Conciliação de Flexibilidade e Verificação Estática em Linguagens Orientadas a Objetos

Noemi de La Rocque Rodriguez
NIC
RNP – CNPq
(noemi@inf.puc-rio.br)

Roberto Ierusalimschy
Departamento de Informática
PUC/Rio
(roberto@inf.puc-rio.br)

José Lucas Rangel
Departamento de Informática
PUC/Rio
(rangel@inf.puc-rio.br)

Sumário

O polimorfismo por herança, tal como usualmente apresentado por linguagens orientadas a objetos, introduz um conflito entre flexibilidade e verificação estática de tipos. Este trabalho descreve um sistema de tipos que resolve este conflito, através de facilidades para definição de hierarquias separadas para tipos e implementações e descrição de tipos genéricos com restrições.

Abstract

Polymorphism by inheritance, as usually presented in object oriented languages, raises a conflict between flexibility and static type checking. This paper describes a type system which solves that conflict, through the use of independent hierarchies for types and implementations and generic types with restrictions.

1 Introdução

A flexibilidade de uso e de reuso é certamente uma das características mais marcantes de linguagens orientadas a objetos (LOOs). Em especial, a possibilidade de se programar usando chamadas a métodos cuja identidade real só é conhecida em tempo de execução (*late binding*) é, obviamente, um dos pontos mais interessantes destas linguagens. Entretanto, esta facilidade cria a possibilidade de erros de execução devidos a não existência de um método apropriado para atender a uma chamada. Um sistema de tipos adequado pode restringir as construções de uma linguagem de modo a garantir, em tempo de compilação, a ausência de tais erros. Infelizmente, muitas LOOs têm aberto mão desta segurança em nome da flexibilidade.

Um exemplo extremo desta opção é Smalltalk [9], onde a ausência total de verificação de tipos durante a compilação tem como conseqüência a ocorrência freqüente do erro "*message not understood*" em tempo de execução. Outras linguagens, como Eiffel [17] e C++ [21], apesar de estaticamente tipadas, ainda abrem brechas para a ocorrência de erros análogos. Em C++,

permite-se a conversão dinâmica de um objeto de determinada classe para uma subclasse desta. Em Eiffel, a não observação da "regra de contra-variância" [6] na redefinição de métodos em subclasses é um exemplo de tal brecha. Madsen [16] também sustenta a não observação desta regra, preconizando ao invés o uso de verificação dinâmica de tipos.

Neste trabalho apresentamos a linguagem School, onde se concilia a flexibilidade de programação usualmente atribuída a LOOs com a possibilidade de verificação estática completa. O sistema de tipos de School é baseado em [19], que também ilustra sua flexibilidade, tipando diversos problemas comumente apresentados como justificativas para tipagem dinâmica. A segurança do sistema de tipos de School é demonstrada em [12]. Basicamente, as características de School que permitem a conciliação de tipagem estática com alta flexibilidade são: a existência de hierarquias separadas para especificações e implementações, o uso de compatibilidade estrutural, e a facilidade de tipos genéricos com restrições.

A separação de hierarquias é apresentada na próxima seção. A seção 3 explica o conceito de compatibilidade estrutural em linguagens orientadas a objetos, enquanto o mecanismo de tipos genéricos é discutido na seção 4. A seção 5 analisa a relação de School com outros trabalhos na área. Finalmente, na seção 6 apresentam-se as conclusões.

2 Separação de Hierarquias

Desde o aparecimento do conceito de Tipo Abstrato de Dado (TAD) [10] reconhece-se a importância de separar implementação de especificação. Várias das linguagens criadas no final dos anos 70, como Mesa [7], Modula-2 [22], e Ada [3], permitem separar, por este motivo, a descrição de um tipo em um módulo de interface e um módulo de implementação. Numa LOO, existe mais uma razão para a separação entre especificação e implementação: a construção de hierarquias independentes para tipos e classes.

A nível de especificação, dizer que um tipo é subtipo de outro equivale, simplificadamente, a afirmar que ele oferece pelo menos todas as operações que o outro oferece. Usando este conceito, pode-se desfrutar das vantagens do polimorfismo sem o perigo de erros de execução como o erro "*message not understood*" de Smalltalk, bastando exigir que um valor do tipo A só possa ser usado onde se espera um valor do tipo B quando o tipo A for subtipo de B. Por outro lado, a nível de implementação a herança traz uma outra facilidade de reuso de código. Dizer que uma implementação herda de outra equivale a dizer que ela automaticamente conta com a representação interna (variáveis de instância) e implementações de procedimentos definidas pela outra.

