



Universidade Federal de Pernambuco
Centro de Informática

Pós-graduação em Ciência da Computação

INTEGRANDO HASKELL À PLATAFORMA .NET

Monique Louise de Barros Monteiro

DISSERTAÇÃO DE MESTRADO

Recife
6 de abril de 2006

Universidade Federal de Pernambuco
Centro de Informática

Monique Louise de Barros Monteiro

INTEGRANDO HASKELL À PLATAFORMA .NET

*Trabalho apresentado ao Programa de Pós-graduação em
Ciência da Computação do Centro de Informática da Uni-
versidade Federal de Pernambuco como requisito parcial
para obtenção do grau de Mestre em Ciência da Com-
putação.*

Orientador: *André Luís de Medeiros Santos*

Recife
6 de abril de 2006

À minha mãe.

AGRADECIMENTOS

A todos aqueles que, direta ou indiretamente contribuíram para a realização desta pesquisa:

- primeiramente a Deus, por Tudo;
- à minha mãe, por todo o seu carinho, incentivo e dedicação;
- a André Santos, idealizador deste projeto, pela amizade, excelente orientação e incentivo que guiaram o desenvolvimento desta pesquisa;
- aos membros da banca examinadora, Hermano Moura e Ricardo Massa, pelos comentários e sugestões para melhoria deste trabalho;
- à Microsoft Research, por ter financiado parte do projeto;
- a Don Syme, Simon Marlow e Simon Peyton Jones, da Microsoft Research, e a todos os outros que contribuíram com informações importantes sobre o GHC e integração de linguagens funcionais com .NET;
- a Mauro Araújo e Rafael Borges, pelas discussões técnicas e contribuições na fase inicial do projeto;
- ao Centro de Informática e à sua ótima equipe de professores, que contribuíram para a qualidade da minha formação profissional e proporcionaram a oportunidade de realização deste trabalho;
- a todos os meus amigos, pelo apoio.

*Uma alta posição na vida é conquistada pela bravura e elegância com que
se enfrentam os mais difíceis desafios.*

—AUTOR DESCONHECIDO

RESUMO

Tradicionalmente, linguagens funcionais fornecem um grau de abstração superior ao encontrado em outros paradigmas (imperativo, orientado a objetos), o que se manifesta por meio de construções de alto nível como funções de alta ordem, aplicações parciais, polimorfismo paramétrico e, em algumas linguagens, avaliação sob demanda. Entretanto, a utilização do paradigma funcional tem-se restringido basicamente a aplicações acadêmicas. Essa restrição é em parte explicada pela ausência de ambientes de desenvolvimento e APIs que melhorem a produtividade do desenvolvedor na construção de aplicações que fazem uso das tecnologias mais recentes de desenvolvimento *Web*, computação distribuída, arquitetura orientada a serviços, entre outras. Por outro lado, plataformas como a *JAVA Virtual Machine* e, mais recentemente, a Plataforma .NET, disponibilizam uma ampla gama de serviços e bibliotecas que satisfazem aos tipos de aplicações supracitados. O ambiente .NET, em particular, destaca-se por suportar múltiplas linguagens, apesar do suporte ser mais amplo a linguagens orientadas a objeto. Dentro desse contexto, surge a oportunidade de portar linguagens funcionais para essa plataforma, permitindo não apenas o acesso aos serviços por ela fornecidos como também uma interoperabilidade natural com outras linguagens. O objetivo deste trabalho é o desenvolvimento de uma implementação da linguagem funcional Haskell para a Plataforma .NET. Tal implementação não é trivial devido ao *gap* semântico que existe entre uma linguagem funcional com avaliação sob demanda e um ambiente como o .NET. Foi desenvolvido um gerador de código capaz de gerar, a partir de um programa Haskell, código em IL - linguagem *assembly* suportada pela máquina virtual. Paralelamente, foram conduzidas medições de desempenho do código gerado. Tais medições demonstraram performance razoável para vários programas. Entretanto, a principal contribuição deste trabalho está na disponibilização de uma implementação Haskell que serve como principal passo rumo à interoperabilidade com a Plataforma .NET. Além disso, a solução desenvolvida serve como um ambiente de teste e validação de diferentes alternativas de tradução de uma linguagem funcional para as construções encontradas em uma plataforma como .NET.

Palavras-chave: compiladores, linguagens funcionais, Haskell, .NET, máquinas virtuais.

ABSTRACT

Traditionally, functional languages offer a higher abstraction level when compared to other paradigms (imperative or object-oriented). Examples include high level features such as higher order functions, partial applications, parametric polymorphism and, in some languages, lazy evaluation. However, the utilization of the functional paradigm has been restricted to academic applications. This restriction is partly explained by the absence of development environments and APIs which improve the productivity of the developer responsible for constructing advanced applications. These applications commonly make use of the most recent technologies for Web development, distributed computing, service oriented architecture and others. On the other hand, platforms such as *JAVA Virtual Machine* and, most recently, the .NET Platform, provide a broad range of services and libraries which support the requirements of the above cited applications. Particularly, the .NET Platform provides a multilanguage support, despite its stronger support for object-oriented languages. In this context, we have the opportunity to port functional languages to this platform, enabling both the access to its services and the interoperability with other languages. The objective of this work is the development of an implementation of the Haskell functional language for the .NET Platform. This implementation is not trivial due to the semantic gap between a non-strict functional language and an environment such as .NET. A code generator capable of generating IL – the assembly language supported by the virtual machine – was developed. Performance measurements were conducted on the generated code. Such measurements demonstrated acceptable performance for several programs. However, the main contribution of this work is to provide a Haskell implementation as the first step towards its interoperability with .NET. Finally, the implemented solution can be used as an environment for test and validation of different choices for translating functional features to the elements available on the .NET Platform.

Keywords: compilers, functional languages, Haskell, .NET, virtual machines.

SUMÁRIO

| | |
|---|----------|
| Capítulo 1—Introdução | 1 |
| 1.1 Contexto e Motivação | 1 |
| 1.2 Contribuições | 3 |
| 1.3 Estrutura da Dissertação | 4 |
| Capítulo 2—Introdução à Plataforma .NET | 5 |
| 2.1 Common Language Infrastructure | 6 |
| 2.1.1 CLI – Principais Conceitos | 7 |
| 2.2 Common Type System | 8 |
| 2.2.1 Value Types | 8 |
| 2.2.2 Reference Types | 10 |
| 2.2.3 Boxing e Unboxing | 11 |
| 2.2.4 Ponteiros | 12 |
| 2.2.5 Delegates | 12 |
| 2.2.6 Generics | 14 |
| 2.2.7 Segurança de Tipos e Verificação | 16 |
| 2.3 Common Language Specification | 18 |
| 2.4 Metadados | 19 |
| 2.4.1 Assemblies, Módulos e Manifestos | 19 |
| 2.5 Ambiente Virtual de Execução | 20 |
| 2.5.1 Máquina Virtual | 20 |
| 2.5.2 Mecanismos de Passagem de Parâmetros | 21 |
| 2.6 Instruções | 22 |
| 2.6.1 Tipos de Dados | 22 |
| 2.6.2 Principais Grupos de Instruções | 22 |
| 2.7 Plataforma Microsoft .NET – Visão Geral | 24 |
| 2.8 Common Language Runtime | 25 |
| 2.8.1 Compilação Just-In-Time | 26 |
| 2.8.2 Delegates | 27 |
| 2.8.3 Generics | 27 |
| 2.8.4 Gerenciamento de Memória | 29 |
| 2.9 Outras Implementações da CLI | 31 |
| 2.9.1 Shared Source CLI (Rotor) | 31 |
| 2.9.2 Mono | 31 |
| 2.9.3 DotGNU Portable .NET | 32 |
| 2.10 Considerações Finais | 32 |

| | |
|---|-----------|
| Capítulo 3—Integração de Linguagens Funcionais ao Ambiente .NET | 33 |
| 3.1 Introdução a Linguagens Funcionais – Principais Conceitos | 33 |
| 3.1.1 Ordem de Avaliação | 33 |
| 3.1.2 Funções de Alta Ordem | 34 |
| 3.1.3 Aplicações Parciais | 35 |
| 3.2 Integração ao Ambiente .NET | 35 |
| 3.2.1 Bridge | 36 |
| 3.2.2 Compilação | 36 |
| 3.3 Linguagem Alvo | 37 |
| 3.3.1 Linguagem de Alto Nível | 37 |
| 3.3.2 Código Final | 38 |
| 3.4 Representação de Closures | 38 |
| 3.4.1 Uma Classe por Closure | 38 |
| 3.4.2 Classes Pré-definidas | 39 |
| 3.5 Aplicação de Funções | 41 |
| 3.5.1 Modelo Push/Enter | 41 |
| 3.5.2 Modelo Eval/Apply | 41 |
| 3.6 Polimorfismo Paramétrico | 42 |
| 3.7 Uniões Discriminadas | 42 |
| 3.8 Controle da Pilha de Chamadas | 43 |
| 3.8.1 Trampolim | 43 |
| 3.8.2 Tail-calls | 44 |
| 3.9 Trabalhos Relacionados | 44 |
| 3.9.1 Linguagens Funcionais Não Estritas | 44 |
| 3.9.2 Linguagens Funcionais Estritas | 45 |
| 3.9.3 Implementações para a JVM | 47 |
| 3.10 Considerações Finais | 48 |
| Capítulo 4—O Compilador Haskell.NET | 49 |
| 4.1 Arquitetura | 49 |
| 4.1.1 Gerador de IL | 51 |
| 4.2 Estratégia de Compilação | 52 |
| 4.2.1 Compilação STG \rightarrow IL | 52 |
| 4.2.2 Ambiente de Execução | 56 |
| 4.2.3 Exemplo | 58 |
| 4.3 Considerações Finais | 61 |
| Capítulo 5—Avaliação de Desempenho e Otimizações | 63 |
| 5.1 Metodologia | 63 |
| 5.2 Análise do Impacto de Casts | 65 |
| 5.2.1 castclass X insinst | 65 |
| 5.2.2 Resultados | 66 |

| | | |
|---|---|------------|
| 5.3 | Análise do Impacto de Tail-Calls | 67 |
| 5.4 | Variações no Uso de Delegates | 69 |
| 5.4.1 | Conclusão | 72 |
| 5.5 | Análise do Impacto de “Constant Applicative Forms” | 73 |
| 5.6 | Otimizações | 74 |
| 5.6.1 | Remoção do “Frame” de Atualização | 74 |
| 5.6.2 | Simulando Atualização “In-place” | 75 |
| 5.6.3 | Compartilhamento de Booleanos | 77 |
| 5.6.4 | Compartilhamento de Valores Inteiros | 79 |
| 5.6.5 | Remoção de Tail-calls do Ambiente de Execução | 79 |
| 5.6.6 | Substituição de Tail-calls por Instruções de Desvio | 81 |
| 5.7 | Haskell.NET X GHC Nativo | 81 |
| 5.8 | Considerações Finais | 83 |
| Capítulo 6—Interoperabilidade em Linguagens Funcionais | | 85 |
| 6.1 | Interoperabilidade em Haskell | 86 |
| 6.1.1 | Foreign Function Interface (FFI) | 86 |
| 6.1.2 | H/Direct | 87 |
| 6.1.3 | Lambda | 88 |
| 6.1.4 | Hugs .NET | 89 |
| 6.2 | Interoperabilidade em F# | 89 |
| 6.3 | Interoperabilidade em SML .NET | 90 |
| 6.4 | Interoperabilidade em Mondrian | 91 |
| 6.5 | Considerações Finais | 91 |
| Capítulo 7—Conclusões e Trabalhos Futuros | | 94 |
| 7.1 | Considerações Gerais e Principais Contribuições | 94 |
| 7.2 | Trabalhos Futuros | 95 |
| 7.2.1 | Suporte a Bibliotecas | 95 |
| 7.2.2 | Integração com Phoenix | 96 |
| 7.2.3 | Análises de Desempenho | 96 |
| 7.2.4 | Suporte a Interoperabilidade | 98 |
| 7.3 | Considerações Finais | 98 |
| Referências Bibliográficas | | 99 |
| Apêndice A—Esquemas de Compilação | | 106 |

LISTA DE FIGURAS

| | | |
|------|--|-----|
| 2.1 | CIL – Modelo de Execução | 7 |
| 2.2 | <i>Common Type System</i> | 9 |
| 2.3 | Exemplo de Enumeração em C# | 9 |
| 2.4 | Exemplo de Enumeração em CIL | 10 |
| 2.5 | Exemplo de Estrutura em C# | 10 |
| 2.6 | Exemplo de Estrutura em CIL | 11 |
| 2.7 | Delegates em C# | 13 |
| 2.8 | Delegate Function em CIL | 14 |
| 2.9 | Uso de <i>Delegates</i> em CIL | 15 |
| 2.10 | Polimorfismo Paramétrico pela Abordagem Tradicional | 16 |
| 2.11 | Polimorfismo Paramétrico através de <i>Generics</i> | 17 |
| 2.12 | Relacionamento entre Programas Válidos e Verificáveis | 18 |
| 2.13 | Plataforma .NET – Arquitetura | 25 |
| 2.14 | <i>Delegates</i> – Implementação | 28 |
| 2.15 | <i>Generics</i> – Implementação | 28 |
| 3.1 | Parte do Ambiente de Execução de F# (versão simplificada) | 46 |
| 4.1 | Arquitetura do Compilador | 50 |
| 4.2 | Arquitetura do Gerador de IL | 51 |
| 4.3 | Linguagem STG | 53 |
| 4.4 | Ambiente de Execução | 57 |
| 4.5 | Compilação de map , fx e fxs | 59 |
| 4.6 | <i>Slow Entry Point</i> de map | 60 |
| 5.1 | Casamento de Padrão | 65 |
| 5.2 | Uso de <i>casts</i> na Compilação de Casamento de Padrão | 65 |
| 5.3 | Separação <i>Closures–Delegates</i> – Diagrama de Classes | 70 |
| 6.1 | Sintaxe das Declarações FFI | 86 |
| 7.1 | Phoenix – Arquitetura | 97 |
| A.1 | Esquemas de Compilação B , FV e I | 107 |
| A.2 | Esquema de Compilação E | 108 |
| A.3 | Esquemas de Compilação A , C , L e T | 109 |
| A.4 | Esquemas de Compilação S e SC | 110 |

LISTA DE TABELAS

| | | |
|------|---|----|
| 2.1 | Exemplos de Instruções de <i>Load</i> e <i>Store</i> | 23 |
| 2.2 | Exemplos de Instruções de Chamada a Métodos | 24 |
| 2.3 | Exemplos de Instruções do Modelo de Objetos | 24 |
| 5.1 | Configuração do Ambiente de Testes | 64 |
| 5.2 | Impacto das Operações de <i>cast</i> | 66 |
| 5.3 | Impacto do Uso de <i>Tail-calls</i> | 68 |
| 5.4 | Valores para <i>%Tempo em GC</i> | 69 |
| 5.5 | Variações no Uso de <i>Delegates</i> | 71 |
| 5.6 | Gerenciamento de Memória com <i>Closures</i> como <i>Delegates</i> | 71 |
| 5.7 | Gerenciamento de Memória com a Separação <i>Closures–delegates</i> | 72 |
| 5.8 | Tamanho das <i>Closures</i> | 72 |
| 5.9 | Remoção do “ <i>Frame</i> ” de Atualização | 75 |
| 5.10 | Simulação de Atualização “ <i>In-place</i> ” | 77 |
| 5.11 | Compartilhamento de Valores Booleanos | 78 |
| 5.12 | Compartilhamento de Valores Inteiros | 80 |
| 5.13 | Impacto do Uso de <i>Tail-calls</i> no Ambiente de Execução | 80 |
| 5.14 | Impacto do Uso de <i>Tail-calls</i> no Código Compilado | 81 |
| 5.15 | Impacto da Substituição de <i>Tail-call</i> por Instrução de Desvio | 82 |
| 5.16 | Haskell.NET vs. GHC Nativo | 82 |

CAPÍTULO 1

INTRODUÇÃO

Este capítulo apresenta uma visão geral do trabalho e está organizado da seguinte forma:

- a Seção 1.1 trata dos fatores que motivaram o presente trabalho, apresentando uma breve descrição sobre plataformas de código gerenciado e linguagens funcionais;
- a Seção 1.2 aponta as principais contribuições;
- a Seção 1.3 descreve, em termos dos capítulos, a estrutura da dissertação.

1.1 CONTEXTO E MOTIVAÇÃO

A indústria de *software* vê a necessidade cada vez maior de atender a um requisito chave: produtividade. Aqui, quando falamos em produtividade, usamos talvez um termo excessivamente genérico, que resume o atendimento a vários outros requisitos, como robustez, portabilidade, manutenibilidade, entre outros. Porém, não devemos esquecer o conceito mais conhecido de produtividade por parte dos desenvolvedores: produzir *software* com mais qualidade em menos tempo. Por qualidade, entendemos o atendimento a pelo menos alguns dos requisitos não funcionais supracitados, entre outras características. Por pouco tempo, entendemos que deve ser fácil e rápido incorporar tais requisitos. Em suma, não é mais viável que a cada novo projeto sejam desenvolvidos e integrados componentes que sempre desempenham os mesmos papéis, de forma pouco customizável, e cujo desenvolvimento desvie a atenção da lógica de negócio do sistema.

Dentro desse contexto, surgiram nos últimos anos plataformas de execução de código cujo objetivo era atender aos principais requisitos não funcionais das aplicações. Essas plataformas cumprem pelo menos três papéis importantes:

- disponibilizar uma ampla gama de serviços e componentes de alto nível, que vão além das funções encontradas em bibliotecas básicas específicas de linguagem;
- prover serviços de baixo nível, tais como gerenciamento automático de memória e de *threads*, evitando erros de programação freqüentemente cometidos nessas áreas;
- implementar um modelo viável de ser facilmente portado para diferentes plataformas de *hardware* e sistemas operacionais.

O primeiro papel diz respeito a auxiliar a montagem da arquitetura de sistemas grandes, fornecendo infra-estrutura que atende a tecnologias cada vez mais sofisticadas como *Web Services* [W3C], desenvolvimento para dispositivos móveis, computação distribuída, etc. O segundo foca na facilidade de desenvolvimento do ponto de vista do programador.

Já o último papel objetiva garantir que toda essa infra-estrutura também será utilizável em diferentes plataformas de *hardware* e sistemas operacionais.

O primeiro ambiente utilizado em larga escala e que desempenhou os papéis acima citados foi a plataforma JAVA [GJSB00], juntamente com suas extensões JAVA 2 *Enterprise Edition* (J2EE) [J2E] e JAVA 2 *Mobile Edition* (J2ME) [J2M], para desenvolvimento corporativo e móvel, respectivamente. Voltada especificamente para a linguagem JAVA, fornece um ambiente baseada em máquina virtual que executa código compilado não para código nativo, mas sim para uma notação intermediária. Utilizando esse modelo, garante que o código fonte é portátil, pois tudo o que muda entre diferentes plataformas é a implementação da máquina virtual.

A próxima evolução seria trazer todos os benefícios oferecidos pelo ambiente JAVA para outras linguagens. Isso não significa reimplementar plataformas de desenvolvimento para cada linguagem, mas sim projetar um único ambiente independente de linguagem de programação. Uma idéia simples, a princípio, uma vez que estamos falando de máquinas virtuais e de linguagens *assembly* intermediárias. Porém, para que esse suporte seja possível, é necessário que o ambiente em questão implemente conceitos de diferentes linguagens e paradigmas.

Dessa necessidade surgiu a motivação para o ambiente Microsoft .NET [NET]: reunir várias linguagens em uma única plataforma. Qualquer linguagem deve ser capaz de acessar os serviços e bibliotecas disponíveis e fazer uso de componentes escritos em outras linguagens. Adicionalmente, o código escrito em uma linguagem A deve ser executado da mesma forma que o código escrito em uma linguagem B, sem que a máquina virtual faça qualquer distinção entre elas. Posteriormente, um “subconjunto” do .NET foi padronizado, permitindo que surgissem outras implementações para diferentes sistemas operacionais. Com isso, ganhou-se portabilidade.

Os ambientes que introduzimos nos parágrafos anteriores são comumente denominados de *ambientes de código gerenciado*, uma vez que toda a execução do código é dirigida por uma máquina virtual onisciente de tudo o que acontece no programa (valores das variáveis, instrução sendo executada no momento, pilha de chamadas, etc.). Embora a Plataforma .NET suporte em alguns casos a execução de código não gerenciado, sua utilização mais comum e recomendável é de fato com código gerenciado. Por isso, daqui em diante nos referiremos a essa plataforma como sendo um ambiente de código gerenciado.

A Plataforma Microsoft .NET é distribuída com um conjunto padrão de compiladores para diferentes linguagens, a saber C# [ECM05], C++ [.NEa], Visual Basic [.NEb] e J# [J#]. Todas são orientadas a objetos. A Plataforma JAVA é composta apenas pelo compilador JAVA, linguagem também orientada a objeto. Vê-se dessa forma a maior atenção comercial dada ao paradigma de orientação a objetos. Por outro lado, esse dado não significa que devemos concluir que outros paradigmas não possam ser suportados, especialmente em um ambiente multi-linguagem como o .NET.

Um exemplo de paradigma de programação importante, o *funcional*, baseia-se no conceito mais básico que pode ser dado a um programa: uma função, que recebe parâmetros, executa um processamento e retorna o resultado. Embora funções também estejam presentes nos paradigmas imperativo e orientado a objetos, existe uma diferença importante: em linguagens funcionais não há efeitos colaterais. Enquanto programas imperativos

executam comandos que modificam e consultam posições de memória, conhecidas como *variáveis*, programas funcionais não possuem comandos ou variáveis, sendo “escritos inteiramente na linguagem de expressões, funções e declarações” [Wat90].

Pela ausência de efeitos colaterais e pela presença de características que lhes conferem maior poder de expressão, tais como funções de alta ordem, polimorfismo e avaliação sob demanda, as linguagens funcionais são possivelmente aquelas que possuem *maior grau de abstração*. As mesmas características também tornam possível provar a corretude dos programas, por meio de provas formais baseadas no princípio da indução matemática. Como consequência, o alto grau de abstração e formalismo contribuem para uma menor quantidade de erros e, por que não dizer, para uma maior *produtividade*, uma vez que menos linhas de código são necessárias para implementar rotinas outrora complexas em linguagens imperativas.

Enfim chegamos ao mesmo requisito – produtividade – através de diferentes caminhos. É importante lembrar que esse requisito é implementado de diferentes formas nos dois contextos. Enquanto em ambientes de código gerenciado a produtividade é facilitada pelos serviços disponibilizados, sejam eles de alto ou baixo nível, em linguagens funcionais ela ocorre a nível de linguagem de programação. Por outro lado, a adoção em larga escala de linguagens funcionais enfrenta um obstáculo: a ausência de APIs ou infra-estrutura de desenvolvimento adequada ao projeto de sistemas “reais”. Concluímos, portanto, que as implementações existentes de linguagens funcionais carecem de serviços como os disponibilizados por plataformas como JAVA ou .NET. Enquanto isso, as linguagens orientadas a objeto suportadas por essas plataformas não possuem o alto grau de abstração e facilidade de programação presentes no paradigma funcional.

Surge então a motivação para combinar as vantagens dos dois tipos de ambientes em um só, estudando meios de integrar linguagens funcionais a plataformas de código gerenciado. Pelo suporte multi-linguagem, escolhemos a Plataforma .NET para esse fim. Como estudo de caso de linguagem funcional, escolhemos Haskell [Has], por esta ser uma linguagem funcional *pura*, sem nenhuma forma de efeito colateral e com as características mais representativas desse paradigma, como funções de alta ordem, avaliação sob demanda e polimorfismo paramétrico.

Apesar do ambiente .NET ser multi-linguagem, as características de uma linguagem como Haskell consistem em um desafio à sua implementação eficiente nessa plataforma. Neste trabalho será estudado como essa integração pode ser feita, por meio do desenvolvimento de um compilador Haskell para .NET. Adicionalmente, serão analisadas alternativas e variações que tenham impacto direto no desempenho de programas funcionais nesse ambiente. A relevância deste trabalho está principalmente na identificação das dificuldades inerentes a essa integração e nos resultados experimentais que corroboram ou descartam determinadas decisões de projeto.

1.2 CONTRIBUIÇÕES

As principais contribuições deste trabalho são:

- uma implementação de Haskell para .NET, baseada em compilação, que permite estudos acerca de diferentes estratégias de integração com o ambiente;

- resultados experimentais acerca de variações nas estratégias de compilação;
- o primeiro passo rumo à interoperabilidade de Haskell com o ambiente .NET.

1.3 ESTRUTURA DA DISSERTAÇÃO

Esta dissertação está estruturada da seguinte forma:

- O presente capítulo contém uma visão geral do trabalho e da sua motivação;
- O Capítulo 2 introduz o ambiente .NET, sob o ponto de vista do seu engenho de execução de código gerenciado. É descrita a sua arquitetura, principais recursos, o sistema de tipos, o ambiente virtual de execução com as principais instruções e os serviços de gerenciamento de memória;
- O Capítulo 3 fornece um panorama das técnicas mais utilizadas para integrar linguagens funcionais a plataformas de código gerenciado;
- O Capítulo 4 introduz o compilador Haskell.NET, sua arquitetura, ambiente de execução e estratégias de compilação;
- O Capítulo 5 analisa o compilador Haskell.NET sob o ponto de vista de desempenho do código gerado. São mostrados os resultados de análises de impacto de determinadas construções, comparações entre diferentes abordagens e otimizações;
- O Capítulo 6 trata da interoperabilidade de linguagens funcionais com ambientes como o .NET, analisando como ela pode ser atingida e mostrando exemplos de linguagens funcionais que a suportam;
- O Capítulo 7 aponta conclusões e possíveis trabalhos futuros;
- O Apêndice A contém os esquemas de compilação utilizados.

CAPÍTULO 2

INTRODUÇÃO À PLATAFORMA .NET

A Plataforma Microsoft .NET é uma plataforma de desenvolvimento multilinguagem composta por um grande conjunto de bibliotecas – para manipulação de banco de dados, processamento de XML, interface gráfica com o usuário, desenvolvimento Web, etc. – e um ambiente de execução – o *Common Language Runtime* [BS03], ou CLR.

A CLR pode ser definida como uma máquina virtual orientada a pilha, responsável por traduzir código escrito em *Common Intermediate Language* – CIL – para instruções nativas – específicas do processador. Essa tradução é executada sob demanda por meio de compilação *Just In Time* (JIT). Portanto, quando dizemos “compilar a linguagem X para a Plataforma .NET” queremos dizer “construir um compilador de X capaz de gerar código CIL”. Por “orientada a pilha” queremos dizer que as instruções da máquina podem empilhar operandos em uma pilha de avaliação, executar operações sobre esses operandos, empilhar os resultados dessas operações, mover operandos da pilha para variáveis e vice-versa.

A CLR possui várias outras funções como segurança de código, gerenciamento de memória através de coleta automática de lixo (“*garbage collection*”), gerenciamento de processos e *threads* e vários outros serviços, de uma forma independente de linguagem.

Um outro exemplo de máquina virtual é a JAVA Virtual Machine (JVM) [LY99], que já foi utilizada como alvo por algumas implementações de linguagens funcionais. Porém, a JVM não é uma plataforma ideal para linguagens funcionais devido a alguns fatores:

- é projetada especialmente para suportar as características de JAVA [GJSB00], portanto suportando essencialmente linguagens orientadas a objeto;
- não suporta *tail-calls* - um mecanismo de invocação de função que não armazena “*stack frames*”, comumente utilizado para a implementação eficiente de linguagens funcionais;
- não suporta nenhum tipo de ponteiro para funções ou outras características de mais baixo nível que, apesar de não serem essenciais, podem ser extremamente úteis para a compilação eficiente de linguagens funcionais.

Por outro lado, a Plataforma .NET, apesar do seu maior foco em orientação a objetos, foi projetada para suportar vários paradigmas de programação. Além disso, ela suporta *tail-calls* e ponteiros para função “seguros” (*type-safe*) através dos *delegates* (Seção 2.2.5). Finalmente, .NET oferece outras características não encontradas na Plataforma JAVA, tais como versionamento e interoperabilidade com outras linguagens. Portanto, a princípio parece ser uma plataforma mais promissora para a implementação de linguagens

funcionais do que a JVM. Comparações entre as duas plataformas podem ser encontrados nos trabalhos de Meijer e Gough [MG00, Gou01].

Neste capítulo, introduziremos a parte padronizada sobre a qual a Plataforma .NET é baseada, conhecida como *Common Language Infrastructure* (CLI) (Seções 2.1 – 2.6), bem como os principais componentes da plataforma em si (Seções 2.7 – 2.8). Grande parte das informações aqui apresentadas são baseadas na especificação da CLI [MR04] e em outras referências bibliográficas [BS03, MVBM04, Wie04].

2.1 COMMON LANGUAGE INFRASTRUCTURE

Por trás da Plataforma .NET existe um conjunto de padrões submetidos pela Microsoft Corporation ao ECMA International ¹ [ECM]. A versão mais recente foi atualizada em junho de 2005: a *Common Language Infrastructure* ou CLI [CLI]. A CLI tem por objetivo prover uma especificação para código executável e para o ambiente onde este código deve ser executado, denominado *Virtual Execution System* ou VES. Portanto, a Plataforma .NET faz uso de uma implementação da CLI – o *Common Language Runtime* (CLR).

O modelo de execução adotado pela CLI é mostrado na Figura 2.1². De fato, compiladores geram código na linguagem CIL, sigla para *Common Intermediate Language*. Não é especificado pelo padrão como esse código deve ser executado, ficando essa decisão a cargo da implementação. A opção mais utilizada é compilação *just-in-time* (JIT), em que o código nativo é gerado sob demanda, em tempo de execução. Entretanto, outras opções incluem a interpretação do código CIL ou a compilação para código nativo no momento da instalação do *software*. Também é possível a execução pelo ambiente de código nativo não gerenciado, conforme mostrado na Figura 2.1.

A especificação da CLI engloba quatro aspectos principais:

- *Common Type System* (CTS): sistema de tipos que suporta os tipos e operações mais comumente encontrados nas linguagens de programação, permitindo a integração entre diferentes linguagens e segurança de manipulação de tipos (“*type-safety*”).
- Metadados: elementos utilizados para descrever e referenciar tipos definidos por meio do CTS. São persistidos em um formato independente de linguagem de programação e podem ser acessados/manipulados por ferramentas diversas como compiladores, depuradores, geradores de código, etc.
- *Common Language Specification* (CLS): subconjunto do CTS e conjunto de convenções de uso que objetivam facilitar a interoperabilidade inter-linguagem, garantindo que apenas construções por ela definidas serão expostas por componentes de *software* utilizáveis em diferentes linguagens.
- *Virtual Execution System* (VES): implementa o modelo do CTS, carregando e executando código escrito para a CLI, e provendo serviços necessários ao gerenciamento

¹Organização européia fundada em 1961, voltada para a definição de padrões nas áreas de Tecnologia de Informação, Comunicação e Eletrônica.

²Adaptada de [MR04].

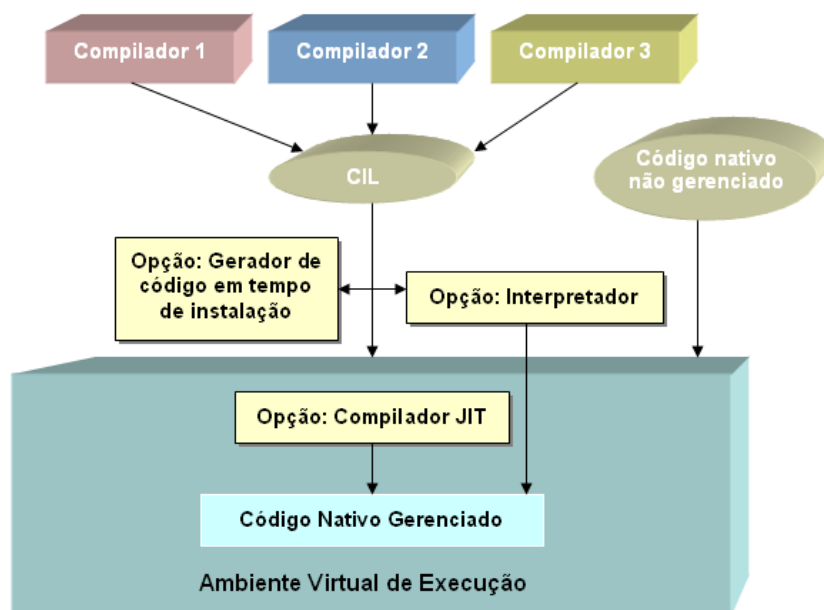


Figura 2.1. CIL – Modelo de Execução

de código e de memória. Por meio dos metadados, conecta diferentes módulos em tempo de execução.

Finalmente, os aspectos acima descritos são suficientes para o suporte à execução de aplicações e componentes distribuídos. A comunicação entre ferramentas e o ambiente virtual de execução por meio de metadados permite funcionalidades diversas como segurança, serviços remotos, *web services*, etc.

Para tornar possível a realização de sua infra-estrutura de execução, a CLI especifica um formato de arquivo e um conjunto de instruções em uma linguagem denominada *Common Intermediate Language (CIL)* (Seção 2.1.1), que devem, respectivamente, conter e representar o código a ser gerenciado e executado pela CLI. Compiladores devem obedecer ao formato e ao conjunto de instruções definido, bem como às suas regras de uso, para produzir código capaz de ser executado pela CLI.

2.1.1 CLI – Principais Conceitos

Aqui apresentaremos uma breve introdução aos conceitos básicos envolvidos no funcionamento da CLI e cujo entendimento guiará o restante da discussão.

Execução Guiada por Metadados. O acesso a metadados em tempo de execução permite, entre outras capacidades: o provimento uniforme de serviços para depuradores, *profilers* e compiladores, o gerenciamento de exceções, a garantia de segurança no acesso a código (*“Code Access Security”*), reflexão e serialização de objetos remotos. Outra vantagem que pode ser citada é a transparência em relação a detalhes de baixo nível como *offsets* e *layouts* físicos, uma vez que nomes podem ser utilizados.

Dessa forma, é possível que sejam realizadas otimizações devido à possibilidade de reorganização dos dados em memória.

Código Gerenciado. O código executado pela CLI fornece informações ao ambiente de execução, permitindo que o mesmo ofereça uma série de serviços. Exemplos de serviços são a localização dos metadados de um método a partir de um endereço em seu código, a monitoração da pilha de chamadas, a manipulação de exceções e o gerenciamento de segurança.

Dados Gerenciados. São dados cuja memória é alocada e liberada automaticamente pela CLI. A liberação de memória é gerenciada por um processo conhecido por *garbage collection* (“coleta de lixo”).

2.2 COMMON TYPE SYSTEM

De acordo com a especificação da CLI, um tipo descreve valores e especifica um contrato que deve ser seguido por todos os valores a ele pertencentes. O “*Common Type System*”(CTS), ou *Sistema Comum de Tipos*, suporta basicamente dois tipos de entidades: valores e objetos. Os primeiros nada mais são do que seqüências de bits comumente encontradas em linguagens de programação. Exemplos são inteiros, caracteres, números em ponto flutuante, enumerações (“*enums*”) e estruturas (“*structs*”) das linguagens C e Pascal. Objetos, por sua vez, são entidades cujos tipos são armazenados internamente em sua representação e possuem identificação única, que não pode ser alterada ao longo do ciclo de vida dos mesmos. Cada objeto pode armazenar outras entidades.

Os tipos de entidades suportados pelo CTS são mostrados na Figura 2.2³. Nas próximas seções detalharemos alguns deles.

2.2.1 Value Types

Acessados diretamente como padrões de bits, os *value types* caracterizam-se por serem armazenados na pilha do ambiente de execução, caso sejam argumentos, variáveis locais, valores de retorno de métodos ou campos de outros tipos por valor. Caso sejam campos de instâncias de *reference types* (Seção 2.2.2), são alocados dinamicamente e armazenados na *heap* – uma região de memória gerenciada pelo ambiente de execução. Podem ser agrupados em:

Pré-definidos: também conhecidos como “*built-in*”, são os tipos numéricos (inteiros e ponto flutuante), com ou sem sinal.

Definidos pelo Usuário: são as enumerações (“*enum*”) e estruturas (“*struct*”). Enumerações são sinônimos de um tipo existente, obrigatoriamente um tipo numérico inteiro. Algumas restrições se aplicam, como por exemplo o fato de não poderem declarar métodos, propriedades, eventos ou implementar interfaces. Herdam do tipo `System.Enum`. Um exemplo de enumeração na linguagem C# e o resultado de sua compilação para CIL podem ser vistos nas figuras 2.3 e 2.4, respectivamente.

³Adaptada de [MR04].

