTagSet)otherObject.expressionType == 2 &&
     (OrderedFiniteTagSet)otherObject.getLower()==this.getLower() &&
     (OrderedFiniteTagSet)otherObject.getUpper()==this.getUpper() &&
     (OrderedFiniteTagSet)otherObject.getStep()==this.getStep())
       result = true;
     break;
   }
  }
 }
 return result;
}
```

Listing 4.10: Algorithm for implementing `equals` on ordered finite tag set

Index Switching Operators

- If *iExpressionType* == 0, *getPrevious()* or *getNext()* returns the next element of the given tag parameter in the enumeration.

- If *iExpressionType* == 1, *getPrevious()* or *getNext()* returns parameter-1 or parameter+1 within the range of tag set. Note that for now we only consider the sub set of integers for this expression.
- If *iExpressionType* == 2, *getPrevious()* or *getNext()* returns parameter-step or parameter+step within the range of the tag set. Note that for now we only consider the sub set of integers for this expression.

The algorithms are listed in Listing 4.11 and Listing 4.12.

```
GIPSYType getPrevious(GIPSYType poTag)
{
  GIPSYType result = null;
  if(isInTagSet(poTag)){
   switch(this.iExpressionType){
    case : 0{
      if(!poTag.equals(this.tagAt(0))) //the parameter is not the first
          element
        result = this.tagAt(getIndex(poTag)-1);
      break;
    }
    case : 1{
      if(poTag != lower)
        result = poTag-1;
      break;
    }
    case : 2{
      if(poTag != lower)
        result = poTag-step;
      break;
    }
   }
  }
  return result;
}
```

Listing 4.11: Algorithm for implementing `getPrevious` on ordered finite tag set

4.4.2.2 Ordered Infinite Tag Set

Although we call this type of set 'infinite', in the actual implementation, there should be a way to handle this 'infinity' to make it 'infinite' allowed by the available storage resources. For now we only consider `Integer` as the type for a tag value, thus the infinity is actually represented by either `Integer.MIN_VALUE` of Java

```
GIPSYType getNext(GIPSYType poTag)
{
  GIPSYType result = null;
  if(isInTagSet(poTag)){
   switch(this.iExpressionType){
    case : 0{
      if(!poTag.equals(this.tagAt(this.size()-1))) //the parameter is
          not the last element
        result = this.tagAt(getIndex(poTag)+1);
      break;
    }
    case : 1{
      if(poTag != upper)
        result = poTag+1;
      break;
    }
    case : 2{
      if(poTag != upper)
        result = poTag+step;
      break;
    }
   }
  }
  return result;
}
```

Listing 4.12: Algorithm for implementing `getNext` on ordered finite tag set

for minus infinity or `Integer.MAX_VALUE` for plus infinity. There are five kinds of expressions for this type, we use *iExpressionType* == 3 to represent expression as `dimension d : ordered infinite {2 to INF+}`; *iExpressionType* == 4 to represent expression as `dimension d : ordered infinite {2 to INF+ step 2}`; *iExpressionType* == 5 to represent expression as `dimension d : ordered infinite {INF- to 100}`; *iExpressionType* == 6 to represent expression as `dimension d : ordered infinite {INF- to 100 step 2}` and *iExpressionType* == 7 to represent the entire stream of integers. The set membership, equality and tag set index switching methods are defined and implemented as the following:

Set Membership

- If *iExpressionType* == 3, *isInTagSet()* returns true if and only if the given parameter is greater than or equal to the lower boundary and less than or equal to `Integer.MAX_VALUE`.

- If *iExpressionType* == 4, *isInTagSet()* returns true if and only if the given parameter *para* is greater than or equal to the lower boundary l and less than or equal to `Integer.MAX_VALUE` and that $((para - l) \bmod p) = 0$, where p is the step specified.

- If *iExpressionType* == 5, *isInTagSet()* returns true if and only if the given parameter is less than or equal to the upper boundary and greater than or equal to `Integer.MIN_VALUE`.

- If *iExpressionType* == 6, *isInTagSet()* returns true if and only if the given parameter *para* is less than or equal to the upper boundary u and greater than or equal to `Integer.MIN_VALUE` and that $((u - para) \bmod p) = 0$.

- If *iExpressionType* == 7, *isInTagSet()* returns true if and only if the given parameter is greater than or equal to `Integer.MIN_VALUE` and less than or equal to `Integer.MAX_VALUE`.

See appendix for the algorithm.

Equality

- If *iExpressionType* == 3, *equals()* returns true if and only if the lower boundaries of two tag sets are equal..

- If *iExpressionType* == 4, *equals()* returns true if and only if the lower boundaries and the steps of two tag sets are equal respectively.

- If *iExpressionType* == 5, *equals()* returns true if and only if the upper boundaries of two tag sets are equal.

- If *iExpressionType* == 6, *equals()* returns true if and only if the upper boundaries and the steps of two tag sets are equal respectively.

- If *iExpressionType* == 7, *equals()* returns true if the two tag sets are of the same expression type.

See appendix for the algorithm.

Index Switching Operators

- If *iExpressionType* == 3 or 5, *getPrevious()* or *getNext()* returns parameter-1 or parameter+1 within the range of tag set.
- If *iExpressionType* == 4 or 6, *getPrevious()* or *getNext()* returns parameter-step or parameter+step within the range of tag set.
- If *iExpressionType* == 7, *getPrevious()* or *getNext()* returns parameter-1 or parameter+1 within the range of tag set.

See appendix for the algorithms.

4.4.2.3 Unordered Finite Tag Set

The set membership method returns true if and only if the given parameter is equal to one of the tag values inside the tag set. The equality method returns true if and only if all the enumerated elements in both tag sets are equal. Note that the equality of two unordered sets has nothing to do with the order that the tags are listed, thus this algorithm is different from the *equals* method in `Ordered Finite Tag Set`. Set index switching operators are not applicable on this type.

```
{
  boolean isInTagSet(GIPSYType poTag)
{
  for(int i = 0; i < this.size(); i++)
  {
    if(poTag == this.tagAt(i))
    return true;
  }
    return false;
}
}
```

Listing 4.13: Algorithm for implementing `isInTagSet` on unordered finite tag set

4.4.2.4 Unordered Infinite Tag Set

Please refer to Chapter 6 for more details.

