DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Relatório Técnico

RT-MAC-2001-13

O DESENVOLVIMENTO DE UM INTERPRETADOR ORIENTADO A OBJETOS PARA ENSINO DE LINGUAGENS TOWARDS A LOGIC OF PERISHABLE PROPOSITIONS

ALAN M. DURHAM

Dezembro de 2001

O Desenvolvimento de um Interpretador Orientado a Objetos para Ensino de Linguagens

6 de dezembro de 2001

Resumo

No ensino de linguagens de programação o que se busca é realçar as diferenças e as semelhanças entre os vários paradigmas. Com o objetivo de auxiliar nessa tarefa, Kamin [3] desenvolveu um interpretador que é modificado a cada estudo de paradigma. Esse artigo apresenta uma arquitetura orientada a objetos para esse interpretador, o que torna sua estrutura ainda mais clara e fácil de ser modificada.

1 Introdução

Este trabalho apresenta um interpretador desenvolvido pelos alunos de pós-graduação do curso de Paradigmas de Programação do Departamento de Ciência da Computação do IME-USP. Este interpretador foi conseqüência da reengenharia do interpretador desenvolvido por Kamin [3]. Esta reengenharia acarretou em uma arquitetura orientada a objetos que facilita o ensino da estrutura de interpretadores de linguagens de programação, deixando claras as semelhanças e realçando as diferenças na implementação dos vários paradigmas. A estrutura faz uso de vários padrões de projeto orientado a objetos [2], o que, além de facilitar seu entendimento, mostra uma arquitetura Orientada a Objetos bem estruturada.

Foram desenvolvidas classes para interpretadores de uma linguagem básica baseada em Lisp, uma linguagem funcional com fechamentos ("closures") baseada em Scheme, e uma linguagem funcional com suspensões ("thunks"),

baseada em SASL. Também indicamos o que precisaria ser acrescentado para a elaboração de classes para implementar uma linguagem orientada a objetos.

Na próxima seção descreveremos brevemente um exemplo de linguagem de programação para ilustrar o funcionamento do interpretador, na seção 3 descreveremos o interpretador básico, na seção 4 as classes necessárias para implementação de linguagens funcionais, na seção 5 as classes para implementar suspensões, na seção 6 discutimos as extensões necessárias para implementação de objetos e, na seção 7 analisamos a estrutura do interpretador salientando suas vantagens e delineamos pesquisa futura.

2 Um exemplo da linguagem

Abaixo segue um trecho de programa que será referenciado nas seções seguintes para ilustrar o funcionamento do interpretador para cada tipo de linguagem abordado nesse artigo[1].

```
->(define func1 (x y)
        (+ x y))
func1
->(set x 20)
20
->(+ 3 x)
23
->(func1 3 20)
23
->(func1 (+ 3 x) (func1 5 6))
34
->(if (< 3 x) 3 x)
3
```

No exemplo acima, o "prompt" do interpretador é a sequência "->", enquanto as demais linhas representam a saída do resultado. O interpretador aceita dois tipos de entrada, definições e operações, sempre imprimindo algo como resultado. Em nosso caso, a primeira entrada é uma definição de função, de nome `func1`. Após a função ser digitada, ela passa a fazer parte

[1]Nosso exemplo assume a sintaxe (ou falta de) do Lisp, mas a tranformação de outro modo qualquer para a forma acima constitui mero açúcar sintático.

do ambiente de execução, e seu nome é impresso indicando que ela foi reconhecida. Em nosso exemplo `func1` foi descrita como uma função que retorna a soma dos parâmetros. A segunda entrada define uma variável `x` e associa a ela o valor 20.

As próximas três entradas ilustram operações que podem ser pedidas. No primeiro caso a aplicação da operação primitiva "+" aos números 3 e 20. No segundo a aplicação da função de usuário `func1` aos mesmos números e, no terceiro caso, a aplicação de `func1` ao resultado de duas outras expressões (obs: não há limite na quantidade de encaixamentos de expressões).

Finalmente a última entrada mostra a execução de um comando de controle de fluxo, aqui modelado também como uma operação primitiva.

