Exploração de reflexão computacional através de um modelo de objetos sem classes

Hisham H. Muhammad
Ana Paula Lüdtke Ferreira (orientadora)

[1]PIPCA- Programa de Pós-Graduação em Computação Aplicada
Centro de Ciências Exatas e Tecnológicas
Unisinos - Universidade do Vale do Rio dos Sinos
Av. Unisinos, 950 - São Leopoldo, RS - CEP93022-000

{hisham,anapaula}@exatas.unisinos.br

Abstract. *Linguagens de programação orientadas a objeto provêem mecanismos para permitir a extensibilidade das aplicações desenvolvidas, permitindo contemplar o surgimento de novos requisitos durante o ciclo de vida de um projeto. Por outro lado, a extensibilidade das linguagens em si tende a ser negligenciada, em favor de uma série de extensões* ad hoc *à linguagem original, aumentando a sua complexidade. Mecanismos de extensibilidade de linguagens requerem acesso e manipulação de meta-informação, o que é tipicamente obtido através de reflexão computacional. Entretanto, o aumento de complexidade no modelo de dados que ocorre ao adicionar-se capacidades reflexivas a uma linguagem, associado à dificuldade de implementação de recursos de reflexão em linguagens cujo projeto não a proveu originalmente, tem impedido que o potencial da reflexão seja explorado em sua totalidade.*
Este trabalho explora as vantagens trazidas pela reflexão computacional através de uma linguagem imperativa orientada a objetos com um sistema de tipos seguro. Para isto, está sendo proposta uma linguagem de programação que possui uma arquitetura de reflexão sobre um modelo de objetos sem classes. As formas de reflexão suportadas por esta arquitetura são introspecção e intercessão estrutural e comportamental, utilizando técnicas aplicáveis em tempo de compilação.

1 Introdução

Linguagens de programação orientadas a objeto provêem mecanismos para permitir a extensibilidade das aplicações desenvolvidas, permitindo contemplar o surgimento de novos requisitos durante o ciclo de vida de um projeto. Muitos destes requisitos implicam na necessidade da adaptação de uma nova camada de suporte à própria linguagem a fim de facilitar o desenvolvimento de uma aplicação a uma determinada área. Exemplos de cenários como este são adição de persistência para banco de dados, suporte a mobilidade para sistemas distribuídos, e paralelismo para processamento de alto desempenho. No entanto, em muitos casos como esses, pode ser mais interessante estender a *definição* de uma linguagem para contemplar finalidades para as quais ela não foi originalmente projetada. Infelizmente, a extensibilidade das linguagens em si tende a ser negligenciada [4], em favor de uma série de extensões *ad hoc* à linguagem original, aumentando a sua complexidade. Exemplos disto são *templates* em C++ [35] e o sistema de eventos em C# [12].

Mecanismos para este tipo de extensibilidade, quando embutidos na própria linguagem, requerem acesso e manipulação de meta-informação, isto é, informação sobre as características ou o estado do próprio programa. Isto é tipicamente realizado utilizando reflexão computacional, através da produção de uma sistemática para manipulação não apenas dos dados e do código, mas também da constituição e comportamento destes, conhecida como *protocolo de meta-objetos* [16]. Reflexão computacional, portanto, pode ser definida como a capacidade de um processo computacional de "manipular formalmente representações de suas próprias operações e estruturas" [32]. Linguagens com suporte a reflexão possibilitam ao programador observar o estado interno do programa (processo denominando *introspecção*) ou alterá-lo (*intercessão*).

Adicionar recursos reflexivos a uma linguagem orientada a objetos típica envolve passar-se de um modelo de "objetos e classes" para um modelo de "objetos, classes, e meta-classes". É difícil, ao se estender uma linguagem existente, manter a ortogonalidade entre construções sintáticas que tratam das estruturas nestes diferentes níveis. Este fato, associado à dificuldade de implementação de recursos de reflexão em linguagens cujo projeto não o proveu originalmente—desde a própria adaptação do modelo de dados até questões de compatibilidade de código—tem impedido que o potencial da reflexão seja explorado em sua totalidade [10, 11]. Um problema comum enfrentado é o enfraquecimento da segurança dos tipos de dados em tempo de execução que ocorre ao se adaptar objetos de um nível a outro do modelo através de coerção. Como resultado disto, linguagens como JAVA possuem apenas recursos limitados de introspecção [15].

Outra peculiaridade de linguagens reflexivas é o fato de que torna-se possível manipular as descrições de objetos de forma similar à que os objetos em si são manipulados, isto é, aquilo que tradicionalmente é visto como classe torna-se também objeto. Este processo no qual entidades abstratas são convertidas a entidades concretas, passíveis de manipulação é chamado de **reificação** [16]. O objeto decorrente deste processo possui uma classe (isto é, uma meta-classe) passível de ser reificada. Torna-se claro que, neste contexto, todas as entidades do programa podem ser encaradas como objetos.

A fim de melhor representar sintaticamente esta fundamental característica da semântica reflexiva, faz sentido optar-se por um modelo de objetos sem classes (também conhecido como baseado em protótipo). No modelo baseado em protótipo, todo objeto é criado a partir de outro objeto que age como superclasse. É importante notar que o poder descritivo de um modelo sem classes é equivalente a um modelo com classes [38].

O objetivo deste trabalho é propiciar a exploração das vantagens trazidas pela reflexão computacional através de uma linguagem com um fluxo de execução imperativo e um modelo de dados orientado a objetos e com um sistema de tipos seguro. Para isto, está sendo proposta uma linguagem de programação que possui uma arquitetura de reflexão sobre um modelo de objetos sem classes onde a mesma construção atua como objeto, classe ou meta-classe, de modo a produzir um protocolo de meta-objetos com menos diferenciação sintática entre os meta-níveis. As formas de reflexão suportadas por esta arquitetura são introspecção e intercessão estrutural e comportamental, com ênfase em técnicas passíveis de implementação em tempo de compilação.

Este trabalho está organizado da seguinte forma: na Seção 2 será apresentada a revisão bibliográfica, traçando um panorama das linguagens de programação que possuem capacidades reflexivas ou possuem um modelo de dados baseado em protótipo. A Seção 3 apresenta a linguagem GLASS, detalhando o modelo de dados e o sistema de tipos utilizado, bem como as construções da linguagem que tratam do seu suporte a reflexão. A Seção 4 relata o desenvolvimento do protótipo tendo como foco as necessidades causadas pelos recursos reflexivos da linguagem e especifica a semântica destes recursos.

