Java como Linguagem Intermediária para Compiladores Multiparadigma

Jorge Luis Victória Barbosa
Escola de Informática (ESIN) – Universidade Católica de Pelotas (UCPel)
Pelotas, RS, Brasil
barbosa@ucpel.tche.br

André Rauber Du Bois
Department of Computer Science – Heriot-Watt University
Edinburgh, Scotland, U. K.
dubois@macs.hw.ac.uk

Laerte Kerber Franco
Gens Sistemas de Informática S/A
Porto Alegre, RS, Brasil
laerte@gens.com.br

Cláudio Fernando Resin Geyer
Instituto de Informática (II) – Universidade Federal do Rio Grande do Sul (UFRGS)
Porto Alegre, RS, Brasil
geyer@inf.ufrgs.br

Abstract

In this paper we describe the design and implementation of a system for translating Holo into Java. 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. Java is well known by being a portable and network-aware object oriented language. We propose the use of Java as an intermediate language in our development environment. There are Holo aspects that we do not translate directly to pure Java. We use special libraries to support these aspects. Specifically, logic programming characteristics are translated using Prolog Café and blackboards are implemented using Jada spaces.

Keywords: Software Engennering, Programming Languages.

Resumo

Este artigo descreve o projeto e a implementação de um sistema para conversão de Holo para Java. Holo é uma nova linguagem multiparadigma orientada ao desenvolvimento de software distribuído. A linguagem é baseada em conceitos de concorrência, blackboards e mobilidade. Além disso, Holo integra aspectos dos paradigmas em lógica, imperativo e orientado a objetos. Java é conhecida pela sua portabilidade e seu suporte ao uso de redes de computadores. O artigo propõe o uso de Java como linguagem intermediária para compilação de programas multiparadigma. Existem aspectos que não são traduzidos diretamente para Java. Neste caso, são usadas bibliotecas, tais como, Prolog Café como suporte à programação em lógica e Jada para implementação de blackboards.

Palavras-Chave: Engenharia de Software, Linguagens de Programação.

1 Introdução

O tema multiparadigma vem sendo pesquisado continuamente [2, 6, 9, 11, 13, 15, 16, 24, 25, 26, 28, 30]. Os pesquisadores deste tema propõem a criação de modelos de desenvolvimento de software através da integração de paradigmas básicos, tais como, paradigma em lógica, imperativo, funcional e orientado a objetos. Através dessa proposta eles buscam dois objetivos: a superação das limitações de cada paradigma e a exploração conjunta das suas características consideradas benéficas.

Neste contexto surge o Holoparadigma [7], um novo modelo multiparadigma orientado ao desenvolvimento de software distribuído. O Holoparadigma possui um modelo de coordenação baseado em *blackboards*. Um nova linguagem de programação (Hololinguagem [6], abreviada como Holo) implementa os conceitos propostos pelo modelo. A linguagem é baseada em concorrência, *blackboards* e mobilidade. Além disso, Holo integra o paradigmas em lógica, imperativo e orientado a objetos.

Diversos trabalhos indicam que Java [20] é adequada como linguagem intermediária [5, 14, 16, 30]. Este artigo propõe seu uso como linguagem intermediária para compilação de programas multiparadigma. A compilação de programas em Holo concretiza a proposta. Existem aspectos multiparadigma que não podem ser traduzidos diretamente para Java. Nestes casos são usadas bibliotecas especiais. Por exemplo, ações lógicas são implementadas usando Prolog Café [5] e *blackboards* usando espaços de objetos em Jada [10].

O artigo está organizado em cinco seções. A seção 2 apresenta o Holoparadigma e descreve os princípios da Hololinguagem. A terceira seção propõe a tradução dessa linguagem para Java. A seção 4 discute trabalhos relacionados. Finalmente, a quinta seção contém conclusões e trabalhos futuros.

2 Holoparadigma e Hololinguagem

O Holoparadigma é um modelo multiparadigma dedicado à exploração automática do paralelismo e distribuição [7]. Este modelo possui como unidade de modelagem o ente e como unidade de informação o símbolo. Um ente elementar (figura 1a) é organizado em três partes: interface, comportamento e história. Um ente composto (figura 1b) possui a mesma organização, no entanto, suporta a existência de outros entes na sua composição (entes componentes). Cada ente possui uma história. A história fica encapsulada no ente e, no caso dos entes compostos, é compartilhada pelos entes componentes. Sendo assim, podem existir vários níveis de encapsulamento da história. Os entes acessam somente a história em um nível. A composição varia de acordo com a mobilidade dos entes em tempo de execução. A figura 1c mostra um ente composto de três níveis e exemplifica a história encapsulada.