3 O interpretador básico

Nosso interpretador é constituído por vários grupos de classes:

- Interpreter
- InterpreterContext
- Expression
- ExpressionParser
- Function
- FuctionParser
- Token
- TokenStream
- Environment

Os grupos essenciais para o funcionamento do interpretador em si são Interpreter, InterpreterContext, Expression e Environment. O interpretador é executado a partir de uma árvore sintática utilizando o padrão *Interpreter* [2]. Assim, a execução de código para cada expressão é feita pela própria representação intermediária da expressão. Esta árvore sintática é representada utilizando subclasses de *Expression*. A classe *Interpreter* funciona apenas

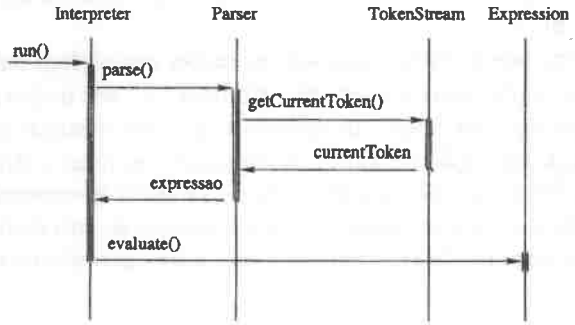

Figura 1: Funcionamento do Interpretador

como iniciadora do processo, cuidando da interação entre os vários componentes. A classe *Environment* encapsula a representação do estado das variáveis locais e globais do ambiente de execução da linguagem. A classe *InterpreterContext* foi a solução encontrada para tratar de maneira modular possíveis variações nos mecanismos de entrada e saída do interpretador. Finalmente, as classes *TokenStream*, *Token*, *FunctionParser* e *ExpressionParser* são utilizadas para leitura da entrada do usuário e geração da árvore sintática.

3.1 O Interpretador

A classe *Interpreter* é a classe "gerente" de nosso sistema, que coordena o funcionamento das várias partes do sistema. Para ativarmos o interpretador criamos um objeto desta classe e enviamos a mensagem `run()`. Este método utiliza a classe *TokenStream* para fazer a análise léxica da entrada, repassando os Tokens obtidos para a montagem da árvore sintática abstrada para a classe *ExpressionParser*. Uma vez obtida a árvore sintática, é pedida a execução do código emitindo a mensagem `evaluate()` para o nó raiz da árvore. O valor retornado é impresso e o processo reiniciado até o final da entrada.

A figura 1 representa a interação entre as principais partes do sistema.

Durante o desenvolvimento do sistema, um dos problemas que se colocou

era como tratar de maneira modular a possível variação de dispositivos de entrada do interpretador[2]. Porém este pode ser visto como um caso particular de um problema mais geral que seria como tratar de maneira modular possíveis variações no ambiente de execução do interpretador, variações estas que não afetem o funcionamento da linguagem. A solução foi a criação da classe *InterpreterContext*. No momento esta classe lida apenas com variação nos dispositivos de entrada e saída, mas isso não limita a ampliação deste domínio para, por exemplo, configuração de comandos de controle do interpretador (poderíamos redifinir qual comando encerraria sua execução). Esta divisão se encaixa um pouco no espírito do padrão *State*. Desta maneira, *Interpreter Context* é a classe responsável pela execução da entrada e saída do interpretador, sendo que em sua interface temos métodos que permitem a alteração de seu funcionamento a partir de mensagens emitidas por *Interpreter*.

Para a classe *Interpreter*, bem como para a classe *InterpreterContext*, utilizamos o padrão *Singleton* [2], utilizado para garantir a existência de apenas um objeto destas classes para cada ativação do interpretador.

Como todo sistema de implementação de linguagens de programação, temos um subsistema para execução da análise léxica e sintática. Estas tarefas são executadas por instâncias das hierarquias das classes *Token* e *TokenStream* para a análise léxica, e *FunctionParser* e *ExpressionParser* para a análise sintática.

3.2 Expressões

A figura 2 mostra a hierarquia completa de *Expression*. Esta hierarquia é fundamental para o entendimento do interpretador e foi projetada com o objetivo de otimizar a reutilização de código, colocando em evidência as semelhanças de implementação entre os vários elementos de uma linguagem. Vejamos em mais detalhe.

A classe raiz desta hierarquia é a classe abstrata *Expression*, que representa qualquer entidade que possa ser avaliada. Seus dois métodos definem bem sua função: `evaluate()`, para retornar o valor da expressão e `toString()` para retornar um valor imprimível identificando a expressão. O método `toString()` é bastante utilizado no retorno das entradas do inter-

[2]No caso, queríamos que este pudesse de maneira alternada utilizar tanto a entrada pelo teclado como por um arquivo.