```
boolean equals(Object otherObject)
{
  if(!otherObject instanceof UnorderedFiniteTagSet)
    return false;
  else{
    if((UnorderedFiniteTagSet)otherObject.expressionType!=this.
        expressionType)
      return false;
    else{
      boolean flag = false;
      Vector enum1 = this.getEnumeratedElements()
      Vector enum2 = (UnorderedFiniteTagSet)otherObject.
          getEnumeratedElements();
      for(int i = 0; i < enum1.size(); i++)
      {
        flag = false;
        for(int j = 0; j < enum2.size(); j++)
        {
          if(enum1.elementAt(i).equals(enum2.elementAt(j)))
          {
            flag = true;
            break;
          }
        }
        if(flag = false)
        break;
      }
      return flag;
    }
  }
}
```

Listing 4.14: Algorithm for implementing `equals` on unordered finite tag set

4.5 Embedding Context and Tag Set Classes into the GIPSY

In order to keep the components of type system in a concise manner, the context class structure has two major changes while being embedded into GIPSY Type System. First change is replacing Micro Context class with Dimension class. As described earlier, a context is actually a collection of <*dimension* : *tag*> pairs, thus in order to build context classes, we also present `Dimension` class. The `Dimension` class has an object of type `GIPSYIdentifier` called `oDimensionName` to specify the dimension object's name and `oTagSet` to keep the information of the tag set attached to it. It also has a reference `oCurrentTag`, which is set to the current tag value inside the

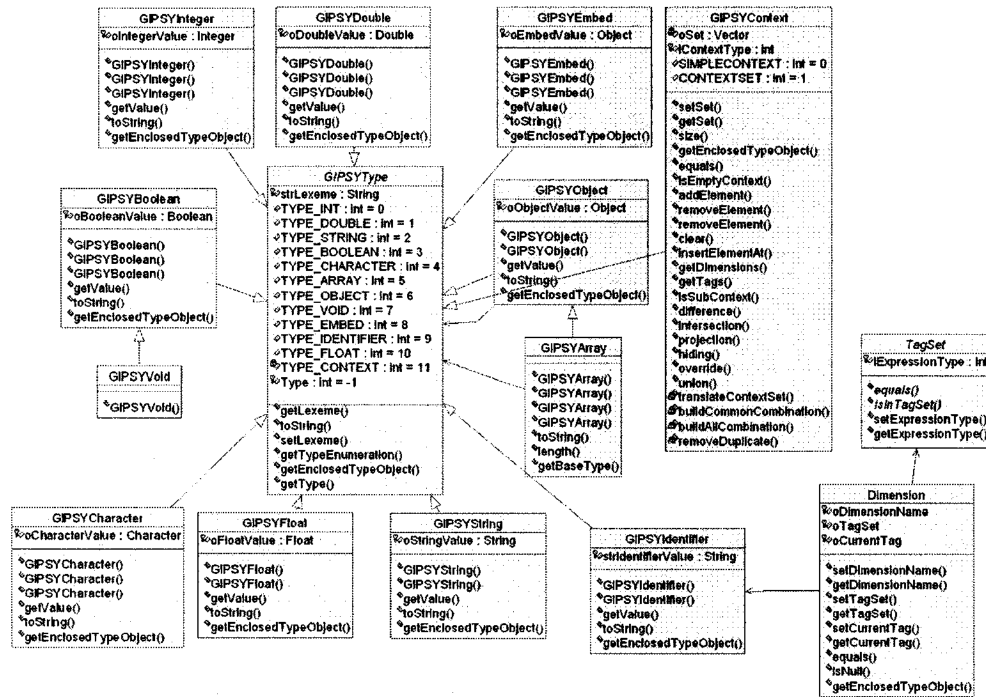

Figure 12: Embed context class into GIPSY

dimension, by adding this field, the notion of micro context can be expressed and consequently, replaced by `Dimension`.

The second change is merging all the context types including simple context and context set into a new class called GIPSYContext. We modified the representation of context classes as shown in Figure 12. Now we keep the context type information in the attribute of `iContextType`. By this means, we still keep the notion of composite design pattern cohesively expressed inside the `GIPSYContext` class. We put `GIPSYContext` under `gipsy.lang` package, and group all the tag set classes and Dimension class into `gipsy.lang.context` package.

4.6 Summary

In this Chapter, we present the implementation of Lucx's context and context calculus. First, we provide the construction process of Lucx compiler; then we present the design and implementation of context type and context calculus in the GIPSY.

Finally, the implementation of tag set types and a set of methods for the types are specified.

Chapter 5

Testing

In order to prove the correctness of the implementation, we present the testing mechanism in this Chapter. There's already an existing testing infrastructure in the GIPSY, we follow its design to perform test on Lucx's parser, semantic analyzer and context type.

5.1 Testing Infrastructure for Lucx

There are three main modules in GIPSY: the General Intensional Programming Compiler (GIPC); the General Eduction Engine (GEE), and the Run-time Iteractive Programming Environment (RIPE). Accordingly, we now have 3 main test packages which are `tests.GIPC`, `tests.GEE` and `tests.RIPE`. We follow this infrastructure to put testing for Lucx compiler under the `tests.GIPC.intensional.SIPL.Lucx` package. We also introduce another unit testing package `tests.junit`, where all the tag set classes and context class are tested. Figure 13 shows the package overview of testing for Lucx.

5.2 Testing for Lucx Parser

As described in Chapter 3, Lucx has extended the syntax of GIPL by introducing the notation of <`dimension` : `tag`> pair in expressions to allow context to be manipulated as first class value. A set of context calculus operators and the concept of tag set are also added to the original GIPL syntax. Here, the tests mainly focus on testing these

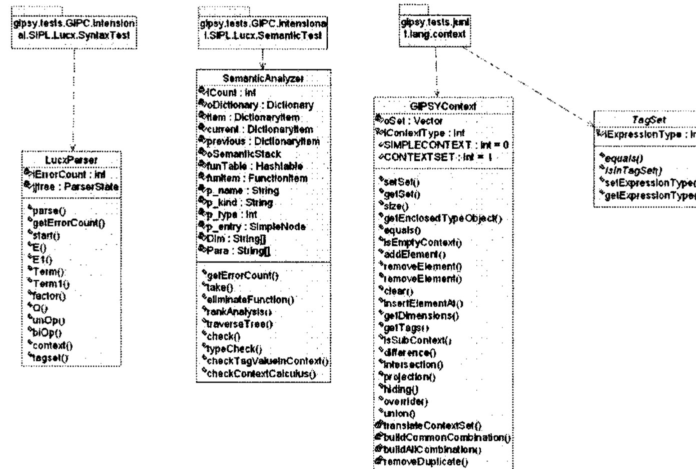

Figure 13: Testing Infrastructure for Lucx

new features in the syntax. We provide Lucx source code in .ipl files as the input of the parser. Design and statistics of test cases are given below:

Context As First Class Value The goal for this kind of testing is to see if context can be used as first class value in Lucx expressions. We created six test cases to test the following aspects.

- `Simple Context` in the first expression and back-compatibility with Lucid operators:

  In Listing 5.1, the Simple Context expression $[\ d : 2\ ]$ is used in the first expression of Lucx program, and in order to show Lucx's back-compatibility with Lucid operators, there are Lucid operators `fby` and #.

- `Context Set` in the first expression and back-compatibility with Lucid operators:

  It's similar with the first test case, the only difference is that the context type is Context Set.

- `Simple Context` in where clause; complex expression in context expression and compatibility with if clause.

```
/* This is to test Simple Context in the first expression
   and back-compatibility with Lucid operators
*/
n @ [d : 2]
  where
    dimension d;
    n = 42 fby #d + 1;
  end
```

Listing 5.1: Simple Context As First Class Value Sample1

In Listing 5.2, the Simple Context expression is in the where clause, and its tag expression is another # expression. This test case also has an if clause, to check the compatibility with old versions of Lucid.

```
/* This is to test Simple Context expression in where clause
   complex expression in Context expression
   and compatibility with if clause
*/
n
 where
   dimension d;
   n = if #d <=0 then 22
   else (n+1) @ [d : (#d-1)]
   fi;
 end
```

Listing 5.2: Simple Context As First Class Value Sample2

- `Context Set` in where clause; complex expression in context expression and compatibility with if clause.

  It's similar with the first test case, the only difference is that the context type is Context Set.

- `Simple Context` with more than one micro contexts and is used as a function parameter

- `Context Set` with more than one Micro Context and is used as a function parameter

  It's similar with the first test case, the only difference is that the context type is Context Set.

```
/* This is to test Simple Context with more than one Micro Context
   used as function parameter
*/
I
 where
   dimension d;
   dimension e;
   I=getI [d, e]([d : 1, e : 2]);
   getI [d , e](cxt)
     where
       x = x @ cxt;
     end;
 end
```

Listing 5.3: Simple Context As First Class Value Sample3

Context Calculus In the following paragraph, context calculus expressions are tested. We have two test cases for each operator, one is on Simple Context and the other is on Context Set. And we also have some test cases over complex context calculus expressions. Here we don't list all the test cases for each operator, but give some typical examples.

Listing 5.4 shows the test on complex context calculus operator expression. Listing 5.5 shows the test on context calculus expression being used as a parameter of a function call.

