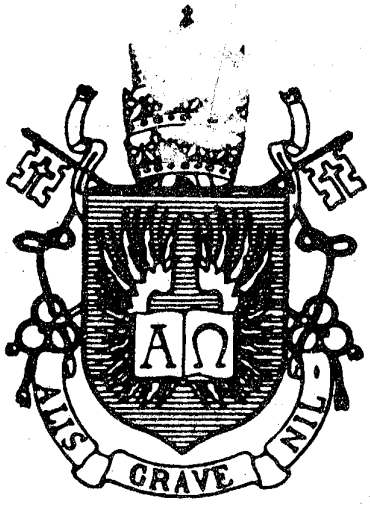

PUC

Série: Monografias em Ciência da Computação, Nº 4/87

ABSTRAÇÃO DE DADOS EM LINGUAGENS DE PROGRAMAÇÃO: TIPOS E OBJETOS

Roberto Ierusalimschy

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 – CEP 22453

RIO DE JANEIRO – BRASIL

PUC/RJ - DEPARTAMENTO DE INFORMÁTICA

Série: Monografias em Ciência da Computação
N: 4/87

Editor: Paulo A. S. Veloso

Dezembro, 1987

ABSTRAÇÃO DE DADOS EM LINGUAGENS DE PROGRAMAÇÃO: TIPOS E OBJETOS *

Roberto Ierusalimschy

* Apresentado por José Lucas M. Rangel Neto.

Trabalho parcialmente financiado pela FINEP e pela SID-Informática.

ABSTRAÇÃO DE DADOS EM LINGUAGENS DE PROGRAMAÇÃO: TIPOS E OBJETOS

Roberto Ierusalimschy

RESUMO

Este trabalho discute os conceitos de tipos e objetos enquanto mecanismos de abstração de dados em linguagens de programação, dando especial atenção às questões de tipagem e reutilização de código. Várias linguagens são citadas ao longo do trabalho, com particular ênfase em Ada, CLU e Smalltalk.

Palavras-chave: Linguagens de programação, abstração de dados, tipos de dados, programação orientada a objetos.

ABSTRACT

This work discusses data types and objects as data abstraction mechanisms in programming languages, with particular emphasis in software reutilization and typing. Many languages are referred to on such discussion, being Ada, CLU and Smalltalk investigated in more detail.

Keywords: Programming languages, data abstraction, data types, object oriented programming.

ABSTRAÇÃO DE DADOS EM LINGUAGENS DE PROGRAMAÇÃO: TIPOS E OBJETOS

1) Introdução

Uma das motivações principais para o uso de linguagens de alto nível é o conjunto de facilidades que estas fornecem para abstração. Em especial, são desejáveis nestas linguagens mecanismos de abstração de dados, que possibilitem ao programador tratar informação de uma forma mais próxima do mundo da aplicação, ao invés de uma forma imposta pela máquina.

Abstração de dados está intimamente relacionada com dois outros importantes conceitos de engenharia de software: "information hiding" e modularidade ([Fairley 85]). O primeiro diz respeito à característica de cada parte de um sistema esconder seus detalhes das outras partes, enquanto o segundo se refere à divisão do sistema em partes logicamente coesas e ao grau de dependência entre estas várias partes. Por um lado, "information hiding" é uma das formas mais comuns de se implementar abstração de dados, escondendo os detalhes de implementação de uma variável e mostrando apenas uma interface com operações de alto nível. Por outro lado, uma estrutura abstrata é uma ótima "unidade" de modularidade, sugerindo a criação de módulos reunindo estruturas de dados e operações sobre estas estruturas que formem abstrações úteis. Com este tipo de módulos conseguimos um mecanismo que une abstração de dados, "information hiding" e modularidade.

O objetivo deste trabalho é discutir alguns mecanismos de abstração de dados usados atualmente em linguagens de programação (LPs), mais especificamente os mecanismos de tipos e de objetos. Não pretendemos repetir tudo o que já foi dito a este respeito, mas dar um panorama

geral da área, ressaltando alguns pontos de particular interesse. Ao longo do texto existem indicações de bibliografia mais específica para os vários pontos abordados. Também não apresentamos nenhum resumo de manual das linguagens comentadas. Vários livros de LPs, entre eles [Ghezzi 85] e [Horowitz 84], apresentam excelentes descrições da maioria das linguagens abordadas neste texto. Além disto, sempre existem os manuais, que são citados nas referências.

