Pós-Graduação em Ciência da Computação

Integração de Linguagens Funcionais à Plataforma .NET utilizando o Framework Phoenix

Por

Guilherme Amaral Avelino

Dissertação de Mestrado

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE, AGOSTO/2008

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Guilherme Amaral Avelino

Integração de Linguagens Funcionais à Plataforma .NET Utilizando o Framework Phoenix

ESTE TRABALHO FOI APRESENTADO À PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DO CENTRO DE INFORMÁTICA DA UNIVERSIDADE FEDERAL DE PERNAMBUCO COMO REQUISITO PARCIAL PARA OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIA DA COMPUTAÇÃO.

ORIENTADOR: Prof. Dr. ANDRÉ LUIS DE MEDEIROS SANTOS

RECIFE, AGOSTO/2008

Avelino, Guilherme Amaral

Integração de linguagens funcionais à plataforma .NET utilizando o framework Phoenix / Guilherme Amaral Avelino. – Recife: O Autor, 2008.

104 folhas : il., fig., tab.

Dissertação (mestrado) – Universidade Federal de Pernambuco. CIn. Ciência da computação, 2008.

Inclui bibliografia e apêndices.

Linguagem de programação. 2. Compiladores. I. Título.

005.1 CDD (22.ed.) MEI2008-100

AGRADECIMENTOS

Agradeço a todos aqueles que, direta ou indiretamente contribuíram para a realização desta pesquisa e em especial:

- Primeiramente a Deus, por ter me dado saúde, inteligência e perseverança necessária à execução deste projeto.
- Aos meus pais, Paulo Lustosa Avelino e Aldênia Maria Amaral Santos Avelino, pelo carinho, amor e dedicação com que se empenharam na minha formação pessoal e profissional;
- A Lyvia Basílio Caland, minha namorada, pela compreensão nos momentos de ausência e pelo apoio e incentivo constante durante esta fase de minha vida;
- Ao professor André Santos, pela oportunidade de desenvolver este projeto e acima de tudo por sua excelente orientação e auxílio nos mais diversos problemas enfrentados durante a realização deste;
- Aos amigos do mestrado, em especial a Armando Soares, Vinícius Pádua e Marcos Duarte, pela motivação, auxílio e companheirismo. Além de um convívio fraterno que proporcionou um ambiente propício ao desenvolvimento deste trabalho;
- A Simon Peyton Jones, Tim Chevalier e demais participantes do fórum do GHC que contribuíram com informações importantes sobre o GHC e a linguagem CORE;
- A Andy Ayers e Matt Mitchell, membros da equipe desenvolvimento do Phoenix, pela sempre atenciosa forma com que responderam as minhas mais variadas dúvidas sobre o uso desta ferramenta.
- A Monique Louise de Barros Monteiro, pelas explicações a respeito do projeto Haskell .NET e pelas dicas e comentários bastante úteis para o desenvolvimento deste projeto.
- *À Microsoft Research* pelo apoio financeiro, permitindo que eu me dedicasse integralmente ao projeto.

- Ao Centro de Informática e a sua excelente equipe de professores e profissionais, que muito contribuíram para minha formação e proporcionaram a base para o desenvolvimento deste trabalho.
- A todos os meus amigos e familiares, pelo apoio.

RESUMO

Linguagens funcionais se destacam pelo seu alto poder de expressão e abstração, promovido por construções de alto nível como polimorfismo paramétrico, funções de alto nível e aplicações parciais. Embora estes recursos sejam bastante úteis, tradicionalmente, linguagens funcionais têm sido pouco empregadas fora do ambiente acadêmico. Esta situação é em parte explicada pela ausência de uma infra-estrutura de desenvolvimento que forneça ferramentas e APIs capazes de aumentar a produtividade e permita o uso das mais recentes tecnologias.

Uma alternativa para fornecer esta infra-estrutura é integrar linguagens funcionais a plataformas que disponibilizem tais facilidades, como a .NET. Embora a plataforma .NET tenha sido projetada de forma a suportar múltiplas linguagens, seu foco foi dado ao suporte dos paradigmas imperativo e orientado a objeto, carecendo de estruturas que permitam um mapeamento direto de linguagens funcionais.

Objetivando estudar novas técnicas de mapeamento de estruturas funcionais na plataforma .NET, neste trabalho foi desenvolvido um compilador funcional que gera código .NET, utilizando o framework Phoenix. O uso do framework Phoenix além de auxiliar na geração inicial do código permitiu que análises e otimizações fossem feitas, posteriormente, melhorando o desempenho dos programas gerados.

