OO-IP Hybrid Language Design and a Framework Approach to the GIPC

Ai Hua Wu

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy (Computer Science) at

Concordia University

Montreal, Quebec, Canada

April 2009

Your file Votre référence
ISBN: 978-0-494-63431-8
Our file Notre référence
ISBN: 978-0-494-63431-8

ABSTRACT

OO-IP Hybrid Language Design and a Framework Approach to the GIPC

Ai Hua Wu, Ph.D.
Concordia University, 2009

Intensional Programming is a declarative programming paradigm in which expressions are evaluated in an inherently multidimensional context space. The Lucid family of programming languages is, to this day, the only programming languages of true intensional nature. Lucid being a functional language, Lucid programs are inherently parallel and their parallelism can be efficiently exploited by the adjunction of a procedural language to increase the granularity of its parallelism, forming hybrid Lucid languages. That very wide array of possibilities raises the need for an extremely flexible programming language investigation platform to investigate on this plethora of possibilities for Intensional Programming. That is the purpose of the General Intensional Programming System (GIPSY), especially, the General Intensional Programming Compiler (GIPC) component.

The modularity, reusability and extensibility aspects of the framework approach make it an obvious candidate for the development of the GIPC. The framework presented in this thesis provides a better solution compared to all other techniques used to this day to implement the different variants of intensional programming.

Because of the functionality of hybrid programming support in the GIPC framework, a new OO-IP hybrid language is designed for further research. This new hybrid language combines the essential characteristics of IPL and Java, and introduces the notion of *object streams* which makes it is possible that each element in an IPL stream could be an object with embedded intensional properties. Interestingly, this hybrid language also brings to Java objects the power which can explicitly express context, creating the novel concept of intensional objects, i.e. objects whose evaluation is context-dependent, which are therein demonstrated to be translatable into standard objects. By this new feature, we extend the use and meaning of the notion of object and enrich the meaning of stream in IPL and semantics of Java.

At the same time, during the procedure to introduce intensional objects and this OO-IP hybrid language, many factors are considered. These factors include how to integrate the new language with the GIPC framework design and the issues related to its integration in the current GIPSY implementation. Current semantic rules show that the new language can work well with the GIPC framework and the GIPSY implementation, which is another proof of the validity of our GIPC framework design.

Ultimately, the proposed design is put into implementation in the GIPSY and the implementation put to test using programs from different application domains written in this new OO-IP language.

ACKNOWLEDGEMENTS

I would like to express my deep gratitude to my supervisor Dr. Joey Paquet for being a great mentor. What I learned from him cannot only be covered by these pages. Ph.D study is a long-term task. His experience, his patience, his encouragement and his understanding give me great help to carry through. I really appreciate that he is always there to help especially at the last hardest step. I would also like to express my sincere appreciation to Dr. Peter Grogono for his insightful comments and valuable advice. Without their contribution, this thesis would not have been possible. Thanks also go to Dr. Weichang Du, Dr. Ferhat Khendek and Dr. Greg Butter for providing great advice and for accepting to judge this thesis. In the mean time, I have to mention my friendly team members, Serguei A. Mokhov, Paula Bo Lu, Yimin Ding, Lei Tao, Emil Vassev, Chun Lei Ren and Kai Yu Wan, for their outstanding team work. It is the whole team effort which makes the GIPSY becoming a running system. Finally and specially, I would like to thank my dearest parents, my sisters and my brother. Their love has always been a great encouragement for me.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF CODE EXCERPTS

Chapter 1 : Introduction

Hybrid programming is the process of building programs in which the source code is written in two or more languages, possibly belonging to different programming paradigms. Frameworks are used in many application domains because of their modularity, adaptability and extensibility aspects. These are two separated domains. This thesis presents how to use the framework approach to develop the General Intensional Programming Compiler (GIPC) in the General Intensional Programming System (GIPSY), which can support hybrid programming between Intensional Programming Languages (IPL) and Java, one of the Object Oriented Programming Languages (OOPL). Based on the feasibility of hybrid programming in the GIPC, as proven by earlier successes by Mokhov [Mok05], the notion of *intensional object* is introduced into the Java language to form a hybrid object-oriented intensional programming language. This thesis presents a new OO-IP hybrid language which combines the essential features of IPL and Java. The integration between this hybrid language and the GIPC framework effectively demonstrates that the design of the GIPC framework achieves the original goals at flexibility, generality and adaptability.

1.1 Problem Description

Intensional Programming (IP) is a declarative programming paradigm in which expressions are evaluated in an inherently multidimensional context space [PGW04]. The Lucid family of programming languages is, to this day, the only programming languages of true intensional nature. Lucid being a member of the family of functional programming languages, Lucid programs are inherently parallel and their parallelism can be efficiently exploited by the adjunction of a procedural language to increase the granularity of its parallelism, forming hybrid Lucid languages. Thus, Lucid itself forms a family of languages, and all these languages can be executed in sequential, parallel, or distributed mode, and the two latter modes of execution can be using various architectures and technologies for exchanging information between processing units.

Motivations

There are two topics in this thesis, one is about GIPC framework design and the other is about OO-IP hybrid language design. The reason why we design GIPC in a framework manner is because that very wide array of possibilities (pure intensional languages, hybrid languages, different possible execution schemes, middleware technologies, and application domains) raises the need for an extremely flexible programming language investigation platform to investigate on this plethora of possibilities for IP. That is the rationale for the development of the GIPSY. As a main component in the GIPSY, the General Intensional

Programming Compiler (GIPC) component plays an important role to achieve the high flexibility, especially from the point of view of developing new variants of the Lucid language, and experimenting with them using a flexible language development platform.

The modularity, reusability and extensibility aspects of the framework approach [FS97] make it an obvious candidate for the development of the GIPC. In this thesis, the GIPC framework is designed in a way that enables the automated generation of framework hot spots to improve the generality of the system in terms of programming language development support.

Regarding why to design an OO-IP hybrid language, there are 3 reasons.

1. Compared to pure dataflow programs and functions, functions written in a procedural language can execute coarser-grained data elements to increase the granularity of computation.
2. Intensional Programming Languages are off-stream languages, but object-oriented languages are very popular language in industry, mixing IP with standard language can help IP to be accepted by more users.
3. This hybrid language is not only one-way hybrid; it is a two-ways hybrid language. It allows the IP language to mix with Java language, which will allow the IP language counterpart to use most features of Java, for example, the object concept, the inheritance of Java, the powerful input/output of Java. These features introduced in this OO-IP hybrid language will make it better than pure Lucid languages. It will also allow Java language to use IP language, which will bring intensional and

dimensional concepts into Java. By invoking simple IPL code, this hybrid language will allow Java to express infinite streams with multi-dimensions and evaluate them lazily using a demand-driven execution mechanism, which is impossible in traditional Java.

These two topics look separated, but in fact, they are closely related. The GIPC framework design makes it is very easy to add a new IP language, and it provides the basic support to make the implementation of the hybrid language possible. At the same time, the easy and correct integration between the hybrid language and the GIPC framework or the current GIPSY system, also proves the effectiveness of the GIPC framework design.

Goals

At the end of this thesis, we aim to achieve the following goals:

1. Generalize my Master thesis result to create a framework that uses the Master thesis concept of component generation.
2. Design the OO-IP hybrid language to clarify the concepts
    a. Define the syntax of the language;
    b. Define the semantics of the language which will describe the requirements precisely for implementers and will tell users what to expect exactly;
    c. Show examples to explain how is the language used and what are its advantages;
3. Implement a compiler for this OO-IP hybrid language, and also make sure the compiler is designed to fit our current GIPSY system's architecture.

1.2 Thesis Contributions

This thesis is about hybrid languages, as well as framework design approach to compiler and programming language development. More specifically, it aims at:

1 **Characteristics analysis of the intensional programming paradigm:** the essential characteristics of IPL are the cornerstone for the current design; the diversity of languages in the Lucid family of languages, and the diversity of their applications tell us on the necessity to develop a flexible platform for this language paradigm.

2 **Dynamic framework design of GIPC:** provides a solution compared to all other techniques currently used to implement the different variants of intensional programming. All of these automated generation units are hot spot generators that generate different components of the framework, which can then be automatically linked to the framework to provide new capacities to the system.

  - o Automated generation of intensional programming language parsers to ease and normalize the generation of parser components.
  - o Automated generation of SIPL-AST to GIPL-AST translators to enable the semantic translation and execution of any IPL flavor through its translation into the Generic Intensional Programming Language primitives, which is the only one understandable by the General Eduction Engine.

    - Automated generation of sequential thread generator to adapt to different imperative languages that are to be used in conjunction with IPLs to form hybrid languages.

3 **Existing compiler construction systems analysis:** discuss the related work and provide the comparison with the GIPC framework design.

    - Select typical systems and analyze its advantages and shortcomings.
    - Define a set of criteria to evaluate compiler construction systems to provide a comparison basis with the capacities of the GIPSY.

4 **OO-IP hybrid language design:** introduces intensional concept into Java and makes Java also to have variables that have explicit dimensions to naturally express the notion of intensional objects. Inversely, a description of the syntax and semantics of the introduction of object data type in IPLs is provided. Detailed constructions include:

    - Introduction of intensionality into the Java language, i.e. *intensional objects.*
    - Introduction of the concepts on *context-mutable object* and *context-immutable object.*
    - Integration of the above within the GIPC framework design and the current GIPSY implementation.
    - Such a successful integration provides a proof for the validity of the GIPC framework design approach.
    - Formally define the syntax and semantics of the hybrid language

- o Implementation of the design into the GIPSY.
- o Use of the resulting implementation to execute some programs written in the new OO-IP language.

5 **Other hybrid programming systems comparison:** investigation to demonstrate that the new hybrid language is an original contribution to the field of IP language development.

1.3 Thesis Organization

Chapter 1 presents the concept of intensional programming paradigm. Chapter 2 presents the overview of the GIPSY system. Chapter 3 presents framework technology and the reason why we adopt this methodology for the GIPC design. Chapter 4 presents the framework design for the GIPC from implementation level and generation level. Chapter 5 compares related work on compiler construction system design. Chapter 6 narrows down the topic to hybrid language and introduces a new OO-IP hybrid language by formally defining the syntax and semantics as well as implementation details and application discussion. Chapter 7 discusses related work on hybrid programming system between IPL and OO. Finally, Chapter 8 states conclusion and future work.

1.4 Introduction to the Intensional Programming Paradigm

There are two main classes of programming languages: imperative languages and declarative languages. The former is characterized as having an implicit state that is modified by imperative constructs in the source language. As a result, such languages generally have a notion of sequencing to permit precise and deterministic control over the state during program execution. Most of the widely used languages in existence today are imperative. The latter is characterized as having no implicit state, and thus the emphasis is placed entirely on programming with expressions providing invariant characteristics of the elements manipulated by the program, as well as invariant relationships between these elements of the program. In particular, functional languages are declarative *languages whose underlying model of computation is strictly the mathematical* notion of function.

Dataflow languages are a variety of functional languages. With the attention being paid on concurrency, both architectures and languages for concurrency have been around for some time. Dataflow is one way to achieve concurrency, particularly at the fine-grain level. Dataflow architectures focus on the data dependencies between the elements declared and manipulated by a program. The languages designed to support such machines are called *dataflow languages* [Fin95]. The Intensional Programming Language we will introduce in this chapter can be categorized into the group of dataflow programming languages.

1.5 Dataflow Languages

Programs in an imperative language are intended to be run on standard von Neumann machines. A von Neumann machine consists of a processor attached to a memory that is an indexed collection of storage locations. A program forms a sequence of control instructions that determine the order in which values are extracted from memory locations; computations using these values are then performed, and the resulting values are stored in the memory. Sequential execution is an essential characteristic of Von Neumann computer architectures. The concepts embodied by the von Neumann architecture have not been directly applicable to the domain of parallel computation.

This limitation led to the introduction of the dataflow architecture that offers a simple yet powerful formalism for describing parallel computation. The dataflow *computation model allows the simultaneous execution of several instructions* purely on the availability of data, provided there is enough concurrency in the application and there are sufficient resources available.

Dataflow languages have no concept of machine state or sequential execution of program segments. Rather, they allow the declarative description of variables. Dataflow languages have no imperative statements or commands instead of expressions [Ost81]. The expression structure of a dataflow language is usually quite powerful; besides the usual operators and function invocations, there are several conditional expressions, for example “if-then-else”. Since there really is no "state", the expressions of a dataflow language do not have side effects. So, dataflow languages are free of side effects [AW77]. This property is all-important. This makes it possible to translate subroutines separately, without unnecessary constraining parallelism.

Moreover, dataflow languages need the locality of effect. They simplified the problem by assigning every variable a definite “scope,” or region of the program in which it is active, and carefully restricting the entry to and exit from the blocks that constitute scopes. This characteristic of dataflow languages makes it possible to execute programs with parallelism.

Finally, functional semantics is an important characteristic of dataflow languages. It means in dataflow languages variables stand for values and not for memory locations. This is different from imperative languages, which allow programmers to be aware of and have some control over the primary memory allocation for both programs and data, dataflow languages only allow programmers to deal with values. Functional semantics offers the advantage of a simplified translation process to parallel processing.

1.6 A View of the History of Lucid

The Lucid programming language has a very non-standard history for a programming language. During its lifetime, it went from and to different goals in mind, had diversified syntactical forms, different execution models were applied to it and even invented for it. Without claiming to explain all this history in details, we present here a perspective of it that pertains and is adapted to the subject we are tackling.

In the 1980s, Lucid was used as a kind of dataflow language designed to experiment with non-Von Neumann programming models. Besides having the characteristics of dataflow languages, it has fundamentally different semantics from a language like C or Lisp: Lucid expressions and their valuations are allowed to vary in an arbitrary number of dimensions [AW77]. From this perspective, we briefly introduce the evolving history of Lucid [Paq99].

1.6.1 Pipeline Dataflow

The original Lucid dates back to 1974 when Ashcroft and Wadge were working on a purely declarative language in which iterative algorithms could be expressed naturally [AW77]. We can say Lucid's history began from the goal of having a language usable for program verification. Ashcroft and Wadge's work at the time was suitable to the broad area of research into programming languages semantics and program verification [AW76]. The original goal was to use Lucid to describe the sequences of values that are theoretically supposed to be taken throughout the lifetime of variables in an imperative program, as well as the declarative expression of dependencies between these variables. In doing so, one could compare the actual values taken by the variables during run-time, and compare them with the values as defined by the Lucid declarations.

In the original Lucid, streams were defined in a pipeline manner, with two separate definitions: one for the initial element, and another one for the subsequent elements. The following are an example:

(1) **`first X = 1`**

(2) **`next X = X + 1`**

The equations define variable **`X`** to be a stream. Equation (1) defines the initial element:

$X_0 = 1;$

and equation (2) gives the definition of the stream

$X : X_{i+1} = X_i+1.$

Based on the equations, we can get the stream

$X = (X_0, X_1, \ldots, X_i, \ldots) = (1, 2, \ldots, i, \ldots)$.

1.6.2 Tagged-token Dataflow

There are limitations in the original Lucid. The first one is that the *(i+1)-th* element in a stream is only computed once the *i-th* element has been computed. This wastes resources especially under the situation in which the *i-th* element might not necessary be required otherwise than to compute the *(i+1)-th* element. More importantly, it only permits sequential access into streams.

This introduces a different approach, which is random access into a stream by using an index # corresponding to the current position. This version of Lucid was eventually called *indexical Lucid* [FJ91]. At this point, we are defining computation according to a context instead of manipulating infinite extensions.

It is only at this point in time that Lucid set out on the road to intensional programming. All operators in original Lucid can be redefined in terms of the operators, # and @. For example,

(1) **`first X ⇔ X @ 0`**

(2) **`next X ⇔ X @ (# + 1)`**

Equation (1) illustrates operator "**`first`**" means the first element in stream **`X`** and can be redefined by "**`X @ 0`**" (literally: the value of stream **`X`** at index 0); equation (2) illustrates operator "**`next`**" means the element just after the current element and can be redefined by "**`X @ (# + 1)`**" (literally: the element lies the position which is current context # increased by 1 in the stream **`X`**). Specific proofs of these equivalences can be found in [Paq99].

Accompanying the introduction of this new version of Lucid, attempts came at implementation. The first widely distributed implementation of Lucid is Ostrum's Luthid interpreter [Ost81]. The technique used in the interpreter is eduction. It can be described as "tagged-token demand-driven dataflow", in which data elements are computed on demand following a dataflow network defined in Lucid.

1.6.3 Multidimensional Dataflow

Until now, Lucid only allows one to define simple stream instead of permitting any sort of sub-computation, in which the sub-computation itself requires streams. To achieve this, a general solution was also provided in Indexical Lucid [FJ91]. Sub-computations could take place in arbitrary dimensions, and all indexical operators would be parameterized by one or several dimensions. The `#` and `@` operators became `#.d` and `@.d`, where `.d` allows one to query about part of the evaluation context, rather than the entire evaluation context [Paq99]. For example, the operator "upon" became as Excerpt 1-1:

```
X upon.d Y = X @.d W
where
  W = 0 fby .d if Y then (W+1) else W;
end;
```

Excerpt 1 - 1: definition of operator "upon"

Here, the "**`where`**" clause in which local dimensions can be locally declared is introduced, so that dimensions can be declared and used as necessary for sub-computations. This solution also solved the problem of representing data such as multidimensional matrices by using additional dimensions.

If Indexical Lucid allows dimensions to be passed as parameters to functions, then it can have dimensions as values, which was introduced in 1999 by Paquet [Paq99]. The definition of operators, `#` and `@`, are changed again by `#.E` and `@.E`. The idea in `#.E` is to evaluate expression `E`, which at some point will evaluate to a dimension `d` used to query the execution context and return the index corresponding to the `d` dimension.

1.6.4 Intensional Programming

Each version of Lucid tended to generalized the concepts of the previous versions. Today, the general idea is to develop an intensional programming language which involves the programming of expressions placed in an inherent multidimensional context space [Paq99].

In conventional programming, values are calculated in a particular context, usually indicated by subscripts. For example, `a[x]` denotes the value of variable `a` at position `x` and `b[x, t]` denotes the value of variable `b` at position `x` and time `t`. In intensional programming, the context is implicit. As a result, programs are more concise and closer to their underlying mathematical formalism. With one implicit dimension, intensional programming is called *unidimensional* and, with more than one dimension, it is called *multidimensional*. The Generic Intensional Programming Language (GIPL) is the latest and most generic offspring of the Lucid family of language.

1.7 A Simple Lucid Program Example – the Hamming Problem

As we discussed above, Lucid has two basic intensional operators, one is used respectively for intensional navigation (`@`) and the other is to query the current context of evaluation of the program (`#`). In order to well understand the concepts of Lucid, we give a typical Lucid program – the Hamming program.

The Hamming problem [WA85] consists of generating the stream of all numbers of the form $2^i3^j5^k$ in increasing order and without repetition. It sounds simple, but we will find it is surprisingly intricate if we use an imperative language to resolve the problem. The conditions under dataflow language are different. The following Indexical Lucid program can easily solve this problem, as shown in Excerpt 1-2. Figure 1-1 represents the dataflow diagram for the Hamming problem [Paq99].

```
H
where
  H = 1 fby merge(merge(2*H,3*H),5*H);
  merge(x,y)= if (xx<=yy) then xx else yy
  where
    xx = x upon (xx<=yy);
    yy = y upon (yy<=xx);
  end;
end;
```

Excerpt 1 - 2: Indexical Lucid program for the Hamming problem

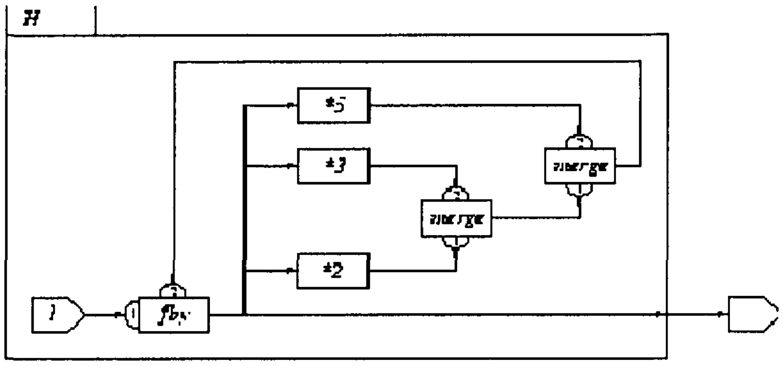


Figure 1 - 1: Dataflow graph for the Hamming problem

The result of the Hamming problem will be the stream:

H = (1,2,3,4,5,8,9,16,25,27,...).

1.8 Abstract Syntax for Lucid

Lucid being a type-less language (i.e. a language where data types are not explicitly referred to), its implementations normally supported a very small set of data types: integer and real numbers. All variants of Lucid include function application (by extension including operators), conditional expressions, intensional navigation (@) and intensional query (#). Table 1-1 summarizes the abstract syntax for Lucid [Paq99].

```
E ::= id
      |   E (E1,...,En)
      |   if E then E′ else E′′
      |   # E
      |   E @ E′ E′′
      |   E where Q
Q ::= dimension id
      |   id = E
```

|	`id (id₁,…,id ₙ) = E`
|	`QQ`

Table 1 - 1: Abstract syntax for Lucid

The non-terminals `E` and `Q` respectively refer to expressions and definitions. The syntax of Lucid was deliberately designed to be unusual and different, to prevent programmers from applying procedural-programming habits that might be inapplicable, and to sustain the illustration of data flows as infinite objects.

1.9 Semantics of Lucid [Paq99]

The operational semantics of Lucid would be the following general form:

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

i.e. under the definition environment ***D***, and in the evaluation context ***P***, expression E would evaluate to value v. The definition environment ***D*** retains the definitions of all of the identifiers that appear in a Lucid program. It is a partial function

$$\mathrm{D} : id \rightarrow IdEntry$$

Where ***id*** is the set of all possible identifiers and ***IdEntry*** has five possible kinds of value. They are:

- **Dimensions**: define the coordinates in which one can navigate with the **#** and `@` operators. The *IdEntry* is `(dim)`.
- **Constants**: are external entities that provide a single value, whatever the context. The *IdEntry* is `(const, c)`.
- **Data operators**: are external entities that provide memory-less functions.

The *IdEntry* is **`(op, f)`**, where **`f`** is the function itself.

- **Variables**: carry the multidimensional streams. The *IdEntry* is **`(var, E)`**, where **`E`** is the expression defining the variable. We assume that all variable names are unique. This constraint should be easy to overcome by performing compiler-time renaming or using a nesting level environment.
- **Functions**: are user-defined functions. The *IdEntry* is **(func, $\mathbf{id}_i$, E)**, where the $\mathbf{id}_i$ are the formal parameters to the function and **`E`** is the body of the function. Lucid encourages the use of iteration rather than recursion even though the semantics for recursive functions is permitted.

will be changed when the `@` operator or a *where* clause is encountered and it associates a tag to each relevant dimension. It is a partial function

The operational semantics is defined in Figure 1-2.

$$\mathbf{E}_{\mathrm{cid}} : \frac{\mathcal{D}(id) = (\mathtt{const}, c)}{\mathcal{D}, \mathcal{P} \vdash id : c} \qquad \mathbf{E}_{\mathrm{did}} : \frac{\mathcal{D}(id) = (\mathtt{dim})}{\mathcal{D}, \mathcal{P} \vdash id : id}$$

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

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

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

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

$$\mathbf{E}_{\mathrm{cT}} : \frac{\mathcal{D}, \mathcal{P} \vdash E : true \qquad \mathcal{D}, \mathcal{P} \vdash E' : v'}{\mathcal{D}, \mathcal{P} \vdash \mathtt{if}\ E\ \mathtt{then}\ E'\ \mathtt{else}\ E'' : v'}$$

$$\mathbf{E}_{\mathrm{cF}} : \frac{\mathcal{D}, \mathcal{P} \vdash E : false \qquad \mathcal{D}, \mathcal{P} \vdash E'' : v''}{\mathcal{D}, \mathcal{P} \vdash \mathtt{if}\ E\ \mathtt{then}\ E'\ \mathtt{else}\ E'' : v''}$$

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

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

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

$$\mathbf{Q}_{\mathrm{dim}} : \frac{}{\mathcal{D}, \mathcal{P} \vdash \mathtt{dimension}\ id : \mathcal{D} \dagger [id \mapsto (\mathtt{dim})], \mathcal{P} \dagger [id \mapsto 0]}$$

$$\mathbf{Q}_{\mathrm{id}} : \frac{}{\mathcal{D}, \mathcal{P} \vdash id = E : \mathcal{D} \dagger [id \mapsto (\mathtt{var}, E)], \mathcal{P}}$$

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

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

Figure 1 - 2: Semantics of Lucid [Paq99]

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.

E$_{\mathbf{at}}$: The rule for the navigation operator corresponds to the syntactic expression **`E @ E′ E′′`**, evaluates **`E`** in context **`[E′:E′′]`**, where **`E′`** evaluates to a dimension and E'' evaluates to a value corresponding to a tag in E'. The function $P'(x) = P \dagger [id \mapsto v'']$, if $x{=}id$, then $P'(x)$ is v''; otherwise it is $P(x)$.

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

$\mathbf{E_{fct}}$: Function calls require the renaming of the formal parameters into the actual parameters (as represented by $E'[id_i \leftarrow E_i]$).

$\mathbf{Q_{dim}}$: adds a dimension to the definition environment and adds this dimension to the context of evaluation with tag 0 (as a convention).

$\mathbf{Q_{id}}$ **and** $\mathbf{Q_{fid}}$: add variable and function identifiers along with their definition to the definition environment.

1.10 Characteristics of Lucid

Being a dataflow language, Lucid inherits features from dataflow languages, such as being free of side-effect. However, it still has its specific features. The following introduces characteristics of Lucid.

1.10.1 Stream

The distinguishing feature of Lucid is that identifiers are used to represent streams of values, thus allowing the expression of iteration in a rather concise manner. It is easy to do nontrivial problems in Lucid without using anything like an array just because Lucid program uses streams. Lucid is based on data in motion and streams play a crucial role.

1.10.2 Loop in Lucid

The simplest loop consists of a single variable specified inductively in terms of itself and some constants. For example,

```
first V = 1
next V = 2 × V
```

We can interpret a loop like this as having the effect of first initializing the variable **V**, and then repeatedly reassigning to it, so that it takes on the values 1, 2, 4 . . . . We can call **V** as loop variable. Lucid manages to treat assignment statements as equations, and to make loops implicit, only by imposing restrictions on the use of assignments. These restrictions all follow from the fact that a variable in a program can have only one specification, whether direct or inductive.

For a directly specified variable, the restriction is that the variable can be assigned to at only one place in the program. For example, the two equations

```
X = Y + I
X = A * B
```

cannot both appear in the same program. For an inductively specified variable, the restriction is that the variable can be assigned to only twice in the program, once for initialization and once for updating. For one thing, this means that a loop variable cannot be updated twice; thus the equations

```
next V = 3 x V
next V = V + 1
```

cannot both appear in the same program. If an intermediate value is needed, a separate variable must be used, for example,

```
V_I = 3 * V
next V = V_1 + 1
```

The restriction on inductive definition also means that every loop variable *must* be updated, whether the value is changed or not. The following statement cannot appear in a program:

```
    if X < Y then next Y = Y - X
```

because it is not even an equation. Instead, we must write

```
    next Y = if X < Y then Y - X else Y
```

In a sense, Lucid allows only "controlled" or "manageable" use of assignment in much the same way as a conventional structured programming language allows only controlled or manageable use of transfer.

1.10.3 Family of IPL — GIPL and SIPLs

The Lucid family of programming languages is evolving and growing. As mentioned in Section 1.6, Lucid is the first intensional programming language. However, its evolution has lead to today's Lucid family of intensional programming languages. For example, the GIPL is the generic language of the Lucid family of languages [Paq99], which is composed of a very basic set of operations and syntactic structures, such as the generic intensional operators `@` and `#`, an **`if`** syntactical structure, functions, dimension declarations, and a **`where`** clause used to introduce local scopes. The `@` operator represents a change of context, whereas the `#` operator represents the interrogation of the inherent context of evaluation.

The SIPLs (Specific Intensional Programming Languages) each have a set of domain-specific operations. Each SIPL is a conservative extension of the GIPL, i.e., all additional operations defined in any SIPL can be translated into GIPL primitive operations [Paq99]. Different sets of operations, pertaining to a different flavor of intensional programming, or to a different application domain, correspond to different SIPL versions. For example, the operations **`first`**, **`next`**, **`prev`**, **`fby`**, **`wvr`**, **`asa`**, and **`upon`**, along with the **`where`** clause, form a SIPL named Indexical Lucid.

It was proven that all SIPLs currently known share the semantics of the GIPL [Paq99]. This means that the GIPL is the core language (hence its name identifying it as *generic*) and all other SIPLs can be translated into its primitives. See [Wu02] for a description of semantic translation rules between the Indexical Lucid SIPL and the GIPL. If we only use two intensional operators, the program in Excerpt 1-1 can be translated into the general intensional program as shown in Excerpt 1-3.

```
H
   where
     dimension d;
     H = if (#.d) == 0 then 1 else merge(merge(2*H, 3*H), 5*H);
     merge(x,y) = if (xx<=yy) then xx else yy
       where
         xx = x @.d W
           where
             W = if (#.d) == 0
                     then 0
                     else ( if (xx <= yy)then W+1 else W fi)
                     fi;
           end;
         yy = y @.d V
           where
             V = if (#.d) == 0
                    then 0
                    else ( if (yy <= xx) then V+1 else V fi)
                    fi;
           end;
       end;
   End
```

Excerpt 1 - 3: Generic Lucid program for the Hamming problem

1.10.4 Different from other Dataflow Languages

An "assignment statement" in Lucid can be considered as a statement of identity or an equation. A correctness proof of a Lucid program proceeds directly from the program text, the statements of the program being the axioms from which the properties of the program are derived. Furthermore, in Lucid we are not restricted to proving only partial correctness or only termination or only equivalence of programs- Lucid can be used to express many types of reasoning.

