Universidade Federal de Minas Gerais

Instituto de Ciências Exatas

Departamento de Ciência da Computação

A Regra da Contra-Variância Guardada

por

Marco Túlio de Oliveira Valente
Roberto da Silva Bigonha

RT 004/95

Caixa Postal, 702

30.161 - Belo Horizonte - MG

4 de abril de 1995

Resumo

Uma questão importante no projeto do sistema de tipos de uma linguagem orientada por objetos é determinar como a assinatura de um método pode ser redefinida em uma subclasse. Existem três possibilidades, chamadas de Regras da Contra-Variância, da Invariância ou da Covariância. A regra da contra-variância, adotada em Sather, é a que garante a correção do sistema de tipos. Já covariância, adotada em Eiffel, apesar de mais flexível, permite a formação de expressões com erros de tipo de difícil detecção. Para solucionar esse impasse contra-variância versus covariância, propõe-se uma solução baseada em contra-variância e verificação dinâmica de tipos, no estilo de Oberon-2.

Abstract

An important question on the design of a type system for object oriented languages is how the signature of a method can be redefined in a subclass. There are three possibilities, namely contravariance, invariance or covariance rules. The contravariance rule, as used in Sather, is the one which easily assures the type system correction. The covariance, as used in Eiffel, although more flexible, allows the creation of expressions whose correction with respect to types is very difficult to check. To solve the contravariance versus covariance problem, we suggest a solution based on the use of contravariance and dynamic type verification, in the style of Oberon-2.

Sumário

1 Introdução

Um ponto fundamental no projeto de qualquer linguagem de programação é a definição de seu sistema de tipos. O sistema de tipos de uma linguagem pode ser *fraco* ou *forte* [Mad90]. Um sistema de tipos é fraco quando admite que expressões com erros de tipo possam fazer com que um programa apresente resultados imprevistos. Já em um sistema de tipos forte, todas as expressões são consistidas quanto a seu tipo, não havendo possibilidade de erros de tipos deixarem de ser detectados, seja durante a compilação ou em tempo de execução.

Um sistema de tipos forte pode ser ainda *estático* quando os tipos de todas as suas expressões são conhecidos em tempo de compilação. Toda linguagem estaticamente tipada é fortemente tipada, mas o contrário não. Tipagem estática é sempre desejável quando possível, pois torna os programas mais fáceis de ler e entender, permite que erros de tipos sejam detectados ainda na compilação e possibilita geração de código mais eficiente [CW85].

A maioria das linguagens orientadas por objetos (LOO) é fortemente tipada. Em C++ [Str91], Eiffel [Mey88, Mey92], Oberon [Mos92a, Mos92b] e Sather [Omo94], atribuições da forma `x:= y` são válidas apenas se o tipo de `y` for um subtipo de `x`. Como um subtipo sempre possui todas as operações de seu supertipo, os compiladores dessas linguagens podem verificar a validade de chamadas subsequentes que têm `x` como objeto receptor, mesmo sendo dinâmica a associação entre uma chamada e o seu código.

No entanto, apenas essa regra de validade da atribuição não é suficiente para assegurar que um sistema de tipos é forte. Uma outra questão importante é a flexibilidade com que se pode redefinir a assinatura de um método em suas subclasses. Em geral, essa redefinição pode ocorrer de acordo com três regras: contra-variância, invariância ou covariância (vide seção 2). A regra da covariância, usada em Eiffel, é a mais flexível, mas introduz dificuldades no sistema de tipos da linguagem. Por outro lado, as regras da invariância e da contra-variância, usadas respectivamente em C++ e Sather, apesar de serem de implementação mais simples, são por demais restritivas na modelagem de alguns problemas (seções 3 e 4).

Nesse trabalho, propõe-se uma nova disciplina de tipo implementável em linguagem da classe de C++, sendo mostrado como alterar o seu sistema de tipo de tal forma que ele passe a dispor da flexibilidade oferecida pela regra da covariância usada em Eiffel, mas de

uma forma disciplinada, segura e de fácil implementação.

A modificação da regra de atribuição em linguagens orientadas por objetos para somente permitir atribuições do tipo `x:= y` quando a classe de `y` for uma subclasse de `x`, torna possível a verificação estática de praticamente todas as expressões envolvendo objetos, inclusive no caso de chamada dinâmica de métodos. No entanto, ainda existem problemas quando da redefinição da assinatura de um método em uma subclasse.