Palavras-chave: Linguagem funcional; NET; Phoenix; STG; Compiladores.

ABSTRACT

Functional languages stand out for their high power of expression and abstraction, promoted by high-level buildings as parametric polymorphism, high-level functions and partial applications. However these features are quite useful, traditionally, functional languages have been little used outside the academic environment. This is partly explained by the lack of a development infrastructure that provides tools and APIs which are capable of increasing the productivity and allow the use of latest technologies.

An alternative to provide this infrastructure is to integrate functional languages to platforms that provide such facilities, such as .NET. Although the platform. NET has been designed in a way that supports multiple languages, its focus was given to the support of imperative paradigms and the object oriented, lack of structures that allow a direct mapping of functional languages.

Aiming to study new techniques for mapping of functional structures on the platform. NET, in this work, a functional compiler that generates .NET code was developed, using Phoenix framework. Apart from helping in generating initial code, the use of the Phoenix framework permitted analyses and optimizations to be made, subsequently, improving the performance of the generated programs.

Keywords: Functional language; NET; Phoenix; STG; Compilers.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS

1 INTRODUÇÃO

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

- A Seção 1.1 apresenta os fatores que motivaram o presente trabalho, dando uma breve introdução sobre linguagens funcionais, máquinas virtuais gerenciadas e motivação para integrá-las.
- A Seção 1.2 descreve a estrutura da dissertação, apresentado os assuntos discorridos em cada capítulo.

1.1 Contexto e Motivação

Linguagens funcionais se caracterizam por tratar funções como unidade fundamental de um programa. Desta forma, um programa é constituído por um conjunto de funções que representam sub-partes do problema a ser resolvido. Este tipo de divisão do problema representa uma forma de modularizar ainda mais um problema, pois funções representam problemas específicos a serem resolvidos que podem ser utilizados em mais de uma solução. Diferentemente de linguagens imperativas, nas quais funções são tratadas como uma série de instruções, em linguagens funcionais elas são tratadas como expressões matemáticas. Na programação funcional é evitado uso de estados ou dados mutáveis e a execução de uma função, quando submetida aos mesmos argumentos, sempre retorna o mesmo valor o que garante a ausência de efeitos colaterais e facilita o processo de prova da correção de um programa [HYPERLINK \l "Hughes1989" **1**].

Versões mais recentes de linguagens de grande popularidade, tais como Java e C#, têm incorporado algumas destas características, antes só encontradas em linguagens funcionais, numa clara demonstração da importância e poder de expressão destas. Polimorfismo paramétrico, através de *generics*, e *closures*[1] são

[1] Inserida a partir da versão 2.0 do C# através de *anonymous delegates* e incrementado na versão 3.0 com a criação de expressões lambdas. Para a linguagem Java closures se encontra em fase de análise da proposta[HYPERLINK \l "Bra08" **67**], a ser incorporada na versão 7.

exemplos dos recursos incorporados a estas linguagens. Tendo em mente este interesse de linguagens orientadas a objetos em características típicas do paradigma funcional, surge uma pergunta: porque tais linguagens não têm seu uso difundido fora do mundo acadêmico?

Um dos principais fatores que dificulta a expansão destas linguagens é a ausência de uma infra-estrutura de desenvolvimento que forneça ferramentas e APIs capazes de aumentar a produtividade e permita o uso das mais recentes tecnologias. Plataformas como Java (JVM) e .NET, fornecem aos programadores tais ferramentas e APIs permitindo um enorme ganho em produtividade e uma rápida integração com os modelos e tecnologias de desenvolvimento mais recentes. Outra característica importante provida por estas plataformas é o uso de máquinas virtuais e código intermediário. Esta característica fornece uma maior abstração sobre a máquina alvo, permitindo que programas e compiladores sejam desenvolvidos sem se preocupar com o hardware ou sistema operacional onde irão trabalhar.

O ambiente .NET destaca-se por prover suporte a múltiplas linguagens de programação, permitindo que programas sejam construídos utilizando qualquer uma das linguagens suportadas, podendo ainda, um programa ser constituído de módulos, escritos em linguagens diferentes, que interagem entre si. Além de já prover inúmeras linguagens (C#, J#, C++, VB .NET, etc.), o ambiente .NET permite fácil incorporação de novas linguagens, desde que, estas sigam as especificações do *Common Language Runtime* (CLR)**2**,**3**]. O CLR é a implementação da *Microsoft* para a *Common Language Infrastructure* (CLI)[HYPERLINK \l "ECMA335" **4**], a qual define um rico sistema de tipos e uma máquina virtual capaz de executar de forma eficiente códigos provenientes de diversas linguagens.