No entanto, nem sempre a hierarquia desejada para as especificações é a mesma desejada para as implementações. Um exemplo fornecido por America em [2] é o do tipo Pilha. Para implementar este tipo pode ser conveniente herdar o código e representação de Array, mas não é desejável que pilha seja um subtipo de Array, dado que nem todas as operações sobre arrays devem poder ser aplicadas sobre pilhas; tipicamente a operação de acesso a um elemento qualquer não deve ser disponível sobre pilhas[1]. Neste exemplo, o que se deseja é apenas o reuso, sem subtipagem. Outro exemplo ilustrativo, apresentado em [13], é a implementação de um tipo Retângulo, tendo Polígono como supertipo. Usando-se herança usual, com compatibilidade de tipos, a operação para introduzir um novo vértice, natural em um polígono, criaria complicações na nova classe.

[1] Muitos autores consideram que este é um uso indevido de herança, e que a relação entre pilha e array contitui um exemplo típico de composição. Esta é uma visão possível, mas também pode ser um vício conseqüente da identificação entre especificação e implementação imposta pela maior parte das LOOs.

Por outro lado, pode-se desejar definir um subtipo de outro sem que exista reuso. Um exemplo disto seria a criação de um tipo *TabelaDeSimbolos*. Este poderia ser um subtipo de um tipo Tabela, dado que teria as operações de busca e inserção oferecidas por tabela, e, como tal, ter valores usados onde programas esperassem valores de tipo Tabela. No entanto, pode acontecer de Tabela ter sido implementada como uma lista encadeada e desejar-se implementar TabelaDeSimbolos através de uma função de hash sobre um array. Neste caso, as variáveis de instância de Tabela seriam herdadas inutilmente, ocupando espaço em cada instância de TabelaDeSimbolos sem jamais serem usadas.

É importante observar que a herança, tanto a nível de implementação como de especificação, é um mecanismo de reuso, mas que os dois níveis representam diferentes formas de reuso. A herança de especificações permite o reuso de código cliente (externo) de um TAD, pois permite que o código aplicável a um determinado TAD seja aplicável a qualquer subtipo deste. A herança de implementações, por outro lado, facilita o reuso de código fornecedor (interno), pois permite que a implementação de um TAD reaproveite uma implementação já existente. Obviamente, em várias situações, como em algoritmos abstratos, as duas hierarquias podem ser isomórficas, mas a identificação completa entre ambas, imposta pela maioria das LOOs, é uma das principais fontes de conflito entre flexibilidade e tipagem estática.

Algumas linguagens orientadas por objetos oferecem um mecanismo de separação parcial entre especificação e implementação. Simula [4], Eiffel, C++ e Tool [20] provêem "classes virtuais" ("*deferred*", segundo o vocabulário de Eiffel). Tais classes podem conter métodos sem código associado, chamados métodos virtuais (em C++ estes são chamados métodos virtuais "puros"). Uma classe virtual não pode ter instâncias, uma vez que uma instância não saberia como reagir a uma mensagem pedindo a execução de um método virtual, mas estabelece um padrão que deve ser satisfeito por todas as suas subclasses. Uma classe virtual pode ser usada para descrever uma especificação, se nenhuma variável de instância ou método não virtual for definido. Neste caso, suas eventuais subclasses podem ser encaradas como implementações alternativas. Observe que classes virtuais servem também para fatorar código comum a várias classes, descrevendo implementações parciais. Por exemplo, uma classe Polígono pode definir um método Desenha que desenha segmentos de reta entre vértices obtidos através de um outro método Vértices. O método Vértices pode ser virtual, precisando ser explicitamente codificado pelas subclasses Triângulo, Retângulo e Pentágono. Pode-se argumentar que classes virtuais fornecem uma maneira de separar implementação de especificação sem onerar o programador com o entendimento e manutenção de hierarquias separadas.

Em uma linguagem com herança simples temos um problema imediato com esta solução. Se usarmos classes virtuais como superclasses das implementações, estas implementações não poderão ter nenhuma outra superclasse. Perde-se assim a possibilidade de reuso de código.

