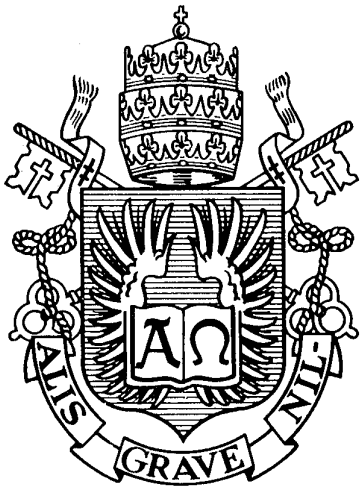

PUC

Série : Monografias em Ciência da Computação
No.2/89

EM BUSCA DE UMA LINGUAGEM ORIENTADA A OBJETOS COMPATÍVEL COM MÉTODOS RIGOROSOS DE DESENVOLVIMENTO DE PROGRAMAS

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: 2/89

Editor: Paulo A. S. Veloso Março, 1989

EM BUSCA DE UMA LINGUAGEM ORIENTADA A OBJETOS COMPATIVEL COM METODOS RIGOROSOS DE DESENVOLVIMENTO DE PROGRAMAS†

Roberto Ierusalimschy

†Trabalho parcialmente financiado pela SID-Informática e pela IBM.

Para obter cópias :
Rosane T. L. Castilho
Assessoria de Biblioteca, Documentação e Informação
Rua Marquês de São Vicente, 225 - Gávea
22.453 - Rio de Janeiro, RJ.
Brasil

EM BUSCA DE UMA LINGUAGEM ORIENTADA A OBJETOS COMPATÍVEL COM MÉTODOS RIGOROSOS DE DESENVOLVIMENTO DE PROGRAMAS

Roberto Ierusalimschy

RESUMO

Este texto pretende servir de base para um projeto de pesquisa visando compatibilizar linguagens orientadas a objetos com métodos rigorosos de desenvolvimento de programas. Para isto investiga-se quais mecanismos são relevantes para orientação a objetos e quais as dificuldades de formalização destes mecanismos. Especial atenção é dedicada à programação em ponto grande.

Palavras-chave: Programação Orientada a Objetos; Métodos Formais de Desenvolvimento de Programas; Linguagens de Programação.

ABSTRACT

The intention of this text is to lay a basis for a research project on the use of formal methods for software development with Object-Oriented Languages. We investigate which mechanisms are relevant for object orientation and which are the dificulties involved in the formalization of these mechanisms. Facilities for Programming in the Large receive special attention.

Keywords: Object-Oriented Programming; Formal Methods for Software Development; Programming Languages.

Roberto Ierusalimschy

Em Busca de uma Linguagem Orientada a Objetos Compativel com Metodos Rigorosos de Desenvolvimento de Programas

> "A most important, but also a most elusive, aspect of any tool is its influence on the habits of those who train themselves in its use. If the tool is a programming language, this influence is - whether we like it or not - an influence on our thinking habits."
>
> [Dijkstra 76] pp. xiii

1) Introdução

A linguagem de programação Simula foi precursora de pelo menos dois importantes conceitos na área de programação. O primeiro foi o de programação com Tipos Abstratos de Dados (TAD), que teve (e ainda tem) forte influência sobre o projeto de linguagens de programação e métodos sistemáticos de desenvolvimento de programas, e deu especial motivação para a área de especificações formais. O segundo conceito ficou relativamente latente até muito tempo depois, quando o lançamento de Smalltalk-80 colocou na moda o termo Linguagens Orientadas a Objetos (LOO). Entretanto, apesar do berço comum, as correntes saídas destes dois conceitos divergiram bastante nos últimos tempos.

Certamente, a principal causa desta divergência foi Smalltalk-80. Esta linguagem foi o resultado de muitos anos de trabalho do Learning Research Group de Xerox PARC, iniciado com o *Dynabook* de Alan Kay. O objetivo desta pesquisa era construir uma máquina que desse suporte a trabalhos criativos, usando para isto o melhor hardware disponível. Bem cedo se percebeu que, para a construção deste sistema, seria necessário o desenvolvimento de

uma linguagem adequada, com características bastante específicas. Não era intenção do grupo produzir algo que chamamos normalmente de linguagem de propósito geral. Desta forma, esta linguagem divergiu bastante das linguagens convencionais desenvolvidas na mesma época. Com o sucesso alcançado por Smalltalk-80 houve um grande interesse em linguagens orientadas a objetos e uma certa vinculação deste conceito com as características particulares do sistema Smalltalk.

Vista como uma linguagem de programação de propósito geral, Smalltalk-80 trouxe várias características marcantes, algumas positivas e outras negativas. No lado positivo, certamente o aspecto mais importante da linguagem é a facilidade para reaproveitamento de código, através da herança e polimorfismo, sem comparação com outras linguagens da época. Algumas facilidades para programação com tipos abstratos de dados também foram muito bem vindas, mesmo não sendo grande novidade.

Dos aspectos negativos da linguagem, a maioria foi consequência das peculiariedades do ambiente ao qual se destinava. Seguindo a filosofia de máquinas pessoais para usuários inventivos, a linguagem incentiva programação experimental, sistemas abertos com detalhes de todas as classes expostas ao programador, alto grau de interdependência entre componentes e baixo nível de redundância. Todos estes pontos são frontalmente contrários as ditas boas práticas de programação que as linguagens de programação convencionais[1] buscam suportar, como modularidade, abstração e o máximo possível de verificações feitas na compilação.

Praticamente todas as linguagens orientadas a objetos posteriores tiveram Smalltalk-80 como paradigma, herdando suas qualidades e seus defeitos. Algumas, denominadas linguagens híbridas (p.e. C++, Objective-C e Object-Pascal), procuraram

[1] o termo *linguagem convencional* está sendo usado em oposição a linguagem orientada a objetos, e se refere as linguagens procedurais descendentes de Algol, como Pascal, Euclid, Modula-2 e Ada.

incorporar os mecanismos de objetos em linguagens convencionais[2]. Outras (p.e. Modular Smalltalk) procuraram o inverso, incorporando em linguagens orientadas a objetos mecanismos de linguagens convencionais. De qualquer forma, muito poucas se preocuparam com uma integração efetiva de LOO com os avanços em outras áreas, notadamente no que se refere a programação em ponto grande, verificação de programas e metodologias de desenvolvimento de programas.

Este texto pretende servir de base para um projeto de pesquisa em projetos de linguagens orientadas a objetos sobre os problemas citados no parágrafo anterior. Para isto, investigamos aqui quais mecanismos e conceitos são relevantes para orientação a objetos, suas vantagens e desvantagens e, em particular, dificuldades para formalização. Esta última "métrica" é justificada não só por sua importância em sí, mas também porque achamos que conceitos de dificil formalização normalmente se traduzem em mecanismos que dificilmente são corretamente usados e compreendidos em todas as suas consequências pelo programador, e portanto não se mostram apropriados para métodos sistemáticos de desenvolvimento.

