Covariância x Contravariância: A Solução de Ita

Marco Túlio de Oliveira Valente
(mtov@dcc.ufmg.br)

Roberto da Silva Bigonha
(bigonha@dcc.ufmg.br)

Universidade Federal de Minas Gerais
Departamento de Ciência da Computação
Caixa Postal 702
30161-970 - Belo Horizonte - MG

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 contravariância, da invariância ou da covariância. A regra da contravariância, 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. Para solucionar esse impasse entre contravariância e covariância, propõe-se uma nova regra, chamada de contravariância guardada, implementada na linguagem Ita. Tal regra defende o uso de verificação dinâmica de tipos ao estilo de Oberon-2.

Abstract

A important question on type system design of 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 who assures the type system correction. The covariance, used in Eiffel, although more flexible, allows the creation of expressions with type errors. To solve the contravariance and covariance problem, we suggest a new rule, named guarded contravariance, implemented in the language Ita. Such rule advocates the use of dynamic type verification in Oberon-2 style.

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*. Um sistema de tipos é fraco quando expressões com erros de tipo podem fazer com que um programa apresente resultados imprevistos. Já em um sistema de tipos fortes, todas as expressões são consistentes quanto a seu tipo, não havendo possibilidade de um programa produzir resultados incorretos devido a erros de tipo.

Um sistema de tipos forte pode ser ainda *estático* quando o tipo de todas as suas expressões é conhecido em tempo de compilação. Ou seja, 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 [Mey92], Oberon-2 [Mos92a] e Sather [Omo94], atribuições da forma `x:= y` são válidas apenas se o tipo de `y` for um subtipo de `x`. Como um subtipo possui todas as operações de seu supertipo, o compilador dessas linguagens pode verificar a validade de chamadas posteriores envolvendo o objeto `x`, mesmo sendo dinâmica a associação entre uma chamada e o seu código. Isso não ocorre, por exemplo, em Smalltalk [Gol89], onde a inexistência de uma disciplina de tipos possibilita que a aplicação de uma mensagem sobre um objeto produza o término do programa com um erro do tipo *message not understood*.

Durante algum tempo, considerou-se que apenas essa modificação da regra de atribuição seria o bastante para manter forte o sistema de tipos de uma LOO. Mais recentemente, no entanto, descobriu-se que a redefinição de parâmetros de métodos em subclasses também pode introduzir falhas de segurança em um sistema de tipos. Em geral, essa redefinição pode ocorrer de acordo com três regras: contravariância, invariância ou covariância (seção 2). A regra da covariância, usada em Eiffel, é a mais flexível, mas introduz algumas brechas no sistema de tipos da linguagem. Por outro lado, as regras da invariância e da contravariância, usadas respectivamente em C++ e Sather, apesar de seguras são por demais restritivas na modelagem de alguns problemas (seções 3 e 4).

Nesse trabalho, propõe-se uma solução para o dilema entre covariância e contravariância baseada no uso de verificação dinâmica de tipos ao estilo de Oberon-2. Essa solução, chamada de regra da contravariância guardada, foi adotada em Ita, uma linguagem de programação desenvolvida no DCC/UFMG [Val96,VB96].

Ita propõe-se a ser uma extensão orientada por objetos de C, a exemplo de C++, mas com uma declarada preocupação de conciliar poder de expressão com elegância, clareza e simplicidade. A fim de fortalecer o seu poder de expressão, Ita suporta a definição de classes, inclusive genéricas e abstratas, programação por contrato, tratamento de exceções e verificação dinâmica de tipos. Visando manter a simplicidade e clareza da linguagem, optou-se por usar semântica de referência, herança simples e significados objetivos para termos como polimorfismo, polissemia e polivalência.

2 Contravariância, Invariância e Covariância

Suponha que uma classe `A` tenha um método `m` com a seguinte assinatura:

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

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 ($x \prec y$ indica 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 contravariância)
- $a_i = b_i$ e $s = r$ (regra da invariância)

O nome das regras vêm do fato da redefinição dos argumentos ocorrer no mesmo sentido ou no sentido inverso da hierarquia de classes. Apesar de não conhecermos uma referência específica que confirme essa afirmativa, acredita-se que tal terminologia foi adotada pela primeira vez por Luca Cardelli (conforme citado em [Mey95]).