Embora de forma não restritiva, o CLR foi desenvolvida com foco na implementação de linguagens que seguem os paradigmas imperativo e orientado a objetos. Desta forma, mapear características de linguagens funcionais, tais como: função de alta ordem, mecanismo *lazy evaluation* e polimorfismo paramétrico, na plataforma .NET representam um desafio. Diminuir este *gap* semântico através de estruturas que mapeiem, eficientemente, características comuns a linguagens funcionais na plataforma .NET é objetivo comum de diversos projetos, tais como: Haskell .NET**5**], ILX[HYPERLINK \l "Syme2001" **6**], Mondrian .NET**7**], Bigloo .NET[

HYPERLINK \l "Bres2004a" **8**] e Nemerle**9**]. Cada uma destas implementações define suas próprias estruturas de mapeamento, não havendo um consenso sobre qual a melhor forma de se representar tais características no ambiente .NET. De modo geral a implementação das estruturas propostas no ambiente gerenciado fornecido pela CRL não possui um bom desempenho, o que abre caminho para estudos de técnicas mais eficientes.

Para mapear tais funcionalidades de forma eficiente é necessária uma série de experimentações e testes, de forma a obter estruturas que as represente com o melhor desempenho possível. O framework Phoenix [2], disponibilizado pela Microsoft, é uma ferramenta que tem como propósito facilitar a construção de compiladores e de ferramentas de teste e análise. Ele utiliza uma representação intermediária fortemente tipada para representar um programa e disponibiliza uma grande quantidade de classes e métodos para manipular esta representação. Dentre os recursos disponibilizados, temos o redirecionamento de código para diferentes arquiteturas e plataformas tais como: x86 e MSIL[2] e mecanismo de *plugin*, o qual permite alterar o comportamento de um programa *Phoenix* sem ter de alterar diretamente seu código fonte.

O presente trabalha faz uso do framework Phoenix para a criação e análise de estruturas que mapeiem, eficientemente, as características específicas de linguagens funcionais no ambiente .NET. Espera-se que os recursos disponibilizados pelo framework auxiliem na construção de um compilador que gere códigos mais expertos, ou seja, que usem menos recursos e sejam mais rápidos que os produzidos atualmente. O compilador gerado servirá ainda como ferramenta para experimentação e desenvolvimento de novas técnicas de compilação de linguagens funcionais no ambiente .NET.

1.2 Organização da Dissertação

Além da introdução esta dissertação conta com mais cinco capítulos e três apêndices, como segue:

[2] Microsoft Intermediate Language

- O **Capitulo 2** apresenta uma definição geral do paradigma funcional e do ambiente .NET, mostrando suas principais características. Após a apresentação das principais características são demonstradas possíveis abordagens de como implementar uma linguagem funcional no ambiente .NET. Por fim, é feito um resumo das principais implementações de linguagens funcionais existentes.
- O **Capítulo 3** discorre sobre o Framework Phoenix. Nele são apresentadas as principais características e recursos desta ferramenta, sempre que possível através de exemplos práticos.
- O **Capítulo 4** trata da implementação do protótipo. Nele é descrito a arquitetura do compilador, seu ambiente de execução e as decisões de projeto tomadas para geração do código.
- O **Capítulo 5** faz a análise do compilador e mostra o resultado das otimizações e testes realizados. Os primeiros resultados se referem a melhorias na transformação do código IR para MSIL e ao final são exibidos os resultados de otimizações no controle da pilha de execução e em instruções de desvios.
- O **Capítulo 6** aponta as contribuições deste trabalho, restrições e opções para trabalhos futuros.
- O **Apêndice A** apresenta tabelas com as classes que representam as unidades de compilação do compilador PhxSTGCompiler.
- O **Apêndice B** apresenta o código da ferramenta construída para gerar o perfil de consumo de memória dos programas analisados.
- O **Apêndice C** mostra o código de um *plugin*, utilizado para substituir *tail calls* por desvios incondicionais em chamadas recursivas.