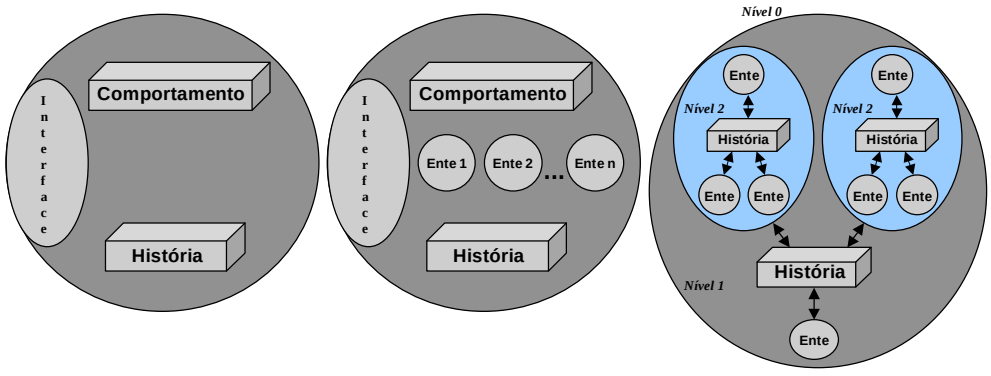


Figura 1. Tipos de entes

No escopo de sistemas distribuídos, um ente pode assumir dois estados de distribuição: centralizado ou distribuído. A figura 2b mostra um possível estado de distribuição para o ente mostrado na figura 1c. Neste caso, a história atua como uma memória compartilhada distribuída (DSM). A mobilidade [29] é a capacidade que permite o deslocamento de um ente. O Holoparadigma utiliza dois tipos de mobilidade (lógica e física). A mobilidade lógica relaciona-se com o deslocamento em nível de modelagem, ou seja, sem considerações sobre a plataforma de execução. Neste contexto, um ente se move quando cruza uma ou mais fronteiras de entes. A mobilidade física relaciona-se com o deslocamento entre nodos de uma arquitetura distribuída. Neste contexto, um ente se move quando se desloca de um nodo para outro. A figura 2a exemplifica uma possível mobilidade lógica no ente apresentado na figura 1c. As mobilidades lógica e física são independentes. A ocorrência de um

tipo de deslocamento não implica a ocorrência do outro. Merece atenção o caso da mobilidade física não implicar obrigatoriamente a mobilidade lógica. A figura 2b mostra uma mobilidade física sem mobilidade lógica. O modelo de coordenação dos entes compostos assemelha-se a um *blackboard* [27]. Os *Knownledge Sources* (KSs) são entes e o *blackboard* é a história.

A Hololinguagem (abreviada como Holo) [6] integra os paradigmas imperativo, em lógica e orientado a objetos. Holo busca não somente o suporte aos paradigmas integrados, mas também as novas oportunidades que surgem através da integração. A linguagem utiliza representação simbólica, unificação e não possui tipos. Estas características são herdadas do paradigma em lógica. Holo suporta programação de alta ordem, ou seja, símbolos representando entes e suas partes (interface, ações, componentes e história) podem ser manipulados através de variáveis e operações da linguagem. Entes elementares, compostos e vários níveis de *blackboards* podem ser utilizados. Existem cinco tipos de ações: ação lógica (*Logic Action* – LA), ação imperativa (*Imperative Action* – IA, ação modular lógica (*Modular Logic Action* – MLA), ação modular imperativa (*Modular Imperative Action* – MIA) e ação multiparadigma (*Multiparadigm Action* – MA).

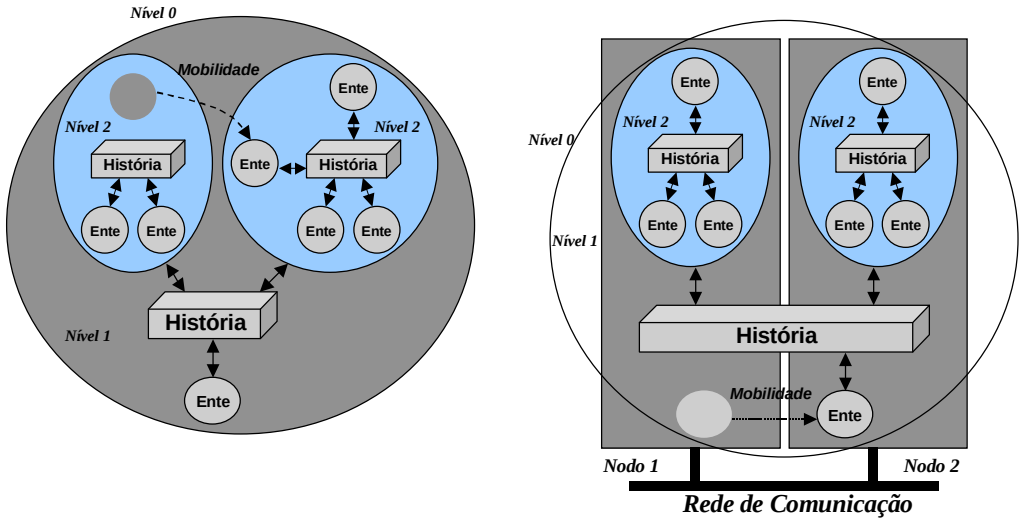


Figura 2. Mobilidade no Holoparadigma

3 Traduzindo Holo para Java

Visando a realização de testes de Holoprogramas (programas em Holo) no menor espaço de tempo possível, foi criada uma ferramenta de conversão de programas denominada HoloJava. Esta ferramenta converte programas em Holo (linguagem origem) para programas em Java (linguagem destino). Esta técnica é bastante usada na avaliação de novas linguagens, pois a utilização de uma linguagem destino que possua uma plataforma (compilação e execução) já disponível, antecipa a realização de testes e a difusão de resultados da pesquisa.

Diversos estudos [5, 14, 16, 30] indicam que Java pode ser utilizada como linguagem intermediária na construção de compiladores. Java suporta diretamente várias abstrações do Holoparadigma (ações imperativas, concorrência, etc). No entanto, algumas características de Holo não são suportadas (ações lógicas, *blackboard* e mobilidade). Por outro lado, existem bibliotecas que complementam a plataforma Java e podem ser utilizadas para implementação dos aspectos não suportados. Por exemplo, ações lógicas em Prolog Café [5], história em Jada [10] ou JavaSpaces [22] e mobilidade física em Voyager [32] ou Horb [18].

A figura 3 apresenta o contexto de desenvolvimento da HoloJava. A criação da ferramenta está sendo realizada com a utilização de JavaCC [21], um gerador de analisadores sintáticos que gera código Java. A entrada do JavaCC é uma gramática Holo acrescida de ações semânticas responsáveis pela conversão.

Atualmente, as seguintes bibliotecas estão complementando a linguagem Java: (1) Prolog Café [5]: sistema que suporta a conversão de Prolog para Java; (2) Jada [22]: biblioteca que permite a implementação de espaço s de objetos (*blackboards*) em Java.

A figura 4 mostra a política de conversão usada na HoloJava. Esta política consiste no mapeamento de estruturas de holoprogramas para estruturas de um programa em Java. Jada e Prolog Café são utilizadas para a

concretização da história e ações lógicas. Por sua vez, a concorrência inter-entes é suportada através de *threads*. Entes compostos são tratados como grupos e a mobilidade lógica é implementada usando técnicas de *membership* [23].