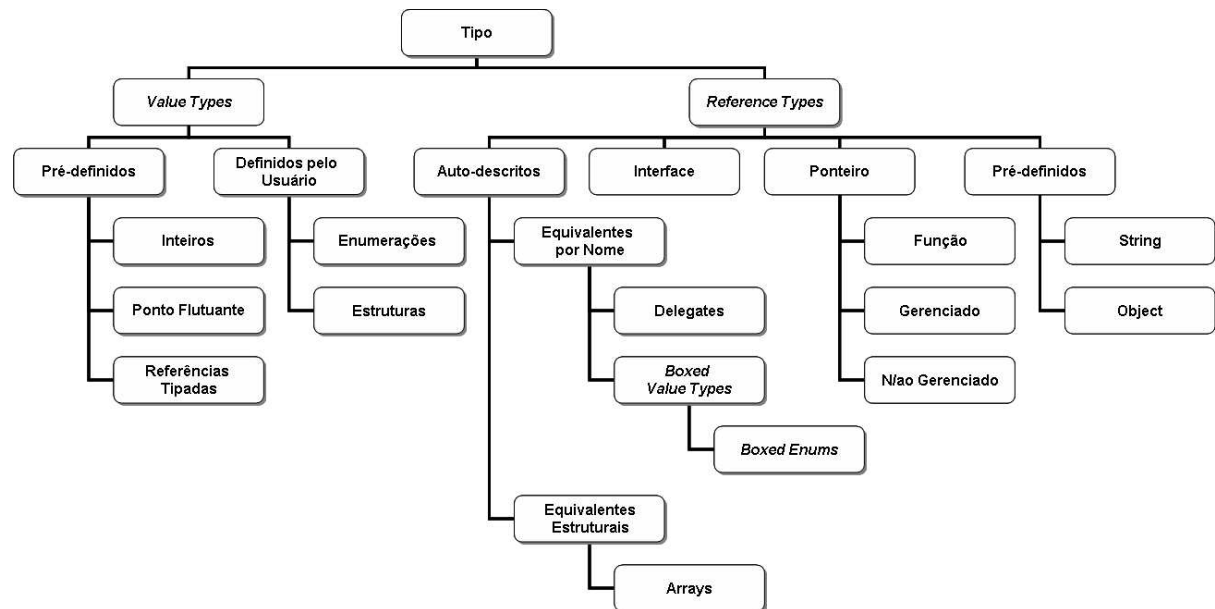


Figura 2.2. *Common Type System*

```

enum Week
{
    Sunday,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
}
  
```

Figura 2.3. Exemplo de Enumeração em C#

```
.class private auto ansi sealed Week extends [mscorlib]System.Enum
{
    .field public specialname rtspecialname int32 value__
    .field public static literal valuetype Week Sunday = int32(0x00000000)
    .field public static literal valuetype Week Monday = int32(0x00000001)
    .field public static literal valuetype Week Tuesday = int32(0x00000002)
    .field public static literal valuetype Week Wednesday = int32(0x00000003)
    .field public static literal valuetype Week Thursday = int32(0x00000004)
    .field public static literal valuetype Week Friday = int32(0x00000005)
    .field public static literal valuetype Week Saturday = int32(0x00000006)
} // end of class Week
```

Figura 2.4. Exemplo de Enumeração em CIL

```
struct Address
{
    public string street;
    public int zip;
    public string city;
    public string country;
}
```

Figura 2.5. Exemplo de Estrutura em C#

Como pode ser visto na Figura 2.4, enumerações possuem um campo de instância, do tipo inteiro correspondente. Já as estruturas assemelham-se a registros em Pascal e *structs* em C. Assemelham-se a classes com relação à estrutura, com exceção do fato de não poderem ser herdadas e de outras restrições. Herdam do tipo `System.ValueType`. Exemplos em C# e CIL são apresentados nas figuras 2.5 e 2.6, respectivamente.

2.2.2 Reference Types

Reference Types descrevem valores que são representados pela *localização* de uma sequência de bits. Agrupam-se em:

Objetos: representam valores que se “auto-descrevem” por meio de metadados, total ou parcialmente (tipos abstratos).

Interfaces: consistem em descrições sempre parciais de valores e podem ser suportados por vários tipos. Uma interface define um contrato, através de um conjunto de métodos que devem ser implementados por todo tipo que declarar suportá-la.

Ponteiros: representados por endereços de máquina de um local em memória.

```

.class sequential ansi sealed beforefieldinit Address
    extends [mscorlib]System.ValueType
{
    .field public string street
    .field public int32 zip
    .field public string city
    .field public string country
} // end of class Address

```

Figura 2.6. Exemplo de Estrutura em CIL

Pré-definidos: devem ser suportados nativamente pelo ambiente de execução. Exemplos incluem a classe `System.Object`, superclasse de todos os tipos de objetos, e a classe `System.String`, para representação de cadeias de caracteres.

2.2.3 Boxing e Unboxing

Para permitir interoperabilidade com código que manipula apenas *reference types*, como objetos, todo *value type* define um *reference type* correspondente – o tipo *boxed*. A operação responsável por gerar o objeto do *reference type* correspondente ao elemento do tipo por valor denomina-se *boxing*. Analogamente, a operação inversa denomina-se *unboxing*. Assim, chamamos instâncias de *value types* de valores *unboxed*. Cada *value type* suporta uma instrução, denominada `box`, responsável por gerar na *heap* um objeto que encapsulará uma cópia dos bits que compõem o valor *unboxed*. A instrução inversa – `unbox` – retorna um ponteiro gerenciado para tal representação em bits do valor.

Abaixo, temos um exemplo em C# das operações de *boxing* e *unboxing*. Na linha 2, um valor inteiro (*value type*) é atribuído a um objeto (*reference type*). Implicitamente, a conversão ocorre. Em seguida, na linha 3, a referência para o objeto é atribuída a uma variável do tipo inteiro. Neste caso, a conversão, que poderia implicar perda de informações, precisa ser explícita.

```

1  int x = 2;
2  object o = x;
3  int y = (int)o;

```

A seguir, temos as instruções CIL que são geradas para a compilação do programa mostrado como exemplo. Na linha 4, a instrução `box` converte o elemento no topo da pilha, do tipo `System.Int32` (*value type* que representa inteiros *unboxed* de 32 bits com sinal), para um objeto (cuja representação exata é de conhecimento apenas da CLI). Finalmente, na linha 7, a instrução `unbox` obtém o valor armazenado no objeto presente no topo da pilha e o atribui à variável `y`, do tipo `System.Int32`. Embora não seja explícito no código gerado, o que essa instrução retorna de fato na pilha é um ponteiro para o valor, porém este será automaticamente desreferenciado.

```
1 ldc.i4.2
2 stloc x
3 ldloc x
4 box [mscorlib]System.Int32
5 stloc o
6 ldloc o
7 unbox.any [mscorlib]System.Int32
8 stloc y
```

2.2.4 Ponteiros

O suporte a um dos elementos mais comumente encontrados em linguagens imperativas dá-se através de duas formas: ponteiros gerenciados e não gerenciados.

Ao contrário das referências, que apontam para a porção inicial dos objetos, os ponteiros gerenciados apontam para o interior dos mesmos, em geral para um dos seus campos. Da mesma forma que as referências, são ajustados quando um objeto é movido durante o processo de coleta de memória. Entretanto, possuem como restrição o fato de somente serem permitidos em parâmetros/retorno de métodos e variáveis locais. Este tipo de ponteiro costuma ser utilizado para a implementação de mecanismos de passagem de parâmetro por referência. Podem apontar não apenas para objetos mas também para instâncias de *value types* armazenados dentro de um escopo local.

Já os ponteiros não gerenciados foram projetados principalmente para interoperabilidade com código nativo, bem como para acomodar as necessidades de linguagens como C/C++. São suportadas operações especiais para leitura e modificação dos valores apontados, bem como operações aritméticas. Ponteiros não gerenciados são ignorados pelo gerenciamento automático de memória da CLI (*garbage collector*).

2.2.5 Delegates

Delegates são a versão segura e orientada a objetos de ponteiros para função. Um *delegate* nada mais é do que um objeto que encapsula um ponteiro para um método a ser invocado, bem como uma referência para o objeto alvo no qual o método deverá ser chamado. Sua criação se dá em princípio pela definição de uma classe que herda de `System.Delegate` e que declara um método denominado `Invoke`, responsável por invocar o método referenciado, devendo ter assinatura idêntica à do mesmo. Além desse, existem os métodos `BeginInvoke` e `EndInvoke`, utilizados para chamada *assíncrona* do método referenciado pelo *delegate*.

Uma outra funcionalidade presente em *delegates*, que não é possível através de ponteiros para função, é que os mesmos podem referenciar vários métodos, mantendo internamente uma lista. Desta forma, todos os métodos são chamados quando da invocação de `Invoke`. A ordem de chamada segue a ordem de inserção na lista. Tal capacidade permite que *delegates* sejam perfeitamente adequados ao modelo de notificação de eventos da CLI, adotados em particular por componentes de interface gráfica.

Na Figura 2.7 temos um exemplo da definição e uso de *delegates* em C#. Na linha 1 definimos um *delegate* específico para métodos de assinatura $\text{int} \times \text{int} \rightarrow \text{int}$, ou

seja, que recebem dois inteiros como parâmetros e retornam um inteiro. A definição do *delegate* cria um *tipo*, o qual denominamos aqui de **Function**. Na linha 4 instanciamos o *delegate* com um método estático. Já na linha 6, instanciamos o mesmo com um método de instância, indicando ao mesmo tempo a instância à qual se deve referir o método passado como parâmetro. Nas linhas 7 e 9, invocamos os *delegates* instanciados com a mesma sintaxe que seria utilizada para chamada normal de método.

```

1 public delegate int Function(int x, int y);
2 class TestDelegates {
3     public static void Main() {
4         Function function1 = new Function(Sum);
5         TestDelegates testDelegates = new TestDelegates();
6         Function function2 = new Function(testDelegates.Subtract);
7         int result1 = function1(10, 20);
8         Console.WriteLine(result1);
9         int result2 = function2(30, 40);
10        Console.WriteLine(result2);
11    }
12    static int Sum(int x, int y) { return x + y;}
13    int Subtract(int a, int b) {return a - b;}
14 }
Saída obtida:
30
-10

```

Figura 2.7. Delegates em C#

Na Figura 2.8, é mostrado o resultado da compilação do *delegate* **Function**. Nela podemos ver os métodos **Invoke**, **BeginInvoke** e **EndInvoke**. Deve-se notar que o construtor e todos os métodos estão marcados com a diretiva **runtime managed**, o que significa que seus códigos são gerados dinamicamente pela CLI e não pelo compilador ou programador. Pode-se perceber que **Invoke** possui exatamente a mesma assinatura especificada na declaração do *delegate*, e que os demais métodos utilizam parâmetros dos tipos **System.AsyncCallback** e **System.IAsyncResult**, ambos utilizados para encapsular informações relacionadas a chamadas assíncronas.

Já na Figura 2.9, podemos observar as principais instruções CIL geradas a partir do código na Figura 2.7. As linhas 1–3 estão relacionadas à instanciação do *delegate* referenciado pela variável **function1**. Nelas pode-se perceber que o primeiro parâmetro do construtor do *delegate*, invocado na linha 3, é **null** (instrução **ldnull** na linha 1), refletindo o fato de que o *delegate* referencia um método estático. A instrução **ldftn**, na linha 2, é responsável por empilhar um ponteiro para o código do método. Instruções análogas são utilizadas para instanciar o *delegate* referenciado por **function2**, com a diferença que, para este, a referência para o objeto alvo é passada para o construtor (linha 9). As chamadas aos *delegates* são traduzidas para chamadas a **Invoke** (linhas 14

```

.class public auto ansi sealed Function
    extends [mscorlib]System.MulticastDelegate {
    .method public hidebysig specialname rtspecialname
        instance void .ctor(object 'object', native int 'method')
        runtime managed { }
    .method public hidebysig newslot virtual
        instance int32 Invoke(int32 x, int32 y)
        runtime managed { }
    .method public hidebysig newslot virtual
        instance class [mscorlib]System.IAsyncResult
        BeginInvoke(int32 x, int32 y,
            class [mscorlib]System.AsyncCallback callback,
            object 'object') runtime managed { }
    .method public hidebysig newslot virtual
        instance int32 EndInvoke(
            class [mscorlib]System.IAsyncResult result)
        runtime managed { }
}

```

Figura 2.8. Delegate Function em CIL

e 19), sendo portanto semelhantes a qualquer chamada a método virtual.

2.2.6 Generics

Nesta seção, descreveremos um recurso do sistema de tipos introduzidos a partir da versão 2.0 da CLI: *Generics*, também conhecido por *polimorfismo paramétrico*.

A idéia do polimorfismo paramétrico foi bastante difundida entre as linguagens funcionais e consiste em escrever código genérico aplicável a argumentos de quaisquer tipos. Entretanto, essa idéia foi incorporada em outros paradigmas, como o imperativo e o orientado a objetos. No primeiro, temos como exemplo a técnica de *templates* da linguagem C++. No último, é comum encontrarmos a implementação de polimorfismo paramétrico a partir do uso de supertipos comuns (ex.: `Object` em JAVA, até a versão 1.4, e nas primeiras versões da CLI).

Na Figura 2.10 temos um exemplo de polimorfismo paramétrico, em C#, que segue a abordagem tradicional de uso de um supertipo comum. O exemplo mostra a implementação de uma pilha, cujos elementos podem ser de qualquer tipo.

A abordagem mostrada na Figura 2.10 apresenta algumas desvantagens, dentre elas:

- menor legibilidade do código, uma vez que conversões explícitas (*casts*) devem ser utilizadas ao desempilhar elementos cujos tipos declarados não sejam `Object`;
- menor robustez, devido à grande possibilidade de erros em tempo de execução causados pelas conversões explícitas citadas no item anterior, em situações em que o tipo de destino não seja compatível com o tipo real do objeto retornado;

```

1 ldnull
2 ldftn int32 TestesDotNet.TestDelegates::Sum(int32, int32)
3 newobj instance void Function::.ctor(object, native int)
4 stloc function1
5 newobj instance void TestDelegates::.ctor()
6 stloc testDelegates
7 ldloc testDelegates
8 ldftn instance int32 TestDelegates::Subtract(int32, int32)
9 newobj instance void Function::.ctor(object, native int)
10 stloc function2
11 ldloc function1
12 ldc.i4.s 10
13 ldc.i4.s 20
14 callvirt instance int32 Function::Invoke(int32, int32)
15 stloc result1
...
16 ldloc function2
17 ldc.i4.s 30
18 ldc.i4.s 40
19 callvirt instance int32 Function::Invoke(int32, int32)
20 stloc result2
...

```

Figura 2.9. Uso de *Delegates* em CIL

- menor desempenho, devido às checagens em tempo de execução que são feitas pelas conversões explícitas e às operações de *boxing/unboxing* necessárias para operar com valores de tipos numéricos *built-in*, por exemplo.

A solução para os problemas citados acima está em determinar ainda em tempo de compilação sobre qual tipo o código será executado, ou seja, ao instanciar um objeto do tipo `Stack`, declarar explicitamente o tipo dos objetos que por ele serão operados. Denominamos essa técnica de *Generics*. Um exemplo de como ficaria a classe `Stack` e de como ela seria instanciada é mostrado na Figura 2.11. Uma variável de tipo (`T`) passa a ser utilizada para representar o tipo genérico sobre o qual a classe deve operar. No exemplo mostrado, vemos que verificações são feitas em tempo de compilação, evitando erros em tempo de execução e tornando desnecessárias conversões explícitas de tipos.

O exemplo da Figura 2.11 mostrou apenas uma das formas de utilização de *Generics*, em que apenas uma variável de tipo é utilizada. Outras formas incluem utilização de mais de uma variável de tipo e métodos genéricos (ex.: `void Push<T>(T elem)`). É possível ainda definir restrições que limitam os tipos a serem suportados, como em:

```
public class Stack<T> where T:IEnumerable {...}
```

```
public class Stack {
    object[] elements;
    int next;
    public Stack() {
        //código de inicialização
    }
    public void Push(object elem) {
        //...
        /* verificação do tamanho e alocação de um array
        maior, se necessário */
        //...
        elements[next++] = elem;
    }
    public object Pop() {
        return elements[next - 1];
    }
    public static void Main() {
        Stack stack1 = new Stack();
        stack1.Push(2);
        int value1 = (int) stack1.Pop();
        Stack stack2 = new Stack();
        stack2.Push("Generics");
        string value2 = (string) stack2.Pop();
        stack1.Push("Test");
        int value3 = (int) stack2.Pop(); //erro em tempo de execução
    }
}
```

Figura 2.10. Polimorfismo Paramétrico pela Abordagem Tradicional

No exemplo acima, a classe `Stack` só pode ser utilizada com tipos que implementem a interface `IEnumerable`.

2.2.7 Segurança de Tipos e Verificação

No contexto da especificação da CLI, tipos especificam *contratos* que definem desde regras de visibilidade até operações suportadas e mapeamentos de nomes em células de memória. Desta forma, um programa é dito *type-safe* apenas se obedecer a todos os contratos dos tipos utilizados. Por este ser um conceito muito restritivo, uma vez que nem todas as linguagens suportam *apenas* construções *type-safe*, processos de verificação automatizada operam sobre o conceito de *memory safety* ou segurança de acesso à memória.

Conforme pode ser visto na Figura 2.12⁴, podemos distinguir as classes dos programas *memory-safe* e *verificáveis*. *Verificação* nada mais é do que um processo mecanizado que

⁴Adaptada de [MR04].

```

public class Stack<T> {
    T[] elements;
    int next;
    public Stack() {
        //código de inicialização
    }
    public void Push(T elem) {
        //...
        /* verificação do tamanho e alocação de um array
        maior, se necessário */
        //...
        elements[next++] = elem;
    }
    public T Pop() {
        return elements[next - 1];
    }
    public static void Main() {
        Stack<int> stack1 = new Stack<int>();
        stack1.Push(2);
        int value1 = stack1.Pop();
        Stack<string> stack2 = new Stack<string>();
        stack2.Push("Generics");
        string value2 = stack2.Pop();
        stack1.Push("Test"); //erro de compilação
        int value3 = stack2.Pop(); //erro de compilação
    }
}

```

Figura 2.11. Polimorfismo Paramétrico através de *Generics*

visa a *garantir* que um programa acesse a memória de forma segura, dentro do espaço lógico de endereços permitido, de modo que nenhum código que não seja *memory-safe* possa ser considerado verificável. Por outro lado, é possível que um programa correto e seguro não seja considerado verificável, uma vez que não é possível computacionalmente termos um processo de verificação correto para todos os programas. Como exemplo, temos programas escritos em C/C++ que fazem operações diretas em memória, como por exemplo aritmética de ponteiros. As regras a serem utilizadas pelos verificadores são ditadas pela especificação da CLI, porém o momento em que o algoritmo de verificação deve ser executado e o comportamento da máquina virtual diante de falhas de verificação não são especificados. Em geral, uma implementação da CLI deve suportar a execução de código não verificável, embora sob circunstâncias sujeitas a permissão administrativa.

Já a *validação* refere-se apenas a assegurar que o formato de arquivo, as instruções e os metadados estão consistentes e em conformidade com o que é determinado pela CLI.

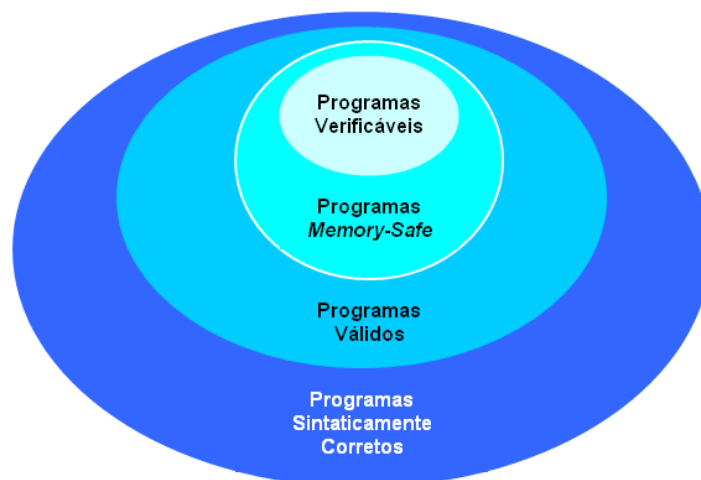


Figura 2.12. Relacionamento entre Programas Válidos e Verificáveis

2.3 COMMON LANGUAGE SPECIFICATION

A *Common Language Specification*, ou Especificação Comum de Linguagens, é um subconjunto do CTS e também um conjunto de convenções de uso que objetivam facilitar a interoperabilidade entre linguagens, garantindo que apenas construções por ela definidas serão expostas por componentes de *software* usados por diferentes linguagens. As regras definidas pela CLS aplicam-se apenas a tipos e membros acessíveis externamente ao módulo onde foram definidos.

A motivação por trás da CLS provém da própria natureza multi-linguagem da CLI, pois é desejado que *frameworks* de propósito geral possam ser utilizados com completa independência de linguagem. O conjunto de regras de conformidade ditadas pela CLS força algumas linguagens a serem estendidas com novas características e, da mesma forma, leva outras a não exporem características particulares que não são cobertas pela especificação. Ao mesmo tempo, o conjunto de elementos que podem ser expostos por *frameworks* é limitado apenas ao conjunto de construções permitidas pela especificação.

A CLS possui atualmente três visões de conformidade diferentes. São elas: *CLS Framework*, *CLS Consumer* e *CLS Extender*.

No contexto da CLS, *frameworks* são bibliotecas que expõem apenas código em conformidade com a CLS, ou *CLS-compliant*. Desta forma, seus autores garantem que tais bibliotecas poderão ser acessadas por um grande número de linguagens e ferramentas. Exemplos de regras que envolvem a criação de *frameworks* são, por exemplo, a não exposição de nomes comumente utilizados como palavras-chave por linguagens de programação, ou de identificadores diferentes apenas com relação ao fato de seus caracteres serem maiúsculos/minúsculos.

CLS consumers são ferramentas ou linguagens capazes de acessar quaisquer *frameworks* em conformidade com a CLS. Entretanto não é exigido desses consumidores que eles sejam capazes de estender *frameworks*. Exemplos de regras necessárias para que

uma linguagem ou ferramentas seja considerada CLS *consumer* são o suporte à chamada de qualquer método ou *delegate* CLS-*complaint*, criação de instâncias de qualquer tipo CLS-*complaint* ou acesso a qualquer membro de tipo CLS-*complaint*. Devem suportar ainda mecanismos de chamada a métodos cujos identificadores sejam palavras-chave em sua sintaxe.

CLS *extenders* são linguagens ou ferramentas às quais são aplicadas as mesmas regras para CLS *consumers*, com a adição da capacidade de extensão de *frameworks* existentes. Devem ser capazes de, entre outras funcionalidades, permitir a definição de novos tipos CLS-*complaint* (com exceção de interfaces e tipos aninhados).

2.4 METADADOS

Com o objetivo de tornar possível o gerenciamento do código e do ambiente de execução, a CLI faz uso de uma representação estruturada da informação necessária para carregar classes, invocar métodos, acessar campos, garantir segurança e código e outras funcionalidades. Tal estrutura, sempre utilizada na declaração de novos tipos, é o que conhecemos por metadados. De forma resumida, podemos dizer que o objetivo básico dos metadados é descrever tipos, métodos, campos, etc. Vários tipos de ferramentas e bibliotecas beneficiam-se da presença de tal descrição, entre elas compiladores – que necessitam percorrer metadados de tipos expostos a um programa – e a API de Reflexão.

Nos componentes, em geral agrupados em *assemblies* (Seção 2.4.1), os metadados equivalem aos já conhecidos arquivos de cabeçalho, arquivos IDL e *typelibs* de plataformas como COM [Rog97] e CORBA [COR]. Porém, pelo fato de serem embutidos diretamente no código CIL, permitem que componentes sejam auto-descritos, evitando assim os problemas inerentes à dependência em relação a arquivos externos. A forma pela qual metadados são embutidos no código se dá através dos chamados *tokens* – identificadores de elementos de metadados (ex.: declarações de tipos ou membros). Em tempo de execução, os metadados são carregados e organizados em *tabelas*. Embora os *tokens* descrevam tipos, em geral sua representação interna consiste apenas de índices para as estruturas de metadados mantidas em memória.

Até mesmo a interoperabilidade com código não gerenciado é beneficiada pela existência de metadados, que permitem descrever estruturas nativas de forma “aparentemente” gerenciada, além de facilitarem os mecanismos automáticos de *marshalling*.

2.4.1 Assemblies, Módulos e Manifestos

Existem no contexto da CLI duas estruturas básicas de agrupamento de tipos, código ou recursos: *assemblies* e módulos. Os primeiros são a unidade mínima de instalação (“*deployment*”) e o menor agrupamento necessário para constituir um componente, seja ele um executável ou uma biblioteca de classes. Em geral encapsulam um conjunto coeso de funcionalidades. Já os últimos são os arquivos propriamente ditos que contêm código CIL e metadados. Um *assembly* é composto por um ou mais módulos, dentre os quais exatamente um deve conter o *manifesto*, uma seção que identifica:

- o nome, versão, idioma e requisitos de segurança do *assembly*;

- arquivos – módulos e outros recursos – que constituem o *assembly*, bem como os tipos que devem ser expostos pelos módulos constituintes e códigos *hash* criptográficos para garantir a integridade dos mesmos;
- assinatura digital e chave pública.

Um módulo, por sua vez, pode estar contido em mais de um *assembly*.

A informação referente à versão do *assembly* constitui a base do sofisticado mecanismo de versionamento empregado pelas implementações da CLI. Tal mecanismo é conhecido por resolver alguns dos problemas relacionados a diferentes versões de bibliotecas compartilhadas, uma vez que permite que diferentes versões de um mesmo componente coexistam no sistema.

2.5 AMBIENTE VIRTUAL DE EXECUÇÃO

Nesta seção será introduzido o Ambiente Virtual de Execução (*Virtual Execution System*) da CLI, cujo objetivo básico é fornecer um ambiente de execução para código gerenciado. Uma implementação de tal ambiente deve obedecer à especificação determinada pela CLI [MR04], sendo livre entretanto para utilizar qualquer técnica de execução do código. A técnica mais comum, adotada pelo ambiente .NET, é a compilação *just-in-time*, embora seja possível uma implementação baseada em interpretação do código gerenciado, por exemplo.

2.5.1 Máquina Virtual

O modelo de execução definido pela CLI é orientado a pilha, ou seja, utiliza uma pilha de avaliação manipulada pelas instruções CIL. Tais instruções permitem, basicamente:

- copiar dados da memória para a pilha e vice-versa;
- realizar operações cujos operandos são obtidos da pilha e cujos resultados nela são armazenados;
- invocar métodos cujos argumentos são depositados na pilha de avaliação e cujos resultados nela são armazenados.

A CLI requer que apenas os tipos `int32`, `int64`, os apontadores em geral, o tipo interno usado para representação de números em ponto flutuante e *value types* definidos pelo usuário sejam diretamente armazenáveis na pilha de avaliação. Inteiros de tamanho definido pela plataforma (`native int`) são utilizados para a representação de ponteiros. Portanto, inteiros com menos de 4 *bytes* devem ser automaticamente expandidos (com ou sem sinal) para `int32` antes de serem guardados na pilha de avaliação e valores recuperados da pilha de avaliação devem ser convertidos para seus tipos de destino ao serem armazenados em memória. É possível ainda que a implementação suporte um conjunto maior de tipos na pilha de avaliação e portanto não realize tais conversões.

Com relação ao funcionamento do ambiente virtual de execução, a CLI não especifica os detalhes referentes a convenções de chamada de procedimentos ou *layout* de *frames*

de chamada. Por outro lado, existe uma especificação abstrata dos estados da máquina virtual, compostos basicamente por *estado global* e *estado de método*.

O estado global consiste de um conjunto de *threads* concorrentes e múltiplas *heaps* gerenciadas e compartilhados. Já o estado de método nada mais é do que uma representação abstrata dos *stack frames* encontrados na tecnologia tradicional dos compiladores para código nativo. Cada estado de método é composto basicamente por:

- um ponteiro de instrução que referencia a próxima instrução a ser executada pelo método;
- uma pilha de avaliação;
- um *array* de variáveis locais;
- um *array* de argumentos;
- metadados que descrevem a assinatura do método, suas variáveis locais e as exceções por ele levantadas;
- um *pool* local de memória, onde são alocados dinamicamente objetos que são coletados ao final da execução do método (o suporte a esse tipo de estrutura não é requerido pela CLI);
- uma referência para o estado do método invocador;
- um descritor de segurança que contém informações relativas às permissões do método;

É importante ressaltar que o modelo acima é abstrato, o que significa dizer que, por exemplo, o *array* de argumentos pode ser compartilhado com a pilha de avaliação para evitar cópias. Quaisquer otimizações a esse modelo ficam à escolha da implementação do ambiente virtual de execução.

2.5.2 Mecanismos de Passagem de Parâmetros

Diferentemente do que ocorre em outros ambientes de execução, como a JVM, em que apenas um mecanismo de passagem de parâmetros é suportado (ie.: por valor), a CLI, com o objetivo de atender às necessidades do maior número possível de linguagens, define três mecanismos, explicitados a seguir.

Valor: De forma semelhante ao mecanismo tradicional de passagem de parâmetro por valor encontrado em um grande número de linguagens, o valor do parâmetro é simplesmente copiado para a pilha de avaliação. As modificações sofridas pelo parâmetro não são refletidas em seu valor original após o retorno do método invocado.

Referência: Neste mecanismo, o endereço do parâmetro é copiado para a pilha de avaliação e, portanto, modificações sofridas pelo mesmo são refletidas em seu valor original. É utilizado o tipo `&` (ponteiro gerenciado) para o ponteiro que armazenará o endereço do parâmetro.

Referência Tipada: Este mecanismo de passagem de parâmetro é útil para linguagens dinamicamente tipadas, pois nele o tipo estático do parâmetro real é passado juntamente com o seu endereço. Existem instruções CIL específicas tanto para instanciar uma referência tipada como para recuperar o tipo/endereço por ela encapsulado.

2.6 INSTRUÇÕES

Esta seção introduz os principais conceitos relacionados às instruções da CIL. Descreveremos os tipos sobre os quais elas podem operar e os principais grupos de instruções.

2.6.1 Tipos de Dados

Além do *Common Type System* (CTS), utilizado por metadados e programadores, existem dois outros sistemas mais simplificados:

- o sistema de tipos utilizado por verificadores, capaz de checar relacionamentos de subtipo e distinguir entre diferentes tipos por valor, porém incapaz de distinguir entre números com ou sem sinal, por exemplo;
- o sistema de tipos manipulado pela CLI através de suas instruções. Os tipos que compõem tal sistema englobam tipos básicos (numéricos, apontadores, etc.) além de todos os tipos por valor, que podem ser distinguidos entre si. É importante ressaltar que todos os objetos são vistos apenas como referências, não havendo distinção entre seus tipos nesse sistema.

Os tipos de dados suportados pelo conjunto de instruções da CLI agrupam-se basicamente em:

- **numéricos:** inteiros com ou sem sinal (`int8`, `unsigned int8`, `int16`, `unsigned int16`, `int32`, `unsigned int32`, `int64`, `unsigned int64`), inteiros cujos tamanhos dependem da plataforma de *hardware* (`native int` e `native unsigned int`), números de ponto flutuante de tamanho fixo (`float32`, `float64`) e nativo (`F`, utilizado apenas internamente);
- **ponteiros:** referências para objetos na memória gerenciada (`O`) e ponteiro gerenciado (`&`) que pode ou não referenciar memória gerenciada. Os tamanhos desses ponteiros são definidos de acordo com a plataforma de *hardware*.

2.6.2 Principais Grupos de Instruções

As instruções da CIL podem ser agrupadas de acordo com a finalidade. Podemos classificá-las em basicamente 4 (quatro) tipos principais: operações de *load* e *store*, aritméticas, de controle de fluxo e operações do modelo de objetos.

Operações de *Load* e *Store*. O grupo mais básico de instruções na CIL são aquelas relacionadas à transferência de elementos entre a pilha de avaliação e posições

memória. As instruções responsáveis por salvar valores no topo da pilha são conhecidas por instruções de *load* – iniciadas pelo prefixo “ld” – enquanto aquelas que carregam valores do topo da pilha em células de memória são denominadas instruções de *store* – iniciadas pelo prefixo “st”. A tabela 2.1 mostra algumas das principais instruções pertencentes a este grupo. Algumas delas – `ldc`, `ldind` e `stind` – requerem um sufixo indicando o tipo sobre o qual elas irão operar. As demais podem ser aplicadas a qualquer tipo, sem distinção.

| Instrução | Significado |
|---------------------|--|
| <code>ldc</code> | Carrega uma constante no topo da pilha. |
| <code>ldnull</code> | Carrega o literal null no topo da pilha. |
| <code>ldarg</code> | Carrega o valor de um argumento no topo da pilha. |
| <code>starg</code> | Salva o valor no topo da pilha em um argumento. |
| <code>ldloc</code> | Carrega o valor de uma variável local no topo da pilha. |
| <code>stloc</code> | Salva o valor no topo da pilha em uma variável local. |
| <code>ldind</code> | Carrega na pilha o valor apontado pelo endereço em seu topo. |
| <code>stind</code> | Salva um valor em um endereço de memória. |
| <code>ldftn</code> | Carrega um ponteiro para um método no topo da pilha. |

Tabela 2.1. Exemplos de Instruções de *Load* e *Store*

Aritméticas. As instruções aritméticas (`add`, `sub`, `mul`, `div`, etc.) são sobrecarregadas para os diferentes tipos numéricos, não sendo necessário nenhum tipo de sufixo ou indicador para especificar o tipo dos operandos, ao contrário do que ocorre na JVM. Em outras palavras, a implementação para código nativo é responsável por identificar qual operação de fato será realizada de acordo com os argumentos presentes na pilha, que devem ser do mesmo tipo (com exceção de `int32` e `native int`, que podem ser operados entre si). Adicionalmente, podem incluir um sufixo `.ovf`, que indica que uma exceção deve ser gerada caso o resultado da operação não possa ser representado no tipo destino. Esse é um exemplo de facilidade de suporte a várias linguagens; a JVM, por exemplo, não permite esse tipo de distinção.

Controle de fluxo. São as instruções de desvio condicional e não condicional comumente encontradas nas CPUs e máquinas virtuais, além das instruções de chamada de métodos. Essas últimas, mostradas na Tabela 2.2, agrupam-se em instruções de chamada a métodos virtuais e não virtuais, podendo ser adicionadas de um prefixo – `.tail` – que indica que o método atual deve liberar a memória alocada ao seu registro de ativação (“*stack frame*”) antes de executar uma chamada.

Modelo de Objetos. Alguns exemplos de instruções pertencentes a este grupo são mostrados na Tabela 2.3.

Ao contrário das instruções básicas, que em geral correspondem diretamente a instruções presentes na CPU, as instruções do modelo de objetos em geral são suportadas por composição das primeiras e/ou por chamadas ao sistema operacional. Em

| Instrução | Significado |
|-----------------------|--|
| <code>call</code> | Chama um método, virtual ou não, determinado estaticamente. |
| <code>calli</code> | Chama um método referenciado por um ponteiro. |
| <code>callvirt</code> | Chama um método virtual determinado em tempo de execução. |
| <code>jmp</code> | Desvia para a primeira instrução de um método. A convenção de chamada e assinatura do método de destino devem ser iguais às do método atual. |

Tabela 2.2. Exemplos de Instruções de Chamada a Métodos

| Instrução | Significado |
|------------------------|---|
| <code>box</code> | Converte um <i>value type</i> em uma referência. |
| <code>unbox</code> | Converte um valor que sofreu <i>boxing</i> para sua forma original. |
| <code>castclass</code> | Converte um objeto para uma classe (exceção em caso de falha). |
| <code>isinst</code> | Testa se um objeto é instância de uma classe. |
| <code>ldfld</code> | Carrega no topo da pilha um campo de um objeto. |
| <code>ldsfld</code> | Carrega no topo da pilha um campo de uma classe. |
| <code>ldstr</code> | Carrega no topo da pilha uma string. |
| <code>stfld</code> | Salva como campo de um objeto o valor no topo da pilha. |
| <code>stsfld</code> | Salva como campo de uma classe o valor no topo da pilha. |

Tabela 2.3. Exemplos de Instruções do Modelo de Objetos

geral, as instruções do modelo de objetos, assim como as de chamada a métodos, são complementadas com os chamados *tokens* de metadados, que identificam sobre quais tipos elas devem operar (ex.: `newobj instance Function::.ctor(object, native int)`).

2.7 PLATAFORMA MICROSOFT .NET – VISÃO GERAL

A Plataforma Microsoft .NET é uma plataforma de desenvolvimento e execução de código gerenciado composta basicamente por uma implementação da CLI – o *Common Language Runtime* – e por uma biblioteca orientada a objetos para funcionalidades diversas. Atualmente, a CLR é a implementação da CLI mais conhecida e utilizada.

A arquitetura da Plataforma .NET é mostrada na Figura 2.13. Nos próximos parágrafos resumiremos o significado de cada um dos componentes nela presentes. Para este trabalho, não entraremos em detalhes sobre todos os componentes, uma vez que apenas um deles é relevante para o restante da discussão.

Common Language Runtime. Composta principalmente pela máquina virtual e ambiente de execução responsáveis por gerenciar e executar o código gerenciado. Esse ambiente será analisado em maiores detalhes na Seção 2.8.

Biblioteca de Classes Básicas. Biblioteca padrão definida pela especificação da CLI.



Figura 2.13. Plataforma .NET – Arquitetura

Disponibiliza as funcionalidades básicas de acesso a arquivo, manipulação de *strings*, coleções, etc.

ADO .NET. API para acesso a dados de fontes diversas (ex.: banco de dados, XML).

Windows Forms. Biblioteca para desenvolvimento de interfaces gráfica para *desktop*. Apesar de serem baseados na API gráfica da plataforma Windows, seus componentes são gerenciados e fornecem uma camada de abstração que facilita o trabalho do desenvolvedor. Para tratamento de eventos, utiliza *delegates*.

ASP .NET. *Framework* voltado para o desenvolvimentos de aplicações *Web*.

Web Services. ⁵ *Framework* de serialização e deserialização que mapeia mensagens SOAP e documentos XML em métodos e objetos, facilitando o desenvolvimento de *Web Services*.

Web Forms. Páginas *Web* compiladas dinamicamente e sob demanda para código CIL, portáveis entre diferentes navegadores e dispositivos.

2.8 COMMON LANGUAGE RUNTIME

Nesta seção introduziremos o *Common Language Runtime*, implementação da Microsoft para a *Common Language Infrastructure* e principal pilar da Plataforma .NET. Nesta seção, daremos uma visão geral do mecanismo de execução da CLR, centrado na tecnologia de *compilação just-in-time*. Em seguida, detalharemos os mecanismos de implementação de alguns dos componentes importantes para a implementação de linguagens funcionais: *Generics* e *delegates*.