Lucid has one great advantage: the programmer is not totally restricted to dataflow and can think in terms of other operational concepts as well [WA85]. It is true that iteration is not dataflow, but why should everything be reduced to "pure" dataflow? Lucid does not force either the programmer or the implementation to make a choice between dataflow and iteration. In fact, it is quite possible that the programmer's view of the computation can be very different from the actual implementation. The actual implementation might not use any data flow at all.

However, Lucid is still a "dataflow language" because 1) Pipeline dataflow model of computation can be used for and 2) the above computation is very effective to implement Lucid programs [WA85].

Lucid differs from that of some other dataflow languages because 1) it is a nonprocedural language with a static denotational semantics; 2) programmers are encouraged to think in terms of dataflow and do not need to understand the actual implementation details; 3) other forms of operational activity, i.e. modes of computation, are available to the programmer [WA85].

Lucid offers the possibility to achieve both efficiency and efficacy. The programmer can influence the way in which the computations are to be performed. Since the operational side is only indicated, not specified, the programmer is spared the chore of planning the entire computation in all its technical details such as computation scheduling. Programmers therefore have no need of extra "dirty" features, of all those levers, handles and switches which the imperative programmers need to control their machine. The language can therefore be relatively simple. [WA85]

1.11 Input/Output in Lucid

The information-stream view of input/output is inadequate. It does not allow the programmer to specify the rate at which output is produced and input consumed. Even, it does not allow the programmer to specify the way in which input and output interleave. The syntax of Lucid also shows this point. This is a weakness of all "pure" (i.e. non-hybrid) Lucid dialects. However, the problem can be solved by using a more elaborate notion of stream, one in which "pause objects" can occur.

1.12 Application Domains

IP can be used to naturally represent and efficiently compute solutions to problems of intensional nature. In fact, there is a plethora of problems of intensional nature in almost all aspects of science and even in real life. For example, natural language makes a ubiquitous use of intensional logics, as represented with intensional words such as "yesterday" and "it", whose meaning depend on the context of utterance.

The vast majority of all pure and applied sciences are also making a ubiquitous use of intensional logics in differential, integral, or tensor equations, that have a consistent meaning across manifolds of different dimensionality. For example, in scientific domains, it is common to do computer simulations that normally correspond to the operational version of a set of differential equations. These intrinsically multidimensional and intensional equations allow complex physical phenomena to be represented very naturally. The programming of such equations in conventional languages does not reflect their original simplicity. The situation gets even much worse when programming tensor equations. Furthermore, very few mathematical programming languages and environments enables the high-speed (i.e. parallel/distributed) execution of naturally-expressed equation system.

It has been proven that intensional programming can be used to build programs to solve such problems and to achieve the high-performance parallel/distributed computation of differential or tensor equations expressed in a natural manner [Paq99]. Moreover, intensional programming has been successfully applied to topics as diverse as reactive programming [HCRP91], software configuration [PW93] and distributed operating systems [Kro99].

Kropf discusses the Web Operating System (WOS) approach to global computing [Kro99]. The heterogeneous and dynamic nature of the Web or Internet makes it impossible to define a fixed set of operating system services, usable for all services. Rather, generalized software configuration techniques, based on a demand-driven technique called eduction, can be used to define versions of a Web Operating System that can be built in an incremental manner. This net-centric approach considers communication as the central issue as opposed to the common notion of central servers.

Plaice and Wadge give an algebraic version language which allows histories (numbered series), subversions, and joins to present a new approach to the control of versions of software and other hierarchically structured entities [PW93].

First conceived in 1996 by Wadge and Yoder, IHTML is an intensional version of popular Hypertext Markup-Language (HTML). The central idea of IHTML is that the markup elements of traditional HTML, such as links, images, and file-inclusion, can be versioned by intensional logic, with the underlying source-files being stored in the intensional repositories. Under this scheme, every requested URL in IHTML contains explicit versioning information; when a browser requests an intensional page, the server makes a best fit to the requested page-version, given existing source-file versions in the repositories under the document root of the server.

However, intensional programming is in its early stages of development, and recent history has proven that it is still an area that is extremely evolutionary and of general application, which impose very stringent flexibility and adaptability constraints on the development of programming tools using this paradigm.

1.13 Summary

In this chapter, we introduced the Intensional programming language from the point of view of its history, characteristics, and application domains. This was meant to give a first look at what exactly this language is; express its special characteristics in order to lay the basic stones for later design; and the variety of application domains tell us the necessity to develop platform for such language paradigm.

In the next chapter, we will move the topic to the system we are developing, i.e. the GIPSY. We will give the architecture of the whole system and then we will narrow the topic down to the compiler component, the GIPC.

Chapter 2 : GIPSY – General Intensional Programming System

The General Intensional Programming System (GIPSY) is a system that aims at effectively demonstrating that Intensional programming can be used as an effective solution to solve problems of intensional nature, and to efficiently develop and execute parallel/distributed programs through code reuse in an intuitive manner. The GLU parallel/distributed programming environment, developed at Stanford Research Institute (SRI) in Menlo Park, was the first intensional programming system that enabled the compilation of Indexical Lucid programs, together with the use of sequential threads written in C. It has proven to be a usable and highly efficient solution for the parallelization of sequential programs. However, the GLU system suffered from a lack of flexibility and adaptability. It could not cope with the latest evolutions of Lucid. Consequently, new tools for intensional programming are required. The design and implementation of GIPSY is done towards generality, flexibility and efficiency.

2.1 Architecture of the GIPSY

This section outlines the theoretical basis and architecture of the different modules of the system. The system is composed of three main subsystems: General Intensional Programming Compiler (GIPC), General Eduction Engine (GEE) and Run-time Interactive Programming Environment (RIPE). All these modules are themselves designed in a modular manner to permit the eventual replacement of each of its components, at compiler-time or even at run-time in some cases, to improve the overall efficiency and flexibility of the system. Figure 2-1 shows the architecture of the GIPSY [PK00].

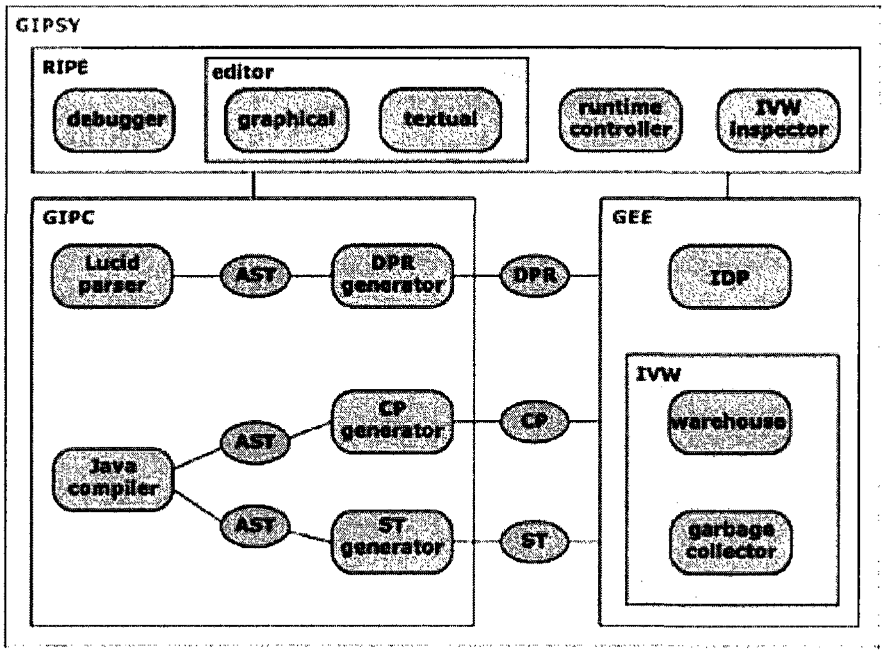


Figure 2 - 1: Architecture of the GIPSY

2.1.1 RIPE

The RIPE is a visual run-time programming environment enabling the visualization of a dataflow diagram corresponding to the Lucid parts of GIPSY programs. The user can interact with the RIPE at run-time in the following ways, among many others:

1 Dynamically inspect the Intensional Value Warehouse (IVW);
2 Change the input/output channels of the program;
3 Recompile sequential threads;
4 Change the communication protocol;
5 Change parts of GIPSY itself (e.g. garbage collector).

Because of the interactive nature of the RIPE, the GIPC is modularly designed to allow the individual on-the-fly compilation of either the DPR (by changing the Lucid code), CP (by changing the communication protocol) or ST (by changing the sequential code). Such a modular design even allows sequential threads to be programs written in different languages (for now, we are concentrating on Java sequential threads).

A graphical formalism to visually represent Lucid programs as multidimensional dataflow graphs had been devised in [Paq99]. The nested definitions will be implemented in the RIPE by allowing the user to expand or reduce sub-graphs, thus allowing the visualization of large scale Lucid definitions.

Using this visual technique, the RIPE will enable the graphic development of Lucid programs, translating the graphic version of the program into a textual version that can then be compiled into an operational version. An extensive and general requirements analysis will be undertaken, as this interface will have to be suited to many different types of applications. There is also the possibility to have a kernel run-time interface on top of which we can plug-in different types of interfaces adapted to different applications [PK00].

2.1.2 GEE

The GIPSY uses a demand-driven model of computation, whose principle is that a computation takes effect only if there is an explicit demand for it. A similar mechanism is used in functional languages such as Haskell, where it is known as call-by-need. GIPSY uses eduction (from the Latin for "to draw" or "to lead"), which is demand-driven computation in conjunction with a value cache called a warehouse. Every demand can potentially generate a procedure call, which is either computed locally or remotely, thus eventually in parallel with other procedure calls. A value is warehoused if it is cheaper to extract it from the warehouse than to re-compute it. Every demand for an already-computed value is extracted from the warehouse rather than computed again. Eduction thus reduces the overhead induced by the procedure calls needed for the computation of demands. The architecture design is as Figure 2-2 [Mok05].

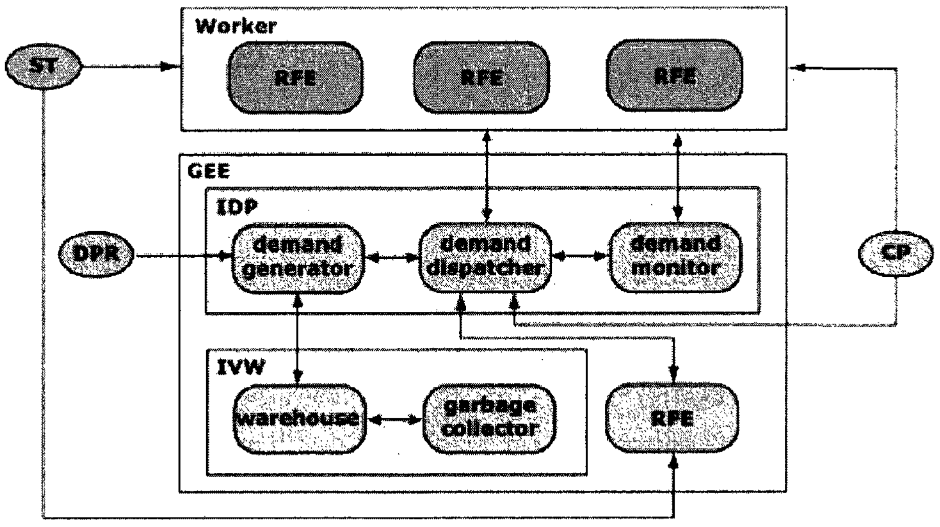


Figure 2 - 2: Architecture of the GEE

The GEE is composed of three main modules: the executor, the intensional demand propagator (IDP) and the intensional value warehouse (IVW). First, the intensional data dependency structure (IDS) which represents GEER is fed to the demand generator (DG) by the compiler (GIPC). This data structure represents the data dependencies between all the variables in the Lucid part of the GIPSY program in input. This directs in what order all demands must be generated to compute values from this program. The demand generator receives an initial demand, which in turn raises the need for other demands to be generated and computed. For all non-functional demands (i.e. demands not associated with the execution of a sequential thread (ST)), the DG makes a request to the warehouse to see if this demand has already been computed. If so, the previously computed value is extracted from the warehouse. If not, the demand is propagated further, until the original demand is resolved and is put in the warehouse for further use.

Functional demands, (i.e. demands associated with the execution of a sequential thread), are sent to the demand dispatcher (DD). The DD takes care of sending the demand to one of the workers or resolves it locally (which normally means that a worker instance is running on the processor running the generator process). If the demands are sent to a remote worker, the communication procedures (CP) generated by the compiler are used to communicate the demand to the worker. The DD receives some information about the lifecycle and efficiency of all workers from the demand monitor (DM), to help it make better decisions in dispatching functional demands.

The demand monitor, after some functional demands are sent to workers, gathers various information of each worker:

1 Its status (is it still alive, not responding, or dead)

2 Its network link performance

3 Its response time statistics for all demands sent to it

This information is accessed by the DD to make better decisions about the load balancing of the workers, and thus achieving better overall run-time efficiency. The details about GEE framework design can be found in [Lu04].

2.2 General Intensional Programming Compiler (GIPC)

The design and implementation of the GIPC is a main topic in this thesis. In this section, we will present the architecture of GIPC, as shown in Figure 2-3.

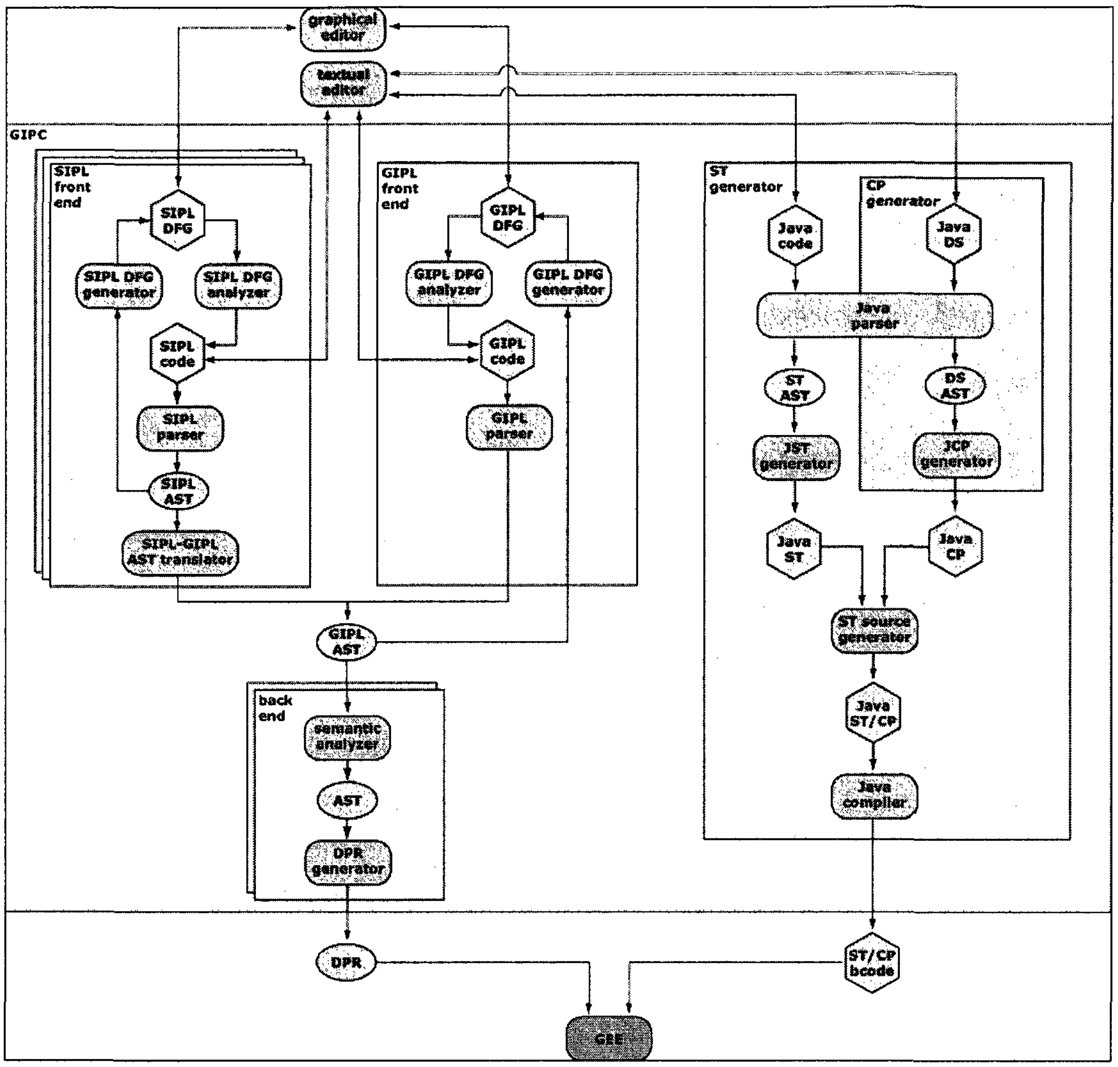


Figure 2 - 3: Architecture of the GIPC

Hybrid GIPSY programs are composed of two parts: a Lucid part that defines the intensional definitions of the variables and a sequential part that defines the granular sequential computation units. Programs are compiled in a two-stage process:

1. The intensional part of the GIPSY program is translated into Demand Propagation Resources (DPR) representing the dependencies between the elements of the program. The sequential part of the GIPSY program

(currently now restricted to Java) is itself composed of two parts: the first part is the Java functions that represent the Sequential Threads (ST), eventually executed in parallel on remote computer nodes. The second part is the data types' definitions representing the data elements that will be exchanged by the different computer nodes. These are translated into Communication Procedures (CP) that are called upon when remote demands are made. In our current implementation, the Communication procedures have been abstracted into calls to the Demand Migration Framework, which uses abstract tuple spaces of demand objects that are migrated between execution nodes [PVP07].

2. The DPR and the appropriate CPs are linked within the General Eduction Engine (GEE) to enable proper demand propagation and execution upon evaluation. This part is compiled into a *demand generator* component. The STs and appropriate CPs are packaged into Java classes and compiled into a *demand worker* component. Then, the resulting Java programs, generator and worker, are compiled in the standard way.

The Sequential Threads (STs) are now written in Java, for the sake of simplicity and code homogeneity with the system's implementation code. However, the ST generator component is developed in a flexible manner that will eventually allow the writing of sequential threads in other languages such as C, C++, Fortran, etc.

2.3 Summary

In this chapter, we give an introductory-level description of the three main components in the GIPSY system. By this big picture, we showed how the GIPC work with other components and what are the requirements for the GIPC. To achieve these requirements, we consider using a framework approach to develop the GIPC. In the next chapter, we proceed with the introduction on what is the framework approach to software development.

Chapter 3 : Software Frameworks

In object-oriented programming, the interpretation of “framework” ranges widely. It is a promising software development approach for reifying proven software designs and implementations in order to reduce the cost and of reusing software and improve the quality of software built through reuse of existing architecture and components.

In this chapter, we discuss features and technical issues about the framework approach to software development, and illustrate the reason why we adopt the framework approach for the GIPC.

3.1 Introduction

In the 1960s, people began to focus on software reuse. In the beginning, reusable software components have been procedural and function libraries; in the late 1960’s, Simula 67 [BDMN73] introduced objects, classes and inheritance which resulted in class libraries. However, both function libraries and class libraries are mainly focused on reuse of code [Mat99] or, to a higher level, the reuse of already established structures and infrastructures for building new programs.

Since design is the main intellectual content of software and it is more difficult to create and re-create than programming code, how to reuse design became the main problem. This is partly achieved by packaging classes, which integrate data and operations into class libraries. The class libraries were further structured using inheritance to facilitate specialization of existing library classes; it delivered software reuse beyond traditional procedural libraries. It would be more beneficial in economical terms.

However, there are still problems with reusing the class libraries, because they only allow for reuse of relatively small pieces of code and they do not deliver enough software reuse in larger scales. The striving to increase the degree of reuse and the desire to reuse higher-level designs has resulted in *object-oriented frameworks.*

There are multiple definitions for what is a framework. In [Joh97], the following two definitions are made: "*a framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact*" and "*a framework is the skeleton of an application that can be customized by an application developer*". The former define frameworks from their structure. It indicates that a framework does not have to address one application domain entirely but that it is possible to develop frameworks for smaller domains, thereby opening up for composition of frameworks. The wording "*set of abstract classes*" implies that one way of extension of a framework has to be done through inheritance. The latter definition focuses on the framework's purpose.

In [JF88, FS97], another definition is given: "*A framework is a reusable, "semi-complete" application that can be specialized to produce custom applications.*" In contrast to earlier OO reuse techniques based on class libraries, frameworks are targeted for particular business units and application domains [FS97].

From these different definitions, we can conclude that a framework consists of a set of classes whose instances collaborate (embodies), is intended to be extended, and does not have to address a complete application domain (a family of related problems) [Mat99]. In addition, frameworks are expressed in a programming language, thereby providing reuse of both code and design.

There is confusion about whether frameworks are large-scale patterns, or whether they are just another kind of component.

Compared to components, frameworks provide a reusable *context* for components; however, frameworks are more customizable, abstract and flexible than most components. It has more complex interfaces due to the generality of their intended purpose. At the same time, components represent code reuse; frameworks are a form of design reuse.

Compared to design patterns, frameworks represent a kind of higher-level pattern. A single framework usually contains many patterns. Frameworks are a program; however, patterns are more abstract than frameworks [PS97]. A pattern describes a problem to be solved, a solution, and the context in which that solution works. It names a technique and describes its cost and benefits. Patterns are illustrated by programs.

3.2 Features of Frameworks

The following example from [Mat99] clearly explains the concept of a framework. The single class framework is an abstract class **File** with **read**, **write** and **size** operations, as shown in Excerpt 3-1.

```
File::copy()(File aFile)
{
    char buffer[BlockSize];
    for (int i = 1; i++; i <= this->size() )
    {
        this -> read(buffer);
        aFile -> write(buffer);
    }
}
```

Excerpt 3 - 1: A single class framework

In this example, if we want to extend the framework with subclasses, for example, **UnixFile** and **InterNetFile**. Because reading and writing files can be different for different file types in **UnixFile** and **InterNetFile** subclasses, the **File** class will not implement **read** and **write** operations. However, the **read** and **write** operations can be used to implement other operations, for example **copy**, by which a file can copy itself to another file. That means the **read**, **write** and **size** operations are defined in class **File**, its subclass can not use these operations directly, but it can use the **copy** operation. The copying of a **UnixFile** to an **InterNetFile** will invoke the **read** and **size** operations on the **UnixFile** and the **write** operation on the **InterNetFile**. The point is that an operation in a superclass, the `copy` operation, can call operations in subclasses, i.e. the superclass is controlling the execution flow. Operations like the `copy` operation is often called *template* methods and operations like `read` and `write` are called *hook* methods [FHLS97].

The use of template and hook methods are a distinguishing feature of a framework compared to a traditional class library [Mat99]. Normally, when a class library is used by an application, it is typically *passive*; the way is from the application to the library only. For example, they perform their processing by borrowing threads of control from self-directed application objects. In contrast, frameworks are *active*, the way can be in the opposite direction, i.e. bi-directional. For example, they control the flow of control within an application via event dispatching patterns like Reactor and Observer. This is one of framework's features: Inversion of control, which characterizes the run-time architecture of a framework. Inversion of control allows the framework to determine which set of application-specific methods to invoke in response to external events.

In the above example, we can see that hooks provide an alternative and supplementary view to the design of the framework. If we scale up the above example to a larger framework which consists of more (abstract) classes and provides more templates and hook operations; then, this framework will provide a large number of extension points for applications (i.e. *instances* of the framework) to extend. The framework approach enhances its extensibility by providing explicit hook methods that allow applications to extend its stable interface [FS97].

The appearance of the framework approach arose from the need of higher-level software reuse. There is no doubt that the reusability benefits for an object-oriented framework emanate from its extensibility and the inversion of control. Moreover, it needs to accumulate the domain knowledge and prior effort of experienced developers to generate a framework. This increases the framework's reusability because for some common solution, new developers can avoid re-creating and revalidating and avoid recurring application requirements and software design challenges.

As an essential characteristic of object-oriented methodology, high modularity is also a feature of framework technology. Framework encapsulates volatile implementation details behind stable interfaces; then the impact of design and implementation changes will be localized, which makes it easier to understand and maintain the existing software.

The foundation of framework technology is the experience arised from large number of applications in similar domains. The purpose of frameworks is to increase software reuse at a higher level. So, it must have high flexibility to support the developers, i.e. at the same time, it must leave enough freedom for users to customize to achieve their specific requirements. This is achieved by what is commonly known as *design encapsulation*, where the flexibility points (i.e. the framework *hot spots*) are hidden from the broader structural perspective and allowed to be used as *black boxes*.

In conclusion, the primary features of the framework approach to software development lie in the extensibility, inversion of control [FS97], reusability, modularity and customizability it provides to developers.

3.3 Framework Classification

A framework provides functionality with certain aspects that are fixed and cannot be changed, and other aspects that are variable and intended to be changed [Sch96]. The former is called "frozen spots" and the latter is called "hot spots". Each hot spot incorporates a single, variable aspect of the application domain. Different programs may be created from a given framework, depending on how its hot spots are filled out.

According to the way in which an application is created, frameworks are classified as white-box framework and black-box framework. A *white-box framework* provides incomplete classes with regard to hot spots, as shown in Figure 3-1.

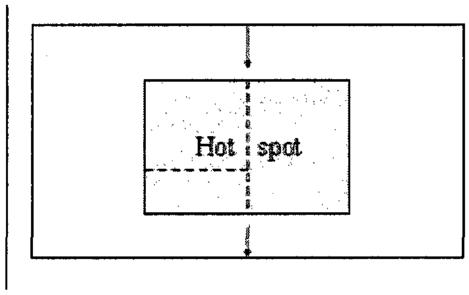


Figure 3 - 1: The way to generate hot spot in white-box framework

It is reused by completing its classes and makes heavily use of inheritance and dynamic binding which are available in object-oriented languages. The extensibility in a white-box framework is achieved by first inheriting from framework super-classes and secondly overriding the pre-defined hook methods. White-box frameworks are sometimes called architecture-driven or inheritance-focused frameworks [BMA97]. An arguably fatal disadvantage of white-box framework is that developers must understand details of how framework works.

For instance, a developer has to derive application-specific classes from abstract classes by inheritance or redefining their methods.

A *black-box framework* contains the complete code and a set of alternative classes for each hot spot, as shown in Figure 3-2.

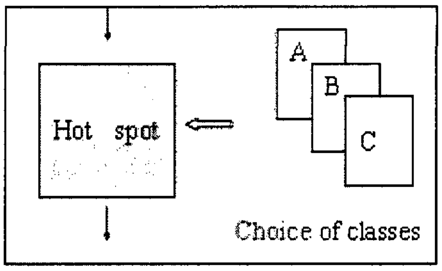


Figure 3 - 2: The way to generate hot spot in black-box framework

It is reuse by composition. The extensibility of a black-box framework is achieved by first defining components that conform to a particular interface and secondly integrating these components (objects) into the framework. Black-box frameworks are sometimes referred to as data-driven or composition-focused frameworks [BMA97]. It is easy for users because an application developer only needs to understand client interface to make the choice.

From the point of view of scope, there are system infrastructure frameworks, middleware integration frameworks and enterprise application frameworks. The first one includes operating systems, communication frameworks, frameworks for user interfaces and language processing tools. Normally, they are primarily used internally within a software organization and are not sold to customers directly. The second one includes ORB frameworks, message-oriented middleware and transactional databases. They are commonly used to integrate distributed applications and components and represent a thriving market and are rapidly become commodities. The last addresses broad application domains which could be telecommunication domain, avionics domain, manufacturing domain and financial engineering domain. They are the cornerstone of enterprise business activities and are more expensive to develop and/or purchase. Enterprise frameworks can provide a substantial return on investment since they support the development of end-user applications and products directly.

3.4 Framework Methodology and GIPC

3.4.1 Justification for the use of the Framework Approach

The GIPC component design has to be considered on:

1 **Frequent requirement change:** In Section 1.10.3, we introduced that the evolution of Lucid is constant and fast. This requests a very flexible programming system to handle frequent changes. The premature death of the GLU system also warns us that the flexibility of a new intensional programming system should be one of the first considerations.

2 **High extensibility:** In Section 1.10.3, we talk about the relation between the core GIPL and new version SIPLs. How to extend the system from the basic to the specific version should be another consideration.

3 **Hybrid programming between IPL and mainstream programming languages:** The mainstream programming languages could be changed based on users' reality. Allowing the system to allow the programmer to use a variety of procedural languages (i.e. a "multi-hybrid" programming environment) adds to the necessity of flexibility and generality of the

programming system implementation. The reasons why to provide hybrid programming include the following 3 aspects:

- **Increased granularity.** It is well-known that functional languages exhibit an inherent parallel evaluation semantics at the operation level. However, the distributed/parallel evaluation of such programs is generally inefficient because of the fine granularity of the operands. A solution to this problem is to increase the granularity of the data elements manipulated by the language and/or permit the language to call procedural functions with a higher granularity, i.e. add granular user-defined operations to the language's underlying arithmetic. This has the effect of reducing the ratio of demand propagation overhead vs. computation time.
- **User acceptance.** Intensional programming languages are far from being mainstream. There is a very strong tendency in Computer Science to use mainstream languages, and to rely on technologies that allow for the reuse of existing code written in these mainstream languages. No matter the magnitude of the advantages of IP, it would not be realistic to expect that the Computer Science community will embrace IP very easily. In this context, we have to find strategies to make IP languages more mainstream. Hybrid versions of IP and mainstream languages, especially object-oriented languages, aims at reaching for this goal.