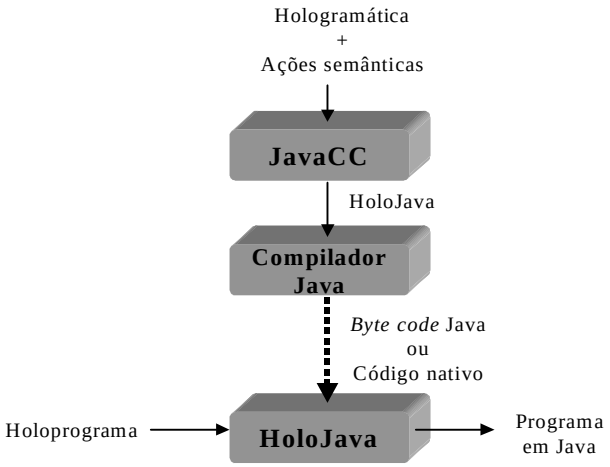


Figura 3. Contexto da criação da HoloJava

A execução de um programa cria uma estrutura hierárquica de entes, denominada Árvore de Entes (***HoloTree***). A HoloTree implementa o encapsulamento de entes em níveis de composição, conforme proposto pelo Holoparadigma. Além disso, a HoloTree suporta o aspecto dinâmico da política de grupos, mudando continuamente durante a execução de um programa. As seguintes ações [6] alteram a HoloTree: (1) clonagem: a clonagem cria um novo ente; (2) mobilidade: a mobilidade altera a HoloTree, deslocando um ente (elementar ou composto) na árvore. A figura 5a exemplifica a HoloTree para a organização em níveis mostrada na figura 1c. Um ente composto possui ligações com seus entes componentes, os quais ficam localizados no nível abaixo. Os entes componentes possuem acesso a história e ao comportamento do composto no qual estão inseridos. Por sua vez, o composto possui acesso aos comportamentos dos seus componentes. Além disso, um ente possui acesso ao comportamento dos demais entes no mesmo nível.

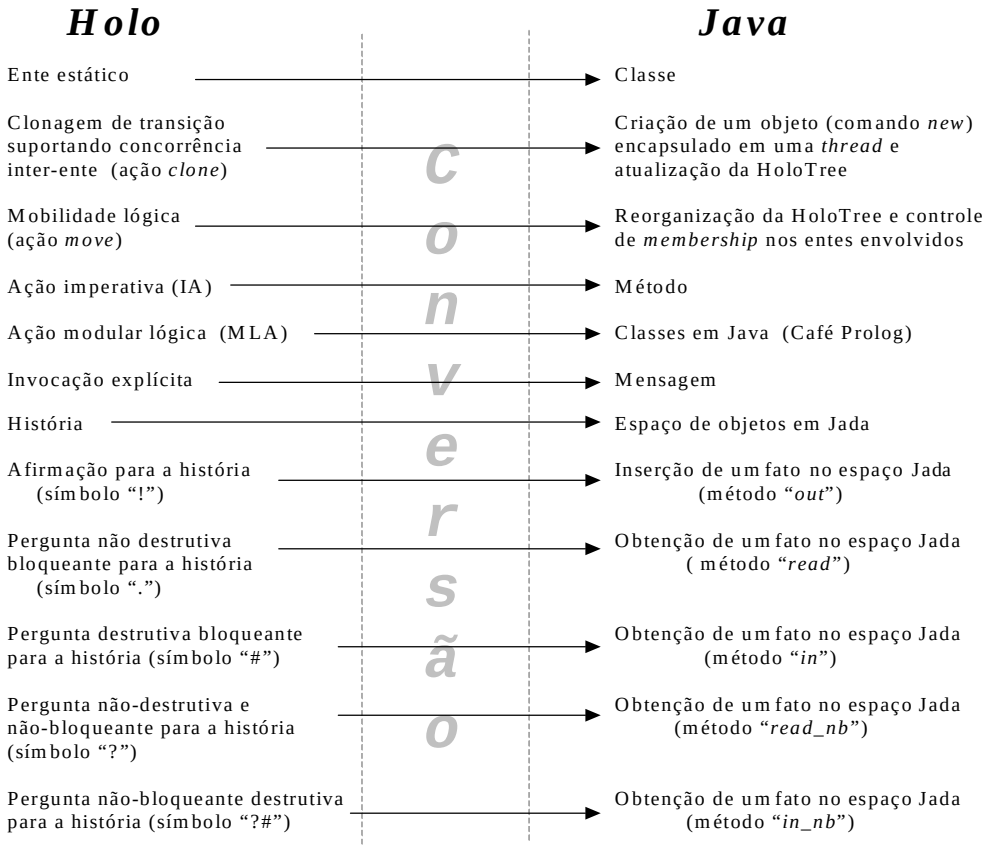
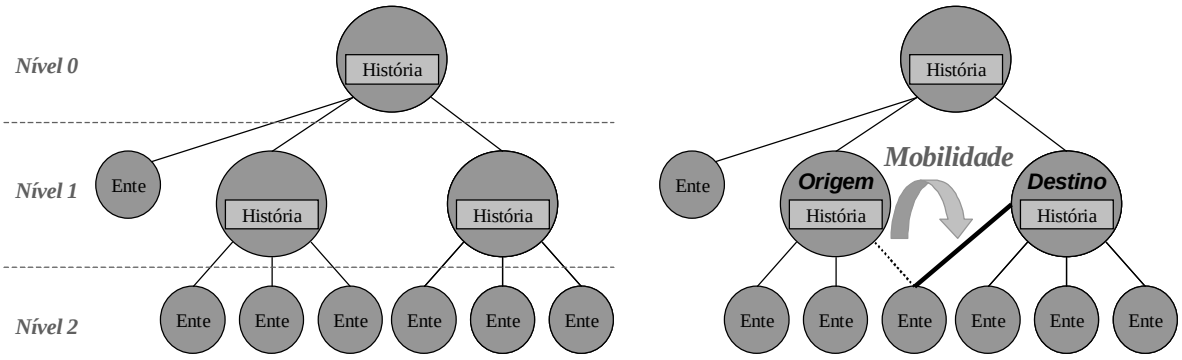


