RENATO FERREIRA BORGES

UMA IMPLEMENTAÇÃO PARA A LINGUAGEM SCHOOL

DISSERTAÇÃO DE MESTRADO

DEPARTAMENTO DE INFORMÁTICA

Rio de Janeiro, 9 de agosto de 1996

Renato Ferreira Borges

Uma Implementação para a Linguagem School

Dissertação apresentada ao Departamento de Informática da PUC-Rio como parte dos requisitos para a obtenção do título de Mestre em Informática: Ciências da Computação.

Orientador: Roberto Ierusalimschy

Departamento de Informática

Pontifícia Universidade Católica do Rio de Janeiro

Rio de Janeiro, 9 de agosto de 1996

Ao meu pai

Agradecimentos

Às minhas mãe e irmã, por tudo.

Aos outros *Jacksons* – Cassino, Clinio, Costa e Renato –, ao Claudio, ao Roberto, à Anna e à Tatiana, como grandes amigos que sempre estiveram do meu lado desde a minha graduação, especialmente o Renato.

A todos os amigos do Departamento de Informática – alunos, ex-alunos, professores e funcionários – e todos do TeCGraf.

Ao CNPq e ao SEPROC/CENPES/Petrobrás pelo auxílio financeiro.

Resumo

Esta dissertação apresenta uma implementação de um compilador para a linguagem orientada a objetos School. O trabalho se detem às técnicas de implementação específicas de linguagens orientadas a objetos, tais como envio de mensagens, tipos genéricos e coleta de lixo.

Inicialmente é apresentado um estudo de várias propostas para a implementação de diferentes aspectos de linguagens orientadas a objetos. Em seguida, o compilador de School é descrito e avaliado levando em consideração as técnicas estudadas.

A dissertação também investiga alguns aspectos de módulos em linguagens orientadas a objetos e propõe um sistema de módulos para linguagem de programação School. A proposta é descrita com a utilização de um método formal.

Abstract

This thesis presents an implementation of a compiler for the School object oriented language. The work is focused on the specifics compilation techniques employed on this kind of language like message passing, generic types and garbage collection.

At first is introduced a study of many proposals for the implementation of different object oriented languages. Based on such study, the School compiler is described and evaluated.

In this work, some aspects of modules in object oriented languages are investigated and a proposal is made of a module system for the programming language School. A formal method is used to describe the module system proposed.

Conteúdo

Capítulo 1

Introdução

Linguagens orientadas a objetos demandam a utilização de técnicas não convencionais de compiladores em suas implementações. Tipicamente, estas linguagens apresentam características como amarração dinâmica e facilidades para tipos genéricos, que não estão presentes na maioria das linguagens convencionais. Muitos trabalhos sobre compiladores se baseiam em linguagens convencionais, deixando de lado os aspectos específicos de linguagens orientadas a objetos.

Enquanto linguagens como Fortran, C e Pascal operam basicamente sobre tipos como inteiros, reais e ponteiros — que têm operações correspondentes a nível de máquina na maioria das arquiteturas —, linguagens orientadas a objetos permitem a criação de novos tipos com novas operações, trabalhando em um nível mais alto. Isto faz com que várias construções nestas linguagens não possuam uma tradução direta para o código de máquina, tornando a geração de código mais complexa.

Amarração dinâmica de mensagens é um dos exemplos destas construções. A arquitetura de programas orientados a objetos envolve constantemente o envio de mensagens entre objetos. Cada envio de mensagem corresponderá à execução de um método que implementa como o objeto responde à mensagem. O método a ser executado depende do objeto que recebe a mensagem, e por isso não pode ser determinado durante a compilação. Um envio de mensagem é traduzido para uma amarração dinâmica. A amarração dinâmica exige a busca, em tempo de execução, do método a ser executado. A maneira como a busca é implementada depende do modelo utilizado na linguagem para a descrição dos objetos, mas em qualquer caso esta busca é menos eficiente do que uma simples chamada de função (amarração estática, utilizada em linguagens convencionais). Em geral, quanto mais flexível é a linguagem, menos eficiente é a implementação da amarração dinâmica adequada à linguagem.

Como os sistema orientados a objetos se baseiam na troca de mensagens, a eficiência da amarração dinâmica é um fator determinante no desempenho destes sistemas. Para minimizar a perda da amarração dinâmica em relação à estática, muitos trabalhos têm sido realizados para permitir a otimização dos envios de mensagem, muitas vezes tentando-se descobrir em tempo de compilação o método a ser executado [CUL89, Lea90, CU90, CU91, LS91, HCU91, HU94, DCG95, GDGC95, AH95, DGC95]. A aplicação ou não destas otimizações pode determinar variações de ordens de grandeza no desempenho de um programa [CU91]. As técnicas desenvolvidas variam bastante, mas em geral elas são feitas a partir de uma análise das descrições dos objetos, e portanto não podem ser feitas sobre um código intermediário, ao contrário de otimizações tradicionais como alocação de registros.

Como um dos benefícios obtidos com a utilização de linguagens orientadas a objetos deve ser a facilidade de construção de sistemas reutilizáveis, linguagens tipadas permitem descrições parametrizadas de objetos, possibilitando, por exemplo, que a mesma descrição de uma pilha possa ser utilizada para a construção de várias pilhas que empilham elementos de tipos diferentes. A compilação de uma classe parametrizada, entre outras coisas, não é trivial e depende de uma série de características da linguagem [DGLM95], que vai determinar, por exemplo, se é possível compilar uma única versão da classe [ML94] ou se é necessário ter uma versão para cada tipo utilizado como parâmetro [ES90].

A orientação por objeto facilita a construção de estruturas de dados complexas, o que dificulta um gerenciamento de memória explícito feito pelo programador. A coleta automática de lixo [Ake95], além de liberar o programador da tarefa de recuperação de áreas de memória não mais usadas, é mais confiável do que a coleta manual, que, se feita sem cuidado, pode trazer consequências imprevisíveis para a execução do programa. Muitas linguagens orientadas a objetos têm algum sistema de coleta de lixo automática [CUL89, LS91, Ake95, Mey96].

Apesar de a coleta de lixo não ser nenhuma característica intrínseca de linguagens de programação, o coletor de lixo é um suporte para a execução dos programas e depende de como a linguagem faz uso da memória. Portanto, a implementação da linguagem precisa levar isto consideração na tradução dos programas, permitindo que o coletor de lixo entre em ação. Além disso, o próprio coletor de lixo deve ser fornecido em conjunto com o compilador.

School [Ier93, RIR93b] é uma linguagem de programação orientada a objetos projetada para oferecer a semântica básica de linguagens orientadas a objetos, mantendo um sistema de tipos seguro e flexível, com toda a verificação de tipos feita em tempo de compilação. De modo a aumentar a flexibilidade, foi adotada em School a separação entre os conceitos de tipo (especificação) e classe (implementação), hierarquias separadas para tipos e classes, compatibilidade estrutural e tipos genéricos com restrições. O resultado é uma linguagem com grande poder de expressão de tipos.

Pelas características da linguagem, uma implementação de School precisa tratar os tópicos apresentados acima. Esta dissertação aborda vários pontos que precisam ser levados em consideração na implementação de um compilador para uma linguagem orientada a objetos e descreve a implementação atual do compilador de School. O trabalho apresenta um estudo das implementações e soluções adotadas em diversas linguagens, como SELF [US87], Sather [SOM93], Cecil [Cha93], Smalltalk [GR83], Trellis [SCB+86], Theta [DGLM95], C++ [Str92, Str95], Modula-3 [Nel91], Oberon-2 [MW91], Eiffel [Mey88a] e Ada [Ada83]. O trabalho também apresenta as decisões que guiaram a implementação do compilador de School, analisando alternativas que podem vir a ser incorporadas no futuro. Em geral, a aplicabilidade de técnicas de implementação depende do sistema de tipos adotado pela linguagem. A implementação adotada em School é contrastada com estas técnicas, onde é feita uma análise de quais delas podem ser aplicadas em School.

Com o advento de linguagens orientadas a objetos, o mecanismo de módulos perdeu parte de sua importância, apesar de vir sendo incorporado nas linguagens de programação no final da década de 70 para apoiar o desenvolvimento de grandes sistemas e a programação com Tipos Abstratos de Dados. Inicialmente as linguagens orientadas a objetos não incorporaram suporte para módulos. A base desta mudança é a idéia de que classes, como mecanismo de encapsulamento e abstração, podem substituir módulos. Mais recentemente, esta equivalência vem sendo rebatida por vários autores, por exemplo [Szy92]. Se por um lado o uso de classes pode substituir o de módulos com vantagens no suporte a Tipos Abstratos de Dados, módulos vistos como subsistemas que apresentam um conjunto de classes, tipos e suas inter-relações estão em outro nível de abstração [PG94, BV93].

Durante o trabalho, sentiu-se a necessidade de incorporar algum suporte a módulos na linguagem. A linguagem foi estendida para incorporar um mecanismo de módulos, possibilitando, entre outras coisas, a compilação em separado. Para chegar à proposta adotada, são definidos dois tipos diferentes de interfaces de módulos, a partir dos quais é feita uma análise em comparação com diversas linguagens. Como descrições anteriores da linguagem não previam esta extensão, o mecanismo proposto foi completamente formalizado, tendo como resultado uma descrição precisa do seu comportamento.

Tendo um mecanismo de módulos, torna-se possível a produção de componentes a serem utilizados na confecção de sistemas maiores. Após a construção do compilador, este seria um segundo passo na direção de tornar a linguagem mais utilizada. Utilizando o modelo de aplicações guiadas por eventos, extremamente adequadas para aplicações interativas, o trabalho apresenta uma proposta para a formação de uma base de módulos — uma biblioteca — a ser disponibilizada com o compilador.

A dissertação discute a implementação dos pontos apresentados acima: amarração dinâmica, tipos genéricos, coleta de lixo e módulos em linguagens orientadas a objetos, apresentando a implementação adotada na linguagem School.

Além da introdução, a dissertação está dividida em mais quatro capítulos. O capítulo a seguir descreve a linguagem School, fornecendo a base para as análises sobre a aplicação das técnicas estudadas na linguagem. O capítulo 3 discute cada um dos quatro pontos apresentados, apresentando uma visão unificada de várias propostas e técnicas utilizadas para a implementação em diferentes linguagens. Como a aplicabilidade de algumas técnicas depende fortemente da linguagem, as apresentações são acompanhadas de uma análise da aplicabilidade das técnicas em linguagens que utilizam modelos diferentes. O capítulo seguinte descreve a implementação dos pontos estudados adotada em School e analisa algumas alternativas que poderiam ser utilizadas. Finalizando, o último capítulo traz algumas conclusões e aponta alguns trabalhos futuros que podem se aproveitar do estudo realizado nesta dissertação.

Capítulo 2

A linguagem School

School[1] é uma linguagem de programação orientada a objetos projetada para oferecer a semântica básica de linguagens orientadas a objetos, mantendo um sistema de tipos seguro e flexível, com toda a verificação de tipos feita em tempo de compilação. Todos valores em School são objetos. Variáveis não contêm objetos, mas suas referências. A comunicação entre objetos é feita através de mensagens com "late-binding", e a ligação mensagem $\times$ método é dependente apenas do objeto receptor da mensagem.

A principal regra que guiou o desenvolvimento do sistema de tipos de School foi evitar qualquer construção que pudesse causar um erro de tipos durante a execução de um programa. Por erro de tipo entende-se enviar uma mensagem para um objeto que não tenha um método para tratá-la [Ier93]. Obedecendo esta restrição, tentou-se fazer a linguagem tão flexível quanto possível. De modo a aumentar a flexibilidade, foi adotada em School a separação entre os conceitos de tipo (especificação) e classe (implementação), hierarquias separadas para tipos e classes, compatibilidade estrutural, e tipos genéricos com restrições [RIR93b]. Como resultado, chegou-se a uma linguagem enxuta, mas com grande poder de expressão de tipos.

A importância da separação de hieraquias de tipos e classes em LOOs finalmente começa a ser mais amplamente reconhecida [Val95]. A principal idéia por trás deste conceito é o entendimento de tipos como especificações (ou "contratos"), e classes como implementações. O tipo de um objeto é sua "aparência" externa, sua interface com o exterior. Em School, uma declaração de tipo descreve todas as operações disponíveis sobre objetos que satisfazem este tipo. Como exemplo, a descrição de um tipo `Stack` é apresentada na figura 2.1.

```
type Stack{T} is
  proc push(v : T);
  proc pop -> T;
  proc empty -> Bool;
end Stack;
```

Figura 2.1: Exemplo de declaração de Tipos em School.

Por outro lado, a classe de um objeto descreve sua estrutura interna, isto é, suas

[1]Este capítulo é parte dos trabalhos [Rod93] e [RIR93a].

variáveis e código. Uma possível implementação para pilhas, usando listas encadeadas, é apresentada na figura 2.2.

```
private class lliststack{T} : Stack{T} is
  include llist{T};
  proc push( v : T);
  proc pop -> T;
  constructor new;
end lliststack;

proc lliststack.push(v : T) is
  self.addHead(v);
end;

proc lliststack.pop -> T is
  Result := self.retrievePos(1);
  self.removePos(1);
end;

constructor lliststack.new is end;
```

Figura 2.2: Exemplo de declaração de Classes em School.

Em geral, um mesmo tipo pode ter várias implementações diferentes. Nas declarações de variáveis, parâmetros etc., somente o tipo é especificado. Uma variável pode referenciar qualquer objeto cuja classe satisfaça o seu tipo. Uma classe é especificada apenas para a criação de um objeto. Novos objetos são criados através de chamadas a *contrutores*, que explicitamente referem o nome da classe. Uma chamada a um construtor aloca espaço para o novo objeto, e depois executa o corpo do construtor. É interessante observar que construtores não fazem parte do tipo de um objeto, pois não são operações aplicáveis em objetos existentes.

2.1 Compatibilidade estrutural

Usualmente um tipo é considerado subtipo de outro apenas quando isto é declarado, explicitamente ou por transitividade. A independência entre subtipos e subclasses de School permite à linguagem adotar compatibilidade estrutural para subtipos. Isto significa que um tipo é considerado subtipo de outro como consequência deles serem compatíveis, e não ao contrário. Um tipo A é considerado subtipo de B sempre que não houver possibilidade de erros quando um objeto A é usado no lugar de B. Como erros ocorrem quando uma mensagem é enviada a um objeto que não tem um método para tratá-la, estes podem ser evitados com a seguinte definição: um tipo A é subtipo de B ($A \prec B$) se e somente se $A = B$ ou, 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 [CW85] e é necessária para garantir a correção do sistema de tipos. Uma definição formal de subtipos, assim como uma prova que esta definição evita erros de tipos, está em [Ier93].

2.2 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.

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 outorga este direito; veja discussão sobre *self* mais adiante. Podemos também definir um tipo *interno* para uma classe, denotado 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 *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. [Ier91]), 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 [Ghe91]. 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 [RIR93b]).

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
  proc M1;
end A;

type A1 is
  proc M2;
end A1;

type B2 is
  proc M (a : A1);
end B2;

Class CA : A is
  var b : B2;
  proc M1;
  proc M2;
  proc M3;
end CA;

proc CA.M3 is b.M(self) end;
```

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.

2.3 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
  proc push (v : T);
  proc pop -> T;
  proc isEmpty -> 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{Int},
    ssi : Stack{Stack{Int}};
```

Ao contrário do que poderia parecer à 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 sequenciais 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}
  where T has proc equal (v : T) -> 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 proc equal (v : T1) -> Boolean; end T1;
```

e um tipo `String`:

```
type String is
  proc equal (v : String) -> Boolean;
  ...
end String;
```

Neste exemplo, `String` se conforma a `T`, mas não é um subtipo de `T1`[2]. 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 proc equal (v : T) -> Boolean; end T2;
```

temos que um tipo `X` se conforma a `T2` se e somente se $\texttt{X} \prec \texttt{T2}\{\texttt{X}\}$. 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 contravariância para parâmetros de métodos. Considere o seguinte exemplo:

```
type Point is
  proc equal (p : Point) -> Boolean;
  proc X -> Int;
  proc Y -> Int;
end Point;

type ColoredPoint is
```

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

```
  proc equal (p : ColoredPoint) -> Boolean;
  proc X -> Int;
  proc Y -> Int;
  proc color -> 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.

```
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 [MM90]. 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
  proc equal (p : t) -> Boolean;
  proc X -> Int;
  proc Y -> Int;
end PointAux;

class ImpPointAux{t} : PointAux{t} is
  -- implementation of Points
end ImpPointAux;

class ImpPoint : Point is
  include ImpPointAux{Point};
end ImpPoint;