A próxima seção discute as principais características de LOO, apontando seus pontos distintivos. A seção 3 analisa aspectos mais convencionais, como tipagem e estruturas de controle, enquanto a seção 4 é dedicada a mecanismos de programação em ponto grande. Ao longo de todas estas discussões são apresentadas algumas sugestões iniciais de como atacar cada problema. A última seção resume os principais problemas encontrados e sumariza estas sugestões. A bibliografia apresenta, além das referências, outros trabalhos considerados importantes para esta pesquisa.

2) Aspectos de Linguagens Orientadas a Objetos

O conceito do que seja uma linguagem orientada a objetos é bastante vago e foco de certa discussão atualmente. Por isto, ao

[2] A exemplo de Simula, uma extensão de Algol-60.

invés de discutirmos diretamente sobre este conceito preferimos enumerar uma lista de mecanismos mais simples e bem definidos que estão presentes em muitas linguagens unanimemente reconhecidas como orientadas a objetos[3], e centrar nossa discussão nas causas e consequências destes mecanismos.

A lista de mecanismos considerados relevantes é a seguinte: abstração de dados com noção de estado, herança, visibilidade instancial, variáveis referenciais e "late-binding". Achamos que uma presença e uma ausência na lista merecem nota. A presença da noção de estado tira da categoria várias linguagens funcionais ditas orientadas a objetos, como Loops e Flavors. Aqueles que considerarem isto inadmissível devem ler o trabalho trocando a expressão "linguagens orientadas a objetos" por "linguagens procedurais orientadas a objetos". A ausência digna de explicação é a de concorrência. Apesar deste conceito estar subjacente a objetos desde seus primórdios em Simula-67, através do mecanismo correlato de corotinas, poucas linguagens atuais incorporam concorrência entre suas facilidades, impossibilitando uma discussão mais geral.

É interessante observar que alguns destes mecanismos podem ser usados para substituir conceitos de dificil formalização em linguagens convencionais. Estes mecanismos são variáveis referenciais (substituindo ponteiros), herança (*variant records*) e late-binding (passagem de procedimentos como parâmetros). Infelizmente, o uso indiscriminado destes mecanismos também apresenta grandes dificuldades para formalização, como veremos ao longo desta seção.

Abstração de Dados com Estado

Este conceito normalmente é satisfeito pela definição padrão

[3]Explicitamente temos Smalltalk, Objective-C, C++, Eiffel, Trellis/Owl, Modula-3 e Simula-67. Em particular, Simula-67 não apresenta mecanismos típicos de abstração de dados, por anteceder este conceito. Vários dos mecanismos discutidos estão presentes também em CLU, apesar desta linguagem não ser considerada OO.

de objetos, no estilo "um objeto é um conjunto de variáveis (estado) e um conjunto de rotinas que manipulam estas variáveis". Normalmente apenas estas rotinas podem ter acesso às variáveis internas de um objeto, o que garantiria a abstração. O ponto que consideramos falho neste aspecto é consequência do uso de variáveis referenciais para representar o estado de um objeto: protege-se a referência mas não se protege o objeto referenciado. Este ponto é melhor discutido na próxima seção, no ítem sobre modularidade. Outros aspectos importantes de abstração também são tratados nas próximas seções.

No aspecto de formalismos, este ponto é razoavelmente bem servido atualmente. Existe uma enorme gama de teorias matemáticas para dar suporte a tipos abstratos de dados, destacando-se especificações algébricas, especificações em lógica e modelos, e várias destas teorias são usadas para desenvolvimento formal. Como consideramos a noção de estado como aspecto fundamental de objetos, é necessário um formalismo que trate este conceito da forma mais natural possível. O uso de modelos, como apresentado por Hoare[4] e agora usado em VDM, nos parece bastante adequado, principalmente quando consideramos o conceito de herança.

Herança

Herança denota a habilidade da descrição de uma classe incorporar a descrição de outras classes, fazendo com que a primeira "herde" as propriedades das outras. Este conceito é considerado por muitos autores como a característica distintiva de LOO, sendo inclusive comum a expressão *tipos abstratos + herança = objetos*. Além de sua importância individual, herança também é a motivação de vários dos outros mecanismos, notadamente visibilidade instancial e late-binding. Para iniciarmos uma discussão sobre este ponto, achamos importante frizar a diferença entre herança de implementação e herança de especificação.

O conceito de herança de implementação corresponde ao mecanismo existente em Smalltalk. A motivação básica é reusar

[4]Em [Hoare 72], sintomaticamente numa axiomatização para Simula.

código interno de uma determinada classe para construir uma nova classe, de forma que a subclasse disponha internamente de todas as operações já existentes para a classe antiga. Para isto é necessário que a nova classe tenha uma representação interna similar à sua superclasse, de modo que os métodos da superclasse possam atuar na nova estrutura corretamente. O conceito de similaridade depende da linguagem; em Smalltalk exige-se que uma subclasse tenha pelo menos todas as váriaveis de sua superclasse. Este mecanismo, do ponto de vista semântico, pode ser substituído por um pré-processador que inclua, de maneira apropriada, o código fonte da definição da superclasse no fonte da subclasse.

O conceito de herança de especificação corresponde ao mecanismo fornecido por Emerald e Trellis/Owl, e tem como motivação básica a reutilização de código externo a um determinado tipo[5], numa forma controlada de polimorfismo. Com este mecanismo, um novo tipo pode ser submetido a todas as operações disponíveis para seu supertipo. Para esta forma de herança não é preciso similaridade de estruturas internas das classes, mas sim de suas interfaces. Novamente o conceito de similaridade é bastante variado, e neste caso está fortemente relacionado com o sistema de tipos da linguagem. O conceito de herança de especificação, por esta sua conexão com sistemas de tipos, é bem mais controvertido e complexo que o de herança de implementação.

A maior parte das LOO mistura esses dois conceitos, obrigando o paralelismo entre as duas hierarquias. Com isto assume-se implicitamente que um tipo que se comporte como subtipo de outro necessariamente segue a mesma implementação, da mesma forma que classes com implementações similares devam apresentar comportamentos compatíveis.

Apesar do quanto já se publicou a respeito do que seja herança, não existe um consenso sobre uma definição formal deste

[5]Estamos usando os termos *tipo* e *classe* com sentidos diversos. Para nós, um *tipo* denota um tipo abstrato no sentido convencional, enquanto classe denota uma implementação específica de um dado tipo.

conceito, sendo muito vaga a noção de similaridade usada nas definições acima. Um exemplo bastante simples ilustra a confusão a respeito.

Consideremos dois tipos de dados, *Integer* e *Natural*, com seus significados usuais. Por um lado, parece bastante intuitivo classificarmos *Natural* como subclasse de *Integer*, pois todo natural é também um inteiro. A noção de subsort usada em OBJ2 formaliza esta situação. Por outro lado, usando a formalização apresentada em [Cardelli & Wegner 85], podemos considerar um natural como um registro com um único campo, algo como [*value* : *Natural*], enquanto um inteiro pode ser representado por [*value* : *Natural* ; *signal* : *Boolean*]. Neste caso teriamos *Integer* como subclasse de *Natural*. Como nenhum modelo admite ciclos nas relações de herança, um sistema formal adequado deveria suportar apenas uma das duas opções (ou talvez nenhuma delas), mas não parece fácil escolher qual a mais razoável.