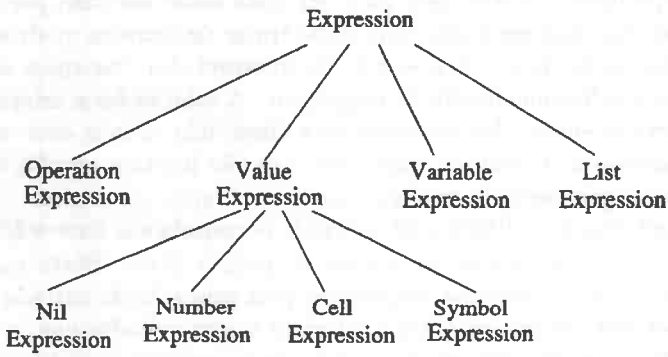

Figura 2: Hierarquia de Expressões

pretador.

A classe *Expression* possui cinco subclasses importantes: *OperationExpression*, representando todas as operações, *ValueExpression*, representando constantes de tipos primitivos, *VariableExpression*, representando o uso de uma variável, *Function*, representando as definições de funções e, finalmente, *ListExpression*, implementando o padrão *Composite* [2] que permite que encaixamentos de expressões sejam visto como uma expressão.

3.2.1 Valores Primitivos

Tipos primitivos são talvez uma das partes mais importantes da descrição de uma linguagem. Estes definem a funcionalidade disponível ao programador para descrever novos processos e definem uma visão de máquina por ele utilizada. Além disso a maior parte da variação entre linguagens de programação se encontra na definição de tipos primitivos [1].

Em nosso interpretador os tipos primitivos são implementados como subclasses da classe *ValueExpression*. Esta serve para definir o tratamento de erros de tipo de operações em tempo de execução, definindo todas as mensagens que suas "filhas" definem e levantando exceções para cada mensagem. Cada uma das sub-classes de *ValueExpression* implementa um tipo primitivo. Esta estrutura é bem modular, pois possibilita acrescentar novos tipos

primitivos colocando novas subclasses de maneira independente, e apenas acrescentando métodos em *ValueExpression* caso haja necessidade de tratamento dinâmico de erros diferenciado para operações inválidas.

Em nosso interpretador básico temos subclasses para células (*CellExp*), para o valor nil (*NilExp*), números inteiros (*NumberExp*) e símbolos (*SimbolExp*).

3.2.2 Variáveis

A classe *VariableExpression* é provavelmente a de funcionamento mais simples, mas sua implementação demonstra bem a importância do ambiente de execução e a distinção entre a variável e seu valor. A função dos objetos desta classe é apenas guardar o nome da variável, o valor é obtido utilizando-se o ambiente de execução (*Environment*) no momento da avaliação. Separando claramente a implementação de variáveis e o armazenamento de seus valores, a distinção de funcionamento de escopo estático e dinâmico fica mais facilmente delineada.

3.2.3 Operações

A classe *OperationExpression* modela todo tipo de expressão que acarreta em algum cálculo. Na verdade esta estrutura é um pouco semelhante a um *Composite* [2] de outras expressões, a diferença é que a primeira das "sub-expressões" tem significado diferenciado, que é o de representar a descrição de uma função: se ela é primitiva ou não.

No funcionamento desta classe é determinado o cálculo dos argumentos de uma operação qualquer, seja ela primitiva ou definida pelo usuário. Uma das vantagens do tratamento dado aqui é ressaltar a semelhança de abordagem de um operador qualquer da linguagem (ou função primitiva) com uma função definida por usuário. Como veremos mais tarde, ambos serão modelados como sub-classes da mesma classe-mãe.

3.3 Funções

O objetivo desta classe é modelar a descrição de todos os procedimentos executáveis de uma linguagem, sejam eles funções primitivas, operações primitivas ou funções de usuário. A classe *Function* é abstrata e possui duas subclasses importantes: *PrimitiveFunction* e *UserFunction*. A primeira é a

classe abstrata de toda a funcionalidade primitiva de uma linguagem. Nossa abordagem segue a filosofia do interpretador original de Kamin, tratando inclusive operadores de controle (if, while, etc.), como funções primitivas. Isto não impede que este arcabouço seja utilizado para interpretar uma linguagem não funcional, uma vez que efeitos colaterais podem ser implementados. Por outro lado, com esta abordagem a visão de implementação da funcionalidade primitiva fica mais homogênea, favorecendo o entendimento.

O principal método da interface é `apply()`, que recebe como argumentos o ambiente ("environment") que define o valor das variáveis, e expressões que definem o valor dos parâmetros da entidade funcional (argumentos a serem utilizados para execução da funcionalidade). A utilização de escopo dinâmico ou escopo estático fica a cargo da implementação de `apply()` nas subclasses.