- o **Parallelization of legacy code.** In many cases, existing programs can be parallelized by using IPLs as an inherently parallel skeleton language that calls legacy functions with minimal changes in the original code. The GLU system has proven that Lucid can be used to parallelize legacy programs effectively with minimal changes to the original code by using Lucid a skeleton language [JDA97].

4 **High modularity:** The individual components designed in the GIPSY request high modularity on the GIPC. Note that most of the other components of the GIPSY are also themselves to be designed as frameworks, so that the GIPSY is in fact to be an aggregation of framework.

5 **Design reuse:** We hope this design can become a typical skeleton and can be reused, in a broader perspective, for the design of compilers for other similar families of programming languages.

Based on all these considerations, the modularity, reusability and extensibility aspects of the framework approach [FS97] make it an obvious candidate for the development of the GIPC.

3.4.2 Automatic Generation of Compiler Components

Because of the wide range of application domains applicable to the GIPSY, we also take for granted that potential users will not be professional computer scientists, and even less compiler designers. From this point of view, we should design the framework in a way that enables the *automated* generation of new compiler components while users develop new languages. That means we will achieve hot spots automatic generation for a new application, which can increase the power of the GIPSY as well as the difficulty of developing such framework.

3.5 Summary

In this chapter, we introduced the concept, features and advantages of the framework approach to software development, and illustrate the rationale of using the framework approach for the GIPC component development. Now, the problem is how to develop the GIPC and to achieve all requirements. In the next chapter, we will introduce details of the GIPC framework design.

Chapter 4 : GIPC Framework Design

In Chapters 2 and 3, we introduced the conceptual design of the GIPSY system and the framework methodology of software development. The main task for us is to design an infrastructure for the compiler construction system embedded in the GIPSY, namely the GIPC, with maximal flexibility with regard to adding new languages to the programming system. Due to the flexibility and variability of a framework, framework design is inherently more complex than application design [GHJV94]. In this chapter, we introduce details on our GIPC framework design.

4.1 GIPC Framework Overview

We described the architecture of the GIPSY in Chapter 2. There are three main components in the GIPSY: RIPE, GIPC and GEE. RIPE is a visual run-time programming environment, at present, the "editor" component in RIPE provides an environment for users to input the source files which will be fed to GIPC. GEE is a general eduction engine, whose input is the output of GIPC, an abstract syntax tree with attributes. In this chapter, we focus on GIPC component. Figure

4-1 shows the overview of the GIPC framework.

GIPC

Pre-processor

Dispatcher

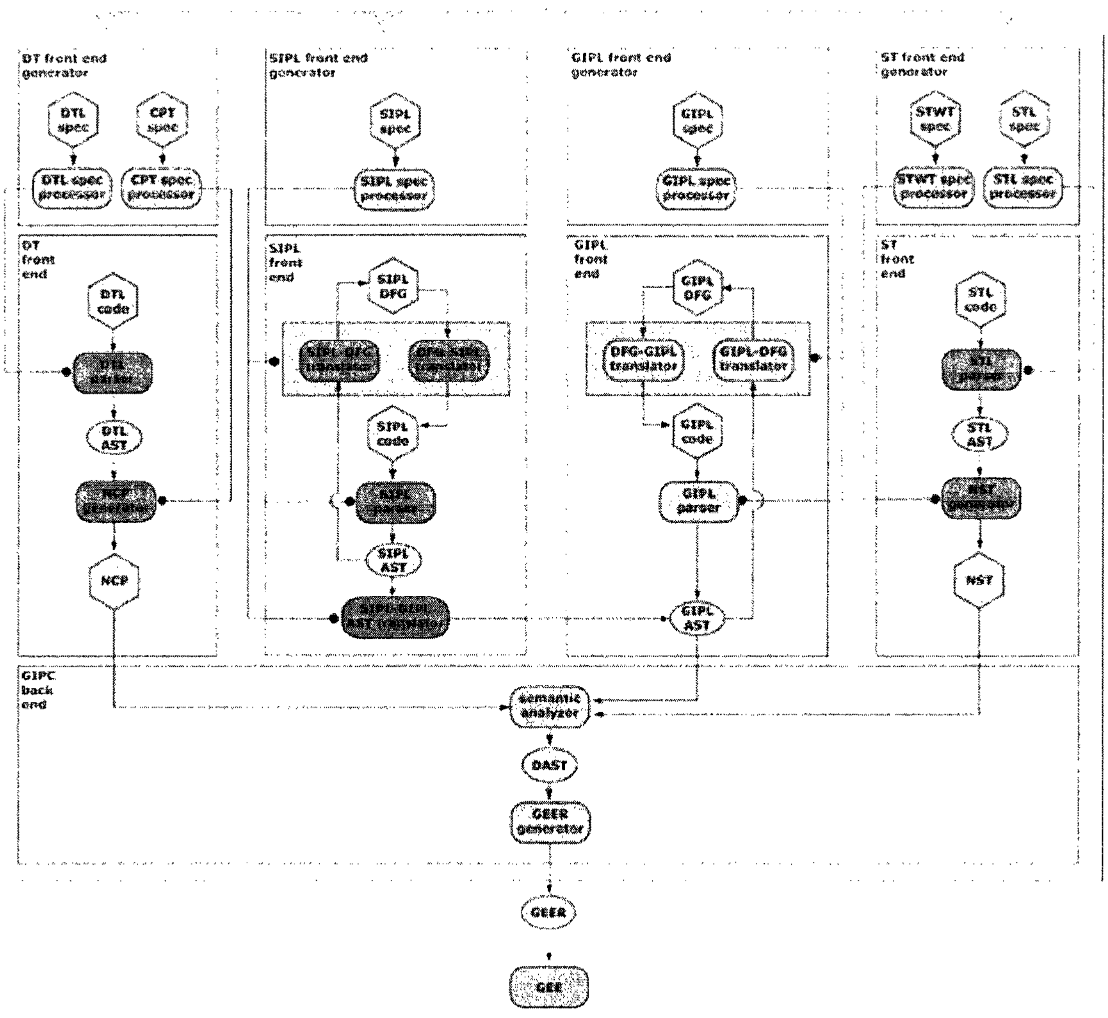


Figure 4 - 1: Overview of the GIPC framework

The design of this framework is done at two different levels of abstraction:

Implementation level: deals with the compilation of a particular flavor of GIPSY programs into DPR, STs and CPs. The implementation level framework design was already shown in Figure 2-2.

Generation level: deals with the automatic generation of compiler components for the compilation of different flavors of GIPSY programs, taking in input the corresponding language specifications, as shown in Figure 4-2.

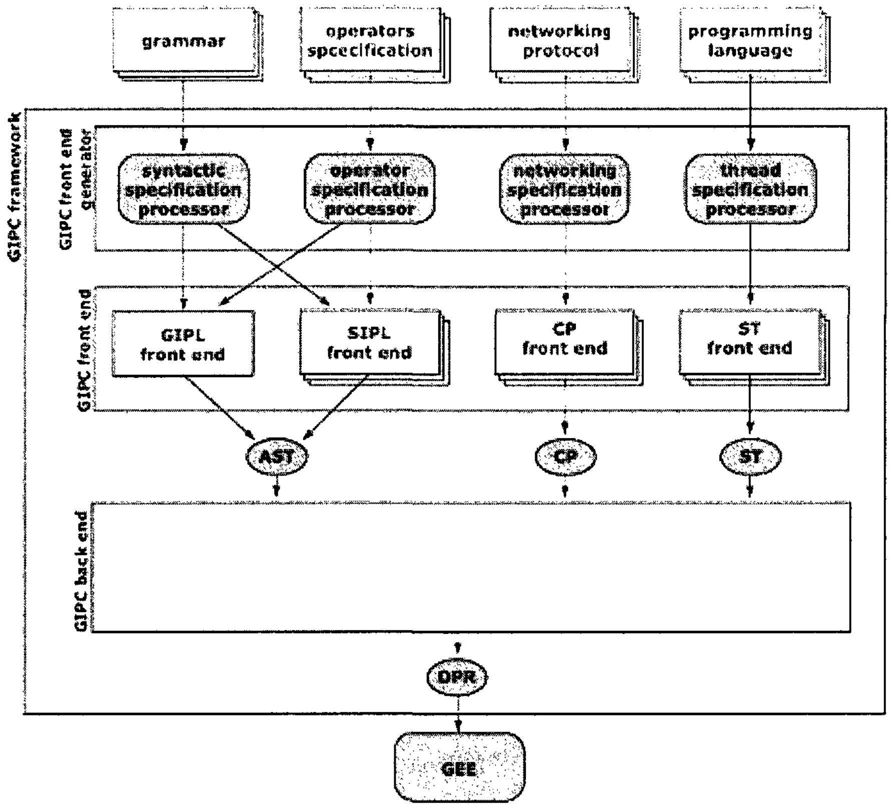


Figure 4 - 2: Generation-level of the GIPC framework

Conceptual description will be given in Sections 4.2 and technical details will be explained in Section 4.3.

4.2 Conceptual View of the GIPC Framework

In this section, we will show the conceptual view of the GIPC. From implementation level:

1. The GIPL component deals with Generic Intensional Programming

Language (GIPL) which is the core language in this system. It is a GIPL compiler with visual programming support. So, except syntax and semantic analysis of GIPL, translators between graphic input and textual input are also required in this component.

2. SIPL components deal with Specific Intensional Programming Languages which are extensions of GIPL. The constant evolution of Lucid language leads to a set of extensions which form different SIPLs. The specific characteristic of Lucid illustrated in Section 1.10.3 let us simplify this part of the design. Because all SIPLs share the same basic semantics with the GIPL and can be translated into GIPL, this component will share the same semantic analyzer as GIPL components. So, the SIPL component will include SIPL parser and a translator from SIPL to GIPL. The GIPL and SIPL components are organized as the middle part in Figure 4-1, which is the declarative part – pure i.e. non-hybrid Lucid – in the GIPC.
3. Sequential Threads (ST) component deals with the code written in procedural languages. The reasons why we consider including other languages are because 1) as we describe in Section 1.11, one weakness of Lucid is lacking input/output, by using other languages, we can give clear input and output; 2) as we describe in Section 3.4, in many application domains of IPL, users would like to use some existing codes. That system can deal with sequential thread will improve the reusability and encourage users to accept this new language and system.

From generation level:

1. Pre-processor layer will dispatch different code segments to different components. For example, if a code segment is written in GIPL, it will be dispatched to the GIPL component. The reason why we consider adding such layer is to support maximal convenience to users. They do not need to input different source codes by different menus instead of giving mixture inputs to the system.
2. Front-end generator layer generates components in front-end layer. It help automatically generates hot spots of the GIPC framework.
3. Front-end layer translates input programs into intermediate representation data structures usable by the back end for further semantic analysis and intermediate code generation.
4. Back-end layer translates intermediate representations into demand propagation resources (DPR) directly usable at run-time by the GEE.

4.3 Technical Details of the GIPC Framework

The procedure of how the GIPC works is: specifications feeding, then hot spots generated automatically, finally concrete system running. In this section, regarding different components, we will introduce based on these 3 steps. The class diagram of whole GIPC design is shown in Figure 4-6.

4.3.1 GIPL component

Figures 4-3 presents detail view on the GIPL component.

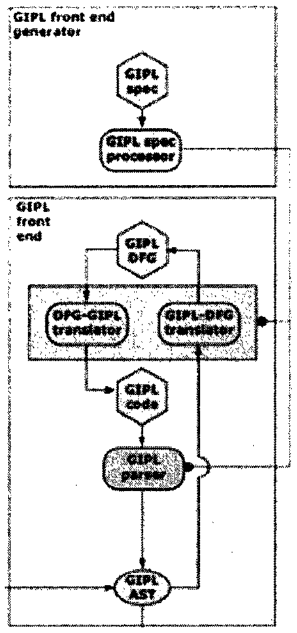


Figure 4 - 3: Detail view on the GIPL

GIPL class will be responsible for compiling the General Intensional Program. **GIPLParser** class parses the GIPL and generates the Abstract Syntax Tree which will be used by **SemanticAnalyzer()**. To support visual programming, GIPL to Data Flow Graph (DFG) analyzer and generator are also included in the GIPL component, even though they are designed in the RIPE component. **DFGAnalyzer** class will translate GIPL code to graph and **DFGCodeGenerator** class will translate graphic source input into GIPL code. Details can be found in [Din04].

In the GIPL front end, the **GIPLParser()** is automatically generated by JavaCC when the GIPL specification is given. The specification is described by grammar expressed in near-BNF format. Examples of GIPL grammar files and the parser generation process can be found in [Ren02]. According to framework concept, **GIPLParser()** is a hot spot. Only when users input the GIPL specification, GIPL front-end generator will automatically generate it.

4.3.2 SIPL component

Figure 4-4 presents a detailed view of the SIPL component.

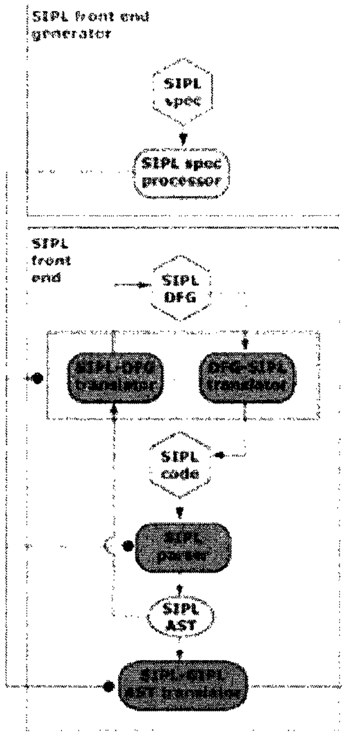


Figure 4 - 4: Detail view on the SIPL

The **SIPL** class will be responsible for compiling the Specific Intensional Program. The **SIPLParser** class parses the SIPL and generates the SIPL Abstract Syntax Tree, the **Translator** class translate SIPL-AST to GIPL-AST which will be used by **SemanticAnalyzer()**. The visual programming part is the same as the GIPL, **DFGAnalyzer** class will translate SIPL code to graph and **DFGCodeGenerator** class will translate graphic source input into SIPL code.

In Figure 4-4, we add operator translation specifications to make sure the SIPL AST – GIPL AST translation can be done automatically. The additional syntactical constructs introduced in each new SIPL are given their GIPL equivalence using a very simple custom specification language which is near-text format; the translator generator will parse the file and generate a SIPL-GIPL AST translator.

The SIPL operator translation rules are used for two different purposes: First, for the generation of an Abstract Syntax Tree (AST) translator that will translate an SIPL AST into a GIPL AST (as the back end is designed only for the processing of GIPL ASTs only). Secondly, the semantic translation rules are used for the generation of a DFG-SIPL translator that will translate the textual version of programs written in this SIPL into a DFG representation, as well as a SIPL-DFG translator that will do the reverse operation. An example of such rule is in the Table 4-1, which defines how to translate SIPL operators into GIPL equivalence.

```
first: R @.D 0

fby: if (#.D==0) then L
     else R@.D(#.D-1);

wvr: L @.D T
        where
            T=if (#.D==0) then U
              else (U@.D(T+1))@.D (#.D-1);
            where
                U=if R then #.D
                  else U @.D (#.D+1);
            end;
         end;

asa: (L @.D T) @.D 0
        where
             T=if (#.D==0) then U
               else (U@.D(T+1))@.D (#.D-1);
             where
                 U=if R then #.D
                   else U @.D (#.D+1);
             end;
         end;
```

Table 4 - 1: A SIPL operator translation rule file

Details of this translator generation can be found in the Chapter 4 of [Wu02].

In the SIPL front end, the **SIPLParser()** is automatically generated by JavaCC when the SIPL specification is given. **GenericTranslator()** is also automatically generated by **TranslatorGenerator()** when the operator translation rules are given. So, **SIPLParser()** and **GenericTranslator()** are hot spots in the GIPC framework.

4.3.3 ST component

ST component deals with sequential threads which are written in different procedural programming languages. Figure 4-5 shows the detail view of ST component.

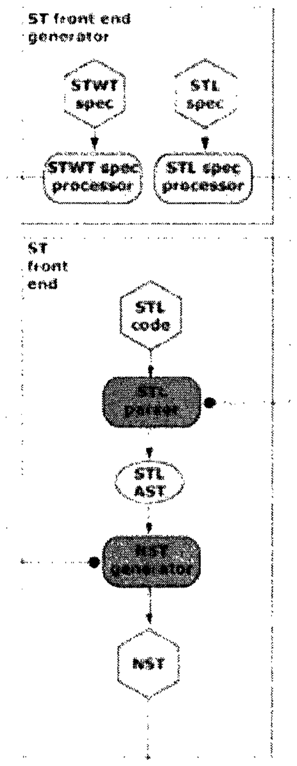


Figure 4 - 5: Detail view on the ST

The **ImperativeCompiler** class provides the most common possible implementation for all imperative compilers respectively, so the underlying concrete compilers only have to override some parts specific to the language they are to compile. The **SequentialThreadParser** will parse the sequential threads, the produced ASTs really contain a single ImperativeNode and are secondary and should be merged into the main when appropriate. The **SequentialThreadGenerator** is an abstract factory for all sequential threads that has to be overridden by a language-specific sequential thread generator, e.g. such as **JavaSequentialThreadGenerator**.

Table 4-2 shows the sequential thread specification.

```
<STs>           ::= <ST><STs>
<ST>            ::= <LANGID><CODESEGMENTs>
<LANGID>        ::= #<CAPLETTER> (<CAPLETTER>)*
<CODESEGMENTs>::=<CODESEGMENT><CODESEGMENTs>
                  | empty
<CODESEGMENT> ::= <LANGDATA> <LANGID>
                  | <LANGDATA> <EOF>
```

Table 4 - 2: ST specification

According to Table 4-2, sequential treads will start with **#language**, then following by specific functions or class. Excerpt 4-1 shows a simple example of sequential threads.

```
#JAVA
Input() {…};
Output() {…};

#C++
c1() {…};
c2() {…};
```

```
#FORTRAN
f1() {…};
```

Excerpt 4 - 1: A simple example of sequential threads

SequentialThreadGenerator() will generate sequential threads, **getSequentialThreads()** will get sequential threads and produce an AST which is an **ImperativeNode**. The compiling process of sequential threads will be a given specific compiler and when the compile is done, the **ImperativeNode** will be replaced by real AST. More details of ST component can be found in the [Mok05].

In the ST component, **SequentialThreadParser()** and **SequentialThreadGenerator()** are hot spots.

4.3.4 CP component

In order to reach for maximal execution flexibility and performance, the communication procedures can be generated according to different networking protocols or middleware technologies. Emil provides a demand migrate framework in [Vas05], which give details of this component.

4.3.5 Class diagram of GIPC design

The class diagram in Figure 4-6 expresses the implementation level of the GIPC design.

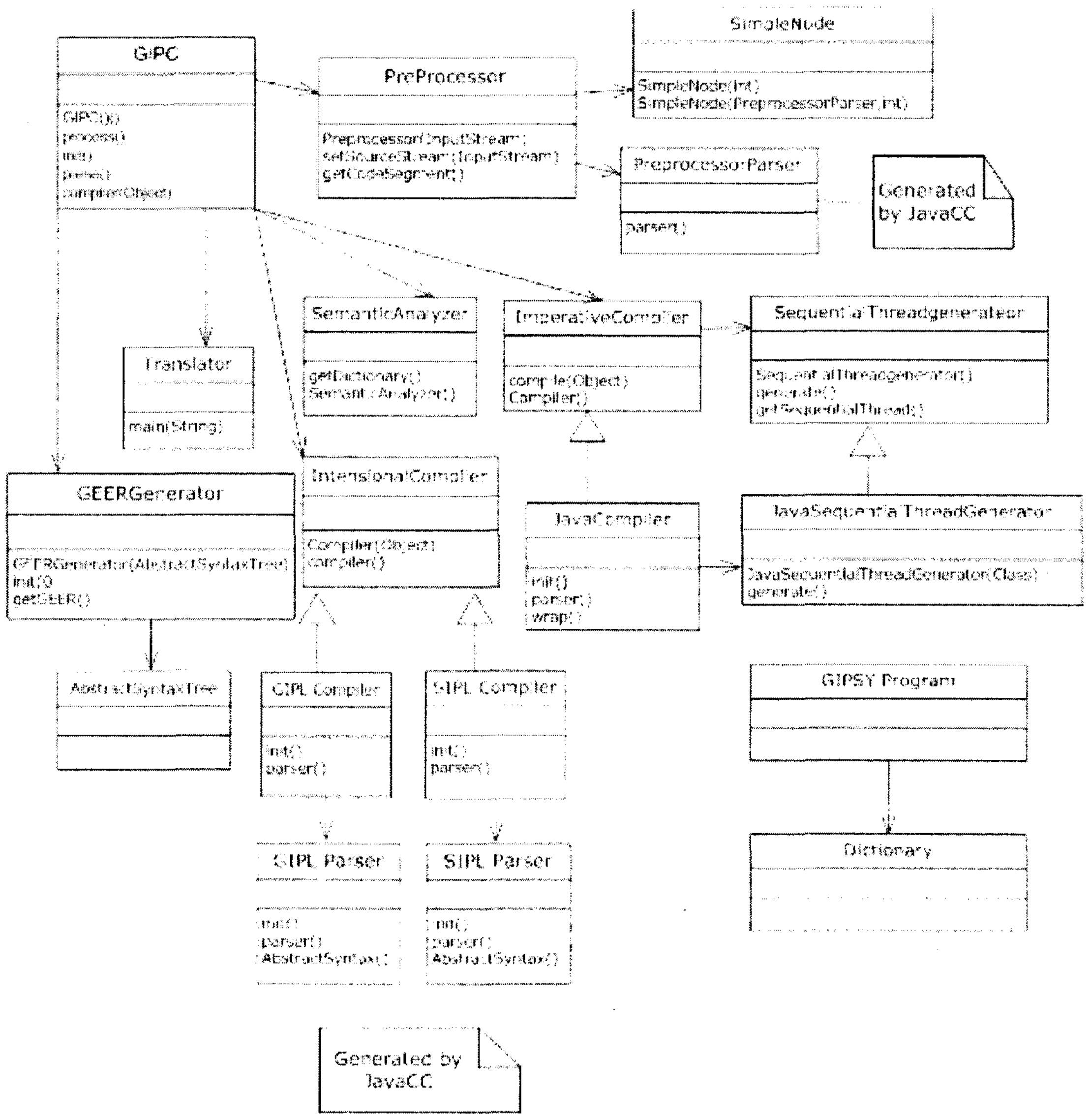


Figure 4 - 6: GIPC class diagram

4.4 A Scenario: Add a New SIPL into GIPC

The sequence diagram showed in Figure 4-7 describes how to add a new SIPL into GIPC framework.

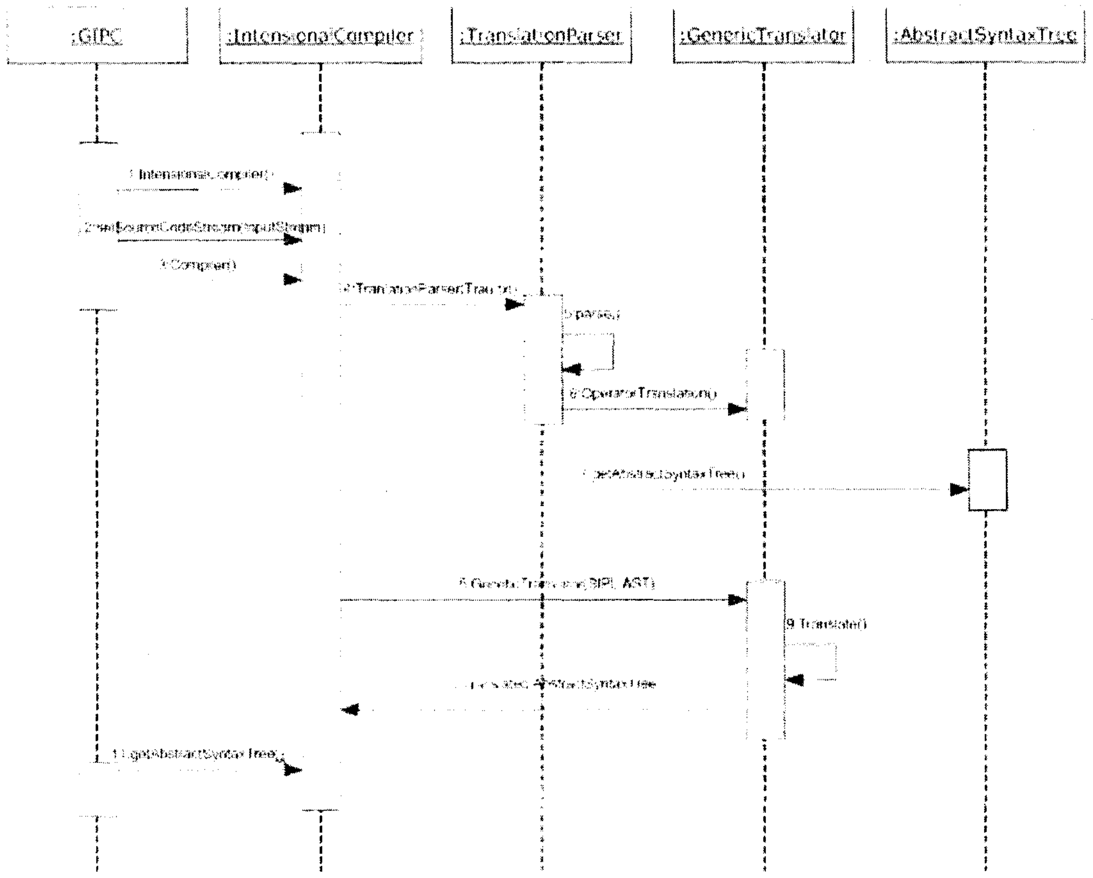


Figure 4 - 7: A sequence diagram of adding a new language in the GIPC

4.5 Contributions of the Framework Design

In traditional frameworks design, developers should leverage all collective knowledge gained from many existing good framework examples to think, design and decide the set of alternatives for each hot spot. Users can write new classes, or choose from a pre-established list of classes to fill out a hot spot for generating a new application.

This GIPC framework design has four major contributions:

1. Introducing the notion of layer into the compiler construction design. For

example, there are 4 layers in GIPC framework: Pre-processor layer, front-end generator layer, front-end layer and back-end layer.

2. Pre-processor layer is introduced in [Mok05], which is like a dispatcher that dispatches different code segments to different components. The introduction of this layer can provide maximal convenience to users. They do not need to input different source codes by different menus instead of giving mixture inputs to the system. The specification of pre-processor can be found in Chapter 6, hybrid language design.
3. Front-end generator layer can automatically generate hot spots. This design greatly enhances the power of the framework, because users only need to know how to input the specifications for the hot spot generators. The framework components are totally black-boxes for the users. The black-box design keeps the back end of the compiler untouched by the addition of new component instances in the front-end. Any eventual change in the back-end will be shielded to the users. It is thus much easier to use for people who are not compiler design initiates, which is our target audience for this system.
4. The automated generation of hot spots provides the required level of flexibility and extensibility for the GIPSY. This approach permits the easy change and addition of compiler components for various IP languages, procedural languages and middleware technologies by hiding the intricacies of implementation from the user as much as possible.

4.6 Limits of the Framework Design

The limitation of this framework is about the difficulty on type adding in the GIPC framework. For example, for each new language adding, we must map its types with the **GIPSYtypes**, and possibly add new **GIPSYtypes** when new languages are added. Details about type mapping discussion are in the Section 6.3.6.

The second important limitation is the fact that all SIPLs must be "translatable" to the GIPL, and that if the GIPL is changing, then all translation rules of all SIPLs have to be rewritten, or a generic translation layer will have to be introduced.

4.7 Summary

The framework presented here is designed for withstanding the evolution and generality of Intensional Programming and its widely different domains of application. In this chapter, we describe the GIPC framework design from 2 different levels as well as technical details. We also define the content for each specification as the input for this framework. Finally, we discuss the contributions and limits for this framework design.

In next chapter, we will compare several typical compiler construction systems to illustrate the contributions of this framework design. Additional explanation of of how this framework works could be found in Chapter 7.

Chapter 5 : Related Work on Compiler Construction System Design

5.1 Introduction

Language design and implementation are one of the main challenges in computer science. The development of the first compiler in the late fifties without adequate tools was a very complicated and time consuming task. Later on, some formal methods were developed which made the implementation of programming languages easier. At the same time, new domain-specific languages appear frequently, so the language design process should be supported by modularity and abstraction in a manner that allows incremental changes as easily as possible. This introduced the field of compiler construction tools design. Tools such as LEX and YACC are examples. They contributed to the automation of the process of implementing programming languages. The possibility of the automatic generation of interpreters or compilers for the formally defined language enables the language developer a quick and simple evaluation of his ideas in the process of language definition.

Compiler construction systems are infrastructure systems. The result of these systems is programming development environment. There are some related works to be done in this topic. In this chapter, we discuss seven different compiler construction systems and attempt to evaluate them from the point of view of their *architecture, advantages and shortcomings compared to the compiler generation* framework that we described in the last chapter. According to the characteristics of such systems, we use a set of custom evaluation criteria to compare these typical existing systems to illustrate the contribution of the GIPC framework design compared to these existing systems.