Se a herança for múltipla, é possível usar o mecanismo de classe virtual para modelar a distinção especificação × implementação. Meyer desenvolve o conceito de "casamento por conveniência" [17], onde se usa a herança múltipla para herdar de uma classe (tipicamente virtual) que contribui com a especificação e outra que contribui com a implementação. Nos parece que a disciplina exigida do programador para gerenciar este "casamento" é um peso maior do que o de trabalhar com um conceito a mais. Em primeiro lugar, para que uma classe virtual seja realmente uma especificação, sua definição não pode conter nenhuma variável de instância (sob pena de se amarrar o tipo a uma implementação particular). Em segundo lugar, uma classe virtual só pode ter como superclasse classes virtuais. Por outro lado, ao se definir uma classe não virtual (uma implementação), esta não pode definir nenhum método visível externamente que não tenha sido declarado na superclasse virtual. Se a linguagem der suporte a estas regras,

então o mecanismo de classe virtual deixa de ter utilidade no caso de implementações parciais, e perde-se a vantagem de economia de conceitos. Além disto, neste caso específico a economia de conceitos pode ser perigosa; as relações existentes entre uma classe e suas subclasses e entre uma especificação e suas implementações são de fato distintas, e esta distinção fica pouco clara quando se usa um mesmo mecanismo para tratar as duas.

No caso de C++, vale salientar ainda uma outra facilidade da linguagem para facilitar a separação de hierarquias. Uma classe C_1 pode herdar de outra, C_2, sem tornar visíveis para seus usuários os métodos visíveis para os usuários de C_2 e sem se tornar subtipo de C_2, bastando para isto não declarar a classe herdada como pública. Isto corresponde exatamente ao conceito de herança de implementações. No entanto, a introdução de mecanismos isolados para resolver cada problema colocado parece dificultar o entendimento da linguagem.

School, de modo a resolver estes problemas, separa completamente os conceitos de *tipo* e *classe*. Em School, o tipo de um objeto é sua especificação, ou seja, sua interface com o mundo externo. A declaração de um tipo define todas as operações disponíveis sobre objetos daquele tipo, e o tipo dos parâmetros e resultados destas operações. Entretanto, um tipo não tem nenhum compromisso com nenhuma implementação particular. Como exemplo, a declaração de um tipo `Point` poderia ser:

```
type Point is
  method x () r : Integer
  method y () r : Integer
  method moveBy (p : Point)
end Point
```

Observe que, como um método pode retornar vários resultados, estes recebem nomes, da mesma forma que parâmetros.

Por outro lado, a classe de um objeto descreve sua implementação. Uma classe em School define o formato interno de seus objetos, ou seja, uma estrutura de dados e o código que a manipula. Uma implementação possível do tipo `Point` seria:

```
class PointCart : Point is
  var xPos,yPos : Integer;    -- variaveis de instancia
  method x () r : Integer is r := xPos end
  method y () r : Integer is r := yPos end
  method moveBy (p : Point) is
    xPos := xPos + p.x;
    yPos := yPos + p.y
  end
end PointCart
```

Observe que o *tipo externo* dos objetos da classe `PointCart`, `Point`, é declarado após o nome da classe. De maneira geral, um mesmo tipo pode ter várias implementações. Apenas o tipo é dado em declarações de variáveis, parâmetros, etc. Uma classe particular precisa ser indicada somente na construção de um novo objeto[2].

A separação entre especificações e implementações em School permite que as respectivas hierarquias sejam definidas de maneiras distintas. O objetivo da hierarquia de implementações, como dito acima, é o reaproveitamento de código "fornecedor". Assim, é essencial que esta

[2]Observe que School adota *semântica referencial*, isto é, variáveis contêm apenas referências para objetos, e nunca os próprios objetos. Portanto, a declaração de uma variável não implica a criação de um novo objeto.

hieraquia seja definida por um mecanismo de herança explícita, onde se nomeie as classes a serem reutilizadas; este mecanismo será explicado mais adiante. Na hierarquia de tipos, por outro lado, o objetivo é garantir a validade do código "cliente". Neste caso, pode-se deixar ao compilador a tarefa de deduzir a relação de subtipo, utilizando-se compatibilidade estrutural.

3 Compatibilidade Estrutural e Herança

Normalmente, em LOOs, um tipo é considerado subtipo de outro apenas quando ele é assim declarado, diretamente ou por transitividade. Em linguagens com independência entre as hierarquias de tipos e de classes é possível se adotar *compatibilidade estrutural*. Neste caso um tipo é subtipo de outro se existe alguma forma de compatibilidade entre eles.

