Adding Functional Programming into the Holo Language

André Rauber Du Bois[1,2], Jorge Luis Victória Barbosa[1,2], and Cláudio Fernando Resin Geyer[1]

[1] UFRGS-Universidade Federal do Rio Grande do Sul,
Instituto de Informática, Programa de Pós-Graduação em Computação (PPGC)
Caixa Postal 15064, 91501-900, Porto Alegre, RS, Brazil
{dubois,barbosa,geyer}@inf.ufrgs.br
[2] UCPel-Universidade Católica de Pelotas
Escola de Informática
Rua Félix da Cunha 412, 96010-000, Pelotas, RS, Brazil
{dubois,barbosa}@atlas.ucpel.tche.br

Abstract. Holo is a new multiparadigm language for the development of distributed systems. This language is based on concepts of concurrency, blackboards and mobility. Besides that, Holo integrates aspects of logic, imperative and object oriented paradigms. This paper presents some ideas on how we can add functional programming into the Holo Language and how HoloJava, a compiler that translates Holo into Java, can be modified in order to support functional programming.

KEYWORDS: Multiparadigm, Implementation of Functional Languages in Java.

1 Introduction

In the last years the multiparadigm theme has been continually researched [2, 3, 5, 15, 23]. Researchers have proposed models of software development through the integration of basic paradigms (mainly: imperative, logic, functional, and object oriented paradigms). Using this approach, they have been looking for two goals: (a) to overcome the specific limitations of each paradigm and (b) to take advantage of the most useful characteristics of each one through their combination.

Each paradigm has sources of implicit parallelism. For example, the exploitation of AND parallelism and OR parallelism in logic programming. Another example is object-oriented paradigm that allows the exploitation of inter-object parallelism and intra-object parallelism. The multiparadigm approach integrates paradigms. Thus, it also integrates their parallelism sources. In this context, interest in automatic exploitation of parallelism in multiparadigm software has emerged. Enlargement of this approach guides the studies to distributed systems where mobility, heterogeneous hardware and the use of networks as parallel architectures are considered. The development of distributed software using multiparadigm models has received attention of the scientific community [5, 10, 11, 26].

In this context, we are creating the Holoparadigm (Holo). Holo is a multiparadigm model oriented to automatic exploitation of parallelism and distribution. Holo has a coordination model based on levels of blackboards (called histories) encapsulated in new programming entities called beings. A new language (Hololanguage [5]) implements the concepts proposed by the Holoparadigm.

In this paper we propose some modifications in the Holo Language in order to support funcitonal programming. We also explain how we can modify the HoloJava compiler, a compiler that translates Holo into Java, so it can support functional programming.

The paper is organized as follow. Section two presents the main concepts of Holoparadigm and describes the principles of Hololanguage. The third section explains how the HoloJava compiler is implemented. Next we present some modifications in the Holo language and in the HoloJava tool, in order to support functional programming. Finally, we present some conclusions and future works.

2 Holoparadigm and Hololanguage

Being is the main Holoparadigm abstraction. There are two kinds of beings: elementary being (atomic being without composition levels) and composed being (being composed by other beings). An elementary being (figure 1a) is organized in three parts: interface, behavior and history. The interface describes relations between beings. The behavior contains actions, which implement functionalities. The history is a shared storage space in a being. A composed being (figure 1b) has the same organization, but may be composed by others beings (component beings).

Each being has its history. The history is encapsulated in the being. In the composed being, the history is shared by the component beings. Therefore, it is possible to exist several levels of encapsulated history. A being uses the history in a specific composition level. For example, figure 1c shows two levels of encapsulated history in a being with three composition levels. Behavior and interface parts are omitted for simplicity.

Automatic distribution exploitation is one of the main Holoparadigm goals. Figure 2b exemplifies a possible distribution of the being presented in figure 1c. The being is distributed in two nodes of the distributed architecture. The history of a distributed being is called distributed history. This kind of history can be implemented using DSM techniques or distributed shared spaces [11].

Mobility is the dislocation capacity of a being. In Holo, there are two kinds of mobility: logical mobility (being is moved when crosses one or more borders of beings) and physical mobility (dislocation between nodes of distributed architectures).