Na seção seguinte são abordados os conceitos de tipos e tipagem. A seção 3 define objeto e discute o conceito Orientação a Objetos. Na seção 4 são tratados alguns mecanismos de reaproveitamento de código, como herança e polimorfismo. A seção 5 discute os mecanismos de abstração de algumas linguagens usuais, com especial atenção para CLU, Ada e Smalltalk. Finalmente, a última seção dá algumas conclusões sobre o trabalho.

2) Tipos e Tipagem

Um dos mecanismos mais usados para abstração de dados em linguagens de programação é o de tipos. Pela definição convencional (p.e. em [ANSI 83]), tipo é um conjunto de valores mais um conjunto de operações sobre estes valores. O mecanismo de abstração consiste em definir os conjuntos de valores e de operações de modo a serem isomórficos a uma estrutura matemática apropriada. Desta forma, enquanto estes valores forem manipulados somente por estas operações a abstração é mantida, permitindo que o programador assuma para valores de um tipo as mesmas propriedades encontradas na estrutura abstrata correspondente.

Esta idéia não é absolutamente nova, tendo sido usada desde os primeiros computadores para manipulação de valores numéricos. Normalmente, o próprio hardware assume isomorfismos entre padrões de bits e elementos de conjuntos numéricos (como reais ou inteiros), com várias operações úteis da CPU sendo definidas com base nestes

mapeamentos. Por exemplo, uma operação de divisão em ponto flutuante, em geral, não é definida em termos de manipulações de bits, e sim em termos do conjunto dos reais.

Uma vantagem importante do conceito de tipos é o de permitir independência de representação. Se manipulamos valores de um dado tipo apenas através de suas operações, é possível trocar a representação física deste tipo sem que isto afete seu comportamento observável, desde que se modifiquem convenientemente as implementações das operações sobre o tipo.

Um conceito que reforça a idéia de independência de representação é o de tipos abstratos de dados (TADs). Um TAD é um tipo que só pode ser manipulado através de um conjunto de operações bem definido e independente de representação. Em linguagens que se propõem a suportar este conceito, também se espera algum mecanismo que garanta uma unidade sintática na descrição das operações de um TAD, de modo a encapsular toda a descrição em algum tipo de módulo da linguagem.

Normalmente, temos necessidade de trabalhar com vários tipos diferentes num mesmo programa, como reais, inteiros, complexos, etc. Para isto precisamos definir funções de abstração distintas entre padrões de bits e cada um destes conjuntos. Com isto surge a possibilidade de, numa situação de erro, tratarmos um determinado valor de um tipo como sendo de outro. Estes erros são dos mais comuns e difíceis de serem achados dentro de um programa, pois com a quebra da abstração do tipo é preciso trabalhar em nível de representação. Para prevenir este tipo de erro a maioria das linguagens incorporam mecanismos de tipagem, isto é, mecanismos que permitem a detecção automática destas situações.

Neste texto, vamos usar o termo fortemente tipada para fazer referência a linguagens cujo mecanismo de tipagem garanta a impossibilidade da ocorrência de erros de tipos, isto é, tratar um valor

de um tipo como sendo de outro. Uma linguagem fortemente tipada será dita estaticamente tipada se todos os testes necessários à tipagem puderem ser feitos através da análise do texto do programa, sem que sua execução seja necessária. Caso contrário diremos que a linguagem é dinamicamente tipada. Normalmente, a tipagem estática baseia-se no endereço onde está um padrão de bits para reconhecer seu tipo, enquanto na tipagem dinâmica a dedução é feita com base no próprio valor armazenado. No primeiro caso a linguagem associa a cada endereço um único tipo, ou seja, num dado instante cada posição de memória associada a uma instância de variável só pode estar contendo valores de um dado tipo. Além disto, são associados tipos a todas as operações do programa, enquanto regras de compatibilidade definidas pela linguagem garantem que nenhuma atribuição desrespeite a tipagem. No segundo caso, os conjuntos de valores possíveis para cada tipo são disjuntos, permitindo que um teste sobre o valor verifique seu tipo, independentemente de sua posição na memória.