Finalmente, na Seção 5 tecem-se considerações finais e enumeram-se possibilidades para trabalhos futuros.

2 Revisão bibliográfica

2.1 Linguagens reflexivash

A pesquisa em reflexão computacional teve suas raízes em LISP, levando à formalização do conceito de reflexão por Smith através das linguagens 2-LISP e 3-LISP [33], com o conceito da *torre reflexiva*: a visão do ambiente de execução como uma torre potencialmente infinita, onde cada estágio de execução representa, metaforicamente, um interpretador sendo interpretado. Este conceito está representado do lado esquerdo da Figura 1.

A partir de então, uma série de linguagens funcionais têm explorado capacidades reflexivas. A linguagem CLOS (COMMON LISP OBJECT SYSTEM, [13]) possui um protocolo de meta-objetos bem definido, com um suporte bastante completo a introspecção e intercessão, disponibilizando ao programador uma interface às estruturas internas do seu interpretador. As linguagens da família ABCL/R [23] exploram técnicas de reflexão em tempo de compilação, postergando a criação dos níveis da torre até o momento onde é realizado acesso a elas, de modo a não impor sobrecusto a código que não realiza manipulações reflexivas.

Técnicas envolvendo reflexão têm sido também exploradas em linguagens imperativas orientadas a objeto. Neste modelo, devido ao polimorfismo, a coleção métodos associados a um objeto ou classe é armazenada em uma estrutura de dados dinâmica. Assim, a chamada de construtores, destrutores e métodos envolve a execução implícita de um algoritmo de *method dispatching*, que numa visão reificada da classe, poderia ser entendida como um método `classe.chamarMetodo()`. Novamente a metáfora da torre infinita, adaptada ao paradigma orientada a objetos (Figura 1).

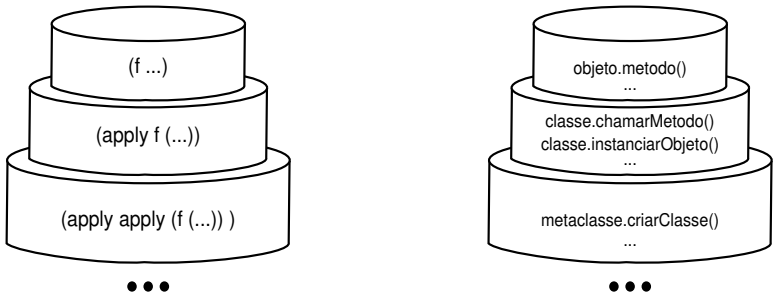


Figura 1: A torre reflexiva, nos paradigmas funcional e orientado a objetos.

A linguagem JAVA [15] possui certo grau de suporte a reflexão, apresentado como uma API especial que dispõe de acesso a informações de estado da máquina virtual. As suas capacidades, entretanto, se restringem a introspecção, permitindo ao programa, por exemplo, consultar quais os métodos de uma classe e realizar chamadas a eles, sem permitir alterá-los.

SMALLTALK [14] possui um modelo de meta-classes implícito. Para cada classe criada, o *kernel* do interpretador cria uma meta-classe correspondente, embora não haja flexibilidade na manipulação desta. NEOCLASSTALK [29] é uma extensão que define uma nova hierarquia de meta-classes sobre a qual a hierarquia de classes original é reintroduzida. Isto expõe uma característica muito importante do uso de reflexão: o programa que está em um nível da torre não precisa possuir conhecimento sobre o que está abaixo de si para que algum nível inferior adicione alguma funcionalidade. De fato, uma das

aplicações de reflexão exploradas na atualidade é a sua utilização como suporte a implementação de *middleware adaptativo* [20].

OPENC++ [6] é um pré-processador para C++ que adiciona à linguagem capacidades reflexivas em tempo de compilação e cuja implementação pode ser encarada como um sistema de macro-processamento de alto nível. Uma abordagem diferente é empregada pelo MOC (META OBJECT COMPILER, [36]), outro pré-processador para C++ que estende a linguagem original. O MOC foi criado especificamente para uso com o *toolkit* de interface gráfica QT, que internamente utiliza os meta-objetos para fazer o controle do *dispatching* de eventos.

Por se tratar de uma característica que permeia o sistema de tipos como um todo, adicionar extensões reflexivas a linguagens já existentes de uma forma transparente é extremamente difícil[1]. Corre-se o risco de o sistema de reflexão apresentar limitações "artificiais" provocadas pelas restrições da linguagem original. Ao mesmo tempo, a complexidade da linguagem aumenta, pois serão sempre acrescentados novos elementos ao *design* mesmo em situações onde poderia ser mais vantajoso modificar o *design* existente, a fim de manter a compatibilidade com versões anteriores.

2.2 Linguagens baseadas em protótipo

O modelo de objetos utilizado pela linguagem proposta por este trabalho utiliza um modelo de objetos sem classes, também conhecido como modelo baseado em protótipos ou *object-based*. Apesar de possuírem poder descritivo equivalente ao de linguagens com classes [38], nenhuma das linguagens baseadas em protótipo obteve grande popularidade, e mesmo no meio acadêmico esta abordagem não é bastante explorada. Os principais motivos para isto são a popularidade de linguagens orientadas a objeto que utilizam classes e a própria equivalência de poder, sem nenhum ganho óbvio que contribua para a opção pelo modelo sem classes. Como veremos, exploradas dentro do contexto de reflexão, linguagens baseadas em protótipo podem carregar consigo um novo potencial.

SELF [37] é o principal exemplo de linguagem baseada em um modelo de objetos sem classes. Ela possui uma forte herança de SMALLTALK, e, como esta, também utiliza tipagem dinâmica (isto é, onde os valores têm tipos, e não as variáveis). Grande número de linguagens baseadas em protótipo possuem uma forte influência de SELF. Uma delas é CECIL [5], que combina a este modelo de protótipos recursos de reflexão em tempo de execução, ao custo de grande queda no desempenho.

A linguagem NEWTONSCRIPT [34] foi criada com o objetivo de explorar a flexibilidade da prototipação característica de SELF no projeto de interfaces gráficas. Esta experiência mostrou que o modelo de objetos sem classes se adaptava bem a linguagens interpretadas, onde o sobrecusto de estruturas dinâmicas já está inerentemente presente. Este mesmo modelo foi depois empregado em JAVASCRIPT, uma linguagem interpretada desenvolvida para *client-side scripting*, isto é, execução no navegador.