2 Contra-variância, Invariância e Covariância

Suponha que uma classe `A` tenha um método `m` com a seguinte assinatura, onde a_1, ..., a_n e r são tipos:

$$m : a_1 \times a_2 \times a_3 \cdots \times a_n \rightarrow r$$

Suponha ainda que `B`, uma subclasse de `A`, redefina m para:

$$m : b_1 \times b_2 \times b_3 \cdots \times b_n \rightarrow s$$

A pergunta que se faz é sobre qual relação deve existir entre a_i e b_i, para todo $i \leq n$, e entre r e s. Existem basicamente três possibilidades ($x \prec y$ ou $y \succ x$ indicam x subtipo (subclasse) de y):

- $b_i \prec a_i$ e $s \prec r$ (regra da covariância)
- $b_i \succ a_i$ e $s \prec r$ (regra da contra-variância)
- $a_i = b_i$ e $s = r$ (regra da invariância)

O nome das regras vem do fato de a redefinição dos argumentos ocorrer no mesmo sentido ou no sentido inverso da hierarquia de classes.

3 Contra-variância x Covariância

Como destacado em [CW85], a regra que preserva a correção do sistema de tipos da linguagem é a da contra-variância (a regra da invariância pode ser vista como um caso especial de covariância). Já a covariância, adotada em Eiffel, embora permita maior flexibilidade,

```
class A                          class B
  feature                          inherit A redefine f
    f (arg: A) is ....             feature
 end -- A                            f (arg: B) is .... -- covariancia
                                 end -- B
class C
   .....
   a1, a2: A;  b1: B;
   !!a2;  !!b1;
   if ... then
      !!a1
   else
      a1:= b1
   end
   a1.f (a2);     -- ERRO se funcao chamada eh a de B (passa-se um
   .....          -- argumento da classe A quando espera-se um B)
end -- C
```

Figura 1: Programa em Eiffel com Erro de Tipo

pode levar a situações de erro de difícil detecção, como mostra Figura 1. Na versão de Eiffel descrita em [Mey88], o erro acima não é está previsto no sistema de tipo da linguagem. Na seção 4.1, será mostrada a solução de Eiffel 3 para este problema.

Por outro lado, a regra da contra-variância, embora mais fácil de ser verificada pelo sistema de consistência de tipo, é por demais restritiva na modelagem de situações reais de programação. Suponha, por exemplo, uma hierarquia de classes $X_n \prec X_{n-1} \prec \cdots \prec X_1$, onde cada classe X_i tem um método `IsEqual`. Provavelmente, esse método teria que ser declarado conforme Figura 2, de forma a permitir o acesso aos componentes dos objetos referenciados para decidir se são iguais ou não, mas essa declaração de `IsEqual` seria rejeitada pela regra da contra-variância.

Outro exemplo a favor da covariância é mostrado na Figura 3, que ilustra com bastante clareza seu poder de expressão[FAQ94]:

Como se vê, o método `eat` de `COW` teve que ser redefinido usando covariância, pois vacas não comem qualquer planta, mas apenas capim.

```
class Xi
   .....
   inherit Xj
   .....
   IsEqual (arg: Xi) is BOOLEAN    -- Covariancia
     -- Testa se o objeto receptor e "arg" sao iguais
   do .....
   end -- IsEqual
   .....
end -- class Xi
```

Figura 2: Comparação de Geometrias em Eiffel

```
class PLANT
  feature .....
end -- PLANT

class GRASS
  inherit PLANT
  feature .....
end -- GRASS

class HERBIVORE
  feature
    eat (food: PLANT) is .....
  .....
end -- HERBIVORE

class COW
  inherit HERBIVORE redefine eat
  feature
    eat (food: GRASS) is ..... -- covariancia
    .....
end -- COW
```

Figura 3: Vacas Comem Capim - I

4 Soluções de Alguns Sistemas de Tipos

Mostra-se nessa seção as soluções adotadas em algumas LOOs para resolver o impasse entre contra-variância e covariância.

4.1 Eiffel 3

A fim de eliminar a falha de segurança causada pelo uso de covariância, a versão 3 de Eiffel [Mey92] propõe algumas modificações no sistema de tipos da linguagem. Nessa versão, continua-se a usar covariância em nome de sua praticidade, mas cria-se um mecanismo para identificar chamadas que violem a validade do sistema.