3.4 O ambiente de execução

O ambiente de excução (ou "environment") é o conjunto de valores de variáveis, parâmetros de execução e funções definidas pelo usuário. Podemos ver o ambiente como um mapeamento de nomes em valores (subclasses de *ValueExpression*) ou em definição de procedimentos (classe *UserFunction*). No interpretador básico dois tipos de ambientes são utilizados, um para os valores e outro para as definições de função. No nosso exemplo, após o segundo comando o ambiente associa "x" a 20 e "func1" a função de dois parâmetros que devolve sua soma.

Em nosso sistema, o ambiente de execução é implementado pela classe *Environment*. A separação do ambiente de execução em uma classe distinta é fundamental para o entendimento do funcionamento de linguagens de programação. Dificilmente se pode reclamar alguma originalidade na abordagem, contudo, no desenvolvimento de nossa classe, conseguimos separar a construção do escopo da busca por valores do ambiente. A distinção entre escopo estático (ex. Scheme e quase todas as linguagens tradicionais) e dinâmico (ex. Lisp), é feita na criação dos objetos da classe. A interface de acesso é independente desta construção, e o acesso aos valores é feito sempre a partir do ambiente local, encapsulando o funcionamento da busca. Isso fica claro na interface definida da classe:

- `Environment(Environment), Environment()`
- `isLocallyBound(String):boolean`

Figura 3: Ambientes na execução de `func1`

- `hasDefinitionFor(String):boolean`
- `defineLocalEntry(String):void`, `defineLocalEntry(String,Object):void`
- `defineGlobalEntry(String):void`, `defineGlobalEntry(String,Object):void`
- `setEntryValue(String,Object):void`
- `getEntryValue(String):Object`

Pode parecer estranho a interface permitir a definição tanto de variáveis locais quanto de variáveis globais a partir do ambiente local, porém isso acontece porque, em algumas versões do Interpretador utilizado, as únicas variáveis locais eram os argumentos e qualquer ocorrência de uma nova variável definia uma variável global. Isto gerava uma situação onde, apesar de existir um ambiente local, precisávamos definir uma variável global (em um dos casos, definição de funções). Acesso a valores de variáveis, por outro lado, é sempre determinado a partir do encadeamento dos ambientes locais.

No nosso caso, durante a execução da função `func1` teríamos, para as variáveis, dois ambientes concatenados. A figura 3 mostra esses ambientes durante a primeira execução de `func1`.

4 Fechamentos: a implementação de Scheme

A implementação de Fechamentos ("closures"), isto é, de funcionais como valores de primeira ordem, implica em uma mudança do interpretador, necessitando de uma nova versão. Agora as antigas "funções" são valores da linguagem e, portanto devem ser sublcasses de *ValueExpression*. Porém, em

muitos aspectos sua implementação é idêntica a hierarquia da classe *Function*, inclusive com subclasses para fechamentos primitivos e fechamentos definidos por usuários. Assim, as subclasses de fechamentos primitivos nada mais são do que as funções primitivas implementadas como herdeiras da classe *PrimitiveClosure*. Uma diferença importante é que a classe de fechamentos de usuário (*UserClosure*) tem uma variável de instância a mais, o "Environment" ativo quando da criação do fechamento.

Em Scheme há uma operação primitiva chamada *Lambda*, que não está implementada no interpertador básico. O lambda tem uma função semelhante ao *define*[3] do interpretador básico, no entanto o lambda retorna um funcional como valor de primeira ordem que pode ou não ser associado a uma variável.

A utilização das operações primitivas não difere do interpretador básico. Já a definição de uma função em nosso interpretador Scheme é feita atribuindo-se o fechamento a uma variável. A definição da função `func1` pode ser realizada da seguinte forma:

```
->(set func1 (lambda (x y) (+ x y)))
<Closure>

-> (func1 3 20)
23
```

Como um fechamento agora é um valor de primeira ordem, a classe *OperationExpression* tornou-se uma subclasse de *ValueExpression*. No trecho abaixo[4], temos um exemplo da possibilidade de escrever fechamentos que retornam fechamentos.

```
->(set soma (lambda (x)
                    (lambda (y) (+ x y)))
  )
<Closure>

-> (set soma2 (soma 2))
<Closure>
```

[3] A operação *define* não é suportada no Scheme.