2.3 Estratégias para exploração de reflexão

O poder de abstração de uma linguagem pode ser largamente ampliado ao explicitar-se a hierarquia de meta-classes; entretanto, a simples disponibilidade desta hierarquia é de pouca valia para o desenvolvimento de software aplicativo. Com o amadurecimento das técnicas para implementação de reflexão computacional, uma série de grupos de pesquisa têm se dedicado a estudar estratégias para exploração eficiente destas técnicas no contexto de engenharia de software. As principais abordagens serão listadas a seguir.

[1] O caso de NEOCLASSTALK foi facilitado pelo fato de SMALLTALK já possuir uma arquitetura, ainda que pouco flexível, de meta-classes.

Component filters [1] foi uma das primeiras técnicas de alto nível para composição de hierarquias de código em diferentes níveis de abstração. Basicamente, interações entre objetos passam a envolver objetos intermediários que funcionam como "filtros", implementando características adicionais a estas interações, como, por exemplo, serialização e sincronização de objetos. De fato, *component filters* foram desenvolvidos visando o seu uso em sistemas distribuídos.

Uma técnica que vem ganhando proeminência é o *Aspect-Oriented Programming* (AOP), apresentado em [17]. A mais proeminente linguagem que implementa AOP é ASPECTJ [18], uma extensão de JAVA. O princípio básico da AOP se baseia no fato de que certas características permeiam classes não relacionadas através da hierarquia de herança. *Aspects* são construções que podem referenciar-se a métodos de diferentes classes, instrumentando-os. Esta capacidade pervasiva dos *aspects*, entretanto, pode ser considerada, de certa forma, uma quebra no encapsulamento das classes. Um *aspect* complexo possuirá uma série de referências a diversos métodos; mudanças nas classes deverão implicar na revisão dos *aspects*.

Generative programming [9] surgiu como um modelo de desenvolvimento de alto nível baseado em *template meta-programming* [8]. Este modelo explora os recursos para composição de código providas por *templates* em C++ de modo a construir objetos complexos baseado em sobreposição de camadas em tempo de compilação. Ao prover recursos genéricos para declaração de transformações, uma linguagem reflexiva pode ser utilizada como ferramenta para generative programming. O comportamento original pode ser programado em um objeto e um conjunto de transformações pode configurar um objeto derivado, que então serve como objeto gerador protótipo. Isto atende às necessidades de composição de código requerido pelo modelo, e evita a necessidade de uma sublinguagem de *template meta-programming* como a (acidentalmente) disponibilizada por C++.

3 A linguagem GLASS

3.1 Entidades primitivas

Em uma linguagem que permite a criação de tipos de dados pelo usuário, os tipos pré-definidos pela própria linguagem são chamados tipos primitivos. O motivo real pelo qual a distinção é feita é o fato de que os tipos primitivos recebem tratamento especial do ambiente de compilação/interpretação, tipicamente em forma de regras sintáticas específicas para receber valores constantes. Em uma linguagem orientada a objetos reflexiva como a proposta aqui, outros tipos de construções também são considerados dados, e portanto, necessitam também receber tratamento especial. São consideradas as seguintes entidades primitivas. **sinais, referências, objetos**, **métodos** e **assinaturas**[2]. Elas serão apresentadas ao longo desta Seção.

Sinais. Definir uma técnica para introspecção comportamental de um programa imperativo envolve inferir sobre o código seqüencial de seus métodos. Buscou-se moldar uma ferramenta lingüística de alto nível que mantivesse as propriedades de encapsulamento dos objetos submetidos à introspecção. À solução encontrada deu-se o nome de **sinalização**[3]. Aos métodos podem ser associados *sinais*, que nada mais são que *decorations* associadas a estes em tempo de compilação. Estes sinais são declarados para que os

[2]Assume-se que entidades que em linguagens procedurais são consideradas primitivas, como `INTEGER`, em GLASS sejam definidas em uma biblioteca padrão como instâncias de algum destes seis elementos primitivos.

[3]Nome dado por existir certa analogia entre o modelo empregado e o envio de sinais em modelos de programação orientada a eventos.

recursos de intercessão não tenham que referir-se aos métodos explicitamente, mas sim através dos sinais, que representam o conjunto de métodos que os declaram.

O conjunto de métodos representados por um sinal é restrito ao escopo do sinal (isto é, um objeto), o que faz com que um sinal `T` declarado dentro de um objeto `A` seja diferente de um sinal `T` declarado em um objeto `B`. Em GLASS, transformações reflexivas são sempre aplicadas a métodos de um objeto específico, ou utilizando a notação `objeto.sinal`, ou inserindo a transformação dentro da declaração do objeto (através de um bloco `new`/`end`). Transformações genéricas, entretanto, podem ser declaradas como uma combinação de declarações reflexivas utilizando um nome de sinal (Seção 3.3). Uma transformação genérica sobre `T` poderá então ser aplicada ao objeto `A` ou `B`.

Operações podem ser especificadas por um objeto não apenas em termos de sinais herdados, mas também em termos de seus próprios atributos. Assim, um conjunto de métodos onde um dado atributo indica um sinal (ou, mais precisamente, onde métodos de um dado atributo que declaram um sinal específico são chamados) podem ser referidos através deste sinal. Por exemplo, não será necessário declarar um sinal na assinatura de todos os métodos que acessam um banco de dados; ao invés disto, os métodos da biblioteca SQL podem declarar sinais, através dos quais métodos que realizam acessos à base de dados podem ser identificados. Sinais tornam-se visíveis a outros objetos apenas se explicitamente exportados, através de uma cláusula `export`. Esta restrição garante que um objeto controle o quanto de seu estado interno pode ser afetado pela manipulação de sinais em objetos derivados ou em objetos que o utilizam como atributo.

Referências. Referências são ponteiros para métodos ou objetos. A palavra reservada `is` descreve a criação de uma nova referência. Ao criar-se uma referência, atribui-se a ela um objeto ou método para o qual ela aponta, seja através de outra referência ou através da criação de um objeto ou método. Uma referência pode ainda ser declarada indicando no comando `is` apenas uma assinatura. Neste caso, a referência tem sua informação de tipagem definida, mas não aponta para objeto algum (ou seja, aponta para *null*).