Introduz-se o conceito de *Conjunto de Classes Dinâmicas*, como sendo o conjunto de todas as possíveis classes a que um objeto pode se referir em tempo de execução. Esse conjunto é determinado durante a compilação do sistema através de um processo iterativo. Iniciando pelos tipos com que as entidades são criadas, esse processo une sucessivamente ao conjunto dinâmico de uma entidade α o conjunto dinâmico corrente de outra entidade β sempre que descobre-se uma atribuição α `:=` β. O processo termina quando deixa de existir incrementos nos conjuntos de classes. Ressalte-se que não é utilizado em nenhum momento técnicas de controle de fluxo.

Supondo o primeiro exemplo da Figura 1, o conjunto de classes dinâmicas das entidades `a1`, `a2` e `b1` seria:

$$a1 : \{A, B\} \quad a2 : \{A\} \quad b1 : \{B\}$$

Através do conjunto dinâmico de `a1` detecta-se que a chamada `a1.f(a2)` não tem validade de sistema.

Um primeiro problema dessa abordagem é a complexidade do algoritmo para determinar o conjunto de classes dinâmicas, que inviabiliza a implementação de um único passo, e, por isto, enseja implementação incorreta em compiladores de Eiffel 3.

Outro problema é a redução no poder de expressão da linguagem, isto é, o compilador de Eiffel 3 rejeita programas logicamente corretos. Por exemplo, uma chamada, perfeitamente correta, da forma `x1.IsEqual(x2)`, quando o tipo de `x1` for uma subclasse própria de `x2`, será rejeitada pelo sistema. O esperado nesse caso seria uma resposta falsa para essa legítima ativação.

4.2 School

A solução adotada em School [Rod93] consiste na separação entre as hierarquias de tipos (especificações) e de classes (implementações). Em School, o tipo de um objeto é a visão que o mundo externo tem dele, isto é, a assinatura de todas as suas operações. Já classes são estruturas que descrevem implementações para os tipos, isto é, as estruturas de dados e o código das operações do tipo do qual a classe pretende ser uma implementação. Com isso, herança de classes é uma herança apenas de implementação, com o objetivo de reuso de código. Já a hierarquia de tipos, deduzida pelo compilador usando-se compatibilidade estrutural, é que define as relações de subtipagem e onde, portanto, adota-se a regra da contra-variância. No entanto, essa solução tem a desvantagem de, às vezes, tornar difícil o reuso do código de uma classe em suas subclasses, devido à inexistência de polimorfismo na hierarquia de classes. Um exemplo desse tipo de dificuldade é mostrado em [Rod93], onde torna-se necessário a introdução de um tipo genérico para garantir o reuso de um método `equal` de uma classe em uma subclasse.

4.3 Sather

Uma solução semelhante à de School é adotada em Sather, onde também existem duas hierarquias. A primeira delas é usada para definir as relações de subtipagem. Nessa hierarquia é que exige-se obediência à regra da contra-variância. Já a hierarquia de classes é usada apenas com o propósito de reuso de código (herança de implementação).

Assim, Sather apresenta as mesmas dificuldades apontadas em School em relação a reúso de código.

4.4 Oberon-2

Solução interessante é a de Oberon-2, onde vale a regra da invariância No entanto, em Oberon-2 objetos levam para tempo de execução informações sobre o seu tipo. Expressões especiais, chamadas de *type tests*, permitem testar o tipo dinâmico de um objeto e asserções chamadas de *type guards* permitem que um objeto seja tratado como um de seus subtipos. Usando essas construções, o método `IsEqual` da hierarquia de classes X_i seria definido conforme Figura 4:

```
PROCEDURE (a: Xi) IsEqual (b: X1): BOOLEAN;    // Invariancia
VAR bxi: Xi;
BEGIN
  IF b IS Xi          (* type test *)
     THEN
       bxi:= b(Xi);   (* type guard *)
       .....          (* Teste se "a" eh igual a "bxi" *)
     ELSE
       RETURN FALSE
     END
END IsEqual;
```

Figura 4: Comparação de Geometrias em Oberon-2

Note que se `a` e `b` da Figura 4 não forem do mesmo tipo, a rotina acima retorna falso.

4.5 C++

No caso da redefinição da assinatura de uma função virtual em uma classe derivada, o que ocorre é que essa função deixa de sobrepor a função virtual da classe base e passa somente a ocultá-la. Em outras palavras, o compilador C++ desconsidera a declaração da função como virtual, podendo também gerar uma advertência [Ell90]. O exemplo da Figura 5 ilustra essa situação.

Segundo João Dantas [Dan94], uma forma de contornar esse problema seria declarar as funções `IsEqual` com um parâmetro formal do tipo `A*` nas duas classes, obedecendo à regra da invariância. Essas funções, no entanto, apenas chamariam funções auxiliares, como mostrado a Figura 6.