```
n @ ([d : 2, e : 3, f : 3] intersection [d : 2] union [f :4, d : 5])
   where
     dimension d;
     dimension e;
     dimension f;
     n = 34 fby n+1;
   end
```

Listing 5.4: Context Calculus Test1

Tag Set Types Testing in this part is mainly to show that each type of tag set declaration can be parsed and an appropriate AST can be generated. Here we only present one example, because the rest is similar to this one.

```
/* This is to test context calculus expression
   used as function parameter
*/
I
 where
   dimension d;
   dimension e;
   I=getI [d, e]([d : 1, e : 2] intersection [e : 2]);
   getI [d , e](cxt)
     where
       x = x @ cxt;
     end;
 end
```

Listing 5.5: Context Calculus Test2

```
n @ [d : red]
  where
    dimension d : unordered finite {red , green , blue};
  end
```

Listing 5.6: Tag Set Test

5.3 Testing for Semantic Analyzer

As stated in the implementation chapter, two main testing points are the validity of context calculus operators and tag values inside a context. And note that because Lucx does not follow the traditional translation path, but extend the existing Semantic Analyzer by adding some new functions to manipulate Lucx nodes in the AST, its backward compatibility should also be checked to see if this modification has any affect on the original Semantic Analyzer. The testing points are listed in the following paragraph.

- Backward Compatibility:

  We borrowed test cases presented by Aihua Wu, who created the Semantic Analyzer class to support GIPL. See details in [48].

- Dimension Variable Definition:

  The testing point is that the dimension variable used inside a context expression is either not defined or redefined. When it's not defined, it's either not declared in the where clause, the variable with the same name is not declared as a

dimension variable or the declaration of the dimension variable is in a different scope. This is shown in Listing 5.7, Listing 5.8 and Listing 5.9.

There are two cases of dimension variable redefined: First is the dimension variable is defined in the same scope for more than once. The other possibility is that the variable is defined again in a different scope. The first one would yield a semantic error, because there is a name conflict in the dictionary. However the second one would not because different scopes has their own name space in the dictionary. They are shown in Listing 5.10 and Listing 5.11.

```
n @ [d:2]
 where
  n=42;
 end
```

Listing 5.7: Dimension Variable Not Defined Sample1

```
n @ [d:2]
 where
   d=12; //Although d is defined, it is not a dimension variable.
   n=42;
 end
```

Listing 5.8: Dimension Variable Not Defined Sample2

```
n @ [d:2]
 where
   n=42;
   where
     dimension d;
   end;
 end
```

Listing 5.9: Dimension Variable Defined In A Different Scope

```
((#d)-1)+n
   where
     dimension d;
     dimension d : unordered finite nonperiodic {red, green};
     n=12;
   end
```

Listing 5.10: Dimension Variable Redefined

```
((#d)-1)+n
   where
     dimension d;
     n=((#d)-1)
       where
         dimension d;
       end;
   end
```

Listing 5.11: Dimension Variable With Same Name In Different Scope

- Context Calculus Type Checking

  As described in the implementation Chapter, type checking here is to see if two operands of a context calculus operator are of the same context type, or if the operand is valid, eg. The second parameter of `projection` must be a dimension set. We have three test cases for each operator: one for the operands which are both Simple Context, second for both Context Set and the last for an error type mismatch scenario. Note that, as the reason stated in the implementation Chapter, here we only check the operands whose expressions can be determined at compile time. We present one example of the test case, the rest is similar to this one.

```
n @ ([a:1] difference {[a:2]})
     where
        dimension a;
        n=1;
     end
```

Listing 5.12: Context Calculus Operator With Operands Type Mismatch

- Tag Validity

  Semantic analysis here mainly checks if the tag value in a context expression is inside the tag set declared in the where clause. We have two test cases for each tag set type, one is the normal case, which means that all the tag values inside a context expression are inside the declared tag set; the other is the reverse. Here we provide one example of the test case.