Figure 2a shows a logical mobility in the being presented in figure 1c. After the dislocation, the moveable being is unable to contact the history of the source being. On the other hand, now the being is able to use the history of the destiny

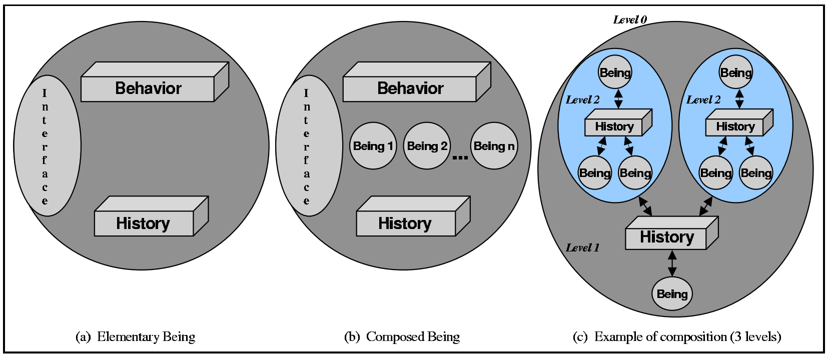

Fig. 1. kinds of Beings

being. Here, physical mobility occurs if the source and destiny beings are in different nodes of the distributed architecture. Logical and physical mobilities are independent. Occurrence of one does not imply in the occurrence of the other. For example, figure 2b shows a physical mobility without logical mobility. In this example, the moved being does not change its history view. This kind of situation could happen if the execution environment aims to speedup execution through locality exploitation.

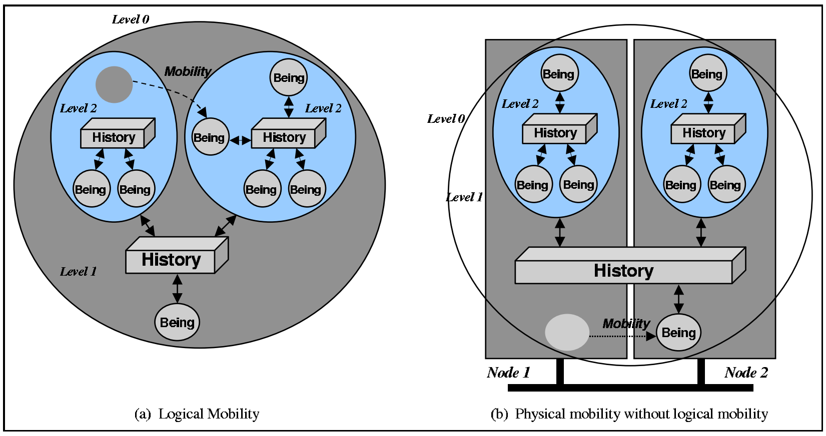

Fig. 2. Mobility

The coordination model used in Holo is based on the blackboard architecture (figure 3a) [24]. This architecture is composed by a common data area (blackboard) shared by a collection of programming entities called knowledge sources (KSs). Control is implicit in the blackboard access operations. The read and write operations in the blackboard are used for communication and synchronization between KSs. This kind of control is called implicit invocation. A composed being architecture is similar to the blackboard architecture, since several components sharing a common data area. In Holo, KSs are beings and the blackboard is the history. Therefore, the history is not only a database. Part of a being control uses the history (implicit invocation). There are several limitations in blackboard implicit invocation. Introduction of explicit invocation in the coordination model eliminates these limitations. Any direct call between entities is called explicit invocation. In Holo, beings influence other beings using history, but they can change information directly too.

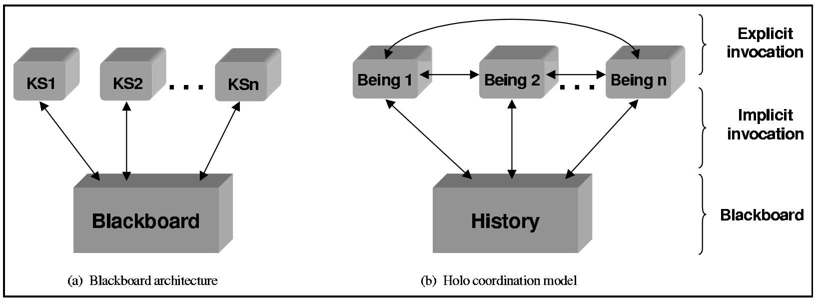

Fig. 3. Coordination Model

Figure 3b shows the Holo coordination model. History is a logic blackboard, i.e., the information stored is a group of logic terms. Interaction with the history uses two kinds of Linda-like [8] operations: affirmation and question. An affirmation puts terms in the history, like asserts in Prolog databases. Moreover, a question permits to consult terms from the history. A consult does a search in the database using unification of terms. A question is blocking or non-blocking. A blocking question only returns when a unifying term is found. Therefore, blocking questions synchronize beings using the implicit invocation. In a non-blocking question, if a unifying term is not found, the question immediately fails. Besides that, a question is destructive or non-destructive. A destructive question retracts the unifying term. The non-destructive does not remove it.