2 PROGRAMAÇÃO FUNCIONAL NA PLATAFORMA .NET

Aliar a alta expressividade e o poder de abstração fornecidos por linguagens funcionais a plataformas de alta produtividade como o .NET não é uma tarefa simples. A plataforma .NET tem um modelo de compilação voltado para os paradigmas imperativo e orientado a objeto, o que dificulta o mapeamento de estruturas características de linguagens funcionais.

Neste capítulo é feito uma introdução a linguagens funcionais e suas principais características, sendo, em seguida dada uma breve introdução sobre a plataforma .NET. Após discorrer sobre estes conceitos básicos são apresentadas técnicas que permitem mapear linguagens funcionais na plataforma .NET. Finalizando o capítulo, alguns projetos de mapeamento de linguagens funcionais são apresentados descrevendo algumas de suas decisões de projetos.

2.1 Introdução a Linguagens Funcionais

Linguagens funcionais se caracterizam por tratar funções como unidade fundamental de um programa. Desta forma, um programa é constituído por um conjunto de funções que representam sub-partes do problema a ser resolvido. Diferentemente de linguagens imperativas, nas quais funções são tratadas como uma série de instruções, em linguagens funcionais elas são tratadas como expressões matemáticas. Na programação funcional é evitado uso de estados ou dados mutáveis e a execução de uma função, quando submetida aos mesmos argumentos, sempre retorna o mesmo valor o que garante a ausência de efeitos colaterais e facilita o processo de provar a correção de um programa [HYPERLINK \l "Hughes1989" **1**].

Linguagens funcionais são caracterizadas por alta expressividade e grande poder de abstração, decorrentes de construções de alto nível tais como funções de alta ordem, aplicação parcial de funções, avaliação preguiçosa e polimorfismo paramétrico. Estas construções não só aumentam expressividade da linguagem,

como também a complexidade de sua compilação, especialmente em ambientes orientados a objetos como o .NET. Tais características são melhores especificadas a seguir.

2.1.1 Funções de alta ordem

Diferentemente de linguagens imperativas e orientadas a objetos, onde há uma clara distinção entre dados e funções, linguagens funcionais não fazem tal distinção, tratando funções como valores de primeira classe. Sendo assim, como qualquer outro valor, elas podem ser passadas como argumentos, retornadas como resultado de outra função, ou ainda armazenadas em estruturas de dados.

Uma função é dita de alta ordem quando recebe outra função como um argumento ou computa outra função como seu resultado. Por exemplo, uma função de alta ordem pode atravessar uma lista aplicando uma função recebida como argumento em cada componente da lista**10**].

Em linguagens funcionais uma função pode ser criada em tempo de execução e referenciar variáveis visíveis apenas onde ela foi declarada. Tais variáveis são denominadas variáveis livres. Os valores referentes a estas variáveis fazem parte da definição da função e por isto a representação de uma função deve conter não só a expressão que a compõe, como também suas variáveis livres. A forma mais direta para esta representação é através de uma *closure*[HYPERLINK \l "Minamide1996" **11**], objeto alocado dinamicamente que encapsula um código a ser executado e um ambiente que pode ser acessado pelo código. *Closure* não é uma estrutura padrão em ambientes orientados a objetos como o .NET. Alternativas para sua representação serão apresentadas na Seção 2.4.1.

2.1.2 Aplicação parcial de funções

Linguagens funcionais permitem descrever funções com mais de um argumento como uma composição de funções de um argumento, de forma que

um argumento seja consumido por vez. Este processo, denominado *currificação*[3] em homenagem a *Haskell Curry*, altera a concepção, popularizada pelas linguagens imperativas, de que todos os argumentos de uma função devem ser passados ao mesmo tempo, como se fosse uma única estrutura de dados.

Embora sua sintaxe favoreça a *currificação* de funções, Haskell permite a criação de funções sem seu uso, utilizando para isto o conceito de tupla. O exemplo a seguir descreve a mesma função com e sem *currificação*.

```
1  multiply :: Int -> Int ->Int
2  multiply x y = x*Y
3
4  multiplyUC :: (Int,Int) -> Int
5  multiplyUC (x,y) = x*Y
```

Código 1. Funções *curry* e *não-curry*

A função *multiplyUC* só é executada ao receber os dois argumentos requeridos através de uma tupla. Já a função *multiply* permite sua aplicação mesmo passando a ela menos argumentos do que o requerido, obtendo assim, uma função parcial que armazena o argumento recebido e pode ter sua execução completada quando aplicada ao argumento restante.