O mecanismo de tipagem estática é, normalmente, preferido em LPs compiladas. Entretanto, a tipagem dinâmica apresenta também uma série de vantagens. Abaixo apresentamos argumentos a favor e contra ambos os mecanismos.

-- Vantagens de tipagem estática:

- melhor desempenho, por não fazer testes durante execução;

- possibilidade de alocação estática e semi-estática de memória, pois normalmente a definição do tipo permite calcular em tempo de compilação o espaço necessário para se armazenar um valor deste tipo;

- deteção de qualquer possibilidade de erro de tipagem em tempo de compilação, sem necessidade de testes exaustivos para garantia de correção neste sentido. Este argumento é considerado como dos mais fortes na escolha de tipagem estática.

-- Vantagens de tipagem dinâmica:

- maior flexibilidade, pois os tipos não precisam ser definidos em tempo de compilação;

- maior facilidade para polimorfismo e reaproveitamento de código em geral (ver seção 4);

- maior facilidade para gerenciamento dinâmico de memória, pela disponibilidade em tempo de execução de informações sobre as estruturas dos diversos valores guardados na memória.

Como forma de manter-se na categoria de estaticamente tipada e se livrar parcialmente de suas limitações, muitas LPs introduzem o conceito de sub-tipo. Um sub-tipo é meramente um tipo com seu domínio restrito a um subconjunto do domínio original, como por exemplo sub-intervalos de tipos escalares. Assim, os mecanismos de verificação para sub-tipos continuam sendo dinâmicos, mas não fazem mais parte da tipagem. Outra solução para este problema é apresentada por [Booth 86], que propõe uma linguagem "multi-fase", em oposição ao sistema convencional de compilação-execução. Nesta linguagem, cada fase do processamento de um programa serve de compilação para a fase seguinte, permitindo que tipos sejam calculados dinamicamente como qualquer outro valor, desde que estejam disponíveis uma fase antes de seu uso para ser feita a tipagem "estática".

3) Objetos

O conceito de objeto, no sentido empregado nas expressões linguagem orientada a objetos (LPOO) e programação orientada a objetos, ainda é bastante controvertido. Na área de linguagens, por exemplo, apesar do consenso sobre o que é ou não um objeto em Smalltalk, não encontramos concordância quando a linguagem em questão é ADA ou Modula-2.

Neste texto vamos usar uma definição de objetos bastante simples: um objeto é uma estrutura de dados mais um conjunto de operações sobre

esta estrutura, com a restrição de que os dados internos de um objeto só podem ser manipulados por suas próprias operações.* Nesta definição não incluímos o conceito de herança, apesar de alguns autores ([Cox 86], [Cardell 84]) o colocarem como fundamental na caracterização de objetos. Apesar de sua importância para o sucesso das LPOOs, não o consideramos um conceito essencial à caracterização de objetos, e sim um mecanismo específico de reaproveitamento de código.

Esta definição obviamente abrange os conceitos de objetos em linguagens como Smalltalk; Flavors ([Weinreb 80]) e Objective-C ([Cox 86]). Em linguagens como ADA, um módulo (Package) que só exporte procedimentos também pode ser considerado um objeto. Entretanto, módulos ou packages não são valores de primeira classe nestas linguagens, pois não podem ser criados dinamicamente, nem atribuídos a variáveis, nem passados como parâmetros para procedimentos, além de várias outras restrições. Desta forma, apesar destas linguagens terem objetos, não achamos que ofereçam facilidades suficientes neste sentido para serem chamadas de orientadas a objetos.

Neste ponto é importante frisar a diferença entre objeto e tipo abstrato de dados. No mecanismo de TADs, a abstração envolve o tipo como um todo, enquanto no mecanismo de objetos a abstração protege cada valor individualmente. Vamos exemplificar imaginando uma rotina para somar dois números complexos. Se os números são instâncias de TADs, esta rotina, sendo interior ao tipo, pode manipular as estruturas internas dos dois números indiscriminadamente. Se, por outro lado, são objetos, a rotina deve estar dentro de um dos números, e portanto só conhece a representação deste, sendo obrigada a manipular o outro número via sua interface externa.