Em School, um tipo `A` é subtipo de outro tipo `B` sempre que não houver uma razão em contrário. De maneira simplificada, `A` é considerado subtipo de `B` se `A` oferecer, pelo menos, todos os métodos que `B` oferece. Mais precisamente, tendo em mente nosso objetivo de permitir verificação estática completa acoplada com o máximo de flexibilidade, desejamos permitir que um tipo `A` seja considerado subtipo de outro tipo `B` sempre que não exista possibilidade do uso de um objeto do tipo `A` no lugar de um objeto do tipo `B` causar um erro de execução. Uma vez que erros ocorrem quando uma mensagem é enviada para um objeto que não possui um método para tratá-la, podemos evitá-los com a seguinte definição. Um tipo A é subtipo de B ($A \prec B$) se e somente se, para cada método X em B com aridade $P_{B_1} \times \ldots \times P_{B_n} \rightarrow R_{B_1} \times \ldots \times R_{B_m}$, existe um método X em A, com aridade $P_{A_1} \times \ldots \times P_{A_n} \rightarrow R_{A_1} \times \ldots \times R_{A_m}$, onde para todo $i \leq m$, $R_{A_i} \prec R_{B_i}$ e, para todo $i \leq n$, $P_{B_i} \prec P_{A_i}$. A aparente inversão na última condição é conhecida como "regra da contra-variância" [6], e é necessária para garantir a correção do sistema de tipos [12]. Vale a pena notar que esta definição não apenas é suficiente como também necessária. Em qualquer linguagem que aceite A no lugar de B com $A \not\prec B$ é possível se escrever uma rotina que gere um erro de execução.

Como exemplo de subtipo, considere as declarações:

```
type Point1 is
  method x () r : Integer
  method y () r : Integer
  method moveBy (p : Point1)
  method copy () p : Point1
end Point1

type Point2 is
  method x () r : Integer
  method y () r : Integer
  method moveBy (p : Point1)
  method copy () p : Point2
  method dist (p : Point2) : Integer
end Point2
```

De acordo com nossa definição, `Point2` $\prec$ `Point1`, mas `Point1` $\not\prec$ `Point` (parâmetro incompatível no método `moveBy`) e `Point` $\not\prec$ `Point1` (falta o método `copy`).

A principal vantagem oferecida por compatibilidade estrutural é sua flexibilidade. Por exemplo, se uma nova biblioteca declara um novo tipo, todos os tipos já existentes compatíveis com

este novo tipo são automaticamente considerados subtipos deste, sem necessidade de alterações na hierarquia. Em [13], os autores apontam para a falta de flexibilidade oferecida por hierarquias convencionais, e sugerem o uso de "*fine-grain inheritance*" como forma de contornar este problema. Com compatibilidade estrutural, a simples declaração de tipos intermediários aumenta a granularidade da hierarquia, sem ser necessário uma granularização exagerada nem a declaração explícita de hierarquias sobrecarregadas.

Herança de Classes

Ao contrário da hierarquia de tipos, a hierarquia de classes é construída por declarações explícitas. Como usual em LOOs, uma classe declarada como herdeira de outras dispõe, automaticamente, de todas as variáveis de instância e código de métodos destas. Diferentemente de outras LOOs, em School o tipo dado a esta classe não precisa ser subtipo dos tipos de suas superclasses.

Como já observado, a definição de uma classe especifica um *tipo externo* para a classe. Este tipo define a interface dos objetos desta classe; objetos criados com determinada classe têm este tipo, e portanto só podem receber mensagens referentes a métodos declarados nele. Métodos definidos em uma classe e não presentes em seu tipo externo são considerados como métodos privados da classe, e só podem ser chamados de dentro da classe ou por outros métodos aos quais a classe ortoga este direito; veja discussão sobre *self* mais adiante. Podemos também definir um tipo *interno* para uma classe, denotado aqui por `typeOf("nome-classe")`. Este tipo é formado por todos os métodos desta classe, independentemente de sua visibilidade externa e de serem definidos localmente ou herdados. De modo a garantir que uma classe `C` implemente todos os métodos que ela exporta, deve ser verificado que `typeOf(`C`)` $\prec$ `T`, onde `T` é o tipo externo da classe `C`.