Com relação ao modelo de execução, a CLR executa apenas código nativo, de forma que qualquer método escrito em CIL deve ser traduzido para código nativo antes de ser executado. Há duas possibilidades para essa tradução. A primeira consiste em executar

⁵Serviços disponibilizados por meio da *World Wide Web*, de forma independente de linguagem de programação, com base em padrões e especificações estabelecidos, como XML [XML], SOAP [SOA], WSDL [CCMW01] e UDDI [UDD].

a compilação apenas no momento em que o componente for carregado em memória e o método for invocado. Esta técnica é o que denominamos de compilação *just-in-time* ou sob demanda e é o mecanismo padrão de execução adotado pela CLR. A segunda possibilidade é a pré-compilação, que ocorre quando uma imagem nativa do componente é gerada e instalada na máquina de destino, em uma *cache* global. Para tal, a CLR fornece um utilitário responsável por traduzir código CIL para código nativo e fazer a instalação na *cache*.

Ao contrário do que leva a crer a intuição inicial, a pré-compilação não necessariamente leva ao melhor desempenho. Muitas vezes rotinas freqüentemente utilizadas pela aplicação estarão em páginas diferentes da memória virtual, gerando um maior crescimento do chamado *working set* – quantidade de páginas em memória – e conseqüentemente do número de paginações. Já a compilação sob demanda é capaz de prover otimizações a partir da análise estatística do comportamento do programa, o que não é possível com a pré-compilação. Além disso, o tamanho do programa em código nativo é muito maior do que em código CIL. Dessa forma, para que seja possível concluir se o *overhead* gerado pela compilação *just-in-time* pode comprometer o desempenho de uma aplicação, faz-se necessária a comparação dos tempos de execução nas versões pré-compilada e compilada sob demanda.

2.8.1 Compilação Just-In-Time

Nesta seção introduziremos a implementação utilizada pela CLR para a técnica de compilação *just-in-time*.

Em geral, quando um programa é executado, a CLR inicialmente busca sua versão pré-compilada na *cache* de código nativo. Caso não seja encontrada, a compilação *just-in-time* será adotada. Além disso, mesmo que uma versão nativa seja encontrada, é possível que a CLR detecte que um dos módulos (pré-compilados) dos quais depende o componente foi modificado ou não foi encontrado na *cache*. Caso isso ocorra, mais uma vez a versão CIL é utilizada e compilada sob demanda.

No cenário da compilação sob demanda, a CLR compila o corpo de um método para código nativo na primeira vez em que o mesmo é chamado. Antes de entrar em detalhes de como decidir quando um método será compilado, é necessário explicar como um componente é representado em memória após seu carregamento.

Cada classe é representada por uma estrutura de dados (gerada a partir dos seus metadados) que contém, entre outras informações, tabelas com os endereços de todos os métodos por ela declarados. Inicialmente o corpo de cada método é representado por um trecho de código (“*stub*”) responsável por extrair o endereço do código do método através da *stream* de instruções e invocar a rotina de compilação para código nativo, implementada pelo compilador *just-in-time* (“*JIT*”). O endereço extraído é passado como parâmetro para a rotina em questão. Após a geração do código nativo correspondente, o endereço do corpo do método na tabela passa a referenciar um outro *stub* que contém apenas uma instrução (nativa) de desvio para o endereço do código gerado.

Em situações em que a CLR detecta que um determinado método é raramente utilizado o *stub* do método volta a referenciar a rotina do compilador JIT e a memória que

antes armazenava o código nativo é desalocada. Isso geralmente ocorre em situações onde existe uma demanda por memória não satisfeita através de outros meios.

O compilador JIT implementa várias implementações clássicas encontradas na tecnologia de compiladores, como “*inlining*” de métodos, “*constant folding*”, “*copy-propagation*”, remoção de código interno a laços, desenrolamento de laços, eliminação de subexpressão comum, alocação de argumentos em registradores, entre outras.

2.8.2 Delegates

Na Seção 2.2.5 vimos o que são *delegates* e como eles são implementados no contexto de linguagens de alto nível e CIL. Vimos também que eles oferecem um mecanismo de ligação entre um método e o objeto no qual o método deverá ser invocado. Nesta seção, daremos uma visão geral do mecanismo básico de implementação dos *delegates* na CLR.

Um *delegate* nada mais é do que uma subclasse de `MulticastDelegate`, definido para representar métodos com uma assinatura específica. Por decisão de projeto da CLI, *delegates* não podem herdar diretamente da classe `Delegate`. Além disso, a classe `MulticastDelegate` fornece suporte à invocação sequencial de múltiplos *delegates* de uma única vez, o que é útil em cenários onde um evento deve notificar vários objetos.

A Figura 2.14 mostra o relacionamento entre *delegates* e os objetos e métodos por eles referenciados. Nela podemos ver que múltiplos *delegates* são mantidos em uma estrutura de lista ligada cujo primeiro método a ser invocado é aquele que é referenciado pela “cabeça” da lista. Além da referência para o próximo elemento da lista, um *delegate* contém um ponteiro para o código do método a ser executado e uma referência para o objeto alvo.

A implementação da CLR para o método `Invoke` consiste em uma única instrução nativa que armazena a referência do método a ser chamado no registrador adequado e desvia para o corpo do mesmo. O endereço de retorno do método chamado não é o método `Invoke` mas sim o endereço do código que invocou o *delegate*.

A implementação acima e a Figura 2.14 são descritas por Box e Sells [BS03] e refere-se em especial à versão 1.1 da Plataforma .NET. A versão 2.0 trouxe otimizações que não estão documentadas oficialmente.

2.8.3 Generics

O suporte a Generics (Seção 2.2.6) na Plataforma .NET, ao contrário do que ocorre na linguagem JAVA, por exemplo, foi incluído diretamente no ambiente de execução. Nesta seção resumiremos a estratégia utilizada para implementar tal suporte. A descrição detalhada pode ser encontrada no trabalho de Kennedy e Syme [KS01].

A Figura 2.15⁶ mostra a representação em memória de objetos que são instâncias de tipos parametrizados. A estrutura conhecida como “*vtable*” é a chamada tabela de métodos virtuais, originalmente compartilhada por todas as instâncias de uma classe, e que agora é replicada para cada especialização utilizada. Essa tabela de métodos é a mesma estrutura citada em 2.8.1, e nos *slots* correspondentes às instanciações referencia

⁶Adaptada de [KS01].

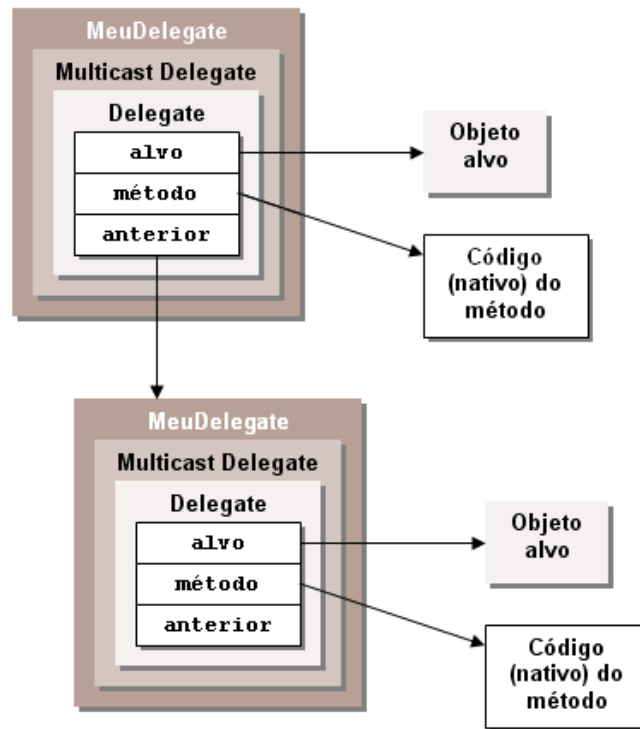


Figura 2.14. *Delegates* – Implementação

os códigos especializados, já compilados para código nativo.

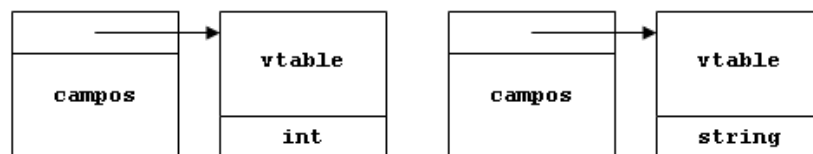


Figura 2.15. *Generics* – Implementação

A implementação de Generics na CLR faz uso do caráter dinâmico do ambiente de execução e caracteriza-se por:

- i) especialização de tipo “*just-in-time*”;
- ii) compartilhamento de código;
- iii) ausência de *boxing/unboxing*;
- iv) *laziness* ou especialização de tipo sob demanda.

O item i) diz respeito ao fato de que a instanciação de classes parametrizadas é executada dinamicamente, em tempo de tradução para código nativo, e não em tempo de compilação para CIL. Por instanciação nos referimos aqui à especialização de uma classe ou método genérico para uma nova “versão” não genérica, que utilize tipos bem definidos, como nas expressões `Stack<int>` e `Stack<string>` nos exemplos da Seção 2.2.6. Ao contrário do que ocorre em outras linguagens, em que análises em tempo de *linkedição* são feitas para identificar todas as possíveis instanciações para um tipo parametrizado, na CLR a montagem da representação especializada ocorre apenas no momento da instanciação.

O segundo item refere-se ao fato de que, sempre que possível, código e representação de metadados é compartilhada entre diferentes instanciações. Em geral, objetos, que são sempre representados por referências, compartilham o código e a representação instanciados, independente da classe à qual pertençam.

A especialização sob demanda ou “*lazy*” é útil na presença de recursão polimórfica, como em:

```
public class B<T>:A<B<B<T>>> {...}
```

Nessa situação, uma instanciação de B não pode ser construída simplesmente a partir da cópia de uma instanciação de A. Em geral, a tabela de métodos virtuais é preenchida à medida que instanciações para determinados tipos vão sendo criadas.

2.8.4 Gerenciamento de Memória

A CLR, no esforço de melhorar a produtividade do desenvolvedor e diminuir a quantidade de erros comuns de programação, implementa o conceito de gerenciamento automático de memória, mais conhecido como *garbage collection*, ou “coleta de lixo”, referindo-se à liberação automática de memória ocupada por objetos não mais utilizados.

A idéia de liberar o programador da obrigatoriedade de controlar explicitamente a alocação e desalocação de memória não é antiga. Bibliotecas para linguagens como C++, implementações de linguagens funcionais e outros ambientes gerenciados de execução como a JVM já implementavam algoritmos de *garbage collection*. Nesta seção será detalhada a forma pela qual a coleta automática de lixo é implementada na CLR.

2.8.4.1 Alocações e Algoritmo de Coleta Objetos (instâncias de tipos por referência) são alocados dinamicamente em uma região da memória gerenciada apenas pela CLR, conhecida como *heap* (Seção 2.8.4.2). Nessa região, uma alocação de memória costuma ser um processo extremamente eficiente – consiste apenas no incremento de um ponteiro. Isso ocorre porque a CLR procura manter a *heap* compactada, armazenando os objetos de forma contígua, de modo que não é necessária a busca por uma região onde haja memória suficiente para alocação.

A coleta de memória não mais utilizada, por sua vez, é implementada por meio de um algoritmo do tipo *mark-and-compact* [Wie04], que consiste em marcar todos os objetos de uma aplicação que são direta ou indiretamente atingíveis a partir de objetos raiz (ex.: objetos na pilha, campos estáticos de classes). Em seguida, os objetos que não

foram marcados são removidos e a memória atribuída àqueles que não serão coletados é compactada, fazendo com que os objetos sobreviventes ocupem posições contíguas, otimizando dessa forma alocações futuras.

É importante ressaltar que um objeto torna-se elegível à coleta (ie.: inatingíveis) quando não há mais referências ao mesmo por parte de objetos não elegíveis. Isso pode ocorrer em duas situações:

- quando a referência para o objeto sai de escopo. Ex.: referência em forma de variável local de um método.
- quando todas as referências para o objeto são modificadas para o valor `null` – um padrão de bits que identifica que uma referência não aponta para nenhum objeto.

Uma segunda *heap* é voltada especificamente para o armazenamento de objetos cujo tamanho excede 20 KB. Devido ao alto custo de realocação de objetos tão grandes, essa região não é compactada.

2.8.4.2 Gerações A *heap* gerenciada pela CLR é dividida no que denominamos de *gerações*, classificadas de acordo com a idade dos objetos nela armazenados. Dessa forma, um objeto que sobrevive a uma coleta é promovido à geração seguinte. Na implementação atual da CLR, existem 3 (três) gerações:

- **Geração 0.** Contém os objetos mais recentemente criados. É frequentemente coletada (a cada 0.2 a 2MB de alocação), para garantir que objetos recentes, que são aqueles com maior probabilidade de ocorrerem na *heap*, são coletados. Coletas nessa geração costumam ser mais rápidas, pois a mesma em geral ocupa a *cache* do processador. Objetos não coletados na geração 0 são realocados para a geração 1;
- **Geração 1.** Contém os objetos não coletados na geração anterior. Sofre coletas com menos frequência. Objetos não coletados nesta geração são movidos para a geração 2;
- **Geração 2.** Contém os objetos mais antigos, não coletados na geração 1. As coletas são ainda menos frequentes nesta geração, uma vez que o *garbage collector*, assume que objetos de vida longa são menos comuns.

A estratégia de gerações, denominada *coleta geracional*, implementa o que chamamos de coletas parciais, em oposição a coletas completas, em que a *heap* inteira é coletada. Por meio dessa técnica, apenas uma parte da *heap* é coletada, em caso de escassez de memória. Obedecendo ao princípio estatístico que objetos mais recentemente alocados terão um tempo de vida menor, as coletas ocorrem primeiramente na geração 0. Caso a necessidade por memória ainda não seja satisfeita, uma coleta na geração 1 é feita, e assim sucessivamente. Em geral, para que um objeto pertencente à geração 2 possa ser coletado, é necessária uma coleta completa. Isso faz com que coletas nessa geração sejam muito mais longas do que nas gerações anteriores.

Para que a estratégia geracional ofereça melhor desempenho, é desejável que os objetos tenham em sua maioria tempos de vida curtos, para que sejam coletados na geração 0. O recomendável é que a frequência de coletas varie de cerca de 1 (uma) ordem de magnitude de uma geração para a posterior. Entretanto, conforme veremos no Capítulo 5, esse comportamento ideal torna-se difícil, senão impossível de ser atingido para programas funcionais na implementação atual da CLR, fazendo com que o gerenciamento de memória torne-se a área mais crítica e de mais difícil otimização.

2.9 OUTRAS IMPLEMENTAÇÕES DA CLI

Apesar do nosso trabalho se concentrar em utilizar a implementação da Microsoft apresentada na seção anterior, existem outras implementações para a CLI.

Nesta seção, descrevemos brevemente as outras implementações disponíveis.

2.9.1 Shared Source CLI (Rotor)

A *Shared Source Common Language Infrastructure* [SNS03], também conhecida como “Rotor”, é uma implementação de código aberto da CLI, de propriedade da Microsoft.

Funciona sobre os sistemas operacionais Windows, FreeBSD e Mac OS. Seu principal objetivo é servir como ferramenta de estudo para aqueles que se interessam em simplesmente compreender a tecnologia utilizada no gerenciamento de memória, compilação *just-in-time*, etc., e como ambiente de pesquisa acerca de extensões e otimizações. Além disso, possui o papel de validar a portabilidade da especificação da CLI, bem como de servir como guia para quem desejar implementá-la.

Em relação à CLI, a SSCLI pode ser vista como um superconjunto, uma vez que inclui compiladores das linguagens C# e JScript, montadores, depuradores, entre outras ferramentas. Adicionalmente, inclui um componente responsável por prover portabilidade entre sistemas operacionais: a *Platform Adaptation Layer* (PAL), ou camada de adaptação de plataforma. Essa camada disponibiliza um subconjunto da API do Windows, porém internamente reimplementa tais rotinas de acordo com as especificidades do sistema operacional.

2.9.2 Mono

O projeto Mono [Mon] consiste em uma implementação de código aberto da CLI para as plataformas Windows, Linux, Unix, Solares e Mac OS. Inclui suporte à linguagem C#, compilação *just-in-time*, depuradores, ASP .NET, *Web Services*, acesso a banco de dados, criptografia e segurança em geral, diversas bibliotecas adicionais, além de várias tecnologias para desenvolvimento de interfaces gráficas, tais como Gtk#, Windows Forms e Cocoa Sharp.

Em comparação à Plataforma .NET, Mono inclui praticamente os mesmos serviços, com os mesmos nomes. Entretanto, o suporte a versões mais recentes de alguns componentes, como ASP .NET 2.0, ainda está em fase de desenvolvimento.

2.9.3 DotGNU Portable .NET

Portable .NET [Dot] é uma implementação da CLI, também de código aberto, voltada originalmente para a plataforma Linux e posteriormente compatível com outras plataformas como Windows, Mac OS, FreeBSD, entre outras. O principal objetivo do projeto é a compatibilidade com a especificação e com a implementação Microsoft, permitindo que qualquer *software* escrito para a Plataforma .NET execute normalmente em seu ambiente de execução e, conseqüentemente, em vários sistemas operacionais. Inclui suporte à biblioteca básica de classes da CLI, XML e Windows Forms.

Um detalhe que difere o Portable .NET das outras implementações é o projeto da sua máquina virtual, que adota uma abordagem interpretativa. Tal abordagem consiste no uso de uma linguagem *assembly* ainda mais simples que a CIL, para a qual o programa CIL é traduzido, e na execução do código resultante por um interpretador de alto desempenho. Existem planos de incorporar um compilador *just-in-time* futuramente.

2.10 CONSIDERAÇÕES FINAIS

Analizamos a Plataforma .NET e o seu “subconjunto” padronizado – a *Common Language Infrastructure* (CLI). Descrevemos características relevantes tanto da CLI como de sua implementação mais utilizada – o *Common Language Runtime* (CLR) – sob a óptica do ambiente de execução. Destacamos o caráter multilinguagem desta plataforma através de seu vasto sistema de tipos e de sua variedade de instruções. Os conceitos aqui apresentados serão úteis ao entendimento das questões relacionados à implementação de linguagens funcionais na Plataforma .NET.

A principal contribuição de uma plataforma de desenvolvimento multilinguagem está na possibilidade de combinar em uma mesma aplicação diferentes linguagens, de modo a fazer uso do que há de melhor em cada uma delas. Como exemplo, podemos ter dentro de um mesmo *software* um módulo da camada de apresentação escrito em uma linguagem orientada a objeto, enquanto outro, responsável por algoritmos matemáticos, pode ser escrito de forma concisa em uma linguagem funcional.

Linguagens de vários paradigmas já se encontram atualmente implementadas para .NET, a saber, Ada, Cobol, Eiffel, Fortran, Lisp, Mercury, Perl, Prolog, Phyton, Scheme, Smalltalk, entre outras.⁷

No Capítulo 3, analisaremos as possíveis abordagens para a implementações de linguagens funcionais em máquinas virtuais orientadas a objetos, dando maior atenção ao ambiente .NET.

⁷Uma lista completa de linguagens implementadas para .NET pode ser encontrada em [Rit].

CAPÍTULO 3

INTEGRAÇÃO DE LINGUAGENS FUNCIONAIS AO AMBIENTE .NET

O paradigma de programação funcional é caracterizado pelo seu alto grau de abstração em relação aos paradigmas mais comumente utilizados (imperativo e orientado a objetos). A presença de elementos de alto nível como aplicações parciais, funções de alta ordem e, em algumas linguagens, avaliação sob demanda ou não estrita, torna desafiadora a compilação eficiente de linguagens funcionais.

A dificuldade de implementação dessas linguagens torna-se ainda maior quando a plataforma de destino, por motivos de segurança, impõe restrições a práticas como acesso direto a memória, uso de ponteiros e de construções que não sejam fortemente tipadas, por vezes impedindo a execução de código que delas faz uso. Tais restrições não estão presentes na compilação direta para código nativo, em que existe total controle sobre o código de máquina gerado.

Neste capítulo, será dada uma visão geral sobre linguagens funcionais (Seção 3.1) e abordagens existentes para integrá-las a plataformas como .NET e a *JAVA Virtual Machine* (Seção 3.2). Entretanto, concentraremos nossa discussão no ambiente .NET, por ser este o foco deste trabalho, apesar de em alguns momentos fazermos referência a implementações para a JVM para ilustrar alguns conceitos. Analisaremos possíveis escolhas para a natureza do código final gerado (Seção 3.3), bem como as técnicas de implementação de elementos pervasivos em linguagens funcionais (seções 3.4, 3.7 e 3.6). As seções 3.5 e 3.8 tratam, respectivamente, dos mecanismos de invocação de função e controle da pilha de chamadas mais utilizados. Por fim, na Seção 3.9, citaremos trabalhos já realizados no campo de integração de linguagens funcionais à CLI.

3.1 INTRODUÇÃO A LINGUAGENS FUNCIONAIS – PRINCIPAIS CONCEITOS

Antes de nos atermos às técnicas de compilação de linguagens funcionais para plataformas como a CLI, introduziremos aqui alguns elementos característicos do paradigma funcional, facilitando o entendimento do conteúdo das próximas seções. Esses elementos são responsáveis não apenas por aumentar o poder de expressão mas também por tornar sua compilação não trivial em comparação a outros paradigmas, como veremos ao longo deste capítulo.

3.1.1 Ordem de Avaliação

Em linguagens imperativas e orientadas a objeto, ao invocarmos uma função, procedimento ou método que declara receber parâmetros, primeiramente avaliamos os argumentos ou parâmetros reais e só então geramos a instrução de chamada. Em outras palavras,

a função invocada assume que seus argumentos já estão avaliados no momento de sua ativação. Tal mecanismo de avaliação é o mais comum e denomina-se *applicative-order evaluation* ou *avaliação estrita*¹ [Wat90].

Alternativamente, poderíamos ter situações em que não desejamos que os argumentos sejam avaliados a menos que essa avaliação seja realmente necessária à execução da função invocada. Da mesma forma, poderíamos ter a necessidade de definir estruturas de dados infinitas que fossem avaliadas apenas sob demanda. O mecanismo de avaliação conhecido por “*normal-order evaluation*” [Wat90] ou avaliação sob demanda torna isso possível, uma vez que os argumentos recebidos por uma função não são avaliados antes da chamada, mas sim no corpo da função, somente quando seus valores forem necessários. Uma evolução desse modelo, conhecida por “*lazy evaluation*”, evita que avaliações ocorram repetidas vezes, fazendo com que o argumento seja avaliado apenas quando referenciado pela primeira vez.

Com respeito aos mecanismos acima definidos, dizemos que uma função é *estrita* com relação a um argumento quando este precisa ser avaliado antes da função ser invocada. Analogamente, dizemos que uma função é *não estrita* quando a ativação da mesma não depende da avaliação desse argumento. Em resumo, uma função não estrita pode ter seus argumentos avaliados sob demanda (*normal-order* ou *lazy evaluation*), enquanto uma função estrita suporta apenas avaliação do tipo *applicative-order*. Classificamos portanto as linguagens em dois grupos principais:

- *linguagens estritas*, onde as funções são por padrão **estritas** para todos os argumentos. Exemplos: linguagens imperativas, orientadas a objeto e funcionais como SML [MTHM97], Scheme [Sch], Caml [Cam] e F#[F#];
- *linguagens não estritas*, onde as funções são por padrão **não estritas** para todos os argumentos. Exemplos: Haskell [Has], Mondrian [MPvY01] e Lazy ML [AJ89].

3.1.2 Funções de Alta Ordem

Em linguagens funcionais, as funções são elementos de primeira classe, ou seja, são valores como quaisquer outros, podendo ser passadas como argumento, retornadas por outras funções ou armazenadas em estruturas de dados. Em geral, dizemos que funções que possuem pelo menos uma função como parâmetro são classificadas como *de alta ordem*, em oposição a *funções de primeira ordem*, em que nenhum parâmetro é uma função [Wat90].

Dessa forma, é necessário que funções passadas como valor sejam representadas como objetos, e que exista algum mecanismo capaz de aplicar tais funções aos argumentos a elas passados. Trataremos desta última questão na Seção 3.5, quando nos concentrarmos nos detalhes relacionados à implementação propriamente dita de linguagens funcionais.

¹Também designado pelos termos “*eager evaluation*” e “*call by value*”.

3.1.3 Aplicações Parciais

A lista de parâmetros esperados por uma função pode ser compreendida como uma *tupla*, em que **todos** os parâmetros devem ser passados para que a avaliação seja possível. É o caso das listas de parâmetros encontradas na maioria das linguagens imperativas, excluindo-se obviamente extensões como listas de tamanho variável. Definimos esse tipo de função como nos exemplos abaixo², escritos em Haskell e C#, respectivamente:

```
power :: (Int, Double) -> Double
power (n, b)
  | n == 0 = 1
  | otherwise = b * power (n-1, b)

double power(int n, double b) {
  if n == 0 return 1;
  else return b * power (n-1, b);
}
```

Por outro lado, poderíamos ter definido a função `power` em Haskell como na forma abaixo:

```
power :: Int -> Double -> Double
power n b
  | n == 0 = 1
  | otherwise = b * power (n-1) b
```

Aparentemente, a diferença é apenas sintática – a ausência dos parênteses ao redor dos parâmetros. Entretanto isso não é verdade: a segunda versão da definição Haskell de `power` permite interpretá-la com uma função que, dado um parâmetro do tipo `Int`, retorna outra função que espera receber um parâmetro do tipo `Double` para retornar um resultado, também do tipo `Double`. Denominamos esta segunda forma de versão *currificada* de `power`.

Denominamos de *aplicação parcial* a técnica de aplicação de uma função currificada a um número de argumentos menor que o número máximo de parâmetros suportados pela função [Wat90]. No exemplo acima, `power 2` seria um caso de aplicação parcial.

3.2 INTEGRAÇÃO AO AMBIENTE .NET

A partir de agora, analisaremos as técnicas de integração de linguagens funcionais a plataformas de código gerenciado como o ambiente .NET. Descreveremos desde as mais simples, que permitem apenas interoperar os dois mundos, até as diferentes estratégias de compilação de construções funcionais nessa plataforma.

Começaremos a discussão acerca de como integrar linguagens funcionais a plataformas orientadas a objeto.

²Os exemplos são adaptados da obra de David Watt, “*Programming Language Concepts and Paradigms*” [Wat90].

3.2.1 Bridge

Em algumas situações, temos a necessidade de desenvolver uma solução a partir de componentes escritos em diferentes linguagens. Especialmente em situações em que tais componentes precisam trocar algum tipo de mensagem entre si, é comum utilizarmos elementos responsáveis por implementar essa comunicação – as chamadas *bridges*, ou “*pontes*”. No contexto de linguagens funcionais, o cenário mais comum é um programa compilado para código nativo acessar um programa escrito em outra linguagem também compilada para código nativo ou mesmo compilada para código gerenciado pela CLR ou JVM. Neste cenário, a *bridge* é responsável por intermediar as chamadas, traduzindo os tipos dos argumentos e retornos, em um processo conhecido por *marshalling*. Exemplos de *bridges* que comunicam a linguagem Haskell a plataformas orientadas a objeto são o Hugs .NET [Hugb], H/Direct [FLMJ98] e Lambada [MF00].

As *bridges* servem bem ao propósito de ocasionalmente acessar rotinas escritas em outras linguagens. Porém, elas apresentam limitações quando desejamos um maior nível de integração, que permita acesso aos recursos mais avançados do outro ambiente. Além disso, algumas vezes os requisitos de desempenho não permitem o *overhead* associado à tradução entre tipos.

Finalmente, a plataforma com a qual se interopera, no nosso caso específico, a CLR, é forçada a fazer chamadas a um código “externo”, que não é gerenciado por ela. Se levarmos em conta que as implementações de uma linguagem como Haskell em geral incluem um ambiente de execução com gerenciamento próprio de memória, ocorrerá a situação em que teremos duas “máquinas virtuais”, com dois coletores de lixo, por exemplo. Nessas situações, é clara a necessidade de uma solução mais sofisticada e eficiente.

3.2.2 Compilação

A alternativa mais natural à integração entre uma dada linguagem fonte – Haskell, por exemplo – e um ambiente de código gerenciado é a *compilação* para outra linguagem já suportada pelo ambiente. Tomando como exemplo a Plataforma .NET, o caminho mais direto é gerar código CIL – uma vez que esta é a linguagem *assembly* padrão da CLI, devendo ser suportada por qualquer implementação da mesma. Uma linguagem de mais alto nível para o qual exista um compilador para CIL também pode ser utilizada como linguagem de destino, como veremos na Seção 3.3. O importante aqui é que, para um programa escrito na linguagem fonte, seria gerado o mesmo tipo de código que é gerado para qualquer outro programa .NET, independente da linguagem no qual este último foi escrito.

As vantagens da compilação podem ser resumidas nos seguintes aspectos:

- um programa .NET que deseje interoperar com um programa escrito em Haskell, segundo nosso exemplo, não mais precisaria acessar código não gerenciado;
- como consequência do item anterior, não haveria o impacto de desempenho associado a trocas de contexto entre os dois ambientes – gerenciado e não gerenciado;
- o *overhead* associado a traduções entre tipos (*marshalling*) seria reduzido ou nulo;

- a memória – agora compartilhada – seria gerenciada de forma mais simples e eficiente, uma vez que apenas um *garbage collector* seria utilizado, diferentemente do que ocorreria se tivéssemos código gerenciado executando em paralelo com código não gerenciado. Nesta última situação, haveria dois *garbage collectors* atuando, aumentando a complexidade de gerenciamento de memória e possivelmente degradando o desempenho da aplicação.
- as possibilidades de interoperabilidade seriam maiores, como por exemplo, a melhor integração com recursos avançados como ASP .NET.

A maioria dos trabalhos voltados para a integração entre linguagem funcionais e ambientes gerenciados como .NET ou JVM utiliza a compilação como estratégia. Daqui em diante nos concentraremos nesta abordagem, analisando as diferentes técnicas para essa compilação.

3.3 LINGUAGEM ALVO

A primeira decisão de projeto a ser tomada na implementação de qualquer linguagem para uma plataforma de execução gerenciada é a linguagem alvo. Nesta seção, usamos o termo “linguagem alvo” para designar a linguagem para a qual a linguagem fonte é compilada. A escolha da linguagem alvo terá impacto em diferentes aspectos, tais como:

- custo de implementação;
- velocidade de geração do código final, diretamente relacionada à quantidade de passos que devem ser executados pelo compilador até a obtenção do código nativo;
- flexibilidade de implementação e controle sobre a eficiência do código final.

O último aspecto será analisado à luz de duas possibilidades estudadas nesta seção.

3.3.1 Linguagem de Alto Nível

A possibilidade mais simples a ser considerada é adotar como linguagem-alvo uma linguagem que já seja suportada por um compilador, como C# [ECM05], no caso do ambiente .NET, ou JAVA, no caso da JVM. Dessa forma, a responsabilidade pela geração do código final – CIL, no nosso caso – passa a ser do compilador dessa linguagem, tornando a tarefa de gerar código final menos suscetível a erros e fazendo uso de otimizações já oferecidas pelo compilador. Essa abordagem é utilizada por Mondrian [MPvY01].

Entretanto, apesar da simplicidade, a geração de código em uma linguagem de alto nível impossibilita o uso de algumas instruções de baixo nível que são encontradas em CIL e que podem ajudar a melhorar o desempenho do código gerado. Como exemplo, podemos citar instruções para gerar *tail-calls*, que implementam um mecanismo para conter o crescimento da pilha de chamadas, mas não são geradas pelo compilador C#. Com isso o projetista deve limitar-se às construções presentes na linguagem alvo, que em geral é apenas um subconjunto reduzido das construções suportadas pelo ambiente de execução. Além disso, essa abordagem cria dependência em relação às otimizações que possam ou não ser implementadas pelo compilador da linguagem alvo.

3.3.2 Código Final

Devido às desvantagens da abordagem anterior, a geração de código diretamente no código final da plataforma – CIL, no nosso caso – é sem dúvida a solução mais flexível e eficiente. Além disso, a velocidade de compilação deve ser favorecida uma vez que passos intermediários não são adicionados.

Com relação ao maior custo de implementação e manutenção, este pode ser amortizado com o uso de ferramentas que verifiquem a corretude do código gerado, ou mesmo de APIs que facilitem a sua geração. Tal suporte é facilmente encontrado na Plataforma .NET.

3.4 REPRESENTAÇÃO DE CLOSURES

Closures são objetos alocados dinamicamente que encapsulam um código a ser executado e um ambiente que pode ser acessado pelo código. Um exemplo é mostrado abaixo:

```
f :: Int -> u -> (Int -> Int)
f x y = let g w = x + w in g
```

No exemplo, *f* é uma função que recebe 2 argumentos (*x* e *y*, de tipos *Int* e *u*, respectivamente) e retorna um elemento de tipo (*Int*->*Int*). *g* é uma *closure* que encapsula um código que espera por 1 argumento (*w*) e acessa um argumento recebido por *f* (*x*). Aqui, *x* é considerada uma *variável livre em g*, pois não é declarada no escopo de *g*. *f* também podem ser considerada uma *closure*, porém sem variáveis livres.

Conforme descrito na Seção 3.1.1, no mecanismo de avaliação *lazy* os argumentos recebidos por uma função serão avaliados no corpo da mesma, somente quando seus valores forem necessários e no máximo uma vez. Portanto, cada argumento deve ser uma *closure* encapsulando o código responsável por avaliá-la. No exemplo, *x* e *y* são *closures* e *f* e *g* podem ser passados como argumentos para quaisquer outras funções.

Existem várias técnicas para representar *closures* em plataformas orientadas a objeto. Aqui introduziremos alguns mecanismos que podem ser utilizadas na Plataforma .NET.

3.4.1 Uma Classe por Closure

Uma estratégia comumente utilizada para representar *closures* em plataformas orientadas a objeto é a geração de uma classe por *closure*. Essa estratégia assume a existência de uma interface ou classe abstrata com um ou mais métodos responsáveis por aplicar o código da *closure* aos seus argumentos. A partir daí, cada *closure* é compilada para uma subclasse. O código da *closure* é inserido no método de aplicação adequado e as variáveis livres são compiladas para campos, por exemplo. Um exemplo desta técnica é mostrado abaixo. Para uma função definida como:

```
foo :: ...
foo <arguments> = /*código de foo*/
```

e uma classe abstrata pré-definida como:

```
abstract class Closure {  
    ...  
    public abstract object Invoke(...);  
    ...  
}
```

a seguinte classe é gerada:

```
class foo: Closure {  
    ...  
    public object Invoke(...){  
        /*código compilado de foo*/  
    }  
    ...  
}
```

Esta técnica pode ser implementada por vários meios: `Invoke` pode receber um *array* de objetos que representam os parâmetros da função ou pode não receber argumento algum e obter os mesmos em um outro local (uma pilha, por exemplo). Uma outra abordagem é usar várias versões de `Invoke`, cada uma recebendo um número diferente de argumentos.

A principal vantagem desta abordagem é que variáveis livres podem ser diretamente acessadas como campos. Entretanto, a desvantagem é o grande número de classes que seriam criadas para um programa de tamanho médio escrito em uma linguagem não estrita, em que há uma grande quantidade de *closures*. Na Plataforma .NET, cada classe possui metadados que devem ser carregados, mantidos em memória e descartados por um processo coletor quando possível. Além disso, a CLR executa rotinas de verificação que podem aumentar o *overhead* geral de execução.

Esta estratégia é adotada por Mondrian (não estrita) e F# (estrita). F#, por exemplo, evita a geração de muitas *closures* através de técnicas de *inlining* de código. Além disso, F# é uma linguagem estrita e nessas linguagens *closures* aparecem com muito menos frequência, sendo utilizadas basicamente em funções de alta ordem, para representar os parâmetros de tipo funcional. Algumas implementações de linguagens funcionais para a JVM também fazem uso desta técnica [CiLH01, Bot98, Tul96]. No caso da JVM, em particular, a degradação de desempenho é ainda mais crítica, uma vez que cada classe é mapeada em um arquivo em disco.

3.4.2 Classes Pré-definidas

Nesta abordagem, um conjunto de classes pré-definidas é adicionado ao ambiente de execução do compilador e cada *closure* é representada como uma instância de uma dessas classes. Argumentos podem ser armazenados em *arrays* ou locais separados, podendo ainda haver classes especializadas para *closures* que recebam $0, 1, \dots, n$ argumentos ou que possuam $0, 1, \dots, n$ variáveis livres. O importante é que um grande conjunto de objetos compartilham a mesma classe e seus códigos podem ser compilados para métodos

estáticos em uma única classe separada, por exemplo. Os custos associados a esta abordagem provém principalmente de indireções adicionais para acessar os métodos estáticos. Entretanto, não podemos garantir que essas indireções causariam impactos significativos no desempenho.

No exemplo mostrado anteriormente, as classes pré-definidas poderiam ser subclasses de `Closure`.

Existem muitas variações desta técnica. Elas diferem basicamente na forma pela qual o código da *closure* é acessado. Aqui consideramos três alternativas:

Ponteiros para métodos. A CLI permite a invocação de métodos a partir de ponteiros.

Dessa forma, uma *closure* poderia simplesmente conter um ponteiro para o seu código. Segundo a especificação da CLI, a chamada de um método a partir de um ponteiro, desde que o mesmo esteja corretamente tipado com a assinatura do método, deve constituir código verificável. Entretanto, experimentos realizados confirmaram que na CLR isso não é verdade. Conforme citado no Capítulo 1, um código não verificável não pode ser considerado seguro através de análises de fluxo de execução ou provas formais. De acordo com [BS03], “a habilidade de carregar código não verificável é uma permissão que deve ser dada explicitamente para o código por meio da política de segurança. A política padrão que é instalada com a CLR garante esta permissão apenas para código instalado no sistema local de arquivos”. Portanto, código transferido da Internet que não é verificável não pode ser facilmente executado. Além disso, rotinas a serem executadas em banco de dados (ex.: procedimentos armazenados e gatilhos) também precisam ser verificáveis.

O uso de código não verificável foi originalmente adotado por F#, mas logo foi abandonado.