Referências podem também ser reificadas, isto é, vistas como objeto. Duas ações básicas relativas a referências podem ser, neste contexto, entendidas como métodos: acesso e atribuição. GLASS define, então, dois sinais—`read` e `write`—como gerados pelo comportamento das próprias referências, e não pela declaração de métodos ou objetos para os quais estas referências apontam.

Objetos. Objetos são coleções de referências. A Figura 2 decreve um exemplo de criação de um objeto. A declaração `new` indica a criação de um novo objeto, com o objeto indicado como parâmetro (`Car`) usado como protótipo. A presença de um bloco `as`/`end` indica que um novo objeto é uma versão modificada do protótipo. Neste exemplo, o objeto apontado por `Engine` é uma cópia idêntica de `F99Engine`, o objeto apontado por `Ferrera` é uma cópia modificada de `Car`, e `Paint` aponta para o mesmo objeto que `Red`.

A descrição de métodos como atributos de um objeto é uma aplicação do conceito de reificação. Uma das principais motivações para o uso de reflexão é disponibilizar ao programador, de forma organizada, as estruturas internas e informações relativas aos ambientes de compilação/execução. Em linguagens orientadas a objeto típicas, métodos são "atributos implícitos" dos objetos, implementados, por exemplo, como ponteiros de função em uma tabela de métodos virtuais. Em GLASS, métodos são definidos como atributos explícitos, permitindo ao usuário manipular a tabela virtual de forma segura, utilizando sintaxe de alto nível.

Métodos. Assim como objetos denotam dados, métodos denotam ações: méto-

```
Ferrera is new Car as
  Paint is Red
  Engine is new F99Engine
  getName is method (type is NameKind) (String) as
     if type = Short : return "F99"
     else if type = Long : return "Blood-red Ferrera F99"
     end
  end
end
```

Figura 2: Uma declaração de objeto, em GLASS.

dos representam fluxos de execução imperativa. Um método possui entradas (parâmetros) e saída (valor de retorno). A representação do fluxo de execução que um método representa se dá através de uma seqüência de comandos (*statements*). Uma construção `method...as/end` declara um bloco de método. À direita da palavra reservada `method`, segue uma lista de paramêtros de entrada, e opcionalmente, uma lista de parâmetros de saída. Parâmetros são descritos por assinaturas de restringem os tipos de objetos válidos passados nas chamadas de métodos. A definição de um parâmetro de entrada, utilizando a palavra-chave `is`, pode receber uma assinatura ou objeto, uma vez que todo objeto também corresponde a uma assinatura.

A declaração de um método se dá através de um bloco `method`. Ao lado da palavra reservada `method`, segue uma lista de parâmetros formais de entrada e, opcionalmente, uma lista de parâmetros de saída. Os parâmetros são descritos por assinaturas que restringem os tipos dos objetos passados nas chamadas dos métodos. A definição de um parâmetro, usando a palavra reservada `is`, pode receber uma assinatura ou objeto; sendo um objeto, a assinatura implícita deste será usada.

Note que a definição de um método é anônima. A denominação do método se dá apenas quando este é associado a uma referência. Por este motivo, chamadas recursivas devem ser realizadas utilizando a palavra reservada `recurse` onde normalmente seria usado o nome do método. O uso do nome do método pode causar um resultado diferente do esperado se o método for atribuído para uma referência com outro nome e a referência original tiver atribuída a si algum outro método. Tomemos como exemplo um método `M is method () as...end` que em seu corpo possui chamadas para `M()`. Enquanto a referência `M` aponta para este método, ele é, de fato, recursivo. A seqüência de atribuições `X <- M : M <- Y`, entretanto, fará com que ele deixe de o ser. O método é atribuído a `X` e `M` passa a apontar para o mesmo método para o qual `Y` aponta. Ao executar `X()`, este método não entrará em recursão, mas sim invocará o método `Y`. A maneira correta de representar a recursão seria substituindo as chamadas a `M()` por chamadas `recurse()`. De forma similar, referências ao próprio objeto dentro de seu corpo devem utilizar sempre o atributo implícito `self` e não o nome do objeto.

Assinaturas. O sistema de tipos de GLASS é definido de modo que haja separação entre herança e sub-tipagem. Para isto, define-se aqui o conceito de assinatura, uma adaptação do conceito homônimo utilizado em ML [25]. Em [3], apresenta-se uma extensão a C++ que adiciona assinaturas ao modelo de classes da linguagem. Os objetivos almejados nesta adaptação de assinaturas ao modelo baseado em protótipos realizado são: **(1)** permitir a verificação estática do sistema de tipos; **(2)** permitir relações de sub-tipagem independentes da hierarquia de herança do objeto, que, em linguagens baseadas em protótipo, é definida por clonagem; **(3)** permitir um uso de migração de métodos entre objetos seguro e o menos restritivo possível.

Uma assinatura é uma representação puramente abstrata de um objeto ou método.

Existem portanto, dois tipos de assinatura: assinaturas de objetos e assinaturas de métodos. Uma assinatura de um objeto é um conjunto de tuplas $\langle S, N \rangle$ onde S é uma assinatura (de objeto ou método) de um atributo e N é o nome deste atributo. A definição é recursiva: as assinaturas contidas nas tuplas podem ser outras assinaturas definidas da mesma forma ou assinaturas correspondentes a objetos internamente pré-definindos pela linguagem, como `integer`, `rational`, `string`, `array` e `object`. A assinatura de um método é uma lista ordenada composta pelas assinaturas dos seus parâmetros formais de entrada e saída e pela assinatura inferida para o *self* do método.

As relações de tipagem de GLASS são definidas como relações de adaptabilidade entre objetos e assinaturas ou de assinaturas entre si. Este tipo de relação – adaptabilidade – é o ponto-chave do sistema de tipos. A definição de adaptabilidade é definida da seguinte forma. Para assinaturas de objetos, uma assinatura A é adaptável a uma assinatura B se para todas as tuplas $\langle S_B, N_B \rangle$ existentes em B exista em A uma tupla $\langle S_A, N_A \rangle$ tal que os nomes N_A e N_B são iguais e S_A é adaptável ou igual a S_B. Para assinaturas de métodos, uma assinatura A é adaptável a uma assinatura B se A e B são duas listas ordenadas $[S_{A,1}, S_{A,2}, ..., S_{A,n}]$ e $[S_{B,1}, S_{B,2}, ..., S_{B,n}]$ e para qualquer x entre 1 e n, $S_{B,x}$ é adaptável ou igual a $S_{A,x}$. Assinaturas de objetos não são adaptáveis a assinaturas de métodos, e vice-versa.