Variáveis Referenciais

Uma variável é dita referencial quando contém referências para objetos, ao invés dos próprios objetos. Na falta de um termo melhor, diremos que uma linguagem usa variáveis referenciais quando todas as variáveis desta linguagem são desta categoria. Com esta definição englobamos as LOO não híbridas (e CLU), e excluímos as LP ditas tradicionais, como Modula-2 e Ada que, aperar de terem variáveis referenciais sob a forma de ponteiros, não as preconizam como única nem mais usada categoria de variáveis.

O mecanismo de variáveis referenciais tem motivações semânticas e implementacionais. Do ponto de vista semântico, a ausência de variáveis globais nestas linguagens cria a necessidade de outra forma de compartilhamento, que é conseguido com cópias das referências. Esta estrutura é bastante adequada para bancos de dados, pela facilidade que oferece para compartilhamento de informações e para navegação. Esta semântica também permite desvincular o escopo dos objetos do escopo das variáveis, facilitando o tratamento de objetos persistentes, outro aspecto importante para bancos de dados. Do ponto de vista implementacional, o uso de referências possibilita uma

implementação eficiente de polimorfismo e herança, permitindo que uma variável de um dado tipo possa "conter" objetos de qualquer subtipo deste sem problemas de espaço de memória. Da mesma forma facilita a implementação de mecanismos de abstração de dados, possibilitando a compilação de um módulo sem conhecimento, por parte do compilador, da implementação dos tipos usados[6]. Finalmente, o uso sistemático de variáveis referenciais simplifica o gerenciamento automático de memória e "garbage collection", facilidades cada vez mais importantes em linguagens de alto nível.

Um problema bastante grave criado pela semântica de variáveis referenciais é a possibilidade de todo objeto ser global, isto é, poder ser acessado de qualquer ponto de um sistema. Esta característica pode comprometer totalmente a modularidade de um sistema, conforme discutimos na próxima seção. Do ponto de vista formal, esta semântica implica no uso de formalizações semelhantes às usadas para ponteiros, com toda complexidade inerente a este tratamento. Parece-nos que algumas restrições na visibilidade de objetos, a exemplo do que foi feito em Euclid, pode facilitar bastante as provas de programas nestas linguagens.

Visibilidade Instancial

Neste trabalho este termo significa que as fronteiras de abstração de cada tipo cercam cada instância deste isoladamente, ao invés de proteger o tipo como um todo. Desta forma, uma operação que manipule dois objetos de um mesmo tipo, por exemplo uma soma de dois números complexos, só tem acesso aos detalhes de implementação de um deles. Esta característica, ausente em CLU, está presente na grande maioria das LOO e, como veremos mais adiante, tem grande influência em diversos aspectos destas linguagens.

Esta estrutura traz diversas vantagens para uma linguagem: facilidade para se tratar tipos com implementações diferentes

[6] É interessante observar que Modula-2 também obriga o uso de variáveis referenciais para tipos abstratos, uma vez que tipos opacos têm que ser ponteiros.

coexistindo num mesmo programa[7], bem como variáveis que podem modificar sua representação em tempo de execução; facilidade para sistemas distribuídos, onde os vários parâmetros de uma operação podem estar em máquinas distintas; maior homogeneidade, com todos os parâmetros de uma operação tendo sua estrutura interna desconhecida, independentemente de seu tipo. Como principal desvantagem existe um problema com eficiência, sempre presente quando se restringe o que é conhecido.

No aspecto formal, este conceito não traz grandes consequências. Não é difícil provar que qualquer operação pode ser implementada desta forma. Tomando-se como exemplo o tipo *Bool*, a função *if_then_else_*, definida como

```
if_then_else_ (true,a,b) = a
if_then_else_ (false,a,b) = b,
```

só precisa conhecer o valor interno de seu primeiro argumento, e com ela conseguimos definir todos os outros conectivos. Com o tipo *Nat*, a operação de soma pode ser implementada usando-se a própria especificação

```
soma (0,n) = n
soma (succ(n),m) = succ(soma(n,m)),
```

que se baseia na representação interna de seu primeiro argumento e na operação de sucessor, supostamente disponível externamente.

Outro impacto importante de visibilidade instancial diz respeito à criação das instâncias. Em linguagens com TAD convencionais, como CLU, todas as operações estão encapsuladas no tipo (cluster), que existe sintaticamente e, portanto, permanentemente. Entre estas operações encontram-se as de criação de instâncias. Em objetos, com as operações encapsuladas dentro de cada objeto, estas só podem ser usadas quando o objeto já existe. Portanto a operação de criar objetos tem que estar em outro lugar.

As soluções mais comuns são o uso de instruções especiais para a criação de instâncias (p.e. Simula-67) ou a existência de objetos correspondentes às classes possuindo uma operação para este fim

[7]Sendo subtipos um caso particular desta situação, tem-se uma boa razão para a adoção deste mecanismo em LOO. Nos parece difícil a implementação de um mecanismo de herança em linguagens sem esta característica.

(p.e. Smalltalk). Um controle adequado de visibilidade destas operações pode vir a melhorar muito a modularidade da linguagem.

Late-Binding

O termo late-binding[8] significa que a ligação entre um nome de rotina e a rotina é feita em tempo de execução. Este tipo de facilidade, presente desde Simula-67, é fundamental para permitir que subclasses redefinam métodos herdados. Como uma variável de uma dada classe pode conter objetos de subclasses dessa, a determinação de que método usar numa chamada não pode ser feito durante a compilação.

O primeiro ponto de discussão deste mecanismo diz respeito à sua eficiência. Em Smalltalk, que usa late-binding em todas as chamadas com o binding sendo feito por comparação de strings e percorrendo a cadeia hierárquica de classes, o problema de desempenho é bastante crítico, o que tem contribuído para a opinião de que late-binding é sempre ineficiente. Outras linguagens, como C++ e Simula-67, tem um esquema bem mais eficiente. Usando tipagem estática e declaração prévia de rotinas que podem vir a ser redefinidas, estas linguagens só usam late-binding quando necessário e mesmo nestes casos o binding se resume a uma indireção. Eiffel parece ter um esquema mais sofisticado, baseado num algorítmo proprietário, que suporta herança múltipla, dispensa pré-declarações e faz a ligação em tempo constante[9].

Outro ponto importante é a disciplina imposta pela linguagem aos usos possíveis deste mecanismo. Em Smalltalk, com tipagem dinâmica, late-binding implica que uma instrução de chamada de método, estaticamente, não significa nada: o método a ser chamado dinamicamente pode ser qualquer um e fazer qualquer coisa. Sob o aspecto de prova de programas esta característica é deplorável. Em

---

[8] O termo dynamic-binding parece mais adequado, mas o primeiro já é consagrado na literatura.