```
n @ [a:1, b : red]
    where
        dimension a;
        dimension b;
        n=1;
    end
```

Listing 5.13: Tag Value Not Inside Tag Set

5.4 Unit Testing for Context Class and Tag Set Classes

Unit testing is the key approach to test-driven development, which allows you to incrementally implement your application with less possible side-effects. There are many implementation dependencies among context calculus operators, meaning that the implementation of one operator might involve in that of another. Thus here we adopt the unit testing strategy and use the facilities provided by Junit. Figure 14 shows the structure of test classes over GIPSYContext class and Tag Set classes.

5.4.1 Test Over GIPSYContext Class

The test is mainly focus on context calculus operators. The general strategy is to instantiate two possible operands for an operator, which mostly are two objects of GIPSYContext and also the expected result, which is also an object of GIPSYContext. Then we invoke `assertEquals(Object expected, Object actual)` method provided by Junit framework to see if the result after computation of a context calculus operator is the same as the expected result. In the following paragraph, we provide some examples of *union* operator according to the test routines listed below. Note that we have test cases over both Simple Context and Context Set. Here we only present examples on Simple Context to avoid redundancy. Here we use Φ to represent empty context.

- Empty Context and Empty Context

  The expected and actual results of *union* two empty Simple Context are both an empty Simple Context.

- Empty Context and Non-empty Context

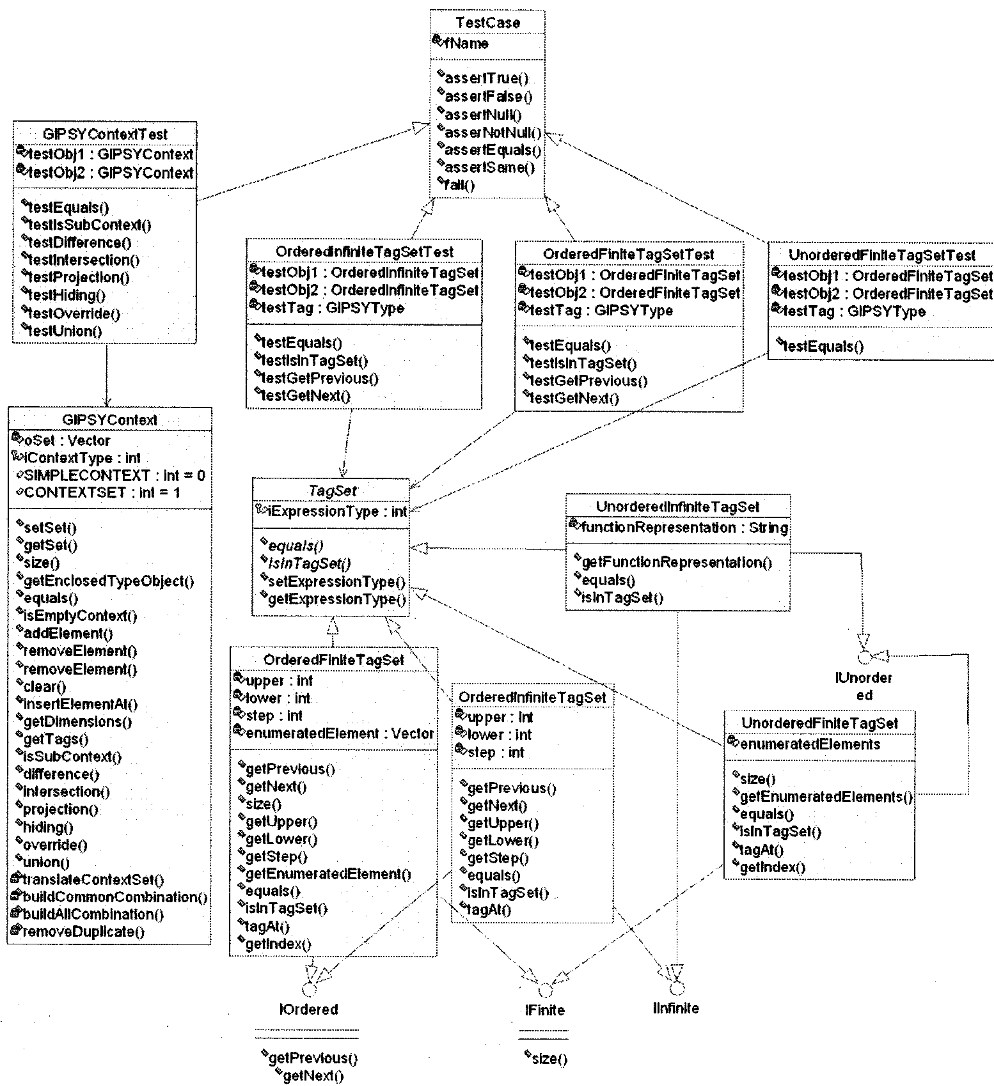

Figure 14: Unit Testing for Tag Set and GIPSYContext

For example, we instantiate an object of Simple Context, whose size is zero to represent an empty Simple Context; and also a non-empty Simple Context object of [d:1,e:2], the expected and actual results are both [d:1,e:2]. Note that for some context calculus operators, the order of presenting two parameters might generate different results. For example, we have the test case Φ *difference* [d:1,e:2] = Φ, however, [d:1,e:2] *difference* Φ = [d:1,e:2].

- A Context and Itself For example, we have two Simple Context object which are both [d:1,e:2], then both the expected result and the actual result are all [d:1,e:2].

- A Context and Its Proper Subset

  Under this category, one test object is the proper subset of the other. For example, we have a test case: [d:1,e:2] *union* [d:1] = [d:1,e:2].

- Two Contexts With Some Common Dimension Elements But Not All

  As shown in Chapter 3, `union` operator over Simple Context has some uniqueness here. The result is no longer a Simple Context but a Context Set. For example, we have the test case $[d:1,\ e:2,\ f:3]$ *union* $[e:4,\ f:5,\ g:6] = \{[d:1,\ e:2,\ f:3,\ g:6],\ [d:1,\ e:2,\ f:5,\ g:6],\ [d:1,\ e:4,\ f:3,\ g:6],\ [d:1,\ e:4,\ f:5,\ g:6]\}$

- Two Contexts With All The Common Dimension Elements

  The test case here is similar to the previous one.

- Two Contexts With No Common Dimension Elements

  For example, we have test case: $[d:1,\ e:2]$ *union* $[f:3,\ g:4] = [d:1,\ e:2,\ f:3,\ g:4]$

5.4.2 Test Over Tag Set Classes

Test here is mainly focus on testing tag validity checking method `isInTagSet` and proper operators on tag set such as `getPrevious` and `getNext`. It is achieved by instantiating a tag value object of GIPSYType, and also an object of different tag set types. Then we invoke `assertEquals(Object expected, Object actual)` method

provided by Junit framework to see if the result after computation of the above operators is the same as the expected result.

5.4.2.1 Test Over *isInTagSet*

We have both the true and false scenarios of this method for each type of tag set.

5.4.2.2 Test Over *getPrevious()* and *getNext()*

We have three types of test cases here over each tag set type:

- The parameter tag is inside the tag set and it's not the boundary
- The parameter tag is the boundary
- The parameter tag is not inside the tag set

5.5 Summary

In this Chapter, we present test strategies and test cases for tests over Lucx compiler, context calculus operators and tag sets. There are some limitations in testing these components, which will be discussed in Chapter 6.

Chapter 6

Conclusion

Along the path of evolution, Lucid has adopted more and more innovative features to adapt to gain in generality and adapt to different application domains. Lucx, as one of the evolution steps, has contributed to the family in introducing context as first class value and a set of context calculus operators. From this regard, Lucx is innovative because of the uniqueness and importance of the notion of context in intensional programming. Here we provide a summary of implementing Lucx's context and context calculus into the GIPSY.

6.1 Extending the GIPC

GIPSY is an intensional programming language framework within which the Lucid family of intensional programming languages is compiled and executed. Thus, the first step of implementing Lucx is to provide its compiler. Following the design of GIPSY framework, all the compilers should be integrated to the GIPC layer. Lucx follows the same route.

The construction of Lucx compiler consists of the following steps: First, derive the concrete syntax of Lucx according to [44], which extends GIPL by introducing *context calculus expressions.* The grammar is then modified to remove all the left recursions and translated into the specification that can be accepted by JavaCC as input to generate the parser for Lucx. And finally the semantic checking for Lucx, including checking for *context calculus* and *tag value validity* were done by extending the existing Semantic Analyzer. Syntactically and semantically, Lucx is an extension

to the GIPL; therefore, it owns those new features and also inherits old features of GIPL. So we take both scenarios into consideration while designing the test strategies. Syntactically, the backward compatibility was tested by enclosing the existing GIPL expressions and operators into Lucx programs; new features such as context as first class value in an expression, context calculus operators and tag set types are all fully tested. Both backward compatibility and new features in the syntactic tests accomplished positive results with proper ASTs generated. Semantically, backward compatibility was examined by referring to the existing well-defined test cases; and the new type checking including type checking for context and its calculus operators and type checking for valid tag value were also properly tested. The results of these two checking points are also positive in that the corresponding error messages are correctly displayed and the output objects of the GEER are properly filled with all the attributes.

6.2 GIPSY Type System

Although all the SIPLs have their unique syntax and other special features, there is a set of general types shared among all of them. Even though in most of the cases, those types are not explicitly declared in the source code at syntactic level, at runtime, the evaluation of expressions would resort to the actual type of variables or literals. The goal of constructing a Type System is to include all the general types among the SIPLs and thus provide the facility for their runtime evaluation. As the notion of *context* is a general property of all the intensional programming languages, a data type representing it has been embedded into the type system. So the current type system has a new variant of `GIPSYContext`, and also a set of tag set classes and dimension class, which are the key attributes when defining a context.

6.3 Context Calculus

Similar to a set of arithmetic operators performed on numeric values, context calculus operators are a set of operators performed on context values. They are implemented inside the `GIPSYContext` class and can be called on instantiated context objects. These operators can be performed on both simple context and context set to generate

new computed contexts. Because simple context is a collection of micro contexts and context set is a collection of simple contexts, the essential notion under the implementation of context calculus operators is the set theory. The correctness of this implementation is tested using the unit testing strategy. We performed the tests by instantiating two context objects with many interesting and exceptional scenarios, such as empty contexts, etc. Then we specified the expected result and called the context calculus operators via the instantiated context objects to see if the result matched what we expected. When we implemented context calculus in the GIPSY, we also introduced tag set types and a set of possible operators on the tag set. Those operators were also tested following the unit testing strategy. We first instantiated tag set objects, then we specified certain tag values to include interesting or eceptional scenarios. For all the test scenarios discussed in Chapter 5, we obtained positive results.

6.4 Discussions and Limitations

In this section, we discuss our approach and results and highlight their limitations. Lucx was defined theoretically. From a certain number of aspects, GIPSY was not compatible with the theoretical vision of Lucx as persented in Wan's thesis. This section presents some of the interesting discussions that we had when incorporating Lucx into the GIPSY.

6.4.1 Context Set: Motivation and Limitations

Motivation As defined in the GIPL semantics, Lucid works fine with simple context, in other words, a point ($\mathcal{P}$) in the context space at both syntactical and semantical level. And the result of evaluating an expression under certain context yields a *single value*. However, there might be certain needs to evaluate an expression at a *set of points* in the context space in order to increase computation granularity and save communication time in case of distributed evaluation.

Limitation Although the notion of *context set* would permit to increase granularity by grouping related demands/values, the evaluation of an expression at a *context set* yields a *set of values*, which cannot be manipulated in our current implementation.

Thus to sum up, there's definitely a need to have context set in Lucid. The only thing needed to be done is to embed operational semantics of context set into the current execution engine.