[4] Vale esclarecer que o "prompt" do interpretador só é liberado para impressão do resultado após todos os encadeamentos abertos serem fechados.

```
-> (soma2 5)
7
```

Foi criada a classe *LambdaParser* para a análise sintática do operador "lambda" com o qual há a definição de fechamentos pelo usuário. Essa classe substitui a antiga *FunctionParser* do interpretador básico.

5 Suspensões - avaliação por demanda

Em nossa segunda extensão ao interpretador inicial, precisávamos modelar a avaliação por demanda ("lazy evaluation") de SASL. Isso quer dizer que algumas expressões não são calculadas ao serem encontradas, mas apenas quando seu valor é necessário. Em particular argumentos de uma função e expressões para construção de células ("cons") são suspensos até que estes valores sejam utilizados na implementação de alguma operação. Suspensões são implementadas pela classe *Thunk*. Uma suspensão pode aparecer em qualquer lugar que um valor aparece, o que determinou sua colocação como subclasse de *ValueExpression*. Na verdade sua interface é também idêntica. Na visão que o usuário tem das implementações as suspensões são criadas nas situações mencionadas acima e, quando seu valor é requisitado inicialmente, são substuídas pelo valor das expressões que elas representam. Todas as referências a suspensão original são automaticamente substituídas por referências ao valor calculado. Como a linguagem de implementação escolhida, Java, não possui uma operação primitiva para substituição de um objeto por outro nas referências do sistema[5], a solução foi tranformar objetos desta classe em "caches" dos valores. Na primeira tentativa de acessar o valor da expressão esta é calculada e seu valor guardado internamente. Nas outras apenas o valor anteriormente calculado é retornado.

5.1 Modificações no Interpretador básico

Basicamente, a única nova classe agregada é Thunk. As outras foram modificadas a partir da versão do interpretador básico.

[5]Como, por exemplo, Smalltalk-80 faz com seu método **becomes:**

5.1.1 Classes envolvidas

- *Car*
  Esta classe, agora herda de *PrimitiveClosure* e não mais de *PrimitiveFunction*. A diferença da versão básica é que agora o método `evaluate()` avalia o "car"obtido da *ListExpression*, substituindo a suspensão na *CellExpression* por seu valor já avaliado.

- *Cdr*
  Analogamente à classe *Car*, esta classe agora herda de *PrimitiveClosure*. Também é processada a possível suspensão no "cdr"antes de ser retornada e substituída na "cache" da *CellExpression*.

- *CellExpression*
  Nesta nova versão é permitido construir objetos usando como argumento cdr uma "thunk". Na versão básica somente eram permitidas *NilExpression* e *CellExpression*.

- *Cons*
  Esta nova versão muda a forma em que é construída a nova *CellExpression*: na versão básica do interpretador, o car e o cdr recebidos como argumentos são avaliados antes de ser construída a *CellExpression*. Nesta versão, são criadas suspensões, isto é, a avaliação destes argumentos é postergada até ser estritamente necessária.

- *Expression*
  A principal diferença é o acréscimo do método

  ```
  delayEval( rho: Environment ): Expression
  ```

  Este tem a responsabilidade de "suspender"expressões, isto é, criar "thunks" que postergarão a avaliação da instância de *Expression* até que seu valor seja necessário.

- *ListExpression*
  As mudanças são restritas a implementar o método `delayEval` e agregar o método `delayEvalNro`. A diferença das versões de `evaluate` e `evalNro`, onde é criada uma *ListExpression* contendo as expressões já avaliadas, é que agora esta lista estará formada pelo resultado de invocar a `delayEval` em cada sub-expressão.

- *NilExpression*
  Simplesmente é agregada uma implementação para o método `delayEval`. Esta implementação retorna o objeto "this".

- *NumberExpression*
  Não apresenta mais modificações que a implementação do método *delayEval*, método no qual é retornado simplesmente this.

- *OperationExpression*
  A diferença da classe original é que nesta versão são armazenados no repositório de definições de funções não instancias de *Function*, mas de *Expression* que serão avaliadas no momento anterior a executar `apply`.

- *SymbolExpression*
  Só é agregada a implementação do método `delayEval` com sua implementação mínima, isto é, retornando "this".

- *Thunk*
  Esta é a nova classe e o coração desta implementação. Ao ser subclasse de *ValueExpression*, é um valor de primeira ordem. Instâncias desta classe sempre referem-se a expressão original e ao Environment do momento em que foi criada. Também controlam o fato de ter sido esta expressão já avaliada ou não. Desta forma é evitada a reavaliação da expressão referenciada e é aproveitado o valor já avaliado.