A técnica de executar uma função *currificada* utilizando menos argumentos do que o número máximo de parâmetros suportados é denominada *aplicação parcial*[**10**].

2.1.3 Avaliação preguiçosa

Uma função nem sempre requer que todos seus argumentos sejam avaliados. Algumas vezes o uso de um argumento depende da avaliação de outra expressão ou mesmo nunca é utilizado dentro do corpo da função. Sendo assim, a decisão de quando deve ser feita a avaliação dos argumentos pode influenciar não só no projeto de uma linguagem como também no seu desempenho. Segundo David Watt[**10**], quanto ao momento em que é feita esta avaliação, podemos distinguir dois mecanismos:

[3] Embora tenha recebido este nome em Homenagem a Haskell Curry, esta técnica foi inventada por Moses Schönfinkel.

- *Eager Evaluation* – todos os argumentos são avaliados apenas uma vez, antes da chamada e o valor obtido é ligado a cada ocorrência do parâmetro formal no corpo da função.
- *Normal-order evaluation* – os argumentos são avaliados após a chamada da função, apenas quando requisitados. Ou seja, cada ocorrência do parâmetro formal na função é substituída pela expressão não avaliada.

O primeiro mecanismo ao requerer que todos os argumentos sejam avaliados antes da chamada pode gastar um tempo desnecessário em casos onde algum dos argumentos não é utilizado no corpo da função. Já o segundo é menos eficiente em funções onde um determinado parâmetro formal é utilizado mais de uma vez no corpo da função, necessitando que a mesma expressão seja avaliada mais de uma vez.

Linguagens funcionais tais como Haskell[**12**], Mondrian[**13**] e Lazy ML[**14**] utilizam um aprimoramento do *normal*-order *evaluation*, denominado avaliação preguiçosa, onde cada argumento é avaliado apenas quando necessário e uma única vez. Tal mecanismo além de evitar avaliações desnecessárias permite a criação de estruturas de dados infinitas tais como *lazy list*[**10**], onde cada elemento é avaliado sob demanda.

Quando uma função sempre usa um determinado argumento, dizemos que ela é estrita para aquele argumento. Sendo assim, linguagens que implementam avaliação preguiçosa ou *normal-ordem* evaluation são denominadas não estritas, pois podem possuir argumentos que não sendo utilizados nunca serão avaliados.

2.1.4 Polimorfismo paramétrico

Grande parte das linguagens funcionais dá suporte a polimorfismo paramétrico, onde uma função ou estrutura de dados pode ser definida para operar sobre diversos tipos. No polimorfismo *ad-hoc,* implementado por linguagens orientadas a objeto através de mecanismos de herança ou sobrecarga, os tipos suportados são restritos e devem ser previamente especificados. Já no polimorfismo paramétrico é permitido o uso de qualquer tipo, devendo a operação que o utiliza ser executada independente do formato do tipo. Como na prática muitas funções

são naturalmente polimórficas, o polimorfismo paramétrico eleva a expressividade da linguagem.

Um exemplo clássico de uma aplicação de polimorfismo paramétrico é a função *length,* que calcula o número de elementos de uma lista. O código Haskell a baixo implementa a função *length* para o cálculo de uma lista de inteiros.

```
1  length :: [Int]->Int
2  length [] = 0
3  length (x:xs) = 1 + (length xs)
```

Código 2. Função length não polimórfica

Embora funcione perfeitamente a função definida desta forma é restrita a listas de inteiros. Como as operações executadas em *length* são independentes do tipo dentro da lista podemos generalizar a função para qualquer tipo.

```
1  length::[t]->Int
2  length [] = 0
3  length (x:xs) = 1 + (length xs)
```

Código 3. Função length polimórfica

Como veremos na Seção 2.1.5 polimorfismo paramétrico também pode ser utilizado para modelar uniões discriminadas, permitindo a construção de tipos de dados complexos que armazenam tipos polimórficos.

2.1.5 Tipos algébricos

Tipos de dados algébricos formam a base do sistema de tipos da maioria das linguagens funcionais modernas. Eles permitem a definição de tipos estruturados, uniões e tipos recursivos. Um tipo algébrico é um tipo de *união discriminada etiquetada*[**10**], onde novos tipos são definidos utilizando construtores (etiquetas) e os tipos dos argumentos.

```
1  data ListInt = Cons Int List | Nil
```

Código 4. Tipo algébrico ListInt

No Código 4 é definido o novo tipo algébrico ListInt o qual pode conter dois tipos de dados, definidos pelos construtores Cons e *Nil*. *Nil* é um construtor vazio, pois não possui nenhum campo, já Cons carrega informações através de argumentos dos tipos Int e List. Desta forma *Cons* recebe um valor inteiro e um valor do tipo ListInt, ou seja é um tipo recursivo, pois recebe um valor que ele próprio define.