---

* - Com esta restrição tiraríamos Simula-67 da categoria de orientada a objetos, por seu deslize ao permitir acesso externo às variáveis internas de um objeto. Em respeito ao seu pioneirismo nesta área vamos ignorar este fato, supondo que o programador disciplinado não lança mão destas facilidades.

Quando utilizamos objetos como mecanismo de abstração, o conceito de tipo perde bastante da sua importância. Uma vez que objetos já são intrinsecamente protegidos contra operações indevidas, não precisamos nem de tipos nem de tipagem para este controle. Por isto a maioria das LPOOs substitui o conceito de tipo pelo de classe. Classes fornecem uma maneira apropriada para a descrição de objetos que tenham a mesma estrutura interna e mesmas operações, sem necessidade de repetições. Com esta facilidade, o programador define as estruturas e operações nas classes, e cada objeto que deseje estas características é criado como instância desta classe. Algumas linguagens, como Smalltalk, exigem que todo objeto tenha uma classe, enquanto outras permitem objetos sem vinculação a qualquer classe. Entretanto, várias linguagens orientadas a objetos (como Simula-67) mantêm conceitos de tipos associados ao de classe, para permitir a deteção de alguns erros em tempo de compilação.

Em algumas linguagens chamadas de híbridas, como Objective-C, os mecanismos de tipos e de objetos coexistem. Nestes casos, todos os objetos são classificados em um só tipo de dado, cuja única operação disponível é a de envio de mensagens. Com isto, o programador pode optar entre a segurança de tipos ou a flexibilidade de objetos sem mudar de linguagem.

4) Reaproveitamento de Código

O termo Reaproveitamento de Código refere-se a qualquer mecanismo que permite que o programador reutilize, dentro de um sistema, trechos de software já escritos para outras aplicações. Em especial, a reutilização pode envolver tanto código fonte quanto código objeto. Nesta seção pretendemos discutir as relações entre algumas técnicas de reaproveitamento e os mecanismos de abstração expostos nas seções anteriores.

A relação entre reaproveitamento e abstração é bastante estreita.

Por um lado, qualquer mecanismo de abstração tende a facilitar o reaproveitamento, na medida em que aumenta a modularidade e coesão dos programas produzidos. Assim, classes, módulos ou clusters são mais facilmente reaproveitados do que rotinas soltas, pois compõem uma abstração completa e fechada sobre um tipo. Por outro lado, os mecanismos normais de tipagem estática tendem a dificultar bastante este reaproveitamento, pois amarram tipos fixos no código que raramente vão corresponder exatamente aos usados em outras aplicações.

Um dos mecanismos mais usados para contornar esta dificuldade é o de polimorfismo ([Milner 78], [Cardelli 85]). Polimorfismo é a característica de um objeto, numa linguagem, ter mais de um tipo. Em particular, funções polimórficas (e módulos polimórficos) são funções que aceitam parâmetros de tipos diversos. Exemplos clássicos de polimorfismo são funções de ordenação que ordenam qualquer tipo de array que possua uma relação de ordem entre seus elementos, e módulos de manipulação de listas que manipulam listas de qualquer coisa.

A combinação de polimorfismo com tipagem estática não é tarefa simples ([Aho 86]). Algumas linguagens permitem polimorfismo abandonando a tipagem, como C e Modula-2 (p.e. usando parâmetros do tipo ARRAY OF WORD). Outras, como CLU e Ada, usam polimorfismo parametrizado, com a introdução de unidades genéricas. Estas são unidades de software (packages, clusters ou procedimentos) parametrizáveis, onde admitem-se parâmetros para especificação de tipos. Existindo uma unidade genérica, o programador pode instanciá-la fornecendo argumentos adequados, criando assim uma unidade que pode ser usada de forma convencional. Esta facilidade serve bastante bem à implementação de TADs parametrizados ([Veloso 86]).

Um detalhe do mecanismo de unidades genéricas é que os parâmetros devem ser fixados em tempo de compilação, pois a geração de código para uma unidade genérica pode depender destes valores. Por isso, nas

implementações normais desta facilidade temos apenas reaproveitamento de código fonte, com o compilador gerando um módulo objeto diferente para cada instância usada.