Cada objeto possui uma assinatura implícita que corresponde ao conjunto das assinaturas e nomes de *todos* os seus atributos. Isto é análogo à construção `sigof(X)` em [3], que retorna a assinatura referente a uma classe C++ qualquer `X`. Ao criar-se um novo objeto, é realizada efetivamente uma cópia de um objeto protótipo: os dois objetos possuem a mesma assinatura implícita. Todavia, dois objetos podem possuir a mesma assinatura implícita mesmo que um não seja criado por cópia do outro.

Uma assinatura é declarada através da construção de bloco `signature of` **...** `as` **...** `end`, que recebe como parâmetro um identificador de assinatura ou objeto, que servem como provedores de uma assinatura protótipo. A partir daí, novos atributos podem ser adicionadas através de declarações `is` similarmente a declarações de referências em objetos, ou removidos utilizando o comando `hide`. É importante notar que, ao utilizar `hide` para remover um atributo de uma assinatura, esta deixa de ser adaptável à assinatura protótipo.

Restrições de tipo podem ser acrescentadas utilizando um comando como `constrain R as TipoNovo`. Através dele, a informação de tipo de uma referência pode ser "especializada", de modo que a referência passará a aceitar apenas uma subconjunto dos objetos que aceitava anteriormente. No exemplo, a referência `R`, que aceitava qualquer objeto com assinatura adaptável à de, digamos, `TipoAntigo`, agora aceita apenas objetos com assinatura adaptável à de `TipoNovo`. A restrição para a realização de um `constrain` é que `TipoNovo` deve ser uma assinatura adaptável à de `TipoAntigo`. Esta restrição garante que as assinaturas dos métodos que eram adaptáveis à assinatura antiga serão adaptáveis à assinatura nova. Polimorfismo paramétrico similar a *templates* em C++ pode ser obtido especificando os parâmetros dos métodos de um objeto utilizando como tipo um atributo do próprio objeto, uma vez que é possível restringir o conjunto de valores aceitos utilizando `constrain`.

A construção `signature of`, sem um bloco associado, realiza a explicitação da assinatura implícita que descreve os tipos de todas as referências de um dado objeto. Assim, o comando `X is Y` define uma nova referência `X` que possui assinatura igual à de `Y` e aponta para `Y`; o comando `X is signature of Y` define uma nova referência `X` que possui assinatura igual à de `Y` mas não aponta para objeto algum. A palavra reservada `signature` permite a descrição da assinatura de um método utilizando uma sintaxe

similar à de uma construção `method` sem um bloco de código.

Ao declarar-se um método, os tipos dos parâmetros formais de entrada e saída são explicitamente especificados através de assinaturas ou objetos (que também correspondem a assinaturas). Um tipo restante é definido implicitamente: o tipo de *self*. Na abordagem tradicionalmente usada em linguagens orientadas a objeto, o tipo de *self* é idêntico ao objeto ou classe onde o método está contido. Como na linguagem proposta um método pode efetivamente migrar de um objeto para outro, definiremos aqui *objeto-hospedeiro* como sendo o objeto no qual o método está contido no estado inicial do programa. Assim, definiremos o tipo de *self* como uma assinatura anônima que contém todos os atributos e métodos do objeto-hospedeiro que são usados por este método.

A sintaxe de criação "`H is method`..." garante que um método sempre está associado a alguma referência `H`, que por sua vez sempre faz parte de algum objeto `OBJ`. Este `OBJ`, enquanto objeto-hospedeiro do método referenciado por `H`, deve ter uma assinatura adaptável à assinatura anônima inferida do *self* do método. Isto impede que uma referência esteja apontando para um método que não possa ser invocado, uma vez que este objeto é o responsável pelo contexto de escopo da execução do método. Isto também garante a possibilidade de se realizar esta verificação em tempo de compilação, pois se um método pudesse ser definido sem correspondência com nenhum objeto, qualquer método ou atributo poderia ser referenciado no seu corpo, pois seriam válidos como métodos ou atributos da assinatura anônima de *self*.

Através deste mecanismo que permite atribuição de métodos, começa-se a diluir a linha entre herança e composição em relação a reaproveitamento de código, pois uma vez que os métodos podem ser atribuídos a qualquer referência à qual eles sejam adaptáveis, pode-se herdar código fazendo a composição de métodos de diferentes objetos em um novo objeto sem que estes objetos sejam atributos do objeto novo, como normalmente ocorre com composição.

3.2 Visibilidade

Realizar o controle de visibilidade de métodos e atributos através de assinaturas é bem mais flexível do que através de níveis definidos estaticamente para cada objeto (como ocorrem com as seções `private`, `public` e `protected` em linguagens como C++, JAVA e OBJECT PASCAL [22], [26]). Em [38] discute-se como é possível controlar visibilidade apenas com objetos que agem como "controladores de *namespace*". Esta abordagem, embora flexível, impõe sobrecusto em tempo de execução. A alternativa proposta aqui é permitir que o usuário componha **pacotes**, coleções arbitrárias de objetos e assinaturas, que podem ser importados em outros contextos[4]. Pacotes são estáticos, definidos em tempo de compilação. Não há uma declaração de bloco específica onde são listados os elementos de um pacote. Ao invés disso, utilizam-se declarações como `export Obj to Pac`, onde o objeto `Obj` é adicionado ao pacote `Pac`. A partir de um contexto qualquer, pode-se tornar os objetos e assinaturas de um pacote `Pac` locais através de `import Pac`.

Em um primeiro momento, pode parecer estranho que as declarações que constituem um pacote (`export`) estejam espalhadas ao longo de diferentes contextos. Todavia, isto permite uma maior flexibilidade na definição dos pacotes permitindo padrões de visibilidade complexos como os vistos na Figura 3[5], sem impor custo extra na exe-

[4]Um contexto pode ser um arquivo, um diretório, uma biblioteca. A definição mantém-se aberta para que a linguagem não seja dependente da disponibilidade de diretórios hierárquicos, ou sequer de um sistema de arquivos. No que concerne à definição da linguagem, o texto do programa pode estar armazenado de qualquer forma, dividido em partes (*contextos*) que, a princípio, não têm acesso umas às outras.