- *VariableExpression*
  As mudanças são duas: primeiro, e como em toda subclasse de *Expression*, é implementado o método `delayEval`. Neste método simplesmente é retornado o valor da variável no *Environment*, sem avaliá-lo. E finalmente, é modificado o método `evaluate`. Originalmente era retornado o valor existente no *Environment*. Agora, se diferencia se é uma "thunk" ou não. No caso de não ser uma "thunk", será tratado como anteriormente. Se for uma "thunk", esta será avaliada e o novo valor substituirá o anterior, antes de ser retornado.

6 Para implementar Orientação a Objetos

Apesar de um interpretador para linguagem orientada a objetos não ter sido implementado, podemos delinear como isso seria feito. Precisaríamos de-

finir três novas classes: *MessageSend*, *InterpreterObject* e *InterpreterClass*. A primeira, *MessageSend* deveria fazer parte da hierarquia de *Operation*[6]. Porém, neste caso o primeiro argumento seria diferenciado, devendo ser um *InterpreterObject*.

Um *InterpreterObject* é a representação de um objeto e deve conter, além de um ambiente com as variáveis de instância, uma referência a um objeto da classe *InterpreterClass*. A classe *InterpreterClass* contém um ambiente (classe *Environment*), pois seu papel é servir de repositório para os fechamentos que constituem os métodos aplicáveis a um objeto. Instâncias de *InterpreterClass* podem possuir referências a uma ou mais instâncias da mesma classe, implementando herança simples ou múltipla. Métodos de uma classe podem ser modelados simplesmente como fechamentos, sendo seu nome a chave da entrada do ambiente ao qual estão associados.

É importante notar que o trabalho para implementação de um novo paradigma é relativamente pequeno, exigindo apenas 3 classes novas de funcionalidade relativamente simples. A implementação de Classes como valores de primeira ordem (como em Smalltalk), implicaria colocá-las dentro da hierarquia de *ValueExpression*, novamente ressaltando as implicações em termos de implementação de características novas da linguagem.

7 Discussão

Podemos notar que a estrutura modular da arquitetura desenvolvida facilita o incremento das linguagens. A maior parte da complexidade das linguagens está nos tipos primitivos [1]. Assim, a separação clara desses tipos e a facilidade de acrescentar novos tipos primitivos simplificados torna trivial o acréscimo de mais linguagens.

Da mesma forma a implementação de funções primitivas ficou mais simples. Basta chamar a operação adequada dos tipos primitivos. A desvantagem é a dependência criada entre tipos primitivos com sobrecarga de operadores, mas isso é inevitável.

Outra vantagem dessa estrutura é que ela valoriza e realça quais são valores de primeira ordem da linguagem (subclasses de *ValueExpression*), bem como valoriza semelhanças e diferenças entre operações primitivas e definidas pelo usuário. A única diferença é que, no caso de operações primitivas,

[6]Ou, caso queiramos uma linguagem puramente orientada a objetos como Smalltalk, colocaríamos *MessageSend* no lugar de *OperationExp*

a descrição do funcionamento está embutida no código do intepretador.

¿¿¿¿¿¿Estrutura da árvore é de arvore sintática abstrata, formato da entrada pode ser mudado.

Uso de padrões e "double dispatch" mostra boa arquitetura OO de um sistema.

Mostra que linguagens mudam apenas no tipo de valor funcional (função, vs closure vs mensagem), e estilo de avaliação (delayed vs. gulosa).

A grande vantagem dessa arquitetura é que o modelo de operação fica claro a partir da própria estrutura de classes. Operações semelhantes com visões diferentes (ex. if em Lisp e em OO) tem suas diferenças ressaltadas pela criação de subclasses da mesma função primitiva.

Referências

[1] Alan M. Durham and Ralph Johnson. A system to implement primitive data types. *Revista da Sociedade Brasileira de Computação*, 1999.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Software.* Addison Wesley, 1995.

[3] Samuel Kamin. *Programming Languages: An Interpreter-based Approach.* Addison Wesley, 1988.

RELATÓRIOS TÉCNICOS

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Instituto de Matemática e Estatística da USP

A listagem contendo os relatórios técnicos anteriores a 1997 poderá ser consultada ou solicitada à Secretaria do Departamento, pessoalmente, por carta ou e-mail (mac@ime.usp.br).