Delegates. *Delegates*, um recurso introduzido pela Plataforma .NET, são a versão segura e orientada a objetos de ponteiros para métodos. São objetos que encapsulam um ponteiro para um método a ser invocado, bem como uma referência para o objeto alvo no qual o método deverá ser invocado. Não encontramos até o momento nenhuma implementação que faça uso de *delegates* para a representação de *closures*. A implementação desses elementos possuía problemas de desempenho na primeira versão da plataforma, uma possível explicação para o fato do uso dos mesmos não se ter difundido. Entretanto, executamos alguns experimentos que mostraram que o desempenho dos *delegates* foi significativamente melhorado na versão 2.0 da CLR. Por esse motivo, optamos pelo uso de *delegates* para representar *closures* no compilador desenvolvido neste trabalho.

Código “indexado”. Esta abordagem é adotada pelo *Bigloo* [BSS04], um compilador para a linguagem funcional estrita Scheme. É também utilizada por algumas implementações que possuem como alvo a JVM [SS02, Wak99]. O uso da técnica de código indexado consiste em armazenar um índice inteiro em cada *closure* e, ao invocá-la, um teste no valor do índice é realizado e a função correspondente é chamada. Cada função é mapeada em um índice diferente. Testes realizados pe-

los implementadores do Bigloo demonstraram desempenho superior ao obtido com *delegates*, porém tais testes foram baseados nas primeiras versões da CLR.

3.5 APLICAÇÃO DE FUNÇÕES

A implementação de aplicação de funções em linguagens funcionais não é tão trivial quanto em linguagens imperativas e orientadas a objetos. Em linguagens funcionais, funções constituem-se em valores de primeira classe, ou seja, podem ser passadas como parâmetros para outras funções, retornadas como resultado ou armazenadas em estruturas de dados. Isso significa que uma função não conhecida estaticamente (ie.: em tempo de compilação) pode ser aplicada a qualquer número de argumentos, inclusive a um número diferente de sua aridade ³. Existem dois modelos de execução para aplicação de funções: *push/enter* e *eval/apply*.

3.5.1 Modelo Push/Enter

No modelo *push/enter*, o código da função chamada é responsável por aplicá-la ao número correto de argumentos. Quando uma função de aridade desconhecida em tempo de compilação é invocada, ou mesmo quando uma função conhecida é aplicada a um número insuficiente de argumentos, os argumentos em geral são armazenados em uma pilha (diferente da pilha da CLR) e uma versão sem parâmetros do código compilado da função – o *slow entry point* – é invocado. Entretanto, se a função for conhecida estaticamente e for aplicada a um número de argumentos igual ou superior à sua aridade, o mecanismo padrão de passagem de parâmetros pode ser utilizado, invocando a versão com parâmetros – o *fast entry point*. O modelo *push/enter* é utilizado (parcialmente) por Mondrian.

3.5.2 Modelo Eval/Apply

No modelo *eval/apply*, o código que invoca a função é responsável por inspecionar sua aridade e aplicá-la ao número correto de argumentos. Se uma função f for invocada com menos argumentos do que o necessário, é retornada uma aplicação parcial – uma estrutura de dados que armazena a referência para a função e os argumentos que foram recebidos. Por outro lado, se for aplicada a um número de argumentos maior ou igual que o necessário, ela é invocada normalmente, passando os argumentos na pilha da CLR e assumindo-se que a função retornada por f será aplicada aos argumentos excedentes, caso eles existam. Dessa forma, o mecanismo padrão de passagem de parâmetros – que utiliza a pilha da CLR – pode ser utilizado mesmo para funções que não são conhecidas estaticamente. Este modelo é utilizado, por exemplo, em F# e no Bigloo.

Marlow e Peyton Jones [MJ04] fornecem uma definição formal para os dois mecanismos e comparam suas implementações para a geração de código C, utilizando o *Glasgow Haskell Compiler* (GHC) [GHC]. A conclusão final é que ambos os modelos apresentam

³Definimos *aridade* como sendo a quantidade de argumentos que uma função exige para realizar uma computação.

desempenhos similares naquele contexto. Não podemos portanto indicar qual é melhor na Plataforma .NET. Para responder a essa questão, ambos os modelos precisam ser implementados e comparados. No modelo eval/apply, podem ser instanciadas aplicações parciais desnecessárias, além do *overhead* gerado por testes de aridade, que em geral podem envolver conversões (“casts”) para funções que esperam determinado número de parâmetros. Por outro lado, no modelo push/enter, uma pilha separada precisa ser construída e gerenciada, por razões técnicas que não permitem uma manipulação mais direta da pilha da CLR.

No compilador Haskell desenvolvido, optamos pelo uso do modelo push/enter, principalmente por questões de facilidade de implementação descritas no Capítulo 4.

3.6 POLIMORFISMO PARAMÉTRICO

Uma característica comum em linguagens funcionais é a habilidade de definir funções que podem receber argumentos de quaisquer tipos.

Em plataformas fortemente tipadas, como .NET e JVM, uma abordagem tradicional para implementar essa funcionalidade é o uso de uma superclasse comum para representar o tipo dos argumentos polimórficos. Em .NET, esse tipo pode ser `System.Object`, ou mesmo um tipo mais específico definido pelo ambiente de execução do compilador (ex.: `Closure`). O uso de uma superclasse comum é adotado por Mondrian.

Uma outra possibilidade é o uso de *Generics*, um novo recurso incorporado na versão 2.0 da CLI que objetiva corrigir algumas limitações da técnica acima, evitando erros de execução comuns em programas orientados a objeto. Generics é utilizada atualmente em F#. Para linguagens que fazem uso de avaliação não estrita, no entanto, os argumentos polimórficos são sempre tipados como *closures*. Portanto, nesse caso particular, o uso de Generics não fornece ganhos de desempenho significativos, não sendo utilizado neste trabalho para representação de polimorfismo paramétrico. Entretanto, conforme veremos no próximo capítulo, Generics foi utilizado para outras finalidades na solução desenvolvida.

3.7 UNIÕES DISCRIMINADAS

Em linguagens funcionais, normalmente definimos tipos de dados conforme mostrado abaixo:

```
data List t = Nil | Cons t (List t)
```

Denominamos tais definições de uniões discriminadas. No exemplo, `Nil` e `Cons` são chamados *construtores*.

Em .NET, não existe um conceito equivalente a uniões discriminadas. Existem as chamadas *enumerações* (Seção 2.2.1), semelhantes aos tipos enumerados da linguagem C. Porém, tais tipos enumerados não recebem argumentos, sendo ao fim traduzidos para constantes inteiras. Por outro lado, temos o conceito de classes, que pode ser utilizado para representar registros e relacionamentos de subtipo. Portanto, a opção mais intuitiva consiste na geração de uma classes abstrata para representar a união discriminada - no

exemplo acima, uma classe denominada `List` - e uma subclasse por construtor. Precisamos apenas encontrar uma forma eficiente de inspecionar qual construtor está sendo utilizado, uma necessidade comum à implementação de casamento de padrão sobre uniões discriminadas. Podemos assim utilizar uma instrução específica da CIL para descobrir dinamicamente a classe de um objeto. Um exemplo é mostrado abaixo, na sintaxe da linguagem C#:

```
List l;
...
if(l is Nil){...}
else if(l is Cons){...}
```

No exemplo, o operador `is` é compilado para uma instrução da CIL. Entretanto, uma abordagem ainda mais simples e eficiente é anotar cada objeto com um valor inteiro que identifica o seu construtor. Desta forma, podemos inspecionar dinamicamente este valor fazendo uso de instruções do tipo `switch`, que em geral utilizam na sua implementação tabelas de consulta (“*lookup tables*”) como alternativa ao teste sequencial mostrado acima. O código em C# é mostrado a seguir:

```
switch(l.tag){
  case 1: /* 1 é a tag correspondente a Nil */
    ...
  case 2: /* 2 é a tag correspondente a Cons */
    ...
}
```

Escolhemos esta última abordagem para a implementação de uniões discriminadas e casamento de padrão no compilador Haskell.NET.

3.8 CONTROLE DA PILHA DE CHAMADAS

No paradigma de programação funcional, o uso de funções e de recursão é muito freqüente, de forma que se não houver cuidado, estouros de pilhas de chamada podem ocorrer facilmente. Além disso, devido ao *overhead* gerado por chamadas a funções e procedimentos, é necessário que essas chamadas sejam implementadas da forma mais eficiente possível, sendo substituídas, sempre que possível, por instruções de desvio. Nesta seção, serão analisadas duas técnicas que objetivam resolver esses problemas.

3.8.1 Trampolim

A técnica conhecida como “trampolim” ou “*tiny interpreter*” apareceu pela primeira vez na *Spineless Tagless G-Machine* [Jon92] e consiste na implementação de um programa por meio de um laço da forma:

```
while(true) f = (*f)();
```

O código acima indica que a chamada a uma função é traduzida pelo retorno do seu endereço, e não pela invocação propriamente dita. Dessa forma, o tamanho da pilha é controlado. Na CLI, uma variação seria algo como:

```
while(f != null) f = f.Invoke();
```

O método do trampolim, por ser genérico com relação à função chamada, possui uma desvantagem: dificulta o mecanismo de passagem de parâmetros em uma plataforma como a CLR, em que a pilha de argumentos não pode ser manipulada diretamente. Por esta e outras razões, esta técnica não tem sido muito utilizada.

3.8.2 Tail-calls

Tail-call é um mecanismo de chamada de função que visa a eliminar o consumo de *stack frames* da seguinte forma: sempre que se garante que o fluxo de execução não retornará à função atual após a chamada, o *frame* atual é descartado, sendo imediatamente substituído pelo novo *frame*. Assim, evita-se o crescimento exagerado da pilha de chamadas.

Em compiladores que geram código nativo, é comum a implementação de *tail-calls* por meio de instruções de desvio para o código de destino, normalmente referenciado por um ponteiro.

A CIL suporta *tail-calls* através de um prefixo que pode ser adicionado a qualquer instrução de chamada (`call`, `callvirt`, etc.) – “`tail`”. Ao contrário do que ocorre na compilação para código nativo, infelizmente *tail-calls* apresentam problemas de desempenho, conforme será mostrado no Capítulo 5. Porém, até o momento parecem ser a única solução para o problema de crescimento da pilha de chamadas. Soluções menos gerais incluem o uso de instruções de desvio quando a função de destino for a função atual [MOS, BKR98, KS03b, BSS04] e, genericamente, da instrução `jmp` [MOS], que desvia para a primeira instrução de um dado método. Em particular, o uso de `jmp` possui a desvantagem de gerar código não verificável.

3.9 TRABALHOS RELACIONADOS

Nesta seção destacamos os principais trabalhos relacionados especialmente à integração de linguagens funcionais à Plataforma .NET. Apesar do foco deste trabalho estar em linguagens *lazy*, citaremos apenas para maior completude algumas implementações de linguagens estritas. Porém é importante ressaltar que estas não podem ser facilmente comparadas às primeiras, devido à diferença semântica e à menor complexidade de implementação.

3.9.1 Linguagens Funcionais Não Estritas

Finne [Hugb] incorporou ao interpretador Hugs [Huga] uma *bridge* voltada para a interoperabilidade entre a linguagem Haskell e o ambiente em .NET: o Hugs .NET. Implementada em C++, permite a comunicação Haskell \rightarrow CLR por meio de reflexão. A comunicação CLR \rightarrow Haskell é feita através da comunicação direta com o código não

gerenciado do ambiente de execução do Hugs. Descreveremos o Hugs .NET em maiores detalhes no Capítulo 6.

Meijer e seus colegas [MPvY01] criaram Mondrian, uma linguagem de *scripting* funcional não estrita com características multi-paradigma, incorporando alguns recursos de orientação a objeto. Seu compilador gera código C#, o que limita as possibilidades de otimizações de controle de fluxo e acesso a construções da CIL. O modelo de execução é o push/enter, e o código gerado lembra a execução de um interpretador, de modo que performance não é o seu principal objetivo. A principal limitação de Mondrian com relação ao desempenho está na geração de uma classe por *closure* – o que levaria a milhares de classes para um programa de médio porte. Além disso, podemos citar como desvantagem a inexistência de mecanismos de passagem de parâmetros que façam uso apenas da pilha da CLR. Finalmente, a implementação atual inclui um compilador Haskell → Mondrian.

Um protótipo de compilador Haskell para .NET chegou a ser desenvolvido como um *backend* para o GHC. O código gerado era em ILX [Sym01], uma extensão da CIL para suporte a linguagens funcionais que adota, em sua implementação, o modelo eval/apply e a geração de uma classe por *closure*. Infelizmente o projeto foi abandonado e nenhum resultado chegou a ser publicado.

Hunt e Perry [HP05] estão atualmente desenvolvendo um compilador Haskell para .NET que faz uso do modelo eval-apply e da geração de uma classe por *closure*⁴. O compilador em questão cobre grande parte da linguagem Haskell porém oferece um suporte limitado a bibliotecas. Até o momento, este trabalho não possui resultados publicados, e nenhuma versão foi disponibilizada.

3.9.2 Linguagens Funcionais Estritas

Existe um grande número de implementações de linguagens funcionais estritas para a Plataforma .NET, embora poucas publicações tenham sido feitas nesta área. Em particular, Scheme [Sch] e as linguagens derivadas de ML possuem uma ou mais implementações que geram código para a CLI. Além disso, novas linguagens surgiram, sendo a maioria com características multi-paradigma e projetadas essencialmente para interoperabilidade com os ambientes .NET ou Mono.

A implementação mais madura de linguagem funcional estrita até o momento encontrada é F# [F#], desenvolvida pela Microsoft Research [Mic]. F# é um “dialetto” de ML, baseada em OCaml [Cam] e adicionada de características imperativas e orientadas a objeto. Projetada especificamente para a Plataforma .NET, inclui suporte a *scripting* interativo, *profiling* de código, bibliotecas próprias, integração com a *IDE Visual Studio .NET*, entre outros recursos. Um dos seus principais objetivos é garantir a interoperabilidade com outras linguagens .NET da forma mais natural possível do ponto de vista do desenvolvedor. O suporte de F# a interoperabilidade será visto em mais detalhes no Capítulo 6. Com relação à estratégia de compilação, o compilador gera código ILX, abstraindo-se dos detalhes de implementação de *closures*, uniões discriminadas e polimorfismo paramétrico⁵. A geração de uma classe por *closure*, em particular, não chega a ser

⁴As informações pertinentes à estratégia de compilação são baseadas meramente em contatos pessoais.

⁵O time responsável por F# é o mesmo que mantém ILX.

um problema, uma vez que *closures* são muito menos comuns em linguagens estritas. A linguagem suporta avaliação não estrita, disponibilizando sintaxe específica para definir que um determinado parâmetro será avaliado sob demanda, além de um conjunto de classes para representar primitivas *lazy*. *Delegates* são utilizados apenas para interoperabilidade, para prover o uso de funções de alta ordem a partir de outras linguagens. Na versão atual utiliza o modelo eval/apply de avaliação, juntamente com a geração de uma classe por *closure* ou por subtipo de uniões discriminadas. Seu ambiente de execução (“*runtime system*”), cujo principal subconjunto é mostrado de forma simplificada na Figura 3.1, é composto por uma hierarquia de classes para representar funções de aridade menor ou igual a 4 (quatro). No modelo de avaliação implementado, para que uma função estaticamente desconhecida seja avaliada, rotinas do ambiente de execução são responsáveis por testar dinamicamente o tipo da função (ie.: sua aridade) por meio de instruções *isinst* e aplicá-la aos argumentos, de forma não currificada. A crítica a essa estratégia é a possível degradação de desempenho devido ao teste citado, particularmente em aplicações com uso intenso de funções de alta ordem. Para representar os tipos dos argumentos, Generics é utilizado, embora exista uma versão do compilador suportado pelas versões 1.x da CLR, em que Generics não era suportado. Infelizmente até o momento não há nenhum artigo ou resultado de performance publicado a respeito de F# de que se tenha conhecimento.

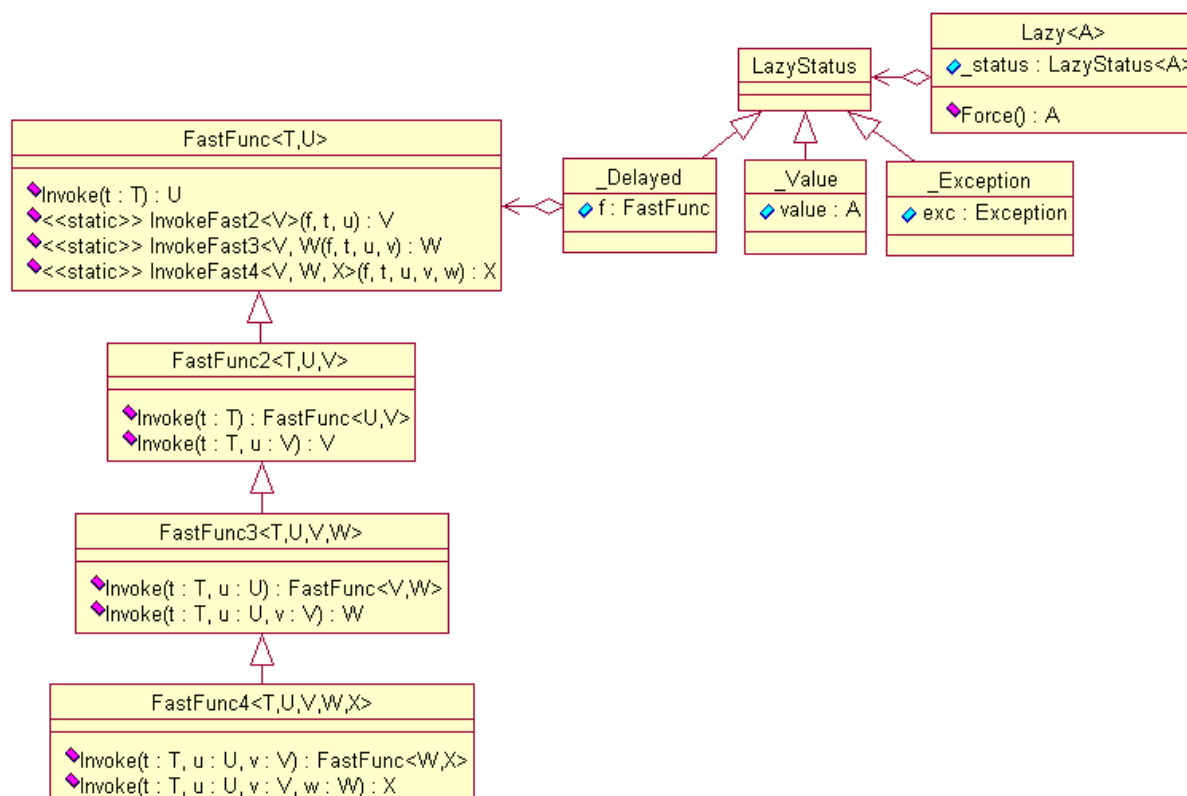


Figura 3.1. Parte do Ambiente de Execução de F# (versão simplificada)

Nemerle [MOS] é uma nova linguagem funcional projetada exclusivamente para .NET. Multi-paradigma, visa a introduzir programadores de linguagens como C ou C# ao paradigma funcional. Seu sistema de tipos é o mesmo da CLI e fornece interoperabilidade com outras linguagens de forma natural. Sua característica mais marcante é o suporte a meta-programação, permitindo que se estenda a linguagem por meio de macros. Não foram encontradas informações relevantes acerca da estratégia de compilação utilizada.

SML .NET [BKR04] é outro exemplo de implementação bem sucedida para a Plataforma .NET. Consiste em um subconjunto de SML'97 [MTHM97] adicionado de uma série de extensões para interoperabilidade com outras linguagens, além de fornecer integração com o *Visual Studio .NET*. Utiliza basicamente a mesma estratégia do seu predecessor para a JVM – MLj [BKR98] – inclui suporte a *tail-calls* através da geração de desvios e do uso do prefixo `.tail` da CIL. Adota uma estratégia de otimização baseada na análise do programa inteiro, o que leva a tempos de compilação muito altos e um suporte limitado a otimizações em compilação separada. Juntamente com a ausência de recursão polimórfica, essa estratégia permite a implementação de polimorfismo por meio de especialização. Tanto uniões discriminadas como *closures* são compiladas de forma a compartilhar classes o mais possível: construtores com os mesmos tipos de argumentos são compilados para uma mesma classe, assim como *closures* com diferentes aridades porém com os mesmos tipos de variáveis livres.

Outras implementações de ML incluem Moscow ML [KS03a, KS03b] e OCAMIL [OCa]. A primeira não utiliza tipos *unboxed*, mapeando tipos numéricos e *strings* em classes específicas. Faz uso do modelo eval/apply e gera uma classe por função e *closure*, além de fazer uso do suporte a *tail-calls* fornecido pela CIL e oferece um suporte limitado a interoperabilidade com outras linguagens. Já a última é um trabalho ainda experimental, não possuindo nenhuma documentação sobre a estratégia utilizada e nenhum suporte a interoperabilidade.

Bres, Serpette e Serrano [BSS04] desenvolveram um gerador de CIL para Scheme [Sch], como uma extensão ao Bigloo, compilador também capaz de gerar código C e JAVA *bytecodes*. O Bigloo para .NET utiliza estratégias de compilação semelhantes à versão para a JVM, destacando-se o compartilhamento de classes por meio do uso de código indexado para a compilação de *closures*. Nenhum suporte a interoperabilidade chegou a ser incluso e desempenho é o principal objetivo do compilador. Por ter sido desenvolvido à época das primeiras versões da CLI, não faz uso de Generics nem das otimizações incorporadas a *delegates*.

3.9.3 Implementações para a JVM

Conforme citado ao longo das seções 3.4 – 3.7, foram desenvolvidas várias implementações de linguagens funcionais tendo como alvo a JAVA *Virtual Machine*, incluindo Haskell [CiLH01, Tul96, Wak99], SML [BKR98] e Scheme [SS02, Bot98]. As principais motivações eram a portabilidade e o acesso à extensa biblioteca disponível para JAVA. Entretanto, devido às limitações encontradas (ausência de *tail-calls* e máquina virtual voltada para os requisitos de uma única linguagem) e ao advento do ambiente .NET e da CLI, a JVM perdeu espaço como plataforma alvo de implementação de linguagens não

orientadas a objeto.

3.10 CONSIDERAÇÕES FINAIS

Neste capítulo, introduzimos alguns elementos típicos de linguagens funcionais e descrevemos possíveis técnicas para implementá-los na Plataforma .NET. Algumas dessas técnicas podem ser utilizadas em outras plataformas de código gerenciado, como a JVM, enquanto outras são específicas da CLR e, conseqüentemente, do ambiente .NET, como por exemplo, *delegates* e *tail-calls*. Finalmente, citamos trabalhos relacionados tanto a linguagens funcionais estritas como não estritas. Nenhum desses pode ser comparado diretamente com o nosso trabalho, seja por utilizar metodologias completamente diferentes da nossa (ex.: uso de *bridge*), por não terem resultados publicados ou não terem sido concluídos, ou mesmo por estarem focados em linguagens estritas.

No Capítulo 3, introduziremos o compilador Haskell para .NET, descrevendo sua arquitetura, decisões de projeto, estratégias de compilação e ambiente de execução.

CAPÍTULO 4

O COMPILADOR HASKELL.NET

Neste capítulo, introduziremos o compilador Haskell desenvolvido ao longo desta pesquisa. A implementação desenvolvida é o ponto de partida para a integração da linguagem à Plataforma .NET. Entretanto, sua principal contribuição vai além do âmbito deste trabalho, pois fornece um ambiente de estudo, validação e comparação de estratégias de compilação de linguagens funcionais não estritas para essa plataforma.

Pelo fato de estar em sua primeira versão, a solução aqui apresentada não possui como principal objetivo oferecer alto desempenho, mas sim permitir a compilação de uma parte considerável da linguagem. Em geral, a compilação de uma linguagem para uma plataforma gerenciada relativamente recente é uma atividade complexa e tipicamente experimental, uma vez que não é possível conhecer *a priori* o desempenho oferecido por esta ou aquela técnica de compilação. Nessas situações, a escolha das técnicas em geral se dá por meio de critérios teóricos ou com base em resultados de implementações de outras linguagens semelhantes. Esse tipo de abordagem é válido em um primeiro momento, tendo sido adotado para esta primeira implementação. Entretanto, vale ressaltar que as escolhas devem ser reavaliadas em um segundo momento com relação ao seu desempenho, e tal reavaliação deve ser firmemente baseada em experimentos práticos. A partir de então ajustes e otimizações devem ser feitos para corrigir eventuais problemas. Trataremos deste tópico no próximo capítulo.

Este capítulo está organizado da seguinte forma: na Seção 4.1, descrevemos em linhas gerais a arquitetura da solução desenvolvida e as principais decisões de projeto; e na Seção 4.2, detalhamos as estratégias utilizadas para compilar programas Haskell para o ambiente da CLR.

4.1 ARQUITETURA

A implementação do compilador Haskell.NET consistiu em uma extensão do *Glasgow Haskell Compiler* (GHC) [JHH⁺93] com um novo gerador de código (“*back-end*”) capaz de gerar código CIL. Deste ponto em diante nos referiremos a CIL simplesmente como IL, por esta ser a sigla mais utilizada na literatura.

O GHC é considerado o compilador estado da arte para Haskell, capaz de executar várias otimizações antes da geração de código final. Dessa forma, o gerador de IL desenvolvido delega para o GHC toda a parte referente à análise léxica/sintática e verificação de tipos, ao mesmo tempo em que tira proveito das otimizações realizadas pelas etapas de compilação que o antecedem [dMS95].

Uma versão simplificada da arquitetura do GHC e do compilador Haskell.NET é mostrada na Figura 4.1. Por simplicidade, omitimos alguns componentes intermediários. A figura, bem como a explanação aqui exposta acerca da arquitetura, são baseadas na descrição dada por Peyton Jones e seus colegas [JHH⁺93].

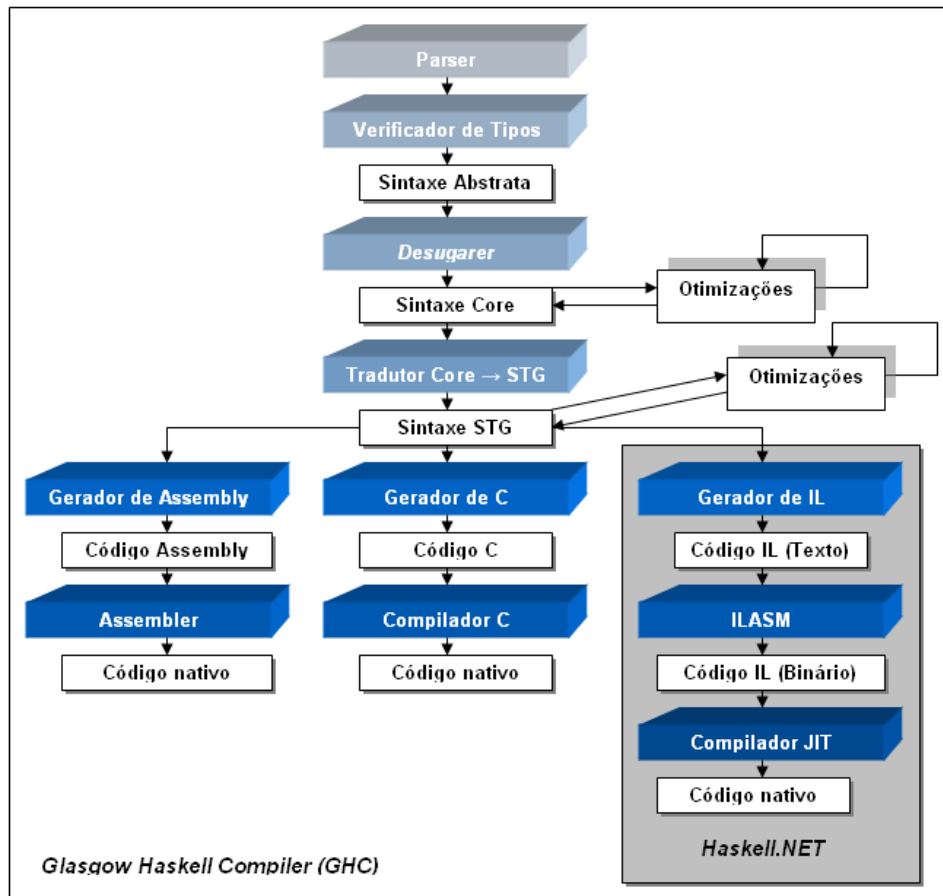


Figura 4.1. Arquitetura do Compilador

Os principais passos envolvidos no processo de geração de código são:

- i) após as etapas de inferência de tipos e *desugaring* do programa de origem, a sua sintaxe abstrata é convertida para uma linguagem funcional mais simples, denominada *Core*;
- ii) transformações opcionais otimizam o programa expresso em *Core*;
- iii) o programa *Core* resultante da etapa ii) é convertido para a linguagem *Shared Term Graph* (STG);
- iv) transformações opcionais otimizam o programa expresso em STG;
- v) o gerador de código final converte o programa em STG para C, traduzindo-o para uma notação intermediária abstrata (*Abstract C*), semelhante a uma árvore sintática. O código em *Abstract C* é impresso em código C para que possa ser compilado por um compilador C. A geração de código C só é adotada no modo de compilação por otimização. Por padrão, é utilizado um gerador de código *Assembly*.

A arquitetura do GHC provê suporte ao uso de quaisquer geradores de código, entre eles o do Haskell.NET. Conforme mostrado na Figura 4.1, o gerador de IL de fato gera código em uma versão textual da IL e invoca o montador ILASM, disponível na instalação da CLR, para gerar o código binário final. A partir de então, o código IL binário, em forma de executável ou biblioteca, pode ser executado pela CLR como qualquer programa .NET, utilizando por padrão compilação *just-in-time*.

Apesar de ser recomendado o uso de Core como linguagem fonte para novos geradores, a escolha de STG como linguagem fonte para o gerador de IL é justificada por três motivos:

- presença de informação que identifica quando uma *closure* deve ser alocada;
- presença de informação relativa a quais *closures* são atualizáveis;
- o fato de incorporar otimizações adicionais às que são realizadas em Core.

A linguagem STG será introduzida na Seção 4.2.

4.1.1 Gerador de IL

Nesta seção detalharemos a estrutura do gerador de IL, composto pelos módulos mostrados na Figura 4.2, em notação UML [RJB99].

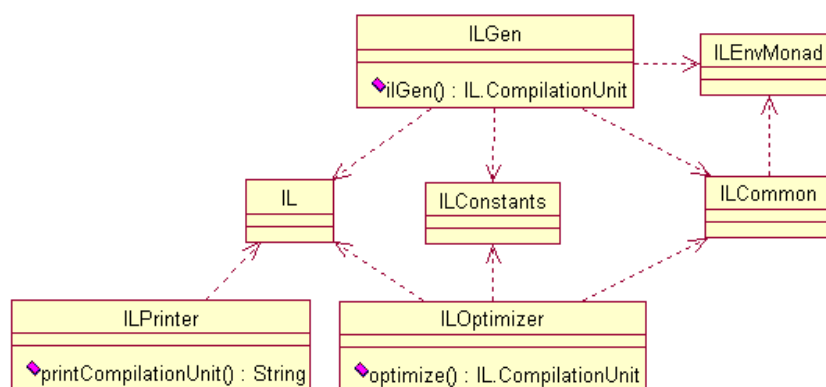


Figura 4.2. Arquitetura do Gerador de IL

A seguir, são detalhados os módulos exibidos na Figura 4.2:

- **IL**: contém os tipos de dados utilizados para representar a árvore sintática de um programa em IL;
- **ILGen**: módulo principal responsável por percorrer a árvore STG e gerar a árvore IL resultante da aplicação das regras de compilação;
- **ILEnvMonad**: gerencia o “estado” do gerador de IL (tabelas de símbolos, contadores de *labels*, etc.), por meio da utilização do conceito de Mônadas [Mog89] – uma técnica que permite a manipulação de estado em programas funcionais;

- **ILOptimizer**: aplica transformações na árvore IL para fins de otimização. Na versão atual, são aplicadas transformações simples, como por exemplo geração de *tail-calls*, geração da diretiva `.maxstack` para definição do tamanho máximo da pilha da CLR, etc.
- **ILPrinter**: *pretty-printer* responsável por gerar o código IL em forma textual. Esse código é em seguida compilado com o montador ILASM.
- **ILCommon**: contém funções utilitárias, usadas por diferentes módulos;
- **ILConstants**: contém constantes que representam nomes de *assemblies*, classes e métodos do ambiente de execução.

4.2 ESTRATÉGIA DE COMPILAÇÃO

A primeira versão publicada [MABS05a] do gerador de código desenvolvido neste trabalho era bastante simples e não fazia uso de recursos mais sofisticados como Generics para representar elementos polimórficos. Já a estratégia de compilação aqui apresentada, descrita em detalhes em um trabalho publicado posteriormente [MABS05b], incorporou o uso de Generics e de outras pequenas modificações.

Conforme já foi citado na Seção 4.1, utilizamos como linguagem de entrada para o gerador de IL a *Shared Term Graph Language* (STG), cuja gramática é mostrada na Figura 4.3¹.

A sigla STG referia-se originalmente à *Spineless Tagless G-Machine* [Jon92], máquina abstrata utilizada pelo GHC e que opera sobre a linguagem em questão. Entretanto, a linguagem STG é completamente independente da máquina abstrata STG, podendo ser empregada no contexto de outras máquinas abstratas. Neste trabalho, em particular, não implementamos a tecnologia STG em sua totalidade: adotamos apenas algumas das abordagens utilizadas por essa máquina, adaptando as técnicas de compilação como um todo à estrutura e restrições da CLR.

Optamos por uma implementação que gera apenas código verificável. Essa decisão de projeto dificulta o uso de recursos que poderiam contribuir para um melhor desempenho, como por exemplo controle direto da memória e o uso de ponteiros (para dados ou código). Por outro lado, garante que o código gerado não sofrerá restrições de segurança, um requisito importante para uma grande quantidade de aplicações.

4.2.1 Compilação STG → IL

Um dos principais objetivos da estratégia de compilação utilizada era evitar a geração de um grande número de classes. A quantidade de *closures* alocadas em um programa funcional não estrito de médio porte levaria a um número proibitivo de classes caso utilizássemos a abordagem da geração de uma classe por *closure*. Dessa forma, optamos por uma solução o mais escalável possível: uma única classe é gerada por módulo Haskell. Tal classe contém métodos e campos estáticos para representar funções e closures globais.

¹Adaptada de [Jon92]

| | | | |
|----------------------------|-----------|---|--------------------------------|
| Programa | $prog$ | $\rightarrow binds$ | |
| Declarações (“bindings”) | $binds$ | $\rightarrow var_1 = lf_1; \dots; var_n = lf_n$ | $n \geq 1$ |
| Formas <i>Lambda</i> | lf | $\rightarrow vars_f \backslash \pi vars_a \Rightarrow expr$ | |
| Indicador de atualização | π | $\rightarrow u$ | Atualizável |
| | | $ n$ | Não atualizável |
| Expressão | $expr$ | $\rightarrow \mathbf{let} binds \mathbf{in} expr$ | Definição local |
| | | $ \mathbf{letrec} binds \mathbf{in} expr$ | Recursão Local |
| | | $ \mathbf{case} expr \mathbf{of} alts$ | Expressão “case” |
| | | $ var atoms$ | Aplicação |
| | | $ constr atoms$ | Construtor saturado |
| | | $ prim atoms$ | Operador saturado |
| | | $ literal$ | |
| Alternativas | $alts$ | $\rightarrow aalt_1; \dots; aalt_n; default$ | $n \geq 1$ (Algébrico) |
| | | $ palt_1; \dots; palt_n; default$ | $n \geq 1$ (Primitivo) |
| Alternativa algébrica | $aalt$ | $\rightarrow constr vars \Rightarrow expr$ | |
| Alternativa primitiva | $palt$ | $\rightarrow literal \Rightarrow expr$ | |
| Alternativa padrão | $default$ | $\rightarrow var \Rightarrow expr$ | |
| | | $ \mathbf{default} \Rightarrow expr$ | |
| Literais | $literal$ | $\rightarrow 0\# 1\# \dots$ | Inteiros primitivos |
| | | $ \dots$ | |
| Operadores primitivos | $prim$ | $\rightarrow +\# -\# *\# /\#$ | Operadores inteiros primitivos |
| | | $ \dots$ | |
| Listas de variáveis | $vars$ | $\rightarrow \{var_1, \dots, var_n\}$ | $n \geq 0$ |
| Listas de átomos | $atoms$ | $\rightarrow \{atom_1, \dots, atom_n\}$ | $n \geq 0$ |
| | $atom$ | $\rightarrow var literal$ | |

Figura 4.3. Linguagem STG

Para atingir a escalabilidade desejada, foi necessário definir *closures* de um modo genérico, que permitisse independência em relação ao código encapsulado por elas. A forma encontrada para atingir tal objetivo foi a representação de *closures* como *delegates*, combinados com o uso de Generics. *Delegates* são objetos nativos da CLR, por si só generalizáveis com relação ao código o qual referenciam.

Agrupamos as *closures* em 4 (quatro) tipos principais:

- *closures* não atualizáveis (funções): mantém campos para as variáveis livres e um campo inteiro para a aridade;
- aplicações parciais: são as funções adicionadas de um campo que encapsula os argumentos previamente recebidos;
- *closures* atualizáveis (*thunks*): funções sem argumentos que mantém campos para as variáveis livres e um campo para o valor disponível após a atualização;
- construtores de dados: mantém campos para armazenar os argumentos. Em geral, dizemos que esses elementos e as aplicações parciais estão em *Weak Head Normal Form* (*WHNF*), ou seja, estão na forma objetivada pela avaliação sob demanda [Jon87].

Os três primeiros tipos encapsulam o código por eles referenciados por meio de uma abordagem baseada em *delegates* que será descrita na Seção 4.2.2.

Foram definidas famílias de classes para *closures* com $1...n$ variáveis livres e construtores com $1...n$ argumentos. O mesmo foi feito para os objetos responsáveis por armazenar argumentos de aplicações parciais de funções de aridade $1...n$. A hierarquia de classes implementada será descrita em detalhes na Seção 4.2.2.