Hololanguage (so-called Holo) is a programming language that implements the concepts of Holoparadigm. A program is composed by descriptions of beings. Figure 4 shows a description of a being using the structure presented in figure 1b. Interface shows the actions accessible to other beings. Behavior contains the actions, which implement the being functionality. History is a logic blackboard used by actions and component beings. Initial state of history is a set of logic terms introduced by the programmer.

The behavior supports five kinds of actions: logic actions (LA), imperative actions (IA), modular logic actions (MLA), modular imperative actions (MIA) and multiparadigm actions (MA). LA is a logic predicate. IA is a group of imperative commands. MLA contains several logic actions encapsulated in a

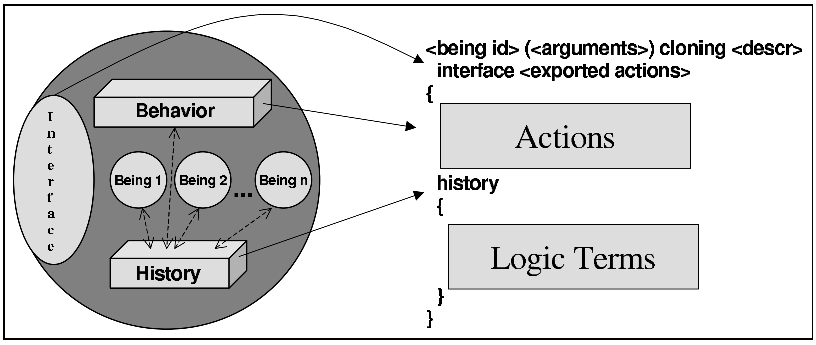

Fig. 4. Being Description

module. MIA encapsulates several imperative actions. MA integrates logic and imperative actions.

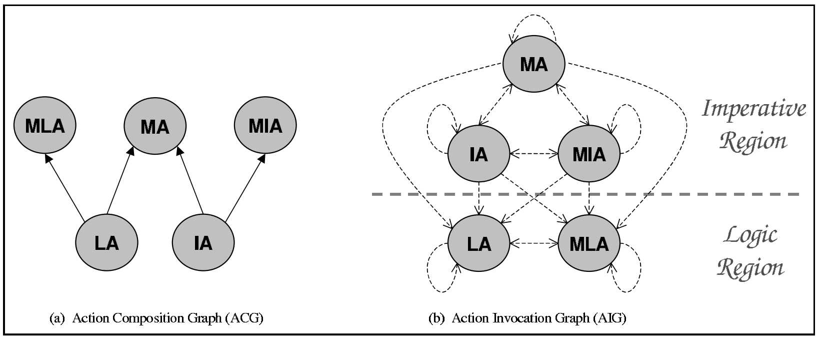

Fig. 5. Action Graphs

Actions are composed using an Action Composition Graph (ACG, figure 5a). Following Peter Wegners opinion [33] about the impossibility of mixing logic and imperative behaviors, we have created the Action Invocation Graph (AIG) presented in figure 5b. This graph determines the possible order of action calls during a program execution. MAs, IAs and MIAs call any action. LAs and MLAs only call LAs and MLAs. Therefore, there are two regions of actions during an execution, namely, imperative and logic regions (figure 5b).

If an execution flow goes in the logic region, the only way to return to the imperative region is finishing the flow (returning the results asked from the imperative region). This methodology eliminates many problems, which emerge when logic and imperative commands are mixed (for example, distributed backtracking). We believe AIG is an important contribution to the discussion presented by Wegner.

The language supports logical mobility (command *move*), concurrency between actions of a being (command *spawn*) and several kinds of cloning (command *clone*). Besides that, the Hololanguage permits both kinds of blackboard interaction proposed by the Holoparadigm. Affirmation uses the symbol "!". Moreover, there are several types of questions (figure 6). The symbol "." indicates a blocking question and the symbol "?" is used for non-blocking. Besides that, there are non-destructive (default) or destructive (symbol "#") questions. Finally, a question can return multiple answers. The symbols ".", "?", "#", and "*" (all answers) configure the questions.