Da mesma forma mostrada com a função length, podemos generalizar tipos algébricos de forma que eles possam representar tipos de dados polimórficos. A definição de List fornecida no Código 5 cria uma lista que pode armazenar qualquer valor suportado pela linguagem.

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

Código 5. Tipo algébrico genérico

2.2 Plataforma .NET

A plataforma .NET[**15**] é um ambiente de desenvolvimento e execução que permite diferentes linguagens de programação e bibliotecas trabalharem juntas na construção de aplicações. A portabilidade destas aplicações também é facilitada, pois um programa criado para a plataforma .NET deve rodar em qualquer dispositivo ou sistema operacional que possua uma implementação de seu ambiente de execução. Com objetivo de ampliar esta portabilidade em diferentes sistemas a Microsoft submeteu o projeto da máquina virtual, *Common Language Infrastructure* (CLI)[**4**], para padronização nos órgãos internacionais ECMA[**16**] e ISO[**17**]. Desta forma, desenvolvedores de diferentes sistemas operacionais e dispositivos podem construir sua própria versão da CLI capaz de executar aplicativos .NET independente de autorização ou suporte da Microsoft.

2.2.1 CLR

O CLR é a implementação da *Microsoft* para o padrão CLI, que define especificações para código executável e ambiente de execução da plataforma.NET. Este ambiente utiliza um compilador *Just-In-Time* (JIT) que permite a execução de programas traduzidos para uma linguagem intermediária comum (MSIL[4][**18**]), carregando e compilando para código binário partes do código sobre demanda. Este modelo de compilação sobre demanda permite que otimizações sejam feitas de acordo com a plataforma na qual o código é executado.

[4] A linguagem intermediária comum implementada na CLR é denominada *Microsoft Intermediate Language* (MSIL) e não *Common Intermediate Language* (CIL), como definido pela CLI. Desta forma sempre que for mencionado MSIL entenda linguagem intermediária comum implementada pela CLR.

O processo de compilação e execução de programas, como observado na Figura 1, pode ser descrito nos seguintes passos:

1. O programa escrito em uma das linguagens suportadas pela plataforma (C#, VB.NET, C++, J#, Haskell, etc.) é compilado para uma linguagem intermediária, a Microsoft Intermediate Language (MSIL).
2. Este código MSIL pode fazer chamadas a métodos e classes escritos em outras linguagens que também tenham sido compilados para MSIL, ou ainda para o conjunto de classes da biblioteca .NET. Desta forma o uso de uma linguagem intermediária facilita a interoperabilidade entre diferentes linguagens.
3. O código MSIL é então submetido ao CLR para que seja feita a execução do programa.
4. O CLR, inicialmente, busca por uma versão pré-compilada do código na *cache*. Caso não encontre ou detecte que a versão resgatada tenha sido alterada é feita a compilação através do JIT.
5. O JIT compilará então cada classe à medida que um método pertencente a esta for requisitado. Isto vale também para métodos provenientes da biblioteca .NET.
6. O código compilado é então executado dentro do ambiente gerenciado .NET, o qual verifica diretivas de segurança e acesso à memória.