5.2 The Criteria for Evaluating Compiler Construction Systems

In [Slo95], the author compares the compiler generated by compiler construction system and hand-written compilers. Previous work evaluating compiler construction systems has largely concentrated on subtasks of the generation problem, for example, lexical analyzer and parser generator. In this section, we provide a set of evaluation criteria for whole compiler construction systems. Same ideas have already been presented in the paper, "Survey, Evaluation and Tendency of Compiler Construction Systems", which was accepted by IASTED International Conference on ACIT-Software Engineering (ACIT-SE 2005). Unfortunately, we could not eventually publish this paper, but its results are presented here. The criteria include:

1. Flexibility

Flexibility is the ability for a system to adapt to changes in its environment or in its requirements. That new language evolutions appear frequently will cause the requirements of a compiler to frequently change, thus requiring to have tools that help to change compilers easily when the syntax or semantics of the language is evolving. In our particular case, GLU was a great achievement, as it effectively demonstrated that the dataflow programming paradigm could be efficiently implemented with a Lucid-C hybrid programming language and executed on distributed and/or parallel computing platforms. However, because GLU's implementation lacked flexibility, it could not adapt to the subsequent evolutions of Lucid. It quickly went obsolete and even unsuited for further research because of this lack of flexibility and was put on the shelf of heroic and defunct "proof-of-concept" rather than to continue on with the evolution of its field of research.

2. Extensibility

Extensibility is the ability of a system design to allow extensions points where necessary. Such extension points are designed in an abstract manner that allow for new design elements to replace old ones, or for new instances of extensions to be added, all with an increase on system capacities in mind. Clearly, extensibility is the cornerstone feature of the notion of framework design, as framework hot spots represent extension points.

In our particular case, evident extension points include the possibility of addition of new SIPLs, or new procedural languages, so that our system becomes a programming environment that is in fact suitable for a family of multi-hybrid intensional programming languages, rather than being suitable to only one flavor of such languages, such as GLU was. From this perspective, it is worth noting that GLU in fact allowed C and FORTRAN procedures to be used. However, its design did not provide extensibility provisions to easily add new procedural languages to be used.

3. Hybrid programming support

Hybrid programming means two or even more kinds of different languages can be used in the same program. Two of the main programming language paradigms are imperative languages, for example Java; and declarative languages, such as Lucid. Language hybrids within the same paradigm are somewhat easy to merge, compared to merging to languages that belong to different paradigms, simply because many languages belonging to the same paradigm will share a similar semantics and execution engine implementation. Different language paradigms will tend to rely on different execution paradigms, underlying different semantics, and thus will be harder to "cross-breed".

However, it is not necessary that every compiler construction system should be suite to consider hybrid programming; however, there is no doubt that for some application domain, hybrid programming looks extremely important, such as with the particular case we are investigating, where an inherently multi-dimensional declarative language executed in demand/context-driven mode is coupled with a procedural language executed with the standard von Neumann model.

4. Visual programming support

Visual programming means the system supports visual input instead of textual input. Visual programming is currently a very active research area. Current compiler generators rely on linear textual specifications, which are for designer less suitable than visual presentation. The main reasons for this are much easier implementation of the tool and easier processing of textual information. Nowadays, developing software with integrating development environments and powerful graphics hardware is much more user friendly. We even enlarge the visual programming to animation of an interpreter's inner workings, i.e. visual run-time animation of the program's execution. These visualizations can help explain the inner workings of programs and are a useful tool for debugging and teaching. In cases where the programming language at hand is cryptic and non-standard (such as Lucid is undoubtedly), and a more standard graphical representation of programs can be inferred from the program (such as representing Lucid programs as standard dataflow graphs), the benefits of visual programming are magnified and even in some cases necessary.

5. Powerful integrated environment

Integrated environment means all tools for a system are packaged together and are controlled by a clear interface. At the beginning, compiler construction tools were normally single-system programming environments. This approach has a high degree of integration, because the tools share the same data structures. On the other hand, it is very difficult to add new tools, especially those designed outside the environment. Then, the approach is extended to package tools as a set of independent tools, which can be called a *federated environment.* Such environments are designed so that it is relatively easy to add new tools by clearly and purposely defining each tool's interface in a manner that each tool can be used by other existing and upcoming tools. However, all following shortcomings limit the power of this integration method: it has risk of poor performance, consistent and integrated environment do not exist, tools may have different interfaces and incompatible interfaces often raise the need for brokers, or translators that inevitably clutters the design and slows down the implementation. So we need a powerful integration mechanism which offers many of the advantages of both the single system and the federated environment. The GIPSY, being designed as a multi-framework, each of which being closely matched with the others, totally adheres with this important concept.

6. Usability

In short, usability means it is easy for users to understand the compiler construction system and to use it. Users will adopt a compiler construction system because it can shorten the process to get a new compiler. If it will take a long time and much energy to learn the system, it will discourage users to accept the system. A friendly approach is to provide a black box approach to the users which, for instance, is implemented through the framework methodology.

7. Automated component generation

Automated generation of compiler components is closely connected to the framework that we have described in the previous chapter. During the framework technique discussion in Chapter 3, we introduced hot spots and frozen spots. This is where the GIPC framework demonstrates one of its most original contributions: it is designed as a framework, providing hot spots to be filled in by the user when creating new programming languages; simultaneously, it also relies on automated compiler construction tools to generate these hot spots in the framework, and thus instantiate new compilers automatically from the languages' specification. To our knowledge, this is a novel way of design for compiler construction tools.

5.3 Related Work on Compiler Construction Systems

Since the 1980s, an increasing number of compiler construction systems appeared. From the CENTAUR system in the early period to today's JastAdd system, each has different emphasis. Here we would like to discuss some typical systems.

5.3.1 CENTAUR

The CENTAUR system [$\mathrm{BCD}_{\mathrm{etc}}$88], which appeared in the late of 1980s, is a generic single interactive environment. It produces a language specific environment by giving formal specifications of a particular programming language - including syntax and semantics. The resulting environment includes a structure editor, an interpreter and other tools. For system itself, CENTAUR is made of three parts: a database component which provides standardized representation and access to formal objects and their persistent storage; a logical engine that is used to execute formal specifications; and an interface. The system is essentially written in Lisp.

CENTAUR system uses formal specification of syntax and semantics to describe a programming language. Then, the engine is designed to execute the formal specification. In the specification level, the specifications of concrete and abstract syntax, together with their relationship, are presently written in METAL [KLMM83], a formalism developed for the MENTOR system [DLM84]. Pretty-printing of abstract trees is defined in the PPML formalism [MC86]. Prolog is used for the compilation of semantic specifications.

The CENTAUR system experimented automatic compilation of syntax and semantic specifications to set up programming language compiler environment. However, in the CENTAUR system, a lot of different formalisms are used for language definition and it is hard for users to learn all of these formalisms. For example, The METAL compiler has been used on large languages (Pascal, Ada, C); however, it is somewhat difficult to use.

Moreover, the analysis of formal definitions on syntax and semantics still needs a lot of work to do. For some language, it might be impossible to give formal semantics specification in certain formalism. CENTAUR system does not support reuse the language specification, either incremental programming development.

In [BCD_{etc}88], authors also indicate that “it is not very easy to control the use of space in CENTAUR and much effort in this direction is necessary”.

5.3.2 FNC-2

FNC-2 system [JP97] started in 1986, and a first running prototype is available since early 1989. It is an attribute grammar processing system which consists of several independent tools (federated environment) such as asx (an attributed abstract syntax compiler), fnc2 (the OLGA compiler, OLGA is the input language used by FNC-2 to describe attribute grammars), ppat (a pretty printer for attribute trees), SYNTAX (parser generator), XVISU (dependency graph visualisation).

FNC-2 is developed at the same organization, INRIA, as the CENTAUR system and gets a step further than the CENTAUR system. Its most important features are: efficient exhaustive and incremental visit-sequence-based evaluation of strongly (absolutely) non-circular attribute grammars [KW94]; extensive space optimizations; a specially-designed AG-description language, with provisions for true modularity; portability and versatility of the generated evaluators; complete environment for application development.

The architecture of FNC-2 system is composed of three parts [JPJ_{etc}90], linked through interfaces: the OLGA front-end, the evaluator generator that is the "engine" of the system, in which all the fundamental knowledge about attributes evaluation is concentrated; and the translators.

FNC-2 system tries the best on efficiency, expressive power, ease of use and versatility. However, as the system expressed, OLGA is a big language, and its analysis and implementation are hard tasks, even with AGs. Not all of the language is implemented; either the front-end or the translators reject some valid programs. The most notable omissions are full polymorphism, parameterized modules and exceptions. In FNC-2 system, the graphic input is not permitted and it does not support the animation of compiler inner working.

5.3.3 Eli

Eli system [GHL_{etc}92], being started in the late of 1980s, is a complete and flexible compiler construction system. It is also a federated environment of several tools such as: LIDO (computations in trees), PTG (Pattern-based Text

Generator), Maptool (mapping concrete and abstract syntaxes), etc.

Eli is a collection of off-the-shelf tools controlled by an expert system whose problem domain is the management of complex user requests [WH88]. It generates a compiler from specifications of the structures of the four objects postulated by this model (source program text, source program tree, target program tree, and machine instruction set) and the relationships between them. Effectively, the designer defines a particular instance of the general compilation problem by providing these specifications. Some of the tools check the specifications for consistency, some extract information relevant to the specific sub-problems, and others generate code to solve those sub-problems. Finally, the generated modules are combined with standard modules from Eli's library to obtain a complete compiler that solves the problem specified by the designer.

Specifications in Eli can be reusable since it is possible to define the attribution module which can be reused in a variety of applications. Eli is used to create compilers for small, special-purpose languages, standard programming languages and extensions to existing languages. However, it does not provide the language design in a visual manner and it is still aimed at further simplifying the use of Eli itself.

5.3.4 LISA

LISA [MLAZ00] system, introduced in 2000, is a generic interactive environment for programming language development, which supports for incremental language development, for language design in a visual manner, for animation of compiler/interpreter inner workings, for high portability of the system and the generated environment. It is a set of related tools such as scanner generators, parser generators, compiler generators, graphic tools, editor and conversion tools, which are integrated by well designed interfaces.

Before LISA system's appearance, there was no available compiler/interpreter generator tools support incremental language development, so the language designer had to design new languages from scratch or by scavenging old specifications. To avoid this weakness; inheritance, a characteristic feature of object-oriented programming, is applied in Lisa by multiple attribute grammars. In attribute grammars, ordinary attribute notation has deficiencies which become apparent in specifications for real programming languages; on some worse situation, small modifications of some parts in the specifications will have widespread effects on the other parts of specifications. Such specifications are not modular, extensible and reusable. Now with multiple attribute grammar inheritance, the lexical, syntax and semantic parts of programming language specification can be extended. It is a structural organization of attribute grammars where the attribute grammar inherits the specifications from ancestor attribute grammars, may add new specifications, and may override some specifications from ancestor specifications. In case when languages have similar semantics and a very different syntax, templates are introduced.

The system LISA has many improvements compared to similar systems. In LISA system, the language design process could be supported by modularity and abstraction in a manner that allows incremental changes as easily as possible. Moreover, in the compiler/interpreter generator system LISA 2.0 the programming language can be specified in visual manner with finite state automata, syntax diagrams and semantic diagrams which are then internally transformed to textual specifications. Finally, users of the generated compiler/interpreter also have the possibility to visually observe the work of lexical, syntax and semantic analyzers by watching the animation of finite state automata, parse and semantic tree.

However, as indicated in LISA system, as an input, formal language specification is written in the domain specific specification language which supports multiple attribute grammar inheritance and templates. The architecture of LISA system does not independent on parser layer, users have to learn the specific specification language to write the input.

5.3.5 Polyglot

Polyglot project [NCM03], first version appeared in 2003, is an extensible compiler framework that supports the easy creation of compilers for languages similar to Java, while avoiding code duplication. There are some Polyglot-based compiler projects [Ploy03], for example, Jif [Mye99], which extends Java with security types that regulate information flow; PolyJ [MBL97], which adds bounded parametric polymorphism to Java; and JMatch [LM03], which extends Java with pattern matching and iteration features.

In Polyglot, the original unmodified language is referred as the *base language*; and the modified language is called a language *extension.* The Polyglot framework implements an extensible compiler for the base language Java 1.4. This framework, also written in Java, is by default simply a semantic checker for

Java. However, a programmer implementing a language extension may extend the framework to define any necessary changes to the compilation process, including the abstract syntax tree (AST) and semantic analysis.

[LM03]A Polyglot extension is a source-to-source compiler that accepts a program written in a language extension and translates it to Java source code. It also may invoke a Java compiler such as Javac to convert its output to byte-code. The *first step in compilation is parsing input source code to produce an* AST. Polyglot includes an extensible parser generator, PG, allows the implementer to define the syntax of the language extension as a set of changes to the base grammar for Java. The core of the compilation process is a series of compilation passes applied to the abstract syntax tree. Both semantic analysis and translation to Java may comprise several such passes. The *pass scheduler* selects passes to run over the AST of a single source file, in an order defined by the extension, ensuring that dependencies between source files are not violated. Each compilation pass, if successful, rewrites the AST, producing a new AST that is the input to the next pass. Some analysis passes (e.g., type checking) may halt compilation and report errors instead of rewriting the AST. A language extension may modify the base language pass schedule by adding, replacing, reordering, or removing compiler passes.

Polyglot adopts extended visitor methodology that supports extension of both compiler passes and AST nodes, including mixin extension. The methodology uses abstract factories, delegation, and proxies [GHJV94] to permit greater extensibility and code reuse than in previous extensible compiler designs.

Compared with GIPSY, both concentrate on family programming languages and consider the compiler framework design. However Polyglot does not support visual programming, hybrid programming and repeat passing the AST algorithm will inevitably affect the efficiency of system itself.

5.3.6 JastAdd

JastAdd [HM03] is an aspect-oriented compiler construction system. It is a Java-based system and is centered around an object-oriented representation of the abstract syntax tree in which reference variables can be used to link together different parts of the tree. JastAdd supports the combination of declarative techniques and imperative techniques in implementing the compiler.

In JastAdd, imperative code is written in aspect-oriented Java code modules. For declarative code, JastAdd supports Reference Attributed Grammars (RAGs) [Hed00]. This is an extension to attribute grammars that allows attributes to be references to abstract syntax tree nodes, and attributes can be accessed remotely via such references. The important extension in RAGs (as compared to traditional attribute grammars) is the support for reference attributes. The value of such an attribute is a reference to an object.

The architecture of JastAdd system includes 2 layers and 2 main modules. Similar to GIPSY, JastAdd system is built on top of the LL parser generator JavaCC that is used to parse the program and to build the abstract syntax tree. On this preparation layer, the abstract grammar is independent of the underlying parsing system. The parser is simply a front end whose responsibility it is to produce ASTs that follow the abstract grammar specification. The two main modules are Jadd and Jrag. Imperative behavior is added in Jadd modules that contain methods and fields. Declarative behavior is added in Jrag modules that contain equations and attributes. Behavior can be added to the generated classes in separate aspect-oriented modules.

For each aspect, the appropriate fields and methods for the AST classes are written in a separate file, a Jadd module. The JastAdd system is a class weaver: it reads all the Jadd modules and weaves the fields and methods into the appropriate classes during the generation of the AST classes.

Jrag modules are aspect-oriented in a similar way as Jadd modules: they add attributes and equations to AST classes analogously to how Jadd modules add fields and methods. The JastAdd system translates the Jrag modules to Java and combines them into a Jadd module before weaving.

When building the AST, information about the semantic values of tokens needs to be included. To support this, JastAdd generates a set-method as well as a get-method for each token class. For example, for the token class BoolDecl, a method void setID(String s) is generated. This method can be called as an action during parsing in order to transmit the semantic value to the AST.

By using aspect-oriented programming methodology [MJW99], JastAdd system avoids serious limitations in Visitors in which modularization is only supported of methods and not of fields, and in which type checking of the method arguments and return values are not supported. Another important strength of the JastAdd system is the ease with which imperative Jadd aspects and declarative Jrag aspects can be combined. A compiler can be divided into many small sub-problems and each can be solved declaratively or imperatively depending on which paradigm is most suitable.

However, in JastAdd, duplicating code may need to be written for each pass to support new nodes. Regarding input, system only considers texture input instead of also supporting visual programming.

5.3.7 GLU

GLU (Granular Lucid) system [JD96, JDA97], developed at Stanford Research Institute (SRI) in Menlo Park, was the first intensional programming system that enabled the compilation of Indexical Lucid programs, together with the use of sequential threads written in C, i.e. a Lucid-C hybrid language.

Using the dataflow programming paradigm, the trivial implementation is of too fine granularity to be efficient; and in most scientific domains, most programmers would like to reuse their code as much as possible. If using Lucid as a parallel programming language would force programmers to entirely rewrite their existing programs, which will discourage most of them from using it. Based on these reasons, GLU system tries to increase the granularity and to reuse existing code in a Lucid program as much as possible.

So, a GLU program includes two distinct source files: one for the Lucid part and one for the C part. A GLU program not relying on any C functions does not have any C part. This will permit programmers to use their existing C functions. With regard to its architecture, GLU uses a generator-worker parallel processing architecture. The Lucid part defines the implicitly parallel execution flow graph of the program in terms of dependencies between operations on data elements.

The (compiled) Lucid part of the program is executed by the generator following the eductive model of computation. The low-charge ripe C functions are evaluated locally by the generator. The high-charge ripe C functions are evaluated on a remote worker. This will increase the granularity, because in the Lucid program, parallelism is at the basic operation level, for example, * and +, which are extremely fine-grained; on the contrary, in the GLU program, parallelism is at the level of C functions, which opens the door for more acceptable levels of granularity, given the problem of overwhelming communication overhead implied by fine granularity.

The GLU system has proven to be a usable and highly efficient solution for the parallelization of sequential programs. However, the GLU system suffered from a lack of flexibility and adaptability. It could not cope with the latest evolution of Lucid. For example, the GLU system does not enable dimensions and functions as first-class values, which is one of the key principles used in Tensor Lucid [Paq99]. Also, it could not be easily extended to include other procedural languages, for example, Java functions. As we will see later in this thesis, GLU did not allow for objects to be first class values in Lucid, a great limitation that our proposed solution is eliminating, under certain constraints stated later.

5.4 Comparative study

In the above compiler construction systems, the Centaur and FNC-2 systems appeared in the late of 1980s, they are relatively old. The Polyglot and JastAdd systems are all quite new systems based on Java. GLU is in the same application domain as the GIPSY. In this section, we would like to compare all these systems with the GIPSY.

From Section 5.3, we can tell that Centaur, FNC-2 and Eli systems lack flexibility, as well as GLU. The Polyglot and GIPSY systems use the framework technology to achieve maximal flexibility.

LISA was one of the first compiler construction systems to support incremental programming language development. Before it, the CENTAUR, FNC-2 and Eli systems did not support such extensibility. After it, Polyglot and JastAdd systems focus the extension based on Java, whereas GIPSY focuses on Lucid. All of these have high extensibility from this regard, whereas GLU lacks such extensibility.

In these systems, JastAdd permits to combine the imperative and declarative languages; GLU supports hybrid programming between Lucid and C; GIPSY can support any version Lucid programs with Java functions, and it was designed to be easily extended to other sequential threads, for example, C++ functions, Pascal functions, etc; other systems cannot provide such functionality. The hybrid language solution that we discuss later even allows GIPSY to use a hybrid of Lucid and Java objects, which can the easily be extended to other languages using the notion of object.

In these systems, only LISA and GIPSY systems provide visual programming; other systems only support traditional textual input.

Regarding the integration mechanism, the CENTAUR system is a single system; the FNC-2 and Eli systems are federated systems, there is a little communication between tools included in the system; LISA, JastAdd and GLU integrates tools with a nice interface, they provide nice integrated environment; and Polyglot and GIPSY systems use a framework approach to system design. In fact, it can be said that GIPSY uses a *framework federation* integration approach.

There is always a relation between system functionality and usability. It is harder to learn a multi-functional system than a single-task system, the key is how to reduce requests on users and let the system handle problems. Here, CENTAUR has multi formalism for language specification definition, it is not easy for users to learn all formalism; FNC-2 system pay more attention on ease of use; Eli and LISA systems need users to learn the specific language for specification input, it is hard for users; Polyglot and JastAdd systems are based on Java extension; GLU permits users to reuse legacy C code; and one of the goals for the GIPSY system is to provide a friendly interface. It proposes to generate framework hot spots automatically, making it easier for users to develop new language variants.

Finally, automatic generation for framework is a new idea in GIPSY. The comparison results can be found in Table 5-1.

System Criteria	Centaur	FNC-2	Eli	LISA	Polyglot	JastAdd	GLU	GIPSY
Flexibility	Low	Low	Low	Medium	High	Medium	Low	High
Extensibility	Low	Low	Mediu m	Medium	High	High	Low	High
Hybrid programming support	No	No	No	No	No	Yes	Yes	Yes
Visual programming support	No	No	No	Yes	No	No	No	Yes
Integrated environment	Single- System	Feder- ated	Fede- ated	Inte- Grated	Frame- work	Inte- grated	Inte- grated	*Frame work federati on*
Usability	No	Yes	No	No	Yes	Yes	Yes	Yes
Automatic generation	No	No	No	No	No	No	No	Yes

Table 5 - 1: The comparison among typical existing systems

Users like to ask for perfect, they hope a system can provide all functionalities they need. In the real world, it is a very hard task. The GIPSY system, from the start, considers a lot of factors and tries to achieve as many requirements as possible. It combines the object-oriented methodology and distributed execution concepts. It uses the characteristics of intensional programming language and keeps the design as simple as possible, despite its numerous and stringent requirements base. Before GIPSY, there was no such compiler construction system which realizes all these functionalities and qualities. Especially on framework design, GIPSY introduces automatic generation to provide maximal power.

5.5 Summary

In this chapter, we discuss some typical existing compiler construction systems and the criteria to evaluate these systems. We can conclude that the GIPSY system get a further step in such system design by the comparison.

This chapter illustrates the contribution of the GIPC framework design from theory aspect; in the later chapters, we will, from a practical aspect, demonstrate the reality of this GIPC framework design. A set of open problems exist here in the GIPC framework design, for example, how to realize the hybrid programming in the GIPC? In next chapter; we will narrow down the topic to hybrid language and in Chapter 7 we will state the implementation specific details on hybrid programming in the GIPC.

Chapter 6 : Object-Oriented Intensional Programming Language Design in the GIPC

The GIPSY system supports hybrid programming between Lucid and standard procedural languages. At first glance, this introduces a challenge - how to address objects in Lucid; descend to particulars, this also triggers off a new hybrid language design. In this chapter, we will introduce this OO-IP hybrid language, which can bring benefits to both languages.

6.1 Introduction

The GIPSY is a programming environment in which users can create and execute Lucid programs that may use functions written in procedural programming languages. These hybrid programs are then executed either in sequential, multithreaded, distributed or parallel mode. The compiler is designed in a way that can support hybrid programming, combining the different members of the Lucid family of intensional programming languages, and various programming languages. In our current implementation, we are concentrating our efforts on Java.

However, Lucid and Java belong to totally different programming language paradigms. Java is an imperative language and Lucid variants are declarative languages. As an object-oriented programming language, Java needs strict type declaration. There are no intensionality and dimension in Java, and the notation of object is an important element in Java which is absent from all Lucid variants. On the contrary, implicit type declaration is allowed in IPLs, types can be inferred by the atomic elements of expressions; so far, there are only basic data types in Lucid, and the notion of multidimensional streams and intensionality are important concepts in IPLs.

Simply put, we consider introducing intensional object and designing an OOIP language, which will combine essential features of IPL and Java and enrich both languages' semantics. Out of simplicity, we want to preserve the standard semantics to Java parts of the program, which is achieved by allowing our new syntactical constructs to be translated to standard Java before execution.

In the following sections, we start with a basic case that IPL uses Java objects as first class value which is introduced as Objective Lucid in [Mok05]; then, we will move to a more complex case where the notion of *intensional object,* i.e. context-aware objects is introduced.

6.2 Objects as First Class Values

As discussed in Section 1.10.1, one distinguishing feature of Lucid is that identifiers are used to represent multi-dimensional streams of values. Currently, values are only simple numbers. If values could be objects in this hybrid OO-IP language, Lucid programs would be expressing streams of objects. Given that objects lay the ground for much more possibilities than atomic data types, this greatly adds to the possibilities of Lucid programs. Because this hybrid language is between IPL and Java, straightforwardly, the class can be defined in Java. In this case, we notice:

- IPL programs declare streams of objects;
- To access data and function elements of a Java class, the object-oriented "dot" operator is introduced in the IPL syntax;
- Classes are defined in Java and used as values in the IPL;
- Classes do not encapsulate any notion of intensionality, i.e. intensional operators cannot be used inside objects, but can only be used in the IPL to manipulate streams of objects;
- Classes do not encapsulate any notion of dimensionality. Data elements inside objects are not dimensional.

The introduction of the "dot" notation to the IPL forms a new member in the family of Lucid programming languages which is called the Objective Lucid in [Mok05]. In [Mok05], the syntax of Objective Lucid is given as Table 6-1.

```
E    ::=  id
       |  <E>(<E>,...,<E>)
       |  if <E> then <E> else <E>
       |  #<E>
       |  <E> @ [<E>:<E>,...,<E>:<E>]
       |  <E> where <Q> end
       |  <E>.id
       |  <E>.id (<E>,...,<E>)
Q    ::=  dimension id,...,id
       |  id=<E>
       |  id(id₁,...,idₙ)=<E>
       |  <Q><Q>
```

Table 6 - 1: Syntax of Objective Lucid

The “dot” notation is added to allow accessing the objects’ data members and member functions in IPL. In [Mok05], the Additional basic semantics to support hybrid OO-IP language are shown as in Table 6-2 [Mok05].

$$\mathbf{E}_{\text{objV}}: \quad \frac{\begin{array}{l} D,P \mapsto E : v \quad T(v) = D(cid) = (class, cid, JavaCDef) \\ D,P \mapsto vid : vid \quad D(cid.vid) = (classV, cid.vid, JavaVDef) \\ D,P \mapsto \texttt{JVM[[v.vid]]} : v' \end{array}}{D,P \mapsto E.vid : v'}$$

$$\mathbf{E}_{\text{objF}}: \quad \frac{\begin{array}{l} D,P \mapsto E : v \quad T(v) = D(cid) = (class, cid, JavaCDef) \\ D,P \mapsto fid : fid \quad D(cid.fid) = (classF, cid.fid, JavaFDef) \\ D,P \mapsto E_1,\dots,E_n : v_1,\dots,v_n \\ D,P \mapsto \texttt{JVM[[v.fid(}v_1\texttt{,...,}v_n\texttt{)]]} : v \end{array}}{D,P \mapsto E.fid(E_1,\dots,E_n) : v}$$

$$E_{\text{freeF}}: \quad \frac{D(\mathit{ffid}) = (\mathit{freefun}, \mathit{ffid}, \mathit{JavaFreeFDef}) \qquad D,P \mapsto E_1,\ldots,E_n : v_1,\ldots,v_n \quad D,P \mapsto \mathbf{JVM[\,[ffw.ffid(v_1,\ldots,v_n)]\,]} : v}{D,P \mapsto \mathit{ffid}(E_1,\ldots,E_n) : v}$$

$$\#\text{JAVA}_{\text{CDef}}: \quad \frac{\mathit{JavaCdef} = \mathtt{class\ \ cid\{\ldots\}}}{D,P \mapsto \mathit{JavaCDef} : D\dagger[\mathit{cid} \mapsto (\mathit{class}, \mathit{cid}, \mathit{JavaCDef})], P}$$

$$\#\text{JAVA}_{\text{VDef}}: \quad \frac{\mathit{JavaCDef} = \mathtt{class\ \ cid\{\ldots JavaVDef\ldots\}} \qquad \mathit{JavaVDef} = \mathtt{public\ \ type\ \ vid\ \ldots;}}{D,P \mapsto \mathit{JavaCDef} : D\dagger[\mathit{cid.vid} \mapsto (\mathit{classV}, \mathit{cid.vid}, \mathit{JavaVDef})], P}$$

$$\#\text{JAVA}_{\text{FDef}}: \quad \frac{\mathit{JavaCDef} = \mathtt{class\ \ cid\{\ldots JavaFDef\ldots\}} \qquad \mathit{JavaFDef} = \mathtt{public\ \ ft\ \ fid(fpt_1\ \ fp_1,\ldots,fpt_n\ \ fp_n)\{\ldots\}}}{D,P \mapsto \mathit{JavaCDef} : D\dagger[\mathit{cid.fid} \mapsto (\mathit{classF}, \mathit{cid.fid}(v_1,\ldots,v_n), \mathit{JavaFDef})], P}$$

$$\#\text{JAVA}_{\text{FreeFDef}}: \quad \frac{\mathit{JavaFFWCDef} = \mathtt{class\ \ ffw\{\ldots JavaFreeFDef\ldots\}} \qquad \mathit{JavaFreeFDef} = \mathtt{ft\ \ ffid(fpt_1\ \ fp_1,\ldots,fpt_n\ \ fp_n)\{\ldots\}}}{D,P \mapsto \mathit{JavaFFWCDef} : D\dagger[\mathit{ffw.ffid} \mapsto (\mathit{freefun}, \mathit{ffw.ffid}, \mathit{JavaFreeFDef})], P}$$

Table 6 - 2: Additional basic semantics to support hybrid OO-IP language

According to the GIPC framework design presented in Chapter 4, there is a pre-processor that splits chunks and feeds different code segments to appropriate compilers. The chunks are written in the OO-IP hybrid language. In [Mok05], Serguei A. Mokhov gave the specification of the chunks and defined four different types of “chunk specifications”:

> **#typedecl:** defines user-defined types which will be defined in an imperative language;
>
> **#funcdecl:** defines imperative functions to be used in the part for a new version of Lucid;
>
> **#JAVA:** includes java code and will be fed to Java compiler; this should be extended to other languages as well, e.g. #C++, #FORTRAN, etc, as the following is referring to the general case of IPLs.
>
> **#Intensional LANG:** written in a Specific Intensional Programming Language (SIPL) or in the Generic Intensional Programming Language (GIPL) and will be fed to the appropriate compiler for this variant of Lucid;

The details of specification are shown in Table 6-3 [Mok05].

```
<GIPSY>           ::=  <DECLARATIONS> <CODESEGMENTS>
<DECLARATIONS>    ::=  <FUNCDECLS><DECLARATIONS>
                   |   <TYPEDECLS> <DECLARATIONS>
                   |   Empty
<FUNCDECLS>       ::=  #funcdecl <PROTOTYPES>
<TYPEDECLS>       ::=  #typedecl <TYPES>
<PROTOTYPES>      ::=  <PROTOTYPE> ; <PROTOTYPES>
                   |   Empty
<TYPES>           ::=  <TYPE> ; <TYPES>
                   |   Empty
<PROTOTYPE>       ::=  [immutable]<TYPE>[ []…[] ]<ID>( <TYPELIST> )
<TYPELIST>        ::=  <TYPE> [ []…[] ]
                   |   <TYPE> [ []…[] ] , <TYPELIST>
                   |   Empty
<CODESEGMENT>     ::=  <LANGDATA> <LANGID>
                   |   <LANGDATA> <EOF>
```

<CODESEGMENTS>	::=	<CODESEGMENT><CODESEGMENTS>
	|	Empty
<ID>	::=	<LETTER> (<LETTER> | <DIGIT>)*
<LANGID>	::=	#<CAPLETTER> (<CAPLETTER>)*
<TYPE>	::=	Int
	|	Double
	|	Bool
	|	Float
	|	Char
	|	string
	|	ID
	|	Void

Table 6 - 3: Hybrid Language Specifications

Excerpt 6-1 is developed based on an example in [Mok05]. The pre-processor parser is already generated by JavaCC and can parse this program correctly.

```
#funcdecl
 Nat42()
 inc()
 print()

 #JAVA
class Nat42
{
    private int n;

    public Nat42()
    {
            n = 42;
    }

    public Nat42 inc()
    {
            n++;
            return this;
    }

    public void print()
    {
          System.out.println("n=" + n);
    }
 }

 #OBJECTIVE LUCID
```

```
(N @ [d:2]).print()
where
   dimension d;
   N = Nat42() fby.d N@[d:d-1].inc();
end;
```

Excerpt 6 - 1: the programming example of OO-IP hybrid language

The result of this example will be "n=44". There is a stream of natural number N and each number in this stream is an object.