6.4.2 The Concept of Box

In [44], there is also a notion of *Box*. a Box is a set of contexts, all of which have the same dimension set and the tags corresponding to the dimensions in each context satisfy a given constraint. It is a special kind of context set. We haven't implemented box into the GIPSY because of the following reasons: First, the semantics of box has not been proved. A simple context is actually a point in the context space, and context set is a set of discrete points. However, according to the definition, a box can be a continuous region. Semantically, such continuity has not been formally proved. Second, the context elements inside a box are determined by the user-defined constraint. Thus we have to find a mechanism to translate Box notation into context set. However, user-defined constraints could be in any form. For example, it could be logical expressions, mathematic functions and so many more. So what kinds of expression should be allowed and then what kinds of translation routines should be provided are still open issues.

6.4.3 The Unordered Infinite Tag Set

Since such a kind of tag set is infinite, we cannot list all the tag values, and also it is unordered, there's no rule to show what the trend of possible tags is. In order to define such a tag set, the user has to provide a function to generate all the tag values. This function could be an intensional function or imperative function. If it is an intensional function, then theoretically it could be flattened by the Semantic Analyzer provided the function is not recursive, but what if the function has to be recursive? And if it is an imperative function, then Lucx somehow becomes hybrid Lucid. Its syntax and semantics all have to be modified to include imperative features. However, such modifications are providing nice new features or adding more complexity is still under discussion.

6.4.4 Test On Context Calculus

Although we provided elaborated test cases for context calculus operators, it is still a simulation of the actual evaluation. The actual execution of context calculus operators should be performed in the execution engine at runtime, however, since we do not have a fully working engine currently, here we can only assume that after instantiation of context object, the implementation of context calculus operators satisfies our expectation.

Chapter 7

Future Work

7.1 The New Generation Of Execution Engine

The final stage of executing a Lucx program is at the runtime execution engine side. The notion of context is represented by a String value in the current engine, because there was no explicit context type in the system. Ongoing efforts have been making to adapt the engine to fully context-awareness. Now we already replace all the String representation of context with the newly added `GIPSYContext` type. In the future, the engine will also be able to evaluate context calculus operators and also perform runtime semantic checking over contexts.

7.2 Formal Verification of Context Set and Box Theory

The proof of semantic rules for Context Set and Box has to be conducted formally, a way of manipulating a set of values returned by an expression should be figured out.

7.3 More Possibilities Of Tag Set

We now have Ordered, Unordered, Finite and Infinite keywords to define tag sets. There is great possibility that when we expand our domain of application, more keywords would be included to describe the new properties of tag set.

Bibliography

[1] In *Proceedings of the 4th annual IEEE international conference on Pervasive Computing and Communications Workshops.* IEEE Computer Society, 2006.

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

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

[4] Edward A. Ashcroft and William W. Wadge. Lucid, a nonprocedural language with iteration. In *Communication of the ACM*, pages 519–526. ACM, July 1977.

[5] Barendregt and Hendrik Pieter. North-Holland Pub. Co. ; sole distributors for the U.S.A. and Canada Elsevier North-Holland, New York, 1984.

[6] Rudolf Carnap. *Meaning and Necessity: a Study in Semantics and Modal Logic.* University of Chicago Press, Chicago, USA, 1947.

[7] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming: an overview of ContextL. In *Proceedings of the 2005 symposium on Dynamic languages*, pages 1–8. ACM, October 2005.

[8] Anind K. Dey and Gregory D. Abowd. Towards a Better Understanding of Context and Context-Awareness. *Graphics, Visualization and Usability Center and College of Computing*, 1999.

[9] Yi Min Ding. Bi-directional Translation Between Data-Flow Graphs and Lucid Programs in the GIPSY Environment. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2004.

[10] Jordi Docter, Carlo Alberto Licciardi, and Marco Marchetti. The Telecom Industry and Context Awareness. In *Proceedings of the International Conference on the Management of Mobile Business*. IEEE Computer Society, 2007.

[11] D. Dowty, R. Wall, and S. Peters. *Introduction to Montague Semantics*. D. Reidel, Dordrecht, The Netherlands, 1981.

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

[13] A. A. Faustini and R. Jagannathan. Indexical Lucid. In *Proceedings of the Fourth International Symposium on Languages for Intensional Programming*, Menlo Park, California, 1991.

[14] Abraham A. Fraenkel. *Abstract set theory*. New York, Amsterdam : North-Holland Pub. Co., fourth revised edition, 1976.

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

[16] D. Gallin. *Intensional and Higher-Order Modal Logic: With Applications to Montague Semantics*. North-Holland, Amsterdam, The Netherlands, 1975.

[17] R. Gold and C. Mascolo. Use of Context-Awareness in Mobile Peer-to-Peer Networks. In *Proceedings of the 8th IEEE Workshop on Future Trends of Distributed Computing Systems*. IEEE Computer Society, 2001.

[18] Masatomo Hashimoto and Atsushi Ohori. A typed context calculus. In *Theoretical Computer Science*, volume 266, pages 249–272. ACM, September 2001.

[19] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. *Context-oriented Programming*. Journal of Object Technology, USA, 2008.

[20] Allen Holub. *Holub on patterns : learning design patterns by looking at code*. Berkeley : Apress, 2004.

[21] Raganswamy Jagannathan and Chris Dodd. GLU programmer's guide. Technical report, SRI International, Menlo Park, California, 1996.

[22] Raganswamy Jagannathan, Chris Dodd, and Iskender Agi. GLU: A high-level system for granular data-parallel programming. In *Concurrency: Practice and Experience*, volume 1, pages 63–83, 1997.

[23] Roger Keays and Andry Rakotonirainy. Context-Oriented Programming. In *Proceedings of the 3rd ACM international workshop on Data engineering for wireless and mobile access*, pages 1–8. ACM, September 2003.

[24] S. Kripke. *Naming and Necessity.* Harvard University Press, Cambridge, MA, 1980.

[25] Saul A. Kripke. *A Completeness Theorem in Modal Logic.* 1959.

[26] Saul A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, (16):83–94, 1963.

[27] Seymour Lipschutz. *Schaum's Outlines of Theory and Problems of Set Theory and Related Topics.* New York : McGraw-Hill, second edition, 1998.

[28] Kenneth C. Louden. *Compiler Construction: Principles and Practice.* PWS Publishing Company, 1997. ISBN 0-564-93972-4.

[29] Steven John Metsker. *Design patterns in Java.* Upper Saddle River, NJ : Addison-Wesley, 2006.

[30] Serguei Mokhov and Joey Paquet. General Imperative Compiler Framework within the GIPSY. In *Proceedings of PLC2005, Las Vegas, Nevada, USA*, pages 36–42. CSREA Press, June 2005.

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

[32] Robert C. Moore. Removing Left Recursion from Context-Free Grammars. In *1st Meeting of the North American Chapter of the Association for Computational Linguistics.* Association for Computational Linguistics, April 2000.

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

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

[35] Chun Lei Ren. General Intensional Programming Compiler (GIPC) in the GIPSY. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2002.

[36] The GIPSY Research and Development Group. *The GIPSYwiki: Online GIPSY collaboration platform.* Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2005. `http://newton.cs.concordia.ca/~gipsy/gipsywiki`.

[37] David Sands. Computing with contexts: A simple approach. In *Second Workshop on Higher-Order Operational Techniques in Semantics (HOOTS II).* Electronic Notes in Theoretical Computer Science, 1998.

[38] William C. Wake Steven John Metsker. *Design patterns in Java.* Upper Saddle River, NJ : Addison-Wesley, 2006.

[39] Lei Tao. Warehouse and Garbage Collection in the GIPSY Environment. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2004.

[40] Emil Vassev and Joey Paquet. A Generic Framework for Migrating Demands in the GIPSY's Demand-Driven Execution Engine. In *Proceedings of PLC2005, Las Vegas, Nevada, USA*, pages 29–35. CSREA Press, June 2005.

[41] Emil I. Vassev. General Architecture for Demand Migration in the GIPSY Demand-Driven Execution Engine. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, October 2005.

[42] Sreenivasa Viswanadha and Contributors. *Java Compiler Compiler (JavaCC) - The Java Parser Generator.* java.net, 2001-2005. `https://javacc.dev.java.net/`.

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

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

[45] Kaiyu Wan, Vasu Alagar, and Joey Paquet. Lucx: Lucid Enriched with Context. In *Proceedings of PLC2005, Las Vegas, Nevada, USA*, pages 48–14. CSREA Press, June 2005.

[46] Wikipedia. *Context-aware pervasive systems — Wikipedia, The Free Encyclopedia.* 2007. `http://en.wikipedia.org/wiki/Context-aware_pervasive_systems`.

[47] Wikipedia. *Lambda calculus — Wikipedia, The Free Encyclopedia.* 2007. `http://en.wikipedia.org/wiki/Lambda_calculus`.

[48] Ai Hua Wu. Semantic Checking and Translation in the GIPSY. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2002.

[49] Ai Hua Wu, Joey Paquet, and Peter Grogono. Design of a Compiler Framework in the GIPSY System. In *Proceedings 15th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2003)*, volume 1, pages 320–328. International Association of Science and Technology for Development, November 2003.

Appendix A

Source Listing For Context Calculus Operators