Outra maneira bastante interessante de se tratar polimorfismo é a usada em ML ([Milner 78], [Harper 85]). Nesta linguagem, apesar de existir tipagem estática, não são necessárias declarações de tipos, que são, sempre que possível, deduzidos automaticamente pelo interpretador. Toda função declarada é a princípio polimórfica, a menos de restrições deduzidas pelo uso que o corpo da função faz de cada parâmetro. Baseada nestas informações, a linguagem usa um algoritmo de unificação para deduzir o Tipo Mais Geral de cada função, e a cada uso verifica se os parâmetros reais satisfazem as restrições impostas.

Um mecanismo particularmente poderoso de reaproveitamento de código, disponível em praticamente todas as LPOOs, é o mecanismo de Herança. Através deste mecanismo, um objeto (ou classe) pode incorporar na sua definição estruturas e operações definidas para outra classe, herdando suas propriedades. Em cima desta herança a classe pode definir novas propriedades, aumentando as estruturas de dados, criando novos métodos ou modificando algumas operações herdadas.

Nas primeiras LPOOs, este mecanismo era restrito ao que se chama de Herança Simples, onde cada classe só pode herdar propriedades de uma única outra classe. Desta forma, as relações de hereditariedade formam uma árvore, onde a classe raiz define propriedades partilhadas por todos os objetos do sistema. A outra forma de herança é a chamada Herança Múltipla, onde uma classe pode herdar propriedades de quantas outras precise, com as relações de hereditariedade formando um grafo acíclico. Esta facilidade é certamente muito mais poderosa do que a herança simples, mas apresenta alguns problemas. O primeiro diz respeito a implementação, sendo difícil implementar eficientemente rotinas para procurar cada operação que um objeto deva executar em todos os seus

ancestrais. Também existe uma certa dificuldade na parte de organização de memória interna dos objetos, sendo necessário linearizar as várias estruturas de dados herdadas de uma árvore de ancestrais. Outro problema é o de possíveis conflitos entre as várias propriedades herdadas, como, por exemplo, duas operações diferentes para uma mesma tarefa. Cada linguagem resolve estes problemas de alguma forma, estabelecendo seu próprio compromisso entre eficiência e flexibilidade. Uma boa discussão deste ponto pode ser encontrada em [Stefik 86].

5) Facilidades de Abstração em Linguagens Usuais

Desde o aparecimento de FORTRAN, em 1957, as linguagens de programação procuram fornecer mecanismos de abstração de dados que facilitem a tarefa de programar. A grande maioria das linguagens usa para isto o conceito de tipo, mas apresenta grande variedade no que diz respeito a mecanismos para construção de novos tipos, tipagem e facilidades para TADs.

As primeiras linguagens de programação (como FORTRAN) apresentavam um repertório fixo de tipos, normalmente baseado nos tipos fornecidos pelo hardware. Pouco depois, várias linguagens já apresentavam mecanismos que permitiam ao programador definir novos tipos, conforme suas necessidades. Um modelo importante desta fase foi ALGOL-68, que usa um conjunto de tipos primitivos mais um conjunto de construtores (arrays, records, etc) que permitem combinar tipos existentes para a criação de novos. Este modelo foi seguido por PASCAL, de onde se irradiou para a maioria das linguagens procedurais modernas.

Apesar de seu sucesso, este modelo tem vários defeitos, como sensibilidade a mudanças de representação, sintaxe nem sempre clara, dificuldade de especificação de procedimentos para inicialização e finalização de variáveis e tratamento de exceções. A maioria destes

problemas foram resolvidos em linguagens mais recentes, com mecanismos de suporte a objetos ou Tipos Abstratos de Dados. A seguir, vamos abordar cada um destes pontos e ver como foram tratados por algumas linguagens mais modernas. Damos especial atenção para Smalltalk ([Goldberg 83]), por ser um paradigma de LPOOs, CLU ([Liskov 81], [Liskov 77]), por sua ênfase em TADs e Ada ([ANSI 83], [Gehani 83]), por sua importância atual, que a faz um parâmetro de comparação nesta área. É importante observar que Ada não oferece suporte explícito a TADs, e sim um mecanismo de abstração geral (packages) que pode, se devidamente usado, servir para implementação de TADs ([Sherman 82]).