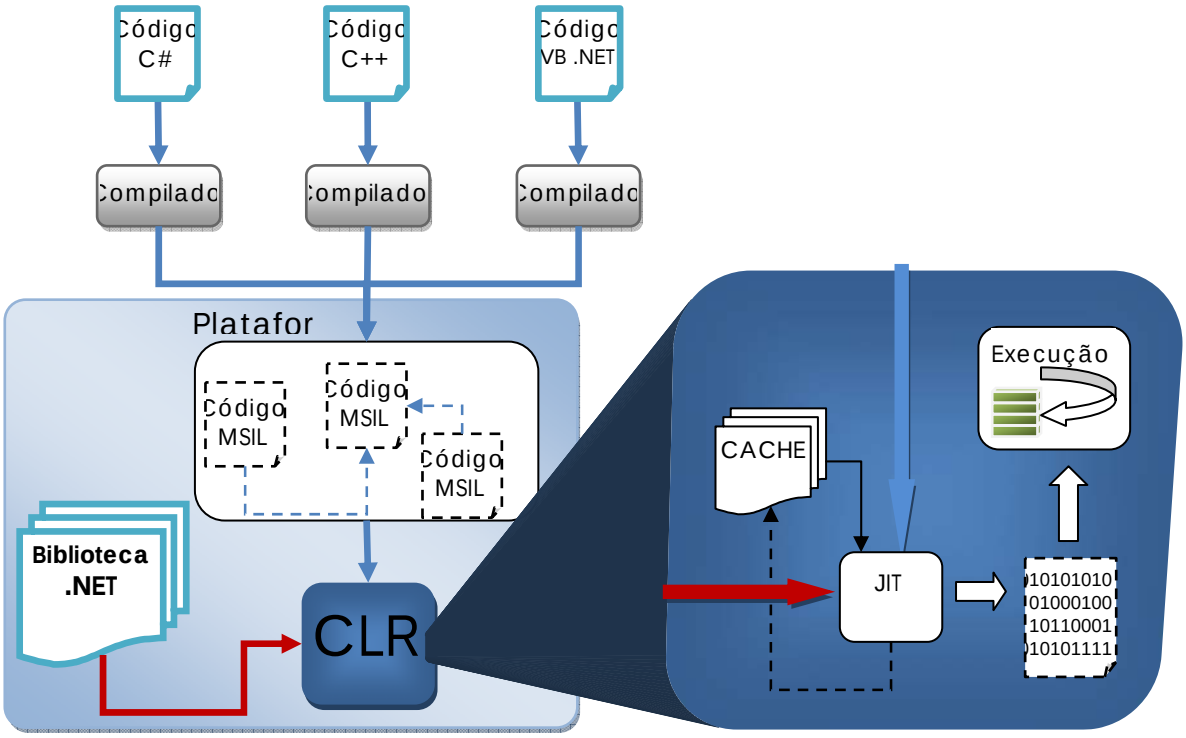

Figura 1. Ambiente .NET

2.2.2 Outras Implementações da CLI

Ao padronizar a CLI a Microsoft possibilitou o surgimento de novas implementações desta para sistemas operacionais e arquiteturas diferentes, promovendo a portabilidade de programas .NET. Dentre as diversas implementações da CLI existentes duas se destacam: a Shared Source CLI (SSCLI ou projeto Rotor)[**19**] e o projeto MONO[**20**].

A SSCLI é uma versão de código livre da CLI e do compilador C# implementada pela própria Microsoft para execução no Windows, FreeBSD e Mac OS X[5]. Esta implementação tem cunho estritamente acadêmico, fornecendo um ambiente de estudo da plataforma .NET e das tecnologias nela empregadas tais como: gerenciamento de memória, coleta de lixo, compilação sob demanda, etc.

[5] Apenas para versão 1.0 da SSCLI, a versão 2.0 não disponibiliza mais versões para FreeBSD e Mac OS X.

Por ser voltada para estudo não há uma preocupação quanto ao desempenho, o que foi confirmado em testes comparando o tempo de execução de programas na SSCLI e na CLR[**21**].

O projeto MONO, financiado pela Novell[**22**], provê implementações de código livre da CLI para sistemas operacionais Windows, Linux, Unix, Solaris e Mac OS X. É um projeto consistente, com uma grande comunidade de desenvolvedores que incrementa a portabilidade de programas .NET para além do ambiente Windows.

2.3 Integração à Plataforma .NET

Antes de definir como será feito o mapeamento das estruturas funcionais na plataforma .NET é necessário escolher uma estratégia através da qual será feita tal integração. Esta estratégia define se será utilizado algum mecanismo responsável pela comunicação entre a linguagem e a plataforma ou se será gerado diretamente código suportado por esta.

2.3.1 Bridge

Permitir a comunicação entre componentes escritos em diferentes linguagens, de forma que, possam trocar informações e acessar recursos uns dos outros é a função de uma bridge, ou "*ponte*". A *bridge* é responsável por intermediar as trocas de mensagens, fornecendo uma sintaxe comum, e pela tradução dos parâmetros e valores de retornos, processo este conhecido como *marshalling*[6]. Antes mesmo de se integrar linguagens funcionais a ambientes gerenciados, como .NET e Java, esta estratégia já era utilizada para permitir tal integração para código nativo, como é caso de HDirect[**23**] e GreenCard[**24**], que implementam a *Foreign Function Interface*[7] (FFI). Em ambientes gerenciados, Hugs .NET[**25**] e Lambada[**26**]

[6] Processo de transformação da representação na memória de um objeto em formato apropriado para armazenamento ou transmissão. O processo contrário no qual os dados são novamente transformados em objetos na memória é denominado *unmarshalling*.