No modelo de compilação adotado, cada módulo Haskell é compilado para uma classe, e cada função ou *closure* do módulo é em geral compilada para dois métodos estáticos dessa classe, conhecidos respectivamente como *fast entry point* e *slow entry point* [MJ04]. O primeiro é invocado na chamada saturada² a uma função/closure conhecida estaticamente. Os argumentos são armazenados diretamente na pilha da CLR, possibilitando dessa forma mecanismos eficientes de passagem de parâmetros por parte do compilador *just-in-time*, com por exemplo o uso de registradores. Caso declare variáveis livres, recebe também como parâmetro a *closure*, que as armazena. Já o *slow entry point* é utilizado em duas situações: quando não conhecemos em tempo de compilação qual função/closure está sendo chamada ou quando o número de argumentos é insuficiente (aplicação parcial). O único argumento que este método recebe é a *closure*: esta encapsula as variáveis livres, quando existentes, e argumentos resultantes de aplicações parciais anteriores. Os argumentos da aplicação propriamente dita são armazenados em uma pilha distinta da pilha de CLR, implementada como uma estrutura de dados baseada em *array*. De fato, mantemos ao todo quatro pilhas como essa, para conter argumentos dos tipos inteiro de 32 bits, ponto flutuante de 64 bits, `Object` e *closure*, respectivamente³. Sendo assim, o *slow entry point* verifica os tamanhos das pilhas correspondentes. Caso haja argumentos suficientes, estes são movidos para a pilha da CLR e o *fast entry point* é invocado. Caso contrário, uma aplicação parcial é instanciada e retornada. Em situações em que uma função/closure conhecida estaticamente é invocada com uma quantidade de argumentos superior à sua aridade, os argumentos excedentes são armazenados na pilha do nosso ambiente de execução e o *fast entry point* é chamado normalmente. O retorno do *fast entry point* nesses casos será uma função. Detalharemos a implementação do mecanismo de chamada de funções na Seção 4.2.2.

Existem situações em que o *slow entry point* não é gerado, como por exemplo para funções sem parâmetros ou quando anotações na árvore STG nos informam que determinada função sempre é aplicada de forma saturada. Nessas situações, instanciamos *closures* que apenas encapsulam as variáveis livres, quando existentes. Entretanto, na presença de *lambda-lifting* – uma transformação voltada para a eliminação de variáveis livres – essa alocação não seria gerada. A versão atual do compilador ainda não utiliza *lambda-lifting*, pois o suporte disponibilizado pelo GHC a essa transformação encontra-se desativado, em benefício de outras otimizações.

²Denominamos de *chamada saturada* aquela em que o número de argumentos passados é igual à aridade de função invocada.

³Outros tipos numéricos são convertidos para que possam utilizar as pilhas existentes. Por exemplo, números de ponto flutuante de 32 bits são convertidos para `float64`, assim como inteiros de 8 ou 16 bits são expandidos para 32 bits. Tipos menos comuns, como inteiros de 64 bits, são convertidos para `Object` via *boxing*. *Strings* e outros tipos de objetos são, da mesma forma, tratados como `Object`.

Adicionalmente aos métodos gerados, para cada função ou constante “*top level*” é gerado um campo estático na classe correspondente ao módulo no qual a função/constante foi declarada. Durante a carga inicial da classe em memória, tal campo é inicializado com a *closure* do tipo correspondente. Assim, o inicializador estático da classe tem por função instanciar todas as funções e constantes globais.

A abordagem para compilação de funções acima descrita, baseada em *fast e slow entry points*, segue o modelo push/enter. Apesar das críticas a esse modelo pelo fato de ele dificultar o uso da pilha nativa para a chamada de funções não conhecidas estaticamente, optamos por adotá-lo na versão inicial do compilador. A explicação para essa decisão de projeto está em uma breve análise de como seria a implementação caso optássemos pelo modelo eval/apply, que permite fazer uso da pilha nativa mesmo em situações em que a função chamada não é conhecida estaticamente.

Em uma implementação baseada em eval/apply, a estratégia mais natural na CLR seria definir diferentes classes para *closures* com diferentes aridades, utilizando Generics para representar os tipos dos seus argumentos, e implementar versões sobrecarregadas do método de avaliação com diferentes quantidades de parâmetros, a exemplo do que é feito em F# (Seção 3.9.2). Rotinas do ambiente de execução seriam por sua vez encarregadas de testar a aridade da *closure* chamada e realizar *casts* para o tipo de aridade correspondente, aplicando-a em seguida aos argumentos recebidos e fazendo uso da pilha da CLR. Existe um impacto negativo de desempenho associado a esses *casts*, principalmente quando utilizados com muita frequência. Alternativas que dispensassem as rotinas em questão e implementassem o teste e a chamada correta por meio de métodos virtuais sobrecarregados dispensariam os *casts*. Entretanto, a combinação de *delegates* com Generics para representar os tipos dos argumentos seria dificultado, ou mesmo inviabilizado, levando a outros de tipos de problemas.

Em segundo lugar, para cada classe representando uma *closure* de aridade *n*, seriam necessárias variações para diferentes números de variáveis livres, caso não quiséssemos usar mecanismos menos eficientes (ex.: *arrays*) para armazenar essas variáveis. Como consequência, o número de classes seria muito maior, gerando um maior custo de manutenção. Na presença de *lambda-lifting*, esse custo seria reduzido.

Uma alternativa cogitada inicialmente para evitar a geração de *slow entry points* e a implementação de pilhas foi a utilização de um recurso disponibilizado pela interface dos *delegates*: o método `DynamicInvoke`, que recebe um *array* de argumentos do tipo `System.Object` e retorna um objeto do mesmo tipo. Dessa forma, é possível criar *delegates* que referenciem qualquer método, independente de sua assinatura. Essa abordagem seria a mais flexível se não fosse por questões de desempenho: além de requerer operações de *boxing* para argumentos *unboxed*, foi verificada uma diferença em tempo de execução de cerca de 100 (cem) vezes em relação a uma chamada via método de interface.

Unões discriminadas também tiram proveito da técnica de compartilhamento de classes, de forma que construtores com a mesma aridade são representados por uma única classe, cujos argumentos possuem tipos genéricos. Cada construtor é composto por um campo inteiro (“*tag*”) que o identifica e por um conjunto de campos que representam os seus argumentos. Para esse tipo de *closure*, o método de avaliação retorna o próprio objeto. Finalmente, expressões `case` (ver Figura 4.3) são implementadas por meio de ins-

truções `switch` que inspecionam a *tag* presente no objeto. Um exemplo dessa compilação é mostrado abaixo:

```
switch (scrutinee.tag) {
  case 1: ...
  case 2:
    Pack_2<IClosure,IClosure>k = (Pack_2<IClosure,IClosure>) scrutinee;
    IClosure arg_1 = k.arg1;
    IClosure arg_2 = k.arg2;
    ...
}
```

Os esquemas representando as regras de compilação utilizadas pelo compilador são mostrados no Apêndice A.

4.2.2 Ambiente de Execução

O ambiente de execução do compilador, mostrado no diagrama de classes da Figura 4.4, contém os tipos necessários à representação de *closures*.

A Figura 4.4 mostra apenas classes com $n \leq 2$, onde n pode ser o número de variáveis livres, parâmetros de construtor, etc. De fato, existe um número muito maior de classes análogas às exibidas no diagrama de classes. Para situações em que não houver classe disponível, serão instanciadas classes que usam uma abordagem baseada em *array*. Nesse caso, utilizamos *array* de `IClosure`, por ser este o tipo mais comum⁴.

Os métodos de acesso às pilhas do ambiente de execução não são mostrados, por simplicidade. Basicamente, são métodos estáticos em uma classe separada que encapsulam o acesso às pilhas, implementadas pelo tipo `System.Collections.Generics.Stack<T>`, do *framework* básico da CLI.

Todos os tipos de *closures* implementam a interface `IClosure`, que disponibiliza o método `Enter`. Esse método é responsável por invocar o *slow-entry-point*, passando como parâmetro a própria *closure*. Para classes que representam construtores – prefixadas no diagrama por “`Pack_`”⁵ – o método `Enter` simplesmente retorna o próprio objeto. Já para os demais tipos de *closures*, a invocação do *slow entry point* é feita por meio da utilização de *delegates*.

Conforme pode ser notado no diagrama da Figura 4.4, *closures* do tipo função e *thunk* são *também* instâncias de *delegates*. Essa técnica não está em conformidade com o padrão da CLI, que estabelece que *delegates* não podem declarar campos. Por essa razão, essas classes não podem ser editadas em linguagens de alto nível como C#, forçando a escrita do código diretamente em IL. Além disso, a não conformidade com o padrão da CLI pode ocasionar problemas de portabilidade para outras implementações além da CLR. Por outro lado, implementar *closures* diretamente como *delegates* gera uma quantidade de objetos duas vezes menor do que se separássemos os dois conceitos e utilizássemos

⁴Tipos primitivos devem ser convertidos para `Pack<T>`, sendo `T` o tipo correspondente na CLR.

⁵O termo “`Pack`” é utilizado por Peyton Jones em “The Implementation of Functional Programming Languages [Jon87].

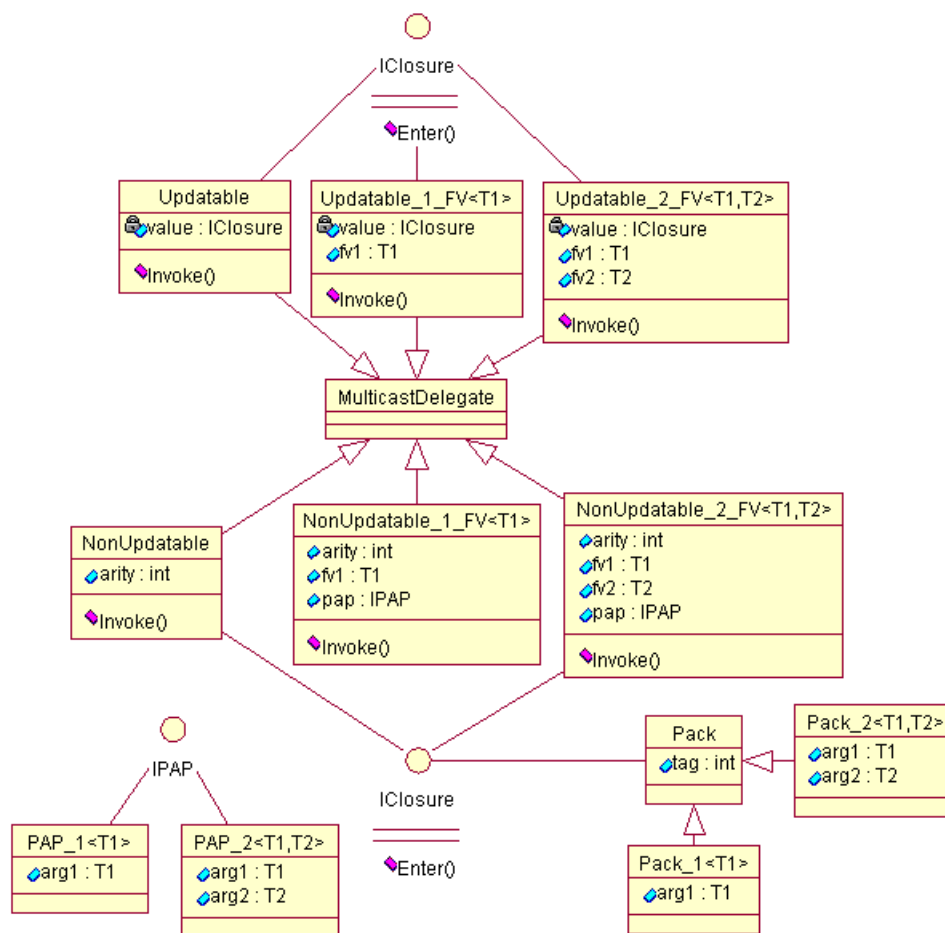


Figura 4.4. Ambiente de Execução

delegates apenas como campos das *closures*. Retomaremos este tópico no Capítulo 5.

O método **Enter** das *closures* não atualizáveis simplesmente chama o método **Invoke**, passando como argumento o próprio objeto. Já para as *closures* atualizáveis, os seguintes passos são executados:

- i) teste no campo **value** para verificar se a *closure* já foi atualizada. Este campo, do tipo **IClosure**, referencia o valor da *closure* após sua avaliação. Caso seu valor seja diferente de **null**, significa que a avaliação já foi feita e uma chamada ao método **Enter** do objeto referenciado por **value** é executada. Esse objeto pode estar em WHNF ou ser uma aplicação parcial. Entretanto, caso **value** seja igual a **null**, a execução procede segundo o passo ii).
- ii) Encapsula os tamanhos das pilhas do ambiente de execução em um objeto conhecido como *frame de atualização*. Esse objeto é então armazenado em uma *pilha de atualizações* que contém outros objetos do mesmo tipo. Os tamanhos das pilhas do ambiente de execução são tornados iguais a 0.

- iii) Executa uma chamada ao método `Invoke`, passando como parâmetro a própria *closure* (referência `this`). O resultado é armazenado no campo `value`.
- iv) Restaura os tamanhos das pilhas do ambiente de execução a partir do *frame* de atualização no topo da pilha de atualizações.

Os passos iii) e iv) têm por finalidade implementar aplicações parciais e segue estratégia semelhante à descrita por Peyton Jones [Jon92] na implementação original do modelo STG no GHC. Entretanto, existem formas de otimizá-los, conforme será mostrado no Capítulo 5.

Uma aplicação parcial é implementada como uma *closure* não atualizável, do mesmo tipo da *closure* aplicada parcialmente. Contém um campo que referencia uma instância da classe `PAPn`, onde n é a aridade da função associada. O campo `arity` é igual à aridade original da *closure* decrementada do número de argumentos recebidos.

A informação de aridade de uma função/*closure* é disponibilizada em tempo de compilação pela árvore STG, mas em geral isso só ocorre quando o código é compilado no modo de otimizações. Caso a aridade da *closure*/função não esteja disponível, as chamadas a ela serão executadas pelo *slow entry point*.

4.2.3 Exemplo

A estratégia de compilação é melhor compreendida por meio de um exemplo: a compilação da conhecida função `map`, que, dada uma função `f` e uma lista `l`, retorna uma nova lista cujos elementos são resultantes da aplicação de `f` aos elementos de `l`.

O código STG de `map` é mostrado abaixo.

```

1 map = {} \n {f,l}-> case l of
2       Nil {}-> Nil {}
3       Cons {x,xs}-> let fx = {f,x} \u {} -> f {x}
4                       fxs = {f,xs} \u -> map {f,xs}
5                       in Cons {fx,fxs}
```

O código resultante da compilação, correspondente ao *fast entry point* de `map` e aos códigos de `fx` e `fxs`, é mostrado na Figura 4.5, escrito em pseudocódigo C# para maior clareza. `fx` e `fxs` não geram *slow entry point* porque não possuem parâmetros. O prefixo `<tail>` é usado apenas para indicar que a chamada é compilada como uma *tail-call*. Esse prefixo não existe na sintaxe da linguagem C#.

O primeiro detalhe importante a ser ressaltado é os tipos dos parâmetros `f` e `l` do método `map`: `IClosure`. `f` é na verdade uma função qualquer, e `l` um objeto não avaliado.

Na linha 2, a avaliação de `l` é disparada pela expressão `case` da linha 1 da versão em STG. Na prática, apenas esse tipo de expressão provoca avaliações.

A linha 3 executa o comando `switch`, que por sua vez possui uma instrução `switch` correspondente em IL. Esse comando recebe como argumento o identificador de qual construtor está sendo utilizado (campo `tag` do objeto referenciado pela variável `scrutinee`).

```

1 public static IClosure map(IClosure f, IClosure l) {
2   Pack scrutinee = (Pack) l.Enter();
3   switch (scrutinee.tag) {
4     case 1: return new Pack(1);
5     case 2:
6       Pack_2<IClosure,IClosure>k = (Pack_2<IClosure,IClosure>) scrutinee;
7       IClosure arg_1 = k.arg1;
8       IClosure arg_2 = k.arg2;
9       Updatable_2_FV<IClosure,IClosure>fx_closure =
10         new Updatable_2_FV<IClosure,IClosure>(fx);
11       fx_closure.fv1 = arg_1; fx_closure.fv2 = f;
12       Updatable_2_FV<IClosure,IClosure>fxs_closure =
13         new Updatable_2_FV<IClosure,IClosure>(fxs);
14       fxs_closure.fv1 = f; fxs_closure.fv2 = arg_2;
15       return new Pack_2<IClosure,IClosure>(2,fx_closure,fxs_closure);
16   }
17 }
19 public static IClosure fx(Updatable_2_FV<IClosure,IClosure> closure){
20   RuntimeSystem.Push(closure.fv1);
21   return <tail> closure.fv2.Enter();
22 }
23 public static IClosure fxs(Updatable_2_FV<IClosure,IClosure> closure){
24   return <tail>map(closure.fv1, closure.fv2);
25 }

```

Figura 4.5. Compilação de `map`, `fx` e `fxs`

Caso o construtor `Nil` esteja sendo utilizado (`tag = 1`), é retornada uma lista vazia, representada simplesmente por uma instância da classe `Pack` cuja `tag` é igual a 1. Embora não seja mostrado nesse exemplo, é importante ressaltar que algumas vezes o GHC gera um código STG que, ao invés de instanciar um construtor nulo (sem argumentos), utiliza uma constante que representa esse tipo de construtor, evitando a alocação de memória.

Caso o construtor `Cons` esteja sendo usado, as linhas 6-8 recuperam os campos (argumentos) do construtor. As linhas 9-14 instanciam as *closures* `fx` e `fxs` declaradas nas linhas 3-4 do programa STG, inicializando suas variáveis livres. Sendo instâncias de *delegates*, elas recebem de fato 2 argumentos: o objeto ao qual o método pertence (no nosso caso, `null`, uma vez que estamos lidando com métodos estáticos), e o ponteiro para o código a ser referenciado. No exemplo, como adotamos a sintaxe de C# para *delegates*, o primeiro argumento, caso seja `null`, não aparece explicitamente, e o segundo é o próprio nome do método. Na prática, o ponteiro para o código é carregado da pilha da CLR por meio da instrução `IL ldftn`, parametrizada pelos metadados do método (nome, assinatura, classe/assembly de declaração, etc.).

A linha 15 instancia a lista referenciada por `Cons fx,fxs` na linha 5 do código STG, passando como argumentos a `tag` identificadora (`tag = 2` para o construtor `Cons`) e os

```

1 public static IClosure map(NonUpdatable closure){
2   PAP_2<Closure, Closure> pap; int stackSize;
3   switch (closure.arity){
4     case 1:pap = (PAP_2<IClosure, IClosure>) closure.pap;
6       stackSize = RuntimeSystem.closureStackSize;
7       if(stackSize >= 1) return map(pap.arg1,
8                                     RuntimeSystem.PopClosure());
9       pap = (PAP_2<IClosure, IClosure>) pap.Clone(); break;
10    case 2:pap = (PAP_2<IClosure, IClosure>) closure.pap;
11      stackSize = RuntimeSystem.closureStackSize;
12      if(stackSize >= 2) return map(RuntimeSystem.PopClosure(),
13                                    RuntimeSystem.PopClosure());
14      if(pap != null)
15        {pap =(PAP_2<IClosure,IClosure>) pap.Clone(); break;}
16      pap = new PAP_2<IClosure, IClosure>();
17      if(RuntimeSystem.closureStackSize > 0){
18        pap.arg1 = RuntimeSystem.PopClosure();
19        if(RuntimeSystem.closureStackSize > 0)
20          pap.arg2 = RuntimeSystem.PopClosure();
21      }
22      break;
23    default:pap = (PAP_2<IClosure, IClosure>) closure.pap;
24      return map(pap.arg1, pap.arg2);
25  }
26  NonUpdatable newClosure = new NonUpdatable(map);
27  newClosure.arity = closure.arity - stackSize;
28  newClosure.pap = pap;
29  return newClosure;
30 }

```

Figura 4.6. *Slow Entry Point* de map

argumentos (*fx* e *fxs*).

Conforme dito anteriormente, em nossa implementação, as *closures* referenciam o *slow entry point*. *fx* e *fxs*, por não declararem receber parâmetros, não possuem *slow entry point*. Dessa forma, as *closures* a elas alocadas referenciam os próprios métodos *fx* e *fxs*.

A começar pelo método *fx*, que implementa o código da *closure fx*: ao ser chamado, ele recebe como argumento o objeto alocado na linha 9. A linha 20 empilha na pilha do ambiente de execução a primeira variável livre e, na linha 21, executa o código da *closure* referenciada pela segunda variável livre. Essas duas linhas de código refletem a linha 3 da versão em STG. Analogamente, o método *fxs* consiste na chamada à função *map* na linha 4 do programa STG. Os argumentos são as variáveis livres da *closure fxs*. Como a aridade de *map* é conhecida estaticamente, uma chamada ao seu *fast entry point* é gerada.

Na Figura 4.6, temos o código correspondente ao *slow entry point* de *map*, gerado na

mesma classe, embora mostrado separadamente para facilitar a discussão.

O código do *slow entry point* mostrado primeiramente testa o campo `arity` da *closure* (linha 3), para verificar quantos argumentos ainda são necessários para que ela possa ser avaliada. Dependendo do valor desse campo e da quantidade de argumentos presentes na pilha do ambiente de execução, a chamada ao *fast entry point* é gerada, utilizando os argumentos guardados previamente na *closure* e/ou na pilha (linhas 7, 12 e 24). Caso não haja uma quantidade de argumentos suficiente para a avaliação, uma aplicação parcial é gerada, por meio de instanciação de uma nova *closure* que, da mesma forma, referencia o código do *slow entry point* (linha 26). Além disso, um novo objeto do tipo `IPAP` (instância de `PAP_2`) é criado, por meio do método `Clone` da interface `Cloneable` da biblioteca padrão do .NET (linhas 9 e 15⁶). Com esse método, cria-se um objeto que contém os mesmos argumentos guardados na instância anterior de `IPAP`, evitando-se gerar o código de cópia desses argumentos no corpo do programa. Além disso, são adicionados a ele os novos argumentos encontrados na pilha do ambiente de execução. Como pode ser visto nas linhas 17-20, esses argumentos são adicionados à medida que são encontrados na pilha. Quando esta “esvazia” (campo `RuntimeSystem.closureStackSize = 0`), a cópia de argumentos é interrompida. Os valores do campo `closureStackSize` e de todos outros campos que contém os tamanhos das outras pilhas são incrementados/decrementados pelos métodos estáticos `Push()` e `PopX()` da classe `RuntimeSystem`.

À primeira vista, o *overhead* de chamada ao método de interface `Clone` para instanciar aplicações parciais pode parecer significativo. Porém, esse impacto não foi uma preocupação na implementação inicial.

Finalmente, o uso de um único objeto para guardar argumentos de aplicações parciais tem como vantagem o fato de não guardarmos argumentos diretamente nas *closures*. Se assim fosse, o gasto de memória associado a estas seria maior, uma vez que elas estariam guardando campos muitas vezes não utilizados. Além disso, essa abordagem simplifica o ambiente de execução, uma vez que não precisamos definir classes para cada número de parâmetros, da mesma maneira que fazemos para os diferentes números de variáveis livres.

4.3 CONSIDERAÇÕES FINAIS

Neste capítulo, descrevemos o compilador Haskell.NET. Analisamos as decisões de projeto, estratégias e regras de compilação e o ambiente de execução que a elas dá suporte. Na versão atual, o compilador suporta apenas um subconjunto do prelúdio Haskell, suficiente entretanto para as primeiras validações e testes de desempenho. Em particular, o suporte a algumas funções de baixo nível, responsáveis pela parte de entrada/saída e tratamento de erros, foi implementado em C#. O suporte a tipos como `BigInteger` também se encontra implementado em C#.

A geração de IL em formato textual será rediscutida no Capítulo 7, no contexto do uso de APIs mais recentes próprias para facilitar e otimizar a geração de código.

⁶Conforme será citado no Apêndice A, uma melhora que pode ser feita no código aqui mostrado é remover as linhas 14-15 e mover o conteúdo da linha 16 para dentro da condição `if(RuntimeSystem.closureStackSize > 0)`, evitando a eventual alocação de objetos `PAP` desnecessários.

O modelo push/enter mostrado pode ainda ser otimizado com relação à estrutura de dados usada na implementação das pilhas. O relacionamento das subclasses de `IPAP` com a interface `Cloneable` pode ser removido e o método `Clone` pode ser redefinido em cada subclasse de `IPAP`, de forma a retornar um objeto com o tipo da própria subclasse, ao invés de `Object`. Dessa forma, evitaríamos os *casts* adicionais. Além disso, permitiríamos que `IPAP` fosse uma classe abstrata ao invés de interface, uma vez que nessas últimas as invocações de métodos são mais lentas.

Ao mesmo tempo, existe a possibilidade de testarmos futuramente o modelo eval/apply, combinando uma estratégia semelhante à utilizada por F# com o uso de *delegates*, para confirmar se de fato os problemas citados na Seção 4.2 podem causar impactos relevantes ao desempenho.

No Capítulo 5, analisaremos a implementação sob o ponto de vista do desempenho, descrevendo variações e otimizações. Em particular, analisaremos uma versão alternativa do ambiente de execução que resolve o problema da não conformidade com a CLI no uso de *delegates* e o impacto obtido no desempenho.

CAPÍTULO 5

AVALIAÇÃO DE DESEMPENHO E OTIMIZAÇÕES

Este capítulo trata das avaliações de desempenho feitas a partir da versão inicial do compilador Haskell.NET, descrita no Capítulo 4, e está estruturado da seguinte forma:

- A Seção 5.1 explica como os experimentos foram feitos;
- As seções 5.2 – 5.4 descrevem resultados obtidos por análises de impacto de certas construções e comparações entre diferentes abordagens;
- A Seção 5.5 descreve o problema das CAFs e como elas afetam o desempenho de programas funcionais não estritos;
- A Seção 5.6 descreve os resultados das otimizações realizadas, a maioria delas no campo do consumo de memória;
- A Seção 5.7 mostra a comparação entre a velocidade de execução do código gerado pela nossa solução e pelo gerador de código nativo do GHC.

5.1 METODOLOGIA

Para medir o desempenho dos programas Haskell foi utilizado um subconjunto do grupo Imaginário do *benchmark* Nofib [Par93]. Idealmente, deveria ser utilizado o grupo Real, uma vez que este, como o próprio nome dá a entender, contém programas mais realistas e complexos. Entretanto, devido ao suporte limitado às bibliotecas do GHC que o gerador de código IL ainda possui, utilizamos um conjunto de programas simples, que acessam o subconjunto do prelúdio suportado atualmente. Mesmo esses programas são de grande importância: eles podem apontar os principais problemas e “gargalos” (ex.: gerenciamento de memória), além de darem uma noção preliminar do desempenho do código gerado pelo compilador e permitirem a comparação entre diferentes estratégias de compilação.

O Nofib não sugere entradas esperadas para os programas, mas sim saídas, o que dificulta a identificação de qual entrada deve ser utilizada, uma vez que ela deve ser inferida por meio de testes. Em geral, as entradas utilizadas nos nossos testes são superiores às aquelas sugeridas por padrão pelo Nofib, uma vez que esse *benchmark* não é atualizado de modo a refletir as plataformas de *hardware* mais recentes. A modificação dessas entradas evita tempos de execução muito curtos, que sejam dominados pelo tempo de inicialização da máquina virtual e dos programas. Para as comparações com o gerador de código nativo do GHC, as entradas utilizadas nos nossos testes foram:

- `Digits.e1`: 2000

- Digits_e2: 3000
- Exp3: 9
- Primes: 4500
- Queens: 13
- Tak: tupla (12,1,25)
- Wheel-sieve1: 300000
- Wheel-sieve2: 80000

Em alguns testes, as entradas são diferentes das que aqui estão relacionadas. Quando isso ocorrer, elas serão especificadas nas tabelas, entre parênteses, ao lado do nome do programa.

Cada medição realizada correspondia à média dos tempos de execução de 5 (cinco) execuções consecutivas de cada programa, excluindo-se a primeira. Os tempos foram medidos utilizando o comando `time` do utilitário *Cygwin*[Cyg]. Em todas as tabelas aqui apresentadas, os tempos são exibidos em segundos, e nas tabelas em que são apresentadas percentagens, o sinal negativo (-) indica piora, enquanto a ausência de sinal indica melhora.

A configuração de *software* e *hardware* utilizada é mostrada na Tabela 5.1.

| Parâmetro | Valor |
|---------------------|--------------------------|
| Sistema operacional | Windows™ XP Professional |
| Versão da CLR | .NET™ Framework 2.0 |
| Versão do GHC | 6.2.2 |
| Processador | Pentium™ 4 HT 3GHz |
| Memória RAM | 1GB |

Tabela 5.1. Configuração do Ambiente de Testes

As ferramentas de análise de desempenho utilizadas incluíram o *CLR Profiler*[Pro], para *profiling* de memória, o analisador de desempenho do *Visual Studio .NET* 2005 [.NEc] para *profiling* de tempo e o *Performance Monitor*, que disponibiliza alguns contadores de desempenho. Infelizmente as duas primeiras mostraram limitações diante de programas com tempos de execução da ordem de alguns segundos, de forma que os testes nelas executados tinham que ser feitos com entradas muito pequenas. Já a última não apresenta essa limitação, porém os seus contadores são atualizados a cada instante com médias sobre as últimas amostras, o que algumas vezes leva a resultados imprecisos.

5.2 ANÁLISE DO IMPACTO DE CASTS

Na programação orientada a objetos, existem várias situações em que são necessárias conversões (“*casts*”) entre objetos pertencentes a diferentes tipos. Em geral, quando o tipo de destino é um supertipo do tipo de origem, a conversão pode ser feita de forma segura, sem necessidade de verificação em tempo de execução. Por outro lado, quando o tipo de destino é mais específico, é necessário que o ambiente de execução verifique se os dois são compatíveis, ou seja, se o objeto é do tipo de destino.

Na CLR, essa verificação é um dos pré-requisitos para a geração de código verificável e é implementada por meio de duas instruções, denominadas instruções de *cast*: **castclass** e **isinst** (Seção 5.2.1). Na implementação do compilador Haskell.NET, apesar de estarmos lidando com uma linguagem estaticamente tipada (Haskell), foi necessário usar tal tipo de instrução. Um exemplo de código Haskell e sua compilação são mostrados nas figuras 5.1 e 5.2, respectivamente.

```
data T t u v = A t u | B v
case p of
  A x y -> ...código (acesso aos campos x e y, etc.)
  B z -> ...código (acesso ao campo z, etc.)
```

Figura 5.1. Casamento de Padrão

```
Pack p = (Pack) closure.Enter();
switch(p.Tag){
  case 1:
    Pack_2<IClosure, IClosure> v1 = (Pack_2<IClosure, IClosure>) p;
    //...código (acesso aos campos x e y, etc.)
  case 2:
    Pack_1<IClosure> v2 = (Pack_1<IClosure>) p;
    //...código (acesso ao campo z, etc.)
}
```

Figura 5.2. Uso de *casts* na Compilação de Casamento de Padrão

Em todo caso, o compilador pode vir a ser configurado para gerar ou não código verificável com respeito a *casts*. Como regra geral, aconselha-se ao usuário medir o desempenho do código compilado nas duas versões antes de decidir se a remoção dessas construções compensa a geração de código não verificável.

5.2.1 castclass X isinst

As operações de *cast* disponíveis na IL possuem o mesmo objetivo: verificar se o tipo de uma referência é compatível com um tipo mais específico ou ortogonal. O que de fato

ocorre não é a modificação do objeto nem mesmo a criação de uma nova instância, mas sim o retorno da própria referência após a checagem. A diferença entre elas reside no comportamento diante da incompatibilidade entre os tipos.

A instrução `castclass` realiza o *cast* de um objeto do tipo *T* para um tipo *U* desde que *T* seja subtipo de *U* ou *T* implemente *U* (sendo *U* uma interface). Caso contrário, uma exceção é levantada.

A instrução `isinst`, da mesma forma, realiza o *cast* de um objeto do tipo *T* para um tipo *U* desde que *T* seja subtipo de *U* ou *T* implemente *U* (sendo *U* uma interface). Porém, ela retorna `null` quando isso não ocorre, ao invés de levantar exceção, como acontece com `castclass`.

Ambas as instruções são implementadas por meio de buscas lineares pela lista de interfaces suportadas por um objeto ou pela sua hierarquia de superclasses, representada por uma lista ligada [BS03]. Portanto, objetos que implementam um grande número de interfaces e/ou que apresentam uma hierarquia de herança muito profunda costumam gerar maior *overhead* em instruções de *cast*.

5.2.2 Resultados

Foram realizados testes para medir os tempos de execução de duas versões do compilador: uma que gera *casts* utilizando a instrução `isinst`, e outra que não os gera, levando a código não verificável. Em seguida, avaliou-se o desempenho da primeira versão com respeito ao uso da instrução `castclass`. Os resultados são resumidos na Tabela 5.2.

| Programa | Sem casts | Com casts – isinst | Com casts – castclass |
|-----------------------|-----------|--------------------|-----------------------|
| Digits.e1 (1500) | 7,74 | 8,28 | 8,00 |
| Digits.e2 (2000) | 12,86 | 13,12 | 13,36 |
| Exp3 | 32,56 | 33,47 | 32,67 |
| Primes (3500) | 9,49 | 9,70 | 9,58 |
| Queens (12) | 3,36 | 4,72 | 4,17 |
| Tak | 22,23 | 22,34 | 22,37 |
| Wheel-sieve1 (200000) | 14,64 | 16,26 | 15,57 |
| Wheel-sieve2 (40000) | 6,33 | 6,44 | 6,52 |

Tabela 5.2. Impacto das Operações de *cast*

Para o primeiro teste, observou-se que o uso de *casts* incorreu em um impacto negativo de 5% a 8%, aproximadamente. Entretanto, observamos significativas variações nos resultados, com o `Queens`, que apresentou impacto negativo de aproximadamente 40% com o uso de *casts*, e `Wheel-sieve1`, que teve seu desempenho piorado em cerca de 11%. Porém, para os demais programas, os impactos concentraram-se na faixa de 0,5% a 7%. O fato de termos evitado hierarquias de herança profundas, mesmo que em detrimento do reuso de código na escrita do ambiente de execução, certamente contribuiu para que os *casts* não causassem um maior impacto no desempenho.

O ponto seguinte a ser considerado é a diferença de desempenho obtida entre o uso das

instruções `castclass` e `isinst`. Foi observada uma pequena diferença (1,36%) em favor da instrução `castclass`. Dentre os programas analisados, 3 (três) apresentaram uma diferença de 0,1% a 1,8% em favor de `isinst`, e 5 (cinco) obtiveram ganhos de 1,25% a 11,6% com o uso de `castclass`. O fato do compilador *just-in-time* utilizar diferentes rotinas para as duas instruções [BS03] pode explicar a diferença de desempenho obtida. Optamos assim por utilizar `castclass` na versão final do compilador.

Conforme pôde ser notado, o uso de *casts* não apresentou *overhead* significativo para a maioria dos programas testados, não justificando portanto a geração de código não verificável como forma de otimização para qualquer programa Haskell. Entretanto, o uso de *casts* gerou maior impacto negativo para dois programas em particular, indicando a necessidade de remoção dos mesmos em sacrifício de verificabilidade em aplicações críticas em desempenho. Finalmente, o impacto negativo obtido para esses programas pode apontar para a necessidade de melhoras por parte da CLR com respeito à implementação de *casts*.

5.3 ANÁLISE DO IMPACTO DE TAIL-CALLS

Conforme vimos no Capítulo 4, *tail-calls* consistem em um mecanismo para evitar o crescimento excessivo da pilha de chamadas, eliminando *frames* que não serão mais utilizados.

Na IL, *tail-calls* são implementadas por meio do prefixo “`.tail,`” que pode ser adicionado a instruções de chamada a método. No compilador Haskell.NET, tal prefixo é adicionado em um passo de otimização, a cada chamada de método seguida por uma instrução de retorno obrigatória – “`ret`” – que sinaliza que o *frame* pode ser descartado.

Existe uma penalidade de desempenho associada ao uso de *tail-calls*, reconhecida pelo time responsável pela CLR, que faz com que seu uso, na maioria das vezes, piore o desempenho de um programa, ao invés de melhorá-lo. A piora do desempenho é devida a várias verificações e controles que são realizados pela CLR [Mic05].

Entretanto, *tail-calls* são necessárias em programas funcionais como forma de evitar o estouro da pilha de chamadas, não podendo ser simplesmente eliminadas. Implementações alternativas como o uso da técnica do trampolim (Seção 3.8.1) podem ser testadas posteriormente.

Avaliamos o impacto de *tail-calls* em programas C# e em programas Haskell compilados com o compilador Haskell.NET. Em C#, para o trecho de código abaixo¹:

```
1 private static double OriginalFunction (double d, int k) {  
2     if (k > 1) return OriginalFunction (d * (k + 1) / k, k - 1);  
3     else return d;  
4 }
```

foram testadas duas versões: uma em que a linha 2 era compilada como uma chamada normal, e a outra em que a linha 2 era compilada como uma chamada adicionada do prefixo `.tail`. Os resultados para 10.000 execuções consecutivas com os valores 1 e 20.000

¹O código mostrado no exemplo foi retirado do endereço <http://www.jelovic.com/weblog/e59.htm>.

para **d** e **k**, respectivamente, levaram a tempos de execução em média 94% maiores na versão com *tail-call* em relação à versão sem *tail-call*.