Como o argumento de chamada das funções auxiliares é sempre o ponteiro `this`, pode-se afirmar com certeza qual é o tipo em tempo de execução de seus parâmetros formais. Por isso, essas funções é que são usadas para realizar as comparações entre objetos.

Apesar de engenhosa, essa solução tem a desvantagem de exigir um número crescente de funções auxiliares à medida que aumenta o número de classes derivadas. Supondo que existisse mais uma classe `C`, derivada de `B`, seria necessário uma terceira função auxiliar esperando um parâmetro do tipo `C*`.

```
class A {                                  class B: A {
  public:                                    public:
    virtual char IsEqual (A*);                 char IsEqual (B*);
    .....                                      // Nao e' redefinicao
}  // class A                                  .....
                                           } // class B
main {
  B b= new B;
  A* a1= &b;
  A* a2= new A;
  char ok= a1->IsEqual (a2);   // Chama A::IsEqual e nao B::IsEqual
  .....
}
```

Figura 5: Comparação de Geometrias em C++ - I

```
class A {
  public:
    virtual char IsEqual   (A* a)  { a->AuxEqualA (this) }
    virtual char AuxEqualA (A* a); // *this do tipo A, *a do tipo A
    virtual char AuxEqualB (B* b); // *this do tipo A, *b do tipo B
}

class B: A {
  public:
    char IsEqual   (A* a) { return (a-> AuxEqualB (this)) }
    char AuxEqualA (A* a);         // *this do tipo B, *a do tipo A
    char AuxEqualB (B* b);         // *this do tipo B, *b do tipo B
}
```

Figura 6: Comparação de Geometrias em C++ - II

Uma outra solução seria o uso de *type casting* aliado a um controle explícito do tipo dinâmico dos objetos pelo programador. No entanto, essa solução apenas transfere o encargo de verificação de tipos para o programador. A linguagem em si continuaria sendo estaticamente tipada.

5 Contra-Variância Guardada

Como mostrado na seção 4.5, a regra da invariância ou da contra-variância são por demais restritiva, e a regra da covariância, demasiadamente cara para ser implementada. Por isso, propõe-se uma nova abordagem, a qual permite que problemas que envolvam uso de covariância sejam modelados de forma mais simples, sem, no entanto, introduzir falhas de segurança.

Basicamente, propõe-se a **Regra da Contra-Variância Guardada**, como uma nova abordagem para a redefinição de métodos. A implementação desta regra em uma linguagem como C++ implicaria nas seguintes alterações no seu sistema de tipos:

- Uso da regra da contra-variância no lugar da regra da invariância, já que essa última não passa de um subcaso da primeira. Aumenta-se assim o poder de expressão da linguagem.
- Uso de tipos dinâmicos na linha de Oberon-2. Todo ponteiro para objeto nessa extensão de C++ possui um tipo estático, com o qual é declarado, e um tipo dinâmico, isto é, seu tipo efetivo em tempo de execução. O tipo dinâmico é sempre derivado do tipo estático.

Tipos dinâmicos são tratados pelos seguintes mecanismos:

Type Test: expressão que verifica se o tipo dinâmico de um ponteiro `a` é `A` (ou um tipo derivado de `A`), com a seguinte sintaxe: `(a is A)`

Type Guard: asserção de que um ponteiro `a` possui tipo dinâmico `A` (ou derivado de A), com a seguinte sintaxe[1]: `(A) a`. Se o tipo dinâmico de `a` for `A`, a expressão como um

[1]Note que a sintaxe é a mesma de um *type casting*. Na verdade, *type guards* são uma forma segura de *type casting*.

```
class A { .....
  public:
    virtual char IsEqual (A*);   // Teste de igualdade entre A's
  .....
}

class B: A { .....
  public:
    virtual char IsEqual (A*);   // preserva invariancia
  .....
}

virtual char B::IsEqual (A* a) {
  B* b;
  if (a IS B) {                  // tipo dinamico de a e B ?
     b= (B) a;                   // assegura que a e um B
     .....                       // Teste de igualdade entre B's
  } else return 0;               // objetos de tipos diferentes
}
```

Figura 7: Comparação de Geometrias sem Restrições

todo é considerada como sendo do tipo estático `A`. Caso contrário, a asserção falha e o programa é abortado.

Usando essas construções, o exemplo do método `IsEqual` para as duas classes `A` e `B` seria modificado conforme Figura 7.

Já o exemplo das *vacas & herbívoros* da seção 3 poderia ser modelado da forma exibida na Figura 8, onde a verificação de tipo em tempo de execução garante a mesma semântica do programa da Figura 3.