```
boolean isSubContext(SimpleContext pC)
{
  if(this.size() == 0)
    return true;
  if(pC.size() < this.size()) //It includes the case where pC.size() ==
     0
    return false;
  else{
     boolean flag;
     for(int i = 0; i < this.size(); i++){
       flag=false;
       for(int j = 0; j < pC.size(); j++){
         if(pC.micro_context(j) == this.micro_context(i)){
           flag=true;
           break;
         }
       }
        if(flag == false)
          break;
      }
      return flag;
  }
}
```

Listing A.1: Algorithm for implementing *isSubContext* on simple context

```
boolean isSubContext(ContextSet pS)
{
  if(this.size() == 0)
    return true;
  if(pS.size() < this.size()) //It includes the case where pS.size() ==
      0
    return false;
  else{
     boolean flag;
     for(int i = 0; i < this.size(); i++){
       flag=false;
       for(int j = 0; j < pS.size(); j++){
         if(pS.micro_context(j) == this.micro_context(i)){
           flag=true;
           break;
         }
       }
        if(flag == false)
          break;
      }
      return flag;
  }
}
```

Listing A.2: Algorithm for implementing *isSubContext* on context set

```
SimpleContext difference(SimpleContext pC){
  SimpleContext result=this.clone();
    if(this.size() == 0)
     return result;
    if(pC.size() == 0)
     return result; //return the result immediately without going into
         the loop
    for(int i = 0; i < this.size(); i++){
     for(int j = 0; j < pC.size(); j++){
      if(pC.micro_context(j)==this.micro_context(i)){
       result.remove(this.micro_context(i));
      }
     }
    }
  return result;
}
```

Listing A.3: Algorithm for implementing *difference* on simple context

```
ContextSet difference(ContextSet pS){
  ContextSet result=new ContextSet();
  for(int i = 0; i < this.size(); i++){
    for(int j = 0; j < pS.size(); j++){
      SimpleContext tempResult=difference(this.simple_context(i), pS.
          simple_context(j));
      if(tempResult.size() != 0) //if all the tempResult.size()==0, the
          result is an empty context
        result.add(tempResult);
    }
  }
  return result;
}
```

Listing A.4: Algorithm for implementing *difference* on context set

```
SimpleContext intersection(SimpleContext pC){
  return this.difference(this.difference(pC));
}
```

Listing A.5: Algorithm for implementing *intersection* on simple context

```
ContextSet intersection(ContextSet pS){
  ContextSet result=new ContextSet();
  for(int i = 0; i < this.size(); i++){
    for(int j = 0; j < pS.size(); j++){
      SimpleContext tempResult=intersection(this.simple_context(i), pS.
          simple_context(j));
      if(tempResult.size() != 0) //if all the tempResult.size()==0, the
          result is an empty context
        result.add(tempResult);
    }
  }
  return result;
}
```

Listing A.6: Algorithm for implementing *intersection* on context set

```
SimpleContext projection(Vector dimSet){
  SimpleContext result = new SimpleContext;
  for(int i = 0; i < dimSet.size(); i++){
    for(int j = 0; j < this.size(); j++){
      if(this.micro_context(j).dimension == dimSet.dimension(i))
        result.add(this.micro_context(j));
    }
  }
  return result;
}
```

Listing A.7: Algorithm for implementing *projection* on simple context

```
ContextSet projection(Vector dimSet){
  ContextSet result=new ContextSet();
  for(int i = 0; i < this.size(); i++){
      SimpleContext tempResult=projection(this.simple_context(i), dimSet
          );
      if(tempResult.size() != 0) //if all the tempResult.size()==0, the
          result is an empty context
        result.add(tempResult);

  }
  return result;
}
```

Listing A.8: Algorithm for implementing *projection* on context set

```
SimpleContext hiding(SimpleContext c, DimensionSet dimSet){
  return(difference(c, projection(c, dimSet)));
}
```

Listing A.9: Algorithm for implementing *hiding* on simple context

```
ContextSet projection(Vector dimSet){
  ContextSet result=new ContextSet();
  for(int i = 0; i < this.size(); i++){
      SimpleContext tempResult=hiding(this.simple_context(i), dimSet);
      if(tempResult.size() != 0) //if all the tempResult.size()==0, the
          result is an empty context
        result.add(tempResult);

  }
  return result;
}
```

Listing A.10: Algorithm for implementing *hiding* on context set

```
SimpleContext override(SimpleContext pC){
  SimpleContext result = new SimpleContext();
  boolean flag=false;
  SimpleContext uniqueMCInC2=pC.clone();//micro contexts whose
     dimensions are unique in pC
  for(int i = 0; i < this.size(); i++){
    for(int j = 0; j < pC.size(); j++){
      if(this.micro_context(i).dimension == pC.micro_context(j).
          dimension)
      {
        flag=true;
        result.add(pC.micro_context(j));
        uniqueMCInC2.remove(pC.micro_context(j));
      }
    }
    if(flag == false)
      result.add(this.micro_context(i)); //Add the micro contexts whose
          dimensions are unique in this
    flag=false;
  }
  for(int k = 0; k < uniqueMCInC2.size(); k++)
  {
    result.add(uniqueMCInC2.micro_context(k));
  }
  return result;
}
```

Listing A.11: Algorithm for implementing *override* on simple context

```
ContextSet override(ContextSet pS){
  ContextSet result=new ContextSet();
  for(int i = 0; i < this.size(); i++){
    for(int j = 0; j < pS.size(); j++){
      SimpleContext tempResult=override(this.simple_context(i), pS.
          simple_context(j));
      if(tempResult.size() != 0) //if all the tempResult.size()==0, the
          result is an empty context
        result.add(tempResult);
    }
  }
  return result;
}
```

Listing A.12: Algorithm for implementing *override* on context set