Nos programas Haskell analisados, foi verificado um impacto negativo para a maioria dos programas com o uso de *tail-calls*, porém com grandes variações. **Tak**, por exemplo, teve seu tempo médio de execução aumentado em cerca de 31%, enquanto outros programas como **Wheel-sieve2** e uma versão modificada de **Exp** que faz grande uso de *tail-calls* obtiveram uma melhora de 16% e 20%, respectivamente. Um dos programas, **Wheel-sieve1**, levantou exceção de estouro da pilha sem *tail-calls*, como era de se esperar. Os resultados estão resumidos no gráfico da Tabela 5.3.

| Programa | Com tail-calls | Sem tail-calls | Impacto das tail-calls |
|-----------------------|----------------|------------------|------------------------|
| Digits_e1 (1500) | 8,04 | 7,39 | -8,82% |
| Digits_e2 (2000) | 13,39 | 12,75 | -5,05% |
| Exp3 | 32,31 | 30,75 | -5,08% |
| Exp3 (modificado) | 26,12 | 32,61 | 19,89% |
| Primes (3500) | 9,66 | 9,06 | -6,61% |
| Queens (12) | 4,19 | 3,68 | -13,89% |
| Tak | 22,41 | 17,11 | -30,93% |
| Wheel-sieve1 (200000) | 14,64 | Estouro da pilha | - |
| Wheel-sieve2 (40000) | 6,42 | 7,68 | 16,39% |

Tabela 5.3. Impacto do Uso de *Tail-calls*

Uma explicação para os ganhos de desempenho em alguns programas com o uso de *tail-calls* reside no consumo de memória. Programas funcionais realizam muita alocação de objetos na *heap*. Em situações em que *tail-calls* poderiam ser utilizadas mas não o são, *frames* desnecessários continuam existindo por mais tempo, levando a execuções mais lentas do *garbage collector*, uma vez que este precisa percorrer um maior número de *frames*. Além disso, referências para objetos que não serão mais utilizados sobrevivem na pilha de chamadas por mais tempo, levando à exaustão da *heap* mais rapidamente e muitas vezes à promoção de objetos para a geração 2, onde as coletas são mais lentas. Para confirmar essa hipótese, foi analisado o tempo médio gasto em coletas nos programas testados. Conforme pode ser visto na Tabela 5.4, todos os programas apresentaram um aumento considerável para o valor médio do contador de performance *%Time in GC*², disponibilizado pelo *Performance Monitor*. **Exp**, em particular, apresentou um aumento de 50% na quantidade de coletas na geração 2. Para os demais programas não se perceberam alterações na quantidade de coletas nas diferentes gerações. Conclui-se portanto que o ganho obtido em gerenciamento de memória pode amenizar o impacto negativo de *tail-calls* e até revertê-lo para positivo em alguns casos.

²Embora essa métrica não seja ideal, pelo fato de levar em conta a média das últimas observações e não da execução como um todo, aqui ela deve ser considerada devido à grande diferença de valores nos dois cenários.

| Programa | Com <i>tail-call</i> | Sem <i>tail-call</i> |
|--------------|----------------------|----------------------|
| Digits.e1 | 35,84% | 50,16% |
| Digits.e2 | 44,35% | 49,34% |
| Exp | 39,16% | 62,27% |
| Primes | 36,43% | 42,57% |
| Queens | 10,80% | 53,69% |
| Tak | 0% | 0% |
| Wheel-sieve2 | 53,33% | 61,18% |

Tabela 5.4. Valores para %Tempo em GC

5.4 VARIAÇÕES NO USO DE DELEGATES

No capítulo anterior, assumimos que o uso de *delegates* separados de *closures* poderia incorrer em um grande prejuízo de desempenho devido à maior quantidade de objetos. Dessa forma, naquele momento optamos por usar *delegates* diretamente como *closures*, mesmo em detrimento da portabilidade entre diferentes implementações da CLI. Na solução apresentada, adicionamos campos e métodos além do que é prescrito pela especificação. Uma outra consequência dessa abordagem é a obrigatoriedade do uso de IL para gerar o código do ambiente de execução. Por outro lado, citamos a possibilidade de implementar um ambiente de execução alternativo para integração com outras implementações da CLI que não suportem o modelo adotado. Tal ambiente de execução foi implementado e nesta seção analisaremos o impacto do mesmo no desempenho do código gerado pelo compilador. A Figura 5.3 mostra o diagrama de classes da implementação gerada.

Um resultado inesperado dos testes de desempenho é que o tempo de execução melhorou para a maioria dos programas analisados no segundo ambiente de execução, como pode ser visto na Tabela 5.5. Apenas dois programas tiveram pior desempenho: **Exp** e **Wheel-sieve2**, que apresentam grande atividade de *garbage collection*, conforme pode ser visto na Tabela 5.4. Para esses programas, a performance piorou em 24% e 11%, respectivamente, enquanto a atividade de GC piorou em 10% e 5%. Porém, um teste posterior para o programa **Wheel-sieve2**, que utilizou uma entrada duas vezes maior, revelou outro resultado inesperado: uma vantagem de cerca de 94% em favor da segunda versão do ambiente de execução. Além disso, outros programas tiveram a atividade de GC aumentada ao passo que seus tempos de execução ainda assim foram melhorados.

Possíveis explicações para as melhoras de desempenho devido à modificação implementada para o uso de *delegates* são:

- Alguma otimização relacionada a *delegates* implementada internamente pela CLR e que é perdida quando acrescentamos campos e métodos aos mesmos. Análises na implementação de *delegates* na CLR precisariam ser feitas para comprovar essa hipótese.
- O fato de o novo ambiente de execução implementar o método **Enter** como um

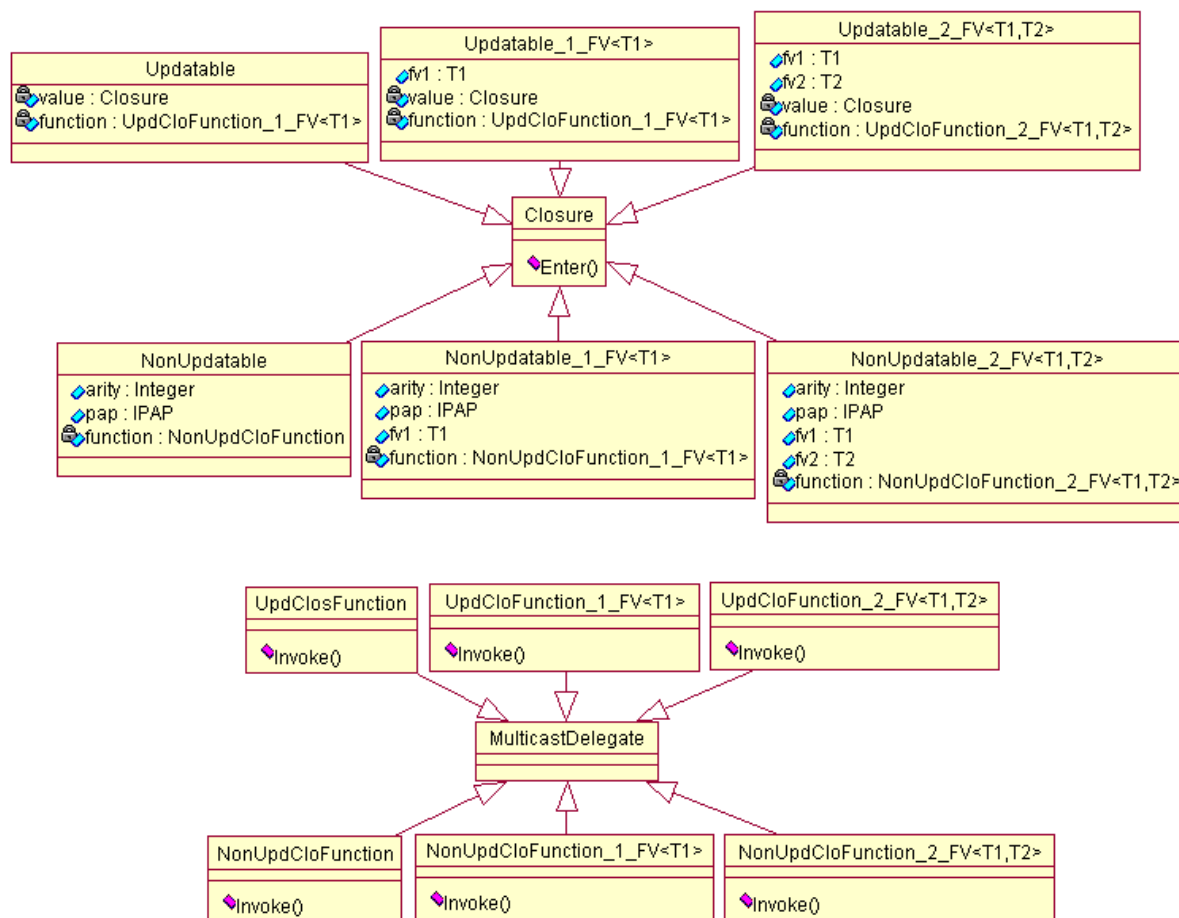


Figura 5.3. Separação *Closures–Delegates* – Diagrama de Classes

método de classe abstrata e não de interface, como ocorria na primeira versão, onde o uso de interface era imperativo devido à ausência de herança múltipla na CLI.

- O fato de as *closures* ocuparem um espaço menor do que ocupavam quando eram instâncias de *delegates*. Isso poderia levar à maior velocidade de compactação da *heap* ou mesmo à redução da memória ocupada por *closures* já atualizadas, o que poderia levar a uma redução na demanda por coletas.

Entretanto o terceiro fator contraria o fato de que houve um aumento de tempo médio gasto em GC para todos os programas analisados. Por isso foram realizadas análises mais detalhadas nos perfis de gerenciamento de memória de cada um dos programas: além da porcentagem de tempo gasto em *garbage collection*, foram analisados também o número de coletas e a quantidade média de memória em cada uma das gerações, bem como a distribuição dos diferentes tipos de objetos entre as mesmas. Tais análises não tinham como objetivo principal explicar a diferença obtida com relação a tempo de execução, mas sim avaliar o impacto causado em geral pela variação no uso de *delegates* sob o ponto de

| Programa | <i>Closures</i> como <i>delegates</i> | <i>Closures</i> separadas de <i>delegates</i> | Diferença |
|-----------------------|---------------------------------------|---|-----------|
| Digits_e1 (1500) | 18,68 | 16,45 | 11,94% |
| Digits_e2 (2000) | 13,31 | 12,81 | 3,79% |
| Exp3 | 32,38 | 40,06 | -23,7% |
| Primes (3500) | 13,49 | 12,76 | 5,42% |
| Queens | 24,85 | 23,09 | 7,09% |
| Tak | 22,47 | 22,35 | 0,53% |
| Wheel-sieve1 (200000) | 15,57 | 8,41 | 46,03% |
| Wheel-sieve2 (40000) | 6,45 | 7,11 | -10,19% |
| Wheel-sieve2 (80000) | 345,91 | 19,23 | 94,44% |

Tabela 5.5. Variações no Uso de *Delegates*

vista do consumo de memória. As tabelas 5.6 e 5.7 mostram os dados médios coletados para gerenciamento de memória com relação à quantidade de coletas.

| Programa | %Tempo em GC | Coletas Ger. 0 | Coletas Ger. 1 | Coletas Ger. 2 |
|-----------------------|--------------|----------------|----------------|----------------|
| Digits_e1 (1500) | 31,93 | 392 | 110 | 22 |
| Digits_e2 (2000) | 39,10 | 198 | 96 | 12 |
| Exp3 | 51,99 | 1028 | 507 | 3 |
| Primes (3500) | 31,60 | 72 | 44 | 1 |
| Queens | 10,10 | 233 | 74 | 0 |
| Tak | 0 | 0 | 0 | 0 |
| Wheel-sieve1 (200000) | 0,77 | 15 | 3 | 1 |
| Wheel-sieve2 (40000) | 50,96 | 60 | 36 | 1 |
| Wheel-sieve2 (80000) | 38,38 | 209 | 157 | 19 |

Tabela 5.6. Gerenciamento de Memória com *Closures* como *Delegates*

A partir dos dados coletados e do *profiling* de memória dos programas, os seguintes fatos foram observados:

- O menor tamanho das *closures* quando estas não são *delegates* mas apenas armazenam a referência para os mesmos pode fazer com que o número de coletas nas gerações 1 e 2 diminua. Esse fato pôde ser observado em programas como **Digits_e1**, **Exp3** e **Wheel-sieve2** e, com a ajuda de *profiling*, observou-se que as gerações 1 e/ou 2 são justamente compostas em sua maioria por *closures* já atualizadas. Na primeira versão do ambiente de execução, tais *closures* atualizadas eram *delegates*, enquanto na segunda versão elas passam a ser objetos comuns. Na nova versão, os *delegates* – que encapsulam o código a ser executado e ocupam 32 bytes por instância — são liberados para coleta logo após a atualização.
- Em todos os programas analisados, a percentagem de tempo gasto em *garbage collection* aumenta na nova versão. Isso era esperado devido à maior quantidade

| Programa | %Tempo em GC | Coletas Ger. 0 | Coletas Ger. 1 | Coletas Ger. 2 |
|-----------------------|--------------|----------------|----------------|----------------|
| Digits_e1 (1500) | 35,28 | 428 | 95 | 19 |
| Digits_e2 (2000) | 40,69 | 211 | 105 | 18 |
| Exp3 | 57,05 | 1134 | 550 | 2 |
| Primes (3500) | 44,46 | 92 | 47 | 1 |
| Queens | 15,38 | 461 | 92 | 0 |
| Tak | 0 | 0 | 0 | 0 |
| Wheel-sieve1 (200000) | 2,92 | 10 | 3 | 1 |
| Wheel-sieve2 (40000) | 53,28 | 77 | 33 | 1 |
| Wheel-sieve2 (80000) | 53,10 | 209 | 91 | 1 |

Tabela 5.7. Gerenciamento de Memória com a Separação *Closures–delegates*

de objetos a serem coletados. Um maior número de objetos possui como potenciais consequências o aumento do número de coletas e/ou do tempo para percorrer o grafo formado pelos objetos alocados no instante atual.

- Mais da metade dos programas apresentaram tamanhos médios das gerações menores após a separação *delegate–closure*. Entretanto, vale ressaltar que esses são valores médios coletados pelo *profiler* a partir de amostras aleatórias.

Finalmente, a Tabela 5.8 mostra a relação entre os tamanhos (em *bytes*) de alguns tipos de *closures* nas duas versões do ambiente de execução. Pode-se notar que quando não implementamos *closures* diretamente como *delegates* obtemos um ganho constante de 20 *bytes* no tamanho das mesmas. No novo modelo, os *delegates*, que apenas apontam para o código a ser executado e que são referenciados pelas *closures* como campos, sempre ocupam 32 *bytes*.

| Tipo de <i>closure</i> | <i>Closures</i> como <i>delegates</i> | <i>Closures</i> separadas de <i>delegates</i> |
|---------------------------------------|---------------------------------------|---|
| Atualizável, sem variáveis livres | 36 | 16 |
| Atualizável, com 1 variável livre | 40 | 20 |
| Atualizável, com 2 variáveis livres | 44 | 24 |
| Atualizável, com 3 variáveis livres | 48 | 28 |
| Atualizável, com 4 variáveis livres | 52 | 32 |
| Atualizável, com 6 variáveis livres | 60 | 40 |
| Não atualizável, sem variáveis livres | 40 | 20 |

Tabela 5.8. Tamanho das *Closures*

5.4.1 Conclusão

Vimos que a implementação do novo ambiente de execução trouxe vantagens e desvantagens de desempenho, melhorando os tempos de execução da maioria dos programas

analisados, apesar do maior consumo de memória e, algumas vezes, piorando os tempos daqueles que apresentam perfis críticos e possivelmente atípicos de alocação. Apesar da análise não ter sido feita em cima de uma grande amostra de programas reais, os resultados obtidos nos levam a concluir inicialmente que, ao contrário do que dizia a expectativa inicial, o maior número de objetos gerados com a nova abordagem nem sempre é um fator crítico. Ao contrário, o *overhead* gerado pode até ser compensado por fatores internos à implementação da máquina virtual ou mesmo ligados ao perfil de consumo de memória.

Na versão final do compilador, adotaremos a segunda opção de ambiente de execução. Em particular, a separação entre *closures* e *delegates* apresentada nesta seção possui pelo menos duas grandes vantagens:

- a implementação resultante está em conformidade com o padrão da CLI, sendo dessa forma portátil entre diferentes implementações;
- o código do ambiente de execução pode ser escrito, mantido e estendido em uma linguagem de alto nível como C#, uma vez que não há mais a necessidade de definir classes que herdam de `MulticastDelegate`, o que é proibido por tal linguagem devido às restrições impostas pela especificação da CLI.

Pelas razões acima, os testes mostrados nas próximas seções serão realizados com a segunda versão do ambiente de execução. É importante ressaltar que as medições apresentadas nesta seção incluíram as principais otimizações a serem mostradas (seções 5.6.3 e 5.6.4).

5.5 ANÁLISE DO IMPACTO DE “CONSTANT APPLICATIVE FORMS”

Constant Applicative Forms (ou “CAFs”) são constantes globais sem argumentos que podem ocupar grandes quantidades de memória à medida que são atualizadas. Um exemplo típico é uma lista infinita avaliada sob demanda. Nesta seção, resumiremos por meio de um exemplo o impacto que esse tipo de valor pode causar no desempenho de um programa funcional na nossa implementação. Para tal, mostraremos o ganho obtido com a remoção de uma CAF do programa `Primes`. Para este programa, modificamos seu código de modo a remover a constante global. Além disso, desabilitamos uma transformação implementada pelo GHC – o “*floating outwards*”. A transformação de “*floating outwards*” [dMS95] é um tipo de otimização aplicada em programas funcionais, cujo objetivo é aumentar o escopo de alguns tipos de construções locais para permitir que outras otimizações possam ser aplicadas. Uma das conseqüências desse tipo de transformação é o risco de gerar *closures* desnecessárias e até algumas CAFs a partir de construções locais presentes em expressões `let` e `case`.

A modificação do programa somada à desabilitação da transformação acima citada removeram a CAF, gerando uma diminuição no tempo de execução de cerca de 37%, confirmando o grande impacto negativo das CAFs na nossa implementação. Apesar disso, os demais programas analisados não apresentaram diferenças significativas com a remoção da transformação de *floating outwards*: um deles (`wheel-sieve1`) chegou a apresentar uma piora de cerca de 34%, o que demonstra que simplesmente desabilitar essa transformação

não chega a ser uma possível solução para amenizar o problema das CAFs. Deixamos portanto a transformação habilitada por padrão. Futuramente, podemos implementar mecanismos capazes de detectar se o *floating outwards* deve ou não ser realizado.

A solução para o problema das CAFs ainda está em aberto. Hunt e Perry decidiram pela não atualização de *closures* globais em seu compilador Haskell [HP05], uma vez que essa atualização fica a cargo da implementação. A mesma técnica foi testada no desenvolvimento do nosso compilador, porém o resultado foi um enorme tempo de execução para um programa em particular – *wheel-sieve2*. Este programa faz uso constante de CAFs à medida em que elas são atualizadas, incorrendo em retrabalho quando a atualização das mesmas não ocorre. Portanto a simples omissão da atualização de *closures* globais não é a solução para o problema.

Apesar da árvore sintática do programa STG disponibilizar estaticamente a informação relativa a quais CAFs são atingíveis por um determinado trecho de código, seria necessário estender o *garbage collector* da CLR para utilizar esse tipo de informação, de forma análoga ao que é feito pelo GHC [MJ98].

5.6 OTIMIZAÇÕES

Nas seções anteriores nos concentramos em avaliar o impacto de determinadas construções (*tail-calls*, *casts*) e comparar variações no uso de uma técnica de implementação (*delegates*). O objetivo era analisar duas ou mais opções e escolher aquela que obtivesse o melhor desempenho ou, alternativamente, a melhor relação entre desempenho e benefício.

Nas próximas seções analisaremos otimizações que foram feitas no gerador de código e/ou no ambiente de execução. É importante ressaltar que a maioria das otimizações aqui descritas teve como objetivo melhorar o consumo de memória, uma vez que este demonstrou ser o fator mais crítico ao desempenho de programas funcionais não estritos em uma plataforma gerenciada como a CLR.

5.6.1 Remoção do “Frame” de Atualização

No Capítulo 4, vimos a necessidade do “*frame* de atualização” – um objeto que armazena os tamanhos das pilhas de argumentos e que é necessário ao funcionamento do modelo push/enter. Utilizávamos portanto uma pilha desses objetos, criadas a cada vez que uma *closure* era atualizada.

O objetivo final do *frame* de atualização nada mais é do que salvar os argumentos presentes nas pilhas, uma vez que estas serão “esvaziadas”. Na implementação adotada, isso equivalia a guardar os tamanhos das mesmas (e não os seus conteúdos!) e zerar seus tamanhos para então atualizar a *closure*. Ao fim da atualização, os tamanhos das pilhas eram restaurados a partir do *frame* removido do topo da pilha de *frames* de atualização. Entretanto, o objetivo do procedimento acima pode ser obtido simplesmente salvando-se os tamanhos das pilhas em variáveis locais e restaurando-os após a atualização, sem a criação de objetos adicionais e sem a necessidade de uma pilha para guardar tal tipo objeto. Dessa forma, substituímos os *frames* de atualização pelo seguinte trecho de código no método **Enter** das *closures* atualizáveis:

```

int intStackSize = RuntimeSystem.intStackSize;
int doubleStackSize = ...
RuntimeSystem.intStackSize = 0;
RuntimeSystem.doubleStackSize = 0;...
value = //...atualização...//
RuntimeSystem.intStackSize = intStackSize;
RuntimeSystem.doubleStackSize = doubleStackSize;
...

```

Dessa forma, o próprio *frame* de execução da CLR correspondente ao método **Enter** é utilizado para atingir nosso objetivo. A Tabela 5.9 mostra o resultado da otimização acima para alguns programas³.

| Programa | Antes da otimização | Após a otimização | Diferença |
|------------------|---------------------|-------------------|-----------|
| Digits_e1 (1000) | 12,43 | 11,99 | 3,58% |
| Digits_e2 (1500) | 11,68 | 10,31 | 11,79% |
| Exp3 (8) | 3,97 | 2,77 | 30,17% |
| Primes (3000) | 13,33 | 12,73 | 4,53% |
| Queens (12) | 9,30 | 8,98 | 3,53% |

Tabela 5.9. Remoção do “*Frame*” de Atualização

5.6.2 Simulando Atualização “In-place”

Ao descrever o mecanismo de atualização de *thunks* na *Spineless Tagless Machine*, Simon Peyton Jones introduziu a técnica da “atualização *in-place*” [Jon92], que consiste em sobrescrever a memória alocada pelo *thunk* com o seu novo valor. Quando isso não fosse possível devido ao tamanho do valor ser maior do que o tamanho da closure (*thunk*), seria introduzida uma indireção para o mesmo.

A motivação para a atualização *in-place* estava não apenas em remover uma indireção, mas, principalmente, liberar a memória alocada por *closures* já atualizadas.

Na CLR, bem como em outros ambiente de código gerenciado em que o principal foco está na segurança de tipos, não é possível sobrescrever um objeto com outro, de forma que esse tipo de otimização não é possível de ser empregado diretamente. A princípio, a seguinte alternativa foi considerada para simular o efeito desejado da atualização “*in-place*”: adicionar ao método **Enter** um argumento passado por referência (tipo *&*) representando o endereço da referência para a *closure* a ser atualizada. Ao final da computação, a referência apontada pelo argumento recebido poderia ser atualizada, passando a referenciar o valor recém-computado. Isso seria possível uma vez que tal valor poderia ser de dois

³A remoção dos *frames* de atualização foi uma das primeiras otimizações implementadas, quando ainda tínhamos menos programas sendo cobertos pelas bibliotecas do GHC. Por esse motivo não mostramos os resultados para o conjunto total de programas mostrados em outras seções.

tipos: aplicação parcial ou um objeto avaliado (*datatype*). Ambos os tipos de objetos herdam da classe `Closure`, tornando possível a atribuição.

Entretanto, a abordagem acima apresentou como principal desvantagem o fato de não ser mais possível invocar o método `Enter` por meio de uma *tail-call* através de código verificável. Tal limitação justifica-se pela natureza segura da CLR: em uma situação em que a referência para uma variável local, por exemplo, fosse passada por referência por meio de uma *tail-call*, os objetos referenciados pelo *frame* descartado seriam coletados e, dessa forma, a única referência existente para o objeto em questão tornar-se-ia inválida. Por esse motivo, descartamos tal possibilidade e sugerimos como uma possível extensão à CLR alguma forma de interação com o *garbage collector* que torne possível “dispará-lo” para coletar *closures* já atualizadas e substituir as referências a elas por seus novos valores. Tal extensão seria bem específica a linguagens funcionais e possivelmente envolveria algum suporte nativo a *closures*. Caberia também avaliar se o *overhead* gerado por uma extensão como essa seria de fato compensado por ganhos significativos de desempenho. Infelizmente ainda não temos respostas para tais questões.

O próprio GHC abandonou a idéia da atualização “*in-place*” [MJ98, JMR99]. Naquele contexto, além de aumentar a complexidade do compilador, esse tipo de atualização poderia levar a múltiplas cópias de um mesmo objeto. Dessa forma, indireções são sempre utilizadas, evitando o problema das múltiplas cópias. Por outro lado, vale ressaltar que o GHC tem maior controle sobre o *layout* e tamanho das *closures* atualizadas, sendo capaz de manipulá-los para liberar memória, ao contrário do que ocorre com o Haskell.NET na CLR (pelo menos a partir de código verificável).

Finalmente, propomos uma técnica que, apesar de não resolver, visa a atenuar o problema da coleta de *closures* já atualizadas, coletando esse tipo de objeto quando possível. A técnica aqui proposta consistiu nos seguintes passos:

- i) foi adicionado um método denominado `GetValue` à classe `Closure`. Para *closures* atualizáveis, o que esse método faz é verificar se a *closure* já foi atualizada: caso tenha sido, seu valor é retornado e, caso contrário, a própria *closure* é retornada (a avaliação propriamente dita **não** é forçada);
- ii) sempre que um campo “*boxed*” de um objeto do tipo `Pack` (argumento) ou *closure* (variável livre) é acessado, seu método `GetValue` é invocado e o valor retornado é atribuído ao campo em questão;
- iii) para a construção de objetos, é invocado o método `GetValue` para os valores que servirão de campos para o objeto sendo construído.

Aplicando o que está descrito acima, evita-se construir novos objetos que referenciem *closures* já atualizadas e, em algumas situações, atualiza-se as referências para *closures* atualizadas contidas em objetos previamente criados.

Os resultados desta otimização podem ser vistos na Tabela 5.10.

Analisando-se os algoritmos de alguns programas que obtiveram pouco ganho ou perda de desempenho, observou-se que esses não poderiam tirar proveito da atualização *in-place* simulada, uma vez que a atualização de *closures* referenciadas por outros objetos ocorreria

| Programa | Antes da otimização | Após a otimização | Diferença |
|-----------------------|---------------------|-------------------|-----------|
| Digits_e1 (1500) | 16,36 | 16,09 | 1,66% |
| Digits_e2 (2000) | 12,85 | 13,14 | -2,27% |
| Exp3 | 40,11 | 42,89 | -6,93% |
| Primes (3500) | 12,89 | 12,61 | 2,21% |
| Queens | 23,15 | 28,44 | -22,85% |
| Tak | 22,35 | 22,36 | -0,06% |
| Wheel-sieve1 (200000) | 8,45 | 6,49 | 23,28% |
| Wheel-sieve2 (40000) | 7,29 | 7,41 | -1,69% |

Tabela 5.10. Simulação de Atualização “*In-place*”

após a criação dos últimos, e a atualização das referências segundo o passo ii) não poderia ser feita durante a execução do código dos programas, e sim por um processo externo aos mesmos, possivelmente acoplado ao *garbage collector*. Já para o programa *Wheel-sieve1*, que apresentou o melhor ganho, foi observado, por meio de *profiling* de memória, que o tempo de vida das *closures* atualizáveis diminuiu, de forma que, ao fim da execução, esse tipo de *closure* ocupa o segundo lugar em espaço na *heap* (33,6%), contra o primeiro lugar (39,44%) sem o emprego da otimização. Em termos de quantidade de memória isso representou uma diferença de cerca de 1,2 MB para a entrada analisada.

Foi notado, portanto, que, quando a otimização em questão não atinge seus objetivos, as chamadas ao método `GetValue` – um método virtual – podem degradar o desempenho do programa. Pelo fato da simulação de atualização *in-place* não ter alcançado sua finalidade para a maioria dos programas, devido à dinâmica de execução dos mesmos, ela encontra-se desativada na versão padrão do gerador de código.

5.6.3 Compartilhamento de Booleanos

Denominamos de construtores nulários aqueles que não possuem argumentos, como no exemplo abaixo:

```
data DiaUtil :: Segunda | Terca | Quarta | Quinta | Sexta
```

O GHC implementa sempre que possível o compartilhamento de construtores nulários, por meio da definição de constantes que representam tais construtores. Entretanto há casos em que, apesar da existência dessas constantes, o compartilhamento não é feito. É o caso de valores booleanos, definidos pela biblioteca padrão da linguagem da seguinte forma:

```
data Bool :: False | True
```

Valores booleanos são sempre *boxed*, portanto.

Em geral, o resultado da avaliação de expressões booleanas não é conhecido em tempo de compilação, fazendo com que constantes não sejam utilizadas e sempre novas instâncias

sejam criadas para representar os valores obtidos. No compilador Haskell.NET, implementamos a seguinte otimização para evitar a alocação de instâncias de booleanos:

- após cada chamada a um operador booleano, é gerado o seguinte código, em IL:

```
brfalse L1
ldsfld Pack RuntimeSystem.TRUE
br L2
L1:
ldsfld Pack RuntimeSystem.FALSE
L2:
//próximas instruções
```

O código acima testa se o valor no topo da pilha é falso (0): se for, é retornado na pilha o objeto que representa a constante FALSE, armazenado como um campo estático da classe `RuntimeSystem`. Caso contrário, o objeto que representa a constante TRUE é retornado.

- Tratamento semelhante é feito para a criação de booleanos através dos construtores `True/False`. Tal situação não é otimizada pelo GHC, restando ao nosso gerador de código verificar estaticamente qual construtor está sendo chamado e gerar apenas a instrução `ldsfld` correspondente.

O ganho em velocidade de execução obtido com essa otimização varia de 13% a 46% segundo os testes realizados. O resultado para os tempos de execução é mostrado na Tabela 5.11. Pode-se notar que, apesar de booleanos serem objetos que quase sempre são coletados ainda na geração 0, é significativo o ganho que se obtém evitando-se a alocação de memória para os mesmos e, conseqüentemente o *overhead* devido à sua coleta.

| Programa | Sem a otimização | Com a otimização | Diferença |
|-----------------------|------------------|------------------|-----------|
| Digits_e1 (1500) | 18,78 | 16,31 | 13,13% |
| Digits_e2 (2000) | 15,93 | 12,81 | 19,56% |
| Exp3 | 40,02 | 40,06 | -0,09% |
| Primes (3500) | 18,63 | 12,84 | 31,08% |
| Queens | 38,80 | 23,12 | 40,42% |
| Tak | 41,60 | 22,32 | 46,34% |
| Wheel-sieve1 (200000) | 11,95 | 8,42 | 29,56% |
| Wheel-sieve2 (40000) | 9,94 | 7,11 | 28,52% |

Tabela 5.11. Compartilhamento de Valores Booleanos

Não investigamos em detalhes em quais situações o GHC gera (e de fato utiliza) constantes para construtores nulários. Ao invés disso nos concentramos em otimizar o tipo provavelmente mais comum de construtor nulário e para o qual sabemos que não existe otimização semelhante feita pelo *front-end* do GHC.

5.6.4 Compartilhamento de Valores Inteiros

Além do compartilhamento de construtores nulos, uma outra técnica utilizada para diminuir a quantidade de memória alocada é a pré-alocação de um conjunto fixo de constantes numéricas, em geral utilizadas para representar números inteiros e caracteres *boxed*. Encontramos a utilização dessa abordagem no GHC e no Bigloo para JVM [SS02], um compilador da linguagem funcional estrita Scheme [Sch]. O primeiro pré-aloca todos o conjunto de caracteres ASCII e todos os inteiros entre -16 e 16. Já o segundo pré-aloca 256 caracteres e os inteiros no intervalo de -100 a 2048.

Para o Haskell.NET, optamos por utilizar um conjunto de 256 caracteres e 256 inteiros (no intervalo de 0 a 256). Não temos comprovação estatística de que esse é um conjunto razoável, porém os valores mínimo e máximo podem ser ajustados futuramente a partir de dados de amostras maiores de programas e de avaliações de desempenho.

Para implementar o compartilhamento de inteiros, utilizamos dois *arrays*, sendo um para objetos do tipo `Pack<int>` e outro para objetos do tipo `Pack<char>`. Sempre que há necessidade de utilizar um inteiro ou caractere “boxed”, invocamos um método do ambiente de execução – `MakeInt` ou `MakeChar`, respectivamente. Tais métodos são responsáveis por verificar se o inteiro desejado pertence ao intervalo de constantes pré-alocadas. Em caso afirmativo, o valor desejado é consultado a partir do *array* correspondente. Caso contrário, uma nova instância é criada. Poderíamos ter utilizado, ao invés de estruturas de dados indexadas, instruções do tipo `switch` para testar o valor desejado. Porém, a menos que implementássemos um gerador de código, tal abordagem comprometeria a manutenibilidade do código do ambiente de execução, uma vez que o mesmo deveria ser alterado sempre que se decidisse modificar a faixa de constantes pré-alocadas. Por outro lado, da forma como essa otimização se encontra implementada, é possível tornar a faixa de constantes configurável para fins de testes futuros. Após termos resultados mais concretos a respeito de qual intervalo de constantes utilizar, poderemos otimizar a implementação atual para remover os *arrays* envolvidos, possivelmente desenvolvendo um gerador de código para o ambiente de execução.

Os resultados preliminares são mostrados no gráfico da Tabela 5.12. 3 (três) dos programas analisados apresentam uma melhora em seus tempos de execução de até 18% com o compartilhamento de inteiros, enquanto 3 (três) apresentam ligeiras pioras, embora em percentagens pequenas (máxima de 5,5%). Por padrão, mantemos portanto a otimização habilitada e com os intervalos de valores acima descritos.

5.6.5 Remoção de Tail-calls do Ambiente de Execução

Na Seção 5.3, analisamos o impacto de *tail-calls* no desempenho dos programas testados. Utilizamos a versão anterior do ambiente de execução, em que *closures* são implementadas diretamente como *delegates*. Nesta seção, analisaremos uma otimização em particular: a remoção de *tail-calls* **apenas** do ambiente de execução, e não do código compilado, que continua fazendo uso do prefixo `.tail` sempre que possível. Com isso, verificamos que *tail-calls* no código do ambiente de execução causaram um impacto negativo médio de aproximadamente 6%. Os resultados são mostrados na Tabela 5.13.

Finalmente, refizemos os testes realizados na Seção 5.3, de modo que agora *tail-calls*

| Programa | Sem a otimização | Com a otimização | Diferença |
|-----------------------|------------------|------------------|-----------|
| Digits_e1 (1500) | 19,94 | 16,34 | 18,08% |
| Digits_e2 (2000) | 13,59 | 12,82 | 5,72% |
| Exp3 | 38,01 | 40,1 | -5,50% |
| Primes (3500) | 12,8 | 12,83 | -0,23% |
| Queens | 27,84 | 23,16 | 16,83% |
| Tak | 22,35 | 22,36 | -0,04% |
| Wheel-sieve1 (200000) | 8,43 | 8,43 | 0% |
| Wheel-sieve2 (40000) | 7,11 | 7,11 | 0% |

Tabela 5.12. Compartilhamento de Valores Inteiros

| Programa | Com <i>tail-calls</i> | Sem <i>tail-calls</i> | Impacto das <i>tail-calls</i> |
|-----------------------|-----------------------|-----------------------|-------------------------------|
| Digits_e1 (1500) | 16,30 | 15,99 | -1,95% |
| Digits_e2 (2000) | 12,84 | 12,51 | -2,52% |
| Exp3 | 40,11 | 39,03 | -2,70% |
| Primes (3500) | 12,83 | 12,41 | -3,25% |
| Queens | 23,14 | 22,86 | -1,22% |
| Tak | 22,34 | 22,29 | -0,21% |
| Wheel-sieve1 (200000) | 8,43 | 5,69 | -32,55% |
| Wheel-sieve2 (40000) | 7,11 | 7,01 | -1,50% |

Tabela 5.13. Impacto do Uso de *Tail-calls* no Ambiente de Execução

são utilizadas apenas no código compilado. Esse teste é importante para avaliarmos o impacto “real” das *tail-calls*, descartando sua utilização no ambiente de execução. Com isso, obtivemos diferentes resultados, conforme esperado. Desta vez, existe uma clara vantagem em favor da utilização de *tail-calls*, devido ao perfil de alocação de memória de programas funcionais não estritos. Os novos resultados são resumidos na Tabela 5.14. Apenas Tak, um programa pequeno que praticamente não realiza nenhuma alocação significativa de memória e cujo objetivo é somente avaliar o desempenho de recursão e de *tail-calls*, demonstrou sofrer um impacto negativo com a utilização do prefixo `.tail` nas chamadas.

Concluimos a partir dos novos dados obtidos que a aparente perda de desempenho mostrada na Seção 5.3 devia-se ao uso de *tail-calls* não no código compilado, mas sim no código do ambiente de execução. Este último nada mais é do que um componente escrito em C#, de forma imperativa, sofrendo dessa forma o *overhead* devido a *tail-calls* na CLR para esse tipo de programa.

Tail-calls não são implementadas de forma tão eficiente como em geradores de código nativo, como o gerador do GHC. Entretanto, a conclusão é que elas, mesmo da forma como estão implementadas atualmente na CLR, trazem um impacto positivo à *performance* de programas Haskell, devido ao perfil de consumo de memória dos mesmos. Como regra

| Programa | Com <i>tail-calls</i> | Sem <i>tail-calls</i> | Impacto das <i>tail-calls</i> |
|-----------------------|-----------------------|-----------------------|-------------------------------|
| Digits_e1 | 29,44 | 33,58 | 12,33% |
| Digits_e2 (2000) | 12,62 | 13,39 | 5,69% |
| Exp3 | 39,41 | 41,59 | 5,25% |
| Primes | 21,14 | 22,52 | 6,14% |
| Queens | 22,92 | 36,88 | 51,11% |
| Tak | 22,37 | 17,37 | -28,78% |
| Wheel-sieve1 (200000) | 5,69 | Estouro da pilha | - |
| Wheel-sieve2 (40000) | 7,08 | 8,82 | 19,79% |

Tabela 5.14. Impacto do Uso de *Tail-calls* no Código Compilado

geral, a ausência de *tail-calls* pode trazer consequências desastrosas, desde a degradação significativa de desempenho – tanto do ponto de vista de tempo como de memória – até o estouro da pilha de chamadas.