[5]Na abordagem hierárquica (`private`→(...)→`public`), cada nível de visibilidade é um subconjunto

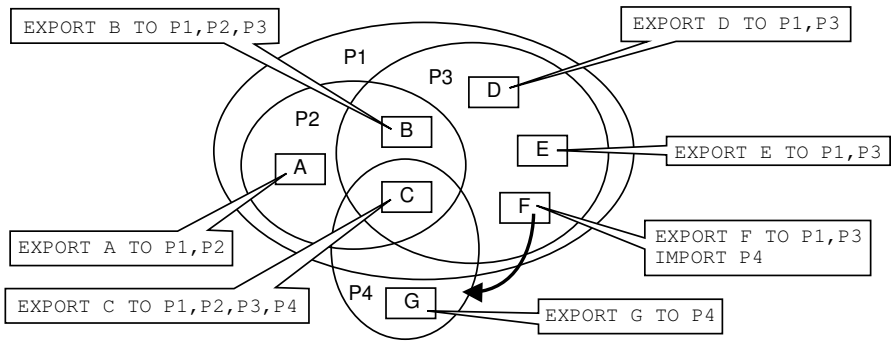


Figura 3: Visibilidade de objetos através de pacotes

cução.

3.3 Recursos reflexivos

Uma vez instrumentado o código dos métodos utilizando sinais, torna-se possível especificar regras de alto nível para realizar intercessão comportamental, isto é, definição de novos objetos através da alteração do código de outros. A seguir, serão apresentadas construções lingüísticas que irão permitir duas abordagens complementares – modificação **intra-método** e **extra-método** – para atingir esta finalidade.

Modificação intra-método consiste em produzir código para um método de um objeto derivado a partir do código alterado do método do objeto pai. Isto é possível pois o código do método no objeto pai é instrumentado pelos sinais: pode-se, então, definir uma regra como "insira o trecho de código `t` em todos os métodos após ocorrências de chamadas que produzam o sinal `S`". Modificação intra-método é realizada utilizando a declaração `when`. Parametriza-se a declaração com um sinal e indica-se em blocos `before` e/ou `after` uma listagem de código seqüencial que deve ser aplicada em métodos onde hajam chamadas que produzam este sinal.

Modificação intra-método, apesar de aparentemente romper com a visão atômica de um método que existe em reflexão comportamental, não deve ser confundida com reflexão lingüística. A instrumentação dentro do corpo de um método limita-se a entradas e saídas de métodos, agindo como um equivalente em tempo de compilação a instrumentar-se o *method dispatcher* que seria o responsável no meta-nível por estas chamadas.

Técnicas de modificação extra-método consistem em, utilizando como critério a ocorrência de sinais, definir regras para substituição ou instrumentação de métodos "cercando" o seu código com chamadas adicionais de forma equivalente aos `before-`, `after-` e `around-methods` de CLOS. A declaração `redefine` possibilita modificação extra-método de um conjunto de métodos. Como parâmetros `where` de `redefine` tem-se um padrão de busca para métodos de modo a indicar a quais métodos aplicar a modificação. Se o parâmetro contiver uma referência a um sinal acompanhado por um atributo, a operação aplica-se a todos os métodos onde ocorre o sinal daquele atributo. Por exemplo, `redefine where list.decrease` refere-se a todos os métodos onde a referência `list` seja invocada de modo a sinalizar `decrease`, ou seja, métodos que invocam métodos que sinalizam `decrease`. A declaração `redefine where decrease`, por outro lado, refere-se a todos os métodos que explicitamente indicam `signals decrease`. Similarmente à declaração `when`, o código a ser instrumentado é listado em blocos `before` e/ou `after`.

Finalmente, adição de nova sintaxe deverá ser possibilitada pela declaração

do nível anterior.

`declaration`, que permitirá a combinação de declarações de forma parametrizável. Isto pode ser considerado um recurso de alto nível para processamento de macros, com raízes na função `defmacro` de LISP. No contexto da declaração `declaration`, referências podem ser reificadas como objetos `handle` e sinais como objetos `signal`, para que estes possam ser usados como parâmetros e empregadas em comandos reflexivos usados dentro do bloco de `declaration`.

O seguinte exemplo demonstra a adição de uma nova declaração, `synchronize`, que por sua vez faz uso da declaração `redefine`. Ao aplicar `synchronize` sobre um objeto `pool` derivado de `stack` (uma pilha que possui um atributo `top`), este passa a possuir acesso mutuamente exclusivo (assume-se que existe, em pacote previamente importado, um objeto `mutex` que implementa a funcionalidade necessária).

```
stack is new object as
  top is node
  push is method (n is node) as
    if top <> null then n.next <- top
    top <- n
  end
  ...outros métodos...
end

synchronize is declaration (a is handle, s is signal) as
  m is new mutex
  redefine where a.s as
    before : m.lock()
    after : m.unlock()
  end
end

pool is new stack as
  synchronize top write
end
```

Note que como as modificações reflexivas aplicadas nos métodos se dão em tempo de compilação, elas se referem ao "estado inicial" dos objetos no programa. Ou seja, mesmo que métodos possam ser movidos a aplicação das declarações é feita em relação aos métodos conforme suas posições originais. Isto deve ser levado em conta ao adicionar código aos métodos de um objeto e depois atribuir a referências destes, métodos de outro objeto. Algo a ser notado também é que o *self* de um método pode ser modificado pelo código adicionado, uma vez que ele pode tornar o método dependente de mais atributos.

Por exemplo, ao utilizar a declaração `SYNCHRONIZE` implementada no exemplo acima, todos os métodos que escrevem em um determinado atributo são cercados por um *mutex* criado para o objeto. Ao atribuir-se um novo método a uma referência durante a execução do programa, este método não possuiria o código de controle do *mutex* criado por `SYNCHRONIZE`, o que poderia ser uma fonte de problemas.

Uma extensão interessante com um impacto bastante significativo seria permitir a inserção de comandos de controle reflexivo como *statements* dentro de métodos, aplicando transformações em outros métodos em tempo de execução. A necessidade de modificação de área de código em tempo de execução para implementação desta técnica, entretanto, é inviável em sistemas modernos, limitando a aplicabilidade desta técnica, ao menos nos dias de hoje, a ambientes de máquinas virtuais. Trata-se de uma limitação provocada pelo ambiente de execução, e não pelo *design* da linguagem.