Figura 4. Política de conversão de Holo para Java

Quando a mobilidade ocorre, torna-se necessária a mudança da visão do grupo dos entes envolvidos. O ente movido possui uma nova visão (novos entes no mesmo nível e um novo composto acima dele). Se o movido for um ente composto, a visão dos seus componentes não muda. A mobilidade implica a atualização da composição dos entes origem e destino. Além disso, os componentes de um ente podem ser entes compostos (grupos de grupos). A mobilidade de um ente elementar equivale a realocação de uma folha da árvore e a mobilidade de um composto transfere um ramo contendo vários entes. A figura 5b apresenta a mudança que ocorreria na HoloTree para a mobilidade na figura 2a.



(a) Árvore de Entes (HoloTree) (b) Mobilidade

Figura 5. Árvore de entes (HoloTree)

A figura 6 mostra um programa utilizado para demonstração da HoloJava. O programa implementa uma mineração (*datamining*) simplificada envolvendo mobilidade. O *datamining* é uma das principais aplicações previstas para o Holo [6]. O programa contém diversos comentários. Além disso, as seguintes observações descrevem o exemplo:

(1) o programa é composto de três entes estáticos (*holo*, *mine* e *miner*);

(2) o ente dinâmico *d_holo* (criado automaticamente para o ente *holo*) cria três minas e um mineiro. Logo após, aguarda os resultados serem colocados na sua história;

(3) as minas (entes *mine_d1*, *mine_d2* e *mine_d3*) notificam sua criação e aguardam a mineração. A história das minas contém três fatos. Cada fato guarda a identificação da mina e um número que indica qual o *Fibonacci* deverá ser calculado pelo mineiro;

(4) o mineiro (ente *miner_d*) entra em uma mina, realiza a mineração, sai da mina e insere o resultado na história de *d_holo*. Estes passos são executados para cada mina.

A figura 7 representa os doze passos do mineiro na árvore de entes. A mineração consiste na busca de um valor para cálculo de *Fibonacci*. O cálculo é realizado usando uma MLA (*Modular Logic Action*). O exemplo possui as seguintes características que merecem atenção especial:

(1) todos os entes são concorrentes, ou seja, são criadas cinco *threads* (uma para *d_holo*, uma para o mineiro e três para as minas);

(2) o ente minerador possui três ações, duas IAs e uma MLA. A MLA exemplifica o uso de ações lógica em Holo;

(3) o ente minerador é responsável pelo controle de sua mobilidade (ação *move*);

(4) os passos 2, 6 e 10 do mineiro utilizam o acesso à história externa. Este acesso é realizado na ação *mining*. Apesar do código em *mining* ser o mesmo nos três passos, o *out(history)* é sensível ao contexto (mina) no qual o mineiro está em determinado momento.

A figura 8 apresenta as quatro etapas envolvidas na conversão e execução de holoprogramas. O programa *datamining.holo* (figura 6) é utilizado no exemplo. As etapas são as seguintes: (1) **conversão HoloJava**: cada ente gera um arquivo *.java* contendo uma classe. Além disso, cada MLA gera um arquivo em Prolog (no exemplo, *fib.pl*); (2) **conversão Prolog Café**: os arquivos Prolog são convertidos para arquivos Java. A conversão gera um arquivo *.java* (no exemplo, *PRED_fib_2.java*) para cada predicado existente no arquivo Prolog (3) **compilação dos arquivos Java**: nesta etapa, os arquivos Java são compilados. O arquivo *holo.class* contém o ponto de partida para a execução; (4) **execução do programa**.

```
/********************  ENTE PRINCIPAL ********************

holo()     // Ente principal.
{
  holo()   // Acao guia.
  {
    writeln('HOLO: Vou criar tres minas e um mineiro'),
    clone(mine(1),mine_d1),  // Cria a primeira mina. O parametro identifica a mina.
    clone(mine(2),mine_d2),  // Cria a segunda mina.
    clone(mine(3),mine_d3),  // Cria a terceira mina.
    clone(miner,miner_d),    // Cria o mineiro.
    for X := 1 to 3 do       // Aguarda pelos resultados da mineracao.
    {
      history#list(#Ident,#Num,#Fibo),   // Obtem o resultado de uma mineracao.
      writeln('HOLO: Terminou a mineracao da mina:',Ident),
      writeln('HOLO: Fibonacci de ',Num,' e ',Fibo)
    }
    writeln('HOLO: Terminou a mineracao')
  }
}

//********************  ENTE MINA ******************

mine()
{
  mine(Ident)
  {
    writeln('MINA ',Ident,': Fui criada')
  }

  history         // A historia da mina de cada mina possui
  {               // o mesmo conteudo. O primeiro numero
    list(1,2).    // identifica a mina. O segundo e o numero
    list(2,4).    // usado para o calculo do Fibonacci.
    list(3,6).
  }
}

//********************  ENTE MINEIRO ******************

miner()
{
  miner()
  {
    writeln('MINEIRO: Inicio da mineracao.'),
    move(self,mine_d1),                       // Passo 1  - Entra na mina 1
    mining(1,Num1,Res1),                      // Passo 2  - Minera a mina 1
    move(self,out),                           // Passo 3  - Sai da mina 1
    out(history)!list(1,Num1,Res1),           // Passo 4  - Salva o resultado 1
    move(self,mine_d2),                       // Passo 5  - Entra na mina 2
    mining(2,Num2,Res2),                      // Passo 6  - Minera a mina 2
    move(self,out),                           // Passo 7  - Sai da mina 2
    out(history)!list(2,Num2,Res2),           // Passo 8  - Salva o resultado 2
    move(self,mine_d3),                       // Passo 9  - Entra na mina 3
    mining(3,Num3,Res3),                      // Passo 10 - Minera a mina 3
    move(self,out),                           // Passo 11 - Sai da mina 3
    out(history)!list(3,Num3,Res3),           // Passo 12 - Salva o resultado 3
    writeln('MINEIRO: Fim da mineracao.')
  }

  mining(Ident,Num,Result)           // IA que realiza a mineracao.
  {
   out(history)#list(Ident,#Num),   // Minera a historia externa.
   fib(Num,#Result)                 // Chama a MLA para determinar Fibonacci.
  }

  fib/2()          // Acao Modular Logica (MLA) que calcula Fibonacci.
  {
    fib(1,1).
    fib(2,1).
    fib(M,N) :-
       M > 2,
       M1 is M-1,
       M2 is M-2,
      fib(M1,N1),
      fib(M2,N2),
      N is N1+N2.
  }
}
```