Question Types	**Non-destructive (default)**	**Destructive #**	**Answer Quantifier (*n* or *, default = 1)**
Blocking .	**.**	**.#** (or only #)	**.*n*.** (non-destructive) **.#*n*.** (destructive)
Non-blocking ?	**?**	**?#**	**?*n*.** (non-destructive) **?#*n*.** (destructive)

Fig. 6. Types of Questions

3 HoloJava

Java is adequate to be used as an intermediate language [16]. On the other hand, standard Java cannot directly support logic actions, history and mobility. These aspects need special support to be implemented. We use JIProlog [9] (Prolog in Java) to implement logic actions, Jada [12] (blackboards in Java) to support the history and Voyager [29] (mobility in Java) to move beings.

Figure 7 shows the HoloJava transformation. Concurrent actions (spawn command) generate threads in Java. Logic actions are converted in JIProlog methods. Besides that, each *move* command can generate a *moveTo* method of Voyager. Each history is equivalent to a Jada tuple space and all kinds of history invocations can be directly converted in operations to spaces in Jada. Therefore, a program written in the Hololanguage is translated to a set of Java classes. These classes are a Java program that is generated based on the translation policy (figure 7). This Java program imports another set of Java classes that are the runtime system of the language. Holo's runtime system only takes care of the state of the Holo program. It contains some data structures that are updated during the execution of the program.

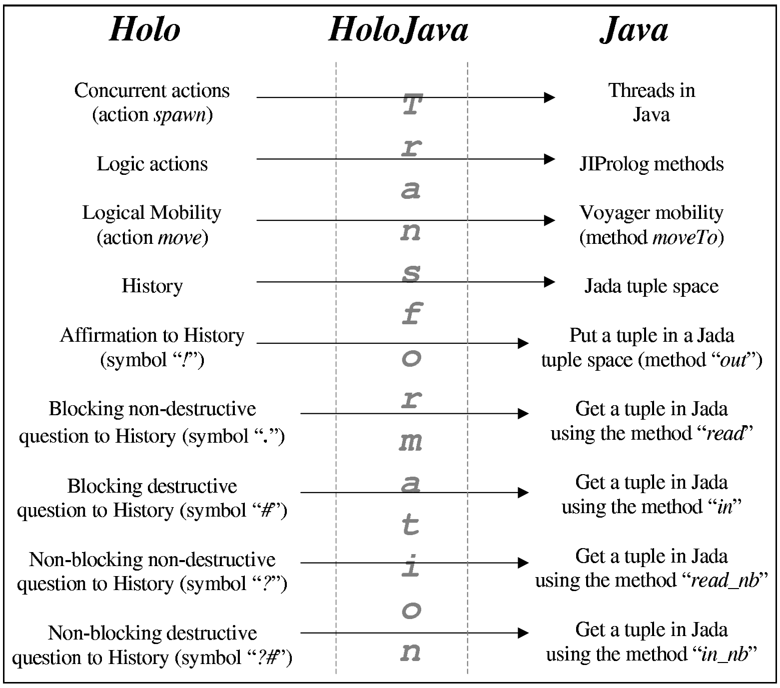

Fig. 7. HoloJava Translation Policy

4 Functional Programming in the Holo Paradigm

We are now working towards the integration of the functional programming paradigm into the Holo Language. The referential transparency and the lack of side effects make the functional paradigm a good source of parallelism. The functional paradigm will be inserted in the Holo context as Functional Actions (FAs) and Modular Functional Actions (MFAs).

Functions defined in a Holo program can only call other functions and can only be accessed by their interface in order to preserve the deterministic property of the functional paradigm. Thus, we have to modify the ACG and the AIG from figure 5 to the ones in figure 8.

In order to include FAs in the HoloJava compiler, we will use Du Bois' Java based implementation of the functional language Fun [13]. Fun is a small parallel functional programming language compiled to Java that can be used as target code for more powerful ones. Its Java based runtime system provides a simple