3 Contravariância x Covariância

Como destacado em [CW85], a regra que preserva a correção do sistema de tipos de uma linguagem é a da contravariância[1]. Já a covariância, adotada na versão 2 de Eiffel [Mey88], apesar de permitir maior flexibilidade pode levar a situações de erro de difícil detecção, como mostra o exemplo abaixo:

```
class A                          class B
  feature                          inherited A redefine f
    f (arg: A) is ....             feature
 end -- A                            f (arg: B) is .... -- covariancia
                                 end -- B
 class C
   .....
   g (....) is
   local
     a1, a2: A;
     b1: B;
   do
     !!a1;  !!a2;  !!b1;
     a1:= b1
     a1.f (a2);    -- ERRO: funcao chamada e B::f (passa-se
     .....         -- um A quando espera-se um B)
   end -- g
   .....
 end -- C
```

[1] A regra da invariância pode ser vista como um caso especial de contravariância.

Em Eiffel 2, o erro acima passa desapercebido pelo sistema de tipos.

Por outro lado, a regra da contravariância, apesar de teoricamente correta do ponto de vista da verificação estática de tipos, é por demais restritiva na modelagem de situações reais de programação. Suponha, por exemplo, uma hierarquia de classes $X_1 \succ X_2 \succ X_3 \cdots$, onde cada classe X_i tem um método `IsEqual`. Provavelmente, esse método teria que ser declarado da seguinte forma:

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

Essa declaração, no entanto, seria rejeitada pela regra da contravariância.

Um outro exemplo de uma situação do mundo real, bastante discutida em alguns *newsgroups* da Internet, ilustra com bastante clareza as restrições impostas pela obediência irrestrita a essa regra [FAQ94]:

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

```
class GRASS
  inherited PLANT
  feature .....
end -- GRASS
```

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

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

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

4 Soluções de Alguns Sistemas de Tipos

Mostra-se nessa seção as soluções adotadas em algumas LOOs para resolver o impasse entre contravariância 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.

Supondo o primeiro exemplo da seção 3, o conjunto de classes dinâmicas das entidades `a1`, `a2` e `b1` seria:

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

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

O prinicipal problema dessa abordagem é a complexidade do algoritmo para determinar o conjunto de classes dinâmicas. Esse algoritmo exige para sua aplicação conhecimento prévio de todo o sistema, ou seja, sua implementação somente é viável através de um programa aplicado ao sistema depois de pronto. Nesse sentido, ele assemelha-se a um verificador de programas como o *lint* em C.

Prova cabal dessa complexidade, é que nenhum dos compiladores atuais de Eiffel implementa essa solução. E provavelmente ela nunca virá a ser implementada, pois o próprio Meyer já a abandonou e atualmente advoga uma nova regra, denominda por ele de *chamadas CAT* (*catcalls*). Essa regra, descrita na subseção 4.1.1 abaixo, será adotada na segunda edição do livro [Mey88].

4.1.1 A Nova Regra de Eiffel

Em [Mey95], Meyer apresenta a regra que será adotada nas próximas versões de Eiffel. Essa regra baseia-se em duas definições:

Entidade Polimórfica: entidade que faz parte do lado esquerdo de alguma atribuição polimórfica, ou que é um parâmetro formal, ou ainda que é uma função externa.

Chamadas CAT: Chamada de uma rotina CAT (*Change Availability or Type*). Uma rotina é CAT se o seu *status* de exportação ou o tipo de algum de seus argumentos é alterado em uma de suas redefinições.

Com isso, a próxima versão de Eiffel incorporará a seguinte regra de tipo: *catcalls polimórficas são inválidas.*

No primeiro exemplo da seção 3, a chamada `a.f` seria então invalidada por essa regra, pois `a1` é uma entidade polimórfica e `f` uma rotina CAT.