[9] estas afirmações constam da referência [Meyer 88], que entretanto não apresenta nenhuma argumentação a respeito.

linguagens estaticamente tipadas o late-binding fica limitado à redefinição de rotinas em subclasses. Esta facilidade, por sua vez, é raramente controlada pela linguagem, que se limita a exigir compatibilidade de tipos entre a versão herdada e a nova.

A linguagem Eiffel procura dar uma semântica mais consistente para este mecanismo. Usando declarações de pré e pós-condições nos métodos, a linguagem exige que um método redefinido tenha pré-condições mais fracas e pós-condições mais fortes que o original. Desta forma, a substituição do método antigo pelo novo não deve causar mudanças na semântica do programa usuário, e não são necessários mecanismos de segunda ordem para formalização. Entretanto, esta restrição ainda está longe de resolver todos os problemas. Como as pré e pós-condições são escritas sobre a representação da classe, e não sobre sua especificação, não há possibilidade de abstração de dados. Além disto, este sistema não está embutido em nenhum modelo formal completo, que inclua as outras modificações possíveis na operação de subclasseamento, como modificação de estrutura ou do invariante.

3) Avaliação de alguns aspectos convencionais em LOO

Nesta seção discutimos linguagens orientadas a objetos sob um ponto de vista mais tradicional, fazendo uma comparação implícita entre estas linguagens e outras convencionais modernas, em particular Ada e Modula-2. Dois aspectos chave nesta comparação, tanto por sua importância numa LP como pela diferença de enfoque adotado pelas duas correntes, são modularidade e facilidades de abstração. Estes aspectos, que já são normalmente interdependentes, em LOO ficam mais interligados pelo fato destas linguagens usarem o mesmo mecanismo para abstração de dados e modularidade. Outros pontos discutidos nesta seção são mecanismos de controle, com ênfase em iteradores, e tipagem - sistemas de tipos.

Modularidade e Facilidades de Abstração

Este aspecto tem sido apontado como uma das grandes vantagens oferecidas por LOO. De fato, o suporte dado por estas linguagens

para programação com tipos abstratos de dados é bastante estimulante para uma programação mais modular. Entretanto, consideramos que ainda existem questões mal resolvidas a este respeito, que citamos a seguir.

O primeiro ponto que consideramos crítico, já citado na discussão sobre variáveis referenciais, é a questão da planaridade ("flatness") do espaço de objetos destas linguagens. No modelo de objetos não existe uma noção de hierarquia que permita encapsular determinados objetos dentro de outros e, em princípio, todos objetos são globais. De maneira geral, quando um método é ativado, não existe uma restrição de quais objetos no sistema ele pode acessar e/ou modificar. Se por um lado, uma especificação rigorosa de cada operação pode indicar quais objetos são afetados em cada caso, por outro lado, o universo de discurso destas especificações é o sistema como um todo[10], além de que, em casos de erros de programação, estes podem se propagar de formas inesperadas.

A planaridade também pode comprometer seriamente os mecanismos de abstração destas linguagens. Com variáveis referenciais, o retorno de um objeto interno (de sua referência, na verdade) como resposta a uma operação pode quebrar completamente a proteção de dados, tornando externamente acessíveis componentes internos de um tipo supostamente abstrato. A falha se deve ao fato do mecanismo de encapsulamento protejer apenas as variáveis internas de um objeto, mas não seus conteúdos. Numa linguagem orientada a objetos não planar deveria ser possível restringir a visibilidade de objetos, e não apenas de variáveis ou outros nomes, a determinados escopos.

Um agravante para o problema colocado acima é a forma de passagem de parâmetros adotada por estas linguagens, normalmente denominada "call-by-sharing". Com este mecanismo, são passados para a rotina os valores das referências contidas nos parâmetros

[10] este ponto é bem visível na linguagem de especificação Larch/CLU, onde existem assertivas do gênero *modifies nothing*, que para serem provadas exigem o uso da pseudo-variável σ, que representa o mapeamento de todos os objetos do sistema para seus valores.

reais, fazendo com que as rotinas invocante e invocada compartilhem os mesmos objetos. Desta forma, as variáveis da rotina invocante não podem ser alteradas (supostamente favorecendo a modularidade), mas os objetos aos quais elas se referem podem ser modificados sem restrição.

Outro ponto fraco na modularidade destas linguagens se refere a visibilidade de tipos ou classes. Nenhuma destas linguagens oferece mecanismos para controlar como as classes se referem entre si, e normalmente todas elas são visíveis globalmente num sistema, sem necessidade de cláusulas de exportação e/ou importação. Esta limitação é bastante sentida quando deseja-se construir um grupo de classes correlatas, como *Grafos*, *Arcos* e *Nós*, que tem que ser definidas como classes independentes, sem poder compartilhar nenhuma informação que não seja global[11]. Da mesma forma, não existe maneira de indicar se uma classe se baseia em outra a nível de especificação (para estender seus conceitos) ou a nível de implementação (para construir sua representação).

Um ponto controvertido a respeito de abstração em LOO diz respeito a facilidade de uma classe conhecer a estrutura interna de suas superclasses. Muitos autores (p.e. [Snyder 86]) consideram isto uma quebra de empacotamento, ao permitir que detalhes de um tipo abstrato sejam usados em outros tipos. De fato, a nível de manutenibilidade esta característica é bastante indesejável, pois modificações na estrutura interna de uma determinada classe podem introduzir erros nas suas subclasses. Desta forma, apesar do mecanismo de visibilidade instancial usado por estas linguagens, não temos o empacotamento usual conseguido em linguagens convencionais: protege-se o objeto mas não se protege a classe.

Estruturas de Controle

A maioria das LOO apresenta estruturas de controle convencionais, do estilo **if then else**, **while do**, etc. A grande

[11] Uma solução parcial para este problema é oferecida em C++, com o mecanismo de *friend classes*. Entretanto, este mecanismo fere o conceito de visibilidade instancial e prejudica a modularidade, expondo toda estrutura de uma classe para suas *friends*.

exceção neste aspecto fica por conta de Smalltalk, com seu conceito de *blocos*. Em Smalltalk, um bloco é um objeto que contém um trecho qualquer de programa, podendo ser executado através da mensagem *value*. Com o uso deste mecanismo, as estruturas de controle podem ser escritas na própria linguagem, sendo passíveis de alterações ou acréscimos por parte do programador. Algumas desvantagens deste mecanismo são 1) ineficiência, 2) pouca utilidade, pois são raros os programas com estruturas de controle não convencionais, e 3) dificuldades semânticas com a generalização do conceito de bloco. Como vantagens temos flexibilidade e economia de conceitos[12].