A maior parte das LOOs oferece uma pseudo-variável que permite que o código de um método se refira ao próprio objeto que está executando o método. Esta pseudo-variável é chamada "self" em Smalltalk, "current" em Eiffel, "this" em C++; em School ela também é chamada de *self*. Em uma linguagem com separação de hierarquias, o tipo de *self* é uma questão delicada. Para garantir a correção de uma classe, é necessário verificar a correção do código definido por ela e também garantir que o código herdado de outras classes continua correto visto no contexto desta classe. As variáveis manipuladas no código de um método podem ser parâmetros deste método, variáveis de instância da classe, variáveis locais do método, ou a pseudo-variável em self. Em relação aos parâmetros e às variáveis locais, nada muda quando um método é herdado, pois seu tipo permanece inalterado. As variáveis de instância de qualquer super-classe são também variáveis de instância da nova classe, com o mesmo tipo de antes, donde qualquer manipulação destas variáveis permanece correta. O caso mais complexo é justamente o da pseudo-variável *self*.

Ao discutirmos o tipo de *self* precisamos levar em conta alguns pontos. Em primeiro lugar, todos os métodos que possam ser invocados sobre *self* devem certamente ser oferecidos pela classe, pois não haveria aonde ir buscar a implementação de outros métodos, dado um objeto qualquer desta classe. Isto significa que a assinatura da classe deve incluir a assinatura de *self*. Em segundo lugar, para que as chamadas a métodos permaneçam válidas ao se herdar uma classe, é necessário que o tipo de *self* nesta subimplementação seja um subtipo de *self* na super-classe. Finalmente, o tipo de *self* deve ser consistente com seu uso como parâmetro para outros métodos.

Em algumas linguagens com separação de hierarquias (p.e. [11]), o tipo atribuído a *self* é o

tipo externo da classe. Com esta escolha, para garantir tipagem correta, é necessário colocar a restrição de que uma classe só pode ser subclasse de outras que implementem supertipos do tipo que ela implementa [8]. Isto faz com que a separação de hierarquias fique incompleta, pois neste caso é possível herdar uma especificação sem herdar a implementação, mas não o contrário.

A solução adotada em School é atribuir a *self* o tipo interno da classe. Sob o aspecto do uso de *self* como receptor de mensagens, esta escolha é bastante intuitiva, pois define que os métodos válidos nestas mensagens são extamente aqueles definidos na classe. Sob o ponto de vista de *self* como parâmetro de outros métodos, estamos, com esta escolha, permitindo que o método chamado tenha acesso aos métodos públicos (o tipo da classe é sempre um subtipo do tipo que ela implementa) e não públicos. Poderia se argumentar que isto implica em uma quebra de abstração. No entanto, é a própria implementação que está outorgando a outros métodos o direito de enxergar *self* como construído por ela, o que parece um mecanismo bastante útil para controle de visibilidade, como discutiremos a seguir.

Muitas LOOs reconhecem a necessidade de um controle de visibilidade explícito. Em Eiffel as classes têm listas de exportação nomeando as operações visíveis externamente. Em C++, cada membro de uma classe pode ser declarado como "*public*", "*private*", ou "*protected*". Em School, conforme já comentado, a separação entre tipos e classes é responsável pelo controle de visibilidade; qualquer método presente em uma classe e não presente em seu tipo não é visível externamente. A distinção entre *private* e *protected* não é provida em School. A classe herdeira controla o que ela precisa da superclasse, usando uma facilidade de renomeação quando necessário (vide [18]).

Algumas linguagens oferecem um controle de visibilidade ainda mais preciso. C++ oferece o mecanismo de classes ou funções "*friend*", através do qual uma classe permite que outra classe ou função tenha acesso aos seus membros privados. Em School, pode-se controlar visibilidade neste nível através do uso da pseudo-variável *self*. O exemplo abaixo ilustra este ponto.

```
type A is
  method M1 ()
end A

type A1 is
  method M2 ()
end A1

type B2 is
  method M (a : A1)
end B2

Class CA : A is
var b : B2
  method M1 () is ... end
  method M2 () is ... end
  method M3 () is b.M(self) end
end CA
```

Como o tipo de *self* é o tipo interno da classe, a variável `self` da classe `CA` é compatível com o tipo `A1`. Assim, a chamada `b.M(self)` outorga ao método *M* acesso ao método (privado) `M2`.

Em [19] discute-se, em particular, o uso desta facilidade de controle de visibilidade na construção de iteradores sobre tipos abstratos.

4 Tipos Genéricos com Restrições

O polimorfismo visto até agora, resultante da noção de uma hierarquia de tipos, pode ser chamado de *polimorfismo vertical*. Esta seção discute tipos e classes genéricos, o mecanismo de School para suporte a *polimorfismo horizontal*.

