Universidade Federal de Minas Gerais

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Uma Linguagem Orientada por Objetos para o Desenvolvimento Sistemático de Programas

por

Marco Túlio de Oliveira Valente
Roberto da Silva Bigonha

RT 022/96

Caixa Postal, 702

30.161 - Belo Horizonte - MG

junho de 1996

Sumário

Resumo

Este artigo apresenta uma extensão de C orientada por objetos, chamada lta. A fim de favorecer a produção de *software* com alto grau de reusabilidade, lta introduz conceitos como os de ocultamento de informação, herança, polimorfismo, polissemia e polivalência. Para implementação de tipos abstratos de dados, existe em lta o conceito de classes, que inclusive podem ser genéricas ou abstratas. Incentiva-se também a produção de *software* correto através de um estilo de programação por contrato. A linguagem propõe ainda uma solução baseada em verificação dinâmica de tipos para a controvérsia sobre o uso de covariância ou contravariância na redefinição de métodos em subclasses. A fim de testar os recursos propostos por lta, foi implementado um compilador para a linguagem.

Abstract

This paper shows an object oriented extension of C, named Ita. To support the production of *software* with high degree of reusability, Ita adds concepts like information hiding, inheritance, polymorphism, polysemantics, and polyvalence. To implement abstract data types, there is the concept of class, which in Ita can also be generic or abstract. The construction of correct software is stimulated by means of the concept of programming by contract. The language still proposes a solution based in dynamic type verification to the problem about the use of covariance or contravariance in the redefinition of methods in subclasses. In order to test such resources proposed by Ita, a compiler for the language has been implemented.

Uma Linguagem Orientada por Objetos para o Desenvolvimento Sistemático de Programas

Marco Túlio de Oliveira Valente
Roberto da Silva Bigonha

1 Introdução

Na década de 80, a área de linguagens de programação foi marcada pelo surgimento das chamadas linguagens de programação orientadas por objetos (LOO). Tais linguagens tinham como objetivo proporcionar um salto de qualidade no processo de desenvolvimento de *software*, enfatizando principalmente a reutilização de código. Reutilização é um desejo antigo nos grandes sistemas de programação, pois possibilita que o custo de um componente de *software* seja amortizado em projetos seguintes. Para reforçar esse desejo, constata-se que parte da atividade de programar consiste essencialmente em adaptar algumas rotinas comuns, como pesquisa, ordenação, alocação, leitura, escrita etc. As LOO pretendem então fornecer meios para que o programador possa reaproveitar, possivelmente modificando ou estendendo, rotinas já existentes.

Atualmente já estão desenvolvidas uma série de LOO, como Smalltalk [**?**], C++ [**?**, **?**], Oberon-2 [**?**], Eiffel [**?**, **?**], Sather [**?**], dentre outras. Dessas linguagens, a que alcançou maior sucesso de mercado foi, sem dúvida, C++, talvez por ser uma extensão da linguagem C [**?**]. No entanto, pesam fortes críticas sobre C++, principalmente quanto ao seu excessivo número de recursos, alguns deles de difícil entendimento e, por isso mesmo, pouco usados. Por isso, alguns temem que a linguagem possa transformar-se em uma espécie de "PL/I dos anos 90".

Por outro lado, ainda na década de 80, surgiram outras linguagens como Eiffel e Oberon-2, mostrando que orientação por objetos não implica necessariamente em linguagens de difícil domínio e com baixa curva de aprendizado. A definição BNF de Oberon-2, por exemplo, ocupa apenas uma página e nem por isso possui poder de expressão menor que suas concorrentes.

Após quase uma década da definição dessas linguagens, seus conceitos já foram devidamente assimilados, podendo-se assim diferenciar aqueles cujo uso demonstrou que são um avanço daqueles cuja importância não é tão destacada. Sendo assim, surge espaço para definição de uma nova LOO, incorporando conceitos clássicos de orientação por objetos propostos na década de 80, com alguns aperfeiçoamentos surgidos nos últimos anos.