5.6.6 Substituição de Tail-calls por Instruções de Desvio

Uma vez que *tail-calls* apresentam alguns problemas de desempenho na implementação adotada pela CLR, considerou-se a sua substituição por uma instrução de desvio incondicional em chamadas recursivas (“recursão na cauda”). Dessa forma, quando a função invocada for a própria função corrente, os argumentos são sobrescritos com os novos valores e a execução é transferida para a primeira linha da função, por meio da instrução `br`. Essa técnica não é nova, tendo sido utilizada, por exemplo, em implementações de SML para .NET e JVM [BKR04, BKR98].

Uma outra alternativa, utilizada na implementação de Nemerle [MOS], seria o uso da instrução `jmp`, que desvia o controle para a primeira instrução de um método qualquer, desde que este tenha a mesma assinatura do método atual. O *frame* de execução é descartado, resultando em um comportamento análogo ao obtido com *tail-calls*. Porém, o emprego dessa técnica apresenta uma desvantagem: a instrução `jmp` não é verificável.

A Tabela 5.15 mostra os resultados obtidos com a substituição da *tail-call* da CLR pela instrução de desvio em chamadas recursivas.

Ao observarmos os resultados mostrados na Tabela 5.15, vimos que os programas que obtiveram maior ganho de desempenho são justamente aqueles que, conforme veremos na Seção 5.7, apresentam menor valor para a métrica *%Tempo em GC*. Esse fato demonstra que, quando outras variáveis, como alto consumo de memória, não estão envolvidas, o *overhead* de desempenho gerado pela *tail-call* da CLR tende a ser maior e, conseqüentemente, uma otimização que diminua sua utilização leva a ganhos consideráveis.

5.7 HASKELL.NET X GHC NATIVO

A Tabela 5.16 mostra a comparação entres os tempos médios de execução para os programas estudados quando compilados pelo gerador de código nativo do GHC e pelo gerador de código IL desenvolvido neste trabalho. A configuração do compilador avaliada

| Programa | Sem a otimização | Com a otimização | Diferença |
|--------------|------------------|------------------|-----------|
| Digits_e1 | 29,46 | 29,30 | 0,54% |
| Digits_e2 | 32,47 | 32,42 | 0,15% |
| Exp3 | 39,49 | 39,44 | 0,13% |
| Primes | 21,15 | 21,10 | 0,22% |
| Queens | 22,92 | 16,72 | 22,07% |
| Tak | 22,39 | 14,59 | 34,86% |
| Wheel-sieve1 | 15,01 | 11,62 | 22,55% |
| Wheel-sieve2 | 18,77 | 18,76 | 19,79% |

Tabela 5.15. Impacto da Substituição de *Tail-call* por Instrução de Desvio

utiliza a segunda versão do ambiente de execução, com as otimizações mostradas neste capítulo – exceto simulação de atualização *in-place*.

Conforme pode ser visto na Tabela 5.16, obtivemos em média tempos da mesma ordem de magnitude que os obtidos pelo gerador de código nativo do GHC, o que é um resultado aceitável para a primeira versão do gerador de IL. A razão média entre os tempos de execução no Haskell.NET e no GHC nativo é de aproximadamente 4 vezes⁴, chegando a pouco mais de 2 vezes quando não levamos em consideração os piores casos.

| Programa | GHC Nativo | Haskell.NET | Razão Haskell.NET/GHC Nativo |
|--------------|------------|-------------|------------------------------|
| Digits_e1 | 6,29 | 29,30 | 4,66 |
| Digits_e2 | 6,69 | 32,42 | 4,85 |
| Exp3 | 2,81 | 39,44 | 14,06 |
| Primes | 0,95 | 21,10 | 22,13 |
| Queens | 10,21 | 16,72 | 1,64 |
| Tak | 13,22 | 14,59 | 1,10 |
| Wheel-sieve1 | 6,78 | 11,62 | 1,71 |
| Wheel-sieve2 | 5,91 | 18,76 | 3,18 |
| Média | | | 4,03 |

Tabela 5.16. Haskell.NET vs. GHC Nativo

Apenas dois programas chegaram à marca de uma ordem de magnitude de tempo de execução em relação ao GHC nativo: **Exp3** e **Primes**.

Foi realizado o *profiling* de memória e análise dos algoritmos de ambos os programas. Também foi investigado se havia alguma falha no gerador de IL capaz de gerar estouros de memória, o que não foi detectado. O *profiling* de tempo também não apontou nenhum padrão expressivo. A começar por **Exp3**, este possui um perfil de consumo de memória em que um excessivo número de alocações é feito, chegando a uma taxa em que cada

⁴Utilizamos aqui a média geométrica, a mais indicada para valores normalizados segundo Fleming e Wallace [FW86]. O uso da média geométrica para valores normalizados é criticado em um trabalho posterior [Smi88]. Porém, não encontramos outra métrica mais adequada.

closure coletada pelo *garbage collector* é substituída por uma ou mais *closures* alocada(s) em seguida. Já **Primes**, que implementa o Crivo de Erastóstenes, possui dois fatores responsáveis pelo mau desempenho em relação ao GHC nativo: a presença de uma CAF e o fato de que, ao longo de sua execução, que envolve o percurso por uma lista infinita, elementos referenciados pela “cabeça” da lista continuam a ser referenciados por sua “cauda”, eliminando a possibilidade de um grande número de *closures* serem coletadas.

Os fatores apontados como explicação para o mau desempenho de **Exp3** e **Primes** demonstram o quanto o mecanismo de gerenciamento de memória difere entre o GHC nativo e a CLR. O primeiro foi desenvolvido tendo por alvo uma linguagem funcional não estrita, sendo capaz de coletar CAFs, por exemplo, enquanto o segundo é mais genérico e suficiente para linguagens imperativas ou orientadas a objeto.

Os programas analisados apresentaram, em média, um valor de 35% para a métrica *%Tempo em GC*, que idealmente deveria assumir valores médios de no máximo 5% [MVBM04]. É importante ressaltar que esse valor ideal foi escolhido tendo em mente aplicações escritas em linguagens imperativas/orientadas a objeto, como C#, por exemplo. Foi observado que os programas que obtiveram melhores tempos de execução apresentaram os menores valores para essa métrica, a saber **Queens** (15,19%), **Tak** (0%) e **Wheel-sieve1** (2,81%). Analogamente, valores altos, próximos a 50%, freqüentemente ocorreram para os programas com pior desempenho em relação ao GHC nativo.

Existem incompatibilidades entre as ferramentas de *profiling* de memória do GHC nativo e da CLR que dificultam comparações entre os dois ambientes: no primeiro, é gerado um gráfico de memória do tipo *quantidade de bytes alocados X tempo*, enquanto o *CLR Profiler* gera um gráfico *endereço de memória X tempo*. Para piorar, a métrica *%Tempo em GC* não é compatível com a de nome correspondente encontrada no GHC, uma vez que reflete sempre a percentagem de tempo gasta desde o último ciclo de coleta e não no programa inteiro, como seria mais natural.

5.8 CONSIDERAÇÕES FINAIS

Conforme pôde ser notado pelas análises e otimizações mostradas neste capítulo, os principais obstáculos à implementação eficiente de linguagens funcionais não estritas na CLR estão no campo do gerenciamento de memória. Podemos citar como exemplos:

- a presença de CAFs e a dificuldade para gerenciar sua coleta, devido à falta de possibilidade de interação com o *garbage collector*;
- a ausência de atualização *in-place*, e a dificuldade de implementá-la ou mesmo simulá-la devido ao mesmo motivo citado no item anterior.

Em geral, a segurança de acesso à memória imposta pela CLR e pela necessidade de gerar código verificável é um fator que, embora aumente a confiabilidade do código gerado, dificulta o gerenciamento de memória em programas funcionais não estritos.

Devido à criticidade observada pelo fator memória nos programas analisados, as principais otimizações implementadas pelo gerador de IL desenvolvido foram voltadas para essa área.

Vale ressaltar que a primeira versão do ambiente de execução desenvolvido continha uma falha: a memória alocada para as variáveis livres referenciadas pelas *closures* atualizáveis não era liberada após a atualização. A consequência deste fato era que *closures* atualizadas e que portanto não mais precisavam conter variáveis livres levavam a uma situação de “vazamento de memória” (“*space leak*”) quando eram referenciadas por alguma outra estrutura de dados (ex.: uma lista infinita). A simples correção da falha em questão levou a um ganho em tempos de execução de até 87%! Tal resultado demonstra o quanto pequenas e aparentemente inofensivas omissões com relação a liberação de memória podem levar a tempos de execução proibitivos. Entretanto falhas desse tipo podem ocorrer facilmente durante o desenvolvimento de um compilador e/ou de um ambiente de execução, consistindo em um fator crítico ao desempenho de programas funcionais. Todos os resultados mostrados ao longo deste capítulo já consideram a correção dessa falha.

Ao contrário do que poderia levar a crer a expectativa inicial, foi verificado que *casts* em geral não causam impacto significativo que motive a geração de código não verificável por padrão. Versões futuras do compilador podem ainda empregar otimizações para reduzir o número de *casts*. De fato, o uso de Generics pode ser considerado uma otimização em relação a uma abordagem mais simples, como uso de *arrays* para representação de argumentos, como era feito nos primeiros protótipos do compilador.

Da mesma forma, *tail-calls*, tidas como causadoras de grande *overhead* em programas imperativos ou funcionais estritos, demonstraram ser necessárias e até mesmo benéficas do ponto de vista de desempenho para programas funcionais não estritos. Apenas a remoção das mesmas de um código orientado a objetos compartilhado por todos os programas – o ambiente de execução – foi suficiente para que o *overhead* a elas associado fosse consideravelmente diminuído.

Uma modificação no uso de *delegates* que aparentemente poderia trazer grande piora de *performance* mostrou em vários casos superior ao obtido com a utilização anterior, além de apresentar como vantagens tornar o compilador compatível com o padrão da CLI e muito mais fácil de manter.

O desempenho médio do código gerado pelo compilador Haskell.NET é da mesma ordem de magnitude que o correspondente obtido pelo gerador de código nativo do GHC para a maioria dos programas analisados. Estudos futuros acerca de estratégias de compilação mais eficientes ou mesmo extensões à máquina virtual da CLR, uma vez que esta não é otimizada para programas funcionais não estritos, podem corrigir os resultados de maior ordem de magnitude.

INTEROPERABILIDADE EM LINGUAGENS FUNCIONAIS

A necessidade de interoperar programas escritos em diferentes linguagens é uma necessidade bastante antiga, tendo surgido quando ainda não se imaginava a criação de uma plataforma multi-linguagem. Interfaces de programação capazes de invocar código nativo para acesso a rotinas do sistema operacional ou mesmo qualquer código escrito em linguagem C surgiram como uma funcionalidade básica para a sobrevivência de qualquer linguagem. Tais interfaces são responsáveis basicamente por fazer a tradução entre tipos (*marshalling*), por invocar procedimentos/funções/métodos na convenção de chamada suportada pela linguagem externa e por permitir que a linguagem externa invoque código na linguagem de origem. Alguns exemplos de tais interfaces são a *Foreign Function Interface* (FFI) [CFH⁺02], um padrão para Haskell que pode ser implementado para permitir a comunicação entre esta linguagem e qualquer outra linguagem, e a *JAVA Native Interface* (JNI) [Gor99], voltada para a linguagem JAVA.

Mecanismos mais sofisticados, capazes de fazer a comunicação entre qualquer par de linguagens, foram desenvolvidos para atender às necessidades da programação baseada em *componentes*, onde um componente pode ser definido, a grosso modo, como qualquer *software* que expõe um conjunto de funcionalidades ou serviços por meio de uma interface. Surgiram modelos de componentes como o *Common Object Model* (COM) [Rog97] e, mais recentemente, os *Web Services*. Ambos possuem em comum o fato de basearem-se em uma linguagem única responsável por expor a interface de cada componente - *Interface Definition Language* (IDL), no primeiro, e *Extended Markup Language* - XML [XML], nos últimos. Portanto, para que dois componentes codificados em diferentes linguagens possam comunicar-se, basta que se implemente, para cada linguagem, um mapeamento dos elementos de interface da mesma - assinaturas de funções/métodos e tipos de dados - para os elementos correspondentes na linguagem intermediária. De fato, a Plataforma .NET consiste em uma evolução do modelo COM em vários aspectos, descritos detalhadamente por Box e Sells [BS03].

Nas próximas seções, apresentaremos os principais trabalhos na área de interoperabilidade de linguagens funcionais (estritas ou não) com outras linguagens, não apenas no contexto da Plataforma .NET. Começaremos por analisar as principais interfaces de comunicação entre a linguagem Haskell e outras linguagens/modelos de programação como C, COM e JAVA (Seção 6.1). Em seguida, descreveremos como a questão da interoperabilidade é resolvida em outras linguagens funcionais implementadas na Plataforma .NET (Seções 6.2 - 6.4).

6.1 INTEROPERABILIDADE EM HASKELL

Nesta seção, citaremos trabalhos existentes no campo da interoperabilidade de Haskell com outras linguagens, concentrando-nos na versão padronizada da linguagem e sem tratar de questões relacionadas à extensão da mesma.

6.1.1 Foreign Function Interface (FFI)

Foreign Function Interface (FFI) é uma especificação que tem por objetivo padronizar tanto a forma pela qual código escrito em outra linguagem pode ser invocado a partir de Haskell, como também a maneira de tornar código Haskell acessível a outras linguagens. A padronização fez-se necessária para evitar que diferentes implementações de Haskell suportassem interoperabilidade com outras linguagens de formas incompatíveis. Fazem parte dos objetivos desse padrão permitir que a interface para funcionalidade externa seja escrita em código fonte Haskell, por meio de *declarações* em sintaxe específica, de uma forma independente do sistema utilizado (compilador, interpretador, etc.) e da linguagem externa.

Na Figura 6.1, temos a definição da sintaxe básica de uma declaração FFI.

| | | |
|-----------------|---|--|
| <i>topdecl</i> | → | foreign <i>fdecl</i> |
| <i>fdecl</i> | → | import <i>callconv</i> [<i>safety</i>] <i>impent</i> <i>var</i> :: <i>fctype</i> export <i>callconv</i> <i>expent</i> <i>var</i> :: <i>fctype</i> |
| <i>callconv</i> | → | ccall stdcall cplusplus jvm dotnet ... |
| <i>impent</i> | → | [<i>string</i>] |
| <i>expent</i> | → | [<i>string</i>] |
| <i>safety</i> | → | unsafe safe |

Figura 6.1. Sintaxe das Declarações FFI

Na sintaxe da Figura 6.1, *callconv* representa a convenção de chamada. O código gerado para implementar uma determinada convenção de chamada é dependente da implementação. Aqui, o que queremos dizer por “implementação” é a instanciação da FFI para fazer a comunicação de Haskell com determinada linguagem ou ambiente. As *strings* que representam os não terminais *impent* e *expent* devem conter as informações necessárias para identificar o elemento externo a ser invocado, no caso de importações, e a forma como um elemento Haskell será visto externamente, em exportações. A sintaxe de tais *strings* é dependente da linguagem de programação externa, pois em linguagens como C, por exemplo, apenas um identificador é necessário, enquanto em linguagens como IL, informações de tipos são usadas para resolver possíveis conflitos de nomes. A informação relacionada a segurança (*safety*) indica se a função importada poderá desencadear uma execução de *garbage collection* (através da chamada a alguma função Haskell), tornando indisponível um objeto Haskell passado como argumento para a mesma. Em caso afirmativo, deverá ser especificado o valor **safe**. Caso contrário, pode-se utilizar o valor **unsafe**. *var* é o identificador a ser utilizado em Haskell para referenciar uma função externa ou

para identificar a função Haskell que será exportada.

Um outro ponto dependente da linguagem externa é a tradução dos tipos de dados entre os dois ambientes (*marshalling*).

O conjunto de tipos dos valores que podem ser passados como argumentos para código externo ou código Haskell exportado restringe-se aos seguintes tipos, referidos aqui como *tipos traduzíveis*:

- primitivos (ex.: inteiros, números em ponto flutuante, caracteres, ponteiros), em geral suportados por linguagens da mesma família de C;
- sinônimos ou tipos de dados compostos a partir de tipos traduzíveis.

As mesmas restrições aplicam-se aos tipos dos valores retornados por funções, externas ou exportadas por Haskell.

Com relação ao mecanismo de avaliação de argumentos em código externo, em geral as implementações da FFI assumem que é estrito. Além disso, para código que possa gerar efeitos colaterais, recomenda-se o mapeamento do código externo em funções monádicas (ie.: cujo tipo de retorno seja `IO t`, sendo `t` qualquer tipo traduzível).

A especificação FFI está restrita à sintaxe básica utilizada para a importação de código externo e exportação de código Haskell, sem detalhes que envolvam verificação da correspondência entre os tipos dos parâmetros ou mesmo verificação da existência do elemento externo (função, procedimento, etc.). Cabe portanto ao programador o uso correto dos mecanismos de importação/exportação oferecidos. Outros aspectos que não são cobertos pela especificação são aqueles que se referem a definir qual é o tipo mais indicado para representar um elemento externo. Esse é o papel de outras ferramentas, responsáveis por gerar automaticamente declarações de importação/exportação. Finalmente, a interação com recursos multi-tarefa também não é especificada.

Um aspecto que não é coberto pela FFI é a importação/exportação direta de tipos (por meio de uma declaração da forma “`foreign type`”, por exemplo). Além disso, não existe sintaxe específica para importar variáveis ou constantes externas, de forma que uma declaração que não receba argumentos é representada da mesma forma que um mapeamento para uma função que não receba argumentos. Entretanto, tal limitação pode ser resolvida por meio de algum tipo de indicação nas *strings* referentes aos não terminais *impent* e *expent*.

6.1.2 H/Direct

A interoperabilidade entre linguagens obtida por meio de declarações FFI é uma atividade tediosa e sujeita a erros, surgindo portanto a necessidade de ferramentas que automatizem o processo de geração de interfaces de comunicação com outras linguagens. Um exemplo é H/Direct [FLMJ98], projetada a partir da experiência adquirida com o desenvolvimento de ferramentas anteriores complexas e/ou incompletas.

H/Direct caracteriza-se pela representação das assinaturas dos métodos ou funções externos em IDL, o padrão de descrição de interfaces utilizado em COM. Até mesmo funções escritas em linguagem C, sem o uso da tecnologia COM, podem ser acessadas,

desde que sua interface seja escrita em IDL. O uso dessa linguagem tem como vantagem o fato de cobrir limitações presentes tanto em Haskell como em outras linguagens de programação para representar certos detalhes (convenção de chamada, passagem por valor, referência ou valor/resultado, etc.). H/Direct também permite gerar componentes COM a partir de código escrito em Haskell, que podem portanto ser utilizados por programas escritos em qualquer linguagem.

Além das declarações FFI geradas pela ferramenta, que fazem o mapeamento dos tipos presentes nas assinaturas dos elementos externos para tipos Haskell de baixo nível (ex.: ponteiros), são gerados *wrappers* que expõem assinaturas com tipos de mais alto nível para o programador (ex.: registros, tipos abstratos, etc.). Tais *wrappers* fazem a transformação de tipos de baixo nível para tipos de alto nível e vice-versa. A transformação entre tipos primitivos não é necessária, enquanto a tradução de tipos estruturados envolve alocação de memória e cópia de valores Haskell para a memória alocada, na representação adequada. O funcionamento de H/Direct independe da implementação da FFI utilizada.

Do ponto de vista da usabilidade, um ponto fraco de H/Direct é a obrigatoriedade da definição da interface IDL por parte do programador. No caso do ambiente .NET, por exemplo, em que os módulos expõem metadados com basicamente o mesmo tipo de informação presente nessas interfaces, poderia ser desenvolvida uma ferramenta capaz de gerar a interface IDL. Entretanto, o uso de tal ferramenta só adicionaria mais um passo ao processo, tornando-o mais lento e suscetível a erros.

6.1.3 Lambada

Lambada [MF00] é uma ferramenta voltada para o ambiente JAVA, que permite interoperabilidade em ambas as direções: a chamada de métodos JAVA a partir de Haskell e a invocação de funções Haskell a partir de JAVA.

A forma pela qual o código JAVA é acessado se dá através de *JAVA Native Interface* (JNI), uma tecnologia de baixo nível que permite inicializar e criar instâncias da máquina virtual, bem como instanciar objetos e invocar métodos JAVA. A JVM é acessada por meio de uma interface COM. Em particular, Lambada é projetada para ser o mais independente possível da implementação local da JVM, permitindo que a informação responsável por identificar tal implementação seja parametrizada.

Em geral, a utilização da ferramenta é feita através de rotinas de biblioteca responsáveis por funções como criação de objetos, chamadas de métodos, etc. Tais rotinas, escritas em Haskell, acessam código JNI por meio de FFI.

Classes JAVA acessam código Haskell por meio do carregamento de uma DLL que contenha a implementação do código desejado ou que possa registrar dinamicamente funções Haskell como métodos “nativos” junto ao ambiente JAVA. No primeiro caso, pode ser utilizado o componente DietHEP, presente nos ambientes de execução do GHC e do Hugs, que permite que qualquer módulo Haskell possa ser visto como uma DLL comum, com isso impondo mais um nível de indireção às chamadas.

6.1.4 Hugs .NET

O Hugs .NET [Hugb] é a versão do interpretador Hugs [Huga] estendida para permitir acesso a código .NET a partir de Haskell e vice-versa, funcionando como uma ponte ou *bridge*. Segundo o autor, permite interoperabilidade completa, suportando invocação de métodos estáticos ou de instância, criação de objetos, acesso a campos de objetos e uso de *delegates*, que podem inclusive referenciar funções implementadas em Haskell. Esta última função entretanto é implementada por meio de um complexo mecanismo de geração dinâmica de código e de classes, em que o código gerado não é necessariamente verificável.

O acesso a código Haskell é feito através de invocação de código não gerenciado, bem como acesso a memória não gerenciada, o que implica a existência de um *garbage collector* no lado Haskell, além daquele que já é fornecido pela CLR. Além disso, existe o *overhead* associado à transição de código gerenciado para não gerenciado.

O Hugs .NET possui ferramentas para geração de *wrappers* a partir de módulos .NET, com declarações FFI que permitem acessar o código externo.

É possível ainda gerar classes cujos métodos sejam todos implementados por funções Haskell. Vale ressaltar que não foi feita nenhuma extensão à linguagem para suportar nem essa nem outras funcionalidades.

Foi feita uma tentativa de incorporar o suporte a interoperabilidade fornecido pelo Hugs .NET ao GHC, porém tal implementação não chegou a ser testada. A descrição da implementação do Hugs .NET que será dada abaixo aplica-se, quando possível, à implementação existente no GHC.

O Hugs .NET, conforme citado no Capítulo 3, acessa o modelo de objetos de .NET por meio de API de Reflexão. Para o GHC, essa API é vista como um componente COM. Portanto, chamadas a código .NET acarretam no custo associado ao uso de COM e de reflexão, além da tradução (*marshalling*) entre tipos. O mecanismo de retorno de condições de erro se dá através de *strings*, não fazendo uso dos recursos de tratamento de exceções presentes tanto em Haskell como em .NET.

Uma última limitação do Hugs .NET é que não é possível passar objetos Haskell para .NET. Além disso, para que uma outra linguagem tenha acesso a funções Haskell, é necessário utilizar uma espécie de “servidor” para localizar a função, não sendo possível passar argumentos.

6.2 INTEROPERABILIDADE EM F#

F# foi projetada especialmente para a Plataforma .NET e assume características multi-paradigma, combinando primitivas funcionais e imperativas/orientadas a objeto. Pelo fato de ter sido desenvolvida especialmente para esta plataforma, tendo em mente portanto os requisitos de interoperabilidade completa com outras linguagens, já provê suporte sintático e semântico a todas as construções presentes no mundo .NET, como criação de objetos, acesso a métodos/propriedades/campos/eventos, conversões de tipos (*upcast* e *downcast*), etc. Dessa forma, a implementação de tal nível de interoperabilidade com .NET em F# não apresenta grandes desafios.

A criação de objetos, por exemplo, é feita por meio do uso de uma palavra reservada especial – “**new**” – como ocorre normalmente em linguagens orientadas a objeto. O acesso a métodos, campos e propriedades, da mesma forma, utiliza notação análoga à encontrada em várias linguagens, por meio do operador “.” (ponto).

Do ponto de vista inverso, F# suporta acesso a tipos nela definidos, como uniões discriminadas, por exemplo, uma vez que estas são compiladas para classes e subclasses. Além disso, é possível invocar funções de alta ordem definidas em F# passando como parâmetro funções implementadas em C#, por exemplo. Porém, isso é feito através do uso de determinadas conversões que fazem uso de *delegates*.

6.3 INTEROPERABILIDADE EM SML .NET

SML .NET é mais um exemplo de linguagem funcional implementada para .NET que objetiva interoperabilidade completa com a plataforma, em ambas as direções. Para tal, diversas extensões foram feitas à linguagem para suportar características de .NET que não possuem equivalentes óbvios em SML. Exemplos dessas extensões são:

- coerções: foi introduzida sintaxe própria para coerções – explícitas quando objetos .NET são passados para funções SML e implícitas apenas quando se invocam métodos .NET;
- sintaxe especial para acesso a métodos e campos de instância: foi adicionada a notação “.#”;
- enumerações;
- definição de novas classes, interfaces, *delegates* e atributos.

Algumas funcionalidades, no entanto, são mapeadas em construções já existentes na linguagem, como:

- classes e espaços de nomes (“*namespaces*”): modelados por meio do sistema de módulos de SML;
- *arrays* unidimensionais: suportados por SML, são tratados de forma equivalente ao tratamento fornecido pela CLR;
- **null**: literal utilizado para indicar referências não inicializadas na CLR, é mapeado na construção *option* da linguagem;
- criação de objetos: objetos são construídos por meio do mapeamento do nome da classe para o construtor da mesma, não tendo sido adicionada nenhuma palavra reservada para tal funcionalidade.

Algumas extensões mais complexas não foram implementadas, como a regra da chamada ao método “mais específico”, usada para resolver certos tipos de sobrecarga de

métodos em C# e JAVA. Tal extensão, segundo os implementadores, levaria a dificuldades na inferência de tipos.

Para que outras linguagens possam acessar construções implementadas em SML .NET, é fornecido um mecanismo de exportação dessas construções. Tal mecanismo faz uso de uma ferramenta capaz de gerar código para interoperabilidade (“*wrapper*”) que é acessível por outras linguagens. Tal código é responsável por redirecionar, por exemplo, chamadas a uma função SML .NET ao código que a implementa. O conjunto de tipos que podem ser exportados nas assinaturas desse código no entanto é restrito.

6.4 INTEROPERABILIDADE EM MONDRIAN

Mondrian é um outro exemplo de linguagem funcional projetada especialmente para interoperar com plataformas orientadas a objeto, suportando execução na JVM e em .NET. Seu objetivo é funcionar como uma linguagem de *script* para aplicações Internet.

Por ter no seu projeto original os requisitos de interoperabilidade com plataformas que suportam orientação a objeto, possui sintaxe própria para criação de objetos, invocação de métodos, acesso a propriedades, por meio das palavras reservadas “create”, “invoke”, “set”/“get”, respectivamente.

Para a representação de parâmetros de métodos de .NET, utiliza tuplas, não suportando porém a utilização das mesmas para outras finalidades. Para a representação de referências, utiliza mônadas [Mog89] para entrada e saída, representando as mesmas como sendo do tipo `IO Object`. Não permite, portanto, que se indique o tipo mais específico do objeto referenciado.

Podem ser citados outros pontos fracos da interoperabilidade fornecida por Mondrian:

- não há qualquer verificação semântica em relação a chamadas a métodos/ construtores/campos, sendo gerada uma mensagem de erro pouco sugestiva em tempo de execução para uma chamada a um método inexistente, por exemplo;
- para criar instâncias de uma classe, é utilizado um longo caminho baseado na API de Reflexão do ambiente .NET, que em geral leva a um tempo de execução superior ao atingido com chamadas normais, diretamente codificadas em IL. O mesmo vale para a invocação de métodos e acesso a campos.

6.5 CONSIDERAÇÕES FINAIS

Até o momento analisamos a interoperabilidade em linguagens funcionais sob o ponto de vista das tecnologias relacionadas à comunicação. Vimos que, com o objetivo de interoperabilidade com o ambiente .NET, novas linguagens foram definidas, enquanto extensões foram feitas a linguagens estritas já existentes.

No caso específico da linguagem Haskell, o problema da comunicação propriamente dita, referente a convenções de chamada e à tradução de tipos já se encontra resolvido, com a existência de um padrão como a FFI.

Por outro lado, o maior desafio para a interoperabilidade entre Haskell e linguagens orientadas a objeto da Plataforma .NET é o chamado “*gap* semântico” que existe entre

os dois ambientes. Haskell não suporta herança nem sobrecarga de nomes de funções, construções praticamente pervasivas no paradigma de orientação a objeto.

Voltaremos agora a atenção para as técnicas mais comumente utilizadas para representar em linguagens funcionais elementos típicos de orientação a objetos, tais como hierarquias de classes (herança) e sobrecarga “*ad-hoc*”. Inicialmente, analisaremos duas abordagens para a representação de herança: *phantom types* e o uso de *classes de tipo*.

Phantom Types A técnica do uso de *phantom types* – ou “tipos imaginários” – é a mais comum para representar subtipos em Haskell, sendo utilizada por Lambada, Hugs .NET e alguns outros trabalhos voltados para o modelo COM [LMH99, FLMJ99, JML98]. Consiste em definir, para cada classe, um novo tipo Haskell (“*datatype*”) e um sinônimo de tipo. O primeiro possui um parâmetro de tipo que não é utilizado em sua definição, sendo este parâmetro portanto chamado de “imaginário”. Abaixo, temos um exemplo dessa codificação:

```
data C_ t = C_; type C t = C_ t
```

onde *C* é o nome da classe que desejamos representar. Qualquer subclasse *S* de *C* pode ser representada como:

```
data S_ t = S_; type S_ = C (S_)
```

À classe *C* propriamente dita deve ser atribuído o tipo *C ()*.

Os problemas desta abordagem são a complexidade resultante para a representação de hierarquias de classe longas – o que pode dar origem a erros de tipos difíceis de serem encontrados – e a impossibilidade de representar herança por interfaces ou qualquer outra forma de herança múltipla.

Classes de Tipo Introduzidas com o objetivo de permitir uma forma de sobrecarga de funções em Haskell, *classes de tipo* [WB89] podem ser utilizadas para a representação de hierarquias de classes. Esta técnica é utilizada por Lambada para implementar herança por interfaces e por trabalhos mais recentes na área de interoperabilidade com ambientes orientados a objeto [SJ01, PC03]. Consiste em termos um tipo Haskell e uma classe de tipo por classe OO (mais uma vez, aqui denominada de “*C*”), de modo que a cada subclasse “*S*” da classe OO corresponde uma *instância* da classe de tipo, como mostra o exemplo abaixo:

```
data C; class SubC t; instance SubC C;
data I; class SubI t;
data S; class SubC t => SubS t;
instance SubS D; instance SubC S; instance SubI S
```

onde *I* é uma interface OO implementada por *S*. Claramente, o uso de classes de tipo

não apresenta nenhuma limitação com relação a herança por interfaces, uma vez que uma declaração de instância pode ser dada à classe para cada interface que ela implementar.

O problema da sobrecarga (*overloading*), foi tratado por Shields e Peyton Jones [SJ01], que propuseram algumas extensões à linguagem, com regras formais para verificação e inferência de tipos. Tais extensões seriam necessárias uma vez que não é possível utilizar o suporte existente de Haskell para sobrecarga de nomes de funções sem anotações explícitas de tipos, aparentemente redundantes do ponto de vista do programador. Entretanto, as extensões propostas não foram provadas formalmente nem implementadas.

O trabalho posterior de Pang e Chackravarty [PC03] propõe soluções para a interface com linguagens orientadas a objeto sem estender a sintaxe ou sistema de tipos de Haskell. São usadas extensões previamente incorporadas ao GHC e ao Hugs, como classes de tipos multi-parametrizadas [JJM97] e dependências funcionais [Jon00]. Tais extensões permitem que se especifiquem classes que recebem mais de um parâmetro de tipo e que possam estabelecer relações funcionais entre tais parâmetros, respectivamente. Tais classes podem ser utilizadas para representar métodos e com isso implementar sobrecarga a partir do suporte já fornecido por Haskell. Entretanto, o trabalho de Pang e Chackravarty não cobre todas as questões levantadas por Shields e Peyton Jones. Seu objetivo final é definir uma biblioteca para implementar chamadas a *frameworks* orientados a objetos seguindo a metodologia básica de definição de uma *bridge* que faz uso de FFI e da geração automatizada de interfaces. Algumas contribuições em relação a trabalhos anteriores são a busca por interfaces mais tipadas e o uso de classes multi-parametrizadas para implementar relações binárias entre tipos (*upcasting* e *downcasting*).

CAPÍTULO 7

CONCLUSÕES E TRABALHOS FUTUROS

Este capítulo apresenta algumas conclusões acerca do trabalho realizado e está estruturado da seguinte forma: na Seção 7.1 apresentamos algumas considerações gerais e destacamos as principais contribuições e, na Seção 7.2, apontamos oportunidades futuras de pesquisa.

7.1 CONSIDERAÇÕES GERAIS E PRINCIPAIS CONTRIBUIÇÕES

Neste trabalho, apresentamos uma implementação da linguagem Haskell para a Plataforma .NET. A implementação desenvolvida poderia ser facilmente estendida para outra implementação da CLI, embora tenhamos focado na mais utilizada – a CLR, infraestrutura de execução do ambiente .NET.

Descrevemos a solução proposta por meio de dois componentes principais: o gerador de código IL e o ambiente de execução. O primeiro mostrou-se bastante abrangente, sendo capaz de compilar as principais construções da linguagem. Atualmente, temos uma parte considerável do prelúdio Haskell implementada (Seção 7.2.1). O suporte a primitivas de I/O, exceções e algumas operações com inteiros de precisão infinita foi desenvolvido em C#. Os esquemas de compilação foram apresentados, permitindo que geradores de código para linguagens com características semelhantes possam ser facilmente derivados.

O ambiente de execução pode ser considerado o núcleo da implementação, pois contém a API utilizada pelo gerador de código. Partimos inicialmente de um ambiente de execução que tinha o risco de não ser portátil para outras implementações da CLI, uma vez que não estava em conformidade com o padrão por ela especificado com relação ao uso de *delegates*. Entretanto, desenvolvemos um ambiente alternativo, que atinge um compromisso entre desempenho e portabilidade. Além disso, a nova versão é mais fácil de manter, podendo ser editada e compilada diretamente em código C#, por exemplo.

Devido ao projeto do ambiente de execução, que combina *delegates* com Generics, não precisamos recorrer à abordagem de geração de uma classe para cada *closure*. Este é o primeiro trabalho que combina esses dois recursos para a representação de *closures*. O desempenho de *delegates* foi bastante otimizado na versão 2.0 da CLR. De acordo com nossos testes, a chamada de código através de *delegates* atinge um desempenho similar a chamadas de métodos de interface. Generics, por sua vez, permite “generalizar” – como o próprio nome diz – tipos de variáveis livres, parâmetros de construtores, etc., atuando como um grande facilitador ao compartilhamento de classes.

Para a maioria dos programas analisados, obtivemos, em comparação com o *Glasgow Haskell Compiler*, tempos de execução da mesma ordem de magnitude. Para aqueles que não atingiram essa meta de desempenho, verificou-se um alto consumo de memória, o que ajuda a concluir que o gerenciamento de memória da CLR ainda é um fator crítico ao desempenho de programas funcionais não estritos.

Vários estudos de desempenho foram mostrados, incluindo análise do impacto de *casts*, *tail-calls*, comparações entre os dois ambientes de execução e avaliações do impacto de algumas otimizações. Os resultados desses testes são muito importantes, pois nos dão uma visão mais realista da atuação de uma série de fatores no ambiente da CLR, ajudando a responder a questões como:

- *Casts* geram *overhead* suficiente ao ponto de serem removidos? E *tail calls*?
- Determinada otimização melhorará de fato o desempenho? Em caso afirmativo, em quanto?

Como pôde ser visto, muitos dos testes deram respostas a essas questões que contrariam a “intuição” inicial.

Em resumo, podem ser citadas como importantes contribuições deste trabalho:

- O desenvolvimento de uma implementação Haskell para .NET, juntamente com a avaliação do seu desempenho sob vários aspectos;
- A implementação desenvolvida, uma vez que se encontra estável, serve como um ambiente de estudo e prototipação de estratégias de compilação alternativas;
- A base para a interoperabilidade com outras linguagens .NET. A simples utilização de código escrito em C# – ambiente de execução e algumas rotinas básicas do prelúdio (I/O, exceções, etc.) – já sugerem um certo grau de interoperabilidade, ainda que no momento não possa ser enxergado pelo desenvolvedor Haskell;
- A identificação de possíveis áreas em que a CLR pode ser estendida ou otimizada.

O objetivo original do projeto era incluir o suporte a linguagens funcionais na própria máquina virtual, estendendo a *Shared Source CLI* [SNS03], implementação de código aberto da CLI disponibilizada para pesquisa. Entretanto, até o momento não foi disponibilizada a implementação desse ambiente para a versão 2.0 da CLI, que inclui Generics e já é utilizada por um grande número de aplicações.