Esta facilidade de School é razoavelmente semelhante à oferecida por outras linguagens, como Ada, CLU, e Eiffel. Qualquer tipo pode ser parametrizado por outros tipos. Para citar o exemplo clássico, uma pilha genérica teria a seguinte declaração:

```
type Stack[T] is
  method push (v : T)
  method pop () v : T
  method isEmpty () e : Boolean
end Stack
```

Quando um tipo genérico é usado em uma declaração, parâmetros reais devem ser fornecidos. Para declarar uma pilha de inteiros e uma pilha de tais pilhas, poderíamos escrever:

```
var si : Stack[Integer] ,
    ssi : Stack[Stack[Integer]]
```

Ao contrário do que poderia parecer a primeira vista, não existe relação de subtipo entre tipos genéricos, mas apenas entre suas instâncias concretas. Como exemplo, para saber se `Stack[Point]` $\prec$ `Stack[Point2]`, devemos tomar a assinatura de `Stack[Point]`, que é a assinatura de `Stack` com `T` substituído textualmente por `Point`, e compará-la com a assinatura de `Stack[Point2]`, obtida de forma análoga. Esta abordagem pode ser contrastada com a de Eiffel, onde `X` $\prec$ `Y` sempre implica em `A[X]` $\prec$ `A[Y]`, qualquer que seja a definição de `A`. A solução de Eiffel pode levar a erros de execução, como ilustrado no exemplo abaixo.

```
var p : Point , p2 : Point2 ,
   sp : Stack[Point] , sp2 : Stack[Point2]
...
sp := sp2 ;          -- permitido em Eiffel, ilegal em School.
sp.push(p) ;
sp2.pop().copy() ;   -- pode causar um erro de execucao, pois
                     -- "p" pode nao ter uma operacao "copy".
```

Classes, assim como tipos, também podem ser parametrizadas. Quando uma classe pode ser instanciada com qualquer parâmetro real, temos a chamada *generalidade irrestrita*. No entanto, muitas vezes uma implementação precisa aplicar operações sobre um objeto com tipo genérico. Como exemplo, conjuntos e *bags* precisam de uma operação de igualdade entre seus elementos (genéricos). Em tais casos, uma classe não funcionará para tipos que não tenham estas operações, fazendo com que a linguagem precise de um mecanismo para expressar esta restrição. É interessante observar que, com exceção de estruturas seqüenciais como listas e pilhas, a maior parte das classes genéricas úteis precisam de restrições.

Para expressar uma restrição sobre um parâmetro de uma classe ou tipo, o programador dispõe em School da cláusula *where*. Nesta cláusula são colocadas as operações que um tipo

deve ter para ser um parâmetro real válido, e a aridade destas operações. O próprio parâmetro formal pode ser usado dentro destas aridades, contribuindo para a flexibilidade do mecanismo. Assim, uma classe que implemente conjuntos pode começar com:

```
class SetImp[T] : Set[T] is
  where T has method equal (v : T) r : Boolean end
is
  ...
```

A definição acima afirma que um tipo, para ser um parâmetro real válido da classe `SetImp`, deve possuir uma operação `equal`, que receba como parâmetro um objeto deste mesmo tipo e retorne um booleano. Dizemos que um tipo que satisfaz uma cláusula **`where`** se *conforma* a ela.

Vale a pena frisar a diferença entre tipos conformes e subtipos. Considere um tipo `T1` definido de maneira similar ao **`where`** acima:

```
type T1 is method equal (v : T1) r : Boolean end T1
```

e um tipo `String`:

```
type String is
  method equal (v : String) r : Boolean
  ...
end String
```

Neste exemplo, `String` se conforma a `T`, mas não é um subtipo de `T1`[3]. Na verdade, não existe nenhum tipo que seja supertipo de todos os tipos que se conformam a `T`. Entretanto, podemos simular conformidade usando subtipos e generalidade. Se declararmos um tipo `T2` como:

```
Type T2[T] is method equal (v : T) r : Boolean end T2
```

temos que um tipo `X` se conforma a `T2` se e somente se $X \prec$ `T2[X]`. Esta correspondência é usada em School como a definição formal de conformidade.