```
Context union(SimpleContext pC){
  //Note that the return type is generic
  //Assume we have [f:1, e:1, d:2] union [e:2, d:1, t:4]
  SimpleContext result1;
  ContextSet result2;
  boolean isContextSet=false;
  for(int i = 0; i < this.size(); i++){
    for(int j = 0; j < pC.size(); j++){
      if(this.micro_context(i).dimension == pC.micro_context(j).
          dimension && this.micro_context(i) != pC.micro_context(j)){
        //[e:1, d:2] union [e:1] is a simple context: [e:1, d:2]
        isContextSet=true;
        break;
      }
    }
    if(isContextSet == true)
      break;
  }
  if(isContextSet == false){
    //No dimension in common, result is the combination c1 and c2.
    for(int i = 0; i < this.size(); i++){
      result1.add(this.micro_context(i));
    }
    for(int j = 0; j < pC.size(); j++){
      result1.add(pC.micro_context(j));
    }
    //remove duplicates e.g. [e:1, e:1, d:1] becomes [e:1, d:1]
    result1.removeDuplicateContext();
    return result1;
  }
  else{
    //There are common dimensions, the result is a non-simple context
    //A function is called to translate it into a context set
    result2=translateContextSet(this, pC);
    result2.removeDuplicateContext();
    return result2;
  }
}
```

Listing A.13: Algorithm for implementing *union* on simple context

```
ContextSet translateContextSet(SimpleContext pC1, SimpleContext pC2){
  //collection of micro contexts in pC1, pC2 having common dimensions
  Vector commonMC1, commonMC2;
  //collection of micro contexts in pC1, pC2 having no common dimension
  Vector uniqueMC1, uniqueMC2;
  //[e:1, e:2] or [d:1, d:2]
  Vector interMicroContext_i;
  //collection of interMicroContext_i: {[e:1, e:2],[d:1, d:2]}
  Vector interMicroContext;
  //{[e:1, d:1], [e:1, d:2], [e:2, d:1], [e:2, d:2]}
  ContextSet commonCombination;
  //{[f:1, e:1, d:1, t:4]...}
  ContextSet result;
  for(int i = 0; i < pC1.size(); i++){
    for(int j = 0; j < pC2.size(); j++){
      if(pC1.micro_context(i).dimension==pC2.micro_context(j).dimension)
          {
        commonMC1.add(pC1.micro_context(i));
        commonMC2.add(pC2.micro_context(j));
        interMicroContext_i.add(pC1.micro_context(i));
        interMicroContext_i.add(pC2.micro_context(j));
        interMicroContext.add(interMicroContext_i);
        break;
      }
    }
  }
  //build commonCombination {[e:1, d:1],[e:1, d:2]...}
  //pointer for combining all the possible micro contexts in
     interMicroContext
  int iniposition=0;
  //any simple context element of the context set commonCombination
  SimpleContext midReslt;
  buildCommonCombination(interMicroContext, commonCombination, midResult
     , iniposition );
  uniqueMC1=getUniqueMCs(c1, commonMC1);
  uniqueMC2=getUniqueMCs(c2, commonMC2);
  //build the final result {[f:1, e:1, d:1, t:4]...}
  result=buildAllCombination(uniqueMC1,uniqueMC2,commonCombination);
  return result;
}
```

Listing A.14: Algorithm for implementing helping method *translateContextSet* for *union* on simple context

```
void buildCommonCombination(Vector pInterMicroContext, ContextSet result
    , SimpleContext pMidResult, int pPosition){
  //Passing by reference is used, thus void the return type
  if(pPosition==pInterMicroContext.size()){
    //Finish one path of combination: eg.[e:1, d:2]
    result.add(pMidResult.clone());
    //Prepare to start another way of combination:
    //eg. if pMidResult=[e:1, d:1], then after this, pMidResult=[e:1]
    //waiting for the construction of pMidResult=[e:1, d:2]
    pMidResult.removeElement(pMidResult.lastElement());
    return;
  }
  else{
    //Constructing the possible combination
    Vector tempSC=pInterMicroContext.elementAt(position);
    position++;
    for(int i = 0; i < tempSC.size(); i++){
      MicroContext tempMC = tempSC.elementAt(i);
      pMidResult.add(tempMC);
      //Call buildCommonCombination to finish one path of combination
      //eg: if pMidResult=[e:1], the call would add [d:1],
      //thus making pMidResult=[e:1, d:1]
      buildCommonCombination(pInterMicroContext, result, pMidResult,
          pPosition);
    }
    if(pMidResult.size()!=0){
      //If no micro context left, the recursive call ends.
      //Preparing for the next combination:
      //eg. if pMidResult=[e:1], this operation clears it,
      //waiting for the next path of [e:2,...]
      pMidResult.removeElement(pMidResult.lastElement());
      return;
    }
  }
}
```

Listing A.15: Algorithm for implementing helping method *buildCommonCombination* for *union* on simple context

```
ContextSet buildAllCombination(Vector pUniqueMC1, Vector pUniqueMC2,
    ContextSet pCommonCombination){
  ContextSet result;
  for(int i = 0; i < pCommonCombination.size(); i++){
    SimpleContext tempSC=pCommonCombination.simple_context(i);
    for(int p = 0; p < pUniqueMC1.size(); p++){
      //eg. tempMC=[f:1]
      MicroContext tempMC=pUniqueMC1.elementAt(p);
      //insert [f:1] before [d:1, e:1] etc.
      tempSC.insertElementAt(tempMC, p);
    }
    for(int q = 0; q < pUniqueMC2.size(); q++){
      //eg. tempMC=[t:4]
      MicroContext tempMC=pUniqueMC2.elementAt(q);
      //append [t:4] after [d:1, e:1] etc.
      tempSC.add(tempMC);
    }
    result.add(tempSC);
  }
  return result;
}
```

Listing A.16: Algorithm for implementing helping method *buildAllCombination* for *union* on simple context

```
Vector getUniqueMCs(SimpleContext pSC, Vector pMicroContext_p){
  Vector microContext_l;
  boolean picked=false;
  for(int p = 0; p < pSC.size(); p++){
    MicroContext tempMC1=pSC.micro_context(p);
    for(int q = 0; q < pMicroContext_p.size(); q++){
      MicroContext tempMC2=pMicroContext_p.elementAt(q);
      if(tempMC1 == tempMC2){
        picked=true;
        break;
      }
    }
    if(picked==false){
      microContext_l.add(tempMC1);
    }
    else
      picked=false;
  }
  return microContext_l;
}
```

Listing A.17: Algorithm for implementing helping method *getUniqueMCs* for *union* on simple context

```
ContextSet union(ContextSet pS){
  DimensionSet interDimSet;
  for(int i = 0; i < this.size(); i++){
    for(int j = 0; j < pS.size(); j++){
      for(int k = 0; k < this.simple_context(i).size(); k++){
        for(int l = 0; l < pS.simple_context(j).size(); l++){
          if(pS.simple_context(j).micro_context(l)== this.simple_context
              (i).micro_context(k))
            interDimSet.add(this.simple_context(i).micro_context(k));
        }
      }
    }
  }
  ContextSet X1;
  for(int i = 0; i < this.size(); i++){
    for(int j = 0; j < pS.size(); j++){
      X1.add(union(this.simple_context(i), hiding(pS.simple_context(j),
          interDimSet)));
    }
  }
  ContextSet X2;
  for(int j = 0; j < pS.size(); j++){
    for(int i = 0; i < this.size(); i++){
      X2.add(union(pS.simple_context(j), hiding(this.simple_context(i),
          interDimSet)));
    }
  }
  for(int t = 0; t < X2.size(); t++){
    X1.add(X2.simple_context(t));
  }
  X1.removeDuplicateContext();
  return X1;
}
```

Listing A.18: Algorithm for implementing *union* on context set

Appendix B

Source Listing For Tag Set Types