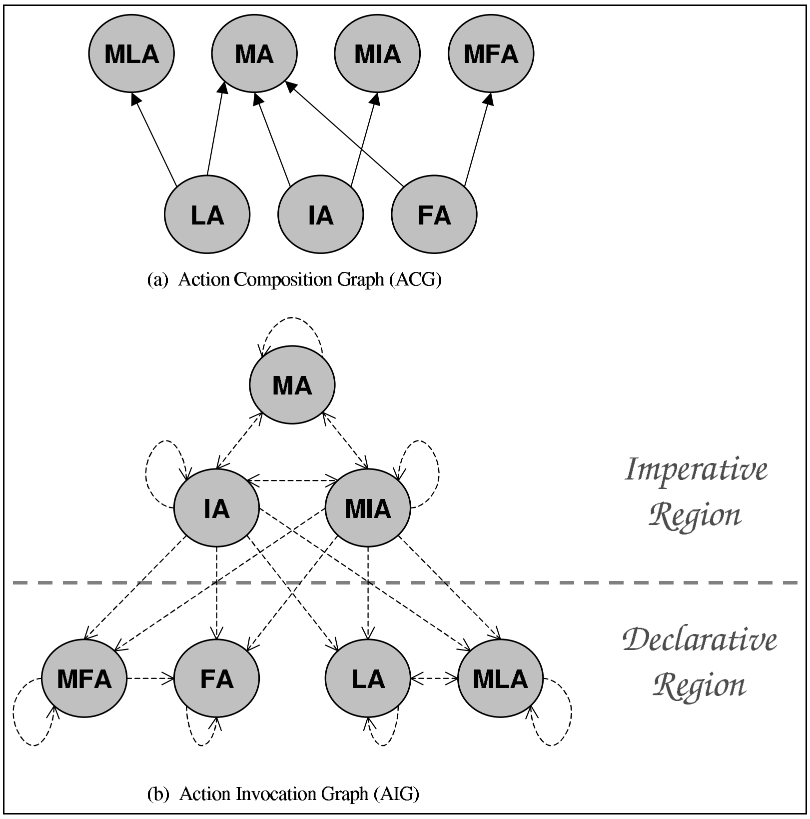

Fig. 8. Action Graphs with FAs and MFAs

interface that can be used by developers of pure lazy functional languages as a backend for the compilation of functional languages to the Java platform.

Functional languages are usually implemented using graph reduction, by compiling them to a graph reduction machine language. The runtime of the Fun language is based on the classic G-Machine [4]. All modern functional languages are implemented using some variant of the G-Machine [30].

The G-Machine is a fast graph reduction machine based on the compilation of expressions. The main idea is to translate each function in the functional program to a sequence of instructions which, when executed, will construct an instance of the function's body [25]. As the G-Machine represents expressions as graphs, its instructions are used to construct the graph representing function application and each expression in the functional program is used as a reduction rule for the graph.

The G-Machine implemented is simply a Java class, the GM class, which has some static methods that are the G-Machine instructions. To run functional programs on the JVM we compile each function in the functional program to one Java Class. These classes contains the G-Machine instructions (which are calls to static methods of the GM class) necessary to instantiate the function in the G-Machine. For example, the function

f x = id x

will be compiled to the following Java class:

```
class f extends Nsuperc{
   f (){
   narg = 1;
   name = new String ("f");}

  public void code(){
   GM.push (0);
   GM.pushglobal (new id());
   GM.mkap();
   GM.update(1);
   GM.pop(1);
  }
}
```

The Nsuperc is an abstract class that encompasses all functions defined in the functional program being compiled. The constructor of the class (in this case f()) always has to define two values: narg, which is the number of arguments for the function and the string name, which gives the name of the function (for debugging reasons).

All classes implementing the Nsuperc class must have a method called code() which has in its body the G-Machine instructions necessary to instantiate the body of the function.

In order to include functional programming in the Holo Language, the Holo-Java compiler will translate functions defined in a Holo program to a set of G-Machine instructions and classes. So the G-Machine implemented will be included in the Holo runtime system. When the Holo runtime system, during the execution of a program, needs to use a FA, the execution will be passed to the G-Machine.

Figure 9 shows a Holo program, in which can be seen the multiparadigm aspects of the language.

An action that has the same name as its being is called *guiding action*. A being will always execute its guiding action automatically after its creation. In figure 9, we can see that the guiding action executes 6 invocations, five of them (1, 2, 3, 4 and 6) directed to its history and one (number 5) to its behavior, which is a small functional program.