A abordagem proposta tem a desvantagem de retirar da interface da classe a informação que *vacas somente comem capim*. Entretanto, esta perda de informação poderia ser recomposta se a linguagem suportasse uma filosofia de *Programação por Contrato* [Mey88]. Neste caso, a *precondição* do método `eat` poderia deixar claro as restrições desejadas.

```
class Cow: Herbivore { .....
  public:
    virtual void eat (Plant*); .....
}

virtual void Cow::eat (Plant* food) {
  Grass cow_food;
  cow_food= (Grass) food;
  .....
}

main {
  Herbivore *h;
  Cow *c= new Cow;
  Plant *p= new Plant;
  Grass *g= new Grass;
  h= c;
  h.eat (g);    // Chama Cow::eat com sucesso
  h.eat (p);    // Chama Cow::eat (programa eh cancelado). Optando
}               // por covariancia, esse erro nao seria facilmente
                // detectado
```

Figura 8: Vacas Comem Capim - II

6 Conclusões

Nesse trabalho, procurou-se mostrar que covariância torna mais simples a modelagem de diversos problemas do mundo real. Ou, nas palavras de Bertrand Meyer, que ela é "um componente chave no esforço de tornar a construção de *software* orientado por objetos não somente uma idéia teoricamente agradável, mas uma maneira prática de produzir sistemas reais [Mey92]."

No entanto, o uso puro e simples de covariância na redefinição da assinatura de métodos em subclasses, como em Eiffel 2, aumenta a complexidade do sistema de tipo da linguagem, tornando possível a rejeição de programas que apresentem expressões mal formadas quanto a tipos. É possível verificar estes error através de técnicas baseadas em análise de controle de fluxo de execução, mas isso ou reduz o poder de expressão da linguagem ou encarece o processo de compilação.

Justifica-se assim a proposta de usar-se contra-variância mais verificação dinâmica de tipos. Acreditamos que essa combinação, além de ser teoricamente segura, apresenta poder de expressão semelhante à regra da covariância.

O uso de verificação dinâmica de tipos possui, porém, a desvantagem de possibilitar o cancelamento de um programa com erro de tipo, devido, por exemplo, a um erro de projeto ou a algum caminho de execução não percorrido durante a fase de testes. Quanto ao *overhead* de se levar para execução o tipo de um objeto, ele não é significativo. Pode-se usar como identificador de um tipo o endereço de sua tabela de métodos virtuais, uma estrutura já existente em tempo de execução em linguagens orientadas por objetos.

7 Agradecimentos

Agradecemos ao professor Carlos Figueiredo Camarão pela cuidadosa revisão deste relatório e valiosas sugestões.

Referências

[CW85] Cardelli, L. and Wegner, P. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys* 17, (4):471-522, December 1985.

[Dan94] Dantas, João E. R. *Comunicação Pessoal*, DCC/UFMG, Junho 1994.

[Ell90] Ellis, M. e Stroustrup, B. *The annotated C++ reference manual.* Addison-Wesley, 1990.

[FAQ94] Browne, R. *Eiffel: frequently asked questions.* Posted in usenet newsgroup comp.lang.eiffel, May 1994.

[Gol89] Goldberg, A. and Robson, D. *Smalltalk the language.* Addison-Wesley, 1989.

[Mad90] Madsen, O.L., Magnusson, B. & Pedersen, B.M. Strong typing of object-oriented languages revisited. *ECOOP/OOPSLA Proceedings' 90*, October 1990.

[Mos92a] Mössenböck, H. and Wirth, N. *The programming language Oberon-2.* Technical Report, ETH, Zurich, January 1992.

[Mos92b] Mössenböck, H. and Wirth, N. *Object-oriented programming in Oberon-2.* Technical Report, ETH, Zurich, January 1992.

[Mey88] Meyer, B. *Object oriented software construction.* Prentice-Hall, 1988.

[Mey92] Meyer, B. *Eiffel the language.* Prentice-Hall, 1992.

[Omo94] Omohundro, S. *The Sather 1.0 specification.* International Computer Science Institute, Berkeley, 1994.

[Rod93] Rodriguez, N.R., Ierusalimschy, R. & Rangel, J.L. Conciliação de flexibilidade e verificação estática em linguagens orientadas a objetos. *Anais do VII Simpósio Brasileiro de Engenharia de Software*, 1993.

[Str91] Stroustrup, B. *The C++ programming language (2nd edition).* Addison-Wesley, 1991.