A sensibilidade a mudanças de representação é um ponto central na qualidade de um mecanismo de abstração de dados, em particular no suporte a TADs. Em PASCAL não existe nenhuma facilidade neste sentido. Assim, como exemplo, uma pilha representada como um array é antes de tudo um array, e como tal pode ser manipulado por todas as operações que a linguagem oferece para arrays. Desta forma, se a representação da pilha for modificada, podem ser necessárias alterações no restante do fonte do programa, e eventualmente até em sua lógica geral.

Numa linguagem que suporte independência de representação, deve ser sempre possível modificar a representação de um TAD, sem necessidade de alterações nos trechos que apenas utilizam o tipo (chamaremos tais trechos de utilização). Normalmente, esta facilidade é implementada pelo compilador através de um controle adequado de visibilidade, que só permite à utilização acessar uma variável de um TAD através de suas funções. Ainda assim, existe uma gradação de independência, de acordo com a necessidade, após uma modificação, de (a) ser preciso recompilar a utilização, (b) apenas religar a utilização com a nova implementação, (c) não ser necessária nem a religação. No primeiro caso temos ADA, no segundo CLU e no terceiro Smalltalk, que faz ligação dinâmica. Em Smalltalk, em especial, não é necessário nem se interromper a

utilização, uma vez que a modificação pode ser feita durante a execução do programa.

O problema sintático diz respeito às diferenças sintáticas na manipulação de tipos, dependendo de sua representação. O ideal é que todos os tipos tenham os mesmos direitos, isto é, que possam ser manipulados com operações infixadas, participando em atribuições e comparações (permitindo semântica própria, como os tipos primitivos tem), sofrendo coerção automática quando necessário, além de outras regalias.

Em CLU este problema foi totalmente resolvido com algumas restrições às facilidades sintáticas para tipos primitivos. Assim, (a) não temos coerção para nenhum tipo, (b) todas as operações são qualificadas para evitar ambiguidades, (c) operadores com sintaxe especial, como indexação em arrays e acesso a campos em registros, são tratados como abreviações para chamadas específicas que qualquer tipo pode implementar. Por seu conceito de variavel como referência a um valor, a operação de atribuição é exatamente igual para todos os tipos, não fazendo sentido redefini-la.

Em Smalltalk o problema também é bem resolvido, mas como consequência da homogeneidade semântica da linguagem. Em especial, como não há tipagem estática, é impossível um tipo ter privilégios sintáticos sobre outros.

ADA resolveu parcialmente o problema, permitindo sobrecarga de operadores e definição de funções para operadores infixados e pré-fixados. Não apresenta homogeneidade nas operações específicas sobre arrays e records (indexação e seleção), nem permite redefinição do operador de atribuição (que admite interpretações diferentes).

Uma linguagem que apresenta algumas características interessantes neste ponto é EL1 [Wegbreit 74]. Em EL1 podemos redefinir as rotinas de atribuição, bem como a rotina de acesso a uma variável (rotina "ValOf"),

para qualquer tipo. Além disto, podemos definir rotinas de coerção entre tipos, com o compilador se encarregando de procurar uma sequência de coerções para compatibilizar dois tipos numa operação.

A questão de inicialização e finalização de valores é bastante simples em linguagens onde os conceitos de visibilidade e de vida de valores são desvinculados, como CLU e Smalltalk. Nestas linguagens sempre é necessário um procedimento explícito para a criação de um objeto, que pode então executar qualquer inicialização desejada. Teoricamente, o objeto continua a existir eternamente, não fazendo sentido procedimentos de finalização. Por outro lado, na prática, o espaço ocupado por um valor pode ser reaproveitado a partir do momento em que este valor não for acessível (garbage colection). Em Ada não existe nenhum suporte para inicialização nem para finalização de variáveis de tipos exportados por módulos, de modo que, se forem necessários, estes procedimentos devem ser explicitamente programados e chamados pelo programador. Um suporte parcial é dado através do mecanismo de inicialização de módulos. Uma solução que poderia ser facilmente encaixada no modelo de execução de Ada é o usado em Euclid ([Lampson 77], [Chang 78]), que especifica nomes padrão para rotinas a serem chamadas automaticamente nestas ocasiões.