Figura 6. Programa de mineração *datamining.holo*

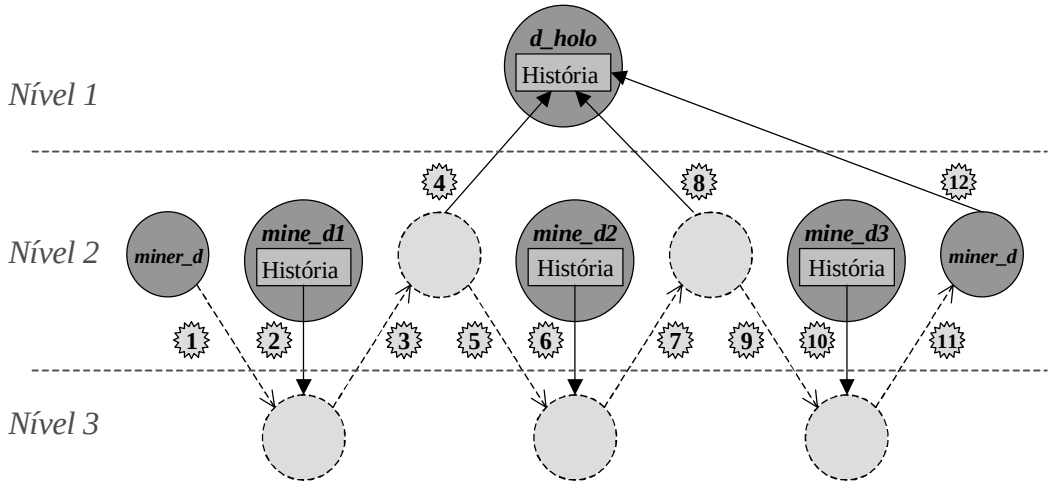


Figura 7. Passos de mineração (*datamining.holo)*

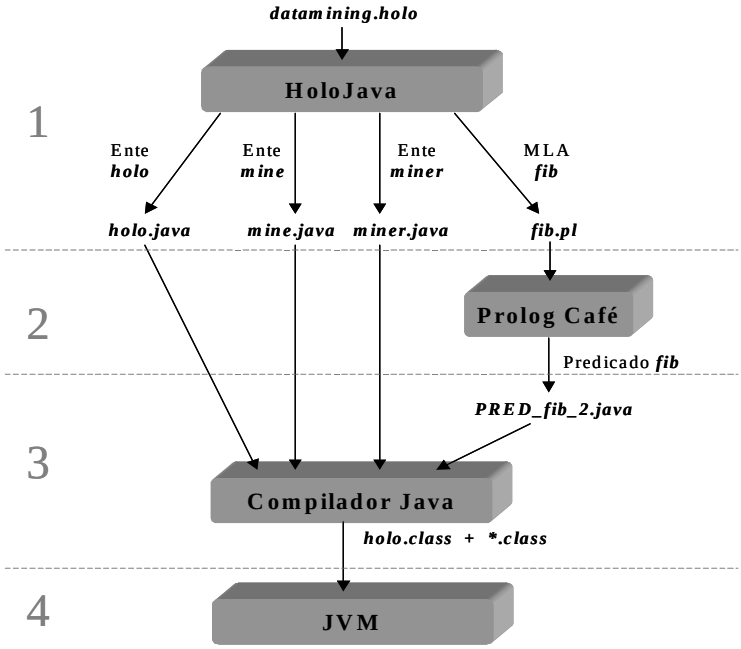


Figura 8. Gerenciamento de arquivos na conversão e execução

A tabela 1 mostra as características de dez programas acompanhadas dos resultados de suas conversões. Por sua vez, a tabela 2 mostra as versões de software e a configuração de hardware usada no experimento. Uma descrição sucinta dos programas é apresentada a seguir: (1) ***datamining.holo*** (figura 6): simula uma mineração usando três minas e um mineiro. O programa utiliza uma MLA para cálculo de números da série de Fibonacci; (2) ***performance.holo***: simula uma mineração usando três minas e um mineiro. Não utiliza uma MLA; (3) ***semaphores.holo***: suporta o gerenciamento de semáforos. Diversos entes concorrem por um recurso compartilhado usando semáforos para controle da exclusão mútua; (4) ***philosophers.holo***: implementa o clássico programa concorrente "Jantar de Filósofos". Os filósofos são implementados como entes ; (5) ***buffers.holo***: suporta o gerenciamento de *buffers*. Diversos produtores e consumidores podem acessar de forma concorrente o *buffer*. Os produtores e consumidores são entes e o sincronismo é realizado pela história; (6) ***travel.holo***: simula uma viagem por dois estados brasileiros, visitando cinco municípios. Os estado e municípios são modelados através da composição de entes; (7) ***hanoi.holo***: implementa o programa Torre de Hanói. Utiliza uma MLA para determinação do resultado; (8) ***fibo.holo***: implementa o cálculo de Fibonacci. Utiliza uma MLA; (9) ***lists.holo***: realiza um conjunto de manipulações de listas usando uma MLA; (10) ***family.holo***: gerenciador de relações familiares. Problema clássico para demonstração de base de dados em Prolog.