5 Implementing Functional Actions

The functional runtime system is implemented using an evaluate and print model. The GM class contains a method called reduce that simply pushes onto

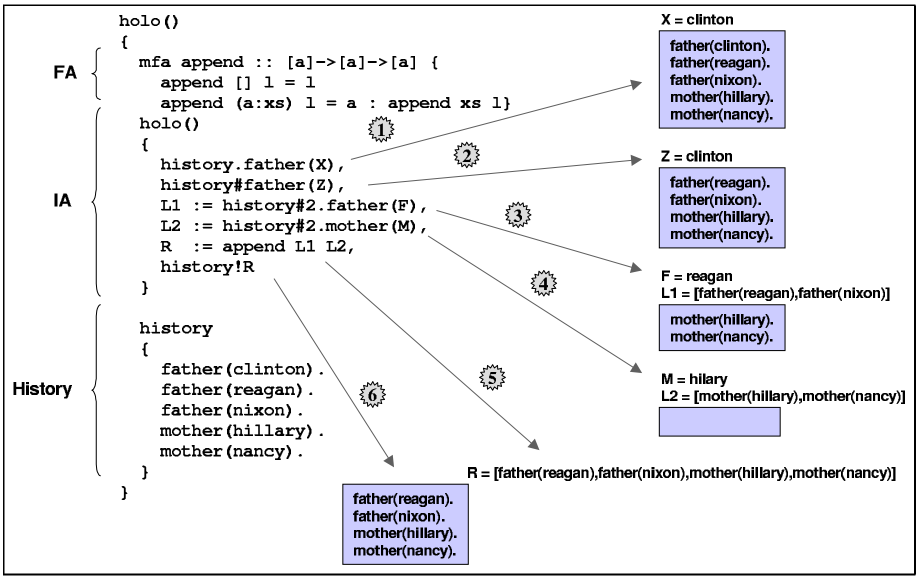

Fig. 9. Holo Program

the G-Machine stack the main function of the functional program and evaluates it to *Weak Head Normal Form* (WHNF) using the EVAL[4] instruction of the G-Machine:

```
public static Node reduce (Node main)
   {
     GM.push_onto_stack(main);
     GM.eval();
     return (GM.first_stack_elem());
   }
```

After executing the EVAL instruction the top element of the stack will contain the result of the evaluation of the program.

The main expression of the functional program is a CAF (constant applicative form) that contains in its body a graph representing function application. Thus, in order to use functional programs inside of a Holo program we need to construct a CAF representing the application of the FA into its actual parameter.

Each function defined in the functional program is compiled to one Java class as described before. For each function exported as a MFA we have a wrapper method that constructs a G-Machine Application Node using the class representing the G-Machine function and its argument. For example, the wrapper method for the f function presented before will be:

```
public Object run_f(Object arg){
        Nap func_app= new Nap (new f(), new Nobject(arg));
```

```
        Node node = GM.reduce (func_app);
        return (((Nobject) node).value());
        }
```

It creates a G-Machine Application Node (Nap class) with the G-Machine f function and its argument. Since f is a polymorphic function, the argument for the wrapper is a Java Object. The Object Class is the root of the Java's class hierarchy. Every class has Object as a superclass, so all objects implement the methods of this class. In the G-Machine implemented, we have a special kind of node, the Nobject node, that represents Java objects for polymorphic functions. After creating the application Node, the wrapper sends it to be evaluated by the GM.reduce method of the G-Machine. The result of this evaluation will be a value in WHNF that is returned as the result of the wrapper.

The G-Machine implemented has special nodes for all Java basic data types, so in some cases the wrapper has to deal with non-polymorphic functions that will receive and return basic data types. For example, the application node for a function g that has two Java integers as arguments will be:

```
        (...)
Nap func_app =
    new Nap (new Nap (new g(), new Int(arg1)), new Int(arg2));
           (...)
```

Currently, a function exported as a MFA can only return basic types and lists, although in its internal computation it can use any abstract data type.

6 Related Works

The Java programming language has been used by many developers of declarative languages as a platform for language implementation. The functional programming community gives us many examples of this approach. For example, there are many compilers for the state of the art functional programming language Haskell [20], that use Java as target code. In Wakeling's works [30–32] he implements a runtime system for functional languages in Java and than compiles Haskell programs to Java classes that uses this system. In Du Bois' work [13] he uses the same approach to generate JavaBeans components from functional programs. As another example of a declarative language running on the Java platform we have to mention the JIProlog, that was used to implement logic actions in the HoloJava system. The JIProlog system gives us a simple API that allows calling Prolog predicates in any Java applet or application.

There are also some multiparadigm languages that are compiled to Java. One example is the Curry language, that amalgamates functional, logic, and concurrent programming paradigms in one language. Its compiler [15] generates Java byte-code. The main emphasis of the Java-based implementation is the exploitation of Java threads to implement the concurrent and non-deterministic features of Curry.