1.1 Descrição do Trabalho

Esse trabalho objetivou projetar e implementar uma nova LOO, chamada Ita[1] [?, ?], incorporando o estado da arte em programação orientada por objetos. A fim de preservar a cultura de programação existente, Ita é uma extensão de C, a exemplo de C++, mas sem excesso de recursos e com sintaxe e semântica que favorecem o entendimento e o uso. Os seus conceitos de orientação por objetos foram inspirados em Eiffel e Oberon-2. Nota-se também alguma influência de C++. Não se pode esconder, no entanto, que o desenvolvimento de Ita procura oferecer uma alternativa a essa última como extensão orientada por objetos de C.

A figura **??**, sem pretensão de ser completa, situa Ita no espaço das LOO.

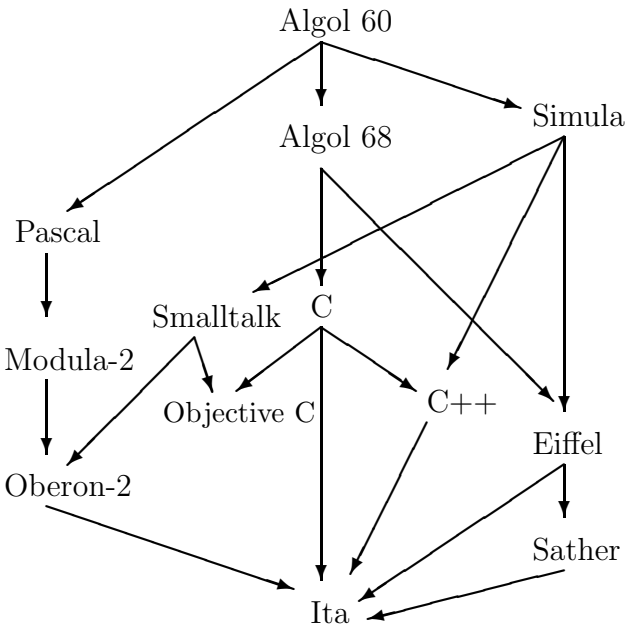

Figura 1: Ita e outras LOO

Como principais recursos de Ita destacam-se os seguintes:

- Suporte a programação orientada por objetos, através dos conceitos de classes (inclusive genéricas e abstratas), herança, polimorfismo, polissemia e polivalência. Além disso, objetos são acessados exclusivamente através de referências.

[1]Uma referência ao Pico do Itacolomi (em tupi-guarani, pedra com filhote) de 1797 metros, localizado na Serra do Espinhaço, em Ouro Preto, MG.

- Suporte a programação por contrato, através dos conceitos de invariantes, pré-condiccões, pós-condições e tratamento de exceções com semântica de retomada.
- Suporte à regra da contravariância guardada para solucionar o impasse entre covariância e contravariância e assegurar que a linguagem é fortemente tipada.

A fim de testar e validar os recursos propostos em Ita, foi implementado um compilador para a mesma, gerando código em C.

A seção **??** desse trabalho apresenta uma definição condensada de Ita, tratando apenas dos principais recursos que ela introduz em C. Simultaneamente, faz-se também uma defesa de seu projeto. Na seção **??**, descreve-se brevemente a implementação do compilador Ita. Por último, a seção **??** faz uma avaliação final da linguagem proposta.

2 A Definição e o Projeto de Ita

2.1 Classes

Em Ita, classes são estruturas usadas para implementar Tipos Abstratos de Dados (TAD). No corpo de uma classe, declaram-se os seus *membros-de-dados* (ou *atributos*) e seus *membros-funções* (ou *métodos*).

Exemplo:

```
class A {                    // Membros privados:
    int a1;                      // atributo
    int f1 (...) { ..... }       // definicao de metodo
    .....
  public:  .....            // Membros publicos:
    int a2;                      // atributo
    init (int p) { ..... }       // metodo inicializador
    finish () { ..... }          // metodo finalizador
    int f2 (...);                // prototipo de metodo
}
int A::f2 (...) { ..... }  // implementacao externa de metodo
```

Os especificadores de acesso `public` e `private` definem o escopo de um método. No caso de atributos públicos, o acesso é apenas para leitura. A fim de privilegiar programação em larga escala, não há o conceito de membros protegidos e considera-se que membros privados o são inclusive para subclasses. Visando flexibilizar de forma controlada essa regra, há o conceito de *classes friends*. Se uma classe `A` diz que outra classe `B` é uma *friend* sua, então membros privados de `A` são acessíveis em `B`.