A tabela 1 está organizada em dez linhas e oito colunas. Cada linha está relacionada com um programa. As oito colunas são as seguintes:

(1) **Holoprograma (*.holo*)**: nome do programa analisado;

(2) **Tamanho em linhas (*.holo*)**: número de linhas do programa;

(3) **Tamanho em bytes (*.holo*)**: número de bytes do programa;

(4) **Arquivos gerados (*.java* e *.pl*)**: nome dos arquivos em Java e Prolog gerados pela conversão. Nesta coluna podem ser encontrados dois tipos de campos. O primeiro tipo representa um arquivo Java gerado diretamente pela HoloJava (por exemplo, o arquivo *holo.java* na conversão de *datamining.holo*). O segundo tipo representa um *.java* resultante da conversão do arquivo Prolog gerado pela HoloJava (por exemplo, o arquivo *PRED_fib_2.java* na conversão de *datamining.holo*). O Prolog Café gera um arquivo Java para cada predicado (veja figura 8). No segundo tipo, a primeira linha do campo contém o nome do *.java*. A segunda mostra o programa Prolog origem e o número de arquivos *.class* gerados pela compilação do arquivo Java;

(5) **Tempo de conversão (milisegundos)**: esta coluna indica o tempo envolvido na conversão de um arquivo *.holo* para os arquivos *.java* e *.pl* (por exemplo, 163,60 ms para geração dos arquivos *holo.java*, *mine.java*, *miner.java* e *fib.pl*, no caso de *datamining.holo*). Além disso, esta coluna mostra o tempo necessário (Prolog Café) para conversão de um programa Prolog para Java (por exemplo, 15.178,80 ms para geração de *PRED_fib_2.java*, no caso de *datamining.holo*). Conforme mostram os exemplos, a HoloJava e o Prolog Café podem gerar diversos arquivos Java;

(6) **Tamanho em linhas (*.java*)**: número de linhas do fonte em Java gerado pela HoloJava ou pelo Prolog Café;

(7) **Tamanho em bytes (*.java*)**: número de bytes dos arquivos *.java* gerados pela HoloJava ou pelo Prolog Café;

(8) **Tamanho em bytes (*.class*)**: tamanho em bytes dos arquivos *bytecode* gerados pela compilação dos programas. A compilação de um *.java* gerado pela HoloJava produz somente um *.class*. No entanto, a compilação de um *.java* gerado pelo Prolog Café pode gerar diversos *.class*. Neste caso, esta coluna apresenta a soma dos tamanhos destes arquivos. A quarta coluna indica quantos *.class* resultaram da compilação de um *.java* gerado pelo Prolog Café.

A análise dos resultados mostrados na tabela 1 permite as seguintes conclusões:

(1) existe uma diferença considerável no tempo de conversão da HoloJava em relação ao tempo de conversão do Prolog Café. Por exemplo, no programa *datamining.holo*, 98,93 % do tempo total equivale a conversão de *fib.pl*. O holoprograma possui 73 linhas, das quais apenas 12 compõem a MLA. Os programas *datamining.holo* e *performance.holo* são semelhantes. A principal diferença consiste na MLA existente no primeiro. No entanto, a conversão de *datamining.holo* é aproximadamente 105 vezes mais lenta do que a conversão de *performance.holo*. O mesmo comportamento pode ser constatado nos demais programas que usam MLAs. Conclui-se assim que o uso deste tipo de ações é um ponto crítico no desempenho da conversão;

(2) o tamanho dos arquivos gerados pelo Prolog Café é consideravelmente maior do que os gerados pela HoloJava. Por exemplo, considerando o programa *datamining.holo*, o Prolog Café gerou 35,17 % do número de linhas em Java, 36,57 % do número de bytes nos arquivos Java e 44,31 % do número de bytes nos arquivos *.class*. Por outro lado, a MLA compõe apenas 16,44 % do tamanho do holoprograma em linhas. Conclui-se assim que as MLAs também são um ponto crítico no tamanho dos arquivos gerados;

(3) tendo como base a tabela 1 podem ser criados um conjunto de índices para avaliação da conversão de holoprogramas para Java. A tabela 3 mostra os índices e sua aplicação em cinco programas. A tabela é composta de oito colunas. A primeira possui o nome do índice e a segunda sua descrição. As demais colunas contêm a aplicação dos índices em cinco programas. Os programas escolhidos não possuem MLAs. Busca-se assim uma maior estabilidade para os índices de conversão.