class ImpColoredPoint : ColoredPoint is
  include ImpPointAux{ColoredPoint};
  -- plus procs specific to this class
end ImpColoredPoint;
```

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

Capítulo 3

Implementações de LOOs

Linguagens orientadas a objetos demandam a utilização de técnicas não convencionais de compiladores em suas implementações. Este capítulo apresenta um estudo sobre a implementação de alguns pontos chave de linguagens orientadas a objetos.

3.1 Módulos

No final da década de 70, o conceito de módulo começou a ser incorporado em linguagens de programação. Representantes típicas desta fase são as linguagens Ada [Ada83] e Modula-2 [Wir85]. As principais motivações para este novo mecanismo eram apoiar o desenvolvimento de grandes sistemas (*programming in the large*), e a programação com Tipos Abstratos de Dados.

Com o advento de linguagens orientadas a objetos (LOOs), o mecanismo de módulos perdeu parte de sua importância. Várias linguagens, como Eiffel [Mey88a] e Sather [SOM93], não incorporaram suporte para módulos. A base desta tendência é a idéia de que classes, como mecanismo de encapsulação e abstração, podem substituir módulos. Alguns autores foram mais longe, chegando ao ponto de identificar os conceitos de classes e módulos [Mey88b].

Mais recentemente, esta equivalência vem sendo rebatida por vários autores, por exemplo [Szy92]. Por um lado, o uso de classes pode substituir o de módulos com vantagens no suporte a Tipos Abstratos de Dados. Entretanto, módulos vistos como subsistemas que apresentam um conjunto de classes, tipos e suas inter-relações estão em outro nível de abstração, geralmente denominados *frameworks*. Para sistemas de porte médio, com um número de classes menor que poucas centenas, frameworks podem ser construídos via disciplina de programação. Em sistemas mais complexos, um mecanismo de suporte a este outro nível de abstração faz-se necessário [PG94, BV93].

Mesmo em sistemas de menor porte, o mecanismo de classes não é suficiente para organizar o uso de diversas bibliotecas desenvolvidas independentemente. Como consequência de um espaço de nomes único para classes de um sistema, bibliotecas que definam classes ou outras entidades homônimas se tornam incompatíveis. Este problema levou C++ [Str92] a incorporar recentemente um mecanismo de *namespaces* [Laj93] para controle de visibilidade. Apesar de *namespaces* não serem exatamente módulos, eles provêem controle de visibilidade de nomes em um nível mais externo às classes. Outras LOOs mais recentes incorporam explicitamente um mecanismo de módulos, além de classes (por exemplo Oberon-2 [MW91] e Modula-3 [Nel91]).

Esta seção analisa alguns aspectos que caracterizam um sistema de módulos considerando uma LOO. A análise é feita a partir de uma proposta de diferenciação entre interfaces conceitualmente diferentes: a interface de acesso ao módulo e a interface do módulo para o compilador. Os sistemas de módulos de algumas LOOs são analisados baseando-se nestas novas definições.

3.1.1 Interfaces

Para analisar os mecanismos de interface e modularização oferecidos por diferentes linguagens, este trabalho sugere uma distinção conceitual entre dois tipos de interface: a *interface de acesso ao módulo* (IAM), que descreve como ele pode ser utilizado externamente, e a *interface do módulo para o compilador* (IMC), que inclui todas as informações necessárias para a compilação de outros módulos que o importem. Módulos que importam um módulo qualquer M são denominados clientes de M.

A interface relevante para aspectos de projeto de sistemas modulares é a IAM. A IAM define o encapsulação de um módulo, descrevendo as operações válidas sobre ele. Por exemplo, se um módulo exporta uma classe, as operações permitidas sobre a classe fazem parte da IAM. Quaisquer outros atributos desta classe que não sejam públicos não afetam a manipulação de objetos desta classe, e portanto não fazem parte da IAM. No entanto, para a geração de código que manipule esta classe, o compilador pode precisar de informações extras. Um dos dados possivelmente necessários à compilação é quanta memória deve ser alocada para a criação de objetos desta classe. Estas informações, apesar de não constarem na IAM, são necessárias para a compilação e, portanto, pertencem à IMC. A IMC é uma necessidade do compilador e não deveria influenciar o projeto dos módulos. Como a interface de acesso é uma das informações necessárias para a compilação de um cliente, a IAM é normalmente um subconjunto da IMC.

3.1.2 Separação entre interface e implementação

Linguagens que separam interface e implementação em duas descrições textuais, como Ada e Modula-3, utilizam uma única descrição de interface, que deve, ao mesmo tempo, documentar a interface do módulo e atender às necessidades do compilador por questões de implementação. Isto acaba por poluir estas descrições com informações que não fazem parte da IAM, além da possibilidade de introduzir inconsistência entre as descrições. Em algumas linguagens nenhum teste é feito para assegurar esta consistência, que fica sob responsabilidade do programador.

Modula-2, uma das primeiras linguagens a incluir o conceito de módulo, utiliza duas descrições complementares para um módulo: um módulo de definição e um módulo de implementação. A consistência entre as descrições é garantida, já que a implementação depende do módulo de definição. O módulo de definição contém todas as declarações exportadas pelo módulo. A razão da separação é preservar os detalhes de implementação, considerados privados do módulo [Gei83].

No entanto, esta separação não cumpre totalmente seu objetivo. Todos os tipos usados nas declarações devem ser também declarados. Isto implica que, além dos tipos exportados, um módulo de definição pode conter declarações adicionais que não fazem parte da IAM, mas são referenciados na declaração de um tipo exportado. Para contornar este problema, Modula-2 permite especificar tipos exportados opacos. Uma definição de tipo em um módulo de definição pode conter apenas seu nome, sem a estrutura. Com isso, módulos clientes conhecem o nome do tipo mas não suas propriedades, impedindo manipulação de objetos deste tipo. A declaração completa do tipo fica no módulo de implementação. Esta solução tem limitações. Para simplificar a geração de código, a exportação de tipos opacos é restrita a ponteiros e inteiros.

Ada é outra linguagem que separa a definição de um módulo (*package* em Ada) em duas descrições: a especificação do pacote (*package specification*) e o corpo do pacote (*package body*). A interface é descrita no pacote de especificação, que pode conter constantes, tipos, variáveis e unidades de programas. Todos os elementos presentes na especificação podem ser acessados por módulos clientes. O corpo do pacote complementa a especificação. Ele só é necessário se a especificação declara unidades de programa; nesse caso ele contém o corpo das unidades.

É possível esconder a estrutura de um tipo para módulos clientes declarando um tipo privado. Nesse caso a especificação deve conter uma parte privada onde o tipo privado é declarado completamente. Tipos privados de Ada são equivalentes aos tipos opacos de Modula-2. A diferença é o local da declaração completa. Modula-2 mantém a declaração completa apenas no módulo de implementação, já que esta é uma informação exclusiva da IMC (e com isso limita tipos opacos apenas a ponteiros e inteiros). Ada não impõe nenhuma limitação aos tipos privados. Em contrapartida, o pacote de especificação acaba contendo informações além da IAM.

C++ também pode ser considerada uma linguagem com descrições separadas. Apesar de não oferecer nenhum suporte a módulos no nível da linguagem, é prática comum a separação de uma unidade de compilação em dois arquivos distintos (`.c` e `.h`) através de disciplina de programação e inclusão textual de arquivos durante o pré-processamento. A análise a seguir considera que um módulo C++ é definido em dois arquivos, um com a implementação e outro com a interface (*header file*).

Por motivos operacionais, a interface descrita em um *header file* não é a IAM. A necessidade da descrição completa das classes, inclusive membros privados, pode levar à exportação de classes que foram projetadas para uso interno ao módulo. Isto acontece por exemplo quando uma classe a ser exportada tem entre seus atributos privados um objeto de uma classe interna. A declaração da classe exportada deve estar no *header file*. No entanto, a descrição da classe interna tem que estar disponível para esta declaração, ou seja, também deve estar no *header file*, o que a torna visível externamente.

O mecanismo de funções em linha de C++ também gera informações para a IMC. Operacionalmente, se esta otimização for utilizada, a descrição do corpo da função deve estar disponível na hora de uma chamada. Isto significa que, para uma função importada de outro módulo ser otimizada desta maneira, não apenas sua declaração, mas o seu corpo também deve estar na descrição da interface.

Outro mecanismo de C++ que pode gerar muitas informações exclusivas da IMC é o de tipos parametrizados, denominado *templates*. Em C++ existe uma implementação para cada classe individual construída a partir de um template. A informação necessária para a geração de uma classe é a declaração completa da *template*, inclusive implementações de métodos. Como o manual de referência da linguagem [ES90] não define se a declaração completa do template deve estar presente no contexto da instanciação, as implementações variam, causando até um problema de portabilidade [Str95]. Muitas delas exigem que o template seja declarado completamente no arquivo `.h`.

3.1.3 Descrição única para módulos

Nas linguagens analisadas na seção anterior, a IAM acaba se diluindo entre várias informações desnecessárias para o uso dos módulos, e a descrição converge para a IMC. A separação deixa de ser um aspecto que facilita a organização dos módulos tornando clara sua interface, e passa a ser guiada pelas restrições técnicas demandadas pelo compilador. No caso de C++, chega-se ao ponto de as próprias implementações de funções aparecerem da descrição da interface, e não na implementação.

Algumas linguagens com mecanismo de módulos usam apenas uma descrição. Oberon é um dos exemplos mais significativos destas linguagens. Uma das diferenças de Oberon em relação à Modula-2, sua predecessora, é que não há mais uma separação física entre interface e implementação [Wir88]. Um módulo de Oberon é descrito por um único arquivo, onde estão juntas interface e implementação. A compilação gera um arquivo de símbolos, que é consultado para a compilação dos módulos clientes. Neste modelo, a IAM não existe como uma descrição separada, e a IMC é gerada durante a compilação do módulo. O fato de não existir a interface separadamente não deve ser considerado uma perda em relação às linguagens que separam um módulo em duas descrições. Ferramentas podem facilmente extrair esta interface do módulo, com a vantagem de esta interface ser exatamente a IAM.

Uma das consequências de a IMC ser gerada pelo compilador é que qualquer informação que possa ser útil para a compilação de módulos clientes pode ser incluída na IMC sem interferência do usuário. Isto evita forçar o usuário a entrar em detalhes sobre quais informações são necessárias para a compilação. O próprio compilador define o conteúdo da IMC.

A otimização de expansão de funções em linha, em particular, pode ser melhor aproveitada em linguagens que geram um arquivo de símbolos durante a compilação de um arquivo fonte único. Em linguagens com separação de interface e implementação, as informações que podem ser utilizadas pelo compilador para a compilação de um módulo cliente são controladas pelo programador, já que a interface é gerada por ele. Para que a expansão seja possível, o programador deve tornar as informações necessárias disponíveis explicitamente. Apesar de o programador indicar quais funções devem ser expandidas, o compilador é que julga se as chamadas serão otimizadas. O controle do programador sobre a otimização não é total. Além disso, funções que poderiam ser otimizadas deixam de ser, simplesmente porque o programador não as indicou como em linha. Em linguagens onde a descrição do módulo é única isto pode ser contornado. Se o próprio compilador é quem gera a interface, é possível, durante a compilação, analisar quando uma função pode ser expandida em linha [DC94]. Se for o caso, o arquivo gerado passa a ter informações de como expandir a função, e a otimização pode ser feita sem a interferência do programador.

3.1.4 Recompilação de módulos clientes

Em linguagens com descrições separadas, a recompilação de um módulo pode ser decidida verificando se alguma interface dos módulos importados foi modificada após a última compilação do cliente. Isto pode ser facilmente verificado estabelecendo as relações de dependência entre os módulos usando uma ferramenta estilo *make*. Esta ferramenta é feita para compilar programas consistentemente e eliminar recompilações desnecessárias. *Make* funciona a partir de uma descrição do projeto que contém, para cada arquivo, uma lista de arquivos pré-requisitos, chamada lista de dependências. A compilação de um arquivo só ocorre se um ou mais arquivos da lista de dependências foi alterado desde a última vez que o arquivo foi compilado. Para verificar se um arquivo está desatualizado, *make* compara a hora de modificação do arquivo com os arquivos da lista.

Em sistemas onde os módulos são descritos por um único arquivo, não é claro quando os módulos clientes devem ser recompilados, o que pode ser considerado desvantajoso. Se a IMC é sempre gerada durante uma compilação e a necessidade de recompilação de um cliente é decidida em função da data da IMC, a compilação de um módulo invalida todos os módulos clientes, mesmo que a alteração não afete a interface.

O caso típico é a alteração da implementação de alguma função do módulo. Esta é provavelmente a modificação mais comum, e não altera a IAM. Em linguagens com descrições separadas, normalmente esta alteração se aplica apenas ao arquivo que contém a implementação. Se os módulos clientes dependem apenas das descrições da interface dos módulos importados, este caso não implica na recompilação de módulos clientes. Apenas o arquivo modificado precisa ser recompilado. Já em linguagens que utilizam uma só descrição, se ao recompilar um módulo alterado o arquivo de símbolos for gerado novamente, todos os módulos clientes serão invalidados, pois a data do arquivo de símbolos será mais recente do que todos os módulos clientes.

Para evitar a recompilação desnecessária dos módulos clientes neste caso, Oberon só gera o arquivo de símbolos diante de um pedido explícito [Wir88]. Esta característica passa o controle da geração do arquivo de símbolos para o usuário. Como nem sempre a necessidade de atualização desta descrição é clara, já que envolve o conhecimento da IMC por parte do usuário, os módulos podem ficar inconsistentes entre si. Se a interface for modificada e o arquivo de símbolos não for atualizado, módulos clientes podem acessar incorretamente este módulo.

Oberon-2 impede esta inconsistência verificando se a edição do módulo causou alterações na interface. Se a interface mudar e não for pedida a geração de um novo arquivo de símbolos, um erro é sinalizado.

Outras políticas podem ser adotadas para minimizar a recompilação de módulos clientes. Uma alteração simples em relação a Oberon-2 seria a geração automática do arquivo de símbolos sempre que for detectada uma alteração na interface. Outra estratégia pode ir além, e evitar a recompilação em alguns casos mesmo que a interface seja alterada, analisando não apenas a data da IMC, mas também as diferenças de conteúdo em relação à última IMC utilizada para a compilação de um determinado cliente. Se a alteração mantém a IMC compatível, não há necessidade de recompilação.

3.2 Amarração dinâmica

Amarração dinâmica[1] é provavelmente o recurso mais poderoso de linguagens orientadas a objetos, ajudando na construção de sistemas extensíveis e reutilizáveis. No entanto, este recurso apresenta problemas em termos de desempenho, sendo o ponto crítico de programas orientados a objetos. A causa do problema é que as chamadas de funções não podem ser amarradas em tempo de compilação. Além da necessidade de busca da função durante a execução do programa, esta amarração dinâmica dificulta a aplicação de outras otimizações. Como exemplo, uma amarração estática pode ser substituída por uma expansão em linha da função a ser chamada. Existem trabalhos sobre como decidir

[1] *late binding, dynamic dispatch* ou ainda despacho dinâmico

automaticamente pela expansão em linha de funções [DC94].

Em algumas linguagens, como C++ [Str92] e Sather [SOM93], a utilização da amarração dinâmica é controlada pelo programador. No entanto, nos pontos onde esta amarração não é utilizada o programa perde em extensibilidade, exatamente um dos maiores benefícios de sistemas orientados a objetos.

Em linguagens tipadas que utilizam uma mesma hierarquia, declarada explicitamente, para tipos e classes é possível fazer amarração dinâmica utilizando-se tabelas de funções para cada classe. Este é o caso de C++ e Sather. Com as tabelas, a perda de desempenho de uma chamada de função em relação à amarração estática pode ser de apenas uma indireção, no caso de herança simples, ou uma indireção e uma soma com herança múltipla [ES90]. Em linguagens sem esta hierarquia explícita, como Smalltalk [GR83], School [RIR93b] e SELF [US87], não é trivial usar esta implementação. Nesses casos a amarração dinâmica exige uma busca real em tempo de execução do método a ser chamado. Por mais eficiente que seja esta busca, o desempenho em relação à amarração estática é muito pior. A amarração estática pode ser traduzida para uma única instrução de máquina, enquanto que a dinâmica exige a execução de uma função que, por menor que seja, é ordens de grandeza maior do que uma simples instrução. Uma amarração dinâmica em School, por exemplo, é de 7 a 20 vezes mais lenta que a estática (seção 4.3). Nestas linguagens, é fundamental a utilização de técnicas de otimização de amarração dinâmica.

A melhor otimização para uma amarração dinâmica é transformá-la em estática. Para isso é necessário conhecer a classe do objeto que receberá a mensagem. Uma técnica básica se utiliza do fato que a classe de um objeto é conhecida no momento de sua criação. Enquanto for possível manter esta informação analisando o fluxo de execução do programa, a amarração estática pode ser usada. Por exemplo:

```
var obj : T := C!; -- declara uma variável obj do tipo T e
                   -- a inicializa com um objeto da classe C
obj.f(); -- chamada pode ser amarrada estaticamente
         -- à implementação de f da classe C
```

Em linguagens onde a amarração dinâmica é muito cara (implementação sem tabelas de funções), é possível aproveitar melhor as informações sobre as classes, otimizando mesmo quando existe mais de uma possibilidade para a classe. Se a classe de um objeto pode ser limitada a um conjunto pequeno de possibilidades, o envio de mensagens pode ser traduzido em alguns testes de tipos, cada caso podendo ser amarrado estaticamente. Considerando o seguinte trecho de código:

```
var obj : T;
obj = if teste then A! else B! end;
obj.f(); -- classe de obj pertence a {A, B}
```

O envio da mensagem pode ser substituído por um teste sobre a classe do objeto; nesse caso em cada ramo a classe será conhecida, permitindo uma amarração estática:

```
if classe(obj) = A then
  call A::f  -- chama a implementação de f definida na classe A
else
  call B::f  -- chama a implementação de f definida na classe B
end;
```

Nas linguagens onde é necessário realizar uma busca para chamar o método, esta técnica substitui a execução da função de busca por um (ou poucos) testes, melhorando bastante o desempenho das amarrações. No entanto, esta técnica não apresenta grandes ganhos. Tipicamente nos sistemas orientados a objetos a criação e a manipulação dos objetos são feitas separadamente, o que reduz em muito a possibilidade desta otimização ser aplicada.

Várias técnicas foram desenvolvidas para se ter, em tempo de compilação, a classe dos objetos que recebem mensagens. As seções seguintes apresentam algumas destas técnicas.

3.2.1 Separação de mensagens

Separação de mensagens — *message splitting* — pode ser considerada uma extensão da ideia básica apresentada na introdução, que mantém a informação de classe enquanto for possível a partir da criação do objeto. Esta técnica procura preservar por mais tempo esta informação.

Uma função pode ter vários ramos internamente, determinados por comandos *if* etc. Nas junções destes ramos pode-se perder informações que seriam úteis para otimizar futuros envios de mensagens. Quando isso acontece, pode ser vantajoso separar o envio de mensagens em vários casos adiando a junção dos ramos.

A figura 3.1 apresenta um exemplo de código que pode ser beneficiado pela separação de mensagens. Sem nenhuma otimização, o envio de mensagem a *obj* será traduzido para uma busca, em tempo de execução, da implementação de *message* a ser executada.

```
proc f (o : T) is
  if not o then
     o = A!;
  elseif exp then
     o = B!;
  end;
  o.message();
end;
```

```
proc f (o : T) is
  if not o then
     o = A!;
     o.message();
  elseif exp then
     o = B!;
     o.message();
  else
    o.message();
  end;
end;
```

Figura 3.1: Aplicação da separação de mensagens

A função f se divide em três ramos. Dentro do primeiro ramo, a variável o referencia um objeto da classe A. No segundo ramo o objeto será da classe B, enquanto que no terceiro a classe não é conhecida. Depois da junção dos ramos a classe é desconhecida, o que evita a otimização. No entanto, esta otimização pode ser feita nos dois primeiros ramos se a junção for adiada para depois do envio. O resultado é equivalente à reescrever a função f como na figura 3.1 à direita. O terceiro ramo continua precisando de amarração dinâmica, mas os dois primeiros podem ser amarrados estaticamente.

A separação de mensagens é usada em SELF. A versão inicial do compilador [CUL89] só era capaz de separar as mensagens enviadas logo após as junções (separação de mensagens local). Versões posteriores usam uma separação de mensagens estendida [CU90], onde todas as oportunidades de separação são usadas, independente de quantos nós existem entre a junção e o envio da mensagem. A separação de mensagens estendida pode aumentar muito o tamanho do código gerado, já que todos os nós situados entre a junção e o envio da mensagem precisam ser duplicados, além de ser um processo exponencial.

Além disso, no primeiro compilador de SELF com separação de mensagens estendida, a detecção de quando separar uma mensagem era feita com *backtraking*, o que resultava em tempos de compilação muito grandes, em média dez vezes maiores do que C otimizado [CU91], gerando um código cinco vezes maior. Estes problemas levaram à uma nova versão do compilador sem *backtracking* [CU91], para tornar a compilação mais rápida, e utilizando técnicas de compilação adiada (*deferred compilation*) — só adequadas a sistemas de compilação interativa — para reduzir o tamanho do código gerado e o tempo de compilação, evitando a compilação de casos incomuns (ver previsão de tipos na seção 3.2.3).

Esta técnica pode ser usada com ganhos mesmo em linguagens que utilizam tabelas de funções para a implementação da amarração dinâmica, pois ela aproveita testes que já seriam realizados mesmo sem a otimização.

3.2.2 Análise de tipos para loops

O início de um loop também é uma junção, já que o código executado antes do loop pode ter sido o final dele próprio. Para manter a informação das classes dentro de um loop, é preciso levar em consideração não apenas as informações de classes disponíveis antes da entrada do loop, mas também as do final. Como as informações no final dependem das informações no início, existe uma dependência circular que dificulta a obtenção destas informações.

Uma solução é compilar o loop sem informação nenhuma, o que anula qualquer possibilidade de otimização nos envios de mensagens. Considerando que a importância dos loops não é desprezível quando se trata de desempenho, seria interessante obter de alguma maneira estas informações também dentro deles.

Uma das maneiras de se ter estas informações é fazendo uma análise iterativa [CU90]. O primeiro passo usa as informações de antes do loop para analisar o seu corpo e obter as informações no final. Se as informações forem as mesmas do início, está terminada a análise. Senão, as informações do início e do final são unificadas e é feita uma nova análise com estas novas informações mais gerais. O processo continua até que se chegue ao ponto fixo, onde as informações do início são compatíveis com as do final.

Esta análise pode ser utilizada para gerar várias versões do corpo do loop, cada uma com informações mais específicas. As versões são conectadas adequadamente de acordo com a análise. Esta técnica é utilizada em SELF [CU90, CU91].

Esta otimização nem sempre apresenta ganhos quando aplicada em linguagens que utilizam tabelas de funções. Isso só acontece se existir apenas uma possibilidade de classe para algum objeto manipulado dentro do loop. Quando há mais de uma possibilidade, a amarração dinâmica não pode ser substituída por uma simples amarração estática, são necessários testes sobre a classe do objeto. Nesse caso, só ocorre algum ganho se a amarração dinâmica é muito cara.

3.2.3 Previsão de tipo

Esta técnica se aplica a linguagens que não possuem tipos primitivos, como inteiros, booleanos etc. Nestas linguagens, até uma simples soma de inteiros pode ser traduzida para uma amarração dinâmica. A fim de evitar esta sobrecarga, o compilador pode tentar prever a classe de determinados objetos baseando-se na mensagem a ser enviada. Algumas operações são típicas destes tipos primitivos, que podem ser previstos na grande maioria das vezes. Mensagens *and* e *or* são quase sempre aplicadas em booleanos; somas e comparações são quase sempre com inteiros; indexações com vetores etc. A partir destes fatos, pode ser mais eficiente gerar um código que verifica se a classe é a esperada antes de enviar a mensagem. Se for, a amarração pode ser estática. Caso contrário é feito a amarração dinâmica. Aliando esta otimização com a expansão de funções em linha o código pode melhorar muito.