A homogeneidade em estruturas de controle simples não é mais encontrada quando se estuda iteradores. De maneira geral, podemos classificar os métodos de iteração em *exportação de dados* e *importação de ações*[13]. No primeiro método, o tipo fornece operações como Primeiro_Elemento e Próximo_Elemento para se percorrer sua estrutura. No segundo, é fornecido um procedimento Iterador que, recebendo como parâmetro uma dada operação, executa esta operação para todos os elementos da estrutura. O primeiro método pode ser usado em qualquer linguagem com abstração de dados, enquanto o segundo exige facilidades para manipulação de procedimentos.

CLU oferece uma estrutura explícita para este problema, sob a forma de *iterators*. Além de não oferecer uma flexibilidade razoável (tente fazer um merge de duas listas usando iteradores de CLU), a nosso ver o problema de iteradores sobre estruturas de dados é suficientemente importante para exigir uma solução elegante por parte de uma LP, mas não é tão relevante a ponto de justificar um novo mecanismo somente para este fim.

Em Smalltalk, com a existência de blocos, é dada certa ênfase

---

[12] esta vantagem é apresentada em [Goldberg & Robson 83]. A nosso ver, o conceito de bloco é bem mais complexo do que os conceitos envolvidos em estruturas convencionais, tanto de um ponto de vista prático como formal.

[13] esta classificação é apresentada em [Eckart 87], onde pode ser encontrada uma boa comparação dos dois métodos.

no esquema de importação de ações, sob a forma de vários métodos na classe *Collection* (métodos *do*, *select*, *collect*, ...). Entretanto, este mecanismo também não oferece flexibilidade suficiente para boa parte das situações comumente encontradas em programas reais (o exemplo do merge é novamente sintomático). Smalltalk apresenta outro esquema de iteração, usado somente para a classe *String*, que nos parece bastante atraente. Neste esquema, ao invés das operações de iteração serem fornecidas pelo objeto a ser iterado (strings), é criado um novo objeto (da classe *Stream*), responsável por guardar o estado da iteração. Desta forma evitam-se vários problemas encontrados no mecanismo normal de exportação de dados, como a possibilidade de esquecimento de inicializar ou finalizar uma iteração (pois o usuário não tem como usar o objeto iterador sem antes criá-lo) e impossibilidade de várias iterações simultâneas sobre uma mesma estrutura (basta criar vários objetos iteradores).

Tipagem

Uma discussão bastante completa sobre sistemas de tipos em LOO pode ser encontrada em [Danforth & Tombinson 88], enquanto [Ierusalimschy 87] apresenta uma boa comparação entre mecanismos de tipagem convencionais e de LOO. Aqui pretendemos dar uma rápida visão do problema, nos concentrando nos aspectos de polimorfismo apresentados nestas linguagens.

Pelo paradigma operacional de Smalltalk, o conceito de tipagem perde parte de sua importância. Graças ao sistema de troca de mensagens e visibilidade instancial, é impossível se aplicar uma operação a um tipo incorreto, pois as operações são vinculadas ao tipo. O que pode ocorrer é um objeto receber uma mensagem para o qual não possui um método adequado, gerando um erro em tempo de execução. Se interpretamos isto como um erro de tipo, podemos considerar que o tipo de um objeto, em Smalltalk, é dado pelo conjunto de mensagens de seu protocolo.

Esta caracterização (implícita) de tipos usada em Smalltalk é responsável pelo alto grau de polimorfismo apresentado pela linguagem. Qualquer rotina que opere sobre um parâmetro de uma determinada classe pode operar sobre parâmetros de qualquer outra

classe que apresente, em seu protocolo, as mensagens usadas pela rotina. Mesmo os casos de herança são tratados desta forma. Como o protocolo das classes geralmente está contido no protocolo das subclasses, estas podem substituir aquelas.

O grande defeito do sistema de tipos de Smalltalk é não permitir tipagem estática[1]. Tentando solucionar este problema as LOO posteriores incorporaram modificações de modo a permitir tipagem estática. A maioria optou por um sistema de tipos similar ao usado em Simula. Esta linguagem apresenta um sistema de tipos bastante convencional, com uma única modificação para tratar herança. Cada classe define um tipo, mas um objeto de uma determinada classe pode ser considerado como sendo de qualquer tipo que seja "ancestral" de seu tipo original. Se por um lado resolve-se o problema de tipagem estática, por outro lado o polimorfismo fica limitado aos casos de herança, o que é uma perda significativa.

Para recuperar parte desta perda, a linguagem Eiffel incorporou um mecanismo para definição de classes genéricas, parecido com o mecanismo de *generics* de Ada. Esta facilidade não é propriamente polimorfismo, pois cada instanciação de uma classe genérica tem um único tipo bem definido. De qualquer forma, aliando este mecanismo com herança múltipla, a linguagem parece ter conseguido um bom grau de flexibilidade.

Um ponto interessante a se observar é que a definição de tipos esboçada para Smalltalk, como sendo o conjunto de operações disponíveis sobre um objeto, é similar à usada em Russel, que também apresenta um grau de polimorfismo bastante elevado e tem tipagem estática. O principal defeito desta linguagem é a necessidade de se fornecer explicitamente os tipos de todos os parâmetros reais nas chamadas de procedimentos polimórficos, o que em determinados casos pode ser bem trabalhoso. Talvez com uso de

[1]Neste contexto, tipagem estática pode ser interpretada como a garantia, em tempo de compilação, de que em todo envio de mensagem esta pertence ao protocolo do receptor, ou seja, é impossível a ocorrência de erros "message not understood" em tempo de execução.

objetos, que de certa forma contém suas próprias operações, esta dificuldade possa ser superada.

4) Programação em Ponto Grande com LOO

Cox [em Cox 86] comenta que orientação a objetos é muito mais uma técnica de empacotamento de software do que de codificação. Nesta seção discutimos as facilidades fornecidas por LOO para estes aspectos, chamados genericamente de programação em ponto grande (em contraste com programação em ponto pequeno). Seguindo a linha adotada na seção anterior, vamos basear esta discussão numa comparação entre os mecanismos adotados por LOO e os adotados por linguagens convencionais, novamente representadas por Modula-2 e Ada.

Em linguagens convencionais, o principal mecanismo de suporte à programação em ponto grande é o módulo (package, em Ada e module, em Modula-2). Para programação baseada em tipos abstratos de dados podemos usar módulos diretamente (dados abstratos) ou tipos opacos exportados por módulos[2]. Em termos de facilidades oferecidas, podemos dizer que objetos ficam entre módulos e tipos opacos, ora se comparando a um (visibilidade instancial, binding), ora se comparando a outro (múltiplas instâncias, criação dinâmica). A comparação de objetos com tipos opacos já foi feita em trabalho anterior [Ierusalimschy 87] e, de certo modo, também na seção 3 deste trabalho. Deste modo vamos nos concentrar aqui num paralelo entre objetos e módulos.