Objetos são instâncias de classes, consistindo em uma área de memória própria, que armazena o valor corrente de seus atributos. São acessados exclusivamente através de *referências*[2]. A efetiva criação e destruição de objetos é feita através dos operadores `new` e `delete`. Após criar um objeto, o operador `new` chama implicitamente um método especial, de nome `init`, para iniciar sua área de memória. Tal método é chamado de *inicializador* (e não de construtor, pois ele nada constrói). De forma oposta, antes de destruir um objeto, o operador `delete` chama implicitamente um método especial, de nome `finish`. Esse método é denominado de *finalizador* (e não de destruidor).

Exemplo:

```
A a1;             // declara-se "a1" como referencia para "A"
a1= new A (10);   // cria-se objeto; associa objeto a "a1"
a1.f (...);       // utiliza-se "a1" para  acessar objeto
delete a1;        // destroi-se objeto referenciado por "a1"
```

[2] Ita conserva o conceito de *pointer* de C, mas este não guarda relação alguma com referências.

Sobre qualquer referência podem ser realizadas as operações de atribuição (=) e comparação (== e !=). Com semântica de valor, podem ser realizadas as operações de atribuição, duplicação e comparação, através, respectivamente, dos operadores `copy`, `clone` e `equal`.

Considera-se que esse modelo baseado em semântica de referência torna mais natural o projeto de sistemas orientados por objetos. Provavelmente, as suas vantagens ficariam ressaltadas quando associado a um mecanismo de coleta de lixo. No entanto, preferiu-se não incorporar coleta de lixo em lta, pois esse mecanismo não combina com os propósitos de uma linguagem de uso geral, que pode ser usada, por exemplo, em aplicações de tempo real.

Existem duas variantes de classes em lta: *classes genéricas* e *classes abstratas*. Estas variantes são descritas nas duas próximas subseções.

2.1.1 Classes Genéricas

Classes genéricas são classes que possuem tipos como parâmetros. Tais parâmetros (tipos) são chamados de parâmetros genéricos (tipos genéricos) e podem ser de duas formas:

Irrestrito: quando aceita-se qualquer tipo como parâmetro genérico. Nesse caso, apenas operações aplicáveis a qualquer referência podem ser aplicadas sobre referências desse tipo, isto é, operações de atribuição, duplicação e comparação.

Restrito: quando o parâmetro genérico é restringido por um tipo, chamado *tipo restringente*; possíveis parâmetros de chamada deverão ser subtipos do tipo restringente. Com isso, todas as operações do tipo restringente poderão ser aplicadas a referências genéricas.

Exemplo:

```
class A <T1, T2: B> {   // T1: parametro generico irrestrito
  .....                 // T2: parametro generico restrito
  T1 a1;
  T2 f1 (T1 x) { ..... }
  .....
}
```

A classificação de tipos genéricos em restritos e irrestritos permite que verificação de tipos possa ser normalmente aplicada a classes genéricas, o que não ocorre, por exemplo,

com os *templates* de C++. Outra vantagem das classes genéricas de Ita em relação aos *templates* de C++ é que estes têm o seu código expandido a cada instanciação, ao passo que aqueles são implementados com uma semântica de código único. Essa estratégia de código único é viabilizada em Ita porque objetos de tipos genéricos são acessados exclusivamente através de referências.

2.1.2 Classes Abstratas

Classes abstratas são classes que não podem possuir instâncias e, por isso mesmo, podem possuir métodos cuja implementação estará disponível apenas em subclasses, chamados de *métodos postergados* (*deferred*).

Exemplo:

```
abstract class A { .....          // classe abstrata
   deferred void f (int x);       // metodo postergado
   .....
}
```

Classes abstratas são usadas para representar tipos abstratos de dados e como ferramenta de projeto orientado por objeto.

2.2 Programação por Contrato

A fim de estimular a produção de *software* não apenas reusável, mas também correto, Ita suporta a metáfora de *programação por contrato*, preconizada em Eiffel [**?**]. Para especificar "direitos" e "deveres" de implementadores e usuários de classes, são introduzidas asserções chamadas de *invariantes*, *pré-condições* e *pós-condições*.