Um uso importante do mecanismo de tipos genéricos diz respeito à regra de contra-variância para parâmetros de métodos. Considere o seguinte exemplo:

```
type Point is
  method equal (p : Point) b : Boolean
  method X () x : Integer
  method Y () y : Integer
end Point

type ColoredPoint is
  method equal (p : ColoredPoint) b : Boolean
  method X () x : Integer
  method Y () y : Integer
  method color () c : Color
end ColoredPoint
```

No sistema de tipos de School, `ColoredPoint` não é um subtipo de `Point` (devido à regra de contra-variância). Se fosse, o trecho de programa abaixo seria aceito pelo compilador, quebrando a segurança da linguagem.

[3]Por causa da contravariância do parâmetro de **`equal`**.

```
var p1,p2 : Point;
    cp : ColoredPoint;
    ...
    p1 := cp;  -- assumed valid
    If p1.equal(p2) then ...
```

O problema reside no fato do método `equal`, definido em `ColoredPoint`, ser chamado com um parâmetro real do tipo `Point`. Este método pode aplicar a operação `color` a seu parâmetro, resultando em um erro de execução.

Em algumas LOOs, `ColoredPoint` poderia ser considerado um subtipo de `Point`, em nome da flexibilidade de programação. Isto é permitido ou através do enfraquecimento do sistema de tipos ou através da introdução de verificação dinâmica. Um exemplo do primeiro caso é Eiffel, enquanto o segundo tratamento é adotado em Beta [16]. A vantagem que se obtém com esta abordagem é a possibilidade de se reusar o código de `Point` na implementação de `ColoredPoint`. Em School, o mesmo resultado pode ser obtido através do uso de tipos genéricos. Considere o código abaixo:

```
type PointAux[t] is
  method equal (p : t) b : Boolean
  method X () x : Integer
  method Y () y : Integer
end PointAux

class ImpPointAux[t] : PointAux[t] is
  -- implementation of Points
end ImpPointAux

class ImpPoint : Point is
  heir to ImpPointAux[Point]
end ImpPoint

class ImpColoredPoint : ColoredPoint is
  heir to ImpPointAux[ColoredPoint]
  -- plus methods specific to this class
end ImpColoredPoint
```

A complexidade relativa desta solução é compensada pela garantia de verificação estática completa.

5 Trabalhos Correlatos

Nesta seção discutimos outros trabalhos que contribuiram para a área de sistemas de tipos em linguagens orientadas a objetos. É importante frisar que a principal contribuição de School não é nenhum mecanismo em particular, mas a forma como diversos mecanismos foram combinados em um sistema seguro, flexível e elegante.

O conceito de separação de hierarquias de tipos e classes, bem como o de hierarquia estrutural de tipos, teve uma forte inspiração em Smalltalk [9]. Nesta linguagem não existe

um conceito explícito de tipo, de modo que toda a herança é relacionada somente com implementações. Por outro lado, qualquer objeto pode, a princípio, ser passado como parâmetro para qualquer método. Somente no caso de um método desconhecido ser aplicado sobre o objeto ocorre um erro, o já citado "*message not understood*". Em particular, se o objeto (parâmetro real) oferece métodos para todas as mensagens que são aplicadas sobre o parâmetro formal, isto é, se suas estruturas são compatíveis, não ocorrem erros de execução.

Outros trabalhos que advocam separação de hierarquias são [5], e as linguagens DuoTalk [15] e Pool [1]. Pool também oferece compatibilidade estrutural. Hierarquias separadas também podem ser construídas em C++ [21], com o uso de classes abstratas para a hierarquia de tipos e de superclasses não públicas para a hierarquia de implementações. Entretanto, tal uso exige grande disciplina por parte do programador.

A facilidade de tipos genéricos com restrições pode ser encontrada em CLU [14] e Eiffel [17]. C++ [21] atualmente oferece um mecanismo para tipos genéricos, denominado "*template*", porém sem restrições. Em Eiffel, o mecanismo permite que se restrinja o uso de uma classe genérica a parâmetros que sejam subclasses de uma dada classe. Este esquema pode ser contrastado com o adotado em School, que usa conformidade ao invés de subtipos.

6 Conclusões

O objetivo principal do projeto de School foi a obtenção de um sistema de tipos flexível, sem abrir mão da segurança proporcionada por verificação estática e sem mudanças na semântica usual de linguagens orientadas por objetos. O sistema de tipos descrito é completamente seguro, isto é, garante, em tempo de compilação, que qualquer programa corretamente tipado executará sem erros "*message not understood*".