Numa visão superficial existem certas semelhanças entre módulos e objetos. Ambos servem para empacotar detalhes de implementação, tornando visíveis apenas uma interface bem definida. Ambos tem como estrutura geral um certo conjunto de

---

[2] Cada uma destas opções tem seus atrativos, mas nenhuma das duas oferece suporte completo para tipos abstratos. Para uma comparação das duas opções ver [Chang, Kaden & Elliot 78]. Os problemas destas linguagens para suportar TADs são bem apresentados em [Sherman, Hisgen & Rosenberg 82].

variáveis internas, normalmente inacessíveis externamente, mais um conjunto de funções e procedimentos de certa forma relacionados, que tem acesso a estas variáveis. Em termos de tipos abstratos, ambos tem visibilidade instancial. Mas as semelhanças param por aí.

A primeira diferença relevante, talvez a mais importante, diz respeito ao status das duas entidades. Um módulo é uma entidade estática, enquanto um objeto é algo dinâmico. Em linguagens convencionais, não é possível a criação dinâmica de módulos nem sua manipulação dentro da linguagem. Em LOO, objetos são considerados cidadãos de primeira classe, pondendo ser criados e destruídos dinamicamente, passados como parâmetros para procedimentos, bem como armazenados em variáveis. Desta forma, estas linguagens servem tanto para programação em ponto pequeno quanto para programação em ponto grande.

Alguns autores (p.e. [Black et alii 87] e [Burstall & Lampson 84]) apontam esta unificação de linguagem em ponto grande e em ponto pequeno como uma das vantagens de LOO. Outros autores (p.e. [Ghezzi & Jazayeri 82]) sustentam a necessidade de linguagens distintas para estes dois níveis. De fato, existem prós e contras para as duas opções. A nosso ver, ogrande problema de duas linguagens é a dicotomia rígida imposta aos níveis de programação. Por que não dispor de uma linguagem diferente para programação em ponto médio? Uma pilha e um sistema de arquivos são ambos abstrações de dados coesas e bem definíveis. Ao colocarmos cada um dentro de um módulo estamos considerando que ambos estão no mesmo "ponto"? Se consideramos uma pilha como algo muito mais simples que um sistema de arquivos (e portanto em"pontos" diferentes) necessitamos de mecanismos de abstração diferentes, bem como diferentes ferramentas para manipulação de ambos (reaproveitamento, generalidade, controle de interfaces).

Uma solução para a unificação dos dois níveis de programação passa necessariamente pelo problema de tempo de binding. Mesmo em linguagens convencionais, o binding entre módulos muitas vezes é feito em tempo de execução. O caso típico desta situação é a ligação entre um programa em execução e o sistema operacional da

máquina. Algumas implementações de Modula-2 também usam binding dinâmico, deixando para carregar e ligar um módulo quando ele é usado pela primeira vez.

A principal motivação para uso de módulos, que é uma separação nítida entre especificação e implementação, não é sustentada por nenhuma das LOO correntes[3]. Em Modula-2 esta separação é bastante bem construída, existindo unidades de compilação independentes para descrever a interface de um módulo e sua implementação. Em Ada, apesar de não haver unidades independentes para as duas descrições, estas são totalmente separadas dentro de cada package. A existência de especificações separadas em LOO ajudaria bastante na distinção entre herança de especificação (hierarquia de módulos de definição) e herança de implementação (hierarquia de módulos de implementação).

Outro ponto fraco de objetos é que estes, ao contrário de módulos, não podem exportar tipos (ou classes). Esta fraqueza gera outra, já relatada na seção anterior, de todas as classes serem globais. Não há maneira de se empacotar tipos fortemente coesos, como por exemplo *Arquivo* e *SistemaDeArquivos*. Da mesma forma, tipos não podem compartilhar detalhes que não sejam globais. Uma solução para isto é apresentada em [Madsen 86]. Nesta solução, cada objeto (e não a classe) pode exportar outras classes, definidas internamente ao objeto e com visibilidade para a estrutura deste objeto. Um primeiro problema com esta proposta é a perda da visibilidade instancial. O segundo problema se refere a compatibilidade destes tipos[4]. A última questão se refere à herança neste contexto, não tratada no trabalho citado.

5) Conclusões

---

[3] O melhor que se tem nesta área é Eiffel, que permite se selecionar os nomes visíveis externamente.

[4] Se uma classe $\mathcal{C}$ exporta um tipo T, x e y são objetos da classe $\mathcal{C}$, e tx tem tipo $x.T$, sabe-se que tx não é compatível com $y.T$. Entretanto, o que ocorre com o tipo de tx se é feita uma atribuição como $x := y$?

Fizemos um apanhado geral das principais características das LOO existentes atualmente. Analisamos uma caracterização destas linguagens através de mecanismos comuns a todas elas e vimos como estes mecanismos afetam a qualidade destas linguagens, tanto para programação em ponto pequeno quanto em ponto grande.

Como principais problemas encontrados nestas linguagens podemos citar:

- Falta de uma separação nítida entre especificação e implementação de classes e, consequentemente, uma mistura entre as hierarquias de classes e de tipos. Para solucionar este ponto é fundamental um conceito de tipos abstratos que inclua estados e herança. A teoria usada em VDM, baseada em modelos, nos parece bastante apropriada. Primeiro por incluir naturalmente a noção de estados. Segundo pela independência entre o espaço de estados, definido pelo modelo e a invariante, e as operações sobre estes estados, facilitando alterações nas operações ou no espaço de estados sem efeitos colaterais.

- Falta de uma semântica mais simples para algumas construções, notadamente late-binding e variáveis referenciais. Achamos que estes mecanismos devem ser limitados a determinados usos para virem a ser melhor compreendidos e, consequentemente, melhor utilizados. Para late-binding, o uso de pré- o pós-condições, como feito em Eiffel, com uma semântica de tipos abstratos mais consistente pode ser uma solução.

- Falta de modularidade, como consequência do uso indiscriminado de variáveis referenciais. Novamente, a limitação do uso, através de um controle de visibilidade sobre objetos (e não apenas sobre nomes), pode melhorar este ponto.

- O mecanismo de encapsulamento, a classe, não é suficientemente versátil para programação em ponto grande. A inclusão da facilidade de exportar classes, como um módulo faz, parece importante. De certa forma, precisamos de um conceito de objeto

que ora se comporte como valor, com early-binding, atribuição e tipagem estática, e ora se comporte como um módulo, algo estático com late-binding e tipagem dinâmica. Tratando classes exportadas como coleções de Euclid, este conceito pode facilitar também o tratamento do ítem anterior.

Por fim, achamos que uma linguagem que solucione estes problemas pode vir a ser bastante útil como forma de juntar as vantagens de orientação a objetos sem perder o que já foi conquistado com linguagens convencionais, do ponto de vista de viabilizar o desenvolvimento sistemático e até formal de programas.

Referências e Bibliografia Comentada

* Descrições e manuais das várias linguagens citadas no texto:

-Ada

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

[Gehani 83]
Gehani,N. **Ada - An Advanced Introduction**, Prentice-Hall (Software Series), 1983.

-CLU

[Liskov & Atkinson 77]
Liskov,B. & Atkinson,R. Abstraction Mechanisms in CLU, **CACM** 20(8), 1977.