A questão de tratamento de exceção, quando vista sob o prisma de abstração de dados, pode ser dividida considerando-se dois tipos de exceção, que na falta de uma nomenclatura corrente passamos a chamar de exceção de especificação e exceção de implementação. O primeiro tipo diz respeito a situações de erro já descritas na especificação das operações do tipo, e portanto independentes de qualquer implementação específica. Exemplos deste tipo de exceção são os famosos casos de desempilhar uma pilha vazia ou dividir por zero. O segundo tipo de exceção refere-se a situações de erro causadas pela incapacidade de uma determinada implementação de se manter de acordo com sua especificação sob

determinadas circunstâncias. Nesta situação temos os também famosos casos de empilhamento em uma pilha cheia e "overflow" em operações aritméticas. Este tipo de exceção tem tanto sua existência quanto sua situação de ocorrência dependentes da implementação, sendo portanto muito difícil manipulá-los ignorando a representação usada.

Mantendo sua ênfase em TADs, CLU oferece um mecanismo de exceção dirigido para os conceitos de exceção expostos acima. Nesta linguagem, a responsabilidade pelo tratamento de uma exceção é da rotina que solicitou a operação, seguindo a idéia de que quem sinaliza uma exceção não tem como tratá-la (se tivesse não seria exceção). Cada rotina, ao ser chamada, pode retornar normalmente, se tudo correu bem, ou retornar sinalizando uma exceção específica, indicando algum problema. No segundo caso o controle não segue para a instrução seguinte à chamada, suposta de tratar os casos normais, mas para um bloco especial dentro da rotina indicado para tratar aquela exceção. Se a rotina solicitante do serviço também não tiver meios de normalizar a situação, pode sinalizar outra exceção ou, simplesmente passar adiante a que recebeu, com o comando resignal.

Ada oferece um mecanismo bem mais flexível para tratamento de exceções, como sempre de caráter geral, sem visar nenhuma metodologia de aplicação em particular. Ao contrário de CLU, o tratamento de uma exceção não precisa ser feito pelo solicitante da rotina sinalizadora, mas sim pela primeira rotina da cadeia dinâmica que ofereça um tratador para a exceção sinalizada. Em particular, uma exceção pode ser tratada pela própria rotina sinalizadora, sempre que esta ofereça um bloco de tratamento adequado. É interesante observar que o uso indiscriminado destas facilidades em Ada dificulta o estabelecimento de fronteiras claras em torno das abstrações usadas.

Smalltalk oferece um mecanismo de tratamento de exceções bastante diferente, mais adequado à opacidade de objetos. Nesta linguagem as

exceções devem ser tratadas pelo objeto sinalizador, e não pelo solicitante da operação. Quando ocorre uma exceção num método de algum objeto, a máquina virtual se encarrega de enviar uma nova mensagem a este objeto, avisando-o da ocorrência. Normalmente, estas mensagens são tratadas por métodos herdados da classe "Object", que suspendem a execução do programa e notificam o programador/usuário. Entretanto, uma classe pode redefinir os métodos para determinadas mensagens de exceção, permitindo ao programador especificar suas próprias rotinas de tratamento.

6) Conclusões

Vimos aqui como o mecanismo de tipos, refinado depois em tipos abstratos de dados, tem sido usado desde os primórdios da computação como mecanismo de abstração de dados. Ao longo do tempo, as linguagens foram incorporando novos mecanismos para facilitar o uso destes conceitos. Em particular, a introdução de linguagens fortemente tipadas teve grande impacto na correção de programas.

Algumas linguagens, como CLU, são centradas neste tipo de paradigma, sustentando uma metodologia de programação baseada em TADs. Outras, como Ada ou Modula-2, se propõem a suportar qualquer metodologia, mantendo entretanto mecanismos de suporte a abstração de dados e TADs. Nos últimos tempos, em especial após Smalltalk-80, vem ganhando importância o conceito de programação orientada a objetos. Este paradigma, como opção em situações onde os conceitos de tipos e tipagem estática não oferecem facilidades adequadas (p.e. em desenvolvimento de protótipos), vem reforçar o uso de abstração de dados como forma de aumentar a modularidade e facilitar o reaproveitamento de código, pontos fundamentais para uma real melhoria no processo de desenvolvimento de software.

Referências Bibliográficas

[Aho 86] - Aho, A.; Sethi, R.; Ullman, J. Compilers - Principles, Techniques and Tools. New York, Addison Wesley, 1986.