7.2 TRABALHOS FUTUROS

Nesta seção, serão apontadas possibilidades futuras de pesquisa, seja relacionadas à extensão do gerador de código atual ou mesmo a novas frentes ligadas à pesquisa de estratégias alternativas.

7.2.1 Suporte a Bibliotecas

Atualmente não são todos os módulos do prelúdio Haskell que são compilados pelo gerador de IL. Características que não são suportadas incluem suporte a números de ponto flutuante (inclui chamadas a funções C), *parsing* de *strings* (módulo “*Read*”), mônadas e suporte a todas as operações de I/O. Por outro lado, elementos necessários ao funcionamento de um grande número de programas, tais como operações envolvendo listas e

enumerações, operações numéricas (módulo “Num”) e de comparação, além de conversão para *string* (módulo “Show”), já se encontram compilados para a CLR. Em particular, esse subconjunto foi suficiente para o *benchmark* utilizado em nossos testes de desempenho.

Uma extensão necessária é compilar os módulos restantes e, adicionalmente, as bibliotecas do GHC. Para tal, é necessário resolver algumas questões como chamadas a código C por meio de declarações FFI presentes no código de alguns dos módulos do prelúdio. Uma solução para esse problema consiste em implementar a FFI para .NET, redirecionando essas chamadas para código equivalente às funções C acessadas, implementado em C# ou qualquer outra linguagem .NET.

Outras bibliotecas voltadas para aspectos mais avançados como concorrência e acesso a rede, por exemplo, podem ser tratadas através da “reescrita” de algumas de suas rotinas mais primitivas no ambiente .NET, utilizando o modelo de *threads* e de acesso a rede desse ambiente. Dessa forma, estaríamos fazendo uso de implementações de alto nível já disponíveis, sem que seja necessário interoperar com rotinas nativas do sistema operacional.

7.2.2 Integração com Phoenix

Phoenix [Pho] é um *framework* extensível para otimização e análise de código, que promete ser “a base para as tecnologias de compilação Microsoft”. Tem por objetivo ser uma solução completa para o desenvolvimento de compiladores para múltiplas plataformas de *hardware*, disponibilizando inclusive ferramentas de *profiling* e teste. Também pode ser visto como *backend* de compilação, capaz de compilar para código nativo ou código compilado “*just-in-time*”.

Como *backend* o Phoenix é baseado em uma representação intermediária (*IR*, de “*Internal Representation*”) sobre a qual são aplicadas otimizações diversas até a geração de código final. Dessa forma, qualquer linguagem pode utilizá-lo como *backend* desde que forneça um leitor (“*reader*”) que a traduza para IR. Alguns leitores são fornecidos juntamente com o *framework*, entre eles o leitor de CIL. A arquitetura básica de Phoenix é mostrada na Figura 7.1[Lef].

Um possível trabalho futuro seria estudar em detalhes este *framework*, analisar quais otimizações são implementadas e se de fato elas poderiam agregar melhor desempenho ao código gerado pelo compilador Haskell.NET. Em caso afirmativo, incorporar o leitor de CIL disponibilizado ao gerador de IL atualmente implementado, redirecionando dessa forma a tarefa de geração de código final para Phoenix. Adicionalmente, é possível que as ferramentas de *profiling* desse *framework* corrijam as limitações dos *profilers* utilizados neste trabalho.

7.2.3 Análises de Desempenho

O próximo passo em termos de avaliações de desempenho seria analisar *benchmarks* mais realistas, como os grupos Espectral e Real do Nofib. Com esses resultados, poderíamos identificar aspectos até então não revelados acerca da estratégia de compilação adotada. Para suportar os grupos Espectral e Real, seria necessário que cobríssemos maiores porções do prelúdio (Seção 7.2.1).

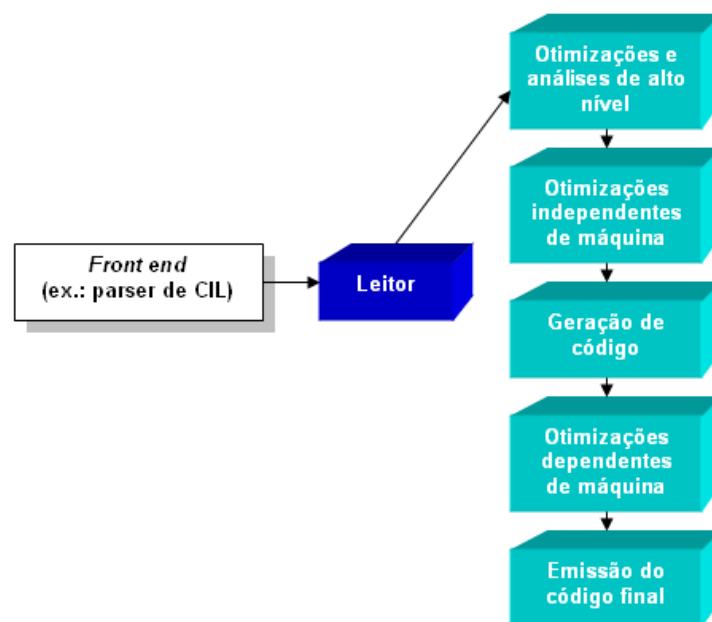


Figura 7.1. Phoenix – Arquitetura

Com relação à implementação, as possibilidades de pesquisa incluem desde variações no modelo atual de compilação até extensões à máquina virtual.

No primeiro caso, temos como exemplos:

- avaliação do desempenho de outras representações para *closures*;
- a implementação de um ambiente de execução baseado no modelo eval/apply, a fim de verificar se as limitações apontadas no Capítulo 3 são de fato um empecilho ao bom desempenho dos programas compilados. Para tal, será necessária também a incorporação de um processo de *lambda-lifting* capaz de atuar sobre a árvore STG. O GHC fornece *lambda-lifting* a partir de Core, embora este esteja desabilitado. Conforme já citado no Capítulo 4, a habilitação do *lambda-lifting* ao nível de Core, segundo o próprio time responsável pelo GHC, poderia comprometer outras otimizações importantes, sendo este o motivo pelo qual não o habilitamos.
- ajuste da transformação de *floating outwards*, de modo a evitar a geração de CAFs.

No segundo caso – extensões à máquina virtual – incluem-se pesquisas acerca de implementações alternativas para *closures* que considerem esses elementos como objetos *built-in*, tal qual ocorre com *delegates*. Adicionalmente, podem ser feitas extensões ao *garbage collector* de forma a incluir um suporte específico para esses objetos, na linha do que já foi citado no Capítulo 5, e a coleta de CAFs. Extensões à máquina virtual deverão ser implementadas e testadas na SSCLI [SNS03].

Um trabalho iniciado com o objetivo de implementar extensões à CLI para suporte a linguagens funcionais foi ILX [Sym01], um superconjunto de IL que incorpora a noção de

closures, implementando-as como elementos nativos da linguagem. Entretanto, nenhuma extensão foi de fato implementada na máquina virtual, de modo que, após a definição da linguagem, foi desenvolvido apenas um tradutor $ILX \rightarrow IL$. Atualmente, *closures* são traduzidas para classes.

7.2.4 Suporte a Interoperabilidade

Finalmente, em meio às diversas oportunidades de pesquisa que podem ser vislumbradas, o suporte a interoperabilidade com o ambiente .NET é o mais promissor no sentido de contribuir com a difusão da linguagem Haskell e conseqüentemente do paradigma funcional.

A extensão do Haskell.NET para interoperabilidade deve consistir de pelos menos três passos:

- suporte a FFI: implementar uma convenção de chamada da FFI que acesse código .NET. Neste passo, poderia ser utilizado como base o trabalho de Finne [Hugb], que definiu uma sintaxe para as declarações de importação. Essa sintaxe já se encontra suportada pelo GHC, sendo necessário apenas que se implementem as chamadas por meio de instruções IL. A partir daí, teremos um nível básico de interoperabilidade em que seja possível, por exemplo, invocar métodos estáticos;
- inclusão do suporte a orientação a objetos: este passo sem dúvida é o mais complexo, uma vez que exige extensões à linguagem semelhantes às descritas na Seção 6.5 do Capítulo 6. Essas extensões são necessárias para que a interoperabilidade seja o mais simples e natural possível do ponto de vista do desenvolvedor Haskell, e para que a linguagem seja classificada como *CLS Consumer*. Inicialmente, o trabalho de Shields e Peyton Jones [SJ01] seria um ponto de partida para as investigações. Esse tipo de extensão pode acarretar uma tendência multi-paradigma à linguagem Haskell;
- implementação de *wrappers* que exportem código Haskell, possibilitando ao mesmo ser invocado de forma natural a partir de outras linguagens .NET.

7.3 CONSIDERAÇÕES FINAIS

O Compilador Haskell.NET mostrou, por meio das análises de desempenho realizadas, que tem potencial para prover a interoperabilidade de Haskell com o ambiente .NET.

Em particular, o seu desenvolvimento contribuiu para melhorar o entendimento da área de implementação de linguagens funcionais para plataformas de código gerenciado, de suas dificuldades e desafios, especialmente se tratando de linguagens funcionais não estritas.

REFERÊNCIAS BIBLIOGRÁFICAS

- [AJ89] Lennart Augustsson e Thomas Johnsson. The Chalmers Lazy-ML Compiler. *The Computer Journal*, 32(2):127–141, 1989.
- [BKR98] Nick Benton, Andrew Kennedy, e George Russel. Compiling Standard ML to Java Bytecodes. Em *Proceedings of the 3rd ACM SIGPLAN Conference on Functional Programming*, Setembro 1998.
- [BKR04] Nick Benton, Andrew Kennedy, e Claudio V. Russo. Adventures in Interoperability: The SML.NET Experience. Em *6th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, páginas 215–226, Verona, Itália, 2004. ACM Press.
- [Bot98] Per Bothner. Kawa: Compiling Scheme to Java. Em *Lisp Users Conference*, Berkeley, California, EUA, November 1998.
- [BS03] Don Box e Chris Sells. *Essential .NET: Volume 1 - The Common Language Runtime*. Addison-Wesley Pearson Education, Boston, 2003.
- [BSS04] Yannis Bres, Bernard Paul Serpette, e Manuel Serrano. Compiling Scheme programs to .NET Common Intermediate Language. Em *.NET Technologies'2004 Workshop*, Plzen, 2004. ©UNION Agency-Science Press.
- [Cam] Caml. The Caml language. Site: <http://caml.inria.fr/>.
- [CCMW01] Erick Christensen, Francisco Curbera, Greg Meredith, e Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. Site: <http://www.w3.org/TR/wsd1.1>, Março 2001.
- [CFH⁺02] Manuel Chakravarty, Sigbjorn Finne, Fergus Henderson, Marcin Kowalczyk, Daan Leijen, Simon Marlow, Erik Meijer, Sven Panne, Alastair Reid, Malcolm Wallace, e Michael Weber. The Haskell 98 Foreign Function Interface 1.0: An Addendum to the Haskell 98 Report. Site: <http://www.cse.unsw.edu.au/~chak/haskell/ffi/>, 2002.
- [CiLH01] Kwanghoon Choi, Hyun il Lim, e Taisook Han. Compiling Lazy Functional Programs Based on the Spineless Tagless G-Machine for the Java Virtual Machine. Em *Functional and Logic Programming: 5th International Symposium, FLOPS 2001*, volume 2024 de *Lecture Notes in Computer Sciences*, página 78, Tóquio, Japão, 2001. Springer-Verlag GmbH.

- [CLI] CLI. Standard ECMA-335 – Common Language Infrastructure (CLI). Site: <http://www.ecma-international.org/publications/standards/Ecma-335.htm>.
- [COR] CORBA. Catalog of OMG CORBA®/IIOP® Specifications. Site: http://www.omg.org/technology/documents/corba_spec_catalog.htm.
- [Cyg] Cygwin. Cygwin Information and Installation. Site: <http://cygwin.com/>.
- [dMS95] André Luís de Medeiros Santos. *Compiling by Transformation in Non-Strict Functional Languages*. PhD thesis, University of Glasgow, 1995.
- [Dot] DotGNU. DotGNU Portable .NET. Site: <http://www.dotgnu.org/pnet.html>.
- [ECM] ECMA. ECMA International. Site: <http://www.ecma-international.org/>.
- [ECM05] ECMA. C# Language Specification, Junho 2005.
- [F#] F#. F#. Site: <http://research.microsoft.com/fsharp/fsharp.aspx>.
- [FLMJ98] Sigbjorn Finne, Daan Leijen, Erik Meijer, e Simon L. Peyton Jones. H/Direct: A Binary Foreign Language Interface for Haskell. Em *International Conference on Functional Programming*, páginas 153–162, 1998.
- [FLMJ99] Sigbjorn Finne, Daan Meijer Leijen, Eric Meijer, e Simon Peyton Jones. Calling Hell from Heaven and Heaven from Hell. Em *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, páginas 114–125, Paris, França, 1999.
- [FW86] Philip J. Fleming e John J. Wallace. How Not To Lie With Statistics: The Correct Way To Summarize Benchmark Results. *Communications of the ACM*, 29(3):218–221, 1986.
- [GHC] GHC. The Glasgow Haskell Compiler. Site: <http://www.haskell.org/ghc/>.
- [GJSB00] James Gosling, Bill Joy, Guy L. Steele, e Gilad Bracha. *The Java Language Specification*. Addison-Wesley Longman, 2nd edição, 2000.
- [Gor99] Rob Gordon. *Essential JNI: Java Native Interface*. Prentice Hall, 1999.
- [Gou01] K John Gough. Stacking them up: a comparison of virtual machines. Em *ACSAC '01: Proceedings of the 6th Australasian conference on Computer systems architecture*, páginas 55–61, Washington, DC, USA, 2001. IEEE Computer Society.

- [Gou02] John Gough. *Compiling for the .NET Common Language Runtime (CLR)*. .NET Series. Prentice Hall PTR, Upper Sadle River, Nova Jersey, EUA, 2002.
- [Has] Haskell. The Haskell 98 Language Report. Site: <http://www.haskell.org/onlinereport/>.
- [HP05] Oliver Hunt e Nigel Perry. Haskell.NET: The Art of Avoiding Work. Site: <http://research.microsoft.com/workshops/SSCLI2005/presentations/Perry-Hunt.ppt.>, 2005.
- [Huga] Hugs98. Hugs 98. Site: <http://www.haskell.org/hugs/>.
- [Hugb] Hugs98.NET. Hugs98 for .NET. Site: <http://galois.com/~{}sof/hugs98.net/>.
- [J#] J#. Microsoft Visual J# Developer Center. Site: <http://msdn.microsoft.com/library/default.asp?url=/nhp/default.asp?contentid=28000519>.
- [J2E] J2EE. Java Platform, Enterprise Edition (Java EE). Site: <http://java.sun.com/javaee/>.
- [J2M] J2ME. Java Platform, Micro Edition (Java ME). Site: <http://java.sun.com/javame/>.
- [JHH⁺93] Simon L. Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, e Phil Wadler. The Glasgow Haskell compiler: a technical overview. Em *Proceedings of the Joint Framework for Information Technology (JFIT) Technical Conference*, 93.
- [JJM97] Simon Peyton Jones, Mark Jones, e Erik Meijer. Type classes: exploring the design space. Em *Haskell Workshop*, 1997.
- [JML98] Simon Peyton Jones, Erik Meijer, e Daan Leijen. Scripting COM components in Haskell. Em *Proceedings of the Fifth International Conference on Software Reuse*. IEEE Computer Society, 1998.
- [JMR99] Simon Peyton Jones, Simon Marlow, e Alastair Reid. The STG Runtime System (revised). Site: <http://haskell.cs.yale.edu/ghc/docs/papers/run-time-system.ps.gz>, Fevereiro 1999.
- [Jon87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. International Series in Computer Science. Prentice Hall, Hertfordshire, Reino Unido, 1987.
- [Jon92] Simon L. Peyton Jones. Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-Machine. *Journal of Functional Programming*, 2(2):127–202, 1992.

- [Jon00] Mark P. Jones. Type Classes with Functional Dependencies. Em *Proceedings of the 9th European Symposium on Programming, ESOP 2000*, volume 1782, páginas 230–244, Londres, Reino Unido, 2000. Springer-Verlag.
- [KS01] Andrew Kennedy e Don Syme. Design and Implementation of Generics for .NET Common Language Runtime. Relatório técnico, Microsoft Research, Cambridge, Reino Unido, Maio 2001.
- [KS03a] Niels J. Kokholm e Peter Sestoft. *Moscow ML .Net Owner's Manual*, Novembro 2003.
- [KS03b] Niels J. Kokholm e Peter Sestoft. *Moscow ML .Net Internals*, Novembro 2003.
- [Lef] John Lefor. Phoenix Architecture. Site: http://research.microsoft.com/phoenix/FS_04_Lefor_v3.ppt#291,7,PhoenixArchitecture.
- [LMH99] Daan Leijen, Erik Meijer, e James Hook. Haskell as an Automation Controller. Em *International Summerschool on Advanced Functional Programming*, volume 1608, páginas 268–289, Braga, Portugal, 1999. Springer Lecture Notes in Computer Science.
- [LY99] Tim Lindholm e Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Longman, 2nd edição, 1999.
- [MABS05a] Monique Monteiro, Mauro Araújo, Rafael Borges, e André Santos. Compiling Non-Strict Functional Languages for the .NET Platform. Em *Proceedings of the 9th Brazilian Symposium on Programming Languages*, Recife, Brasil, 2005.
- [MABS05b] Monique Monteiro, Mauro Araújo, Rafael Borges, e André Santos. Compiling Non-strict Functional Languages for the .NET Platform. *Journal of Universal Computer Science*, 11(7):1255–1274, Julho 2005.
- [MF00] Erik Meijer e Sigbjorn Finne. Lambada: Haskell as a better Java. Em *Proceedings of the Haskell Workshop. Electronic Notes in Theoretical Computer Science*, volume 41, Montréal, Canadá, 2000.
- [MG00] Erik Meijer e John Gough. Technical Overview of the Common Language Runtime. Relatório técnico, Microsoft, 2000.
- [Mic] Microsoft. Microsoft Research. Site: <http://research.microsoft.com/>.
- [Mic05] Microsoft. Bug Details: .tail call is much slower than standard call instruction in IL. Site: <http://lab.msdn.microsoft.com/productfeedback/viewfeedback.aspx?feedbackid=58ece8e2-6914-4049-9d1a-2d040b705512>, Janeiro 2005.

- [MJ98] Simon Marlow e Simon Peyton Jones. The New GHC/Hugs Runtime System. Site: <http://haskell.cs.yale.edu/ghc/docs/papers/new-rt.ps.gz>, Agosto 1998.
- [MJ04] Simon Marlow e Simon L. Peyton Jones. Making a fast curry: Push/enter vs eval/apply for higher-order languages. Em *Proceedings of the International Conference on Functional Programming*, Snowbird, 2004.
- [Mog89] Eugenio Moggi. Computational Lambda-Calculus and Monads. Em *Proc. of 4th Ann. IEEE Symp. on Logic in Computer Science, LICS'89 (Pacific Grove, CA, USA)*, páginas 14–23. IEEE, Washington, DC, Junho 1989.
- [Mon] Mono. Mono. Site: http://www.mono-project.com/Main_Page.
- [MOS] Michal Moskal, Pawel W. Olszta, e Kamil Skalski. Nemerle: Introduction to a Functional .NET Language. Site: <http://nemerle.org/intro.pdf>.
- [MPvY01] Erik Meijer, Nigel Perry, e Arjan van Yzendoorn. Scripting .NET Using Mondrian. *Lecture Notes in Computer Science*, 2072:150–164, 2001.
- [MR04] James S. Miller e Susann Ragsdale. *The Common Language Infrastructure Annotated Standard*. Microsoft .NET Development Series. Addison-Wesley, Boston, EUA, 2004.
- [MTHM97] Robin Milner, Mads Tofte, Robert Harper, e David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.
- [MVBM04] J.D. Meier, Srinath Vasireddy, Ashish Babbar, e Alex Mackman. *Improving .NET Application Performance and Scalability*. Patterns and Practices. Microsoft, 2004.
- [.NEa] C++ .NET. Microsoft Visual C++ Developer Center. Site: <http://msdn.microsoft.com/library/default.asp?url=/nhp/default.asp?contentid=28000519>.
- [.NEb] Visual Basic .NET. Microsoft Visual Basic Developer Center. Site: <http://msdn.microsoft.com/library/default.asp?url=/nhp/default.asp?contentid=28000519>.
- [.NEc] Visual Studio .NET. Microsoft Visual Studio Developer Center. Site: <http://msdn.microsoft.com/vstudio/products/>.
- [NET] NET. .NET Framework Developer Center. Site: <http://msdn.microsoft.com/netframework/>.
- [OCa] OCaml. The OCaml project. Site: <http://www.pps.jussieu.fr/~montela/ocaml/index.html>.

- [Par93] Will Partain. The nofib Benchmark Suite of Haskell Programs. Em *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, páginas 195–202. Springer-Verlag, 1993.
- [PC03] André Pang e Manuel M. T. Chakravarty. Interfacing Haskell with Object-Oriented Languages. Em *Implementation of Functional Languages: 15th International Workshop, IFL 2003*, Edimburgo, Escócia, Reino Unido, 2003.
- [Pho] Phoenix. Phoenix Framework. Site: <http://research.microsoft.com/phoenix/>.
- [PM04] Nigel Perry e Erik Meijer. Implementing Functional Languages on Object-Oriented Virtual Machines. *IEEE Proceedings - Software*, 151(1):1–9, 2004.
- [Pro] CLR Profiler. CLR Profiler for the .NET Framework 2.0. Site: <http://www.microsoft.com/downloads/details.aspx?familyid=A362781C-3870-43BE-8926-862B40AA0CD0&displaylang=en>.
- [Rit] Brian Ritchie. .NET Languages. Site: <http://www.dotnetpowered.com/languages.aspx>.
- [RJB99] James Rumbaugh, Ivar Jacobson, e Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley Reading, Massachusetts, 1999.
- [Rog97] Dale Rogerson. *Inside COM*. Microsoft Press, 1997.
- [Sch] Scheme. The Scheme Programming Language. Site: <http://swiss.csail.mit.edu/projects/scheme/>.
- [SJ01] Mark Shields e Simon Peyton Jones. Object-Oriented Style Overloading for Haskell. Em *Workshop on Multi-Language Infrastructure and Interoperability (BABEL'01)*, Florença, Itália, 2001.
- [Smi88] James E. Smith. Characterizing computer performance with a single number. *Communications of the ACM*, 31(10):1202–1206, 1988.
- [SNS03] Dabid Stutz, Ted Neward, e Geoff Shilling. *Shared Source CLI Essentials: Exploring Microsoft's Rotor & the ECMA CLI*. O'Reilly. O'Reilly, 2003.
- [SOA] SOAP. SOAP Messaging Framework Specification. Site: <http://www.w3.org/TR/soap12-part1/#soapenvelope>.
- [SS02] Bernard Paul Serpette e Manuel Serrano. Compiling Scheme to JVM Bytecodes: A Performance Study. Em *Proc. ACM SIGPLAN International Conference on Functional Programming*, páginas 259–270, Nova Iorque, 2002. ACM, ACM Press.

- [Sym01] Don Syme. ILX: Extending the .NET Common IL for Functional Language Interoperability. *Electronic Notes in Theoretical Computer Science*, 59(1), 2001.
- [Tul96] Mark Tullsen. Compiling Haskell to Java. Relatório técnico, Department of Computer Science – Yale University, New Heaven, 1996.
- [UDD] UDDI. UDDI Version 3.0.2 Specification. Site: http://uddi.org/pubs/uddi_v3.htm.
- [W3C] W3C. Web Services. Site: <http://www.w3.org/2002/ws>.
- [Wak99] David Wakeling. Compiling lazy functional programs for the Java Virtual Machine. *Journal of Functional Programming*, 9(6):578–603, 1999.
- [Wat90] David A. Watt. *Programming Language Concepts and Paradigms*. International Series in Computer Science. Prentice Hall, 1990.
- [WB89] Philip Wadler e Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. Em *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, páginas 60–76, Austin, Texas., 1989.
- [Wie04] Nick Wienholdt. *Maximizing .NET Performance*. The Expert’s Voice. Apress, 2004.
- [XML] XML. Extensible Markup Language (XML). Site: <http://www.w3.org/XML/>.

APÊNDICE A

ESQUEMAS DE COMPILAÇÃO

O conjunto de regras de compilação utilizadas pelo compilador Haskell.NET é mostrado nas figuras A.1 a A.4. Embora a versão compilada seja de fato gerada em IL, aqui ela é mostrada em C# por simplicidade.

As figuras A.1 a A.3 mostram a compilação das construções STG, de modo que expressão a ser compilada é mostrada à esquerda, enquanto sua compilação é mostrada no lado direito. Já a Figura A.4 mostra a geração do *slow entry point*.

Os seguintes esquemas são utilizados nas regras de compilação:

- **B** (Figura A.1), responsável pela compilação de declarações;
- **I** (Figura A.1), responsável pela instanciação de closures;
- **FV** (Figura A.1), responsável pela inicialização de variáveis livres;
- **E** (Figura A.2), responsável pela compilação de expressões;
- **A** (Figura A.3), responsável pela compilação das alternativas das expressões `case`;
- **C** (Figura A.3), responsável pela compilação de expressões auxiliares;
- **T** (Figura A.3), responsável por retornar o tipo .NET correspondente a um tipo Haskell;
- **L** (Figura A.3), responsável por retornar o tipo “não avaliado” correspondente a um tipo Haskell. Aqui, o termo “tipo não avaliado” significa closure se o tipo Haskell não for primitivo, e o tipo .NET correspondente caso contrário;
- **S** (Figura A.4), responsável por gerar o *slow entry point* de uma dada closure, representada por suas variáveis livres e parâmetros;
- **SC** (Figura A.4), esquema auxiliar utilizado por **S**.

Outros símbolos são usados nos esquemas de compilação:

- τ é o tipo STG de um elemento;
- “SEP” é uma abreviação para “*slow entry point*” e $\ll id \gg_FEP$ significa “o *slow entry point* da função de nome $\ll id \gg$ ”. Em C# este é o nome do método, mas em IL um ponteiro para o método é de fato utilizado;
- “FEP” é uma abreviação para “*fast entry point*” e $\ll id \gg_SEP$ significa “o *fast entry point* da função de nome $\ll id \gg$ ”;

- “push” e “pop” são abreviações para os métodos responsáveis por empilhar/desempilhar objetos na pilha correspondente do ambiente de execução;
- Γ é o tamanho de uma pilha do ambiente de execução, sendo parametrizado pelo tipo Haskell. O mapeamento dos tipos Haskell nas pilhas correspondentes é feito conforme descrito na Seção 4.2.

Embora não seja explicitamente mostrado em todos os esquemas de compilação, a invocação de uma função é compilada sempre que possível para uma *tail-call*, ou para uma instrução de desvio, no caso de chamadas recursivas.

Por simplicidade, não incluímos as declarações dos métodos gerados, nem os tratamentos alternativos para closures, construtores ou objetos de aplicação parcial não suportados pelo ambiente de execução. Nesses casos, o tratamento realizado por meio de *arrays* é semelhante ao aqui mostrado, apenas incluindo as conversões necessárias.

| | | |
|---|-----|---|
| $\mathbf{B}[prog \rightarrow bind_1, \dots, bind_n]$ | $=$ | $\mathbf{B}[bind_1] \dots \mathbf{B}[bind_n]$ |
| $\mathbf{B}[id = f_1 \dots f_m \setminus n \ a_1 \dots a_n \rightarrow expr]$ | $=$ | NonUpdatable_<m>FV $\langle T[\tau(f_1)], \dots, T[\tau(f_m)] \rangle$ id = $\mathbf{I}[id = f_1 \dots f_m \setminus n \ a_1 \dots a_n \rightarrow expr];$ id.arity = <n>; |
| $\mathbf{B}[id = f_1 \dots f_m \setminus u \rightarrow expr]$ | $=$ | Updatable_<m>FV $\langle T[\tau(f_1)], \dots, T[\tau(f_m)] \rangle$ id = $\mathbf{I}[id = f_1 \dots f_m \setminus u \rightarrow expr];$ |

| | | |
|--|-----|--|
| $\mathbf{FV}[id = f_1 \dots f_m \setminus n \ a_1 \dots a_n \rightarrow expr]$ | $=$ | id.fv1 = $\mathbf{C}[f_1]; \dots; \text{id.fv} \langle m \rangle = \mathbf{C}[f_m];$ |
| $\mathbf{FV}[id = f_1 \dots f_m \setminus u \rightarrow expr]$ | $=$ | id.fv1 = $\mathbf{C}[f_1]; \dots; \text{id.fv} \langle m \rangle = \mathbf{C}[f_m];$ |

1a versão do amb. de execução:

| | | |
|---|-----|--|
| $\mathbf{I}[id = f_1 \dots f_m \setminus n \ a_1 \dots a_n \rightarrow expr]$ | $=$ | new NonUpdatable_<m>FV $\langle T[\tau(f_1)], \dots, T[\tau(f_m)] \rangle \langle id_SEP \rangle;$ |
| $\mathbf{I}[id = f_1 \dots f_m \setminus u \rightarrow expr]$ | $=$ | new Updatable_<m>FV $\langle T[\tau(f_1)], \dots, T[\tau(f_m)] \rangle \langle id_SEP \rangle;$ |

2a versão do amb. de execução:

| | | |
|---|-----|---|
| $\mathbf{I}[id = f_1 \dots f_m \setminus n \ a_1 \dots a_n \rightarrow expr]$ | $=$ | new NonUpdatable<sufixo> (new NonUpdCloFunction<sufixo><id_SEP>); |
| $\mathbf{I}[id = f_1 \dots f_m \setminus u \rightarrow expr]$ | $=$ | new Updatable<sufixo> (new UpdCloFunction<id_SEP>); onde <sufixo> = $_FV_ \langle m \rangle \langle T[\tau(f_1)], \dots, T[\tau(f_m)] \rangle$ |

Figura A.1. Esquemas de Compilação **B**, **FV** e **I**

O esquema **SC** pode gerar alocações desnecessárias para o caso $i = n$, uma vez que um objeto PAP pode ser instanciado mesmo que não haja argumentos na pilha. Ele pode

ser melhorado de forma a evitar gerar esse tipo de código em implementações futuras. No entanto, no *benchmark* analisado, não foi verificada a existência dessa situação, de modo que esta limitação não afetou os resultados aqui mostrados.

| | | |
|--|-----|--|
| $E[\text{let } bind \text{ in } expr]$ | $=$ | $B[bind]; FV[bind]; E[expr]$ |
| $E[\text{let } bind_1, \dots, bind_n \text{ in } expr]$ | $=$ | $B[bind_1]; FV[bind_1];$ $\dots;$ $B[bind_n]; FV[bind_n];$ $E[expr]$ |
| $E[\text{letrec } bind_1, \dots, bind_n \text{ in } expr]$ | $=$ | $B[bind_1]; \dots; B[bind_n];$ $FV[bind_1]; \dots; FV[bind_n];$ $E[expr]$ |
| $E[\text{case } expr \text{ of } alts]$ | $=$ | $T[\tau(expr)] \text{ scrutinee} =$ $(T[\tau(expr)]) E[expr];$ $A[alts]$ |
| $E[\text{var } atom_1 \dots atom_n, atom_{n+1}, \dots, atom_m]$ $m \geq 0,$ var não tem variáveis livres e tem aridade n . | $=$ | $\text{push } C[atom_m];$ $\dots;$ $\text{push } C[atom_{n+1}];$ $\text{var } (C[atom_1], \dots, C[atom_n]);$ |
| $E[\text{var } atom_1 \dots atom_n, atom_{n+1}, \dots, atom_m]$ $m \geq 0,$ var tem variáveis livres e aridade n . | $=$ | $\text{push } C[atom_m];$ $\dots;$ $\text{push } C[atom_{n+1}];$ $\text{var } (\text{var}, C[atom_1], \dots, C[atom_n]);$ |
| $E[\text{var } atom_1 \dots atom_m], var$ tem aridade $n,$ $m < n$. | $=$ | $\text{push } C[atom_m];$ $\dots;$ $\text{push } C[atom_1];$ $\text{var.Enter}();$ |
| $E[\text{var } atom_1 \dots atom_m],$ var tem aridade desconhecida ou igual a 0, $m \geq 0$. | $=$ | $\text{push } C[atom_m];$ $\dots;$ $\text{push } C[atom_1];$ $\text{var.Enter}();$ |
| $E[\text{constr}_i \text{ } atom_1 \dots atom_n]$ | $=$ | $\text{new Pack}_{<n>} <L[\tau(atom_1)], \dots, L[\tau(atom_n)]>$ $(i, C[atom_1], \dots, C[atom_n])$ |
| $E[\Theta \text{ } atom_1 \dots atom_n], \Theta$ é primitivo. | $=$ | $C[atom_1] \Theta \dots \Theta C[atom_n]$ |
| $E[n\#],$ onde $n\#$ é um literal de tipo primitivo (<i>unboxed</i>) | $=$ | n |

Figura A.2. Esquema de Compilação **E**

| | | |
|---|---|--|
| $\mathbf{A}[constr_1 \ arg_0 \dots arg_m \rightarrow expr_1;$ $\dots;$ $constr_n \ arg_0 \dots arg_p \rightarrow expr_n;$ $default]$ | = | $switch(scrutinee.tag)\{$ $\quad case \ constr_1's \ tag:$ $\quad \quad arg_1 = scrutinee.arg1;$ $\quad \quad \dots$ $\quad \quad arg_m = scrutinee.arg<m>;$ $\quad \quad \mathbf{E}[expr_1] \dots$ $\quad case \ constr_n's \ tag:$ $\quad \quad arg_0 = scrutinee.arg1;$ $\quad \quad \dots$ $\quad \quad arg_j = scrutinee.arg<p>;$ $\quad \quad \mathbf{E}[expr_n]$ $\quad \quad \mathbf{A}[default]$ $\quad \}$ |
| $\mathbf{A}[default \rightarrow expr]$ | = | $case \ default: \ \mathbf{E}[expr];$ |
| $\mathbf{A}[var \rightarrow expr]$ | = | $case \ default: \ \mathbf{C}[var] = scrutinee;$ $\quad \quad \mathbf{E}[expr];$ |
| $\mathbf{A}[literal_1 \rightarrow expr_1;$ $\dots;$ $literal_n \rightarrow expr_n;$ $default]$ | = | $switch(scrutinee)\{$ $\quad case \ \mathbf{E}[literal_1]: \ \mathbf{E}[expr_1] \dots$ $\quad case \ \mathbf{E}[literal_n]: \ \mathbf{E}[expr_n]$ $\quad \mathbf{A}[default] \}$ |

| | | |
|---|---|-----------------------|
| $\mathbf{C}[var]$ | = | var |
| $\mathbf{C}[literal]$ | = | $\mathbf{E}[literal]$ |
| $\mathbf{L}[int\#] = \mathbf{T}[int\#]$ | = | $int32$ |
| $\mathbf{L}[char\#] = \mathbf{T}[char\#]$ | = | $char$ |
| $\mathbf{L}[float\#] = \mathbf{T}[float\#]$ | = | $float32$ |
| $\mathbf{L}[t]$, t é um tipo <i>boxed</i> | = | $IClosure$ |
| $\mathbf{T}[t]$, onde t tem a forma $data \ X = \dots$ $\quad \quad constr_i \ t_1 \dots t_m$ $\quad \dots$ $\quad \quad constr_j \ u_1 \dots u_n$ $\quad \{t_1, \dots t_n\} \neq \{u_1, \dots u_n\}$ | = | $Pack$ |

Figura A.3. Esquemas de Compilação **A**, **C**, **L** e **T**

| | | |
|--|---|--|
| $\mathbf{S}[id = f_1 \dots f_m \setminus n \ a_1 \dots a_n]$ | = | <pre> PAP_<n<T[τ(a₁)],...,T[τ(a_m)]> pap; int stackSize; switch(closure.arity) { case 1: SC[id = f₁...f_m \ n a₁...a_n, 1]; ... case n: SC[id = f₁...f_m \ n a₁...a_n, n]; default: SC[id = f₁...f_m \ n a₁...a_n, 0]; } NonUpdatable_<m>FV <T[τ(f₁)],...,T[τ(f_m)]> newClosure = newClosure.arity = closure.arity - stackSize; newClosure.pap = pap; return newClosure; </pre> |
| $\mathbf{SC}[id = f_1 \dots f_m \setminus n \ a_1 \dots a_n, i],$ $i < n$ | = | <pre> pap = (PAP_n) closure.pap; stackSize = Γ(τ(a_{i+1}) + ... + Γ(τ(a_n)); if(stackSize >= i){ return <tail><id.FEP>(pap.arg1,..., pap.arg<n - i>, pop<τ(a_{n-i+1})>(),..., pop<τ(a_n)>()); } pap = (PAP_<n<T[τ(a₁)],...,T[τ(a_m)]>) pap.Clone(); break; </pre> |
| $\mathbf{SC}[id = f_1 \dots f_m \setminus n \ a_1 \dots a_n, i],$ $i = n$ | = | <pre> pap = (PAP_n) closure.pap; stackSize = Γ(τ(a_{i+1}) + ... + Γ(τ(a_n)); if(stackSize >= i){ return <tail><id.FEP>(pop<τ(a₁)>(),..., pop<τ(a_n)>()); } if(pap != null){ pap = (PAP_<n<T[τ(a₁)],...,T[τ(a_m)]>) pap.Clone(); break; } pap = new PAP_n<T[τ(a₁)],...,T[τ(a_m)]>; if(Γ(τ(a₁)) > 0){ pap.arg1 = pop<τ(a₁)>(); ... if(Γ(τ(a_{n-1})) > 0){ pap.arg<n> = pop<τ(a₁)>(); } } break; </pre> |
| $\mathbf{SC}[id = f_1 \dots f_m \setminus n \ a_1 \dots a_n, i]$ $i = 0$ | = | <pre> pap = (PAP_<n<T[τ(a₁)],...,T[τ(a_m)]>) closure.pap; return <tail><id.FEP>(pap.arg1,..., pap.arg<n>); </pre> |

Figura A.4. Esquemas de Compilação **S** e **SC**