Flávio Soares Corrêa da Silva e Daniela Vasconcelos Carbogim
A TWO-SORTED INTERPRETATION FOR ANNOTATED LOGIC
RT-MAC-9801, fevereiro de 1998, 17 pp.

Flávio Soares Corrêa da Silva, Wamberto Weber Vasconcelos, Jaume Agustí, David Robertson e Ana Cristina V. de Melo.
WHY ONTOLOGIES ARE NOT ENOUGH FOR KNOWLEDGE SHARING
RT-MAC-9802, outubro de 1998, 15 pp.

J. C.de Pina e J. Soares
ON THE INTEGER CONE OF THE BASES OF A MATROID
RT-MAC-9803, novembro de 1998, 16 pp.

K. Okuda and S.W.Song
REVISITING HAMILTONIAM DECOMPOSITION OF THE HYPERCUBE
RT-MAC-9804, dezembro 1998, 17pp.

Markus Endler
AGENTES MÓVEIS: UM TUTORIAL
RT-MAC-9805, dezembro 1998, 19pp.

Carlos Alberto de Bragança Pereira, Fabio Nakano e Julio Michael Stern
A DYNAMIC SOFTWARE CERTIFICATION AN VERIFICATION PROCEDURE
RT-MAC-9901, março 1999, 21pp.

Carlos E. Ferreira e Dilma M. Silva
BCC DA USP: UM NOVO CURSO PARA OS DESAFIOS DO NOVO MILÊNIO
RT-MAC-9902, abril 1999, 12pp.

Ronaldo Fumio Hashimoto and Junior Barrera
A SIMPLE ALGORITHM FOR DECOMPOSING CONVEX STRUCTURING ELEMENTS
RT-MAC-9903, abril 1999, 24 pp.

Jorge Euler, Maria do Carmo Noronha e Dilma Menezes da Silva
ESTUDO DE CASO: DESEMPENHO DEFICIENTE DO SISTEMA OPERACIONAL LINUX PARA CARGA MISTA DE APLICAÇÕES.
RT-MAC-9904, maio 1999, 27 pp.

Carlos Humes Junior e Paulo José da Silva e Silva
AN INEXACT CLASSICAL PROXIMAL POINT ALGORITHM VIEWED AS DESCENT METHOD IN THE OPTIMIZATION CASE
RT-MAC-9905, maio 1999, pp.

Carlos Humes Junior and Paulo José da Silva e Silva
STRICT CONVEX REGULARIZATIONS, PROXIMAL POINTS AND AUGMENTED LAGRANGIANS
RT-MAC-9906, maio 1999, 21 pp.

Ronaldo Fumio Hashimoto, Junior Barrera, Carlos Eduardo Ferreira
A COMBINATORIAL OPTIMIZATION TECHNIQUE FOR THE SEQUENTIAL DECOMPOSITION OF EROSIONS AND DILATIONS
RT-MAC-9907, maio 1999, 30 pp.

Carlos Humes Junior and Marcelo Queiroz
ON THE PROJECTED PAIRWISE MULTICOMMODITY FLOW POLYHEDRON
RT-MAC-9908, maio 1999, 18 pp.

Carlos Humes Junior and Marcelo Queiroz
TWO HEURISTICS FOR THE CONTINUOUS CAPACITY AND FLOW ASSIGNMENT GLOBAL OPTIMIZATION
RT-MAC-9909, maio 1999, 32 pp.

Carlos Humes Junior and Paulo José da Silva e Silva
AN INEXACT CLASSICAL PROXIMAL POINT ALGORITHM VIEWED AS A DESCENT METHOD IN THE OPTIMIZATION CASE
RT-MAC-9910, julho 1999, 13 pp.

Markus Endler and Dilma M. Silva and Kunio Okuda
A RELIABLE CONNECTIONLESS PROTOCOL FOR MOBILE CLIENTS
RT-MAC-9911, setembro 1999, 17 pp.

David Robertson, Fávio S. Corrêa da Silva, Jaume Agustí and Wamberto W. Vasconcelos
A LIGHTWEIGHT CAPABILITY COMMUNICATION MECHANISM
RT-MAC-9912, novembro 1999, 14 pp.

Flávio S. Corrêa da Silva, Jaume Agustí, Roberto Cássio de Araújo and Ana Cristina V. de Melo
KNOWLEDGE SHARING BETWEEN A PROBABILISTIC LOGIC AND BAYESIAN BELIEF NETWORKS
RT-MAC-9913, novembro 1999, 13 pp.