Apesar de fortemente tipada, School provê grande flexibilidade. As facilidades resultantes da independência de hierarquias e da tipagem estrutural trazem ganhos ao polimorfismo vertical. Um objeto só é incompatível com uma variável (ou parâmetro) quando existe um bom motivo para isto, ou seja, quando o objeto não tem todas as operações especificadas para a variável. Em relação ao polimorfismo horizontal (tipos e classes paramétricos), o uso do mecanismo de conformidade permite restrições bastante precisas, mantendo a segurança e o poder de expressão da linguagem. Este poder de expressão é discutido mais extensamente em [19], onde são descritas soluções em School para problemas apresentados na literatura como motivação para a utilização de tipagem dinâmica ou brechas de segurança.

Temos atualmente um compilador para a linguagem School. Este compilador está implementado em C++, e roda em estações Sun e PCs. O compilador foi construído com auxílio das ferramentas yacc e lex, e tem aproximadamente quatro mil linhas de código. Uma característica interessante da linguagem, mostrada pelo compilador, é a completa independência entre o sistema de tipos e a semântica de execução. As rotinas de verificação de tipos funcionam como uma verificação de correção parcial do programa, e não tem nenhuma influência na geração de código. Em particular, programas mal-tipados também podem ser executados; apenas perde-se a garantia da ausência de erros de tipos durante a execução.

O sistema de tipos apresentado em [19], base do sistema de School, engloba também o tratamento de exceções. Neste trabalho, esta facilidade é considerada uma parte do sistema de tipos e não meramente um mecanismo de controle. Isto é feito a partir da inclusão das exceções sinalizadas na aridade de cada função. A extensão é feita de forma compatível com o objetivo exposto neste trabalho, isto é, mantendo-se verificação estática. A facilidade de tratamento de exceções será abordada em um trabalho futuro.

Referências Bibliográficas

[1] P. America and F. van der Linden. A parallel object-oriented language with inheritance and subtyping. *Sigplan Notices*, 25(10), 1990. OOPSLA/ECOOP'90 Proceedings.

[2] Pierre America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4), 1989.

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

[4] G. Birtwistle, O. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin*. Petrocelli Charter, 1975.

[5] P. Canning et al. Interfaces for strongly-typed object-oriented programming. *Sigplan Notices*, 24(10):457–467, 1989. OOPSLA'89 Proccedings.

[6] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4), 1985.

[7] Charles Gescke and James Morris. Early experience with Mesa. *Comm. ACM*, 20(8), August 1977.

[8] Giorgio Ghelli. A static type system for message passing. *SIGPLAN Notices*, 25(10), October 1991. (ECOOPS/OOPSLA'91 Proceedings).

[9] Adele Goldberg and Dave Robson. *Smalltalk-80 : The Language and its Implementation*. Addison-Wesley, 1983.

[10] C. Hoare. A proof of correctness of data representations. *Acta Informatica*, 1(4), 1972.

[11] Roberto Ierusalimschy. The O=M programming language. Monografias em Ciência da Computação 3/91, PUC-Rio, Rio de Janeiro, Brazil, 1991.

[12] Roberto Ierusalimschy. A denotational approach for type-checking in object-oriented programming languages. *Computer Languages*, 19(1):19–40, 1993.

[13] P. A. Johnson and C. Rees. Reusability through fine-grain inheritance. *Software — Practice and Experience*, 22(12):1049–1068, 1992.

[14] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. The MIT Press, 1986.

[15] C. Lunau. Separation of hierarchies in Duo-Talk. *Journal of Object-Oriented Programming*, 2(2):20–26, 1989.

[16] Ole Madsen and Boris Magnusson. Strong typing of object oriented languages revisited. In *ECOOP/OOPSLA '90 Proceedings*, pages 140–150, 1990.

[17] Bertrand Meyer. Eiffel – a language and environment for software enginnering. *The Journal of Systems and Software*, 8(3):129–46, 1988.

[18] N. Rodriguez, R. Ierusalimschy, and J. L. Rangel. Types in School. *Sigplan Notices*, 28(8), 1993.

[19] Noemi Rodriguez. *Um Sistema de Tipos Orientado por Objetos, Incluindo Persistência.* PhD thesis, Dep. Informática, PUC-Rio, Rio de Janeiro, Brazil, 1993.

[20] SPA, Sistemas, Planejamento e Análise. *TOOL for Windows: the object oriented language*, 1992.

[21] B. Stroustrup. *The C++ Programming Language.* Prentice-Hall, second edition, 1992.

[22] N. Wirth. *Programming in Modula-2.* Springer-Verlag, third edition, 1985.