No entanto, consideramos que a restrição imposta sobre o sistema de tipos por essa nova regra é demasiadamente pessimista, implicando portanto em uma redução

do poder de expressão da linguagem. Apenas para ilustar, se no exemplo citado `a1` fosse um parâmetro formal de `g`, a função `f` deixaria de ser aplicável a `a1`, qualquer que fosse o tipo do parâmetro de chamada correspondente.

4.2 Sather

A solução adotada em Sather[Omo94,Szy93] consiste na separação entre as hierarquias de tipos (especificações) e de classes (implementações). A primeira delas é usada para definir as relações de subtipagem. Nessa hierarquia é que exige-se obediência à regra da contravariância. Já a hierarquia de classes é usada apenas com o propósito de reuso de código (herança de implementação).

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.

4.3 Oberon-2

Alternativa interessante é a de Oberon-2 [Mos92a,Mos92b], 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 da seguinte forma:

```
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;
```

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

4.4 C++

Em C++ [Str91,Ell90], 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. O exemplo abaixo ilustra essa situação [Ell90]:

```
class A {
  public:
```

```
class B: A {
  public:
```

```
    virtual char IsEqual (A*);          char IsEqual (B*);
    .....                               // Oculta A::f
}  // 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
}
```

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 abaixo [Dan94]:

```
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
}
```

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*`.

Recentemente, C++ passou a incorporar o conceito de tipo dinâmico, chamado na sua terminologia de RTTI (*Run-Time Type Information*) [Str94]. Com a idéia de RTTI, a modelagem dos métodos IsEqual fica bem mais limpa e natural, sendo bastante similar à solução de Oberon-2.

5 Contravariância Guardada: A Solução de Ita

Como mostrado na seção 4.4, a regra da invariância de C++ é por demais restritiva. Por isso, propõe-se introduzir alguns recursos em seu sistema de tipos a fim de permitir que problemas que envolvam uso de covariância sejam modelados de forma mais simples, sem, no entanto, introduzir falhas de segurança.

Basicamente, propõe-se as seguintes alterações no sistema de tipos de C++:

- Uso da regra da contravariância 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[2]: `(A) a`. Se o tipo dinâmico de `a` for `A`, a expressão como um 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 para:

```
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
}
```

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

Já o exemplo das vacas & herbívoros da seção 3 poderia ser modelado da seguinte forma:

```
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 detectado
```

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 [**?**]."

No entanto, o uso puro e simples de covariância na redefinição da assinatura de métodos em subclasses, como em Eiffel 2, abre brechas no sistema de tipos da linguagem, tornando possível que um programa apresente resultados imprevisíveis devido a expressões mal formadas quanto a tipos. É possível fechar essas brechas de segurança 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.

Daí a proposta de usar-se contravariância 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.

Referências

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

[Cas94] Castagna, G. *Covariance and contravariance: conflict without cause* Technical Report LIENS-94-18, Laboratoire d'Informatique, Ecole Normale Supérieure, Paris, October 1994.

[Dan94] Dantas, João E. R. *Comunicação Pessoal*, 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.

[Mey95] Meyer, B. *Static typing and other misteries of life.* http://www.eiffel.com, dezembro 1995.

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

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

[Str94] Stroustrup, B. *The design and evolution of C++.* Addison-Wesley, 1994.

[Szy93] Szypersky, C. and Omohundro, S. *Engineering a programming language: the type and class system of Sather.* Technical Report TR-93-064, International Computer Science Institute, Berkeley, novembro 1993.

[VBi95] Valente, M.T.O e Bigonha, R.S. *A regra da contravariância guardada.* Relatório Técnico 004/95, DCC/UFMG, abril 1995.

[Val96] Valente, M.T.O. *Projeto e implementação de uma linguagem orientada por objetos para o desenvolvimento sistemático de programas*. Dissertação de Mestrado, DCC/UFMG, dezembro 1995.

[VB96] Valente, M.T.O e Bigonha, R.S. *A Linguagem* Ita. Relatório Técnico 005/96, DCC/UFMG, fevereiro 1996.