In this section, we presented the syntax and semantics of adding "objects as first class values" in Lucid according to [Mok05]. The syntax and semantic rules presented here are a complement to the syntax and semantic rules of any variant of Lucid such as presented in [Paq99] and reproduced here, i.e. it naturally allows for the creation of hybrid versions of any member of pure Lucid languages to be extended with objects as first class values. This was achieved quite simply by introducing the object-oriented "dot notation" in language syntax, adding some semantic rules allowing for the introduction of class definitions and their respective public members into the definition environment D, evaluating the operands of the dot operator with the existing semantic rules, and then simply calling the underlying Java Virtual Machine to access the members and let its own semantics take care of this part of the evaluation.

However, this hybrid language design only permits the introduction of objects as first class values into Lucid dialects and thus involves some changes to Lucid, but the syntax and semantics of Java is totally preserved in this case. There are no intensional and dimensional notions inside an object. Moreover, it introduces objects into Lucid, but does not introduce Lucid into objects. So in a sense, this is a "one-way hybrid" language where Java code is "injected" or used by Lucid code, but not the inverse. To have a 2-ways hybrid language, we have to enhance our design; the next section will introduce the other way around, where Java objects are allowed to use Lucid code in their class definitions.

6.3 Intensional Classes Using Java and Lucid

This section introduces Object-Oriented Intensional Programming (OO-IP), a new hybrid language between Object-Oriented and Intensional Programming Languages in the sense of the latest evolutions of Lucid. This new hybrid language combines the essential characteristics of Lucid and Java, and introduces the notion of object streams which makes it is possible that each element in a Lucid stream to be an object with embedded intensional properties. Interestingly, this hybrid language also brings to Java objects the power to explicitly express and manipulate the notion of context, creating the novel concept of intensional object, i.e. objects whose evaluation is context-dependent, which are here demonstrated to be translatable into standard objects. By this new approach, we extend the use and meaning of the notion of intensional objects and enrich the meaning of object streams in Lucid and semantics of intensional objects in Java.This will form a full-fledged hybrid OOIP language, which is the main goal if this thesis. However, as will be uncovered in the remainder of this chapter, introducing Lucid into Java, albeit providing much more possibilities, is matched with proportional difficulties.

6.3.1 Preliminary discussions

To introduce intensional and dimensional concepts into objects, which introduces intensional objects into the OO-IP hybrid language design; there are two ways to do this. One is to allow the use of Lucid syntactical constructs into classes (e.g. Java classes), which we might call "Intensional Object Oriented Programming" (IOOP); the other way is to devise new Lucid constructs to allow the declaration of "Lucid objects", which we might call "Object Oriented Intensional Programming" (OOIP). Before we proceed any further, we must analyze what both mean, what would be their respective benefits and possible drawbacks and difficulties, and make a choice as to which approach we are adopting in this research work.

Solution 1: Lucid classes

One approach requires changing the syntax of Lucid to allow class declarations to be expressed inside the Lucid syntax. This might seem an interesting concept, and we have been lured into it in the course of this research work. But eventually, we have figured out different important reasons why this is not a viable solution. The main reason is that Lucid is a type-less language, i.e. it does not declare types, nor does it explicitly refer to types anywhere in its syntax. For sure its semantics and model of execution use type inference and type analysis, but types are not explicitly stated in Lucid programs. Even when we have introduced object as first class values in Lucid in Section 6.2, we have isolated type references outside of the Lucid part of the syntax, thus preserving Lucid's "syntactical typelessness". As classes are abstract data types, introducing classes inside of Lucid would introduce explicit reference to types into Lucid, and we felt that this would considerably change the language, up to a point where the new language would cease to be in the "standard" Lucid family of languages, and would thus require that our existing execution engine and compiler frameworks would have to be redesigned.

Solution 2: Intensional Java objects

This approach requires an extension of the syntax of Java so that Lucid code is allowed to be included inside otherwise standard Java classes. Here the difficulty lies not so in the syntax, but rather in the semantics of the thus created language. Java has a very well established and standard semantics as embedded in Java Virtual Machines. We certainly do not want to have to dig deep and bury ourselves in Java semantics and JVM implementation. A workaround solution to this problem comes in providing a translation of the embedded Lucid constructs into standard Java, thus having the intensionality syntactically expressed in Lucid but whose semantics rules are semantic translation rules that translate such constructs into standard Java, thus allowing ourselves to use the underlying Java semantics rather than change it. This of course limits certain qualities of our solution, e.g. the resulting implementation would inevitably be less efficient than if directly translated into bytecode. But we have to keep in mind that this work's main goal is language development and proof of concept of the developed languages. Execution optimization will come in time if our language concepts prove to be fruitful.

In order to provide a fully integrated OO-IP hybrid language, we need to provide integration of objects into Lucid, as well as Lucid into objects. Our solution aims at: (1) allowing Lucid to define streams of objects, and to introduce the dot notation in Lucid syntax, allowing Lucid to use objects and their members; (2) allowing classes to define *intensional data members*, as well as allowing methods to use intensional expressions as part of their statements, yielding *intensional methods*; implementation of the proposed solution inside the GIPSY infrastructure, thus permitting (3) the introduction of any flavor of Lucid inside of class declarations by the automated translation of Lucid variants into Generic Lucid prior to execution, as well as (4) the execution of such hybrid OO-IP programs in a scalable distributed environment using the GIPSY's run-time engine architecture. The big picture of our particular vision of OO-IP can be itemized as the following:

- The object-oriented "dot" notation is introduced in Lucid to access data and function elements of a class similarly as it was first introduced in [Mok05].
- Intensionality is introduced into Java classes by the embedding of Lucid expressions into otherwise standard Java classes.
- Java classes encapsulate the notations of intensionality and dimensionality, thus creating *intensional classes*.
- For an intensional class varying over a multidimensional manifold, one instance (i.e. object) of this class exists for each point in this manifold.
- As intensional data members are declared using Lucid expressions and translated into standard Java classes calling the eduction engine for the evaluation of their embedded Lucid expressions, the standard syntax and semantics of Java is preserved.

• An intensional data member in fact declares an intensional relationship applying to the stream of values that it declares across the objects of the intensional class in which it is declared.

• All intensional members' object instances are dynamically generated by the execution of the intensional execution engine, driven by the current context and the Lucid expression defining that intensional member.

• All Java classes embedding Lucid expressions implies intensional evaluation only when evaluating the part of their definition that embeds Lucid expressions. Standard Java classes are still evaluated using the standard Java semantics, i.e. the execution of OO-IP programs is driven by standard Java semantics, unless in the punctual presence of Lucid expressions, in which cases the evaluation is switched to the intensional evaluation engine.

• The hybrid language being proposed here is bound to the Java syntax and semantics, so we thus name it "JOOIP".

6.3.2 Syntax of JOOIP

We describe the syntax of JOOIP in Table 6-4, explaining (1) how can Lucid use objects' members, and (2) how can Lucid expressions be embedded into Java classes.

```
                                        ... standard Java syntax
(1) <data member declaration>  ::= (standard Java data member declaration)
(2)                              | type id = <embedded Lucid expr>;
(3)           <expression term>  ::= (standard Java expression terms)
(4)                              | <embedded Lucid expr>
                                    standard Java syntax...

(5)     <embedded Lucid expr>  ::= /@ <Lucid variant tag> <E> @/
(6)         <Lucid variant tag>  ::= # <Lucid variant id>
```

(7)		| empty
(8)	<Lucid variant id>	::= GIPL
(9)		| INDEXICALLUCID
(10)		| JLUCID
(11)		| OBJECTIVELUCID
(12)		| LUCX
(13)	<E>	::= id
(14)		| <E>(<E>,...,<E>)
(15)		| if <E> then <E> else <E>
(16)		| #<E>
(17)		| <E>@[<E>:<E>,...,<E>:<E>]
(18)		| <E>where<Q>end
(19)		| <E>.id
(20)		| <E>.id(<E>,...,<E>)
(21)	<Q>	::= dimension id,...,id
(22)		| id=<E>
(23)		| id(id_1,...,id_n)=<E>
(24)		| <Q><Q>

Table 6 - 4: Syntax of JOOIP

In order to achieve the stated features set in Section 6.3.1, we have determined that the solution is composed of two separate and complementary aspects. One is about the change to the original syntax of Lucid in order to allow Lucid syntax to manipulate objects' members using the standard "dot notation" used in object-oriented languages. That has already been achieved in various solutions. The other is about the change to the original syntax of Java to allow Java classes to define intensional data members, as well as intensional methods. Syntactically, both are achieved by allowing the embedding of Lucid expressions inside of an otherwise standard Java class. In Table 6-4, productions (13)-(24) provide the first part, and productions (1)-(12) shows the second part. In Table 6-4, productions (19) and (20) are added into the original Lucid syntax to allow Lucid to access, respectively, data and function elements of a class as it was done in Objective Lucid in [Mok05]. The syntax of JOOIP shows in productions (1)-(12)

how to integrate the Lucid expressions into a Java class. We use a tag "/@" to start an embedded Lucid expression, which is to be ended by a corresponding "@/" tag (see production (5)). Preceded by "#", the <Lucid variant tag> indicates which Lucid variant is used for the following embedded Lucid expression (see production (6)-(12)). This will enable our preprocessor to send the Lucid code segment to the corresponding Lucid compiler during translation time. These are *the same tags and design solution used by the* Preprocessor *of GIPC in* generalized compilation [MP05]. This solution permits the JOOIP to embed Lucid expressions written in any variant of Lucid supported by the GIPSY, a unique feature of our solution.

6.3.3 Operational Semantics

We use operational semantics to model the computations of programs, which is a sequence of steps between states [Mos01]. An operational semantics in the style of involves pairs that consist of two components: a program and an environment [Plo81]. According to [Paq99], the semantics introduce the third component: the context; which form a duet <D,P>, where D stands for the intensional definition environment encompassing the definitions defined by the Lucid program which are stored as a cross-referenced dictionary pointing to different AST nodes corresponding to the compiled Lucid program, and P stands for the current context of evaluation. (Details of Lucid semantics could be found in Section 1.9.) The component "Environment" defines an "environment" for storing and retrieving definitions of program identifiers. In this component, the following operations on environment are defined:

- $x \leftarrow v$: Binds a value v to a variable x;
- $D, P \dagger [id \leftarrow v]$: In the definition environment D, and in the evaluation context P, binds a value v to a variable id.
- $D, P \mapsto E: v$: In the definition environment D, and in the evaluation context P, expression E evaluates to value v.
- $E' [id_i \leftarrow E]$: Function calls require the renaming of the formal parameters into the actual parameters.
- $D \dagger [id' \mapsto id], P$: In the definition environment D, and in the evaluation context P, variable id' points to the variable id.

The standard operational semantic rules of generic Lucid from [Paq99, Mok05, MPG05] are extended as shown in Table 6-5. We present here only the rules that require changes from the standard semantic rules, as well as additional rules related to features uncovered by generic Lucid. In these semantic rules, the semantic operator $\dagger$ represents the addition of a mapping in the definition environment D, associating an identifier with its corresponding semantic record, here represented as a tuple. Following is detailed textual description of the meaning of each semantic rule:

$$\mathbf{L_{objV}} \quad : \quad \frac{\begin{array}{l}\mathcal{D}, \mathcal{P} \vdash E : v \quad \mathcal{T}(v) = \mathcal{D}(cid) = (\texttt{class, cid, }\underline{\texttt{JavaCDef}}) \\ \mathcal{D}, \mathcal{P} \vdash vid : vid \quad \mathcal{D}(cid.vid) = (\texttt{classV, cid.vid, }\underline{\texttt{JavaVDef}}) \\ \mathcal{D}, \mathcal{P} \vdash \mathbf{JVM}[[v.vid]] : v_r\end{array}}{\mathcal{D}, \mathcal{P} \vdash E.vid : v_r}$$

$$\mathbf{L_{objF}} \quad : \quad \frac{\begin{array}{l}\mathcal{D}, \mathcal{P} \vdash E : v \quad \mathcal{T}(v) = \mathcal{D}(cid) = (\texttt{class, cid, }\underline{\texttt{JavaCDef}}) \\ \mathcal{D}, \mathcal{P} \vdash fid : fid \quad \mathcal{D}(cid.fid) = (\texttt{classF, cid.fid, }\underline{\texttt{JavaFDef}}) \\ \mathcal{D}, \mathcal{P} \vdash E_1, \ldots, E_n : v_1, \ldots, v_n \\ \mathcal{D}, \mathcal{P} \vdash \mathbf{JVM}[[v.fid(v_1, \ldots, v_n)]] : v_r\end{array}}{\mathcal{D}, \mathcal{P} \vdash E.fid(E_1, \ldots, E_n) : v_r}$$

$$\mathbf{L_{FF}} \quad : \quad \frac{\begin{array}{l}\mathcal{D}(ffid) = (\texttt{freefun, ffid, }\underline{\texttt{JavaFFDef}}) \\ \mathcal{D}, \mathcal{P} \vdash E_1, \ldots, E_n : v_1, \ldots, v_n \\ \mathcal{D}, \mathcal{P} \vdash \mathbf{JVM}[[ffw.ffid(v_1, \ldots, v_n)]] : v_r\end{array}}{\mathcal{D}, \mathcal{P} \vdash ffid(E_1, \ldots, E_n) : v_r}$$

$$\mathbf{J_{CDef}} \quad : \quad \frac{\underline{\texttt{JavaCDef}} = \texttt{class cid } \{\ldots\}}{\mathcal{D}, \mathcal{P} \vdash \underline{\texttt{JavaCDef}} \; : \; \mathcal{D}\dagger[\texttt{cid} \mapsto (\texttt{class, cid, }\underline{\texttt{JavaCDef}})], \; \mathcal{P}}$$

$$\mathbf{J_{VDef}} \quad : \quad \frac{\begin{array}{l}\underline{\texttt{JavaCDef}} = \texttt{class cid } \{\ldots \underline{\texttt{JavaVDef}} \ldots\} \\ \underline{\texttt{JavaVDef}} = \texttt{public type vid } \ldots;\end{array}}{\mathcal{D}, \mathcal{P} \vdash \underline{\texttt{JavaVDef}} \; : \; \mathcal{D}\dagger[\texttt{cid.vid} \mapsto (\texttt{classV, cid.vid, }\underline{\texttt{JavaVDef}})], \; \mathcal{P}}$$

$$\mathbf{J_{FDef}} \quad : \quad \frac{\begin{array}{l}\underline{\texttt{JavaCDef}} = \texttt{class cid } \{\ldots \underline{\texttt{JavaFDef}} \ldots\} \\ \underline{\texttt{JavaFDef}} = \texttt{public ft fid(fpt}_1\texttt{ fp}_1, \ldots, \texttt{fpt}_n\texttt{ fp}_n)\{\ldots\}\end{array}}{\mathcal{D}, \mathcal{P} \vdash \underline{\texttt{JavaFDef}} \; : \; \mathcal{D}\dagger[\texttt{cid.fid} \mapsto (\texttt{classF, cid.fid, }\underline{\texttt{JavaFDef}})], \; \mathcal{P}}$$

$$\mathbf{J_{FFDef}} \quad : \quad \frac{\begin{array}{l}\underline{\texttt{JavaFFWCDef}} = \texttt{class ffw } \{\ldots \underline{\texttt{JavaFFDef}} \ldots\} \\ \underline{\texttt{JavaFFDef}} = \texttt{ft ffid(fpt}_1\texttt{ fp}_1, \ldots, \texttt{fpt}_n\texttt{ fp}_n)\{\ldots\}\end{array}}{\mathcal{D}, \mathcal{P} \vdash \underline{\texttt{JavaFFDef}} \; : \; \mathcal{D}\dagger[\texttt{ffid} \mapsto (\texttt{freefun, ffid, }\underline{\texttt{JavaFFDef}})], \; \mathcal{P}}$$

Table 6 - 5: Additional basic semantics of JOOIP

J$_{\text{CDef}}$: semantically identifies a Java class by the syntactical form: class **cid** {...} , associates this class declaration to the identifier **cid**, and stores it in the definition environment D as the semantic record (class, cid, JavaCDef). A class can contain member variables (JavaVDef) and member functions (JavaFDef, also called "methods"). These are processed in a similar manner by the two following semantic rules.

J_{VDef} : semantically identifies a Java class member variable (or data member) in a Java class JavaCDef by the syntactical form: public type vid... found inside of this class declaration. The semantic record (classV, cid.vid, JavaVDef) is used to represent a Java class data member **vid** declared inside a class declaration JavaVDef for the class **cid**.

J_{FDef} : semantically identifies a Java member function in a Java class JavaCDef by the syntactical form: public ft fid(fpt_1 fp_1,..., fpt_n fp_n){...}. The semantic record (classF, cid.fid, JavaFDef) is used to represent a Java member function **fid** declared inside a class declaration JavaCDef for the class **cid**.

J_{FFDef}: semantically identifies a "Java free function" (i.e. a free function such as in C++, but written in Java syntax) that is not explicitly defined in a given class, and has the syntactical form: ft ffid(fpt_1 fp_1,..., fpt_n fp_n){...}. The semantic record (freefun, ffid, JavaFreeFDef) is used to represent a "Java free function" **ffid**, i.e. a function that is directly available in the Lucid program, and that is not a member of any class. Note that free functions are not allowed in standard Java. In terms of implementation, these "free functions" are all put inside a wrapper class to be part of the worker component of the execution engine as originally defined in [Mok05, MPG05].

L_{objV}: defines the semantics of the evaluation of a reference to a class data member by a Lucid expression using the object-oriented dot operator. The top part of the rule insures that, in **E.vid**: (1) the Lucid expression **E** evaluates to a value **v** that is an object of type **cid**, as being associated in the definition environment D to the tuple (class, cid, JavaCDef), (2) the variable **vid** is a public member of the class **cid**. Once this is established as holding, the Java Virtual Machine can be called upon to evaluate ***v.vid*** (noted as **JVM[[*v.vid*]]**), to yield a value v_r.

$\mathbf{L_{objF}}$: defines the semantics of the evaluation of a reference to a class member function by a Lucid expression using the object-oriented dot operator. The top part of the rule insures that, in **E.*fid*(E$_1$,..., E$_n$)**: (1) the Lucid expression **E** *evaluates to a value* **v** *that is an object of type* **cid**, *as being associated in the* definition environment D to the tuple (class, cid, <u>JavaCDef</u>), (2) the method **fid** is a public member of the class **cid**. Once this is established as holding, all actual parameters are evaluated to values $v_{1,...,}v_n$, the JVM can be called upon to evaluate ***v.f id(v$_1$,..., v$_n$)*** (noted **JVM*[[v.fid(v$_1$,..., v$_n$)*]]**), to yield a value v_r.

$\mathbf{L_{FF}}$: defines the semantics of the evaluation of "Java free functions". The rule is a simpler version of $\mathbf{L_{objF}}$ with no class type identifiers present, and no object to compute upon which to call the function. As mentioned earlier, Java does not have free functions, so we cannot claim that the Java Virtual Machine can execute them. In fact, all free functions are wrapped in a "free function wrapper" class at compilation, with all free functions inserted in it as static functions [Mok05, MPG05]. The $\mathbf{J_{FFDef}}$ rule is inserting all the free functions in this wrapper class, which we called **ffw**. Then, upon calling such "free functions", this rule is called and assumes that the "free functions" have been wrapped as static functions into the **ffw** class, then call the appropriate function. This mechanism is an improvement and refinement over [Mok05, MPG05].

The semantics describes precise requirements for developers. Here, we give an example to show how the semantics is applied. Suppose there is a Lucid expression *E* of type class *IA* which includes a Lucid stream *N* = 0,1,2.... , and a member function *average(n, m)* that will get average value of n-th and m-th elements' value of the Lucid stream *N*. Then *E.average(m, n)* yields (**N***vm* + **N**vn)/2, where **N***vm* and **N***vn* are the m-th and n-th elements' values of stream *N*, respectively.

The rule to use is $\mathbf{L_{objF}}$, which would conclude

D, P |- E.average(m, n) : (Nvm + Nvn)/2

To establish this, if working from bottom to top, there will be

D, P |- JVM[[v.average(vm, vn)]] : (Nvm + Nvn)/2

Since the whole system depends on the correctness of the underlying Java implementation, here we suppose Java compiler translates 'v.average(vm,vn)' into byte codes correctly and that the JVM interprets this byte code in a standard manner.

In the above rule, Lucid expression E will evaluate to a value *v* which is an object of class *IA,* as being associated in the definition environment **D** to the tuple (class, cid, JavaCDef).

The second line of the rule explains how to find *average(x, y)*: it must be contained within **D**, so that the application **D**(average) yields a tuple which says that there is a member function named *average* that has been declared with a JavaFFDef.

The third line of the rule explains how the variables get evaluated:

D, P |- m, n : vm, vn

The instantiation of this rule for the example would be:

Def1 = class IA { ... Def2... }

Def2 = Stream N;

int average(int x, int y) { return N@x + N@y; }

D, P |- Def2 : D † [v.average -> (v, N, average, Def2)], P

In this way, we also get context for each intensional object.

Table 6-6 shows the translation rules applied to translate JOOIP code into standard Java code. The following paragraphs explain each of the translation rules.

```
(TR1)  Start  =>
       (1)  new file:  cid.java
       (2)  new StringBuffer:  HdBuf, IdentifierBuf, StaticBuf, MethodBuf, JavaBuf
       (3)  HdBuf             << "import gipsy.lang.*;
                                   ......
                                   import gipsy.GEE.GEE;"
                                   public class cid implements ISequentialThread{ "
       (4)  IdentifierBuf     << "private GIPSYContext oContext;"
       (5)  StaticBuf         << "static {"
       (6)  MethodBuf         << "public WorkResult work(){cid.main(null); return null;}"
(TR2)  standard Java syntax  =>
       (7)  JavaBuf           <<  standard Java data member declaration
       (8)  JavaBuf           <<  standard Java expression terms
(TR3)  <data member declaration> ::= qualifier type id = <embedded Lucid expr>  =>
       (9)  IdentifierBuf     << "private type id =0;
                              <<  private Boolean b(id)IsWritten = false;"
       (10) IdentifierBuf     <<  see (TR4);
(TR4)  <embedded Lucid expr>  ::= /@ <Lucid variant tag> <E> @/    =>
       (11) IdentifierBuf     << "private static GIPSYProgram SoGEER_i;
                              <<   public GEE oGEE_i = new GEE(SoGEER_i);"
       (12) StaticBuf         << "GIPC oGIPC_i = new GIPC(<E>);
                              <<  oGIPC_i.compile();
                              <<  SoGEER_i = oGIPC_i.getGEER();"
       (13) JavaBuf           << "oGEE_i.eval(oContext);"
(TR5)  Finish  =>
       (14) StaticBuf         << "}"
       (15) cid.java          << HdBuf;
       (16) cid.java          << IdentifierBuf
       (17) cid.java          << StaticBuf
       (18) cid.java          << MethodBuf
(19) cid.java        << JavaBuf
(20) cid.java        << "}"
```

Table 6 - 6: Translation rules to translate JOOIP into Java

TR1. When starting to translate a JOOIP program, the translator first creates a new Java file and five string buffers, the latter being rendered necessary by the fact that the translation output cannot be done sequentially upon sequential scanning of the input JOOIP program. The first translation step is to generate the class declaration header part to the Java file (3), then generate the code for the instance context member of the intensional class (4), then initiate a static block that is to contain static initialization of one GEER data member per each Lucid expression embedded in the intensional class (5). Finally, a **work()** method is generated that is a simple wrapper that calls the main() function of the intensional class; this method is the one called by the GIPSY run-time system when run from within a GIPSY instance as prescribed by the **ISequentialThread** interface that the class is declared to implement at (3). The **work()** method refers to a "WorkResult" class which collects results of work performed by a sequential tread (ST) or a worker. It is the base class for the work results and it is one of the interfaces between ST generators, GIPC in general, GEE in general, IDP and warehouse of GEE.

TR2. This rule signifies that the translator simply copies any standard Java code (i.e. code for which no translation rule applies) as-is to JavaBuf.

TR3. This rule signifies the translation to occur when an intensional data member is encountered by the translator. On the surface, the processor will replace the intensional data member declaration with a standard Java data member declaration. For each such occurrence, the processor records the identifier in a table and leaves a field for later link to the corresponding GEER as processed by (TR4). As translation rule (TR5) takes effect, it will replace the Lucid expression by a call to the GEE to execute the Lucid expression. The qualifier is one of the Java's visibility qualifiers, such as private, protected, or public, where we currently opting out for private. The type denotes a Java member type that Java part expects the Lucid expression to produce. At run-time, when the Lucid expression is done evaluating, the Lucid-to-Java type matching and conversion occurs according to the definition of the type system presented in Section 6.3.6.

TR4. This rule signifies the translation of the embedded Lucid code segments, as identified by the occurrence of the opening /@ and closing @/ markers. For each such occurrence, the translator generates an identifier of type **GIPSYprogram** (representing a GEER) and generates code in the StaticBuf to generate the instance GEER at run time by calling the appropriate parser in the GIPC framework, as directed by the Lucid variant tag. The translator then effectively replaces the embedded Lucid expression invocation by a call to the GEE that evaluates this Lucid expression in the context of the object in question.

TR5. Finally, the translator aggregates all generated parts in the proper order to finish the generation of the intensional class in pure Java.

6.3.4 Implementation Details of the JOOIP Compiler

The JOOIP hybrid language is designed as a 2-way hybrid: we allow the embedding of Lucid code in the Java language and the Lucid code may refer to Java objects' members. Thus, at the implementation level, we design the modules in such a way as to make sure that the JVM can work smoothly with the GIPSY run-time engine. To avoid changing the Java compiler, we have chosen to implement the translation process as a multi-pass compiler that translates the JOOIP classes into pure Java classes, according to the semantics and translation rules presented earlier in Section 6.3.3. This compiler is fully integrated into the GIPSY, and, thus, is a GIPSY compiler instance in the GICF framework. Integrating our solution into the GIPSY permits us to reuse the existing Lucid compiler components, as well as executing the Lucid parts using the GIPSY's run-time environment, the GEE. The compilation process can be summarized as the following:

1. Firstly, the translator, as it parses a JOOIP program, identifies all embedded Lucid expressions and conceptually replaces them by a regular Java variable *Lucid_expr_i*, where *i* means it is the *i-th* embedded Lucid expression placed in the JOOIP. Concretely, in order to yield a value from the evaluation of this Lucid expression, the occurrence of the Lucid expression is replaced by a call to the GEE for the evaluation of this compiled Lucid expression (which has a corresponding GEER) by the GEE in the context *oContext* associated with the object in question. At the same time, the translator creates entries in a Lucid identifier symbol table to record *Lucid_expr_i*, as well as its corresponding Lucid expression in the form of a pointer to the root node of that expression in the AST. After all such embedded Lucid expressions are translated in this manner, the JOOIP program becomes an intermediate Java program, which can be syntactically parsed by a standard Java parser, but the parser will not be able to capture the special meaning for the *Lucid_expr_i* variables.