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

-C++

[Stroustrup 86]
Stroustrup,B. **The C++ Programming Language**, Addison-Wesley, 1986.

-Eiffel

[Meyer 88]
Meyer,B. Eiffel - A Language and Environment for Software Engineering, **The Journal of Systems and Software** 8(3), pp. 129-246, 1988.

-Emerald

[Black et alii 87]
Black,A. et alii. Distribution and Abstract Types in Emerald,

IEEE Trans. on Soft. Eng. SE-13(1), pp 65-76, 1987.

-Euclid

[Lampson et alii 77]
Lampson,B. et alii. Report on the Programming Language Euclid, Sigplan Notices 12(2), 1977.

-Modula-2

[Wirth 85]
Wirth,N. Programming in Modula-2, Springer-Verlag, 1985.

-Modula-3

[Glassman & Nelson 88]
Modula-3 Report, Preliminary draft.

-Objective-C

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

-OBJ2

[Futatsugi et alii 84]
Futatsugi,K. et alii. Principles of OBJ2, in proceedings of Twelfth ACM Symposium on Principles of Programming Languages, pp. 52-66, 1984.

-Russel

[Boehm, Demers & Donahue 85]
Boehm,H.; Demers,A.; Donahue,J. A Programmer's Introduction to Russel, TR 85-16, Computer Science, Rice University, 1985.

-Simula

[Birtwistle et alii 75]
Birtwistle,G. et alii. Simula Begin, Petrocelli/Charter, 1975.

-Smalltalk-80

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

[Goldberg 84]
Goldberg,A. Smalltalk-80: The Interactive Programming Environment, Addison-Wesley, 1984.

[Krasner 83]
Krasner,G. (ed.) Smalltalk-80: Bits of History, Words of Advise, Addison-Wesley, 1983.

[Byte 81]
Byte 6(8), 1981. (edição dedicada a Smalltalk-80)

-Trellis/Owl

[Schaffert et alii 86]
Schaffert,C. et alii. An Introduction to Trellis/Owl,

(OOPSLA'86 proceedings) Sigplan Notices 21(11), pp. 9-16, 1986.

* Outras Linguagens Relevantes:

-Mesa e Cedar

[Sweet 85]
Sweet,R. The Mesa Programming Environment, Sigplan Notices 20(7), 1985.

[Donahue 85]
Donahue,J. Integration Mechanisms in Cedar, Sigplan Notices 20(7), 1985.

[SwineHart, Zellweger & Hagmann 85]
SwineHart,D.; Zellweger,P.; Hagmann,R. The Structure of Cedar, Sigplan Notices 20(7), 1985.

[SwineHart et alii 86]
SwineHart,D. et alii. A Structured View of the Cedar Programming Environment, ACM TOPLAS 8(4), pp. 419-490, 1986.

[Teitelman 85]
Teitelman,W. A tour Through Cedar, IEEE Trans. on Soft. Eng. SE-11(3), 1985.

A linguagem Cedar, evolução de Mesa, também foi desenvolvida por Xerox PARC, e inicialmente planejava incorporar num mesmo ambiente as vantagens de Smalltalk, Mesa e InterLisp. A linguagem e o ambiente tem uma forte preocupação com modularidade e níveis de abstração. Como mecanismo de integração entre módulos sustenta um uso intensivo de passagem de procedimentos, e em várias aplicações simula objetos com registros alocados dinamicamente (a linguagem tem garbage collection) e contendo em alguns campos referências a procedimentos.

-ML

[Harper 85]
Harper,R. **Introduction to Standard ML**, (preliminary draft), Computer Science Department, University of Edinburg, october, 1985.

A linguagem ML foi a precursora de linguagens polimórficas estaticamente tipadas. Sem declarações de tipos, a linguagem deduz para cada função o seu tipo mais geral, usando um algorítmo de unificação. Posteriormente incorporou um mecanismo para definição de tipos abstratos, mas o polimorfismo não se aplica a estes tipos.

-Oberon

[Wirth 88]
Wirth,N. Type Extensions, ACM TOPLAS 10(2), pp. 204-214, 1988.

[Wirth 88]
Wirth,N. The Programming Language Oberon & From Modula to Oberon, Software-Practice and Experience 18(7), pp. 661-670.

1988.

Esta nova linguagem de N. Wirth simplificou bastante Modula-2. A maior parte destas simplificações são mal justificadas e parecem piorar muito a linguagem. O ponto relevante para nós é o conceito de *type extensions*, onde records podem ser definidos como extensões de outros, numa clara analogia com subclasseamento. A modificação do sistema de tipos para esta situação é bastante convencional, mas a linguagem também permite atribuições de valores de tipos para supertipos, através de projeções.

* Tipos e Tipagem:

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

Apresenta um modelo semântico para herança múltipla. O modelo é totalmente baseado na implementação dos tipos, sem nenhuma preocupação com seus comportamentos.

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

Apresenta um modelo matemático para tipos com várias formas de polimorfismo, incluindo herança múltipla e TAD. Seguindo ML, procura sistemas de tipos que não necessitem de declaração, o que não é razoável de um ponto de vista prático.

[Cardelli 89]
Cardelli,L. **Typefull Programming**, in State of the Art Seminar on Formal Description of Programming Concepts, Rio de Janeiro, Brasil, 1989.

Apresenta a linguagem Quest, que une polimorfismo parametrizado com subtipos. A linguagem é basicamente funcional, mas apresenta alguns rudimentos de linguagens imperativas. O sistema de tipos é de segunda ordem, existindo a noção de Kind onde 'habita' o conceito Type, o tipo (na verdade o Kind) de todos os tipos.

[Danforth & Tombinson 88]
Danforth,S. & Tombinson,C. Type Theories and Object-Oriented Programming, **ACM Computing Surveys 20(1)**, pp. 29-71, 1988.

Ótima pesquisa sobre sistemas de tipos para suportar herança, mas muito centrado em linguagens funcionais. Não dá muita atenção para outras formas de polimorfismo. Alguns dos problemas colocados no final do artigo também nos afligem.

[Booth 86]
Booth,D. **Multiple Strongly Typed Evaluation Phases: A Programming Language Notion**, Technical Report CSD-860042,

UCLA, Computer Science Department, 1986.

Apresenta uma linguagem onde a execução de um programa acontece em várias fases, e cada fase calcula os tipos necessários para a fase seguinte. Uma generalização das fases de compilação e execução.

[Donahue & Demers 85]
Donahue,J. & Demers,A. Data Types are Values, **ACM TOPLAS** 7(3), 1985.

Apresenta o sistema de tipos de Russel, onde tipos são considerados como coleções de operações. Apesar de considerar uma ordem parcial para tipos baseado nestas coleções, não chega a herança. Sistema altamente polimórfico.

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

Lança as idéias do sistema de tipos de ML.

[Morris 73]
Morris,J. Types are Not Sets, **ACM Symposium on Principles of Programming Languages**, 1973.