Tabela 1. Informações sobre conversão de holoprogramas

Holoprograma (*.holo*)	**Tam. em linhas (*.holo*)**	**Tam. em bytes (*.holo*)**	**Arquivos gerados (*.java* e *.pl*)**	**Tempo de conver. (*ms*)**	**Tam. em linhas (*.java*)**	**Tam. em bytes (*.java*)**	**Tam. em bytes (*.class*)**
datamining.holo	73	1.391	holo.java	163,60	50	1.461	2.222
			mine.java		31	595	1.279
			miner.java		166	4.813	3.761
			PRED_fib_2.java (fib.pl) (7 x .class)	15.178,80	134	3.961	5.779
			Total	15.342,40	381	10.830	13.041
performance.holo	62	1.206	holo.java	146,80	51	1.424	2.186
			mine.java		29	507	1.210
			miner.java		99	2.936	2.766
			Total	146,80	179	4.867	6.162
semaphores.holo	50	879	holo.java	126,40	58	1.291	1.967
			competidor.java		45	874	1.508
			Total	126,40	103	2.165	3.475
philosophers.holo	51	1.084	holo.java	138,00	54	1.497	2.012
			filosofo.java		56	1.636	1.988
			Total	138,00	110	3.133	4.000
buffers.holo	63	1.248	holo.java	151,00	105	2.397	2.373
			produtor.java		49	1.096	1.563
			consumidor.java		49	1.101	1.568
			Total	151,00	203	4.594	5.504
travel.holo	112	2.848	holo.java	193,00	44	1.099	1.869
			Estado.java		57	1.541	2.094
			municipio.java		32	719	1.452
			viajante_e.java		102	4.054	3.475
			Total	193,00	235	7.413	8.890
hanoi.holo	28	605	holo.java	126,60	121	3.684	2.945
			PRED_hanoi_5.java (hanoi.pl) (5 x .class)	17.238,80	155	5.194	6.009
			PRED_append_3.java (hanoi.pl) (5 x .class)		131	3.990	4.814
			Total	17.365,40	407	12.868	13.768
fibo.holo	25	454	holo.java	115,80	85	2.131	2.435
			PRED_fib_2.java (fib.pl) (7 x .class)	15.130,20	134	3.961	5.779
			Total	15.246,00	219	6.092	8.214
lists.holo	39	858	Holo.java	131,40	118	3.455	2.815
			PRED_listas_4.java (listas.pl) (1 x .class)	17.126,80	52	1.379	1.389
			PRED_nrev_2.java (listas.pl) (5 x .class)		119	3.483	4.765
			PRED_append_3.java (listas.pl) (5 x .class)		131	3.991	4.814
			PRED_gera_lista_2.java (listas.pl) (5 x .class)		112	3.438	4.790
			PRED_size_2.java (listas.pl) (5 x .class)		117	3.423	4.759
			Total	17.258,20	649	19.169	23.332
family.holo	30	577	holo.java	117,20	88	2.231	2.601
			PRED_familia_2.java (familia.pl) (8 x .class)	15.854,40	134	3.721	5.947
			PRED_pai_2.java (familia.pl) (11 x .class)		178	5.433	8.838
			PRED_mae_2.java) (familia.pl) (11 x .class)		178	5.427	8.832
			Total	15.971,60	578	16.812	26.218

Tabela 2. Hardware e software usados no experimento

Software	**Versão**
Conectiva Linux	versão 6.0
HoloJava	Versão 1.0
Jada	Versão 3.0 beta 7
Java	Versão 1.3.1
Prolog Café	Versão 0.44
Hardware – Configuração	
Intel Pentium II – 233 MHz – 64 MBytes Ram	

Tabela 3. Índices de conversão

Índice	**Descrição**	**Holoprogramas**					**Média**
		performance	*semaphores*	*philosophers*	*buffers*	*travel*	
JHL (*Java/Holo Lines*)	Número de linhas em Java geradas para cada linha no holoprograma	2,89	2,06	2,16	3,22	2,10	2,49
JHB (*Java/Holo Bytes*)	Número de bytes nos arquivos Java para cada byte no holoprograma	4,04	2,46	2,89	3,68	2,60	3,14
CHB (*Class/Holo Bytes*)	Número de bytes nos arquivos *.class* para cada byte no holoprograma	5,11	3,95	3,70	4,41	3,12	4,10
HJLT (*Holo/Java Lines Time*)	Tempo de conversão (ms) de uma linha em Holo para Java	2,37	2,53	2,70	2,40	1,72	2,34
HJBT (*Holo/Java Bytes Time*)	Tempo de conversão (ms) de um byte em Holo para Java	0,12	0,14	0,13	0,12	0,07	0,12

4 Trabalhos Relacionados

Nos últimos anos vem crescendo o uso da plataforma Java para implementação de linguagens declarativas. Na programação funcional existem diversos exemplos dessa abordagem. Por exemplo, vários compiladores para a linguagem Haskell [19] usam Java como linguagem intermediária. Wakeling [33] implementa um ambiente de execução para linguagens funcionais usando Java. Os programas Haskell são convertidos para classes Java que utilizam o ambiente. Du Bois [12] usa a mesma abordagem, convertendo programas em Haskell para componentes JavaBeans.

Existem linguagens multiparadigma que são baseadas na estratégia de conversão. Por exemplo, Hanus [16] propôs a linguagem Curry que permite o uso conjunto da lógica e funções. A compilação de Curry gera *byte code* Java e o uso de *threads* suporta a concorrência e o não-determinismo naturais da linguagem. O modelo I^+ suporta objetos que implementam métodos usando funções e predicados lógicos. Funções são convertidas para Lazy ML (LML) e predicados para módulos Prolog. Ciampolini et al [9] criaram DLO, um sistema que suporta objetos lógicos distribuídos. Os programas DLO são convertidos para uma linguagem em lógica concorrente denominada Rose.

Tarau propôs Jinni [30] uma linguagem de programação em lógica que suporta concorrência, mobilidade e *blackboards* distribuídos. Jinni é implementada usando BinProlog um *multi-threaded* Prolog que suporta geração de código C/C++. Além disso, existe suporte para geração de Java.

5 Conclusões

Este artigo propôs o uso de Java como linguagem intermediária para compiladores multiparadigma. Neste sentido, o texto descreveu a HoloJava, um sistema para conversão da Hololinguagem para Java. HoloJava busca um mapeamento direto entre as linguagens. No entanto, Holo suporta uma maior abstração do que Java. Sendo assim, tornou-se necessário o uso de bibliotecas especiais para tratamento das ações lógicas (Prolog Café [5]) e *blackboards* (Jada [10]). A baixa eficiência da conversão realizada pelo Prolog Café é uma constatação que merece destaque.

Futuros trabalhos aperfeiçoarão a proposta. O paradigma funcional será integrado à Hololinguagem [11]. A tecnologia da linguagem Fun [12] suportará a integração. Fun é uma pequena linguagem funcional compilada para Java e executada usando uma G-Machine [3] implementada em Java. Além disso, melhorias serão introduzidas para suporte às características da Hololinguagem [6] ainda não implementadas, tais como, clonagem múltipla e clonagem seletiva. Atualmente, o Holoparadigma está sendo aplicado na computação móvel [4]. Neste contexto, a HoloJava receberá aperfeiçoamentos para implementação deste tipo de aplicação. Merecem destaque ainda, os esforços para criação de um depurador para a linguagem Holo (*HoloDebugger*). Esta ferramenta será baseada em adaptações da HoloJava para geração de informações de depuração. Finalmente, torna-se importante a criação de uma ferramenta para conversão de Prolog para Java que permita maior eficiência do que o Prolog Café.