Invariantes são asserções associadas a classes e que devem ser válidas para todas as instâncias das mesmas. Já pré e pós-condições são asserções associadas a métodos e que devem ser válidas, respectivamente, no início e ao término da execução dos mesmos.

Exemplo:

```
class A {
  int a;
  .....
  int f (int x)
```

```
  pre (x > 0)                                    // pre-condicao
  pos ( (result > 0) && (a == initial (a) + 1)) // pos-condicao
  { ..... }
  rescue { .... retry; .... }      // bloco rescue da funcao
  .....
  invariant: (a >= 0)              // invariante da classe
}
```

Quando uma asserção é violada, produz-se uma exceção. Desvia-se então para o tratador de exceção, chamado em lta de bloco `rescue`, associado à função onde a exceção ocorreu. Exceções em lta são tratadas com uma semântica de retomada. Ou seja, ou a execução de um bloco `rescue` prossegue até o seu final (caso em que a função retorna com a exceção ativada) ou pode-se tentar uma nova execução da função (usando-se, por exemplo, uma outra estratégia). Dentro do espírito de programação por contrato, o que não pode ocorrer é o tratador retornar silenciosamente, sem que o cliente seja notificado, como acontece, por exemplo, em Ada e C++.

2.3 Herança

Herança, em lta, é um mecanismo para construir classes, chamadas de *subclasses*, através da especialização de classes já existentes, chamadas de *superclasses*.

Exemplo:

```
class A {                        class B: A {
  int a1;                          int b1;
  int f1 (int x) { ..... }         int g1 (int x ) { ..... }
public:                          public:
  int a2;                          int b2;
  init (int x) { .... }            init (int x, int y): A (y) ...
  int f2 (int) { ..... }           int g2 (int) { ..... }
} .....                          } .....
```

Uma subclasse herda *todos* os membros públicos ou privados de sua superclasse, com exceção dos métodos inicializadores e finalizadores. No entanto, apenas os membros públicos são acessíveis. No exemplo acima, `B` possui os atributos `a1` (não acessível), `a2`, `b1` e `b2` e os métodos `f1` (não acessível), `f2`, `g1` e `g2`.

Provê-se suporte apenas a herança simples. Poder-se-ia argumentar que com a ausência de herança múltipla, o poder de reutilização da linguagem fica prejudicado. No entanto, entende-se que os problemas que realmente beneficiam-se de herança múltipla não são tão freqüentes a ponto de justicarem o custo extra de sua implementação. Além disso, herança múltipla tem problemas inerentes, como herança repetida e colisão de nomes, que exigem a introdução de regras idiossincráticas na linguagem a fim de serem solucionados. Ressalte-se que outras linguagens recentes, como Ada 95 e Java, também não suportam herança múltipla.

2.3.1 Redefinição de Métodos

Uma polêmica que surgiu recentemente no projeto de linguagens orientadas por objetos diz respeito à redefinição de parâmetros de métodos em subclasses. Suponha, por exemplo, 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:

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

Existem basicamente três possibilidades de relacionamento entre a_i e b_i, para todo $i \leq n$, e entre r e s, para que a redefinição seja válida ($x \prec y$ indica x subtipo (subclasse) de y)[3]:

- $b_i \prec a_i$ 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)

A regra da contravariância, usada em Eiffel, é a mais flexível de todas, permitindo, por exemplo, a modelagem de um método `IsEqual`, com um parâmetro formal X_i, redefinido ao longo de uma hierarquia de classes $X_1 \succ X_2 \succ X_3 \cdots$. No entanto, a adoção pura e simples dessa regra abre uma "brecha" no sistema de tipos da linguagem, como mostrado, usando a sintaxe de Eiffel, no exemplo abaixo [**?**]:

[3]O nome das regras vêm do fato da redefinição dos argumentos ocorrer no mesmo sentido ou no sentido inverso da hierarquia de classes.

```
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
  .....
  a1, a2: A;  b1: B;
  !!a1;  !!a2;  !!b1;
  a1:= b1;
  a1.f (a2);   -- ERRO se funcao chamada e B::f (passa-se
  .....        -- arg. da classe A quando espera-se um B)
end -- C
```

Por outro lado, a regra da contravariância, usada em Sather, apesar de teoricamente segura do ponto de vista de verificação estática de tipos, é por demais restritiva. Ela dificulta, por exemplo, a modelagem dos métodos `IsEqual`. A regra da invariância, usada em C++, pode ser vista como um caso particular de contravariância.