I+ model [21] supports the distribution of objects, which implement methods using functions (functional classes) and logic predicates (logic classes). Its implementation is based on the translation of functional classes into Lazy ML (LML) modules and translation of logic predicates into Prolog modules. The distributed architecture is a network of Unix workstations using 4.3 BSD sockets to implemented message passing. The runtime environment was initially implemented using the C language, Quintus Prolog and LML. In a second phase, programs were only translated into C. Ciampolini et al [10] have proposed DLO, a system to create distributed logic objects. The implementation translates DLO programs into clauses of a concurrent logic language called Rose [7]. The support to Rose execution is implemented on a MIMD distributed memory parallel architecture (transputer-based Meiko Computing Surface). The runtime environment consists of a parallel abstract machine which is an extension of the WAM [1].

Oz multiparadigm language [23] is used to create a distributed platform called Mozart [26]. Oz uses a constraint store similar to a blackboard and supports the use of several paradigm styles [23, 19]. Besides, Mozart has special support for mobility of objects [17] and distributed treatment of logic variables [18]. Mozart distributed architecture is a network of workstations providing standard protocols such as TCP/IP. The run-time environment is composed by four software layers [26]: Oz centralized engine (Oz virtual machine [22]), language graph layer (distributed algorithms to decide when to do a local operation or a network communication), memory management layer (shared communication space and distributed garbage collection) and reliable message layer (transfer of byte sequences between nodes). The physical mobility supported by Mozart is completely transparent, i. e., the system decides when to move an object. None kind of logical mobility is used. The shared spaced supported by Mozart is monotonic and stores constraints. Being's history is a non-monotonic logic blackboard that stores logic tuples (terms). In addition, Mozart does not provide levels of encapsulated contexts composed by objects accessing a shared space.

Tarau has proposed Jinni [27], a logic programming language that supports concurrency, mobility and distributed logic blackboards. Jinni is implemented using BinProlog [6] a multi-threaded Prolog system with ability to generate C/C++ code. Besides that, it has special support to Java, such as a translator allowing packaging of Jinni programs as Java classes.

7 Concluding Remarks

We have presented our first steps to integrate functional programming into the Holo Paradigm. The main contributions of this paper were to present how pure functions could interact with the other types of actions that can occur in a being (see the new versions of the ACG and AIG in figure 8) and how we could, in a first stage, use an implementation of the G-Machine in Java so that the HoloJava compiler can support functional programming.

Currently we have a small prototype of the system that only handles Java's basic data types and lists, although polymorphic functions can manipulate any

Java object using the Nobject node of the G-Machine. Lazy evaluation and curried functions can only be used inside of the functional program. Thus the system works as a component based system in which the software developer can choose which language suits better to implement parts of the algorithm (actions). Of course there are better approaches for integrating programming paradigms. One interesting example is the runtime system of the curry language, based on a (narrowing) abstract machine that integrates functional, logic and concurrent programming [15]. Our current work consist in finding better ways implement interactions between different programming paradigms.

Although the FUN language supports parallel evaluation using the par and seq combinators [28], currently this feature is not working in the Holo language.