```
a=dispatch(b,not)(b);
```

(a)

```
if (b->tipo==Boolean)
 a=boolean_not(b);
else
 a=dispatch(b,not)(b);
```

(b)

```
if (b->tipo==Boolean)
 a=!b;
else
 a=dispatch(b,not)(b);
```

(c)

Figura 3.2: Previsão de tipo

A figura 3.2 mostra três possíveis traduções para a expressão booleana `a := not b`. A primeira (a) não usa previsão de tipo. Como o método `not` pode se aplicar a qualquer objeto, é necessário fazer uma amarração dinâmica. Na segunda versão (b) a previsão de tipo é usada para que, se b for um booleano, a amarração seja estática, já que esta probabilidade é muito grande e seria um custo enorme realizar uma busca do método apenas para esta operação trivial. A terceira versão (c) vai mais longe. Como a amarração pode ser estática se a previsão se confirmar, é possível expandir a função `boolean_not` em linha, minimizando os custos da linguagem não possuir tipos primitivos.

3.2.4 Customização

A customização consiste na compilação de várias versões de um mesmo método, uma para cada classe do receptor. O ganho é que, dentro de cada método, a classe exata do receptor é conhecida e portanto qualquer envio de mensagem ao próprio objeto pode ser amarrado estaticamente.

A customização equivale à redefinição automática, nas classes derivadas, de todos os métodos definidos nas classes base. Suponha a hierarquia de classes apresentada na figura 3.3a, onde a implementação de `g` envia a mensagem `f` para o próprio objeto. O método `g` pode ser invocado para qualquer objeto das classes `A`, `B`, `C` e `D`. Como a classe `C` redefine o método `f`, a implementação de `f` a ser executada, quando chamada por `g`, depende da classe do receptor e não pode ser amarrada estaticamente. Com a customização, a hierarquia seria compilada como na figura 3.3b, onde os métodos x' tem o mesmo corpo da última definição de x segundo a hierarquia. Nesta nova hierarquia, cada classe tem sua implementação de `g`, e nesse caso, em todos os métodos a classe do receptor é conhecida.

Com isso, os envios de mensagens ao receptor podem ser amarrados estaticamente. A implementação de `g` de uma classe X está amarrada à implementação de `f` da mesma classe X. O ganho não é só na chamada da função propriamente dita. A partir do momento em que a função é conhecida em tempo de compilação é possível a expansão do seu corpo na chamada e assim novas otimizações podem ser feitas.

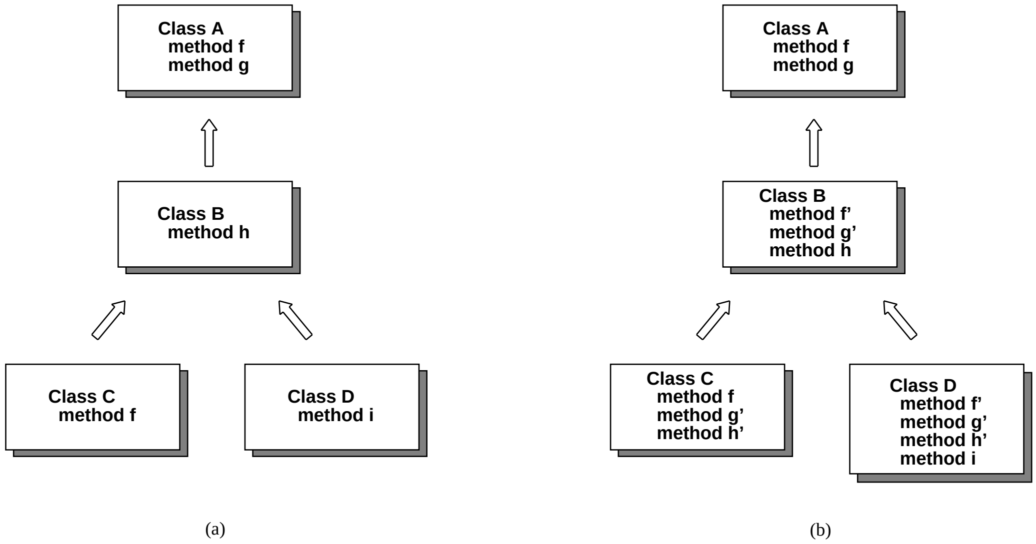

Figura 3.3: Hierarquias original e customizada

Esta técnica é utilizada em SELF [CUL89], Sather [LS91] e Trellis [SCB+86], e foi proposta para C++ [Lea90].

Apesar dos benefícios, a customização apresenta alguns problemas que tornam o seu uso restrito. Em sistemas grandes com hierarquias profundas e classes com muitos métodos, a customização de cada um dos métodos levaria à explosão do código, tornando impraticável a sua utilização. Ao mesmo tempo, várias customizações são desnecessárias: uma mesma versão de um método pode se aplicar a vários receptors sem perda nenhuma. Trellis minimiza este efeito comparando todas as versões geradas e eliminando as repetições.

A customização é adequada à sistemas interativos que compilam sob demanda, como SELF. Nesse caso, a compilação só é feita quando o método é usado, minimizando o problema da explosão do código. Para outras linguagens, a aplicação da customização seria mais interessante seletivamente, com uma análise de custos e benefícios.

Uma generalização da customização é a especialização. Enquanto a customização só usa o receptor para gerar versões dos métodos, a especialização pode usar também os parâmetros. Se com a customização a explosão do código já é um problema, a aplicação da especialização só faz sentido seletivamente. O trabalho [DCG95] apresenta técnicas para identificação de onde a especialização pode ser usada com grandes ganhos, com um estudo de caso na linguagem Cecil [Cha93].

3.2.5 Feedback de tipos

A idéia do feedback de tipos é extrair as informações das execuções anteriores do programa e fornecê-las ao compilador. O feedback pode ser dinâmico, durante a execução do programa, ou estático, após a execução do programa. Nesse caso, a informação é obtida por uma versão instrumentada do programa, que guarda uma lista das classes utilizadas e suas frequências em cada envio de mensagem do programa. Esta técnica também é chamada de previsão da classe do receptor baseada em profiles [GDGC95].

Esta técnica é semelhante à previsão de tipos (seção 3.2.3). Enquanto esta última se baseia nas mensagens para fazer a previsão, o feedback de tipos usa estatísticas da execução de programa para esse fim. O feedback de tipos foi implementado em algumas versões do compilador de SELF [HU94, AH95].

3.2.6 Caches

Caches também podem ser usadas para melhorar o desempenho da amarração dinâmica quando esta é muito cara. A cache mapeia pares <classe do receptor, mensagem> em métodos, mantendo os últimos resultados das buscas feitas. Um envio de mensagem consulta a cache; se a consulta não der resultado, o mecanismo normal de amarração dinâmica é usado e a cache é atualizada. Este tipo de cache é utilizado no Berkeley Smalltalk, que seria 37% mais lento sem a cache [Kra83].

Caches em linha

O envio de mensagens com cache ainda é muito mais lento do que uma simples chamada a um procedimento, já que a cache precisa ser varrida a cada mensagem enviada.

Na prática, a classe do receptor raramente varia em um determinado envio de mensagem. Se uma mensagem é enviada a um objeto da classe X em um ponto particular, é muito provável que a próxima vez que este envio for feito a classe do receptor continue sendo a mesma X. Esta localidade pode ser explorada mantendo-se o último método usado no próprio ponto de envio da mensagem. A instrução *call* pode ser sobreescrita para que o fluxo seja desviado diretamente para o último método usado, o que evita qualquer busca. Com esta implementação, o tamanho do código gerado para o envio da mensagem não aumenta.

Como a classe pode não ser a mesma do último envio, os métodos precisam, no seu início, verificar a classe do objeto. Se esta não for a esperada pelo método, é feita a busca tradicional. Este tipo de cache é chamada cache em linha, já que a informação é guardada na própria chamada [DS84]. As implementações de Smalltalk usam este tipo de cache, obtendo um surpreendente *hit ratio* de 95% [HCU91].

Uma extensão das caches em linha são as caches em linha polimórficas [HCU91], usadas para guardar mais do que um resultado em cada envio de mensagem. Envios de mensagens verdadeiramente polimórficos, onde um envio de mensagem é aplicado a uma série de objetos de classes diferentes não são beneficiados por caches em linha. Pior, a sobrecarga dos *cache misses* piora o desempenho destes envios, já que a cache vai sendo trocada entre vários métodos. Este panorama não é incomum. Um sistema gráfico que armazena em uma lista vários objetos gráficos de classes diferentes e envia a mensagem `desenha` para cada um deles é um caso típico muito usado, assim como vários casos semelhantes.

As caches polimórficas se comportam como as caches em linha enquanto um envio de mensagem só for aplicado a uma única classe. Se o envio for realizado sobre uma classe diferente, é criada uma pequena rotina, para onde o envio é desviado. A rotina compara a classe do receptor com todas as classes que já foram usadas e nesses casos é feita uma amarração estática com o método correspondente.

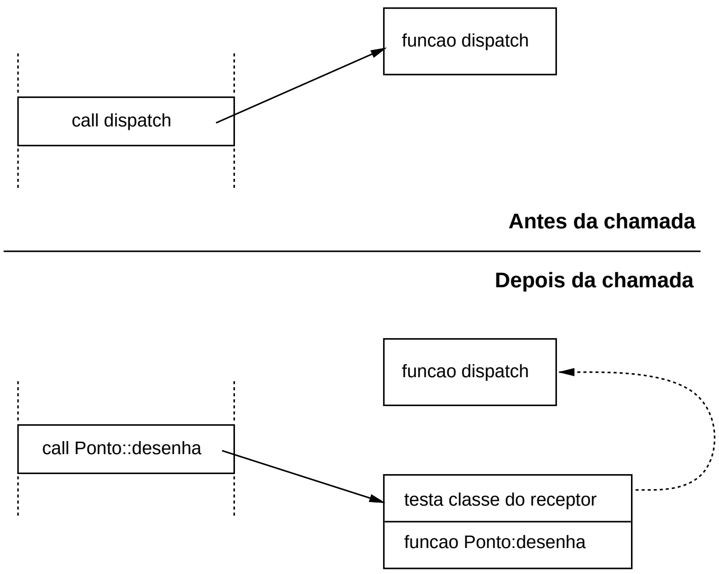

Figura 3.4: Caches em linha

Os métodos continuam precisando de um prólogo para testar se a classe do receptor é a mesma do método. Isso porque enquanto só uma classe for usada o método é o mesmo das caches em linha padrão.

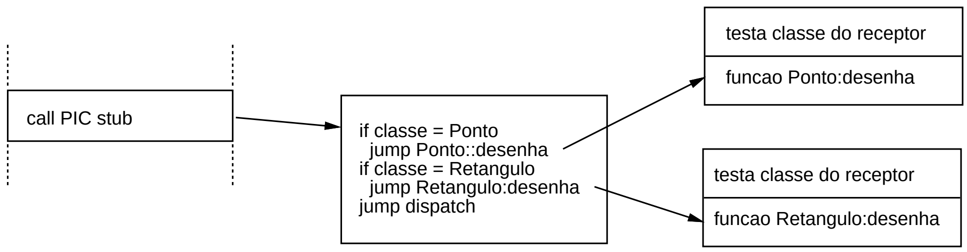

Figura 3.5: Caches em linha polimórficas

Se o envio for realizado sobre uma nova classe, a rotina é estendida para tratar o novo caso. Em algum momento a cache terá todas as classes usadas na prática, e a partir daí não ocorrerão mais *cache misses*.

Algumas variações podem ser usadas, como a reordenação dos testes para que a classe mais frequente seja testada antes, uma limitação no tamanho da cache no caso de muitas classes serem usadas em um determinado ponto e até mesmo a expansão em linha de métodos muito simples dentro da própria cache.

Na verdade estas caches polimórficas são uma forma de se implementar o feedback de tipos dinamicamente, podendo também ser usada como um pequeno banco de dados com informações a serem usadas em futuras compilações (seção 3.2.5).

3.2.7 Análise estática de hierarquia

Analisando a estrutura da hierarquia de classes, o compilador pode obter informações que possibilitem a transformação de uma amarração dinâmica em estática. As informações são extraídas analisando as possíveis classes de um objeto receptor e onde o método é definido na hierarquia.

A partir da hierarquia da figura 3.6, considere que o método `p` da classe `F` envia a mensagem `m` para o próprio objeto. Com apenas uma análise intra-procedural, o envio não tem como ser amarrado estaticamente. No entanto, analisando `F` e suas subclasses — possíveis classes para o receptor —, nota-se que nenhuma delas redefine o método `m`. Em todos os casos, a implementação a ser chamada é a definida na classe `C`, e portanto o envio pode ser substituído por uma amarração estática. Em linguagens tipadas que usam a hierarquia de classes também como declaração de tipos, a otimização pode ser aplicada não apenas em mensagens enviadas ao receptor. Ainda sobre a mesma hierarquia, a classe de um objeto referenciado por uma variável do tipo `F` pertence ao conjunto formado por `F` e suas subclasses e portanto qualquer envio da mensagem `m` a este objeto pode ser otimizado. Esta técnica se diferencia da maioria por não procurar descobrir a classe exata do receptor, mas sim se basear na hierarquia para descobrir onde os envios podem ser substituídos por amarrações estáticas.

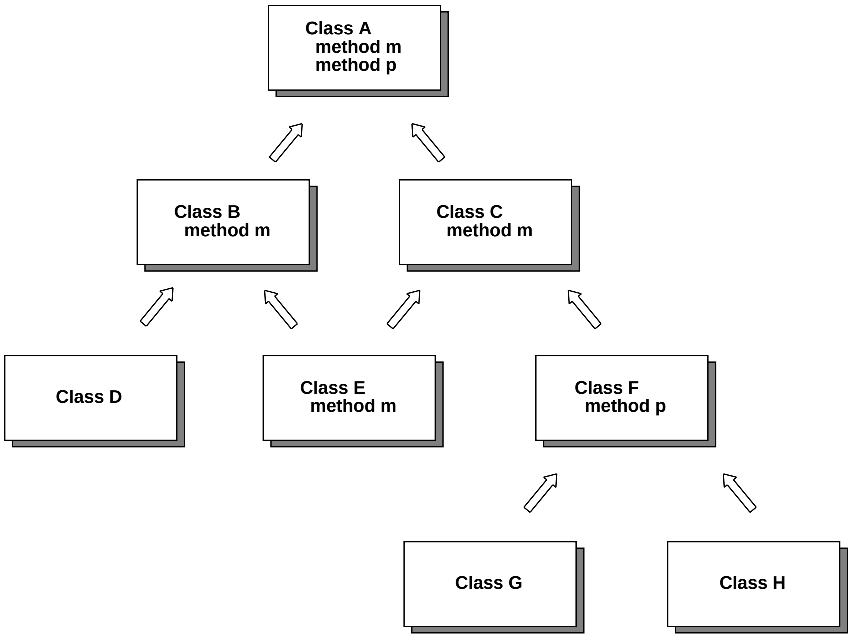

Figura 3.6: Hierarquia beneficiada pela análise estática de hierarquia

Uma alternativa semelhante pode ser obtida em C++ com a declaração combinada de métodos virtuais e não virtuais. No entanto, a análise da hierarquia obtém melhores resultados, já que que um método virtual de C++ o será em toda a hierarquia, enquanto que a análise trata as sub-árvores com sucesso. Além do fato que a análise é feita automaticamente sem a intervenção do programador, não influindo na extensibilidade das classes, o que não acontece em C++.

A desvantagem é que a análise precisa ser feita sobre toda a hierarquia a cada compilação, o que pode ser pouco eficiente. Existem propostas para possibilitar a utilização incremental desta análise, assim como a sua utilização com multi-métodos [DGC95].

A análise pode ser feita mesmo que o fonte de todo o programa não esteja disponível, fato comum em sistemas que se utilizam de bibliotecas independentes. Se a descrição das bibliotecas tiver a hierarquia das suas classes com os métodos correspondentes, a análise pode ser totalmente feita. A perda fica por conta do código da biblioteca, que não pode se utilizar desta análise para otimizações.

3.2.8 Inferência de tipos

Inferência de tipos é usada em linguagens não tipadas para determinar a correção dos envios de mensagem da aplicação. Esta correção implica que, para cada mensagem enviada aos objetos, existe um método para tratá-la. Nesse caso não há risco de ocorrerem erros de execução causados por mensagens não entendidas.

No contexto de inferência de tipos, normalmente tipos são definidos como conjuntos finitos de classes [PS91, APS93, AH95]. A inferência pode ser feita a partir de uma análise global do código, onde é computado um tipo para cada expressão no programa. Se o tipo $\{Class_1, Class_2, ..., Class_i\}$ é inferido para alguma expressão E, garante-se que, para qualquer execução do programa, sempre que a expressão E for avaliada o resultado será um objeto de umas das classes pertencentes ao seu tipo.

Além do seu objetivo principal – garantir a correção do programa –, a inferência pode ser usada para otimizar amarrações dinâmicas. Os tipos inferidos para as expressões podem ser usados como base de informações para a compilação, da mesma maneira que o feedback de tipos (seção 3.2.5). A diferença é que, ao contrário deste último, a inferência de tipos não depende da execução do programa e fornece informações exatas.

Trabalhando com aproximações para os tipos, algoritmos para inferência de tipos foram propostos para linguagens como Smalltalk [PS91] e SELF [APS93].

3.3 Tipos parametrizados

Em linguagens não tipadas, nenhum mecanismo de tipos parametrizados é necessário. Já uma linguagem tipada sem este recurso impede a construção de tipos e classes genéricos, muito comum para a implementação de *containers* por exemplo. Se uma das propostas de linguagens orientadas a objetos é facilitar a construção de sistemas reutilizáveis, o suporte para a construção de classes e tipos parametrizados é mandatório.

A dificuldade de implementação está em como gerar código para operar sobre um tipo desconhecido e como armazenar dados deste tipo. Se a linguagem usa a mesma representação para todos os seus elementos, ou seja, não existem tipos primitivos, o armazenamento deixa de ser um problema já que todos os dados são objetos. Se envio de mensagem é feito da mesma forma independente do tipo do objeto (implementação de amarração dinâmica sem tabelas de funções), as operações sobre os tipos desconhecidos também podem ser feitas sem problema. No entanto, poucas linguagens satisfazem estas condições, o que faz com que a implementação deste recurso não seja trivial na prática.

As soluções se dividem em duas frentes: a declaração de uma classe parametrizada não gera código; ela é apenas um modelo para a geração das classes reais. Nesse caso, existe uma instância para cada utilização da classe com parâmetros diferentes. A outra solução gera uma classe genérica de fato; todas (ou quase todas) as utilizações desta classe compartilham o mesmo código.

3.3.1 Modelos para a geração de classes

A implementação de uma classe genérica exige um mecanismo de envio de mensagens independente da classe dos objetos. Quando a implementação da amarração dinâmica de uma mesma mensagem varia de acordo com a declaração do tipo do objeto — é o caso com tabelas de funções —, a implementação da classe genérica fica difícil. Se a linguagem permite expressar restrições para os tipos genéricos, esta implementação é possível com tabelas auxiliares (ver seção 3.3.2). Na ausência destas restrições, a solução é ter várias implementações de uma classe genérica. A classe genérica não é uma classe, mas sim um modelo para a geração de uma família de classes. Esta solução acrescenta mais um nível na relação objeto-classe. Objetos são instâncias de classes e classes passam a ser instâncias de modelos de classes. Modula-3 e C++ utilizam esta solução; nesta última as classes genéricas são chamadas *templates*.

Esta solução apresenta alguns problemas. A geração das classes depende de como o programa usa o modelo, de quais tipos são usados na instanciação destes modelos. Isto implica que as informações necessárias para gerar uma classe completa devem estar disponíveis. Linguagens estaticamente tipadas sem restrições para tipos genéricos só podem verificar a correção de uma instanciação expandindo a classe. Este fato viola os princípios de encapsulação e modularidade, já que uma alteração na implementação da classe genérica pode invalidar o código que a use.

Outro problema se refere ao tamanho do código gerado, já que cada uso do modelo com tipos diferentes pode gerar uma classe completa. Se a geração das classes é feita à medida que estas são instanciadas no programa, é difícil evitar a duplicação de código se existe suporte para compilação em separado, piorando o problema do tamanho do código gerado. Para evitar a duplicação do código, é necessário analisar o programa inteiro antes de gerar as classes. Isto pode ser feito atrasando esta geração para depois da ligação. Esta implementação evita ainda a geração de versões de métodos não utilizados efetivamente pelo programa. Esta implementação já exige um suporte maior do ambiente de geração de sistemas; não depende apenas do compilador. Além disso, o tempo de compilação pode aumentar muito.

3.3.2 Classes genéricas de fato

O tratamento de uma classe genérica como uma classe de fato, e não um modelo para a geração de classes, resolve os problemas citados na seção anterior. Para isso é preciso gerar um código para a classe que se adapte a qualquer tipo. Se a implementação permite o envio de mensagem a um objeto de forma única independente do seu tipo, é possível gerar um código que manipule quaisquer objetos, tendo uma única versão da classe para todas as instanciações. É importante ressaltar que se, mesmo nesse caso, a linguagem não suporta restrições aos parâmetros, a checagem de tipos só pode ser feita se o código (ou informações extraídas do código) da classe parametrizada estiver disponível nos pontos de instanciação.

As restrições aos parâmetros possibilitam ainda a implementação de uma única versão da classe mesmo que a forma de enviar mensagens não seja única. Em uma implementação com tabelas, por exemplo, a classe genérica pode ter uma tabela de métodos adicional, com uma entrada para cada mensagem presente nas restrições. Esta tabela seria montada de acordo com a instanciação da classe, fazendo o mapeamento das restrições para os métodos dos objetos. Esta solução é utilizada em Theta [DGLM95, ML94].

Ainda que seja possível o envio de mensagens de uma forma única, a presença de tipos primitivos na linguagem apresenta problemas, já que as manipulações sobre estes tipos não são traduzidas para envios de mensagens. Algumas soluções podem ser usadas para permitir a utilização de tipos primitivos com classes genéricas. Uma delas é gerar mais de uma implementação da classe. Por exemplo, uma declaração `class C{T}` (classe `C` parametrizada pelo tipo `T`) geraria uma versão para operar com objetos e mais uma para cada tipo primitivo. Considerando que a parametrização pode ser feita para mais de um tipo, o número de versões geradas seria

$$(\textit{número de tipos primitivos} + 1)^{\textit{número de parâmetros}},$$

o que é impraticável. Para evitar a geração de todas estas versões seria necessário analisar o programa inteiro para descobrir quais são efetivamente usadas, caindo nos problemas discutidos na seção anterior.

Outra solução seria criar classes internas para a representação dos tipos primitivos como objetos comuns. A utilização de uma classe genérica com um tipo primitivo estaria usando as classes internas, e não os tipos primitivos. A vantagem é que a classe genérica tem apenas uma versão que pode ser usada sempre. As desvantagens são eficiência das classes genéricas — tipos primitivos não são acessados diretamente —, maior gasto de memória – cada elemento de um tipo primitivo precisa ser encapsulado por um objeto para ser usado em classes genéricas — e eficiência de passagem de parâmetros e retorno de funções, já que várias conversões implícitas entre tipos primitivos e classes internas que o modelam são necessárias.

Este trabalho apresenta uma proposta para uma implementação única de classes genéricas sem sacrificar muito a eficiência das operações sobre tipos primitivos. A geração de código da classe genérica seria um pouco diferente da normal. Cada envio de mensagem seria traduzido para uma série de testes sobre o tipo do objeto. Por exemplo, o método `menor` da classe `C`, parametrizada com o tipo `T`:

```
proc C.menor( v1, : T, v2 : T ) -> Bool is
  Result := v1 < v2;
end
```

seria traduzida para (considerando que a linguagem tem um tipo primitivo Int):

```
C_menor( self, v1, v2 )
{
  if (tipo(v1) == Int)
    Result = v1 < v2 ? True : False;
  else
    Result = dispatch(v1, "lessthan")(v1,v2);
  return Result;
}
```

Foi usado um teste — `tipo(v1)` — para checar em tempo de execução o tipo real de `v1`. Esta expressão deve retornar com qual tipo foi instanciada a classe, informação que deve estar presente em cada objeto de uma classe genérica. Mesmo que o programa não disponha de informações sobre os tipos em tempo de execução, o compilador pode gerar código para preencher estas informações a cada instanciação de uma classe genérica.

Na presença de restrições sobre os tipos parâmetros, pode-se evitar estes testes a cada envio de mensagem no corpo da função, o que certamente causa um aumento no tamanho do código gerado. Os testes podem ser feitos em *stubs* — um para cada restrição —, e os envios de mensagem seriam desviados para estes *stubs*.

Para a implementação de funções parametrizadas, a informação do tipo real dos parâmetros — que para os métodos fica guardada no próprio objeto — precisa ser passada como parâmetro adicional da função, o que pode ser feito automaticamente pelo compilador de acordo com a instanciação.

3.4 Coletor de lixo

A orientação por objetos facilita a construção de estruturas de dados complexas, o que dificulta um gerenciamento de memória explícito feito pelo programador. A coleta automática de lixo, além de liberar o programador da tarefa de recuperação de áreas de memória não mais usadas, é mais confiável do que a coleta manual, que, se feita sem cuidado, pode trazer consequências imprevisíveis para a execução do programa. A maioria das linguagens orientadas a objetos tem algum sistema de coleta de lixo automática, considerada por alguns (por exemplo [Mey88b]) um dos requisitos obrigatórios de qualquer sistema intitulado orientado a objetos.

3.4.1 Contagem de referências

O algoritmo mais simples de coleta de lixo é o de contagem de referências. Neste algoritmo, cada objeto tem um contador que indica a quantidade de referências existentes para o objeto no momento. O contador é atualizado a cada atribuição, e assim que ele chega a zero a memória ocupada pelo objeto pode ser recuperada. Esta técnica não é realista. É pouco eficiente em termos de tempo e espaço. Todas as atribuições terão uma sobrecarga de somas e testes e cada objeto precisa de um campo a mais — o contador. Além do problema da eficiência, esta técnica não recupera estruturas circulares.

3.4.2 Marcar e varrer

Uma técnica bastante usada é a chamada marcar-e-varrer (*mark-and-sweep*). O algoritmo é composto de duas fases: a primeira identifica todos os objetos que são referenciados pelo programa. Esta fase parte de um conjunto raiz, que é marcado. Todos os objetos referenciados por esse conjunto vão sendo marcados por uma busca no grafo formado pelos objetos e suas referências. A segunda fase varre todos os objetos e libera os não marcados. Este algoritmo tem a vantagem de coletar lixo mesmo na presença de estruturas circulares. No entanto, a sua desvantagem é que a primeira fase causa uma parada temporária do programa. A segunda fase, de recuperação, pode ser feita em paralelo com a execução, já que uma vez identificado como lixo um objeto não pode mais ser referenciado.

O coletor de lixo de Sather [LS91] utiliza este algoritmo.

Coleta incremental

Para evitar a parada do sistema sempre que o coletor entra em ação, o algoritmo de marcar-e-varrer pode ser adaptado para realizar uma coleta de lixo incremental.

Na coleta incremental, o primeiro passo do algoritmo — a marcação dos objetos —, que causa a parada da aplicação, passa a ser feito paralelamente à aplicação como uma co-rotina. Para manter o estado entre as ativações, o coletor faz uma marcação nos objetos, que podem assumir três estados:

1. o objeto é usado pela aplicação e suas referências já foram percorridas;

2. o objeto é usado pela aplicação e suas referências ainda não foram percorridas ou

3. o objeto ainda não foi percorrido.

Um objeto usado pela aplicação é denominado *vivo*. O processo parte do conjunto raiz, cujos objetos são vivos e precisam ser percorridos. Este panorama equivale à marcação dos objetos do conjunto raiz com o estado 2 e o restante com o estado 3. Cada passo toma um objeto no estado 2 e percorre suas referências. Todos os objetos referenciados serão alcançáveis a partir do conjunto raiz e portanto vivos; se algum deles estiver no estado 3, será promovido para o estado 2. Ao final do passo, com suas referências já percorridas, o objeto passa ao estado 1 e não será mais considerado até o próximo ciclo.

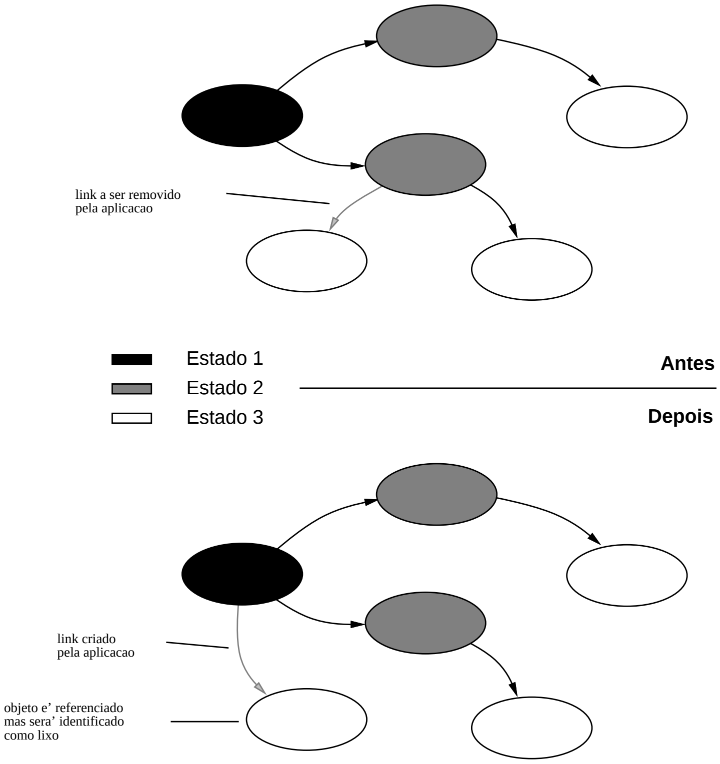

Figura 3.7: Coleta incremental sem coordenação entre coletor e aplicação

Os objetos no estado 2 representam uma fronteira entre os objetos vivos (estado 1) e o lixo em potencial (estado 3). Nunca um objeto no estado 1 referencia diretamente outro no estado 3. A cada passo do processo, os objetos vão sendo promovidos do estado 3 para o estado 2 e deste para o estado 1, até que não existam mais objetos no estado intermediário 2. Neste ponto, os objetos no estado 1 são vivos e o restante é lixo, podendo ser coletado. Como no algoritmo não incremental, a recuperação das áreas de memória pode ser feita em um outro processo paralelo.

A dificuldade da coleta incremental está no fato de que o grafo formado pelos objetos e suas referências é acessado paralelamente pelo coletor e pela aplicação. Se a aplicação altera o grafo, o coletor pode se perder levando à liberação de áreas referenciadas pela aplicação. O invariante que diz que um objeto no estado 1 nunca aponta diretamente para outro no estado 3 não pode ser quebrado por mudanças feitas pela aplicação. Por exemplo, se entre dois passos do coletor a aplicação alterar um objeto no estado 1 fazendo com que este aponte para um objeto no estado 3 e destruir a referência original para este último (figura 3.7), este será identificado como lixo erradamente, pois o algoritmo considera que as referências dos objetos no estado 1 já foram percorridas e por isso o objeto no estado 3 permanecerá neste estado até o fim do ciclo.

Uma maneira de coordenar o coletor com o mudador é utilizar uma *barreira de escrita*. A barreira de escrita é um meio de registrar, para o coletor, as alterações que a aplicação faz no grafo formado pelos objetos e suas referências. Sempre que a aplicação muda o valor de um ponteiro, a escrita é registrada. A maneira mais comum de se fazer este registro é marcar, na hora da atualização, o novo valor como vivo. Se o objeto está no estado 3, passa ao estado 2, garantindo que ele não será liberado. Para implementar a barreira de escrita, é necessário alterar a geração de código para a atribuição, o que envolve uma perda no desempenho. O compilador de SELF tenta minimizar esta perda com uma barreira de escrita otimizada [Höl93].

Uma sofisticação deste algoritmo baseada em gerações (*generation scavenging* [Ung84]) vem sendo utilizada em várias implementações de linguagens orientadas a objetos, por exemplo Smalltalk [Ake95], SELF [CUL89] e Eiffel [Mey96].

Capítulo 4

O compilador de School

Este capítulo descreve o compilador de School, apresentando as estratégias e técnicas adotadas em sua implementação, assim como possíveis alternativas. As apresentações fazem referência às técnicas utilizadas em outras linguagens, estudadas no capítulo anterior.

4.1 Modelo de implementação

O compilador School está escrito e gera código ANSI C. Com isso, o compilador pode ser portado e utilizado diretamente em qualquer ambiente que suporte C.

C foi escolhida como linguagem intermediária pela sua portabilidade; provavelmente esta é a melhor aproximação portátil de assembly. Além do aspecto portabilidade, o uso de C como linguagem intermediária traz outras vantagens. Se o compilador gerasse assembly diretamente, seria necessário implementar várias técnicas de otimização complexas que já estão presentes nos compiladores C. Gerando código C, este trabalho fica para os compiladores C. Mais ainda, novas tecnologias incorporadas em futuros compiladores C são automaticamente aproveitadas em School. Como consequência, os esforços de implementação do compilador puderam ser direcionados para os aspectos orientados a objetos, o objetivo real do trabalho.

4.1.1 Tradução dos elementos de School

Quase todos os elemento de School são traduzidos para alguma construção em C. A única exceção são os tipos, que só existem em tempo de compilação.

Os objetos School são representados em C por uma estrutura que contém todos os seus atributos e um ponteiro para a sua classe. A compilação de uma classe gera tanto a estrutura que representará os objetos quanto a descrição da classe, que é usada basicamente para realizar a amarração dinâmica (seção 4.3) e a coleta de lixo (seção 4.6). Para qualquer objeto, a estrutura usada para a sua descrição tem como primeiro campo um ponteiro para a classe do objeto. Assumindo que uma descrição de classe tem o tipo `struct Class`, um objeto O qualquer terá a seguinte representação:

```
struct O {
   struct Class *clss;
   /* atributos que definem O */
};
```

Como o primeiro campo das estruturas dos objetos é sempre o ponteiro para a classe, um ponteiro para um objeto pode ser visto como um ponteiro para um ponteiro da estrutura `Class`. O tipo `S_Obj` definido abaixo pode ser usado como um ponteiro para um objeto:

```
typedef struct Class **S_Obj;
```

Qualquer objeto não primitivo em School terá o tipo `S_Obj` em C. Os tipos primitivos são traduzidos diretamente para tipos básicos de C. A equivalência entre os tipos de School[1] e C é mostrada na tabela 4.1.

School	C
`Int`	`int`
`Real`	`float`
`CPointer`	`void*`
objeto	`S_Obj`

Tabela 4.1: Equivalência entre tipos C e School

Os métodos e funções são traduzidos para funções C, sendo que os métodos têm um parâmetro a mais (*self*). O primeiro resultado de uma função School é o resultado da função C. Os seguintes são retornados em parâmetros adicionais. Por exemplo, o método abaixo:

```
proc classe.metodo(p1:tipo, p2:Int) -> (r1:Real, r2:tipo) is ... end;
```

é traduzido para:

```
float _classe6_metodo(S_Obj self, S_Obj p1, int p2, S_Obj* r2) { ... }
```

Variáveis School são traduzidas para variáveis C, assim como os valores que não são calculados em tempo de compilação (outros valores são substituídos diretamente pelo seu valor e não existem durante a execução).

O mapeamento dos nomes School para C é feito de maneira que não haja possibilidade de conflito com qualquer outra declaração School válida.

4.1.2 Ligação com C

Considerando que os programas são traduzidos para C, a linguagem foi expandida para permitir a comunicação com código escrito em C. A partir daí, várias bibliotecas ficaram automaticamente disponíveis. Para esta comunicação, uma nova palavra reservada — `cproc` — foi incluída na linguagem para declarar uma função C externa e possibilitar que o código School a acesse diretamente. Por exemplo, para disponibilizar para School a função `exit`, padrão da biblioteca de C, basta declarar:

```
cproc exit(Int);
```

[1] O tipo `CPointer` será explicado na seção seguinte.

A partir daí esta função pode ser chamada da mesma maneira que uma função School.

Para possibilitar a troca de dados entre códigos School e C, foi criado um novo tipo primitivo na linguagem — `CPointer` — que permite que o código School armazene um ponteiro C qualquer. Por exemplo, as funções `malloc` e `free`, da biblioteca padrão de C:

```
void* malloc( unsigned int );
void  free( void* );
```

que retornam e recebem um ponteiro, respectivamente, podem ser declaradas em School da seguinte forma:

```
cproc malloc(Int) -> CPointer;
cproc free(CPointer);
```

4.2 Módulos

Durante o trabalho sentiu-se a necessidade de incorporar um mecanismo de módulos à linguagem. A partir da análise feita na seção 3.1.1, conclui-se que linguagens que adotam descrições separadas para interface e implementação apresentam problemas na descrição da interface. Seria interessante que a descrição da interface fosse a IAM. No entanto, esta descrição tende para a IMC e muitas vezes o fato dela ser feita pelo programador impõe restrições às possibilidades do compilador. Levando isso em consideração, um módulo em School, assim como em Oberon-2, é descrito por um único arquivo. A compilação de um módulo gera um arquivo de símbolos que descreve a IMC.

Esta não é a única característica a ser considerada no projeto do um sistema de módulos. Outros aspectos devem ser levados em consideração, como por exemplo a exportação de identificadores de um módulo, a utilização de identificadores importados em módulos clientes e o tratamento de elementos privados referenciados por elementos exportados.

4.2.1 Descrição do sistema proposto

Com a finalidade de cobrir todos os detalhes envolvidos e complementar formalizações anteriores [Ier93], esta seção apresenta uma descrição formal do mecanismo de módulos adotado em School, usando VDM [Jon90] como meta-linguagem de especificação.

Declarações

Um módulo em School é uma série de declarações, que podem ser variáveis, valores, tipos, classes e procedimentos. No seu ambiente estão presentes, além destas declarações, todos os elementos importados pelo módulo. Assumindo representações para cada um destes elementos[2], um elemento do ambiente de um módulo pode ser definido da forma abaixo. A necessidade do elemento *Opaque-elem* ficará clara na seção seguinte.

[2]As representações não são necessárias por enquanto, e são definidas na seção 4.2.1. A única hipótese feita é que as referências a tipos (tipos declarados ou tipos de classes) são feitas por nome através do tipo composto *Elem-name*:

$$\begin{aligned} \textit{Elem-name} \ :: \ \textit{mod-name} &: \mathsf{Name} \\ \textit{name} &: \mathsf{Name} \end{aligned}$$

$Elem = Type \mid Class \mid Proc \mid Value \mid Var \mid Opaque\text{-}elem$

À exceção de variáveis, que são sempre privadas, qualquer declaração é exportada a não ser que seja explicitamente declarada privada, o que é feito precedendo a declaração com a palavra reservada `private`. Todos os elementos não privados serão visíveis em módulos clientes. Para manter esta informação, define-se um novo tipo — *Decl* — que representa uma declaração de um elemento. O invariante garante que as variáveis são sempre privadas:

$$\begin{array}{lrl} Decl :: & elem & : Elem \\ & private & : \mathbf{B} \end{array}$$

where

$$inv\text{-}Decl(mk\text{-}Decl(e, p)) \quad \triangleq \quad e \in Var \;\Rightarrow\; p$$

Um ambiente de um módulo contém todas as declarações pertencentes ao módulo. Ele pode ser definido como uma série de declarações acessíveis a partir do nome do elemento, ou seja, um mapeamento de nomes em declarações:

$$Mod\text{-}env = \mathsf{Name} \xrightarrow{m} Decl$$

O ambiente global é composto de vários ambientes de módulos. Este ambientes são o local mais um ambiente para cada módulo importado. Cada um destes ambientes é acessado pelo nome do módulo correspondente, definindo um mapeamento de nomes de módulos em ambiente de módulos. Além deste mapeamento, é usado um nome para identificar o módulo local:

$$\begin{array}{lrl} Env :: & mod\text{-}name & : \mathsf{Name} \\ & modules & : \mathsf{Name} \xrightarrow{m} Mod\text{-}env \end{array}$$

where

$$\begin{array}{l} inv\text{-}Env(mk\text{-}Env(name, mod)) \quad \triangleq \\ \quad name \in \mathsf{dom}\, mod \wedge \\ \quad (\forall d \in \mathsf{rng}\,(mod(name)) \cdot elem(d) \notin Opaque\text{-}elem) \wedge \\ \quad (\forall m \in \mathsf{dom}\, mod \cdot \\ \qquad (\nexists n \in \mathsf{dom}\, modules(m) \cdot n = m) \wedge \\ \qquad (m \neq name \;\Rightarrow\; \forall d \in \mathsf{rng}\, mod(m) \cdot \neg\, private(d)) \wedge \\ \qquad (\forall type \in ref\text{-}types(mod(m)) \cdot \\ \qquad\quad mod\text{-}name(type) \in \mathsf{dom}\, mod \wedge \\ \qquad\quad name(type) \in \mathsf{dom}\, mod(mod\text{-}name(type)))) \end{array}$$

O invariante garante que: o ambiente local faz parte do mapeamento; não existe nenhum elemento opaco no ambiente local; não existe nenhum identificador com mesmo nome do módulo que o define; os ambientes dos módulos importados não tem elementos privados; todos os tipos referenciados por todos os elementos de todos os ambientes pertencem a algum ambiente de algum destes módulos. No invariante foi usada a função *ref-types*:

$$ref\text{-}types : Mod\text{-}env \to Elem\text{-}name\text{-}\mathsf{set}$$

Esta função retorna o conjunto de todos os nomes de tipos referenciados pelos elementos de um ambiente de módulo. Sua definição está na seção 4.2.1.

A partir das definições acima, já é possível descrever como as declarações locais são inseridas no ambiente global. Utilizando uma sintaxe abstrata, uma declaração pode ser descrita como[3]:

$$ElemDecl = TypeDecl \mid ClassDecl \mid ValueDecl \mid ProcDecl \mid VarDecl$$

Assumindo que a denotação de cada uma das declarações é dada pelas funções:

$$\begin{array}{l}
createType\colon TypeDecl \rightarrow (Env \rightarrow Type) \\
createClass\colon ClassDecl \rightarrow (Env \rightarrow Class) \\
createValue\colon ValueDecl \rightarrow (Env \rightarrow Value) \\
createVar\colon VarDecl \rightarrow (Env \rightarrow Var) \\
createProc\colon ProcDecl \rightarrow (Env \rightarrow Proc)
\end{array}$$

a denotação de uma declaração pode ser definida assim:

$$elem\text{-}decl : ElemDecl \rightarrow (Env \rightarrow Env)$$

$$\begin{array}{l}
elem\text{-}decl(decl) \;\triangleq\; \lambda env \cdot \\
\quad \textsf{cases}\ decl\ \textsf{of} \\
\qquad \begin{array}{rl}
TypeDecl(d) \rightarrow & \textsf{let}\ type = createType(d)env\ \textsf{in} \\
 & declare(env, name(type), mk\text{-}Decl(type, private(d))) \\
ClassDecl(d) \rightarrow & \textsf{let}\ class = createClass(d)env\ \textsf{in} \\
 & declare(env, name(class), mk\text{-}Decl(class, private(d))) \\
ValueDecl(d) \rightarrow & \textsf{let}\ value = createValue(d)env\ \textsf{in} \\
 & declare(env, name(value), mk\text{-}Decl(value, private(d))) \\
ProcDecl(d) \rightarrow & \textsf{let}\ proc = createProc(d)env\ \textsf{in} \\
 & declare(env, name(proc), mk\text{-}Decl(proc, private(d))) \\
VarDecl(d) \rightarrow & \textsf{let}\ var = createVar(d)env\ \textsf{in} \\
 & declare(env, name(var), mk\text{-}Decl(var, \textsf{true}))
\end{array} \\
\quad \textsf{end}
\end{array}$$

Para completar a semântica das declarações é preciso definir a função *declare*, que insere uma nova declaração no ambiente:

$$declare : Env \times \textsf{Name} \times Decl \rightarrow Env$$

$$\begin{array}{l}
declare(env, name, decl) \;\triangleq \\
\quad \textsf{let}\ env = mk\text{-}Env(mod\text{-}name, modules)\ \textsf{in} \\
\quad \textsf{if}\ name \notin \textsf{dom}\ modules(mod\text{-}name) \\
\quad \textsf{then let}\ new\text{-}module = modules(mod\text{-}name) \cup \{name \mapsto decl\}\ \textsf{in} \\
\qquad mk\text{-}Env(mod\text{-}name, modules \dagger \{mod\text{-}name \mapsto new\text{-}module\}) \\
\quad \textsf{else}\ error
\end{array}$$

Tipos opacos

Um tipo opaco é um tipo cuja definição não é conhecida, apenas seu nome. Objetos com um tipo opaco não podem ser manipulados; qualquer tentativa de enviar uma mensagem

[3] A sintaxe completa e a denotação de cada uma das declarações não é relevante para esta formalização; assume-se que cada uma das declarações que pode ser exportada é um tipo composto com um campo *private* booleano.

para um objeto destes resultará em um erro de compilação. Como as definições não são conhecidas, a diferenciação é feita apenas pelo nome, usado para testar a compatibilidade nesse caso. Um tipo opaco só é subtipo dele mesmo (mesmo nome) ou de um tipo com interface vazia.

Tipos podem aparecer com a importação de módulos. Se algum elemento exportado pelo módulo incluído referencia um tipo ou classe cuja definição não é visível no módulo cliente, estes tipos e classes referenciados serão tratados como opacos no módulo. Isto acontece quando um elemento exportado referencia um privado ou um elemento definido em outro módulo não importado. Este último caso mostra que School não permite importação implícita. Ou seja, um módulo M_1 que importe um outro módulo M_2 não conhece a interface dos módulos incluídos por M_2, a não ser que sejam importados também em M_1. Com isso, qualquer tipo referenciado na interface de M_2, mas não visível em M_1, será tratado como opaco.

Apesar de não serem elementos da linguagem, tipos opacos podem estar associados a identificadores, e portanto podem ser referenciados. Por isso, o ambiente global de um módulo pode conter elementos opacos. Um elemento opaco representa um tipo opaco, que pode ser tanto um tipo como uma classe. É preciso fazer esta diferenciação:

$$Opaque\text{-}elem = Type\text{-}name \mid Class\text{-}name$$

A representação é a mesma, contendo o nome do módulo que o define e o nome do tipo ou classe:

$$\begin{aligned} Type\text{-}name :: \ mod\text{-}name &: \mathsf{Name} \\ name &: \mathsf{Name} \end{aligned}$$

$$\begin{aligned} Class\text{-}name :: \ mod\text{-}name &: \mathsf{Name} \\ name &: \mathsf{Name} \end{aligned}$$

Acesso aos elementos

Em School, o acesso aos nomes importados é sempre qualificado. A experiência com Modula-2, uma das linguagens que permite importações qualificadas e não qualificadas, mostrou que o resultado do uso de importações não qualificadas é a perda de controle sobre a origem dos nomes, e até mesmo sobre o significado dos nomes. Além disso, a qualificação dos nomes diminui a possibilidade de conflito, já que cada módulo tem seu espaço de nomes. Conflitos ficam limitados à utilização de módulos com nomes iguais, já que o espaço de nomes dos módulos é o global.

No caso de School, a sintaxe para acesso a um nome importado é:

$$modulename \ ' \ identifier$$

Utilizando uma sintaxe abstrata, podemos definir um identificador como um nome qualificado ou um nome local:

$$\begin{aligned} QualifiedName :: \ module &: \mathsf{Name} \\ name &: \mathsf{Name} \end{aligned}$$

$$LocalName :: \mathsf{Name}$$

$$Identifier = LocalName \mid QualifiedName$$

A denotação de um identificador é uma função que, dado um ambiente, retorna o elemento correspondente. No caso do nome ser qualificado, o elemento é obtido do módulo correspondente. Se o nome for local, a busca é feita no ambiente do módulo corrente:

$$
\begin{array}{l}
ident : Identifier \rightarrow (Env \rightarrow Elem) \\
ident(id) \quad \triangleq \quad \lambda env \cdot \\
\quad \textsf{cases } id \textsf{ of} \\
\qquad\quad mk\text{-}LocalName(n) \rightarrow retrieve\text{-}name(env, mod\text{-}name(env), n) \\
\quad mk\text{-}QualifiedName(m, n) \rightarrow retrieve\text{-}name(env, m, n) \\
\quad \textsf{end}
\end{array}
$$

$$
\begin{array}{l}
retrieve\text{-}name : Env \times \textsf{Name} \times \textsf{Name} \rightarrow Elem \\
retrieve\text{-}name(mk\text{-}Env(m, mod), mod\text{-}name, name) \quad \triangleq \\
\quad \textsf{if } mod\text{-}name \in \textsf{dom } mod \\
\quad \textsf{then let } m = mod(mod\text{-}name) \textsf{ in} \\
\qquad \textsf{if } name \in \textsf{dom } (m) \\
\qquad \textsf{then } elem(m(name)) \\
\qquad \textsf{else } error \\
\quad \textsf{else } error
\end{array}
$$

Inicialização do módulo

Em School, cada módulo tem um construtor, que é um procedimento especial sem parâmetros para inicialização do módulo. A linguagem garante que um módulo cliente só é inicializado após todos os módulos importados terem sido inicializados. Esta garantia implica na impossibilidade de relações de dependências cíclicas entre os módulos. Se módulos são subsistemas independentes, a importação é apenas um mecanismo para a utilização dos serviços de outro módulo. Se existe uma importação mútua, significa que estes módulos não são independentes, e portanto, seria mais natural que estes pertencessem a um mesmo subsistema. Neste caso, estes módulos não deveriam estar separados, mas sim ser um único módulo. É interessante observar que a facilidade de tipos e classes genéricos de School facilita a independência entre módulos. Por exemplo, uma classe Lista implementada sem esta facilidade necessariamente precisa incluir um módulo que descreve o tipo a ser armazenado. Muitas vezes este tipo é descrito no módulo que utiliza a lista, o que causaria a dependência mútua.

Comunicação entre os módulos

Ainda não foi explicado como os identificadores importados são inseridos no ambiente dos módulos clientes, o ambiente global de um módulo não é suficiente para descrever a comunicação entre os módulos.

Os módulos estão contidos em um ambiente externo, de onde deve ser possível obter informações para a importação. Da mesma maneira, a denotação de um módulo deve descrever quais são as informações deste módulo presentes no ambiente externo. Externamente, um módulo não é representado pelo seu ambiente global. Do ponto de vista da comunicação entre os módulos, apenas as declarações locais exportadas são relevantes.

Para representar um módulo neste ambiente externo, é introduzido um novo tipo composto pelo nome do módulo e pelas suas declarações exportadas:

$$\begin{array}{lll} \textit{Module} :: & \textit{name} & : \mathsf{Name} \\ & \textit{elems} & : \textit{Mod-env} \end{array}$$

where

$$\textit{inv-Module}(\textit{mk-Module}(n, e)) \quad \triangleq \quad n \notin \mathsf{dom}\, e \wedge \forall x \in \mathsf{rng}\, e \cdot \neg\, \textit{private}(x)$$

O invariante garante que não existe nenhuma declaração com o mesmo nome do módulo e que todas as suas declarações não são privadas.

A partir desta representação de um módulo, podemos definir o ambiente externo como um mapeamento de nomes em módulos:

$$\textit{Ext-env} = \mathsf{Name} \xrightarrow{m} \textit{Module}$$

A cláusula de importação deve aparecer antes de qualquer declaração. Esta restrição facilita a inspeção das relações de dependência do módulo, deixando claro quais módulos são utilizados. A importação é indicada com a palavra reservada `include`:

`include` $\textit{name}_1$; $\textit{name}_2$; ...

onde $\textit{name}_i$ são os nomes dos módulos importados.

A sintaxe abstrata da descrição de um módulo é a seguinte:

$$\begin{array}{lrl} \textit{ModuleDecl} :: & \textit{name} & : \mathsf{Name} \\ & \textit{includes} & : \textit{IncludeClause} \\ & \textit{decls} & : \textit{ElemDecl}^* \\ & \textit{constructor} & : \textit{Constructor} \end{array}$$

$$\textit{IncludeClause} :: \textit{mod-names} : \mathsf{Name\text{-}set}$$

A denotação de um módulo pode ser definida como uma função que tem como entrada um ambiente externo e retorna a representação do módulo:

$$\textit{createModule} : \textit{ModuleDecl} \rightarrow (\textit{Ext-env} \rightarrow \textit{Module})$$

$$\begin{array}{l} \textit{createModule}(\textit{mk-ModuleDecl}(\textit{name}, \textit{includes}, \textit{decls}, \textit{constr})) \quad \triangleq \quad \lambda \textit{ext} \cdot \\ \quad \mathsf{let}\ \textit{env}_1 = \textit{mk-Env}(\textit{name}, \{\textit{name} \mapsto \{\,\}\})\ \mathsf{in} \\ \quad \mathsf{let}\ \textit{env}_2 = \textit{include-clause}(\textit{mod-includes})\textit{ext}\ \ \textit{env}_1\ \mathsf{in} \\ \quad \mathsf{let}\ \textit{env}_3 = \textit{decl-sequence}(\textit{decls}, \textit{env}_2)\ \mathsf{in} \\ \quad \mathsf{let}\ \textit{locals} = \textit{modules}(\textit{env}_3)(\textit{name})\ \mathsf{in} \\ \quad \mathsf{let}\ \textit{exportable-decls} = \\ \qquad \{x \in \mathsf{dom}\, \textit{locals} \mid \neg\, \textit{private}(\textit{locals}(x))\} \lhd \textit{locals}\ \mathsf{in} \\ \quad \textit{mk-Module}(\textit{name}(\textit{env}), \textit{exportable-decls}) \end{array}$$

$$\textit{decl-sequence} : \textit{ElemDecl}^* \times \textit{Env} \rightarrow \textit{Env}$$

$$\begin{array}{l} \textit{decl-sequence}(\textit{seq}, \textit{env}) \quad \triangleq \\ \quad \mathsf{if}\ \textit{seq} = [\,] \\ \quad \mathsf{then}\ \textit{env} \\ \quad \mathsf{else}\ \textit{decl-sequence}(\mathsf{tl}\ \textit{seq}, \textit{elem-decl}(\mathsf{hd}\ \textit{seq})\textit{env}) \end{array}$$

Na função *createModule* acima, o símbolo env_1 é associado ao ambiente inicial. Este ambiente inicial é a entrada para a função semântica de inclusão de módulos, que retorna um novo ambiente associado a env_2, já com todas as declarações de módulos importados e tipos referenciados por estes. Para inserir as declarações locais no ambiente é usada uma função auxiliar — *decl-sequence* — que para cada elemento da sequência de declarações aplica a função semântica *elem-decl*. A partir deste ambiente global, as declarações locais não privadas são associadas ao símbolo *exportable-decls*, que é utilizado na criação da representação do módulo. O construtor não é relevante para os aspectos analisados.

A denotação da inclusão de módulos é feita pela função *include-clause*. Quando um módulo é incluído, o compilador lê a IMC que o descreve, e todos os nomes exportados pelo módulo passam a ser acessíveis no cliente. O retorno é um ambiente com todas as declarações dos módulos incluídos.

$$include\text{-}clause : IncludeClause \rightarrow (Ext\text{-}env \times Env \rightarrow Env)$$

$$\begin{array}{l} include\text{-}clause(mk\text{-}IncludeClause(names)) \quad \triangleq \\ \qquad \lambda ext, env \cdot include\text{-}modules(names, ext, env) \end{array}$$

A inclusão dos módulos é feita por uma função auxiliar, *include-modules*, definida na seção 4.2.1. Esta função retorna um novo ambiente que insere todas as declarações exportadas por um determinado conjunto de módulos.

A única possibilidade de criação de tipos opacos é via inclusão de módulos. Esta criação é tratada exatamente pela função *include-modules*. As descrições de módulos obtidas do ambiente externo podem conter referências a tipos cuja declaração não faz parte de nenhuma descrição incluída. A função *include-modules* insere todas as declarações presentes no ambiente externo e, para cada tipo referenciado por algum módulo cuja declaração não está no ambiente, cria um *Opaque-elem* correspondente.

A partir das definições acima é possível descrever como o ambiente externo é construído. Como não existe importação mútua, as relações de dependência de um sistema constituído dos módulos M_1, M_2, ..., M_n podem ser representadas por um grafo acíclico. Definindo como $import(M)$ o conjunto de módulos importados por M é possível construir uma sequência onde os módulos estão ordenados de maneira que o i-ésimo módulo na sequência não depende de nenhum módulo anterior a ele:

$$\begin{array}{l} ORDER\text{-}MODULES\ (mods\colon ModuleDecl\text{-}\mathsf{set})\ seq\colon ModuleDecl^* \\ \mathsf{post}\ \mathsf{len}\, seq = \mathsf{card}\, mods \wedge \\ \qquad \mathsf{rng}\, seq = mods \wedge \\ \qquad \forall i, j \in \mathsf{dom}\, seq \cdot i < j \;\Rightarrow\; seq(i) \notin import(seq(j)) \end{array}$$

O ambiente externo pode ser construído a partir desta sequência:

$$make\text{-}env : ModuleDecl^* \rightarrow Ext\text{-}env$$

$$\begin{array}{lll} make\text{-}env(seq) & \triangleq & \mathsf{if}\ seq = [\,] \\ & & \mathsf{then}\ \{\,\} \\ & & \mathsf{else}\ \mathsf{let}\ e = make\text{-}env(\mathsf{tl}\, seq)\ \mathsf{in} \\ & & \qquad e \dagger \{\mathsf{hd}\, seq \mapsto createModule(\mathsf{hd}\, seq) e\} \end{array}$$

Vale a pena observar que a ordem definida pela operação *ORDER-MODULES* surge naturalmente durante a construção da aplicação.

Elementos de School

As seções anteriores fizeram uso das funções *include-modules* e *ref-types* sem que estas tenham sido definidas. Para estas definições, é preciso fornecer a representação de cada um dos elementos da linguagem, ou seja, dos tipos, classes, procedimentos, valores e variáveis.

Todos esses elementos fazem referência a tipos; variáveis têm um tipo associado, procedimentos têm uma assinatura etc. Estas referências são representadas pelo nome do tipo, pelo nome da classe (no caso de tipos de classes) ou por um número (no caso de um tipo-parâmetro). *Elem-type* representa essa referência:

$$Elem\text{-}type = Type\text{-}name \mid Class\text{-}name \mid \mathbf{N}_1$$

Um valor pode ser representado pelo seu nome e pelo seu tipo. Como um valor não pode ser parametrizado, o invariante garante que o seu tipo não é um número:

$$\begin{aligned} Value :: \; name &: \mathsf{Name} \\ type &: Elem\text{-}type \end{aligned}$$

where

$$inv\text{-}Value(mk\text{-}Value(n,t)) \quad \triangleq \quad t \notin \mathbf{N}_1$$

A representação de uma variável pode ser a mesma de um valor:

$$\begin{aligned} Var :: \; name &: \mathsf{Name} \\ type &: Elem\text{-}type \end{aligned}$$

where

$$inv\text{-}Var(mk\text{-}Var(n,t)) \quad \triangleq \quad t \notin \mathbf{N}_1$$

Para representar os outros elementos é necessário definir o que é uma assinatura:

$$\begin{aligned} Signature :: \; params &: Elem\text{-}type^* \\ results &: Elem\text{-}type^* \end{aligned}$$

Agora pode-se definir um tipo. Ele é composto do seu nome, do número de parâmetros e da sua interface. O invariante de *Type* diz que todos os tipo-parâmetros referenciados nas assinaturas da interface são válidos:

$$\begin{aligned} Type :: \quad name &: Type\text{-}name \\ n\text{-}params &: \mathbf{N}_1 \\ interface &: \mathsf{Name} \xrightarrow{m} Signature \end{aligned}$$

where

$$\begin{aligned} & inv\text{-}Type(mk\text{-}Type(name, np, i)) \quad \triangleq \\ & \quad \forall mk\text{-}Signature(p, r) \in \mathsf{rng}\, i \cdot \\ & \qquad \forall type \in (\mathsf{rng}\, p \cup \mathsf{rng}\, r) \cdot type \in \mathbf{N}_1 \;\; \Rightarrow \;\; type \leq np \end{aligned}$$

Antes das definições de procedimento e classe, é preciso definir uma cláusula *where*, usada para impor restrições aos tipos genéricos. Um mapeamento de nomes em assinaturas pode ser usado:

$$\mathit{Where\text{-}clause} = \mathsf{Name} \xrightarrow{m} \mathit{Signature}$$

Um procedimento pode ser representado pelo seu nome, pelo número de parâmetros, pelas cláusulas *where* e pela sua assinatura. O invariante garante que todos os tipos-parâmetros referenciados na sua assinatura e nas cláusulas *where* são válidos e que existe uma cláusula para cada parâmetro.

$$\begin{array}{lrl} \mathit{Proc} :: & \mathit{name} & : \mathsf{Name} \\ & \mathit{n\text{-}params} & : \mathbf{N}_1 \\ & \mathit{where\text{-}clause} & : \mathit{Where\text{-}clause}^* \\ & \mathit{signature} & : \mathit{Signature} \end{array}$$

where

$$\begin{array}{l} \mathit{inv\text{-}Proc}(\mathit{mk\text{-}proc}(\mathit{name}, \mathit{np}, \mathit{wc}, \mathit{mk\text{-}Signature}(p, r))) \quad \triangleq \\ \quad (\mathit{np} = \mathsf{len}\ \mathit{wc}) \wedge \\ \quad (\forall \mathit{type} \in (\mathsf{rng}\ p \cup \mathsf{rng}\ r) \cdot \mathit{type} \in \mathbf{N}_1 \ \Rightarrow\ \mathit{type} \leq \mathit{np}) \wedge \\ \quad (\forall c \in \mathsf{rng}\ \mathit{wc} \cdot \\ \quad\quad \forall \mathit{mk\text{-}Signature}(p, r) \in \mathsf{rng}\ c \cdot \\ \quad\quad\quad \forall \mathit{type} \in (\mathsf{rng}\ p \cup \mathsf{rng}\ r) \cdot \mathit{type} \in \mathbf{N}_1 \ \Rightarrow\ \mathit{type} \leq \mathit{np}) \end{array}$$

A classe é composta pelo seu nome, pelo número de parâmetros, pelas cláusulas *where*, pelos valores, variáveis e procedimentos. Assim como no procedimento, o invariante garante que todos os tipo-parâmetros referenciados na sua assinatura e nas cláusulas *where* são válidos e que existe uma cláusula para cada parâmetro.

$$\begin{array}{lrl} \mathit{Class} :: & \mathit{name} & : \mathit{Class\text{-}name} \\ & \mathit{n\text{-}params} & : \mathbf{N}_1 \\ & \mathit{where\text{-}clause} & : \mathit{Where\text{-}clause}^* \\ & \mathit{values} & : \mathsf{Name} \xrightarrow{m} \mathit{Elem\text{-}type} \\ & \mathit{vars} & : \mathsf{Name} \xrightarrow{m} \mathit{Elem\text{-}type} \\ & \mathit{procs} & : \mathsf{Name} \xrightarrow{m} \mathit{Signature} \end{array}$$

where

$$\begin{array}{l} \mathit{inv\text{-}Class}(\mathit{mk\text{-}Class}(\mathit{name}, \mathit{np}, \mathit{wc}, \mathit{val}, \mathit{var}, \mathit{proc})) \quad \triangleq \\ \quad (\mathit{np} = \mathsf{len}\ \mathit{wc}) \wedge \\ \quad (\forall \mathit{type} \in \mathsf{rng}\ \mathit{val} \cup \mathsf{rng}\ \mathit{var} \cdot \mathit{type} \in \mathbf{N}_1 \ \Rightarrow\ \mathit{type} \leq \mathit{np}) \wedge \\ \quad (\forall c \in \mathsf{rng}\ \mathit{wc} \cdot \\ \quad\quad \forall \mathit{mk\text{-}Signature}(p, r) \in \mathsf{rng}\ c \cdot \\ \quad\quad\quad \forall \mathit{type} \in (\mathsf{rng}\ p \cup \mathsf{rng}\ r) \cdot \mathit{type} \in \mathbf{N}_1 \ \Rightarrow\ \mathit{type} \leq \mathit{np}) \end{array}$$

Definidos os elementos, é possível agora definir as funções *ref-types* e *include-modules*.

A função *ref-types* retorna um conjunto com todos os tipos referenciados por um determinado ambiente de módulos. O conjunto é montado a partir dos tipos de valores e variáveis globais e de classes e dos tipos referenciados nas assinaturas presentes nos procedimentos globais e de classes e nas interfaces dos tipos.

$$\mathit{Elem\text{-}name} = \mathit{Type\text{-}name} \mid \mathit{Class\text{-}name}$$

$$
\begin{array}{l}
\mathit{ref\text{-}types} : \mathit{Mod\text{-}env} \times \mathit{Elem\text{-}name}\text{-}\mathsf{set} \\[1ex]
\mathit{ref\text{-}types}(\mathit{env}) \quad \underline{\triangle} \\
\quad \mathsf{let}\ \mathit{var\text{-}value\text{-}types} = \\
\qquad \{\mathit{type}(x) \mid x \in \mathsf{rng}\, \mathit{env} \land x \in \mathit{Value}\} \cup \\
\qquad \{\mathsf{rng}\, \mathit{vars}(x) \mid x \in \mathsf{rng}\, \mathit{env} \land x \in \mathit{Class}\} \cup \\
\qquad \{\mathsf{rng}\, \mathit{values}(x) \mid x \in \mathsf{rng}\, \mathit{env} \land x \in \mathit{Class}\}\ \mathsf{in} \\
\quad \mathsf{let}\ \mathit{where\text{-}clauses} = \\
\qquad \{\mathsf{rng}\, \mathit{where\text{-}clause}(x) \mid x \in \mathsf{rng}\, \mathit{env} \land (x \in \mathit{Proc} \lor x \in \mathit{Class})\}\ \mathsf{in} \\
\quad \mathsf{let}\ \mathit{signatures} = \\
\qquad \{\mathsf{rng}\, \mathit{interface}(x) \mid x \in \mathsf{rng}\, \mathit{env} \land x \in \mathit{Type}\} \cup \\
\qquad \{\mathit{sig}(x) \mid x \in \mathsf{rng}\, \mathit{env} \land x \in \mathit{Proc}\} \cup \\
\qquad \{\mathsf{rng}\, x \mid x \in \mathit{where\text{-}clauses}\} \cup \\
\qquad \{\mathsf{rng}\, \mathit{procs}(x) \mid x \in \mathsf{rng}\, \mathit{env} \land x \in \mathit{Class}\}\ \mathsf{in} \\
\quad \mathsf{let}\ \mathit{sig\text{-}types} = \\
\qquad \{\mathsf{rng}\, \mathit{params}(x) \mid x \in \mathit{signatures}\} \cup \\
\qquad \{\mathsf{rng}\, \mathit{results}(x) \mid x \in \mathit{signatures}\}\ \mathsf{in} \\
\quad \{x \in (\mathit{sig\text{-}types} \cup \mathit{var\text{-}value\text{-}types}) \mid x \notin \mathbf{N}_1\}
\end{array}
$$

A função *include-modules* retorna um novo ambiente que insere todas as declarações exportadas por um determinado conjunto de módulos. O novo ambiente conterá ainda um elemento opaco para cada tipo referenciado não definido:

$$
\begin{array}{l}
\mathit{include\text{-}modules} : \mathsf{Name\text{-}set} \times \mathit{Ext\text{-}env} \times \mathit{Env} \to \mathit{Env} \\[1ex]
\mathit{include\text{-}modules}(\mathit{names}, \mathit{ext}, \mathit{env}) \quad \underline{\triangle} \\
\quad \mathsf{if}\ \mathit{name}(\mathit{env}) \notin \mathit{names} \\
\quad \mathsf{then}\ \mathsf{let}\ \mathit{included\text{-}module\text{-}envs} = \\
\qquad\qquad \{\mathit{mod} \mapsto \mathit{elems}(\mathit{ext}(\mathit{mod})) \mid \mathit{mod} \in \mathit{names}\}\ \mathsf{in} \\
\qquad \mathsf{let}\ \mathit{types} = \\
\qquad\qquad \{x \in \mathit{ref\text{-}types}(x) \mid x \in \mathsf{rng}\, \mathit{included\text{-}module\text{-}envs}\}\ \mathsf{in} \\
\qquad \mathsf{if}\ \nexists x \cdot \in \mathit{types} \mid \mathit{mod\text{-}name}(x) = \mathit{name}(\mathit{env}) \\
\qquad \mathsf{then}\ \mathsf{let}\ \mathit{opaque\text{-}only\text{-}module\text{-}names} = \\
\qquad\qquad\quad \{\mathit{mod\text{-}name}(x) \mid x \in \mathit{types} \land \mathit{mod\text{-}name}(x) \notin \mathit{names}\}\ \mathsf{in} \\
\qquad\quad \mathsf{let}\ \mathit{empty\text{-}opaque\text{-}only\text{-}module\text{-}envs} = \\
\qquad\qquad\quad \{\mathit{mod} \mapsto \{\,\} \mid \mathit{mod} \in \mathit{new\text{-}module\text{-}names}\}\ \mathsf{in} \\
\qquad\quad \mathsf{let}\ \mathit{envs\text{-}without\text{-}opaques} = \\
\qquad\qquad\quad \mathit{included\text{-}module\text{-}envs} \cup \mathit{empty\text{-}opaque\text{-}only\text{-}module\text{-}envs}\ \mathsf{in} \\
\qquad\quad \mathsf{let}\ \mathit{new\text{-}module\text{-}envs} = \{\mathit{mod} \mapsto \\
\qquad\qquad\quad \mathit{insert\text{-}opaques}(\mathit{mod}, \mathit{envs\text{-}without\text{-}opaques}(\mathit{mod}), \mathit{types}) \mid \\
\qquad\qquad\qquad \mathit{mod} \in \mathsf{dom}\, \mathit{envs\text{-}without\text{-}opaques}\}\ \mathsf{in} \\
\qquad\quad \mathit{mk\text{-}Env}(\mathit{name}(\mathit{env}), \mathit{new\text{-}module\text{-}envs}) \\
\qquad \mathsf{else}\ \mathit{error} \\
\quad \mathsf{else}\ \mathit{error}
\end{array}
$$

Para definir a função foi usada uma outra função auxiliar — *insert-opaques* — que, dados o nome do módulo, o ambiente do módulo e um conjunto com todos os tipos referenciados, retorna um novo ambiente de módulo adicionado com elementos opacos para cada tipo referenciado não presente no ambiente original:

$$
\begin{aligned}
&insert\text{-}opaques : \mathsf{Name} \times Mod\text{-}env \times Elem\text{-}name\text{-}\mathsf{set} \rightarrow Mod\text{-}env \\
&insert\text{-}opaques(name, mod\text{-}env, types) \quad \triangleq \\
&\quad \mathsf{let}\ mod\text{-}opaques = \\
&\qquad \{x \in types \mid mod\text{-}name(x) = n \wedge name(x) \notin \mathsf{dom}\ mod\text{-}env\}\ \mathsf{in} \\
&\quad mod\text{-}env \cup \{name(x) \mapsto mk\text{-}Decl(x, false) \mid x \in mod\text{-}opaques\}
\end{aligned}
$$

4.2.2 Implementação

O mecanismo descrito foi implementado no compilador de School. A compilação de um módulo gera um arquivo descrevendo o seu ambiente exportado (extensão `.sym`). A importação de um módulo depende deste arquivo, que é lido para incluir seus elementos na tabela de símbolos. O processo de compilação de uma aplicação School com dois módulos é exemplificado na figura 4.1.

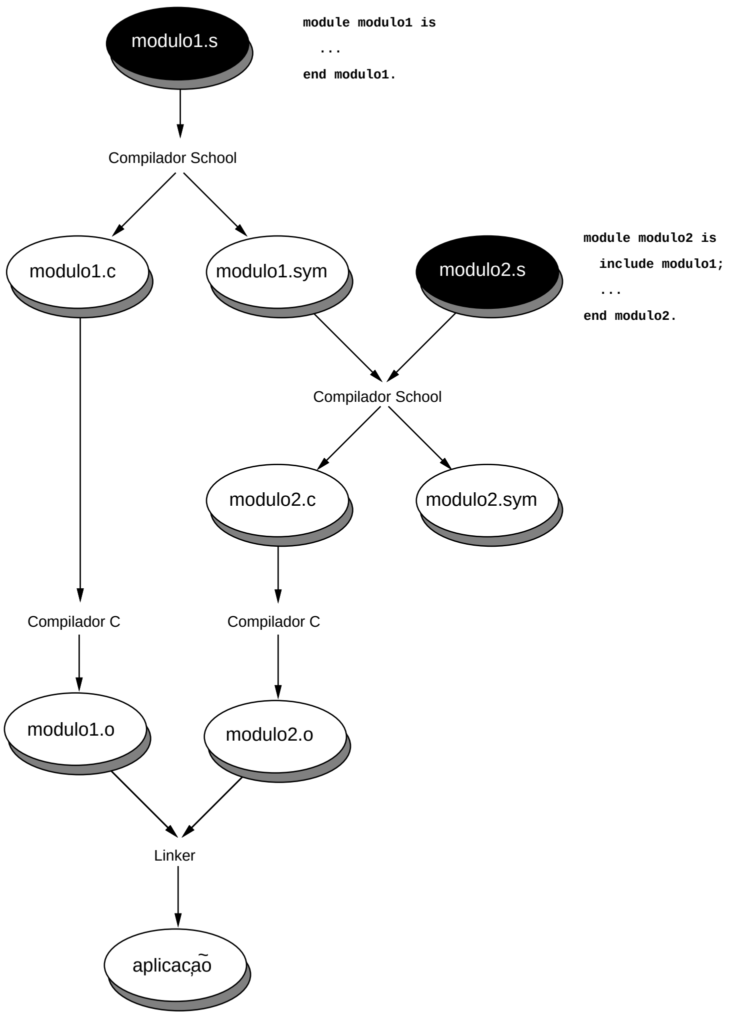

Figura 4.1: Geração de uma aplicação School

Usando o mecanismo de importação, o compilador foi modificado para que os elementos pré-definidos fossem lidos como de um módulo importado. Os tipos pré-definidos, como `Bool`, `Int` e `Real`, estão em um arquivo que descreve um ambiente exportado —

`PREDEF.sym` —, lido antes da compilação de todo módulo School. Assim, o compilador deixou de codificar as declarações internamente, além de facilitar possíveis extensões de elementos pré-definidos. Para possibilitar esta utilização do arquivo de símbolos, a implementação foi estendida para permitir a descrição de tipos primitivos, como `Int` e `Real`.

4.3 Amarração dinâmica

A implementação de amarração dinâmica em School é feita com tabelas hash. Cada objeto tem um ponteiro para uma descrição da sua classe, que por sua vez contém a tabela hash que associa os nomes dos métodos às suas implementações. Como a tabela é construída em tempo de compilação, ela pode ser grande o suficiente para que a probabilidade de colisão seja pequena.

Como foi visto na seção 4.1, qualquer objeto School tem como primeiro campo na sua representação um ponteiro para a sua classe. Como apenas a classe é utilizada para definir o método a ser chamado em um envio de mensagem, não é necessário conhecer a estrutura completa do objeto para implementar a amarração dinâmica. Isto permite que a amarração dinâmica possa ser feita da mesma forma para todos os objetos.

A função `dispatch` faz a busca na tabela hash da classe, retornando o método a ser chamado. Esta função recebe o objeto receptor da mensagem, uma string com o nome da mensagem e um inteiro que representa a chave correspondente à string na tabela hash, calculada durante a compilação. Com isso, uma chamada `obj.message` é traduzida para:

```
dispatch(p_obj, "message", 27417))(p_obj);
```

Quando uma variável é declarada com o tipo de uma classe (`#`*ClassName*), ela só poderá referenciar objetos desta classe. Nesse caso, os envio de mensagens a estas variáveis podem ser amarrados estaticamente.

4.3.1 Eficiência

Enquanto uma amarração estática é codificada como uma simples chamada de função, a amarração dinâmica exige a execução de uma outra função — `dispatch` — antes de realizar o desvio para a função que implementa o método. Na prática a diferença é que, em vez de uma única instrução de desvio, a amarração dinâmica executa todas as instruções da função `dispatch`, além do desvio. Compilado em uma Sparc, no melhor caso a amarração dinâmica executa 22 instruções de máquina para encontrar o método a ser chamado.

Usando o código da figura 4.2 para medir a diferença entre as amarrações estática e dinâmica, o resultado variou muito de acordo com a máquina — usando o mesmo compilador com os mesmos flags. Em uma Sparc, ela é em média 22 vezes mais lenta do que uma amarração estática. Já em uma estação Indy, a amarração dinâmica foi 6.7 vezes mais lenta.

Por ter uma amarração dinâmica cara, técnicas de otimização poderiam melhorar muito a eficiência de aplicações School. Quase todas as técnicas analisadas na seção 3 podem ser aplicadas sem necessidades de adaptações. A exceção é a de previsão de tipos, já que esta técnica tenta mapear algumas operações (por exemplo somas) para as nativas da máquina testando o tipo do objeto, o que não faz sentido sobre School por a

```
constructor is
  var i : Int := 0;
  var j : Int := 0;
  var o : A := C!;
  -- variável do tipo A,
  -- amarração dinâmica

  while i < 5000000 loop
    j := j + i - o.f;
    i := i + 1;
  end;
end;
```

```
constructor is
  var i : Int := 0;
  var j : Int := 0;
  var o : #C := C!;
  -- variável do tipo #C,
  -- amarração estática

  while i < 5000000 loop
    j := j + i - o.f;
    i := i + 1;
  end;
end;
```

Figura 4.2: Teste de eficiência da amarração dinâmica de School

linguagem oferecer tipos primitivos. Apesar de ser uma linguagem tipada, uma adaptação da inferência de tipos poderia ser usada para obter informações sobre possíveis classes de objetos. A inferência seria feita com o único objetivo de otimização.

4.4 Tipos parametrizados

A implementação atual de School não permite a instanciação de classes e funções parametrizadas com tipos primitivos. A geração de código assume que os parâmetros são objetos não primitivos, e nenhum tratamento diferenciado é necessário, já que o código para manipulação de objetos é o mesmo, independente da classe.

Nenhuma das soluções adotadas em outras linguagens orientadas a objetos (seção 3.3) se adequa bem ao modelo de School. As características que precisam ser levadas em consideração para esta implementação são:

- School permite restrições aos parâmetros,
- o envio de mensagens pode ser feito de forma única em School e
- School tem tipos primitivos.

A primeira característica permite que toda checagem de tipos seja feita localmente independente das instanciações; a segunda permite a geração de um código que pode ser usado com qualquer objeto. No entanto, a terceira característica dificulta as manipulações sobre tipos-parâmetro, já que mensagens não são enviadas para tipos primitivos. A proposta apresentada na seção 3.3 pode ser usada para resolver de forma eficiente este problema. Analisando as *where-clauses* de School, é possível descobrir, para cada classe parametrizada, se algum parâmetro pode ser instanciado com um tipo primitivo. Se for, pode-se gerar um *stub* para cada cláusula, que faz um teste sobre o tipo usado na instanciação e realiza a operação correspondente. Nesse caso os envios de mensagem são desviados para estes *stubs*.

Por exemplo, considere a seguinte classe parametrizada:

```
class C{T}
  where T has
    proc lt (v: T) -> Bool; -- less than
  end;
is
   ...
end C;
```

O parâmetro T pode ser instanciado com qualquer tipo que trate a mensagem `lt` (que pode ser enviada com o operador `<`). Os tipos primitivos `Int` e `Real` possuem esta operacao e portanto podem ser usados como parâmetro, assim como qualquer objeto que defina este método. Nesse caso, o objeto manipulado pela classe `C` pode ser um `int`, um `float` ou um `S_Obj`, e um tipo-parâmetro pode ser traduzido para:

```
union T {
  int Int;
  float Real;
  S_Obj Obj;
};
```

Sempre que a classe envia a mensagem `lt` para o objeto, a execução é desviada para o *stub* correspondente à clausula, que pode ser implementado assim:

```
S_Obj stub_lt(S_Obj self, union T v1, union T v2)
{
  if (self->tipo[1] == IntType)
    return v1.Int < v2.Int ? TrueObj : FalseObj;
  else if (self->tipo[1] == RealType)
    return v1.Real < v2.Real ? TrueObj: FalseObj;
  else
    return dispatch(v1.Obj, "lt")(v1.Obj,v2.Obj);
}
```

Para isso, a estrutura que representa o objeto teria mais um campo — `tipo` no exemplo acima —, um vetor com uma posição para cada tipo parâmetro onde fica armazenada a informação de instanciação, a ser preenchida durante a execução, no ponto onde os objetos são criados. Esta informação pode ser gerada pelo compilador, sendo passada como uma parâmetro adicional para o construtor.

Para a implementação de funções parametrizadas, a informação do tipo real dos parâmetros — que para os métodos fica guardada no próprio objeto — precisa ser passada como parâmetro adicional da função, o que pode ser feito automaticamente pelo compilador de acordo com a instanciação.

Esta implementação requer que um tratamento semelhante seja realizado em funções que recebem como parâmetros tipos que sejam supertipos dos primitivos.

4.5 Modelo de eventos

Com a introdução de suporte a módulos e ligação com C na linguagem School, tornou-se possível o desenvolvimento de aplicações maiores, com a utilização de módulos já prontos.

Uma base desses módulos é fundamental para tornar a linguagem mais utilizada. A orientação desta base, de certo modo, influencia o tipo de aplicação que é desenvolvida com a linguagem. Como aplicações gráfico-interativas são umas das mais beneficiadas com a utilização de School, foi decidido que esta base deve dar suporte a este tipo de aplicação, que são tipicamente orientadas a eventos.

No modelo de orientação a eventos, difundido a partir de Smalltalk, as ações executadas por um programa são ativadas a partir de eventos gerados pelo usuário. Com a popularização dos sistemas de interface gráfica com o usuário, este modelo tornou-se um padrão para o desenvolvimento de aplicações interativas.

Neste modelo a aplicação tem definidas as ações para os eventos que o usuário pode gerar. Como consequência deste paradigma, tem-se que o controle de fluxo sai da aplicação e passa para o usuário.

Linguagens tradicionais, como C ou Pascal, não oferecem mecanismos de suporte para este novo enfoque. Desta forma, torna-se necessária uma simulação desta inversão de controle. Geralmente a simulação é implementada através de um *loop* principal que lê os eventos e os processa. Neste caso, as próprias aplicações implementam este *loop* ou passam explicitamente o controle a uma rotina do sistema que o faz.

A solução adotada para School segue um modelo puro de orientação a eventos, onde inexistem *loop* e programa principal. Isto traz como consequências para a aplicação a perda do conceito de "modo" — uma vez que ela não tem controle sobre qual ação será executada a cada momento. Como não há um programa principal, os objetos são criados nos construtores dos módulos que formam a aplicação, ou simplemente inicializando-se variáveis dos módulos.

Para permitir a construção destas aplicações, foi criado um núcleo a ser embutido nas aplicações. Este núcleo controla os eventos. Como os eventos propriamente ditos não podem ser determinados, já que cada aplicação trata um conjunto de eventos, o núcleo não está amarrado a nenhum evento específico, sendo simplesmente um gerenciador para tratadores de eventos, cada um responsável pelo seu conjunto determinado. Alguns conjuntos típicos são:

- eventos do sistema de janelas,
- eventos de temporizadores e
- eventos de rede.

Cada caso acima corresponderia a um tratador específico, escrito em School. Utilizando este modelo, foram implementados os tratadores de temporizadores, totalmente feito em School, e o do sistema de janelas, implementado em School e C sobre XWindows.

O *loop* principal, inexistente em School, fica embutido no núcleo, que tem como função ativar cada um dos tratadores cadastrados continuamente. Como consequência desta arquitetura, novos eventos podem ser usados em uma aplicação School, bastando para isso construir um tratador para eles, já que o núcleo não fixa nem limita os eventos. Uma arquitetura semelhante é utilizada no ambiente Tche [CI95].

4.5.1 Implementação do núcleo

Para evitar o consumo excessivo de CPU e permitir que o loop termine assim que não houver mais a possibilidade de um evento ser gerado, foi definido um protocolo a ser utilizado na comunicação entre o núcleo e os tratadores. Cada tratador retorna um inteiro representando um tempo em milissegundos para o núcleo, indicando quanto tempo pode passar antes que ele seja novamente ativado (isto não significa que o núcleo vai esperar exatamente este tempo, é um tempo máximo — o tratador pode ser chamado antes). Assim, o valor 0 é retornado quando o tratador precisa ser chamado novamente o mais rápido possível. É o caso típico de quando o tratador retorna e ainda existem eventos na fila prontos para serem tratados. Uma constante — `FINISHED` — é definida como o maior inteiro representável, usada para o tratador indicar ao núcleo que ele não precisa mais ser chamado pois novos eventos não serão gerados. Por exemplo, quando não existir mais nenhuma janela o tratador do sistema de janelas pode retornar `FINISHED`. Quando todos os tratadores retornarem `FINISHED` a aplicação chega ao fim. Valores intermediários entre 0 e `FINISHED` permitem ao núcleo tornar a aplicação *idle*, evitando o consumo desnecessário de CPU.

```
loop
{
   sleep_time := FINISHED;
   for each registered handler
   {
      time := call handler
      if (time < sleep_time) sleep_time := time;
   }
   if sleep_time = FINISHED then exit
   if sleep_time > 0 then sleep_ms(sleep_time);
}
```

Figura 4.3: Núcleo gerenciador dos tratadores de eventos

Uma possível implementação para o núcleo está na figura 4.3. Uma outra alternativa de implementação de uma arquitetura semelhante com tratadores independentes poderia ser feita com a utilização de *threads*.

4.5.2 Tratadores de eventos

Os tratadores de eventos são objetos School. Estes objetos podem ser cadastrados no núcleo, sendo ativados para fazer o processamento dos eventos. Os tratadores devem satisfazer o seguinte tipo:

```
type EventHandler is
   proc handle -> Int;
end EventHandler;
```

O método handle é o chamado pelo núcleo para fazer o tratamento. O registro é feito via uma função do núcleo que recebe como parâmetro um objeto to tipo `EventHandler`.

Tipicamente, um módulo que define um tratador de eventos qualquer importa o módulo `eventhandler`, define uma classe que satisfaz o tipo `EventHandler`, cria um objeto desta classe — que será o tratador — e o cadastra no núcleo. Normalmente o cadastro é feito no construtor do módulo. Desse modo, a simples inclusão deste módulo registra o tratador no módulo automaticamente.

4.6 Coletor de lixo

O coletor de lixo usa o algoritmo clássico de marcar-e-varrer. Ele foi implementado à parte do compilador, utilizando o núcleo desenvolvido para o modelo de eventos (seção 4.5). A ativação do coletor é feita pelo núcleo. Com isso, a implementação fica facilitada pelo fato de o coletor nunca ser chamado de dentro de uma função School: nesse caso não existe pilha e portanto apenas as variáveis globais precisam ser consideradas.

Internamente, o compilador dispõe de um suporte mínimo para a implementação de coletores de lixo. O suporte consiste em: manter uma lista das variáveis e valores globais, que é o conjunto raiz usado na primeira fase do algoritmo; manter uma lista de todos os objetos alocados por School, necessária na segunda fase do algoritmo e ter, para cada classe, uma representação simples da sua estrutura para que os campos dos objetos que são referências para outros objetos sejam identificados e percorridos.

O conjunto raiz é formado durante a inicialização dos módulos. Cada construtor de módulo cadastra suas variáveis e valores globais em uma lista.

As referências para todos os objetos alocados são mantidas encadeando-se cada objeto alocado. Este encadeamento causa um aumento no espaço de cada objeto alocado do tamanho de um ponteiro, mas é inevitável na falta de um gerenciador próprio de memória. Com isso, cada objeto passa a ter um campo (não disponível via School) que aponta para o próximo objeto na lista. Um ponteiro global mantém o primeiro elemento da lista. A cada criação de objeto, este é colocado no início da lista.

Para possibilitar o caminhamento no grafo, identificando nos objetos as referências para outros objetos, cada classe tem uma representação da estrutura dos objetos. Esta representação é feita por uma string que descreve cada atributo da classe — inteiro, real, objeto ou ponteiro C —, permitindo que o coletor localize todas as referências feitas por um objeto qualquer.

A tradução de classes parametrizadas também afeta a coleta de lixo. Em School, a compilação gera uma única versão de uma classe parametrizada, compartilhada por todas as instanciações. Na implementação atual a instanciação com tipos primitivos não é permitida; nesse caso todos os atributos parametrizados da classe (que tem como tipo um parâmetro) serão sempre objetos e podem ser tratados como explicado no parágrafo anterior. Se a implementação de tipos parametrizados permitisse a instanciação com tipos primitivos, a representação da estrutura precisaria diferenciar um atributo parametrizado. O tipo real deste depende da instanciação feita. Para descobri-lo, é preciso consultar também as informações de instanciação do objeto, que precisa de uma representação como a discutida na seção 4.4.

O coletor ainda disponilibiza duas funções, `lock` e `unlock`, para que objetos em uso por um código C não sejam erradamente liberados. Se o código C mantém algum objeto School em suas estruturas, é necessário fazer esta indicação ao coletor pela função `lock`. Assim que o objeto não for mais usado, o coletor deve ser informado pela função `unlock`.

A ativação do coletor se dá quando a quantidade de memória ocupada ultrapassa um determinado limite. O limite vai sendo ajustado durante a execução, procurando adequá-lo à ocupação de memória particular da aplicação.

4.6.1 Coleta incremental

Uma coleta de lixo incremental, além de não impor paradas ao programa, se encaixaria bem na estrutura do núcleo apresentado na seção 4.5; nesse caso o coletor de lixo agiria como um tratador de eventos concorrente aos cadastrados pela aplicação.

Seria preciso usar uma barreira de escrita, cuja implementação altera a geração de código para a atribuição. Como consequência, além de perda no desempenho, existiria uma maior integração entre o código gerado e o coletor. O código gerado passaria a ser dependente do coletor, o que não acontece na implementação atual, não incremental.

Capítulo 5

Conclusões

O trabalho procurou cobrir a implementação de linguagens orientadas a objetos, abordando os pontos característicos destas, como amarração dinâmica e tipos genéricos. Foi feito um estudo das alternativas utilizadas atualmente em diversas linguagens, resultando numa visão unificada e global das várias soluções já propostas para linguagens orientadas a objetos.

A seção que discute alternativas para a amarração dinâmica reune técnicas diretas para a sua otimização e também indiretas, como inferência de tipos, cujo objetivo principal não é a otimização, mas sim garantir a correção dos envios de mensagens dos programas. Muitas das soluções apresentadas vêm de trabalhos relacionados à linguagem SELF, cujos compiladores — em sua quarta geração — são agressivos na tentativa de melhorar as amarrações dinâmicas. Isto se deve à características da linguagem, que não possui o conceito de classes e por isso qualquer operação sobre um objeto deve ser amarrada dinamicamente, inclusive a manipulação de seus atributos. Além de SELF, foram analisadas implementações adotadas em Sather, Cecil, Smalltalk, Trellis e C++, abrangendo vários modelos diferentes.

Algumas das técnicas apresentadas poderiam ser adotadas em School com o objetivo de otimizar os envios de mensagem, já que sua amarração dinâmica é bem menos eficiente do que a estática. Trabalhos futuros podem analisar o benefício das técnicas quando aplicadas em School, propondo uma estratégia a ser adotada em implementações posteriores.

A tradução de tipos genéricos também foi abordada, com uma análise das soluções adotadas em C++, Modula-3 e Theta. Pelas características de School — tipos primitivos, separação das hierarquias de tipos e classes e restrições para tipos genéricos —, estas soluções não se encaixam perfeitamente na linguagem. Sendo assim, foi proposta uma nova técnica, adequada a School, para a implementação de classes genéricas.

A coleta de lixo foi feita de modo que o código gerado fosse minimamente alterado. Em particular, o suporte do compilador para a coleta de lixo não influi em nada no desempenho da aplicação, tudo é feito a partir de informações estáticas. Como resultado, chegou-se a um coletor simples, independente do compilador e que não altera o desempenho da aplicação (enquanto não entra em ação). A desvantagem é que a coleta não pode ser feita de forma incremental, podendo impor paradas ao sistema durante a coleta, e o coletor não pode ser ativado durante a execução de uma função School, já que a pilha não é considerada. Qualquer solução que resolva estes problemas envolve a alteração da geração de código do compilador e torna o compilador dependente do coletor.

Outro ponto abordado — módulos — foi analisado nas linguagens Modula-3, Oberon-

2, Ada e C++. O trabalho apresentou uma proposta de distinção entre dois tipos de interface conceitualmente diferentes – IAM e IMC. Baseando-se nestas novas proposições, foi constatado que linguagens que utilizam descrições separadas para interface e implementação tendem a confundir os conceitos de interfaces. Neste aspecto, linguagens que utilizam uma descrição única do módulo apresentam vantagens operacionais e de legibilidade. Trabalhos futuros podem explorar otimizações que se beneficiam de uma descrição única de módulos. Algumas otimizações extras não seriam possíveis se os módulos fossem descritos em dois arquivos. Em particular, a otimização de funções em linha pode ser feita de forma transparente ao programador.

O sistema de módulos da linguagem School foi apresentado com uma descrição formal, o que possibilitou esclarecer detalhes como o tratamento de importação implícita, negligenciado na maioria das descrições informais de linguagens.

Apêndice A

Exemplo de compilação

Este apêndice apresenta um exemplo de compilação de um programa School. O programa utilizado como exemplo define uma árvore binária simples, procurando cobrir a maioria dos pontos da linguagem. A primeira seção apresenta o código School e a seção seguinte, o código C gerado.

A.1 Código School

```
module apendice is

include print;

-------------------------------------------------
-- Declaracao do tipo container               --
-------------------------------------------------

type container{T} is
  proc insere( T );
  proc busca( T ) -> Bool;
  proc vazio -> Bool;
end container;


-------------------------------------------------
-- Classe para armazenar um elemento da arvore --
-------------------------------------------------

class elem_arvore{T} is
  var valor : T;
  var direita: #elem_arvore{T};
  var esquerda: #elem_arvore{T};
  proc dir( novo : #elem_arvore{T} );
  proc esq( novo : #elem_arvore{T} );
  proc consulta -> (dir : #elem_arvore{T},
                    esq : #elem_arvore{T}, val : T );
```

```
  constructor new(val : T, dir : #elem_arvore{T}, esq: #elem_arvore{T});
end elem_arvore;

proc elem_arvore.dir( novo : #elem_arvore{T} ) is
  direita := novo;
end dir;

proc elem_arvore.esq( novo : #elem_arvore{T} ) is
  esquerda := novo;
end esq;

proc elem_arvore.consulta -> (d : #elem_arvore{T},
                              e : #elem_arvore{T}, v : T ) is
  d := direita;
  e := esquerda;
  v := valor;
end consulta;

constructor elem_arvore.new( v : T, d : #elem_arvore{T},
                                    e: #elem_arvore{T} ) is
  valor := v;
  direita := d;
  esquerda := e;
end new;


---------------------------------------------------------
-- Classe arvorebinaria, satisfazendo o tipo container --
---------------------------------------------------------

class arvorebinaria{T} : container{T}
  where T has
    proc lt( T ) -> Bool; -- lessthan, <
    proc gt( T ) -> Bool; -- greaterthan, >
  end;
is
  var raiz : #elem_arvore{T};
  proc insere( T );
  proc busca( T ) -> Bool;
  proc vazio -> Bool;
  proc insere_interno( T, #elem_arvore{T} );
  proc busca_interna( T, #elem_arvore{T} ) -> Bool;
  constructor new;
end arvorebinaria;

proc arvorebinaria.insere( val : T ) is
  if not raiz then
```

```
    raiz := elem_arvore{T}!(val, nil, nil );
  else
    self.insere_interno( val, raiz );
  end;
end insere;

proc arvorebinaria.insere_interno( v : T, elem : #elem_arvore{T} ) is
  assert( elem );
  var esq : #elem_arvore{T};
  var dir : #elem_arvore{T};
  var val : T;
  elem.consulta() -> (dir, esq, val);
  if v < val then
    if not esq then
      elem.esq( elem_arvore{T}!(v, nil, nil) );
    else
      self.insere_interno( v, esq )
    end;
  elseif  v > val then
    if not dir then
      elem.dir( elem_arvore{T}!(v, nil, nil) );
    else
      self.insere_interno( v, dir )
    end;
  end;
end insere_interno;

proc arvorebinaria.busca( val : T ) -> Bool is
  Result := self.busca_interna( val, raiz );
end busca;

proc arvorebinaria.busca_interna( v : T,
                                  elem : #elem_arvore{T} ) -> Bool is
  if not elem then
    Result := false;
  else
    var esq : #elem_arvore{T};
    var dir : #elem_arvore{T};
    var val : T;
    elem.consulta() -> (dir, esq, val);
    if v < val then
      Result := self.busca_interna( v, esq );
    elseif v > val then
      Result := self.busca_interna( v, dir );
    else
      Result := true;
    end;
```

```
  end;
end busca_interna;

proc arvorebinaria.vazio -> Bool is
  Result := if raiz then true else false end;
end vazio;

constructor arvorebinaria.new is
  raiz := nil;
end new;


-----------------------------------------------
-- Classe que modela um inteiro, para teste  --
-----------------------------------------------

class inteiro is
  var val : Int;
  proc lt( i : #inteiro ) -> Bool;
  proc gt( i : #inteiro ) -> Bool;
  proc valor -> Int;
  constructor new ( v : Int );
end inteiro;

proc inteiro.valor -> Int is
  Result := val;
end valor;

proc inteiro.lt( i : #inteiro ) -> Bool is
  Result := val < i.valor;
end lt;

proc inteiro.gt( i : #inteiro ) -> Bool is
  Result := val > i.valor;
end gt;

constructor inteiro.new( i : Int ) is
  val := i;
end new;


----------------------------------------------------
-- O construtor insere alguns elementos na arvore --
-- e depois procura quais foram esse elementos    --
----------------------------------------------------

constructor is
```

```
  var bt : container{#inteiro} := arvorebinaria{#inteiro}!;
  bt.insere( inteiro!(10) );
  bt.insere( inteiro!(1) );
  bt.insere( inteiro!(20) );
  bt.insere( inteiro!(15) );
  bt.insere( inteiro!(18) );
  bt.insere( inteiro!(5) );
  bt.insere( inteiro!(30) );
  var i : Int := 0;
  while i < 40  loop
    if bt.busca( inteiro!(i) ) then
      print'int(i);
    end;
    i := i + 1;
  end;
end;

end apendice.
```

A.2 Código C

Esta seção apresenta o código C gerado pelo compilador a partir do programa School da seção anterior. É interessante notar que as declarações de tipos não geram código, já que elas fornecem informações exclusivamente para a checagem de tipos, feita totalmente durante a compilação.

Inicialização

Toda saída gerada pelo compilador inclui o arquivo `school.h`, que contém as definições utilizadas pela linguagem:

```
#include "school.h"
```

Cláusula include

A cláusula `include` insere no arquivo todas as declarações exportadas pelo módulo incluído, no caso o módulo `print`:

```
void __print();
void print_int3();
void print_real4();
void print_bool4();
void print_char4();
void print_nl2();
```

Classe elem_arvore

A declaração da classe `elem_arvore` gera estruturas em C que modelam um objeto da classe e uma variável que representa a própria classe, com a tabela para a amarração dinâmica e informações para o coletor de lixo:

```
struct apendice_elem_arvore11
{ struct Class *clss;
  S_Obj valor;
  S_Obj direita;
  S_Obj esquerda;
}
;
void apendice_elem_arvore11_dir();
void apendice_elem_arvore11_esq();
S_Obj apendice_elem_arvore11_consulta();
static procType pr_elem_arvore[] = {
  0, (procType)apendice_elem_arvore11_consulta,
  0, (procType)apendice_elem_arvore11_dir,
  (procType)apendice_elem_arvore11_esq};
static char *pn_elem_arvore[] = {"", "consulta", "", "dir", "esq"};
struct Class apendice_elem_arvore11 = {
  5, "ooo", pr_elem_arvore, pn_elem_arvore};
```

Os métodos da classe estão a seguir:

```
void apendice_elem_arvore11_dir(self, p_novo)
struct apendice_elem_arvore11 *self;
S_Obj p_novo;
{ self->direita = p_novo;
  return;
}

void apendice_elem_arvore11_esq(self, p_novo)
struct apendice_elem_arvore11 *self;
S_Obj p_novo;
{ self->esquerda = p_novo;
  return;
}

S_Obj apendice_elem_arvore11_consulta(self, rr_e, rr_v)
struct apendice_elem_arvore11 *self;
S_Obj *rr_e;
S_Obj *rr_v;
{ S_Obj r_d;
  S_Obj r_e;
  S_Obj r_v;
  { r_d = self->direita;
```

```
    r_e = self->esquerda;
    r_v = self->valor;
  }
  { *rr_e = r_e;*rr_v = r_v;return r_d;}
}

struct apendice_elem_arvore11 *apendice_elem_arvore11_new
  (self, p_v, p_d, p_e)
struct apendice_elem_arvore11 *self;
S_Obj p_v;
S_Obj p_d;
S_Obj p_e;
{ if (!self){ self = NEW(apendice_elem_arvore11);
    self->clss = &apendice_elem_arvore11;
  }
  ;
  { self->valor = p_v;
    self->direita = p_d;
    self->esquerda = p_e;
  }
  return self;
}
```

Classe arvorebinaria

```
struct apendice_arvorebinaria13
{ struct Class *clss;
  S_Obj raiz;
}
;
void apendice_arvorebinaria13_insere();
S_Obj apendice_arvorebinaria13_busca();
S_Obj apendice_arvorebinaria13_vazio();
void apendice_arvorebinaria13_insere_interno();
S_Obj apendice_arvorebinaria13_busca_interna();
static procType pr_arvorebinaria[] = {
  (procType)apendice_arvorebinaria13_busca_interna,
  0, 0, 0, (procType)apendice_arvorebinaria13_insere_interno,
  (procType)apendice_arvorebinaria13_vazio,
  (procType)apendice_arvorebinaria13_insere,
  (procType)apendice_arvorebinaria13_busca};
static char *pn_arvorebinaria[] = {
  "busca_interna", "", "", "", "insere_interno",
  "vazio", "insere", "busca"};
struct Class apendice_arvorebinaria13 = {
  8, "o", pr_arvorebinaria, pn_arvorebinaria};
void apendice_arvorebinaria13_insere(self, p_val)
```

```
struct apendice_arvorebinaria13 *self;
S_Obj p_val;
{ if (((self->raiz != S_nilObj)?0:1))
  self->raiz = (S_Obj)apendice_elem_arvore11_new(0, p_val,
    S_nilObj, S_nilObj);
  else ((int (*)())DISPATCH(((S_Obj)self), "insere_interno", 55796))
       (((S_Obj)self), p_val, self->raiz);
  return;
}

void apendice_arvorebinaria13_insere_interno(self, p_v, p_elem)
struct apendice_arvorebinaria13 *self;
S_Obj p_v;
S_Obj p_elem;
{ { if(!(p_elem != S_nilObj)){
      printf("assertion failed in module apendice"
             "(procedure arvorebinaria.insere_interno, line 89)\n");
      abort();
    }
    else;
    { S_Obj l_esq;
      { S_Obj l_dir;
        { S_Obj l_val;
          l_dir = apendice_elem_arvore11_consulta(p_elem,&l_esq,&l_val);
          if ((((S_Obj (*)())DISPATCH(p_v, "lt", 8288))(p_v, l_val) !=
              S_nilObj))
          if (((l_esq != S_nilObj)?0:1))
          apendice_elem_arvore11_esq(p_elem,
            (S_Obj)apendice_elem_arvore11_new(0,p_v,S_nilObj,S_nilObj));
          else ((int (*)())DISPATCH(((S_Obj)self), "insere_interno",
                55796))(((S_Obj)self), p_v, l_esq);
          else if ((((S_Obj (*)())DISPATCH(p_v, "gt", 8103))
                   (p_v, l_val) != S_nilObj))
          if (((l_dir != S_nilObj)?0:1))
          apendice_elem_arvore11_dir(p_elem,
            (S_Obj)apendice_elem_arvore11_new(0,p_v,S_nilObj,S_nilObj));
          else ((int (*)())DISPATCH(((S_Obj)self), "insere_interno",
                55796))(((S_Obj)self), p_v, l_dir);
          else ;
        }
      }
    }
  }
  return;
}

S_Obj apendice_arvorebinaria13_busca(self, p_val)
```

```
struct apendice_arvorebinaria13 *self;
S_Obj p_val;
{ S_Obj r_Result;
  r_Result = ((S_Obj (*)())DISPATCH(((S_Obj)self),
    "busca_interna", 50838))(((S_Obj)self), p_val, self->raiz);
  { return r_Result;}
}

S_Obj apendice_arvorebinaria13_busca_interna(self, p_v, p_elem)
struct apendice_arvorebinaria13 *self;
S_Obj p_v;
S_Obj p_elem;
{ S_Obj r_Result;
  if (((p_elem != S_nilObj)?0:1))
  r_Result = S_nilObj;
  else { S_Obj l_esq;
    S_Obj l_dir;
    S_Obj l_val;
    l_dir = apendice_elem_arvore11_consulta(p_elem, &l_esq, &l_val);
    if ((((S_Obj (*)())DISPATCH(p_v, "lt", 8288))(p_v, l_val) !=
         S_nilObj))
    r_Result = ((S_Obj (*)())DISPATCH(((S_Obj)self),
      "busca_interna", 50838))(((S_Obj)self), p_v, l_esq);
    else if ((((S_Obj (*)())DISPATCH(p_v, "gt", 8103))(p_v, l_val) !=
             S_nilObj))
    r_Result = ((S_Obj (*)())DISPATCH(((S_Obj)self),
      "busca_interna", 50838))(((S_Obj)self), p_v, l_dir);
    else r_Result = S_trueObj;
  }
  { return r_Result;}
}

S_Obj apendice_arvorebinaria13_vazio(self)
struct apendice_arvorebinaria13 *self;
{ S_Obj r_Result;
  r_Result = ((self->raiz != S_nilObj)?S_trueObj:S_nilObj);
  { return r_Result;}
}

struct apendice_arvorebinaria13 *apendice_arvorebinaria13_new(self)
struct apendice_arvorebinaria13 *self;
{ if (!self){ self = NEW(apendice_arvorebinaria13);
    self->clss = &apendice_arvorebinaria13;
  }
  ;
  self->raiz = S_nilObj;
  return self;
```

```
}
```

Classe inteiro

```
struct apendice_inteiro7
{ struct Class *clss;
  int val;
}
;
S_Obj apendice_inteiro7_lt();
S_Obj apendice_inteiro7_gt();
int apendice_inteiro7_valor();
static procType pr_inteiro[] = { 0, (procType)apendice_inteiro7_valor,
                                 0, (procType)apendice_inteiro7_lt,
                                 (procType)apendice_inteiro7_gt};
static char *pn_inteiro[] = {"", "valor", "", "lt", "gt"};
struct Class apendice_inteiro7 = {5, "i", pr_inteiro, pn_inteiro};
int apendice_inteiro7_valor(self)
struct apendice_inteiro7 *self;
{ int r_Result;
  r_Result = self->val;
  { return r_Result;}
}

S_Obj apendice_inteiro7_lt(self, p_i)
struct apendice_inteiro7 *self;
S_Obj p_i;
{ S_Obj r_Result;
  r_Result = ((self->val < apendice_inteiro7_valor(p_i))?
    S_trueObj:S_nilObj);
  { return r_Result;}
}

S_Obj apendice_inteiro7_gt(self, p_i)
struct apendice_inteiro7 *self;
S_Obj p_i;
{ S_Obj r_Result;
  r_Result = ((self->val > apendice_inteiro7_valor(p_i))?
    S_trueObj:S_nilObj);
  { return r_Result;}
}

struct apendice_inteiro7 *apendice_inteiro7_new(self, p_i)
struct apendice_inteiro7 *self;
int p_i;
{ if (!self){ self = NEW(apendice_inteiro7);
    self->clss = &apendice_inteiro7;
```

```
  }
  ;
  self->val = p_i;
  return self;
}
```

Construtor

```
void __apendice ()
{ static int init = 0;
  if (init) return; else init = 1;
  __print();
  { { S_Obj l_bt = (S_Obj)apendice_arvorebinaria13_new(0);
      ((int (*)())DISPATCH(l_bt, "insere", 23902))(l_bt,
                           (S_Obj)apendice_inteiro7_new(0, 10));
      ((int (*)())DISPATCH(l_bt, "insere", 23902))(l_bt,
                           (S_Obj)apendice_inteiro7_new(0, 1));
      ((int (*)())DISPATCH(l_bt, "insere", 23902))(l_bt,
                           (S_Obj)apendice_inteiro7_new(0, 20));
      ((int (*)())DISPATCH(l_bt, "insere", 23902))(l_bt,
                           (S_Obj)apendice_inteiro7_new(0, 15));
      ((int (*)())DISPATCH(l_bt, "insere", 23902))(l_bt,
                           (S_Obj)apendice_inteiro7_new(0, 18));
      ((int (*)())DISPATCH(l_bt, "insere", 23902))(l_bt,
                           (S_Obj)apendice_inteiro7_new(0, 5));
      ((int (*)())DISPATCH(l_bt, "insere", 23902))(l_bt,
                           (S_Obj)apendice_inteiro7_new(0, 30));
      { int l_i = 0;
        while(1) if ((l_i < 40))
        { if ((((S_Obj (*)())DISPATCH(l_bt, "busca", 19462))
              (l_bt, (S_Obj)apendice_inteiro7_new(0, l_i)) != S_nilObj))
          print_int3(l_i);
          else ;
          l_i = (l_i + 1);
        }
        else break;
      }
    }
  }
}
```

Bibliografia

[Ada83] ANSI. *Ada Programming Language*, 1983. ANSI/MIL-STD 1815A.

[AH95] Ole Agesen e Urs Hölzle. Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages. Em *Proceedings OOPSLA '95*, Austin, Texas, Outubro 1995. Publicado como ACM Sigplan Notices, 30(10).

[Ake95] Claudio Luiz Akerman. Coleta de lixo em sistemas distribuídos orientados a objetos. Dissertação de Mestrado, PUC-Rio, 1995.

[APS93] Ole Agesen, Jens Palsberg, e Michael I. Schwartzbach. Type inference of SELF: Analysis of objects with dynamic and multiple inheritance. Em O. M. Nierstrasz, editor, *Proceedings ECOOP '93*, LNCS 707, páginas 247–267, Kaiserslautern, Germany, Julho 1993. Springer-Verlag.

[BV93] G. Booch e M. Vilot. Using namespaces. *C++ Report*, 5(9):14–22, 1993.

[Cha93] Craig Chambers. The Cecil language: Specification and rationale. Relatório Técnico TR-93-03-05, Department of Computer Science and Engineering, University of Washington, Março 1993.

[CI95] A. Carregal e R. Ierusalimschy. Tche — a visual environment for the Lua language. Em *VIII Simpósio Brasileiro de Computação Gráfica*, páginas 227–232, São Carlos, 1995.

[CU90] Craig Chambers e David Ungar. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. Em *Proceedings PLDI '90*, páginas 150–164, White Plains, New York, Junho 1990. Publicado como ACM Sigplan Notices, 25(6).

[CU91] Craig Chambers e David Ungar. Making pure object-oriented languages practical. Em Norman Meyrowitz, editor, *Proceedings OOPSLA '91*, páginas 1–15, Phoenix, Arizona, Novembro 1991. Publicado como ACM Sigplan Notices, 26(11).

[CUL89] Craig Chambers, David Ungar, e Elgin Lee. An efficient implementation of SELF – a dynamically-typed object-oriented language based on prototypes. Em Norman Meyrowitz, editor, *Proceedings OOPSLA '89*, páginas 49–70, New Orleans, Louisiana, Outubro 1989. Publicado como ACM Sigplan Notices, 24(10).

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

[DC94] Jeffrey Dean e Craig Chambers. Toward better inlining decisions using inlining trials. Em *ACM Conference on LISP and Functional Programming 1994*, páginas 273–282, Orlando, FL, Junho 1994.

[DCG95] Jeffrey Dean, Craig Chambers, e David Grove. Selective specialization for object-oriented languages. Em *Proceedings PLDI '95*, páginas 93–102, La Jolla, California, Junho 1995. Publicado como ACM Sigplan Notices, 30(6).

[DGC95] Jeffrey Dean, David Grove, e Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. Em W. Olthoff, editor, *Proceedings ECOOP '95*, LNCS 952, páginas 77–101, Aarhus, Denmark, Agosto 1995. Springer-Verlag.

[DGLM95] M. Day, R. Gruber, B. Liskov, e A.C. Myers. Subtypes vs. where clauses: Constraining parametric polymorphism. Em *Proceedings OOPSLA '95*, Austin, Texas, Outubro 1995. Publicado como ACM Sigplan Notices, 30(10).

[DS84] L. Peter Deutsch e Alan Schiffman. Efficient implementation of the Smalltalk-80 system. Em *Proceedings of the 11th Symposium on the Principles of Programming Languages*, Salt Lake City, UT, 1984.

[ES90] M. Ellis e B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[GDGC95] David Grove, Jeffrey Dean, Charles Garrett, e Craig Chambers. Profile-guided receiver class prediction. Em *Proceedings OOPSLA '95*, Austin, Texas, Outubro 1995. Publicado como ACM Sigplan Notices, 30(10).

[Gei83] Leo Bernhard Geissmann. *Separate Compilation in Modula-2 and the Structure of the Modula-2 Compiler on the Personal Computer Lilith.* Tese de Doutorado, Swiss Federal Institute of Technology Zurich, 1983.

[Ghe91] Giorgio Ghelli. A static type system for message passing. Em Norman Meyrowitz, editor, *Proceedings OOPSLA '91*, páginas 129–145, Phoenix, Arizona, Novembro 1991. Publicado como ACM Sigplan Notices, 26(11).

[GR83] A. Goldberg e D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[HCU91] Urs Hölzle, Craig Chambers, e David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. Em P. America, editor, *Proceedings ECOOP '91*, LCNS 512, páginas 21–38, Geneva, Switzerland, Julho 1991. Springer-Verlag.

[Höl93] Urs Hölzle. A fast write barrier for generational garbage collectors. Em *OOPSLA'93 Garbage Collection Workshop*, Washington, D.C., Outubro 1993.

[HU94] Urs Hölzle e David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. Em *Proceedings PLDI '94*, páginas 326–336, Orlando, Florida, Junho 1994. Publicado como ACM Sigplan Notices, 29(6).

[Ier91] R. Ierusalimschy. The O=M programming language. Monografias em Ciência da Computação 3/91, PUC-Rio, Rio de Janeiro, Brazil, 1991.

[Ier93] R. Ierusalimschy. A denotational approach for type-checking in object-oriented programming languages. *Computer Languages*, 19(1):19–40, 1993.

[Jon90] C. B. Jones. *Systematic Software Development using VDM.* International Series in Computer Science. Prentice Hall, second edition, 1990.

[Kra83] Glen Krasner, editor. *Smalltalk-80: Bits of history and words of advice.* Addison-Wesley, 1983.

[Laj93] J. Lajoie. Standard C++: Namespaces. *C++ Report*, 5(9):52–56, 1993.

[Lea90] Doug Lea. Customization in C++. Em *Proceedings of the Usenix C++ Conference*, San Francisco, CA, Abril 1990.

[LS91] Chu-Cheow Lim e A. Stolcke. Sather language design and performance evaluation. Relatório Técnico TR-91-034, International Computer Science Institute, Berkeley, Maio 1991.

[Mey88a] B. Meyer. Eiffel – a language and environment for software enginnering. *The Journal of Systems and Software*, 8(3):129–46, 1988.

[Mey88b] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1988.

[Mey96] B. Meyer. The ISE Eiffel garbage collection mechanism: An overview. Relatório Técnico TR-EI-56/GC, Interactive Software Engineering Inc., Fevereiro 1996.

[ML94] Andrew Myers e Barbara Liskov. Efficient implementation of parameterized types despite subtyping. Relatório Técnico Thor Note 9, MIT Laboratory for Computer Science, Julho 1994.

[MM90] O. Madsen e B. Magnusson. Strong typing of object oriented languages revisited. Em *ECOOP/OOPSLA'90 Proceedings*, páginas 140–150, 1990.

[MW91] H. Mössenböck e N. Wirth. The programming language Oberon-2. *Structured Programming*, 12:179–195, 1991.

[Nel91] G. Nelson, editor. *Systems Programming with Modula-3.* Prentice Hall, 1991. (see also http://www.research.digital.com/SRC/modula-3/html/home.html).

[PG94] M. Potel e J. Grimes. The architecture of the Taligent system. *Dr. Dobb's Special Report*, 19(16):36–40, 1994.

[PS91] Jens Palsberg e Michael I. Schwartzbach. Object-oriented type inference. Em Norman Meyrowitz, editor, *Proceedings OOPSLA '91*, páginas 146–161, Phoenix, Arizona, Novembro 1991. Publicado como ACM Sigplan Notices, 26(11).

[RIR93a] N. Rodriguez, R. Ierusalimschy, e J. L. Rangel. Conciliação de flexibilidade e verificação estática em linguagens orientadas a objetos. Em *VII Simpósio Brasileiro de Engenharia de Software*, páginas 282–294, Rio de Janeiro, Brazil, 1993.

[RIR93b] N. Rodriguez, R. Ierusalimschy, e J. L. Rangel. Types in School. *Sigplan Notices*, 28(8), 1993.

[Rod93] N. Rodriguez. *Um Sistema de Tipos Orientado por Objetos, Incluindo Persistência.* Tese de Doutorado, Dep. Informática, PUC-Rio, Rio de Janeiro, Brazil, 1993.

[SCB+86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Killian, e Carrie Wilpolt. An introduction to Trellis/Owl. Em Norman Meyrowitz, editor, *Proceedings OOPSLA '86*, páginas 9–16, Portland, Oregon, Novembro 1986. Publicado como ACM Sigplan Notices, 21(11).

[SOM93] C. Szypersky, S. Omohundro, e S. Murer. Engineering a programming language: The type and class system of Sather. Relatório Técnico TR 93-064, ICSI, Berkeley, CA, 1993.

[Str92] B. Stroustrup. *The C++ Programming Language.* Prentice-Hall, second edition, 1992.

[Str95] Bjarne Stroustrup. *The Design and Evolution of C++.* Addison-Wesley, 1995.

[Szy92] Clemens A. Szyperski. Import is not inheritance – why we need both: Modules and classes. Em O. Lehrmann Madsen, editor, *Proceedings ECOOP '92*, LNCS 615, páginas 19–32, Utrecht, The Netherlands, Junho 1992. Springer-Verlag.

[Ung84] David Ungar. Generation scavenging: a non-disruptive high performance storage reclamation algorithm. Em *Proceedings of the ACM Sigsoft/Sigplan Software Engeneering Symposium on Pratical Software Development Environments*, páginas 157–167, Pittsburgh, Pennsylvania, Maio 1984. Publicado como ACM Sigplan Notices, 19(5).

[US87] David Ungar e Randall B. Smith. Self: The power of simplicity. Em Norman Meyrowitz, editor, *Proceedings OOPSLA '87*, páginas 227–242, Orlando, Florida, Dezembro 1987. Publicado como ACM Sigplan Notices, 22(12).

[Val95] Ray Valdés (editor). The interoperable objects revolution. *Dr. Dobb's*, 19(16), 1995.

[Wir85] N. Wirth. *Programming in Modula-2.* Springer-Verlag, third edition, 1985.

[Wir88] N. Wirth. From Modula to Oberon. *Software – Practice and Experience*, 18(7):661–670, 1988.