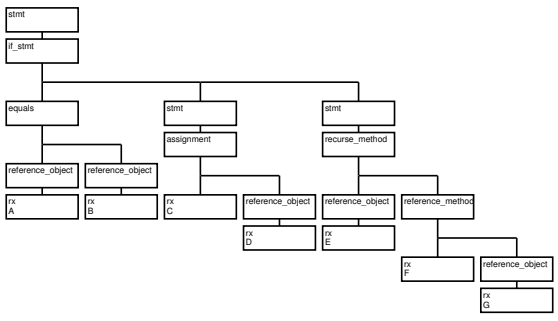
4 Implementação

O tipo de inserção de código necessário para construções como as apresentadas neste trabalho não podem ser realizadas por um macro processador no estilo do pre-processador da linguagem C. A realização correta de manipulação de código deste tipo deve ser feita sobre a árvore sintática abstrata (AST) gerada após o parsing do código-fonte. Na implementação do protótipo, a AST é uma coleção de nodos que correspondem às regras da gramática da sintaxe da linguagem, onde os nodos que serviam finalidade puramente sintática, como `<End>`, foram removidos. A Figura 4 foi gerada utilizando um módulo do compilador protótipo que exporta as suas estruturas internas como um gráfico no formato FIG, posteriormente convertido para POSTSCRIPT.

Uma vez que a AST está em memória, o passo seguinte é processar os comandos reflexivos que geram as novas estruturas que representam o código modificado. O que temos aqui é claramente a aplicação de transformações sobre uma árvore. Uma forma clara e concisa de representar estas transformações é através de *gramáticas de transformação de árvores*, ou TTGs [27]. Uma TTG é uma gramática livre de contexto onde cada produção possui um termo à sua esquerda, e uma ou duas partes à direita, uma parte indicada com uma seta simples que representa a sub-árvore a ser reconhecida, e outra indicada com uma seta dupla, representando a sub-árvore a ser inserida no lugar da que foi reconhecida. Cada comando reflexivo, então, corresponde a um padrão de aplicações de gramáticas transformacionais a ser seguido.

A especificação da semântica dos comandos reflexivos de GLASS foi feita utilizando gramáticas transformacionais de atributos, ou TAGs [27]. Como o nome indica, uma TAG combina uma gramática de transformação de árvores a uma gramática de atributos [19]. Computações são especificadas declarativamente sobre estruturas de dados descritas pelos atributos. Atributos herdados podem ser vistos como dados computados *top-down*, e atributos sintetizados como dados computados *bottom-up* [7].

O comando `when` corresponde a um caminhamento *top-down* através do corpo dos métodos, verificando através de uma assertiva a presença dos métodos localizados em uma tabela de sinais previamente montada e inserindo as sub-árvores referentes aos blocos `before` e/ou `after`. A gramática do comando `redefine` descreve uma procura por chamadas de métodos similar à do comando `when`. Adicionalmente, um atributo sintetizado indica se a procura obteve sucesso. Este atributo retorna, *bottom-up*, à regra inicial, de modo a instrumentar os blocos `before` e/ou `after` no início/final do método. Por questões de espaço, as TAGs completas não estão incluídas neste artigo, mas podem ser encontradas em [ref].




```
if a = b
   c <- d
else
   recurse(e, f(g))
end
```

Figura 4: Um bloco `if` e a árvore sintática abstrata correspondente

5 Conclusão

Este trabalho apresentou técnicas para exploração de reflexão computacional contextualizadas a uma linguagem orientada a objetos com modelo de dados baseado em protótipo. A definição da linguagem apresentada aqui, embora não completa, descreve como prova-de-conceito os aspectos mais relevantes em relação à reflexão computacional e ao modelo de objetos e como estes dois elementos colaboram para um modelo de programação consistente.

Este trabalho propôs um modelo no qual reflexão é utilizada para realizar composição de código utilizando construções para especificar modificações de métodos existentes e especificar instrumentações utilizando sinais, através dos quais são especificados grupos de pontos onde determinada modificação deve ser aplicada, sem interferir com os princípios básicos de encapsulamento do paradigma orientado a objetos. As técnicas propostas são aplicáveis em tempo de compilação, garantindo que não haverá penalização em *runtime* para código composto utilizando construções reflexivas, em oposição a código programado monoliticamente.

O segundo aspecto para o qual se focou este trabalho foi o modelo de objetos. O modelo de objetos sem classes, embora pouco difundido em comparação com o modelo com classes, não é uma idéia nova. A contribuição deste trabalho nesse âmbito foi desenvolver uma adaptação do também conhecido conceito de assinaturas a esse modelo de dados. O uso de assinaturas proporciona um sistema de tipos que permite hieraquias distintas de herança e sub-tipagem. Esta separação se mostrou especialmente interessante porque enquanto o sistema de herança em uma linguagem baseada em protótipos – mecanismo de compartilhamento de código através de clonagem – é bastante aberto, as relações de sub-tipagem tornaram-se estaticamente verificáveis, caracterizando um sistema de tipos seguro.

Numerosas possibilidades se abrem para trabalhos futuros, indo desde o desenvolvimento continuado do compilador protótipo e sua adaptação a *backends* genéricos como C– [28] até a formalização dos conceitos apresentados neste artigo, como a extensão das TAGs em gramáticas de tradução completas.

Uma série de extensões à linguagem podem ser consideradas. Permitir ao usuário manipular as referências em si, e não somente os dados para os quais elas apontam, como objetos abriria possibilidades para que políticas de gerenciamento de objetos pudessem ser especificadas de modo transparente à aplicação, permitindo a comunicação com *garbage collectors*, implementação de *smart pointers* ou mesmo semântica de ponteiros de baixo nível. Associar pré e pós-condições [24] ao uso de um sinal pode garantir que a introdução de novos efeitos colaterais não afetem o funcionamento dos já existentes. A adição de *closures* seria um passo importante no sentido de tornar a linguagem ainda mais ortogonal.

Como GLASS introduz novos tipos de construções, novas possibilidades de design surgem. Uma área interessante de pesquisa seria investigar como estas construções afetariam a implementação de design patterns existentes, possivelmente simplificando ou generalizando-os, e quais novos patterns poderiam ser concebidos, explorando o uso de `declaration` como uma forma de desenvolver padrões reutilizáveis de especialização de objetos.