2. Then, the translator passes the intermediate Java program generated in the first step to our customized Java parser generated from the Java 1.5 grammar specification (found in [JavaCC]) altered to allow extraction of the Lucid code segments in order to get the symbol table that records Java class data members and becomes aware of the *Lucid_expr_i* corresponding to an *intensional data member declaration* or an *intensional expression term*. After learning that some *Lucid_expr_i* corresponds to an intensional data member declaration, the processor will apply translation rule (TR3) to generate standard Java code. If the *Lucid_expr_i* corresponds to an intensional expression term, the processor will apply translation rule (TR4) to generate standard Java code.

3. Next, the translator passes each embedded Lucid expression to the corresponding Lucid compiler, whose parser name is specified by the *Lucid variant id* tag as signified in translation rule (TR4). The tag identifiers here correspond exactly to the language ID tags as defined in GICF [MOK05] of the language parsers currently available in the GIPSY. In GICF *Lucid variant id* is known as LANGID and is presented in the syntax rules (8)-(12) in Table 6-4. Every Lucid compiler will return a GEER for each *Lucid_expr_i*, which is stored in the Lucid identifier table.

4. After that, the translator does semantic checking only for the embedded Lucid expressions (the Java complete code checking is deferred to later analysis by the standard Java compiler). The limited semantic checking at this stage involves checking the symbol tables produced in the steps 1 and 2 for undefined or multiply defined identifiers in their corresponding scope according to the Lucid and Java semantics. This is especially important when a standard Java identifier is used inside an intensional expression.

5. Finally, the translator generates a pure Java program, in which the **work()** method automatically generated is the one to be called by the GIPSY run-time system, thereby correctly linking this generated class with other such classes through the run-time system. It is also possible to invoke the pure Java program by itself as it defines the **main()** method to be the starting point of the computation, that eventually may invoke the GEE if the compiled Lucid expression is encountered at run-time. In such a case, the invocation of the GIPSY run-time (GEE) is deferred until some later point when an **eval()** call is made to the GEE. For it work, the GIPSY compiled code should be in the CLASSPATH of our generated program at this step to be runnable.

To explain the implementation details of the compilation process described above, we show a class diagram of the JOOIP Compiler in Figure 6-1. The **main()** method starts in class **JOOIPCompiler**, and then, by using available Lucid parsers in the GIPSY, **JavaParser** and **SemanticAnalyzer**, which are already in the GIPC framework, it produces a compiled standard Java class. On the way, we can get all information in the symbol hashtable and ready for the GEE to use when we call it indirectly through the intensional code segments. Just like other compilers in the framework, the large portion of the JOOIP compiler was generated using JavaCC [JavaCC] and the Java 1.5 grammar specification written in java15.jjt. The grammar file was altered to allow for the intensional code segments to be extracted as well as the identifiers as described in the steps above. Later the compiler was customized with some GIPSY-specific annotations to allow it to be a part of the GIPC framework.

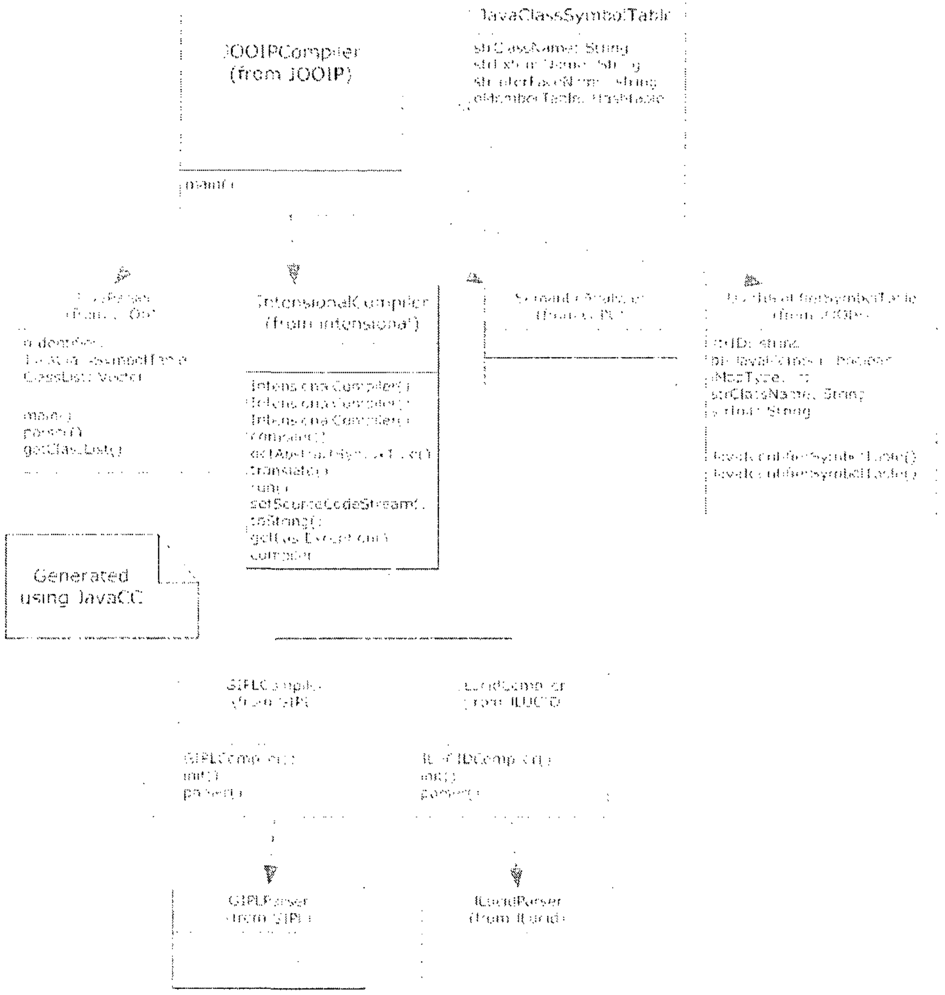


Figure 6 - 1: Class diagram of the JOOIP compiler

6.3.5 Data structures

The main data structure used is a symbol table which records all Java classes, variables, and Lucid identifiers tentatively called **JavaClassSymbolTable**. The detailed design of this table is as shown in Figure 6-1. The string **strClassName** records the fully-qualified Java class name. The string **strExtendName** records the parent's class name. The string **strInterFaceName** records the interface names. The hashtable **oMemberTable** records all members of this Java class. The detailed design of each identifier instance in the **oMemberTable**, called **JavaIdentifierSymbolTable**, is str presented in Figure 6-1. While designing our JOOIP *language and early testing its properties we opted to have these two* classes implemented ourselves. In **JavaIdentifierSymbolTable**, the string **strID** records the variable's name. The Boolean **bIsJavaMember** records if this variable is a plain Java class member or is an intensional identifier, which is defined in the Java class in the form of a Lucid expression. The integer **iMapType** records the data type of this variable, which is used in type matching between Lucid expression and Java types. Further details about the type mapping can be found in Section 6.3.6. The string **strClassName** records the fully-qualified Java class name this variable or intensional identifier belongs to. It is added here to simplify the semantic scope check for the Lucid code segments. The string **strClassInit** records the declaration of this variable, if it is an intensional identifier, the **strClassInit** will represent the intensional source code fragment. The **oEntry** member records the entry of an AST of the intensional identifier. This AST will be ready only after the Lucid code segments are passed to the corresponding Lucid parsers and collected back as ASTs for re-linking. The dictionary **oLucidIdentifierDictionary** records the semantic table for each intensional identifier. The final preparation of the dictionary instance completes only after the semantic checking of Lucid code is performed. In this hashtable, the **oEntry** and **oLucidIdentifierDictionary** will have value only when the boolean **bIsJavaMember** is set to false.

6.3.6 Type system

Just like JLucid or Objective Lucid (the predecessor hybrid intensional-imperative languages studied in the GIPSY environment), the JOOIP language cannot avoid the type mapping between Java and Lucid as the data type sets of both languages are not identical. In Table 6-7 we summarize the type mapping between Java and GIPSY types and the intensional counterpart. A particularity here, that any intensional dialect has a dimension type and its implementing class **GIPSYContext** in the GIPSY type system, but its value cannot be directly mapped to a specific Java type in JOOIP, and the dimensions can presently be, but may not be limited to in the future, integers and strings and have the corresponding types of tag sets [TPM07] attached. This especially concerns the Java method parameters and return types. Additionally, each intensional class instance has its own context managed internally as described earlier in the *oContext* data member of the generated Java class. Instead of doing any re-mapping to the Java type in that regard, we simply kept the object context reference as **GIPSYContext**. The type matching and conversion happens at run-time, when the GEE completes the evaluation of an intensional expression and comes back with a value. The final GIPSY type of the value is determined, examined, and, if possible, converted by the Lucid-Java adapter just before returning to the calling Java class. A run-time type check semantic error may happen at this point in type if it is not possible to match the type according to the table shown. A similar process is invoked in the other direction of evaluation, when the intensional segment uses the result of an imperative evaluation, which was broadly discussed in JLucid and Objective Lucid works earlier. For an in-depth discussion of the GIPSY type system, please refer to [MPT07].

Lucid type	Java type	GIPSY Type System
dimension	int, String	`GIPSYContext`
char	char	`GIPSYCharacter`
int	byte	`GIPSYInteger`
int	short	`GIPSYInteger`
int	int	`GIPSYInteger`
(-)	long	`GIPSYInteger`
float	float	`GIPSYFloat`
double	double	`GIPSYDouble`
bool	boolean	`GIPSYBoolean`
[]	array	`GIPSYArray`
string	String	`GIPSYString`
object	class	`GIPSYObject`
( )	interface	( )
(-)	enum	`GIPSYObject`
bool:true	void	`GIPSYVoid`

Table 6 - 7: Summary of type mappings between Java and Lucid in JOOIP

6.4 Discussions on JOOIP

JOOIP is the intensional programming language, Lucid, conservatively extended with intensional objects, class hierarchy with inheritance and encapsulation; it is an object-oriented imperative language, enhanced with explicit dimension and objects organization. In this section, we will discuss the JOOIP from both object-oriented aspect and intensional aspect.

6.4.1 Object Mutability

Object mutability, i.e. the fact that an object's state or behavior can vary in time, is of central importance in the design of a hybrid OO-IP language. We discuss here the notions of mutable/immutable objects, as well as the notion of context-mutability i.e. the fact that, given that our OO-IP objects are defined in a multidimensional context space, some of these objects are to be constant or mutable across the context space i.e. context-immutable or context-mutable.

Mutable objects

"Mutable object" means when one has a reference to an instance of an object, the contents or state of that instance (i.e. the values of its data members) can be altered, thus making it so that the same object changes its state in time [Java05], and even that its methods would expose an apparently non-functional behavior over time, if the methods are referring to the changing values of its data members. The use of mutable objects with Lucid is highly problematic because the semantics of Lucid assumes that the values associated with expressions are constant in a given context, and thus only have to be evaluated once, thus permitting the storage of evaluation results, and their retrieval when the same expressions are to be evaluated. Permitting to have streams of mutable objects would thus be semantically invalid, as each stream element's value could possibly change in time. One interpretation to mutable objects from a valid IP perspective would be that mutable objects could be represented as a stream of immutable objects, where the stream is "recording" the changes in an object's state in the time dimension. That, in fact, brings us to the absolute beginning of

Lucid where Lucid streams were used to represent the changes of values of program variables upon and throughout execution, in the context of program verification.

Immutable objects

Immutable objects are simply objects whose state (the object's data) does not change after construction [Java05]. Immutable objects thus expose strict functional behavior throughout their entire lifetime. An immutable object is an object whose evaluation of any of its member always return the same value, behaving like as if all its members would be preceded by *const* in C++ or *static final* in Java, and all its functions always return the same value when given the same parameters. Immutable objects work nicely in our view of OO-IP, because of the converse of the reasons provided in the previous section discussing mutable objects.

Context-immutable objects

This is a concept arising from the field we are discussing in this research, i.e. objects whose evaluation is context-independent. In Intensional Programming, expressions are evaluated in a multidimensional context, possibly yielding a different value for the same expression when evaluated in different contexts. In Object-Oriented Programming, expressions evaluate to objects, and there is no such thing as the context of evaluation. Combining these two paradigms, a context-immutable object is an object whose evaluation is independent on the context of evaluation, i.e. an object that is invariant in a multidimensional context space. This means that a "context-immutable object" does not embed any intensional expression, or it may be an object that embeds an intensional expression, but that expression is context-invariant. We can say any traditional Java object is a context-immutable object.

Context-mutable objects

An object of this classification is one whose evaluation is dependent on the context of evaluation. This means a context-mutable object has an embedded *dimensional concept in its class declaration and that the evaluation of its* members varies in a multidimensional context space, i.e. this class embeds at least one Lucid expression. In our approach, intensional objects are declared by the inclusion of Lucid expressions in their class declarations, either by having data members declared as a Lucid expression, or having member functions to embed Lucid expressions as part of their Java statements and expressions. By this way, we can permit both mutable and immutable objects by embedding or not embedding such Lucid expressions in their class declarations. Interestingly, as soon as the context of evaluation is decided, objects are evaluated to context-immutable objects which are normal Java objects.

6.4.2 Demand-Driven Constructors

Object-oriented programming uses constructors to generate objects. However, Lucid uses demand-driven evaluation, and may generate a demand for objects that has not been constructed yet. We cannot design constructors for Lucid streams because most Lucid streams are infinite. However, we translate *intensional data members* in an *intensional class* to standard Java data members by adding a constructor that calls the eduction engine for the creation of the new object. When there is a demand for an object, the constructor of the corresponding class is called, potentially generating demands for the computation of each *intensional attribute* in this class. For example, if each element on the stream **S** is an object. Then "**S@[d:3]**" generates a demand for an object, which generates other demands on evaluation at context: **[d:3]**. This demand for an object is translated by calling **GEE(variable, context)** in the constructor in the translated Java class. This mechanism is in fact defining *demand-driven constructors*.

6.4.3 Inheritance

In the JOOIP semantics presented in Figure 6-5, we only allow embedded Lucid expressions to refer to members defined locally in the current class. We chose to do so out of simplicity of the expression of the semantics and its implemented solution that parses the JOOIP classes and extracts its local members in order to process the first semantic checking/translation step. Java reflection provides a powerful mechanism for the extraction of class members, even across an inheritance tree. That can then be used to do semantic checking taking into consideration an inheritance tree. However, reflection can only be used on standard Java classes, which JOOIP classes are not. We could further extend the possibility of the use of inheritance in JOOIP classes by using reflection in the semantic checking/translation steps occurring after the standard Java code has been generated.

6.4.4 Intensional Relationship Across Object of an Intensional Class

A standard Java object is an instance of a Java class. These objects are isolated, except by the fact that they belong to the same class. For example, if we have a class **version** which the declaration is as in Excerpt 6-2.

```
public class version
{
      public int  iNum;
      public String sAuthor;
      public String sChange;

      public void print()
      {
        System.out.println("version is " + iNum);
      }

      public static void main(String[] argv)
      {
        version oTest = new version();
        oTest.print();
      }
}
```

Excerpt 6 - 2: A class version

Let us assume that there are a set of objects **a, b, c, d, e** which are different versions of class **version.** As shown in Figure 6-2, the relationship among objects only lies on they have same structure as the class **version,** the individual evaluation is unrelated to the others. If object **a** wants to communicate with object **b**, you have to pass clear instruction messages.

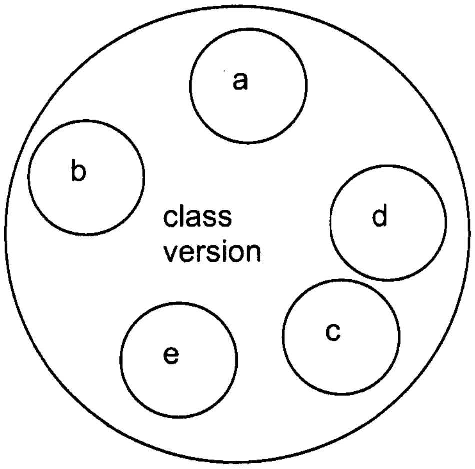


Figure 6 - 2: The class **version** and its objects

JOOIP improves the case by providing clear relationship among objects. If we define a stream which elements are objects of class **version**, for the same case, objects **a, b, c, d, e** which are different versions of class **version** not only have same interface inheriting from Java, but also have very clear relationship which is defined by the IPL operator, **fby,** with explicit dimension **time** as shown in Figure 6-3.

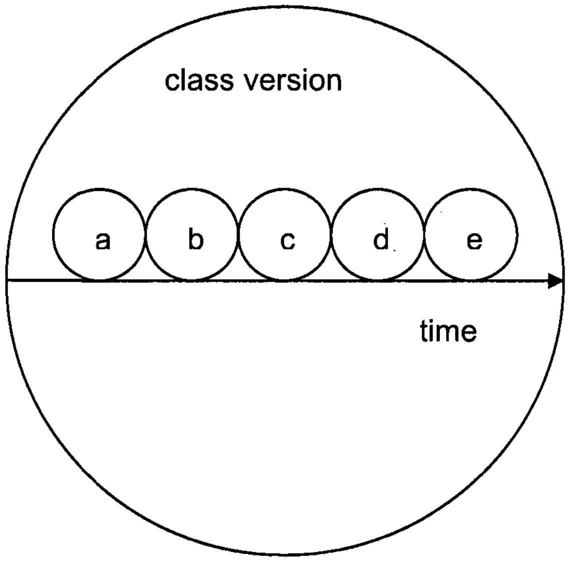


Figure 6 - 3: Java objects in Hybrid JOOIP

JOOIP allow implicitly defining intensional relationships among objects of a JOOIP intensional class. JOOIP inherits the essential feature of Lucid that permits it to use intensional operators to express relationship between across a multidimensional context space, in turn permitting to define streams of elements whose values depend on other elements. In JOOIP, since we have objects as first class values, we can in fact create intensional relationships among Java objects of a JOOIP class. By declaring an intensional data member inside a JOOIP intensional class, we in fact implicitly define an intensional relationship *across* the objects of this class.

This feature can help update attributes in objects by group and it will be extremely useful for database storage, which would become an *intensional database*.

6.5 Examples of Application

Having described the syntax and semantics of JOOIP, then discussed some important issues of the resulting hybrid paradigm, we have kept our explanations at a conceptual level that maybe does not allow the reader to see concretely how can JOOIP be used to write programs, how such programs are effectively translated, and what are the advantages brought forth by the use of JOOIP. This section presents simple JOOIP program examples that will concretely demonstrate the capacities of JOOIP, as well as illustrate its translation process and its resulting output code.

6.5.1 Simple Example Illustrating the JOOIP-to-Java Translation Process

Excerpt 6-3 shows the typical natural number example derived from [Paq99] of Lucid written in JOOIP.

```
public class GIPLtest
{
    private int N = /@#GIPL
                  if (#.d) <= 0 then 42 else (N+1) @.d (#.d)-1 fi
                    where
                      dimension d;
                    end @/;

    public int computeLocalAverage(int f)
    {
       return  ( /@ N@.d f - 1 where dimension d; end @/
               + /@ N@.d f + 1 where dimension d; end @/) / 2;
    }

    public void print()
    {
        System.out.println("N=" + N);
    }

    public static void main(String[] argv)
    {
        GIPLtest oTest = new GIPLtest();
        oTest.N = oTest.computeLocalAverage(2);
        oTest.print();
    }
}
```

Excerpt 6 - 3: GIPLtest.jooip - natural number example written in JOOIP

The problem is to extract a value from the stream representing the natural numbers, beginning from the ubiquitous number 42. Let us arbitrarily pick the third value of the stream and set the stream's variance in the d dimension; then in the **main()** method, the tag number two will be assigned to the method **computerLocalAverage()**. The method **computerLocalAverage(2)** will ask the average value of its neighbours, which is the average of the second and forth value of the stream. With not much intuition, one can readily expect the program to return the value 44 which is the result of calculation (43+45)/2.

The program will start with the Java **main()** method which will call **computeLocalAverage(int f)** method. According to the semantic translation process, the Lucid expression embedded in **computeLocalAverage(int f)** will be changed to method calls, **GEE.eval([d:f-1],N)** and **GEE.eval([d:f+1],N)**. Here we will pass parameter **f** as context for Lucid expression. The run-time engine will drive two demands to calculate the value of the first and third element of the stream **N**. The definition of **N** is already stored in the corresponding GEER. According to the definition of **N**, after executing, engine returns the **Nat42** object with value 44. The method **print()** will display the result on the screen. The translation process can be abstractly described as the following:

- According to the semantic rules, the JOOIP processor parses this program first and extracts all Lucid code segments as shown in the intermediate file in Excerpt 6-4.

```
public class GIPLtest
{
    private int N = IPL_CODE_1;

    public int computeLocalAverage(int f)
    {
       return  ( IPL_CODE_2
               + IPL_CODE_3) / 2;
    }

    public void print()
    {
        System.out.println("N=" + N);
    }

    public static void main(String[] argv)
    {
        GIPLtest oTest = new GIPLtest();
        oTest.N = oTest.computeLocalAverage(2);
        oTest.print();
    }
}
```

Excerpt 6 - 4: GIPLtest.jop – intermediate file of natural number example

- The JOOIP compiler then passes the three Lucid segments to the right GIPL parser to get the corresponding GEERs.
- Then, the JOOIP compiler passes the GIPLtest.jop program to the Java Parser to get the class's symbol table.
- With the class symbol table and GEERs in hand, the JOOIP compiler calls upon the GIPSY's Semantic Analyzer to do semantic check for each Lucid code segment.
- If the semantic check passes, at the last step, the JOOIP compiler generates a standard Java class, GIPLtest.java, which is shown in Excerpt 6-5.

```
package gipsy.tests.jooip;
import gipsy.GIPC.GIPC;
import gipsy.lang.*;
import gipsy.lang.converters.type.*;
import gipsy.interfaces.*;
import gipsy.lang.context.Dimension;
import gipsy.GEE.GEE;
import gipsy.util.*;
import java.lang.reflect.Method;

public class GIPLtest  implements ISequentialThread
{
      private static GIPSYProgram soGEER1;
      private static GIPSYProgram soGEER2;
      private static GIPSYProgram soGEER3;
    private int N = 0;
      private boolean bNIsWritten = false;
      private GIPSYContext oContext;

      public GEE oGEE3 = new GEE(soGEER3);
      public GEE oGEE2 = new GEE(soGEER2);
      public GEE oGEE1 = new GEE(soGEER1);

      static{
            try
            {
                  GIPC oGIPC1 = new GIPC(new StringInputStream("N
where N =                            if (#.d) <= 0 then 1 else (N + 1)
@.d (#.d) - 1 fi                     where
dimension d;                      end ; end"), new String[] {"--gipl",
"--debug"});
```

```
                    oGIPC1.compile();
                    soGEER1 = oGIPC1.getGEER();
                    GIPC oGIPC2 = new GIPC(new StringInputStream(" N@.d
f - 1 where dimension d; end "), new String[] {"--gipl", "--debug"});
                    oGIPC2.compile();
                    soGEER2 = oGIPC2.getGEER();
                    GIPC oGIPC3 = new GIPC(new StringInputStream(" N@.d
f + 1 where dimension d; end "), new String[] {"--gipl", "--debug"});
                    oGIPC3.compile();
                    soGEER3 = oGIPC3.getGEER();
              }
              catch(Exception e)
              {
                    System.err.println(e);
                    e.printStackTrace(System.err);
              }
        }

        public GIPLtest(GIPSYContext poContext)
        {
              this.oContext = poContext;
        }

        public GIPLtest()
        {
        }

        public WorkResult work()
        {
              GIPLtest.main(null);
              return null;
        }
        @Override
        public WorkResult getWorkResult()
        {
              return null;
        }

        @Override
        public void setMethod(Method poSTMethod)
        {
        }

        @Override
        public void run()
        {
              work();
        }
/**
 *     @version $Id: GIPLtest.jooip,v 1.10 2002/11/14 18:50:05 aihua_wu
Exp $
 */


    public int computeLocalAverage(int f)
    {
            GIPSYContext oContext2 = new GIPSYContext();
            Dimension oDimension2 = new Dimension();
            oDimension2.setDimensionName(new GIPSYIdentifier("d"));
            oDimension2.setCurrentTag(new GIPSYInteger(f - 1));
            oContext2.addDimension(oDimension2);

            GIPSYContext oContext3 = new GIPSYContext();
            Dimension oDimension3 = new Dimension();
            oDimension3.setDimensionName(new GIPSYIdentifier("d"));
            oDimension3.setCurrentTag(new GIPSYInteger(f + 1));
            oContext3.addDimension(oDimension3);

        return
( IPLToJava.convertToInteger(this.oGEE2.eval(oContext2))
              +
IPLToJava.convertToInteger(this.oGEE3.eval(oContext3))) / 2;
    }

    public void print()
    {
        System.out.println("N=" + N);
    }

    public static void main(String[] argv)
    {
        GIPLtest oTest = new GIPLtest();
        oTest.N = oTest.computeLocalAverage(2);
        oTest.print();
    }
}
```

Excerpt 6 - 5: The translated pure Java class – GIPLtest.java

Notice here that, following our analysis/translation method, the embedded Lucid expressions ought to be "self-contained". As it is now, upon semantic check, the processor will report "undefined dimension **d**" for expressions using a dimension without a corresponding dimension declaration inside the **where** clause. That can be fixed by allowing our semantic checker to have awareness of the dimensionality of the intensional data members defined in the class, thus eliminating the need to redeclare dimensions in each embedded Lucid expression that refers to these dimensions. Once the JOOIP class has been translated to a regular Java class, the standard Java compiler will take care of the program compilation and running in within the JVM will print the result on the screen. Since the program also implements the **ISequentialThread** interface, it can be run by the GEE as well as being transported as demands distributedly.

The above GIPLtest.java program is already correctly parsed by the Java parser and is smoothly integrated with GEE. From output, it shows that the GIPLtest.java calls engine and generate correct ASTs and dictionary.

6.5.2 Euler and Feynman Algorithms in JOOIP

Euler and Feynman Algorithm are very famous in physics. This section will show how to use the JOOIP to apply the Euler and Feynman Algorithm as well as the comparison with a traditional implementation in Java.

By Newton's second law [Phy_Java] we can calculate the acceleration of a body once we know the forces acting on it. The forces are either contact forces or field forces and may vary with time, position and velocity as the body moves. In order to describe the kinematics of the motion we need expressions for where the particle is and how fast it is moving at any time. A table which consists of values of position and velocity at specified time intervals is a numerical approach to kinematics. The accuracy of the tabled values depends on the approximations involved in their calculation.

In the Euler algorithm, the average velocity and acceleration are replaced by the velocity and acceleration at the beginning of the interval as the equations (1)&(2) where ***t*** is the beginning of the time interval, ***dt*** is time interval, ***v*** is velocity, ***a*** is acceleration and ***x*** is position:

$$v(t+dt) = v(t) + a(t).\ dt \qquad (1)$$

$$x(t+dt) = x(t) + v(t).\ dt \qquad (2)$$

The values at the beginning of the interval are known, and although they are not the best approximation for the average values, they are not bad if the time interval is short enough.

The Feynman algorithm approximates the average acceleration and velocity over a time interval by their values at the midpoint (in time). The equations on which the Feynman algorithm are based can be written in equations (3)&(4) with same notations as equations (1)&(2).

$$x(t+dt) = x(t) + v(t+dt/2).\, dt \quad (3)$$

$$v(t+dt/2) = v(t-dt/2) + a(t).\, dt \quad (4)$$

In equations (3)&(4), changes in position are calculated using a velocity value that is half a step ahead in time. Likewise, changes in velocity are calculated using an acceleration which is half a step ahead in time. Position and acceleration are therefore 'in-phase' that is, they are calculated at the same points in time, and velocity is stepped half a step out of phase with both position and acceleration.