A fim de solucionar o impasse covariância x contravariância, propõe-se em Ita uma nova abordagem, chamada de *Regra da Contravariância Guardada* [**?**], que defende o seguinte:

- Uso de contravariância, por ser a regra que mantém forte o sistema de tipos da linguagem;
- Uso de tipos dinâmicos, ao estilo de Oberon-2 [**?**], com testes-de-tipo (*type tests*) e guardas-de-tipo (*type guards*) para modelar de forma segura problemas que naturalmente demandam emprego de covariância.

Guardas-de-tipo possuem a mesma sintaxe de um *type casting* de C, isto é, em Ita, a semântica de um *type casting* envolvendo referências e classes é dada por uma guarda-de-tipo.

Em Ita, os métodos `IsEqual` seriam implementados assim:

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

```
  .....
}

class B: A { .....
  public:
    char IsEqual (A a) {    // preserva invariancia
      B b;
      if (a is B) {         // teste-de-tipo
         b= (B) a;          // guarda-de-tipo
         .....              // teste de igualdade entre B's
      } else return 0;      // objetos de tipos diferentes
    } .....
}
```

2.4 Polimorfismo, Polissemia e Polivalência

Certamente, polimorfismo é o termo com maior número de interpretações em orientação por objetos. Provavelmente, essa variedade de significados é devida à amplitute da definição normalmente utilizada para o termo, proposta por Cardelli & Wegner [**?**]:

Definição de Cardelli & Wegner : Polimorfismo designa a capacidade de valores e variáveis de uma linguagem possuirem mais de um tipo. Propõem a seguinte classificação para as diferentes formas de polimorfismo:

$$\text{Polimorfismo} \begin{cases} \text{Universal} \begin{cases} \text{Paramétrico} \\ \text{Inclusão} \end{cases} \\ \text{Ad-hoc} \begin{cases} \text{Sobrecarga} \\ \text{Coerção} \end{cases} \end{cases}$$

Em lta, adota-se uma definição menos abrangente para polimorfismo:

Definição de lta: Polimorfismo é a propriedade de referências denotarem em tempo de execução objetos de uma classe e de suas subclasses.

Ou seja, em lta referências é que são polimórficas. Ressalte-se que a definição de lta é mais próxima do significado da palavra que consta dos dicionários.

Para as outras manifestações de "polimorfismo" não enquadradas na definição acima, lta propõe os conceitos de *polissemia* e *polivalência*.

Em Ita, polissemia designa a capacidade de uma função possuir várias implementações, todas elas denotadas pelo mesmo nome. A escolha da implementação a ser executada como resultado de uma chamada é feita avaliando-se nessa ordem:

1. O número e tipo dos parâmetros (polimorfismo *ad-hoc* de sobrecarga, de acordo com [**?**]);
2. A forma do objeto denotado pela referência sobre a qual aplica-se a função;
3. O estado do objeto denotado pela referência sobre a qual aplica-se a função.

A terceira forma de polissemia, chamada de *polissemia por estado*, visa importar para orientação por objetos parte da "inteligência" das máquinas de inferência da programação lógica. Para isso, introduz-se nas classes de Ita o conceito de *estado* para especificar determinados conjuntos de valores de seus atributos e associam-se cláusulas envolvendo tais estados aos seus métodos.

Exemplo:

```
class A {
    int x;
    state s1= x < 30;              // variaveis de estado
    state s2= x > 100;
  public:
    int f (int y) at s1 { ..... }  // polissemia por estado
    int f (int y) at s2 { ..... }
  .....
}
```

Já polivalência designa a capacidade de uma função, com uma única implementação, trabalhar com um conjunto infinito de tipos (respeitada uma certa estrutura hierárquica). Mostra-se abaixo dois exemplos de funções polivalentes:

```
void f (A a)  { ..... }     // A uma classe

void f (T t)  { ..... }     // T um parametro generico
```

2.5 Estrutura de Programa

Os especificadores de acesso `public` e `private` são usados também para determinar o escopo de declarações de um arquivo `.ita`

Exemplo:

```
// arq1.ita
  .....
public:
  .....
private:
  .....
```

A existência de um especificador de acesso para cada declaração permite que o compilador gere automaticamente um arquivo de cabeçalho (extensão `.ih`) para cada arquivo Ita (extensão `.ita`).