Referências

[1] Aksit, M. et alli. “Abstracting Object Interactions Using Component Filters”. In *Object-based Distributed Processing*, R. Guerraoui, O. Niestrasz e M. Riveill (Eds.), LCNS, Springer-Verlag, págs. 152-184, 1993.

[2] Backus, J.W. "The syntax and semantics of the proposed international algebraic language of the Zurich-ACM-GAMM Conference." Proc. International Conf. Information Processing, UNESCO, Paris, 1959, R. Oldenbourg, Munich; Butterworth, London, 1960, pp. 125-32.

[3] Baumgartner, G., Russo, V. F. "Signatures: A Language Extension for Improving Type Abstraction and Subtype Polymorphism in C++". In *Software - Practice and Experiences*, Volume 25, Número 8, Ed. Wiley, Agosto 1995.

[4] Brandt, S. "Dynamic Reflection for a Statically Typed Language", Technical Report, Dept. of Computer Science, University of Aarhus, Junho 1996.

[5] Chambers, C. *"The Cecil Language, Specification and Rationale"*, Technical Report 93-03-05, University of Washington, 1993.

[6] Chiba, S. "A Metaobject Protocol for C++". In *OOPSLA '95 Conference Proceedings*. Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '95), ACM. Austin, 1995.

[7] Correnson, L., Duris, E., Parigot, D., Roussel, G. "Declarative Program Transformation: A Deforestation case-study" In *Principles and Practices of Declarative Programming*, Vol. 1702 of Lecture Notes in Computer Science, Springer-Verlag, 1999.

[8] Czarnecki, K., Eisenecker. U. *Template-Metaprogramming*, `http://home.t-online.de/home/Ulrich.Eisenecker/meta.htm`

[9] Czarnecki, K., Eisenecker. U. "Synthesizing Objects", In *Proceedings*, European Conference on Object-Oriented Programming (ECOOP), Portugal, 1999.

[10] Demers, F.-N., Malenfant, J. "Reflection in Logic, Functional and Object-Oriented Programming: a short comparative study" In *Proceedings of the Workshop on Reflection and Meta-Level Architectures and their Applications in AI*, IJCAI'95, pp 29-38, 1995.

[11] Douence, R., Südholt, M. *"The Next 700 Reflective Object-Oriented Languages"*, École des Mines de Nantes, Technical Report, no. 99-1-INFO, 1999.

[12] *"Draft C# Language Specification"*, ECMA TC39/TG2 Working Document, Março 2001.

[13] Graham, P. *"ANSI Common Lisp*." Ed. Prentice-Hall. Upper Saddle River, 1996.

[14] Goldberg, A., Robson, D. *"Smalltalk-80: The Language and Its Implementation."* Addison-Wesley, Reading, MA, 1983.

[15] Gosling, J., et alli. *"The Java Language Specification"*, Java Series. Addison-Wesley, Segunda Edição, Junho 2000.

[16] Kiczales, G., des Rivières, J., Bobrow, D. *"The Art of the Metaobject Protocol*". Ed. MIT. Cambridge, 1991.

[17] Kiczales, G. "Aspect-Oriented Programming". In *Proceedings*, European Conference on Object-Oriented Programming (ECOOP), Finlândia. Springer-Verlag. 1997.

[18] Kiczales, G. "An Overview of AspectJ". In *Proceedings*, European Conference on Object-Oriented Programming (ECOOP), Budapeste, Hungria. 2001.

[19] Knuth, D. "Semantics of context-free languages". *Mathematical Systems Theory Journal*, Vol. 2, 1968.

[20] Kon, F., Costa, F., Blair, G., Campbell, R. "The Case for Reflective Middleware". In *Communications of the ACM*, Volume 45, Número 6. Junho, 2002.

[21] Lim, C.-C., Stolcke, A. *"Sather Language Design and Performance Evaluation"*. Technical Report TR-91-034, Berkeley, California, 1991.

[22] Tesler, L. *"Object Pascal Report"*. Apple Computer, 1985.

[23] Matsuoka, S. *"Language Features for Re-use and Extensibility in Concurrent Object-Oriented Programming"*. Tese de Doutorado, The University of Tokyo, Junho 1993.

[24] Meyer, B. *"Object-Oriented Software Construction"*, 2a. Edição, Prentice Hall, 2000.

[25] Milner, R., et alli. *"The Definition of Standard ML"*, MIT Press, 1997.

[26] "*Object-Oriented Extensions to Pascal*", Technical Committee X3J9, Programming Language Pascal, Setembro 1993.

[27] Pittman, T., Peters, J. *"The Art Of Compiler Design: Theory And Practice"*, Prentice Hall, Upper Saddle River, 1992.

[28] Ramsey, N., Peyton-Jones, S. "A Single Intermediate Language That Supports Multiple Implementations of Exceptions". In *ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (PLDI '00), Junho 2000.

[29] Rivard, F. "A New Smalltalk Kernel Allowing Both Explicit and Implicit Metaclass Programming", In OOPSLA '96 Workshop: *"Extending the Smalltalk Language"*, San Jose, California, Outubro 1996.

[30] Shalit, A. et alli. *"The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language"*, Addison-Wesley Pub. Co., Setembro 1996.

[31] Shivers, O. "Supporting Dynamic Languages on the Java Virtual Machine." In *Proceedings of the Dynamic Objects Workshop*, Maio 1996, Boston.

[32] Smith, B. C. *"Reflection and Semantics in a Procedural Language"* Technical Report 272, MIT Laboratory of Computer Science, 1982.

[33] Smith, B. C. "Reflection and Semantics in Lisp". In *Proceedings of ACM POPL '84*. ACM Symposium on Principles of Programming Languages, 1984.

[34] Smith, W. R. "Using a Prototype-based Language for User Interface: The Newton Project's Experience" In *OOPSLA '95 Conference Proceedings*. Publicado em SIGPLAN Notices, 30, 10 (1995).

[35] Stroustrup, B. *"The C++ Programming Language"*, Addison-Wesley Pub. Co., Julho, 1997.

[36] *"Qt 3.0 Whitepaper"*, Technical Report, Trolltech AS, Oslo, 2002.

[37] Ungar, D., Smith. R. B. "Self: The Power of Simplicity". In *OOPSLA '87 Conference Proceedings*. Publicado em SIGPLAN Notices, 22, 12 (1987) 227-241.

[38] Ungar, D. et alli. "Organizing Programs Without Classes" In *Lisp and Symbolic Computation* 4(3), Kluwer Academic Publishers, Junho, 1991.