We can use the Euler and Feynman algorithms to follow the motion of a mass on a spring with assumption that acceleration depends only on time and position. Consider a mass of 2kg attached to a spring with a force-constant of 8N/m. It is passing through its equilibrium position at a velocity of 2.8 m/s at time zero, and we want to follow its motion for one second with a time step of 0.2 seconds. In [Phy_Java], a Java program is provided. The main part is shown in Excerpt 6-6:

```
    k=8; m=2; dt=0.2;
    t=0; y=0; v=2.8; a=-k*y/m;    // Euler Algorithm
    for (int i=1; i<=5; i++) {
```

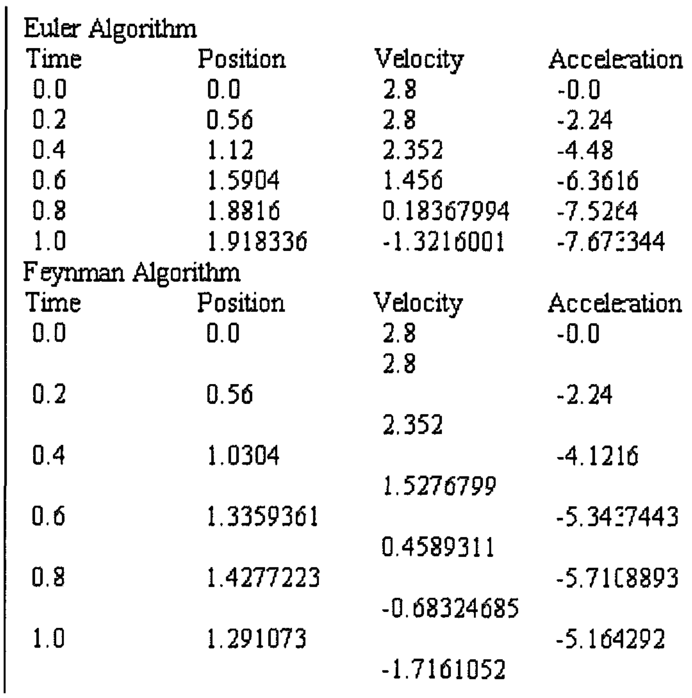
```
    t=t+dt;
    y=y+v*dt;
    v=v+a*dt;
    a=-k*y/m;
  }

  t=0; y=0; v=2.8; a=-k*y/m;    // Feynman Algorithm

  v=v+a*dt/2;
  for (int i=1; i<=5; i++) {
    t=t+dt;
    y=y+v*dt;
    a=-k*y/m;
    v=v+a*dt;
  }
```

Excerpt 6 - 6: Java code for the Euler and Feynman Algorithm application

Figure 6-4 shows the result:



```
Euler Algorithm
Time        Position     Velocity      Acceleration
 0.0         0.0          2.8           -0.0
 0.2         0.56         2.8           -2.24
 0.4         1.12         2.352         -4.48
 0.6         1.5904       1.456         -6.3616
 0.8         1.8816       0.18367994    -7.5264
 1.0         1.918336     -1.3216001    -7.673344
Feynman Algorithm
Time        Position     Velocity      Acceleration
 0.0         0.0          2.8           -0.0
                          2.8
 0.2         0.56                       -2.24
                          2.352
 0.4         1.0304                     -4.1216
                          1.5276799
 0.6         1.3359361                  -5.3437443
                          0.4589311
 0.8         1.4277223                  -5.7108893
                          -0.68324685
 1.0         1.291073                   -5.164292
                          -1.7161052
```

Figure 6 - 4: The result of above program example

We can find the above Java code has the following limits:

1. the expression is very confusing, it cannot tell the natural meaning of the original differential equations easily;

2. infinity cannot be expressed in Java code, we have to pose a beginning and end of time explicitly;
3. it is not very good at description of kinematics of the motion, because we cannot arbitrarily ask for information at a time, i.e. it has a purely extensional view and model of computation;
4. if forces depend on velocity, at the time acceleration is calculated, the only value available for velocity is the one from a half-step earlier. In this algorithm, the acceleration cannot dependent on velocity;

If we assume that time interval ***T***, position **Y**, acceleration ***A*** and velocity ***V*** are streams, we find they have same dimension ***time*** and the process is shown in Figure 6-5.

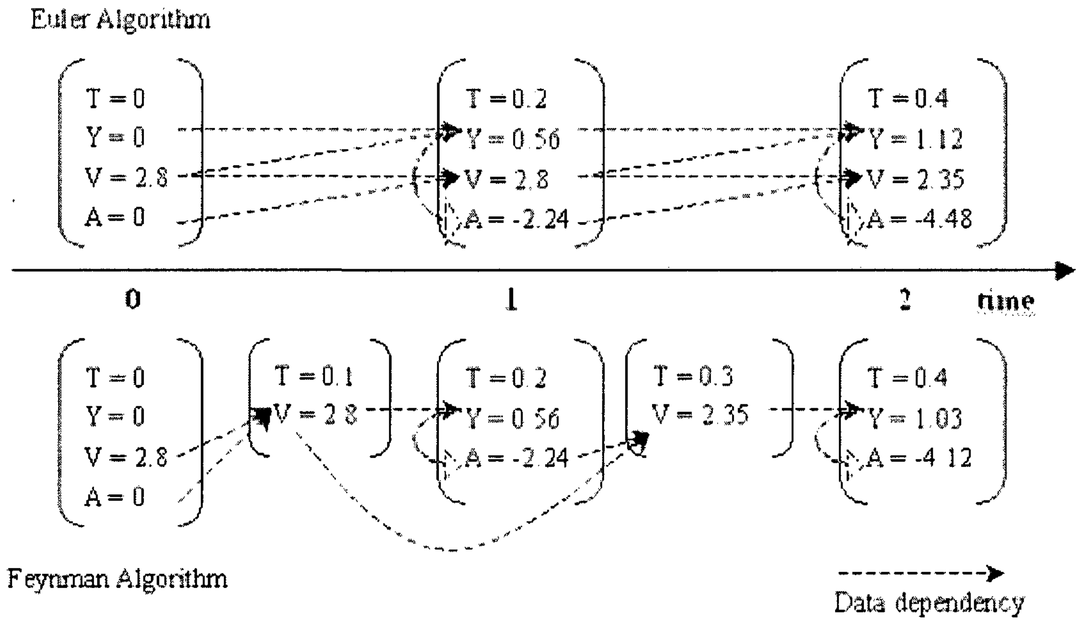


Figure 6 - 5: The process of generating data by both algorithms

From the above figure, we find that the Euler is easier to program in Lucid and results in a program that resembles much more the original differential equations.

However, there is more trouble with the Feynman algorithm because the data is "out of phase". If we consider each group of data as an individual object on the same dimension, objects have different structure. This also conflicts with the semantic restriction that each element of a stream has same type.

Object-oriented concept in the hybrid JOOIP language helps to resolve the trouble. What need to be done is to encapsulate same structure objects into a class, and then we will have 2 classes, class **InPhase** and class **OutPhase**. Even, we can let objects from the two classes to have same context but different value. Figure 6-6 shows the solution:

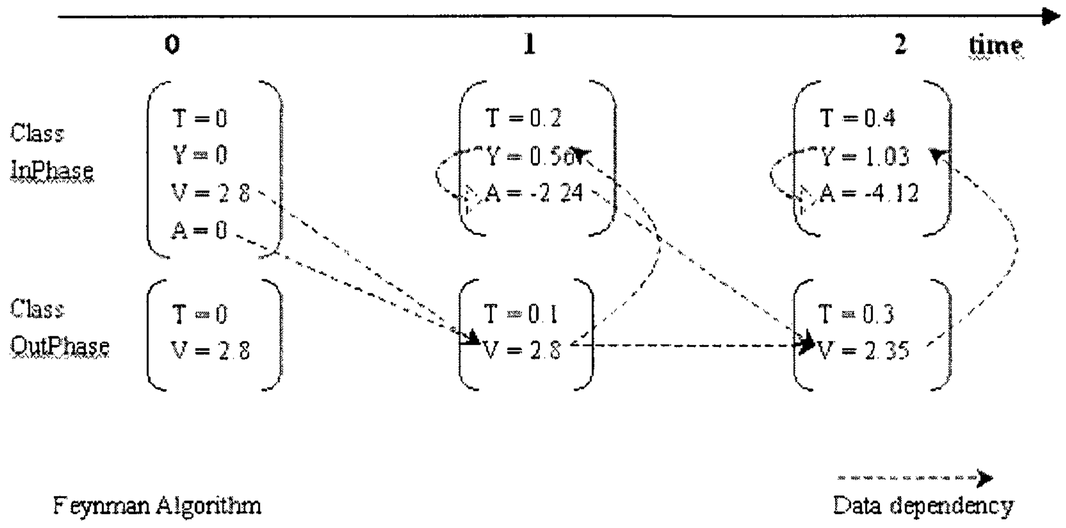


Figure 6 - 6: Solution for the Feynman algorithm with JOOIP

Even, if we keep track of changes in velocity as well as velocity itself and using the last change to project forward a half-step for the acceleration calculation [Phy_Java], we can include another ***V*** stream in class **A** to record the velocity at the same time as ***Y*** and ***A***. This also fixes the "out-of-phase" problem. The program is in Excerpt 6-7.

```
public class InPhase
{
  InPhase  C1 = new InPhase();
  OutPhase C2 = new OutPhase();

  int k = 8;
  int m = 2;

  double T = /@#INDEXICALLUCID 0 fby.time (T + 0.2)
               Where dimension time; end @/;
  double Y = /@#OBJECTIVELUCID 0 fby.time (Y + C2.V @.time #.time
* 0.2) where dimension time; end @/;
  double A = /@#OBJECTIVELUCID 0 fby.time (-k / m) * Y @.time
#.time where dimension time; end @/;

  public void output(double interval, double distance, double
speed, double accel)
  {
    System.out.println("Feynman Algorithm");
    System.out.println("Time = " + interval);
    System.out.println("Position = " + distance);
    System.out.println("Acceleration = " + accel);
    System.out.println("Velocity = " + speed);
  }

  public static void main(String[] argv)
  {
    InPhase oMotionIn = new InPhase();
    OutPhase oMotionOut = new OutPhase();

    double T_value = /@#OBJECTIVELUCID oMotionIn.T @.time 3
                         where dimension time; end @/;
    double Y_value = /@#OBJECTIVELUCID oMotionIn.Y @.time 3
                         where dimension time; end @/;
    double A_value = /@#OBJECTIVELUCID oMotionIn.A @.time 3
                         where dimension time; end @/;
    double V_value = /@#OBJECTIVELUCID oMotionOut.V @.time 3
                         where dimension time; end @/;
    oMotion.output(T_value, Y_value, A_value, V_value);
  }
}

public class OutPhase
{
  InPhase C3 = new InPhase();

  double T = /@#INDEXICALLUCID if (#.time==0) then 0 else (-0.1
fby.time (T+0.2)) fi  where dimension time; end @/;

  double V = /@#OBJECTIVELUCID if ((#.time == 0) || (#.time ==
1)) then 2.8 else V + (C3.A @.time #.time - 1) * 0.2 fi
   where dimension time; end @/;
}
```

Excerpt 6 - 7: Feynman Algorithm in JOOIP

Figure 6-6 shows the reason why we need two classes, **InPhase** and **OutPhase,** it is because the distance **Y,** acceleration **A** and velocity **V** do not change at the same time. The two former change at time = 0.2, while the latter changes at time = 0.1. Moreover, acceleration **A** and velocity **V** depend on each other even though they don't change with the same pace. Let us go back to check the program, Identifiers **T**, **Y** and **A** in class **InPhase** are three intensional streams which stand for time interval, distance and acceleration; identifiers **T** and **V** in class **OutPhase** are two intensional streams which stand for time interval and velocity. The dependencies among these identifiers are shown in Figure 6-6, for context **[time: t]**, distance **Y** in **InPhase** depends on the velocity **V** in **OutPhase** at the same context; acceleration **A** in **InPhase** depends on distance **Y** in **InPhase** at the same context; velocity **V** in **OutPhase** depends on the acceleration **A** in **InPhase** at context **[time: t-1]**.

The program starts with the **main()** method. The run-time engine will generate three demands in class **InPhase,** they are **eval([time:3],T)**, **eval([time:3],Y)** and **eval([time:3],A)** and one demand in class **OutPhase** which is **eval([time:3],V)**. The definition of **T, Y, A** and **V** are already stored in the corresponding GEER. By object access and intensional operator, the program can easily get correct values while not like traditional sequential programs. The method **output(...)** will display the result on the screen. We find the following advantages by using JOOIP for this program:

1. A stream could be infinite, and is in fact defining the intension of the series of values, as opposed to the Java program that computes the *extension* of

a portion of the infinite stream. With the intensional description available, we can ask kinematics of the motion at any time or interval of time in this intensional definition, yielding an extensional infinite portion of the infinite stream;

2. It fixes the "out-of-phase" problem and can be used in the velocity-dependent case;
3. The potential parallelism in the computation is exploited by the eductive model of computation as implemented in the GEE and GIPSY;
4. Intensional objects are organized as a group;
5. Redundant computation is avoided by an intensional value warehouse;
6. The same intensional description can be used to compute values in any context by changing the initial demand;
7. It provides Lucid with richer input/output capabilities provided by the Java counterpart;

Figure 6-6 shows the fact that we move the class **OutPhase** half step back on time dimension to get same context value for both classes. That means if we ask for context [time:1], the exact time interval for class **InPhase** is 0.2 and the exact time interval for class **outPhase** is 0.1. The time here is relative instead of absolute. Non-sequence of JOOIP makes the implementation possible.

6.5.3 Application on CVS

In Section 6.4.4, we presented an example about the versioning problem to explain the objects relationship in JOOIP. This also brings an application in CVS which is used to record file versions and their changes. As the code in Excerpt 6-

8, there is a stream **oVersion** which contains different file versions in CVS, each version is an object of class **version.** Using JOOIP, we can easily choose a version by indicating context **[time: tag]**.

```
public class version
{
    public int iNum = 0;
    Public int iMaxVersion = 5;
    public String sAuthor;
    public String sChange;
    public Vector oVersionStore = new Vector();

    version oInitVersion = new version();

    private version oVersion = /@#OBJECTIVELUCID
            oInitVersion.iNum fby.time oInitVersion.next();
            where
                    dimension time;
            end@/.getVersionContent();

    public version()
    {
        this.iNum = iNum;
        this.sAuthor = sAuthor;
        This.sChange = sChange;
    }

    int next()
    {
        int t = iNum + 1;
        return t;
    }

    version getVersionContent()
    {
      version oTempVersion

      if (iNum > iMaxVersion)
        System.out.println(("The requested version is not
exit! The newest version is:" + iMaxVersion);
      else
      {
        oTempVersion.iNum = iNum;
        oTempVersion.sAuthor
            = oVersionStore.ElementAt(iNum).sAuthor;
        oTempVersion.sChange
            = oVersionStore.ElementAt(iNum).sChange;
      }

      return oTempVersion;
    }
    public void print()
    {
      System.out.println(("version is " + oVersion.iNum);
      System.out.println(("Author is " + oVersion.sAuthor);
      System.out.println(("Change is " + oVersion.sChange);

    }

    public static void main(String[] argv)
    {

           /@#OBJECTIVELUCID oVersion @.time 2
               where dimension time; end @/.print();
    }
}
```

Excerpt 6 - 8: CVS example - version.jooip

The program will start with the Java **main()** method, according to semantic translation rules, the Lucid expression embedded in **main()** will call method **GEE.eval([time:2],oVersion),** according to the definition of **oVersion**, the embedded Lucid expression will generate a new method call **GEE.eval([time:2], oInitVersion.iNum)**. The run-time engine will drive demand after demand until reach the correct context **[time=2]**. In the **getVersionContent()** method, the program will judge if the version is already there or not; if yes, it will return the version content; otherwise, the program will report an error. We notice that the program is easily extended by adding functions to allow users creating a new version if exceeding the maximal version number. After getting the correct version content, method **print()** will print all content.

This example is only a sketch of CVS application; however it demonstrates the point that we can use any class member which could be expressed by IPL operators to organize a stream's elements which are objects of the class. This also could be used to many applications which are time-oriented, like Web management and search engine design.

6.5.4 Application on Accounting – Inheritance

In this example, there are two classes: **InterestBearingAccount.java** and **Account.jooip**. **InterestBearingAccount.java** is a pure Java class and it inherits from **Account.jooip** which is written in JOOIP.

In Excerpt 6-9, a stream **InterestBaseNumber** is defined in **Account.jooip** which varies on dimension **m**. The method **Cal_interest()** will get the number of months for interest calculation which can be used directly in class **InterestBearingAccount**.

```
package gipsy.tests.jooip;

public class InterestBearingAccount extends Account
{
      private static double default_interest = 7.95;
      private double interest_rate;
      private double month_number;

      public InterestBearingAccount()
      {
            balance = 0.0;
            interest_rate = default_interest;
      }

      public InterestBearingAccount( double amount, double interest)
      {
            balance = amount;
            interest_rate = interest;
      }

      public InterestBearingAccount( double amount )
      {
            balance = amount;
            interest_rate = default_interest;
      }

      public static void main(String args[])
      {
            InterestBearingAccount my_account = new
                              InterestBearingAccount();
            my_account.deposit(250.00);

            System.out.println ("Current balance " +
                                 my_account.getbalance());

            my_account.withdraw(80.00);

            System.out.println ("Remaining balance " +
                                 my_account.getbalance());

            my_account.month_number = my_account.Cal_interest(3);
            my_account.balance = my_account. balance +
     (my_account.month_number * my_account.interest_rate / 100) / 12;

            System.out.println("Remaining balance " +
                                my_account.getbalance());

      }
}
```

```
public class Account
{
      protected double balance = 0.0;

      private double InterestBaseNumber = /@#GIPL
      if (#.m) <=1 then 0 else (InterestBaseNumber+1)@.m (#.m)-1 fi
      where
            dimension m;
      end @/;


      public Account( double amount )
      {
            balance = amount;
      }

      public void deposit( double amount )
      {
            balance += amount;
      }

      public double withdraw( double amount )
      {
            if (balance >= amount)
            {
                  balance -= amount;
                  return amount;
            }
            else
                  return 0.0;
      }

      double Cal_interest(Integer month)
      {
            double BaseNumber;
            BaseNumber = /@ InterestBaseNumber@.m month
                            where dimension m; end @/;
            return BaseNumber;
      }

      public double getbalance()
      {
            return balance;
      }
}
```

Excerpt 6 - 9: Accounting example - Account.jooip

The program will start with the Java **main()** method which will call **Cal_interest(Integer month)** method. According to the semantic translation process, the Lucid expression embedded in **Cal_interest(Integer month)** will be changed to method calls, **GEE.eval([m:3], InterestBaseNumber)**. The run-time engine will drive a demand to calculate the value of the third element of the stream **InterestBaseNumber**. The definition of **InterestBaseNumber** is already stored in the corresponding GEER. According to the definition of **InterestBaseNumber**, after executing, engine returns the value 2.

This example is simple but it shows how to use inheritance which is an important feature in objected oriented programming. It is already translated to pure **Account.java** by JOOIP compiler correctly as in Excerpt 6 -10.

```
package gipsy.tests.jooip;
import gipsy.GIPC.GIPC;
import gipsy.lang.*;
import gipsy.lang.converters.type.*;
import gipsy.interfaces.*;
import gipsy.lang.context.Dimension;
import gipsy.GEE.GEE;
import gipsy.util.*;
import java.lang.reflect.Method;

public class Account  implements ISequentialThread
{
      private static GIPSYProgram soGEER1;
      private static GIPSYProgram soGEER2;
      private double InterestBaseNumber = 0;
```

```
	private boolean bInterestBaseNumberIsWritten = false;
	private GIPSYContext oContext;

	public GEE oGEE2 = new GEE(soGEER2);
	public GEE oGEE1 = new GEE(soGEER1);

	static{
		try
		{
			GIPC oGIPC1 = new GIPC(new
StringInputStream("InterestBaseNumber where InterestBaseNumber =
if (#.m) <=1 then 0 else InterestBaseNumber@.m (#.m) - 1 fi
where dimension m; end ; end"), new String[] {"--gipl", "--debug"});
			oGIPC1.compile();
			soGEER1 = oGIPC1.getGEER();
			GIPC oGIPC2 = new GIPC(new StringInputStream("
InterestBaseNumber@.m month where dimension m; end "), new String[]
{"--gipl", "--debug"});
			oGIPC2.compile();
			soGEER2 = oGIPC2.getGEER();
		}
		catch(Exception e)
		{
			System.err.println(e);
			e.printStackTrace(System.err);
		}
	}

	public Account(GIPSYContext poContext)
	{
		this.oContext = poContext;
	}

	public Account()
	{
	}

	public WorkResult work()
	{
		Account.main(null);
		return null;
	}
	@Override
	public WorkResult getWorkResult()
	{
		return null;
	}

	@Override
	public void setMethod(Method poSTMethod)
	{
	}

	@Override
	public void run()
	{
		work();
```

```
        }

        double balance = 0.0;


        public Account( double amount )
        {
              balance = amount;
        }

        public void deposit( double amount )
        {
              balance += amount;
        }

        public double withdraw( double amount )
        {
              if (balance >= amount)
              {
                    balance -= amount;
                    return amount;
              }
              else
                    return 0.0;
        }

        double Cal_interest(Integer month)
        {
              double BaseNumber;

              GIPSYContext oContext2 = new GIPSYContext();
              Dimension oDimension2 = new Dimension();
              oDimension2.setDimensionName(new GIPSYIdentifier("m"));
              oDimension2.setCurrentTag(new GIPSYInteger(month));
              oContext2.addDimension(oDimension2);

              BaseNumber =
              IPLToJava.convertToInteger(this.oGEE2.eval(oContext2));

              return BaseNumber;
        }

        public double getbalance()
        {
              return balance;
        }
}
```

Excerpt 6 - 10: Translated Accounting example

6.5.5 Application on Satellite Tracking – Infinite Stream Expression in Java

Excerpt 6-11 shows an example about the movement of a satellite. Suppose there is a satellite that moves around the earth, it will stay 15 minutes on a zone, for total 24 zones around the earth, it will take the satellite 6 hours to finish one orbit. Suppose the satellite starts from zone number 1, with any time we get, we will know which zone the satellite is on. The satellite will not stop moving in this case, it is hard to express this infinite situation in traditional Java. Here in **Satellite.jooip**, the stream **CurrentPosition** easily expresses this case.

```
class Satellite
{
    public Integer iZone = 1;
    public Integer timer = 15;
    public Integer zoner = 24;
    public String sZoneName = iZong.toString();

    Satellite statu = new Satellite();

    private String CurrentPosition = /@#OBJECTIVELUCID
                                      statu.sZoneName fby.t
                                      statu.next();
                                      where
                                            dimension t;
                                      end@/;

    int num;

    public Satellite()
    {
        this.iZone = iZone;
        this.timer = timer;
        this.zoner = zoner;
    }

    String next()
    {
        int t = timer - 1;
        int position = iZone;

        String CPosition = "";

        if(t <= 0)
        {
            t = 15;
            iZone ++;
            position = iZone % 24;
        }

        timer = t;

        CPosition = position.toString();

        return CPosition;
    }

    public void print()
    {
        System.out.println(CurrentPosition);
    }

    public static void main(String[] argv)
    {
        for (num=0; num<100; num++) /@#GIPL
                        CurrentPosition@[t:num] @/.print();
    }
}
```

Excerpt 6 - 11: satellite example - satellite.jooip

The program will start with the Java **main()** method which will ask the value of elements of a stream **CurrentPosition** from the 1st element to 100th element. According to the semantic translation process, the Lucid expression will be changed to method calls, **GEE.eval([t:num],CurrentPosition)**. Because the satellite will stay a zone for 15 minutes, so from the 1st element to the 15th element of the stream **CurrentPosition**, the return value will be zone 1. From the 16th element to 30th element of the stream **CurrentPosition**, the return value will be zone 2. The method **print()** will display the result on the screen.

6.5.6 Application on Geography – Context Driven Computation

In Excerpt 6-12, an IPL variable **Area** varies on two dimensions: longitude and latitude, the stream **Area** shows the way how the **Area** gets enlarged – by size of a grid or by size of a strip. Figure 6-7 shows the intuitive expression:

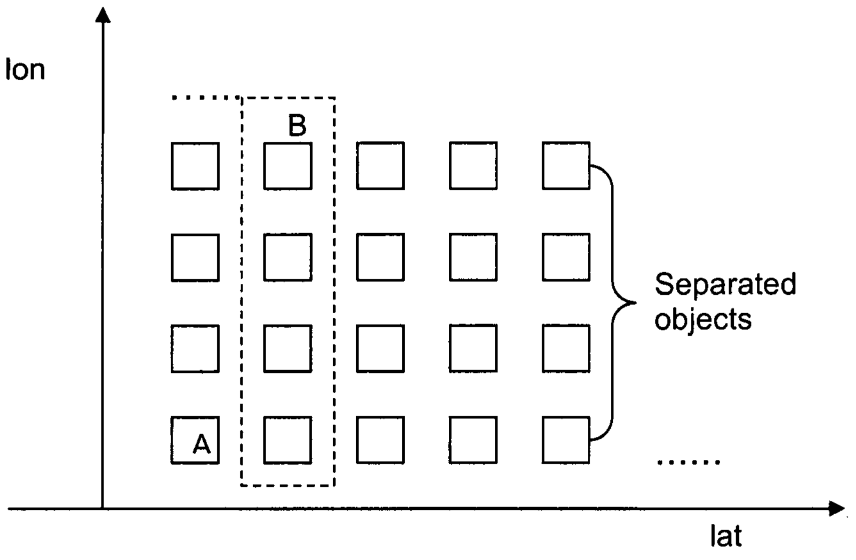


Figure 6 - 7: Intuitive expression of above IPL class

In Figure 6-7, when the context is [lon=1, lat=1], the size of Area will be enlarged by adding the size of a grid (A). When the context is [lon= Φ, lat=2], dimension longitude is missing which means objects only vary on dimension latitude and keep the same on dimension longitude. However, for the reality in this example, the longitude can not be infinite, so we define range in the dimension declaration according to the real case on the earth which longitude is from 0 to 360 degree (from east to west) and the latitude is from -90 to 90 degree (from south to north). Then, the size of Area will be enlarged by adding the size of a strip, like the object B in the dash line area.

In this case, dimensions need to be defined in a range to match the reality, the new introduced Lucid dialect, Lucx, exactly meets this requirement. According to [Wan06], the syntax of a dimension definition will be "dimension A [x, y]" which means identifier A is a dimension and its range is between x and y. In this example, we also show how to use the context driven concept in JOOIP.

```
public class Geo
{
      public Integer iLatArea = 360;
      public Integer iLonArea = 180;
      public Integer iGrid = 1;
      public String oDimension ="";

      public Geo oInformation = Geo();
      public Integer iAreaExtend = 0;

      private Integer Area =
                        /@#Lucx 0 fby [.lat][.lon] Area +
iAreaExtend
                        where
                              dimension lat [-90,90];
                              dimension lon [0,360];
                        end @/;


    public Geo getInfo(int iLatitude, int iLongitude)
    {
       boolean bHasLat = true;
       boolean bHasLon = true;

       if ((iLatitude<-90)||(iLatitude>90))
       {
            bHasLat = false;
       }

       if ((iLongitude<0)||(iLongitude>360))
       {
            bHasLon = false
       }

       if ((bHasLat)&&(bHasLon))
       {
            iAreaExtend = iGrid;
            oDimension = "Latitude and Longitude dimensions.";
            return(/@#Lucx Area@[lat:iLatitude][lon:iLongitude]
@/);
       }
       else if ((!bHasLat)&&(bHasLon))
       {
            iAreaExtend = iLonArea;
            oDimension = "Longitude dimension.";;
            return(/@#Lucx Area@[lat:-90..90][lon:iLongitude] @/);
       }
       else if ((bHasLat)&&(!bHasLon))
       {
            iAreaExtend = iLatArea;
            oDimension = "Latitude dimension.";
            return(/@#Lucx Area@[lat:iLatitude][lon:0..360] @/);
        }
        else
        {
            oDimension = "constant.";
            return(/@#Lucx Area@[lat:-90..90][lon:0..360] @/);
        }
    }

    public void print()
    {
        System.out.println("Currently the stream varies on " +
oInformation.iDimension);
        System.out.println("The Area is " + oInformation.Area);
    }

    public static void main(String[] argv)
    {
        Geo oTest = new Geo();
        oTest.oInformation = oTest.getInfo(2,3);
        oTest.print();
    }
}
```

Excerpt 6 - 12: Geography example - Geo.jooip

The program will start with the Java **main()** method which will call **getInfo(int iLatitude, int iLongitude)** method. According to the semantic translation process, the Lucid expression embedded in **getInfo(int iLatitude, int iLongitude)** will be changed to method calls, **GEE.eval([lat: iLatitude, lon: iLongitude], Area)**. The run-time engine will generate six demands to calculate the corresponding value of the stream **Area**. The definition of **Area** is already stored in the corresponding GEER. According to the definition of **Area**, after executing, the engine returns an **Area** value that is enlarged by adding the size of grid of **A** times 6. The method **print()** will display the result on the screen.

All examples in Section 6.5 show different features of JOOIP, they also show the easiness to integrate different Lucid dialects, for example IndexicalLucid, ObjectiveLucid and Lucx to Java class. Currently, our GIPSY system can execute JOOIP correctly which only mix GIPL and Java. Regarding other Lucid dialect mixture, there is still work needed to be done.

6.6 Summary

As research effort, the hybrid language provides a platform for cases which uses hybrid concepts. However, it is still at an intermediate stage. The translation process presented here will limit certain qualities of our solution, e.g. the resulting implementation would inevitably be less efficient than if directly translated into object code. Moreover, the intensional variable definition in the JOOIP only can be used when the relationship of neighbours can be expressed by IP operators. The application domain will be affected by this limit. However, we have to keep in mind, that this work's main goal is language development and a proof-of-concept of the developed language.

By integrating Lucid and Java, we combine the essential characteristics of object-oriented languages with the basic elements of Lucid. We make it possible that each element in a stream could be an object. By this new point, we extend the use of objects and enrich the meaning of a stream in Lucid, which can greatly increase the power of Lucid. The hybrid OO-IP approach proposed here and adopted by JOOIP is enabling the novel concept of intensional member and intensional class.

There is much research work done on similar topics. In Chapter 7, we will discuss related work on combination between OO and intensional programming languages.

Chapter 7 : Related Work on OO-IP Hybrid Languages

7.1 Introduction

Object-Oriented Languages are quite popular and they are used extensively in industry; even though it is not completely type-safe and need extensive runtime testing and debugging. On the contrary, functional languages are completely type safe and the key property of referential transparency ensures that the encapsulation cannot be breached; however, it has low industry usage base. As one of functional languages, intensional programming language (IPL) is particularly suited for programming dynamic systems whose state varies in one or more dimensions. Unfortunately, not many people from industry are familiar with this language.

The two paradigms used in this work have a generally poor interface among each other: on the one hand are conventional imperative programming languages that have no room for multidimensionality or intensional or demand-driven evaluation; on the other hand, existing multidimensional languages that cannot take advantage of imperative features and techniques. Developed over years of research, the combination typically results in much better performance. The following solutions are typical combination in this domain.