3 O Compilador de Ita

A fim de não apenas disponibilizar uma ferramenta para permitir o uso prático da linguagem, mas principalmente de testar e validar o seu projeto, foi um implementado um compilador para Ita. Esse compilador foi desenvolvido em MS-DOS com o auxílio das ferramentas Lex & Yacc e gera código em ANSI C [**?**].

Optou-se por gerar código em C pelas suas características de ser uma espécie de "assembly portável". No entanto, paga-se o preço de o código gerado ter que passar por mais um processo de compilação antes de se tornar de fato executável. Ressalte-se que essa opção por geração de código em C pode dar origem à interpretação de que foi implementado apenas um pré-processador para os recursos que Ita introduz nessa linguagem. Não é esse o caso: o compilador implementado realiza análise léxica, sintática e semântica de todo código fonte, o que elimina, por exemplo, a existência de qualquer erro no código C gerado.

Uma descrição detalhada da implementação desse compilador é realizada em [**?**].

4 Conclusões

Esse artigo descreveu o projeto da linguagem Ita, uma extensão de C orientada por objetos que procura conciliar poder de expressão com segurança, clareza e simplicidade.

Como principais pontos positivos do projeto de Ita, destacam-se os seguintes:

- Adoção de definições coerentes com seu uso para conceitos como polimorfismo, polissemia, polivalência, funções inicializadoras e finalizadoras. Essa nomenclatura clara estimula o uso da linguagem no ensino de orientação por objetos.
- Adoção de verificação dinâmica como solução para o impasse sobre o uso de covariância ou contravariância quando a redefinição de um método em uma subclasse se faz com alteração de sua assinatura.
- Definição do conceito de encapsulamento de forma a privilegiar programação em larga escala.
- Introdução de "inteligência" em classes através da definição de funções polissêmicas por estado.

Quando comparada especificamente a C++, Ita mostra ser uma linguagem mais simples, clara e com poder de expressão senão superior pelo menos equivalente. Além dos quatro pontos relacionados acima, tais qualidades são justificadas pelas seguintes características de seu projeto:

- Implementação de classes genéricas através de uma semântica de código único, o que estimula o reuso das mesmas. Além disso, classes genéricas, em Ita, são passíveis de verificação de tipos.
- Tratamento de objetos através de semântica de referência, o que facilita o projeto de sistemas orientados por objetos.
- Suporte a programação por contrato, o que estimula a produção de *software* correto.

Uma questão que pode suscitar críticas à linguagem é a ausência de herança múltipla. Nessa linha, o principal argumento seria de que o poder de reutilização da linguagem fica reduzido. No entanto, acreditamos que essa redução não é tão significativa a ponto de justificar o custo extra de implementação e o desconforto ocasionado pela introdução de regras pouco intuitivas para solucionar os problemas de colisão de nomes e herança múltipla.

Como trabalhos futuros, destacam-se como mais importantes a padronização de uma biblioteca de classes para a linguagem e o desenvolvimento de extensões com suporte a persistência e programação paralela. Atualmente, encontram-se em desenvolvimento um ambiente para programação visual em Ita e uma ferramenta para auxiliar o projeto de classes em sistemas orientados 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.

[GR89] Goldberg, A. and Robson, D. *Smalltalk the language*. Addison-Wesley, 1989.

[KR88] Kernighan, K. and Ritchie, D.M. *The C programming language (second edition)*. Prentice-Hall, 1988.

[MW92] Mössenböck, H. and Wirth, N. *The programming language 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.

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

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

[VB95] Valente, M.T.O e Bigonha, R.S. *A regra da contravariância guardada.* Relatório Técnico 004/95, DCC/UFMG, abril, 1995.

[Val95] 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.