References

1. Ait-Kaci, H. Warren's Abstract Machine - A Tutorial Reconstruction. MIT Press, 1991.
2. Ambriola, V.; Cignoni, G. A.; Semini; L. A Proposal to Merge Multiple Tuple Spaces, Object Orientation and Logic Programming. Computer Languages, Elmsford, v.22, n.2/3, p.79-93, July/October 1996.
3. Apt, R. et al. Alma-0: An Imperative Language that Supports Declarative Programming. ACM Transactions on Programming Languages ans Systems, New York, v.20, September 1998.
4. Augustsson, L. A Compiler for Lazy ML. In Proc. ACM Symposium on Lisp and Functional Programming, Austin, pp. 218-27, 1984.
5. Barbosa, J. L. V.; Geyer, C. F. R. Uma Linguagem Multiparadigma Orientada ao Desenvolvimento de Software Distribudo. V Simpsio Brasileiro de Linguagens de Programao (SBLP), maio 2001.
6. Bosschere, K.; Tarau, P. Blackboard-based Extensions in Prolog. Software - Practice and Experience, v.26, n.1, p.49-69, January 1996.
7. Brogi, A. AND-parallelism without shared variables. Seventh International Conference on Logic Programming. MIT Press, p.306-324, 1990.
8. Carriero, N.; Gelernter, D. Linda in context. Communications of the ACM, v.32, n.4, p.444-458, 1989.
9. Chirico, U. JIProlog - Java Internet Prolog. www.geocities.com/jiprolog, May 2001.
10. Ciampolini, A.; Lamma, E.; Stefanelli, C; Mello, P. Distributed Logic Objects. Computer Languages, v.22, n.4, p.237-258, December 1996.
11. Ciancarini, P. Distributed Programming with Logic Tuple Spaces. New Generating Computing, Berlin, v.12, n.3, p.251-283, 1994.
12. Ciancarini, P.; Rossi, D. JADA: A Coordination Toolkit for Java. www.cs.unibo.it/rossi/jada May 2001.
13. Du Bois, A. R.; Rocha Costa, A. C.. Distributed Execution of Functional Programs using the JVM. In *EUROCAST 2001*. Las Palmas Gran Canaria - Spain, 2001.
14. Flanagan, D. *Java in a Nutshell*, Third Edition. O'Reilly & Associates, 1999.
15. Hanus, M.; Sadre, R. An Abstract Machine for Curry and its Concurrent Implementation in Java. In *J. Functional and Logic Programming* 6, MIT Press, 1999.
16. Hardwick, J. Java as an Intermediate Language. School of Computer Science, Carnegie Mellon University, Technical Report CMU-CS-96-161, August 1996
17. Haridi, S. et al. Programming Languages for Distributed Applications. New Generating Computing, v.16, n.3, p.223-261, 1998.

18. Haridi, S. et al. Efficient Logic Variables for Distributed Computing. ACM Transactions on Programming Languages and Systems, v. 21, n.3, p.569-626, May 1999.
19. Henz, M. Objects in Oz. Saarbrchen: Universitt des Saarlandes, May 1997. (PhD Thesis)
20. Hudak, P.; Peterson, J.; Fasel, J. H. A Gentle Introduction to Haskell - Version 1.4. http://www.haskell.org, May 2001.
21. Lee, J. H. M.; Pun, P. K. C. Object Logic Integration: A Multiparadigm Design Methodology and a Programming Language. Computer Languages, v.23, n.1, p.25-42, April 1997.
22. Meh, M.; Scheidhauer, R.; Schulte, C. An Abstract Machine for OZ. Seventh International Symposium on Programming Languages, Implementations, Logics and Programs (PLIP'95), Springer-Verlag, LNCS, September 1995.
23. Muller, M.; Muller, T.; Roy, P. V. Multiparadigm Programming in Oz. Visions for the Future of Logic Programming: Laying the Foundations for a Modern Sucessor of Prolog, 1995.
24. Nii, H. P. Blackboard systems: the blackboard model of problem solving and the evolution of blackboard architectures. AI Magazine, v.7, n.2, p.38-53, 1986.
25. Peyton Jones, S.L.; Lester D. *Implementing Functional Languages. A Tutorial.* Prentice Hall, 1992.
26. Roy, P. V. et al. Mobile Objects in Distributed Oz. ACM Transactions on Programming Languages and Systems, v.19, n.5, p.804-851, September 1997
27. Tarau, P. Jinni: Intelligent Mobile Agent Programming at the Intersection of Java and Prolog. PAAM'9, The Practical Applications Company, 1999.
28. Trinder, P.W., Hammond, K., Loidl, H.-W., Peyton Jones, S.L.: Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23-60, January (1998)
29. Voyager. www.objectspace.com, April 2001.
30. Wakeling, D.: A Haskell to Java Virtual Machine Code Compiler, . In C. Clack, A.J.T. Davie and K. Hammond, editors, *Proc. 9th. International Workshop on the Implementation of Functional Languages (IFL '97), St Andrews, Scotland, September 1997*, volume 1467 of LNCS, Springer-Verlag (1998) 39-52
31. Wakeling, D.: Mobile Haskell: Compiling Lazy Functional Programs for the Java Virtual Machine. In *Proc. Principles of Declarative Programming (PLIP '98)*, Pisa, Italy, Springer-Verlag, September (1998) 335-352
32. Wakeling, D.: Compiling Lazy Functional Programs for the Java Virtual Machine. Journal of Functional Programming 1(1):1-000, Cambridge University Press, United Kingdom, January (1998)
33. Wegner, P. Tradeoffs between Reasoning and Modeling. In: Agha, G.; Wegner, P.; Yonezawa, A. (eds.). Research Direction in Concurrent Object-Oriented Programming. Mit Press, p.22-41, 1993.