```
boolean isInTagSet(GIPSYType poTag)
{
  boolean result = false;
  switch(this.iExpressionType){
    case : 0 {
      for(int i = 0; i < this.size(); i++){
        if(poTag == this.tagAt(i))
         result = true;
      }
      break;
    }
    case : 1 {
      if(poTag >= lower && poTag <= upper)
        result = true;
      break;
    }
    case : 2 {
      if(step > 0){
        if(poTag >= lower && poTag <= upper && (poTag-lower) mod step
            ==0)
          result = true;
      }
      else {
        if(poTag <= lower && poTag >= upper && (poTag-lower) mod step
            ==0)
      result = true;
      }

      break;
    }
    return result;
  }
}
```

Listing B.1: Algorithm for implementing `isInTagSet` on ordered finite tag set

```
boolean equals(Object otherObject)
{
  boolean result = false;
  if(otherObject instanceof OrderedFiniteTagSet){
   switch(this.iExpressionType){
    case : 0 {
      if((OrderedFiniteTagSet)otherObject.expressionType == 0 &&
      (OrderedFiniteTagSet)otherObject.getEnumeratedElements().equals(
          this.getEnumeratedElements()))
        result = true;
      break;
    }
    case : 1 {
      if((OrderedFiniteTagSet)otherObject.expressionType == 1 &&
      (OrderedFiniteTagSet)otherObject.getLower()==this.getLowewr() &&
      (OrderedFiniteTagSet)otherObject.getUpper()==this.getUpper())
        result = true;
      break;
    }
    case : 2 {
      if((OrderedFiniteTagSet)otherObject.expressionType == 2 &&
     (OrderedFiniteTagSet)otherObject.getLower()==this.getLower() &&
     (OrderedFiniteTagSet)otherObject.getUpper()==this.getUpper() &&
     (OrderedFiniteTagSet)otherObject.getStep()==this.getStep())
       result = true;
     break;
   }
  }
 }
 return result;
}
```

Listing B.2: Algorithm for implementing `equals` on ordered finite tag set

```
GIPSYType getPrevious(GIPSYType poTag)
{
  GIPSYType result = null;
  if(isInTagSet(poTag)){
   switch(this.iExpressionType){
    case : 0{
      if(!poTag.equals(this.tagAt(0))) //the parameter is not the first
          element
        result = this.tagAt(getIndex(poTag)-1);
      break;
    }
    case : 1{
      if(poTag != lower)
        result = poTag-1;
      break;
    }
    case : 2{
      if(poTag != lower)
        result = poTag-step;
      break;
    }
   }
  }
  return result;
}
```

Listing B.3: Algorithm for implementing `getPrevious` on ordered finite tag set

```
GIPSYType getNext(GIPSYType poTag)
{
  GIPSYType result = null;
  if(isInTagSet(poTag)){
   switch(this.iExpressionType){
    case : 0{
      if(!poTag.equals(this.tagAt(this.size()-1))) //the parameter is
          not the last element
        result = this.tagAt(getIndex(poTag)+1);
      break;
    }
    case : 1{
      if(poTag != upper)
        result = poTag+1;
      break;
    }
    case : 2{
      if(poTag != upper)
        result = poTag+step;
      break;
    }
   }
  }
  return result;
}
```

Listing B.4: Algorithm for implementing `getNext` on ordered finite tag set

```
boolean isInTagSet(GIPSYType poTag)
{
  boolean result = false;
  switch(this.iExpressionType){
    case : 3 {
      if(poTag >= lower && poTag <= Integer.MAX_VALUE )
        result = true;
      break;
    }
    case : 4 {
      if(poTag >= lower && (poTag-lower) mod step==0 && poTag <= Integer
          .MAX_VALUE)
        result = true;
      break;
    }
    case : 5 {
      if(poTag <= upper && poTag >= Integer.MIN_VALUE)
        result = true;
      break;
    }
     case : 6 {
      if(poTag <= upper && (upper-poTag) mod step==0 && poTag >= Integer
          .MIN_VALUE)
        result = true;
      break;
    }
     case : 7 {
      if(poTag >= Integer.MIN_VALUE && poTag <= Integer.MAX_VALUE)
        result = true;
      break;
    }
    return result;
  }
}
```

Listing B.5: Algorithm for implementing `isInTagSet` on ordered infinite tag set

```
boolean equals(Object otherObject)
{
  boolean result = false;
  if(otherObject instanceof OrderedInfiniteTagSet){
   switch(this.iExpressionType){
    case : 3 {
      if((OrderedInfiniteTagSet)otherObject.expressionType == 3 &&
      (OrderedInfiniteTagSet)otherObject.getLower.equals(this.getLower))
        result = true;
      break;
    }
    case : 4 {
      if((OrderedInfiniteTagSet)otherObject.expressionType == 4 &&
      (OrderedInfiniteTagSet)otherObject.getLower()==this.getLower() &&
      (OrderedInfiniteTagSet)otherObject.getStep()==this.getStep())
        result = true;
      break;
    }
    case : 5 {
      if((OrderedInfiniteTagSet)otherObject.expressionType == 5 &&
     (OrderedInfiniteTagSet)otherObject.getUpper()==this.getUpper())
       result = true;
     break;
   }
   case : 6 {
      if((OrderedInfiniteTagSet)otherObject.expressionType == 6 &&
     (OrderedInfiniteTagSet)otherObject.getUpper()==this.getUpper()&&
     (OrderedInfiniteTagSet)otherObject.getStep()==this.getStep())
       result = true;
     break;
   }
   case : 7 {
      if((OrderedInfiniteTagSet)otherObject.expressionType == 7)
       result = true;
     break;
   }
  }
 }
 return result;
}
```

Listing B.6: Algorithm for implementing `equals` on ordered infinite tag set

```
GIPSYType getPrevious(GIPSYType poTag)
{
  GIPSYType result = null;
  if(isInTagSet(poTag)){
   switch(this.iExpressionType){
    case : 3{
      if(poTag != lower)
        result = poTag-1;
      break;
    }
    case : 4{
      if(poTag != lower)
        result = poTag-step;
      break;
    }
    case : 5{
      if(poTag != Integer.MIN_VALUE)
        result = poTag-1;
      break;
    }
    case : 6{
      if(poTag != Integer.MIN_VALUE)
        result = poTag-step;
      break;
    }
    case : 7{
      if(poTag != Integer.MIN_VALUE)
        result = poTag-1;
      break;
    }
   }
  }
  return result;
}
```

Listing B.7: Algorithm for implementing `getPrevious` on ordered infinite tag set

```
GIPSYType getNext(GIPSYType poTag)
{
  GIPSYType result = null;
  if(isInTagSet(poTag)){
   switch(this.iExpressionType){
    case : 3{
      if(poTag != Integer.MAX_VALUE)
        result = poTag+1;
      break;
    }
    case : 4{
      if(poTag != Integer.MAX_VALUE)
        result = poTag+step;
      break;
    }
    case : 5{
      if(poTag != upper)
        result = poTag+1;
      break;
    }
    case : 6{
      if(poTag != upper)
        result = poTag+step;
      break;
    }
    case : 7{
      if(poTag != Integer.MAX_VALUE)
        result = poTag+1;
      break;
    }
  }
 }
  return result;

}
```

Listing B.8: Algorithm for implementing `getNext` on ordered infinite tag set