Ronaldo F. Hashimoto, Junior Barrera and Edward R. Dougherty
FINDING SOLUTIONS FOR THE DILATION FACTORIZATION EQUATION
RT-MAC-9914, novembro 1999, 20 pp.

Marcelo Finger and WanbertoVasconcelos
SHARING RESOURCE-SENSITIVE KNOWLEDGE USING COMBINATOR LOGICS
RT- MAC-2000-01, março 2000, 13pp.

Marcos Alves e Markus Endler
PARTICIONAMENTO TRANSPARENTE DE AMBIENTES VIRTUAIS DISTRIBUÍDOS
RT- MAC-2000-02, abril 2000, 21pp.

Paulo Silva, Marcelo Queiroz and Carlos Humes Junior
A NOTE ON "STABILITY OF CLEARING OPEN LOOP POLICIES IN MANUFACTURING SYSTEMS"
RT- MAC-2000-03, abril 2000, 12 pp.

Carlos Alberto de Bragança Pereira and Julio Michael Stern
FULL BAYESIAN SIGNIFICANCE TEST: THE BEHRENS-FISHER AND COEFFICIENTS OF VARIATION PROBLEMS
RT-MAC-2000-04, agosto 2000, 20 pp.

Telba Zalkind Irony, Marcelo Lauretto, Carlos Alberto de Bragança Pereira and Julio Michael Stern
A WEIBULL WEAROUT TEST: FULL BAYESIAN APPROACH
RT-MAC-2000-05, agosto 2000, 18 pp.

Carlos Alberto de Bragança Pereira and Julio Michael Stern
INTRINSIC REGULARIZATION IN MODEL SELECTION USING THE FULL BAYESIAN SIGNIFICANCE TEST
RT-MAC-2000-06, outubro 2000, 18 pp.

Douglas Moreto and Markus Endler
EVALUATING COMPOSITE EVENTS USING SHARED TREES
RT-MAC-2001-01, janeiro 2001, 26 pp.

Vera Nagamura and Markus Endler
COORDINATING MOBILE AGENTS THROUGH THE BROADCAST CHANNEL
RT-MAC-2001-02, janeiro 2001, 21 pp.

Júlio Michael Stern
THE FULLY BAYESIAN SIGNIFICANCE TEST FOR THE COVARIANCE PROBLEM
RT-MAC-2001-03, fevereiro 2001, 15 pp.

Marcelo Finger and Renata Wassermann
TABLEAUX FOR APPROXIMATE REASONING
RT- MAC-2001-04, março 2001, 22 pp.

Julio Michael Stern
FULL BAYESIAN SIGNIFICANCE TESTS FOR MULTIVARIATE NORMAL STRUCTURE MODELS
RT-MAC-2001-05, junho 2001, 20 pp.

Paulo Sérgio Naddeo Dias Lopes and Hernán Astudillo
VIEWPOINTS IN REQUIREMENTS ENGINEERING
RT-MAC-2001-06, julho 2001, 19 pp.

Fabio Kon
O SOFTWARE ABERTO E A QUESTÃO SOCIAL
RT- MAC-2001-07, setembro 2001, 15 pp.

Isabel Cristina Italiano, João Eduardo Ferreira and Osvaldo Kotaro Takai
ASPECTOS CONCEITUAIS EM DATA WAREHOUSE
RT – MAC-2001-08, setembro 2001, 65 pp.

Marcelo Queiroz , Carlos Humes Junior and Joaquim Júdice
ON FINDING GLOBAL OPTIMA FOR THE HINGE FITTING PROBLEM
RT- MAC –2001-09, novembro 2001, 39 pp.

Marcelo Queiroz , Joaquim Júdice and Carlos Humes Junior
THE SYMMETRIC EIGENVALUE COMPLEMENTARITY PROBLEM
RT- MAC-2001-10, novembro 2001, 33 pp.

Marcelo Finger, and Fernando Antonio Mac Cracken Cezar
BANCO DE DADOS OBSOLESCENTES E UMA PROPOSTA DE IMPLEMENTAÇÃO.
RT- MAC - 2001-11- novembro 2001, 90 pp.

Flávio Soares Correa da Silva
TOWARDS A LOGIC OF PERISHABLE PROPOSITIONS
RT- MAC- 2001-12 - novembro 2001, 15pp.

Alan M. Durham
O DESENVOLVIMENTO DE UM INTERPRETADOR ORIENTADO A OBJETOS PARA ENSINO DE LINGUAGENS
RT-MAC-2001-13 – dezembro 2001, 21 pp.