Referências Bibliográficas

[1] 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.

[2] Apt, R. et al. Alma-0: An Imperative Language that Supports Declarative Programming. *ACM Transactions on Programming Languages and Systems*, New York, v.20, September 1998.

[3] Augustsson, L. A. Compiler for Lazy ML. *ACM Symposium on Lisp and Functional Programming*, Austin, p.218-227, 1984.

[4] Augustin, I.; Yamin, A. C.; Barbosa, J. L. V.; Geyer, C. F. R. Towards a Taxonomy for Mobile Applications with Adaptive Behavior. *International Symposium on Parallel and Distributed Computing and Networks* (PDCN), Innsbruck, February 2002.

[5] Banbara, M.; Tamura, N. Translating a Linear Logic Programming Language into Java. Workshop on Parallelism and Implementation Technology (Constraint) Logic Programming Languages, Las Cruces, p. 19-39, December 1999.

[6] Barbosa, J. L. V.; Geyer, C. F. R. *Uma Linguagem Multiparadigma Orientada ao Desenvolvimento de Software Distribuído*. V Simpósio Brasileiro de Linguagens de Programação (SBLP), Curitiba, 2001.

[7] Barbosa J. L. V.; Yamin, A. C.; Vargas, P. K.; Augustin, I.; Geyer, C. F. R. *Holoparadigm: a Multiparadigm Model Oriented to Development of Distributed Systems International Conference on Parallel and Distributed Systems* (ICPADS), Jung-Li City, New York, IEEE Press, 2002.

[8] Briot, Jean-Pierre; Guerraoui, Rachid; Löhr, Klaus-Peter. Concurrency and Distribution in Object-Oriented Programming. *ACM Computing Surveys*, New York, v.30, n.3, p.291-329, September 1998.

[9] Ciampolini, A.; Lamma, E.; Stefanelli, C; Mello, P. Distributed Logic Objects. *Computer Languages*, v.22, n.4, p.237-258, December 1996.

[10] Ciancarini, P.; Rossi, D. *JADA: A Coordination Toolkit for Java*. http://www.cs.unibo.it/~rossi/jada, 2003.

[11] Du Bois, A.; Barbosa, J. L. V.; Geyer, C. F. R. *Adding Functional Programming into the Hololanguage*. Functional and (Constraint) Logic Programming (WFLP), Kiel, p.45-58, September 2001.

[12] Du Bois, A. R.; Costa, A. C. R. *Distributed Execution of Functional Programs using the JVM*. XIII International Conference on Computer Aided Systems Theory (EUROCAST), Las Palmas, Spain, 2001.

[13] Hailpern, B. Multiparadigm Research: A Survey of Nine Projects. *IEEE Software*, New York, v.3, n.1, p. 70-77, January 1986.

[14] Hardwick, J. *Java as an Intermediate Language*. School of Computer Science, Carnegie Mellon University, Technical Report CMU-CS-96-161, August 1996.

[15] Hanus, M. The Integration of Functions into Logic Programming from Theory to Practice. *Journal of Logic Programming*, New York, v.19/20, p.583-628, May/July 1994.

[16] Hanus, M. Sandre, R. An Abstract Machine for Curry and its Concurrent Implementation in Java. *Journal of Functional and Logic Programming*, New York, v.6, 1999.

[17] Haridi, S. et al. Programming Languages for Distributed Applications. *New Generating Computing*, v.16, n.3, p.223-261, 1998.

[18] Horb. http://horb.a02.aist.go.jp/horb, 2003.

[19] Hudak, P.; Peterson, J.; Fasel, J. H. *A Gentle Introduction to Haskell – Version 1.4*, http://www.haskell.org, 2003.

[20] Java. http://www.sun.com/java, 2003.

[21] JavaCC – *The Java Parser Generator*. http://www.experimentalstuff.com/Technologies/JavaCC, 2003.

[22] JavaSpaces. http://www.sun.com/jini/specs/jini1.1html/js-title.html, 2003.

[23] Lea, D. *Objects in Groups*. http://gee.cs.oswego.edu/dl/groups, 2003.

[24] 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.

[25] Muller, M.; Muller, T.; Van Roy, P. *Multiparadigm Programming in Oz*. Visions for the Future of Logic Programming: Laying the Foundations for a Modern Successor of Prolog, 1995.

[26] Ng, K. W.; Luk, C. K. I+: A Multiparadigm Language for Object-Oriented Declarative Programming. *Computer Languages*, v.21, n.2, p. 81-100, July 1995.

[27] 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.

[28] Pineda, A.; Hermenegildo, M. *O'CIAO: An Object Oriented Programming Model Using CIAO Prolog*. Technical report CLIP 5/99.0 , Facultad de Informática, UMP, July 1999.

[29] Van Roy, P. et al. Mobile Objects in Distributed Oz. *ACM Transactions on Programming Languages and Systems*, v.19, n.5, p.804-851, September 1997.

[30] Tarau, P. *Jinni: Intelligent Mobile Agent Programming at the Intersection of Java and Prolog*. PAAM, The Practical Applications Company, 1999.

[31] Vargas, P. K.; Barbosa, J. L. V.; Ferrari, D.; Geyer, C. F. R.; Chassin, J. *Distributed OR Scheduling with Granularity Information*. XII Simpósio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho, 2000.

[32] Voyager. http://www.recursionsw.com/products/voyager/voyager.asp, 2003.

[33] Wakeling, D. Compiling Lazy Functional Programs for the Java Virtual Machine. *Journal of Functional Programming*, v.1, n.1, January 1998.