Sustenta a idéia de que o conjunto de operações permitidas sobre um tipo faz parte da definição deste tipo. Deu origem ao sistema de tipos de CLU.

*Programação com Tipos Abstratos de Dados:

[Berztiss & thatte 83]
Berztiss,A. & Thatte,S. Specification and Implementation of Abstract Data Types, **Advances in Computers** 22, pp. 296-354, 1983.

Survey sobre TAD, muito baseado em especificações algébricas. Na parte referente a implementações, apresenta uma boa discussão do problema de iteradores e de identidade de estruturas sob modificação de seus elementos.

[Chang, Kaden & Elliot 78]
Chang,E.; Kaden,N.; Elliot,W. Abstract Data Types in Euclid, **Sigplan Notices** 13(3), 1978.

Apresenta as duas maneiras de se implementar tipos abstratos em Euclid: através de tipos opacos ou usando módulos diretamente.

[Eckart 87]
Eckart,J. Iteration and Abstract Data Types, **Sigplan Notices** 22(4), 1987.

Coloca e discute a oposição entre importar ações versus exportar dados em iteradores.

[Ierusalimschy 87]
Ierusalimschy,R. **Abstração de Dados em Linguagens de Programação: Tipos e Objetos**, Depto. Informática, PUC-RJ.

Monografias em Ciência da Computação 4/87, 1987.

Compara os mecanismos para programação com tipos abstratos fornecidos por linguagens convencionais e por linguagens orientadas a objetos.

[Sherman, Hisgen & Rosenberg 82]
Sherman,M.; Hisgen,A.; Rosenberg,J. A Methodology for Programming Abstract Data Types in Ada, **Proceedings of the AdaTEC Conference on Ada**, 1982.

Apresenta várias regras para programação com TAD em Ada. A necessidade destas regras mostra claramente os defeitos de Ada para suportar programação com tipos abstratos.

[Veloso 86]
Veloso,P. **Tipos (Abstratos) de Dados: Programação, Especificação, Implementação**, V Escola de Computação, 1986.

Teoria axiomática de tipos abstratos de dados.

* Sistemas de Modularização:

[Agnarsson & Krishnamoorthy 85]
Agnarsson,S. & Krishnamoorthy,M. Towards a Theory of Packages, **Sigplan Notices** 20(7), 1985.

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

Apresenta uma linguagem funcional para descrição de mecanismos de encapsulamento. Tratando módulos como valores, compara especificações com tipos para estes valores.

[Drossopoulou, Eisenbach & McLoughlin 87]
Drossopoulou,S.; Eisenbach,S.; McLoughlin,L. **Module and Type Systems: a Tour**, Imperial College, august 1987.

Não apresenta nada de novo, mas é um ótimo survey sobre o assunto.

[Drossopoulou 87]
Drossopoulou,S. **Proposal for a Module System: Parametrization and Abstraction**, Imperial College, november 1987.

[Madsen 86]
Madsen,O. Block Structure and Object Oriented Languages, **Sigplan Notices** 21(10), pp. 133-142, 1986.

Apresenta a idéia de objetos exportando classes como forma de estruturação de programas em linguagens orientadas a objetos.

[Snyder 86]
Snyder,A. Encapsulation and Inheritance in Object-Oriented Programming Languages, (OOPSLA'86 Proceedings) **Sigplan Notices** 21(4), pp. 38-45, 1986.

Discute o problema de encapsulamento gerado pela visibilidade que subclasses tem de suas superclasses, e defende como

solução que subclasses devem usar funções para acessar variáveis definidas em suas superclasses.

[Wirfs-Brock & Wilkerson 88]
Wirfs-Brock,A. & Wilkerson,B. An Overview of Modular Smalltalk, (OOPSLA'88 Proceedings) Sigplan Notices 23(11), pp.123-134, 1988.

Apresenta a linguagem Modular Smalltalk. Fazendo algumas restrições aos mecanismos dinâmicos de Smalltalk, implementa sobre a linguagem um sistema de módulos convencional, para controle sintático de visibilidade de nomes.

* Métodos Formais para Especificação e Desenvolvimento de Programas:

[Bjorner & Jones 82]
Bjorner,D. & Jones,C. **Formal Specification & Software Development**, Prentice-Hall International, (Series in Computer Science), 1982.

Coletânea de trabalhos apresentando os vários aspectos de VDM, desde a linguagem até especificações completas de algumas linguagens de programação, passando por seus fundamentos matemáticos.

[Dijkstra 76]
Dijkstra,E. **A Discipline of Programming**, Prentice-Hall (Series in Automatic Computation), 1976.

Apresenta o seu método, baseado no conceito de pré-condições mais fracas.

[Gehani & McGettric 86]
Gehani,N. & McGettric,A. **Software Specification Techniques**, Addison-Wesley (Computer Science Series), 1986.

Coletânea de artigos procurando apresentar o estado da arte em técnicas de especificação formal. A maioria dos artigos é bastante antiga, dada a data de publicação do livro.

[Hoare 72]
Hoare,C. Proof of Correctness of Data Representation, **Acta Informatica 1**(4), pp. 271-281, 1972.

Apresenta uma técnica para prova de programas que usam abstração de dados. Esta técnica se baseia em criar modelos matemáticos isomórficos ao tipos sendo definidos. Este artigo axiomatiza as classes de Simula, mas depois a mesma técnica foi usada para Euclid.

[Jones 86]
Jones,C. **Systematic Software Development Using VDM**. Prentice-Hall International, (Series in Computer Science), 1986.

Apresenta uma técnica de desenvolvimento de programas apoiada em tipos abstratos de dados. As especificações de funções, procedimentos e tipos abstratos é feita em VDM.

[Liskov & Zilles 75]
Liskov,B. & Zilles,N. Specification Techniques for Data Abstractions, **IEEE Trans. on Soft. Eng. SE-1**(1), 1975.

[Liskov & Guttag 86]
Liskov,B. & Guttag,J. **Abstraction and Specification in Program Development**, The MIT Press, 1986.

Livro didático sobre desenvolvimento de programas usando CLU. Grande ênfase em tipos abstratos. Especificações formais são escritas em Larch/CLU, linguagem de especificação baseada na teoria algébrica de TAD. Muita atenção para especificações semi-formais.

[Martins & Moura 88]
Martins,R. & Moura,A. **Desenvolvimento Sistemático de Programas Corretos: A Abordagem Denotacional**, VI Escola de Computação, 1988.

Ênfase no background matemático para semântica denotacional.

[Turski & Maibaum 87]
Turski,W. & Maibaum,T. **The Specification of Computer Programs**, Addison-Wesley, 1987.

Sustenta o uso de lógica para especificação de programas. Apresenta uma formalização para o método de desenvolvimento por refinamentos sucessivos, usando interpretação entre teorias em lógica de primeira ordem.

* Fundamentos de Linguagens de Programação:

[Ghezzi & Jazayeri 82]
Ghezzi,C. & Jazayeri,M. **Programming Language Concepts** (2/e), John Wiley & Sons, 1982.

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