[ANSI 83] - Ada Programming Language. ANSI/MIL-STD 1815A-1983, 22 january 1983.

[Birtwistle 75] - Birtwistle, G.; Dahl, O.; Myhrhaug, B.; Mygaard, K. Simula Begin. New York, Petrocelli/Charter, 1975.

[Booth 86] - Booth, D. Multiple Strongly Typed Evaluation Phases: A Programming Language Notion. Los Angeles, UCLA, Computer Science Department, 1986. Technical Report CSD-860042.

[Burstall 84] - Burstall, R. & Lampson, B. A Kernel Language for Abstract Data Types. In: Kahn, G.; MacQueen, D. B.; Plotkin, G. Semantics of Data Types. New York, Springer-Verlag, 1984. (LNCS 173)

[Cardelli 84] - Cardelli, L. A Semantics of Multiple Inheritance. In: Kahn, G.; MacQueen, D. B.; Plotkin, G. Semantics of Data Types. New York, Springer-Verlag, 1984. (LNCS 173)

[Cardelli 85] - Cardelli, L. & Wegner, P. On Understanding Types, Data Abstraction and Polymorphism. ACM Computing Surveys, 17(4), 1985.

[Chang 78] - Chang, E.; Kaden, N.; Elliott, W. D. Abstract Data Types in Euclid. Sigplan Notices, 13(3), 1978.

[Cox 86] - Cox, B. J. Object Oriented Programming - An Evolutionary Approach. New York, Addison-Wesley, 1986.

[Fairley 85] - Fairley, R. Software Engineering Concepts. New York, MacGraw Hill, 1985.

[Gehani 83] - Gehani, N. Ada - An Advanced Introduction. New Jersey, Prentice-Hall, 1983.

[Ghezzi 85] - Ghezzi, C. & Jazayeri, M. Conceitos de Linguagens de Programação. Rio de Janeiro, Campus, 1985.

[Goldberg 83] - Goldberg, A. & Robson, D. Smalltalk.80 - The Language and its Implementation. New York, Addison-Wesley, 1983.

[Harper 85] - Harper, R. Introduction to Standard ML. Edinburgh, Univerity of Edinburgh, Computer Science Departament, 1985. (preliminary draft).

[Horowitz 84] - Horowitz, E. Fundamentals of Programming Languages. Berlin, Springer-Verlag, 1984 (segunda edição).

[Lampson 77] - Lampson, B. W.; Horning, J. J.; London, R. L.; Mitchell, J. G.; Popek G. L. Report on the Programming Language Euclid. Sigplan Notices, 12(2), 1977.

[Liskov 74] - Liskov, B. & Zilles, S. Programming with Abstract Data Types. Sigplan Notices, 9(4), 1974.

[Liskov 75] - Liskov, B. & Zilles, S. Specification Techniques for

Data Abstractions. IEEE Transactions on Software Engineering, SE-1(1), 1975.

[Liskov 77] - Liskov, B. & Atkinson, R. Abstraction Mechanisms in CLU. Communications of the ACM, 20(8), 1977.

[Liskov 81] - Liskov, B. et alii. CLU Reference Manual. New York, Springer-Verlag, 1981. (LNCS 114)

[Milner 78] - Milner, R. A Theory of Type Polymorphism in Programming. Journal of Computer and System Sciences, 17(3), 1978.

[Sherman 82] - Sherman, M. et alii. A Methodology for Programming Abstract Data Types in ADA. In: AdaTEC Conference on ADA, 1982.

[Stefik 86] - Stefik, M. & Bobrow, D. G. Object-Oriented Programming: Themes and Variations. The AI Magazine, winter, 1986.

[Veloso 86] - Veloso, P. A. S. Tipos (Abstratos) de Dados: Programação, Especificação, Implementação. Belo Horizonte, UFMG, 1986. (V Escola de Computação)

[Webreit 74] - Webreit, B. The Treatment of Data Types in EL1. Communications of the ACM, 17(5), 1974.

[Weinreb 80] - Weinreb, D.; Moon, D. Flavors: Message Passing in the Lisp Machine. Cambridge, Ma., MIT, Artificial Intelligence Laboratory, 1980. A.I. Memo 602.

[Wirth 85] - Wirth, N. Programming in MODULA-2. Berlim, Springer-Verlag, 1985 (terceira edição).