7.2 GLU$^{\#}$ - Intensional Langauge and C++

In Chapter 2, we mentioned the GLU system which was the first try on combination between imperative language and intensional language [JD96]. It was a usable and efficient solution for the parallelization of sequential programs; however, it died due to a lack of flexibility and adaptability. GLU# is a small subset of GLU. Its approach embeds a small multidimensional core (GLU#) in a mainstream object-oriented programming language (C++) [PK04]. By this way, without changing the syntax and semantics of the host language, multidimensionality can be supported and constitutes the core of multidimensional features. In addition, it encompasses a lazy expression language with two basic data types (real and Boolean) and a primitive language of recursive definitions. It can be considered as a language orthogonal to C++ and is implemented as a collection of C++ classes and class templates. The syntax and semantics of C++ remain unchanged. Programmers, however, are able to use multidimensional objects which can take the form of lazy arrays and lazy functions.

7.2.1 Introduction

According to [PK04], a program (*P*) in GLU# is a sequence of definitions (*D*) followed by an expression (*E*) that must be evaluated. However, GLU# does not support functions directly. Instead, functions are to be defined in C++.

On implementation, programmers need to include the header file **glu.hpp** in their C++ source code. Programmers may declare new dimensions by creating new instances of the class dimension. A multidimensional object is represented as an instance of the class **GLU<T>**, where T is the type of the object's extensions. GLU# distinguishes between *wrapper objects* and *implementation objects*, following a model known in object-oriented development as the letter/envelope idiom [Cop92]. The objects that are frequently copied are wrappers, which contain pointers to implementations and whose copying is inexpensive. Implementations are seldom copied. The programmer is not allowed to use arbitrary C++ code to alter their values.

GLU# uses the dimensionality analysis (rank analysis) to improve the overall performance. Moreover, a *warehouse* of evaluated expressions is equipped in GLU#. Values of arbitrary *expressions*, including their sub-expressions are stored in the warehouse instead of only storing values of *variables*. The hashing function uses the identifier of *E* and a combination of the indices in *X*; when a possible match is found in the hash-table, it must be determined whether *X* is really a sub-world of *w*. Regard to garbage collection, priority is a function of age, hits and effort required to re-compute a value.

GLU# provides a bridge between IPL and OO. However, it just includes the basic features of IPL and embeds into C++. Moreover, GLU# is implemented as an interpreter embedded in C++, with objects modeling both values and expressions that have not yet been evaluated. In contrast to C++ which supports mutable variables and objects, multidimensional objects in GLU# are immutable.

7.2.2 Comparable examples in JOOIP

To show the similarities and differences, we provide the translation of some of the examples given in [PK04] into JOOIP for the comparison reasons and to show how our approach is more general, adaptive, and flexible than that of GLU#. Excerpt 7-1 shows the Prime example and Excerpt 7-2 shows the Hamming program.

```
class Prime
{
  private int prime = /@#INDEXICALLUCID
                first.x sieve
                where
                   dimension x, y;
                   sieve = ints fby.y (sieve wvr.x sieve % prime != 0);
                   where
                       ints = 2 fby.x ints + 1;
                   end
                end@/;

  int num;

  public void print()
  {
        System.out.println(prime);
  }

  public static void main(String[] argv)
  {
        for (num=0; num<100; num++) /@ prime@.x num
                                  where dimension x; end @/.print();
  }
}
```

Excerpt 7 - 1: Prime.jooip – Sieve of Eratosthenes in JOOIP

```
public class Hamming
{
    private int H = /@#INDEXICALLUCID
                        1 fby .d  if (xx<=yy) then xx else yy fi
                        where
                              dimension d;
                              xx = 2*H  upon .d  (xx<=yy);
                              yy = 3*H  upon .d  (yy<=xx);
                        end @/;

    int num;

    public void print()
    {
        System.out.println(H);
    }

    public static void main(String[] argv)
    {
        for (num=0; num<100; num++) /@ H@.d num
                                     where dimension d; end @/.print();
    }
}
```

Excerpt 7 - 2: Hamming.jooip – Hamming example in JOOIP

It is very easy for Lucid programmers to integrate Lucid code in a Java class in JOOIP, because Lucid code keep the same nature. You do not need to change Lucid expressions into function format which is the case in GLU#. These two examples are very similar to the nature example in Section 6.5.1; the only difference is that the definition of stream **prime** and **H** are written in Indexical Lucid, the remaining process are the same as the natural number example which already works in the GIPSY system. Excerpt 7-3 shows the traffic light example.

```
class TrafficLight
{
    String[] StateName = new String[]{"GREEN", "YELLOW", "RED"};
    int[] timePerLight = new int[]{5, 1, 8};

    int getState(String statename)
    {
        if (statename.equals("GREEN")) return 0;
        else if (statename.equals("YELLOW")) return 1;
        else if (statename.equals("RED")) return 2;
    }

    public String state = "RED";
    public int timer = 8;

    TrafficLight statu = new TrafficLight();

    private String light = /@#OBJECTIVELUCID
                              statu.state fby.t statu.next()
                              where
                                  dimension t;
                              end@/;

    int num;

    public TrafficLight()
    {
        this.state = state;
        this.timer = timer;
    }

    String next()
    {
        int t = timer - 1;
        int position;
        String LightColor = "";

        if(t <= 0)
        {
            position = (getState(state)+1) % 3;
            t = timePerLight[position];
            state = StateName[position];
        }

        timer = t;
        LightColor = state;

        return LightColor;
    }

    public void print()
    {
        System.out.println(light);
    }

    public static void main(String[] argv)
    {
        for (num=0; num<100; num++) /@ light@.t num
                                  where dimension t; end @/.print();
    }
}
```

Excerpt 7 - 3: TrafficLight.jooip – Traffic Light example in JOOIP

For the traffic light example, the JOOIP code is a lot shorter and simpler than GLU#'s. Functions are not allowed in GLU#, any time there is need for functions, programmers have to add C++'s templates. Because of the flexibility of our existing framework, it is very easy for us to extend Lucid code to any Lucid dialect, which is not the case in GLU#. The most important, OBJECTIVELUCID makes it is possible for Lucid code to access identifiers and methods of Java class by introducing the object-oriented dot operator, this also helps JOOIP to become a 2-way hybrid intensional-imperative programming language. This is also not the case in GLU#.

7.3 Embedding IPL as lazy multidimensional arrays

A similar embedding of multidimensional characteristics in a conventional programming language has been proposed by Rondogiannis [Ron99]. In his approach, Java is used as the host language and intensional languages are embedded into Java as a form of definitional lazy multidimensional arrays. The introduction of lazy arrays into conventional programming languages is very useful in cases where only a small fragment of the elements of an array are needed in order to compute a desired result.

The basic idea of this approach is that the data definition section of a conventional program can be extended to include a multidimensional lazy array definition part. The definitions of this part are in fact definitions of a multidimensional program and can be used in the remaining conventional part.

This approach tries to introduce multidimensional lazy array into Java and let two different paradigms benefit each other. However, compared to [Ron99], there are still many limitations. We will provide JOOIP code from the same **multi** program as in Excerpt 7-4 to show the difference.

This approach:

```
public class multi {
        /* The multidimensional array definition section */
        [[
            dimensions x, y;
            p = 1 fby_x (0 fby_y (next_y(p)+next_x(p))/2);
        ]]
        public static void main(String [] argv){
             System.out.println(p[100][100]);
         }
}
```

JOOIP:

```
public class multi
{
    private int P = /@#INDEXICALLUCID
                     1 fby.x (0 fby.y (next.y(p)+next.x(p))/2)
                     where
                        dimension x,y;
                     end @/;

    public void print(f)
    {
        System.out.println(/@LUCX P @[x:f][y:f] @/);
    }

    public static void main(String[] argv)
    {
        GIPLtest oTest = new multi();
        oTest.print(2);
    }
}
```

Excerpt 7 - 4: Comparison on the same multi program

From Excerpt 7-4, we can tell:

1. The embedded language supports only a subset of the dimensional operators of GLU, the user-defined functions and the nested **where**

clauses are not allowed in the language. Different Lucid dialects, e.g. #INDEXICALLUCID and #LUCX can be used in JOOIP.

2. The value of a specific element of a lazy array can not be altered by a procedural program. In Excerpt 7-4, we pass **f** as parameter to be the tag of context **[x:f][y:f]** of stream **P**. This parameter pass as tag solution is already implemented in Natural number example in Section 6.5.1.
3. Only *one multidimensional array definition section is allowed in every* class definition. In JOOIP, we don't limit the number of places where the IPL could appear.

However, this approach concentrates more on implementation efficiency improvement with lazy array evaluation comparing to imperative array instead of language design itself like JOOIP.

7.4 Object-oriented IPL implementation

In [Du94], there is another discussion on issues about object-oriented implementation of intensional languages. In this approach, each variable in a Lucid program is considered as a class and an object of a class is the variable in a context. Each variable definition in a Lucid program is compiled into a C++ class definition which has the same name as the variable.

In the traditional implementation of intensional languages, it receives demands for variable values, checks the value warehouse, fetch and interprets variables definitions, creates more demands, evaluates expressions on the stack machine, store values in the value warehouse, and switch context in the contexts registers. However, in this object-oriented implementation of IPL, a variable in a context is considered as an object which is identified by the variable and the context. All objects of the variable constitute a class named by the variable.

In this implementation model, the evaluation engine, value warehouse and context registers are distributed to individual objects, and context switching means sending messages to other objects with switched contexts. Depending on the dimensionality of a variable, an instance of a variable class has a set of private context members. The value of any context members is not mutable after the object is created.

This approach focuses on implementation level by creating a class for each Lucid variable, it helps the system to execute in a distributed manner. However, the objects introduced here do not contain information from C++ variables, which is provided by JOOIP.

7.5 Introduction of Objects into IPL

The concept about objects in Lucid first appeared in [Fre91] in the early 1990s, however, it did not clearly define how to realize this idea. In the later 1990s, Peter Kropf and John Plaice talk about this topic in their paper "intensional objects" [KP99]. In this paper, intensional objects are considered as open-able boxes labeled by Lucid contexts. This paper focuses on intensional versioning whose task is to build a system from versioned components, which are already sitting in the warehouse. This warehouse is different as the warehouse in intensional programming. The latter is like a cache to improve the performance. The former contains the source of everything, it is like a "catalog" or a "repository", in which the boxes are put. Each box is of some contents and a tag that is context. So, in this approach, these labeled boxes are called intensional objects, which are re-openable and re-packageable. However, in this approach, authors did not clearly define the relationship among these “boxes” and if these boxes could include intensional concept. Moreover, the idea is only on conceptual level.

7.6 Summary

This chapter introduces other hybrid approaches in the same domain. Most systems focus on implementation instead of language itself. All intensional objects concepts stated here are different as what we discussed in Chapter 6. However, what we state in this thesis, the GIPC framework and the OO-IP hybrid language, are still at an intermediate stage. In the next chapter, we will discuss the future work and give conclusion.

Chapter 8 : Conclusion and Future Work

This chapter draws a conclusion with our work on the GIPC framework design and the new OO-IP hybrid language design of JOOIP. It also discusses future work arising from the limitations of this research.

8.1 Conclusion

This thesis develops a framework design based on wide fundamental research; in parallel, based on the characteristics of the Lucid family of languages, a new OO-IP hybrid language is designed which has unique features. Implementation details of this new hybrid language are discussed and preliminary implementation proves that the framework design can adapt to the particularities of this new language, which also testifies our original framework design.

The conclusion is organized in four main parts: fundamental research, framework design, OO-IP hybrid design and implementation; all description also correspond to the stated contributions listed in Section 1.2.

Fundamental research: is the cornerstone of this thesis. It includes the research on IPLs and the investigation on suitable method to realize their implementation. Regarding characteristics analysis on intensional programming paradigm, we did research on:

- IPL's history, syntax and semantics;
- The diversity of languages of Lucid family;
- The diversity of their applications;
- The evolution of Lucid;

We come up with the conclusion that the Lucid family of intensional programming languages evolves and grows very quickly. In this dynamic and highly evolutionary context, the standard methods of compiler generation do not provide enough flexibility. That is the trigger of designing a new flexible system. Regarding investigation on suitable method to realize the design, we did:

- Investigating the application of object technology on infrastructure design;
- Research the features of framework methodology;
- Research the classification of framework methodology;
- Research the application of framework methodology;

All research provides good reasons in favour of adopting an object-oriented framework methodology for the GIPC design.

Dynamic framework design of GIPC: provides a solution compared to all other techniques currently used to implement the different variants of intensional programming. Detailed introduction of the framework is in Chapter 4, as conclusion the main tasks include:

- o Integrate the design with the original GIPSY architecture;
- o Build layers for the framework;
- o Define functionalities for separate layers in the framework;
- o Introduce hot spots automatic generation in the framework design;
- o Separate front and back end of framework;

Layer design bring easy maintenance for the framework, the black-box framework keep the back end of the compiler untouched by the addition of new component instances in the front end. Any eventual change in the back end will be shielded to the users. Automated generation units are hot spot generators that generate different components of the framework, which can then be automatically linked to the framework to provide new capacities to the system.

Compared to existing compiler construction systems (detailed comparison presented in Chapter 5), the GIPC design has better flexibility and extensibility than the Centaur, FNC-2, Eli and GLU systems, as it resolves new SIPLs introduction in one stable system. Furthermore, framework features as well as visual programming support make that GIPSY has high usability. Moreover, GIPSY supports any Lucid flavour programs with Java functions, and it can be easily extended to other sequential threads, for example, C++ function, Pascal function, etc; other systems cannot provide such functionality. Finally, hot spot automated generation makes the GIPSY framework extremely extensible.

OO-IP hybrid language design: By integrating Lucid and Java, we combine the essential characteristics of object-oriented languages with the basic elements of Lucid. We make it possible that each element in a stream could be an object. By this new point, we extend the use of objects and enrich the meaning of a stream in Lucid, which can increase the power of Lucid. Detailed constructions include:

- Introduce intensionality into the Java language;
- Concepts of *context-mutable object* and *context-immutable object;*
- Formally define the syntax of the hybrid language;
- Formally define the semantics of the hybrid language;
- Application discussion using various examples;

Compared to other hybrid programming systems (detailed comparison are in Chapter 7), we provide a new OO-IP language design instead of only focusing on implementation. Intensional objects can reflect the relationship among Java objects. The hybrid OO-IP approach proposed here and adopted by JOOIP is enabling the novel concept of *intensional member* and *intensional class* into object-oriented programming.

Implementation-specific details: define the data type mapping system in the GIPC framework for more generic situations. Our current implementation enables us to realize the power of the framework, as well as to testify that our design suits the tendency of evolution of intensional programming and it achieves our original aims at flexibility, generality and adaptability.

This thesis presents the research work on the GIPC framework design, the hybrid language design, the interacting between two issues and implementation specific issues on hybrid programming in the GIPC. Partial work is based on the current research result and makes a progress, for example, the using of framework methodology. The originalities include:

1. Adding hot spots automatic generation in the framework design. Nowadays, the state-of-art of compiler construction is using automated compiler generation tools; however, in our method we use these tools integrated in a framework for the automatic generation of framework hot spots.
2. Raising ideas, context mutable object and context immutable object;
3. Introduce new concept of intensional member and intensional class;
4. Designing the hybrid language which can express the relationship among objects and form objects group for management;
5. Permitting intensionality to exist in Java objects;
6. Enhance Java language with explicit dimensions;
7. Applying it on the Feynman algorithm widely used in physics applications;

The use of the GIPC compiler framework has proven to meet its flexibility promises in this work by facilitating the development of a hybrid language permitting the use of any variant of Lucid to be embedded in Java classes. By the use of this facility, JOOIP is the first language to allow the embedding of different variants of Lucid. Moreover, as soon as a new compiler for a variant of Lucid is added to the GIPC, it is automatically available to JOOIP. This feat proves the validity of the framework approach adopted for the design of the GIPC.

As we were working on the design of JOOIP, it initially appeared that we would have to apply profound changes to our run-time system (GEE). Interestingly, the solution we present here did not require any such change, again proving the generality of our design. Generality and flexibility has been one of our major goals in the design of the GIPSY. This research experience and result provides us with a positive evaluation of these requirements. We can conclude that this research work meet all the goals which were shown in Section 1-1.

8.2 Limitations and Future work

Limitations

There are still some limitations of the JOOIP design.

1. Some important Java features such as exceptions, static members and threads cannot be used in JOOIP. Because of potential distributed execution in the GIPSY system, anything between objects with shareable memory is not supported in JOOIP.
2. Because there are many translation processes in the JOOIP compiler design, the efficiency of execution of the generated code might be affected.
3. The application domain is limited by the fact that objects need to be organized by IPL operators.

In order to make the GIPC framework and the new JOOIP to be used widely in real life, there are still lots of work to be done.

Future work

Framework implementation. There are still components need to be done in the GIPC, for example, CP and ST front end generators which work similar to the other front end generators. After designing each component in the GIPC, the main task is to integrate them together and implement it. We will apply the GIPC framework to generate compiler components for other known IP languages, such as TensorLucid. Then we will carry out some practical experiments using the generated compilers, such as particle in-cell simulation using the Maxwell equations [Paq99], and eventually apply the GIPSY to genomics computations. All these experiments will be made with a constant focus on adapting the framework to reach for a constant increase of flexibility and, later on, efficiency of computation.

Extend the concept to other families of languages' compiler design. We hope to extend this framework design to the general compiler design for families of programming languages. This should be done step by step. For example, first changing mixture language from Java to other language, then changing the original Lucid family languages with other family languages. We will constantly improve the framework and make it more mature.

REFERENCES

[AW76] E.A.Ashcroft and W.W. Wadge. *Lucid —A formal system for writing and proving programs*. SIAM Journal on Computing, pp. 336-354. September 1976.

[AW77] E.A.Ashcroft and W.W. Wadge. *Lucid, a nonprocedural language with iteration.* Communication of the ACM, pp. 519-526. July 1977.

[BCD$_{etc}$88] P. Borras, D. Clement, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, V. Pascual. *CENTAUR: The System.* ACM Sigplan Notices, Vol. 24, No. 2, pp. 14 – 24. 1988.

[BDMN73] G. Birtwistle, O. Dahl, B. Myhrtag and K. Nygaard. *Simula Begin.* Auerbach Press. Philadelphia, 1973.

[BMA97] David Brugali, Giuseppe Menga and Amund Aarsten. *The Framework Life Span.* Communication of the ACM, Vol. 40, pp. 65-68. October 1997.

[Cop92] Coplien JO. *Advanced C++ Programming Styles and Idioms.* Addison-Wesley. 1992.

[Din04] Yi Ming Ding. *Bi-directional translation between dataflow graphs and Lucid programs in the GIPSY Environment.* M.Sc. Thesis, Computer Science Department, Concordia University, Quebec, Canada. June 2004.

[DLM84] V. Donzeau-Gouge, B. Lang, B. Malese. *Practical applications of a syntax directed program manipulation environment.* Proceedings of the 7th international conference on software engineering, pp. 346-354. Orlando, Florida, U.S.A., 1984.

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

[FHLS97] Garry Froehlich, H. James Hoover, Ling Liu, Paul Sorenson. *Hooking into Object-Oriented Application Frameworks.* Proceedings of the 1997 International Conference on Software Engineering, pp. 491-501. Boston, May 1997.

[Fin95] Raphael Finkel. *Advanced Programming Language Design.* Addison Wesley. December 1995.

[FJ91] A.A.Faustini and R.Jagannathan. *Indexical Lucid.* Proceedings of the Fourth International Symposium on Languages for Intensional Programming, pp. 19-34. Menlo Park, California, April 1991.

[Fre91] B. Freeman-Benson. *Lobjcid: Objects in Lucid.* Proceedings of the 1991 Symposium on Lucid and Intensional Programming, pp. 80-87. Menlo Park, CA, April 1991.

[FS97] Mohamed Fayad and Douglas C. Schmidt. *object-oriented Application Frameworks.* Communication of the ACM, Vol. 40, No. 10, pp. 32-38. 1997.

[GHJV94] Erich Gramma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[GHL$_{etc}$92] R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane, W. M. Waite. *Eli: a complete, flexible compiler construction system.* Communications of the ACM, Vol. 35, No. 2, pp. 121-130, 1992.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond and D. Pilaud. *The synchronous dataflow programming language LUSTRE.* Proceedings of the IEEE, Vol. 79, No. 9, Special Issue, pp. 1305-1320. September 1991.

[Hed00] G. Hedin. *Reference attributed grammars*. Informatica, Vol. 24, pp. 301-317, Slovenia, 2000.

[HM03] Gorel Hedin and Eva Magnusson. *JastAdd-an aspect-oriented compiler construction system.* Science of computer programming, Vol. 47, pp. 37-58, 2003.

[Java05] http://www.javapractices.com/Topic29.cjp. Viewed in November, 2005.

[JavaCC] JavaCC, The Java Parser Generator, http://www.metamata.com/. Viewed in 2003.

[JD96] Raganswamy Jagannathan and Chris Dodd. *GLU programmer's guide*. Technical report, SRI International, Menlo Park, California, 1996.

[JDA97] R. Jagannathan, C. Dodd and I. Agi. *GLU: A high-level system for granular data-parallel programming*. Practice and Experience, pp. 63-83, 1997.

[JF88] R. E. Johnson and B. Foote. *Designing reusable classes*. Journal of Object-oriented Programming, Vol. 1, No. 2, pp. 22-35. July 1988.

[Joh97] Ralph E. Johnson. *Frameworks = (Components + Patterns).* Communication of the ACM, Vol. 40, No. 10, pp. 39-42. October 1997.

[JP97] M. Jourdan and D. Parigot. *The FNC-2 System User's Guide and Reference Manual. Release 1.19*. Technical report, INRIA Rocquencourt, 1997.

[JPJ$_{etc}$90] Martin Jourdan, Didier Parigot, Catherine Julie, Olivier Durin and Carole Le Bellec. *Design, implementation and evaluation of the FNC-2 attribute grammar system.* Proceedings of the ACM SIGPLAN'SO Conference on Programming Language Design and Implementation. Vol. 25, No. 6, pp. 209-222. New York, June 1990.

[KLMM83] G. Kahn, B. Lang, B. Malese, E. Marcos. *METAL: a formalism to specify formalisms*. Science of Computer Programming, Vol. 3, pp. 151-188. North-Holland, 1983.

[KP99] Peter Kropf and John Plaice. *Intensional objects*. In the 12th International Symposium on Languages for Intensional Programming, pp. 180-187. Demokrito Institute, Athens, Greece, June 1999.

[Kro99] P.G. Kropf. *Overview of the Web Operating System (WOS) project*. In Proceedings of the 1999 Advanced Simulation Technologies Conference (ASTC 1999), pp. 350-356. San Diego, California, April 1999.

[KW94] U. Kastens and W. M. Waite. *Modularity and reusability in attribute grammars*. Acta Informatica, Vol. 31, No. 7, pp. 601- 627. 1994.

[LM03] Jed Liu and Andrew C. Myers. *JMatch: Abstract iterable pattern matching for Java*. In Proceedings of 5th International Symposium on Practical Aspects of Declarative Languages, pp.110-127. New Orleans, LA, January 2003.

[Lu04] Bo Lu. *Framework for the General Eduction Engine (GEE) in the GIPSY*. Ph.D. Thesis. Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2004.

[Mat99] Michael Mattsson. *Evolution and Composition of Object-Oriented Frameworks*. Ph.D thesis. University of Karlskrona/Ronneby, December 1999.

[MBL97] Andrew C. Myers, Joseph A. Bank and Barbara Liskov. *Parameterized types for Java*. In Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL), pp.132–145. Paris, France, January 1997.

[MC86] E. Morcos-Chounet and A. Conchon. *PPML, a general formalism to specify prettyprinting*. In H.-J Kugler, editor, Information proceedings 86. pp. 583-590. Elsevier Science Publisher, 1986.

[MJW99] O. de Moor, S. Peyton Jones, E. van Wyk. *Aspect-oriented compilers*. In proceedings of the First International Symposium on Generative and Component-based Software Engineering, pp. 121-133. September 1999.

[MLAZ00] Marjan Mernik, Mitja Lenic, E. Avdicausevic, V. Zumer. *Compiler/Interpreter Generator System LISA*. In proceedings of the 33rd Hawaii International Conference on System Sciences, Vol. 8, pp. 8059. January, 2000.

[Mok05] Serguei A. Mokhov. *Towards Hybrid Intensional Programming with JLucid, Objective Lucid, and General Imperative Compiler Framework in the GIPSY*. Master Thesis. Concordia University. October 2005.

[Mos01] Peter D. Mosses. *The varieties of programming languages semantics and their uses.* Lecture notes in computer science, University of Aarhus, Danemark. 2001.

[MP05] Serguei Mokhov and Joey Paquet. *General Imperative Compiler Framework within the GIPSY.* In Proceedings of PLC2005, pp. 36-42. Las Vegas, Nevada, USA. CSREA Press, June 2005.

[MPG05] Serguei Mokhov, Joey Paquet, and Peter Grogono. *Towards JLucid, Lucid with Embedded Java Functions in the GIPSY.* In Proceedings of PLC2005, pp. 15-21. Las Vegas, Nevada, USA. CSREA Press, June 2005.

[MPT07] Serguei A. Mokhov, Joey Paquet, and Xin Tong. *Hybrid Intensional-Imperative Type System Enhanced with Context and Semantic Annotations.* Mathematics in Computer Science, Special Issue on Intensional Programming & Semantics in honour of Bill Wadge on the occasion of his 60th cycle, 2007. (unaccepted)

[Mye99] Andrew C. Myers. *JFlow: Practical mostly-static information flow control.* In Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL), pp. 228–241. San Antonio, TX, January 1999.

[NCM03] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. *Polyglot: An Extensible Compiler Framework for Java.* Proceedings of the 12th International Conference on Compiler Construction, pp. 138-152. Warsaw, Poland, April 2003.

[Ost81] C.B.Ostrum. *The Luthid 1.0 Manual*. Department of Computer Science, University of waterloo, Waterloo, Ontario, Canada. 1981.

[Paq99] Joey Paquet. *Intensional Scientific Programming*. Ph.D. Thesis, Departement d'Informatique, Universite Laval, Quebec, Canada, 1999.

[PGW04] 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). Pp. 164-165. Vancouver, Canada. October 24-28, 2004.

[Phy_Java] http://www.kw.igs.net/~jackord/bp/f1.html. Physics and Java. Viewd in Jan, 2006.

[PK00] Joey Paquet, Peter Kropf. *The GIPSY Architecture*, Proceedings of Distributed Computing on the Web, pp. 144-153. Quebec City, Canada, 2000.

[PK04] Nikolaos S. Papaspyrou and Ioannis T. Kassios. *GLU$^{\#}$ embedded in C++: a marriage between multidimensional and object-oriented programming.* Software Practice and Experience, Vol. 34, pp. 609-630. 2004.

[Plo81] G. D. Plotkin. *A structural Approach to Operational Semantics.* Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981.

[Ploy03] http://www.cs.cornell.edu/Projects/polyglot/. Viewed in 2003.

[PS97] Wolfgang Pree and Hermann Sikora. *Design Patterns for Object-Oriented Software Development*. Proceedings of the 19th international conference on software engineering, pp. 663-664. Boston, Massachusetts, U.S.A., 1997.

[PVP07] Amir Hossein Pourteymour, Emil Vassev, Joey Paquet. *Towards a New Demand-Driven Message-Oriented Middleware in GIPSY*. Proceedings of The 2007 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'07), pp. 91-97. Las Vegas, USA, June, 2007.

[PW93] J. Plaice and W. W. Wadge. *A new approach to version control.* IEEE transactions on Software Engineering, Vol. 3, No. 19, pp. 268-276. 1993.

[Ren02] Chun Lei Ren. *Parsing and Abstract Syntax Tree Generation in the GIPSY System.* Master Thesis, Computer Science Department, Concordia University, Quebec, Canada. September 2002.

[Ron99] P. Rondogiannis. *Adding multidimensionality to procedural programming languages.* Software: Practice and Experience, Vol. 29, No. 13, pp. 1201–1221. November 1999.

[Sch96] Hans Albrecht Schmid. *Design patterns for constructing the hot spots of a manufacturing framework.* J. Object-Oriented Programming, Vol. 9, No. 3, pp. 25-37. June 1996.

[Slo95] Anthony M. Sloane. *An Evaluation of an Automatically Generated Compiler*. ACM Transactions on Programmmg Languages and Systems, Vol. 17, No. 5, pp. 691-703. September 1995

[TPM07] Xin Tong, Joey Paquet, and Serguei A. Mokhov. *Context Calculus in the GIPSY*. Mathematics in Computer Science, Special Issue on Intensional Programming & Semantics in honour of Bill Wadge on the occasion of his 60th cycle, 2007. (unaccepted)

[Vas05] Emil Iordanov Vassev. *General Architecture for Demand Migration in the GIPSY Demand-Driven Execution Engine*. Master Thesis, department of Computer Science and Software Engineering, Concordia University, Montreal, Canada. 2005.

[WA85] W. W. Wadge and E. A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, London, 1985.

[Wan06] Kaiyu Wan. *Lucx : Lucid enriched with Context*. Ph.d Thesis. Concordia University, Montreal, Canada. June 2006.

[WH88] W. M. Waite, V. P. Heuring. *Configuration control in compiler construction*. In International Workshop on Software Version and Configuration Control'88. 1988.

[Wu02] Aihua Wu. *Semantic Analysis and SIPL AST Translator Generation in the GIPSY*. Master Thesis, Department of Computer, Concordia University, Quebec, Canada, December 2002.