[7] Definição da interface para funções externas para linguagem Haskell98.

são exemplos de integração para a linguagem Haskell, respectivamente para as plataformas .NET e Java.

Esta é uma estratégia interessante quando o objetivo é obter a integração sem a necessidade de grandes alterações no compilador ou na plataforma, pois toda a complexidade das operações de conversões de tipos e estruturas fica a cargo da *bridge*. Entretanto esta integração é superficial, no geral apenas chamada de funções, não disponibilizando o acesso a recursos avançados. Outra limitação desta estratégia é quanto ao desempenho, o processo de conversão de tipos é custoso e este *overhead* deve ser levado em consideração em um projeto de integração.

Na plataforma .NET outro fator deve ser considerado: este tipo de integração requer chamadas a código não gerenciado, pois o código gerado pelo compilador funcional gera código nativo, ou seja, não gerenciado pela plataforma . Embora seja permitido este tipo de chamada ela requer que uma série de operações como confirmação de permissões e importação de bibliotecas, que degradam seu desempenho. Há ainda que se considerar que implementações de linguagens funcionais, geralmente, inclui seu próprio ambiente de execução com coletor de lixo e gerenciamento de memória próprios, sendo assim teríamos um cenário onde dois ambientes de execução estariam rodando ao mesmo tempo e consumindo recursos do sistema.

2.3.2 Compilação

Gerar código suportado diretamente pela plataforma, através de um processo de compilação, é a forma mais direta de integração. Este processo pode tanto ser feito utilizando como destino uma linguagem de alto nível que possua um compilador para o ambiente, como diretamente, gerando código MSIL. A primeira abordagem é mais fácil, pois delega ao compilador da linguagem escolhida a responsabilidade de gerar corretamente o código para a plataforma, além de se valer de otimizações implementadas por esta. A segunda abordagem embora seja mais complexa e susceptível a erros, permite um maior controle sobre o código gerado e uso de instruções não contempladas pelas linguagens de alto nível. Para

auxiliar a geração direta de código podemos utilizar ferramentas tais como *peverify*[8], *ildasm*[9], *ilasm*[10] e Phoenix. Esta última será detalhada no Capítulo 3.

A integração utilizando compilação possui diversas vantagens em relação ao mecanismo de *bridge*. O compartilhamento de uma mesma representação facilita a comunicação com programas escritos em outras linguagens, reduzindo o *overhead* causado pelo processo de *marshilling/unmarshalling* e pela chamada a código não gerenciado. O uso de um mesmo ambiente de execução diminui o uso de recursos do sistema que antes teria que ser compartilhado por dois ambientes com coletores de lixo e gerenciamento de memória separados.

A maioria dos projetos de integração de linguagens funcionais à plataforma .NET utilizam a compilação como abordagem. Mondrian[**13**] e Making Haskell .NET Compatible [**27**] fazem uso de uma linguagem de alto nível para gerar código enquanto que Nemerle[**9**] e Haskell .NET[**5**] geram diretamente código MSIL.

2.3.3 Estendendo a CLI

Os tipos e a linguagem intermediária descritos pela *Common Language Infrastructure* (CLI) visam proporcionar um ambiente que suporte a implementação de diversas linguagens capazes de interagir entre si, entretanto seu foco é dado a linguagens imperativas e orientada a objetos. Desta forma, faltam a este ambiente estruturas básicas para a representação de funcionalidades comuns a linguagens funcionais. Modificar a CLI adicionando extensões necessárias para representar estruturas funcionais facilitaria a compilação de linguagens funcionais para a plataforma .NET. O projeto ILX [**28**] utilizou esta abordagem, adicionando a CLI novas características como closures, polimorfismo paramétrico, uniões discriminadas e funções de alta ordem.

Alterar a máquina virtual permite a implementação de linguagens funcionais com um ganho expressivo no desempenho, além de deixar um legado para futuras

[8] Ferramenta, disponibilizada com o *framework* .NET, que verifica se o código MSIL esta de acordo com as especificações definidas pela CLI.

[9] *MSIL disassembler*. Gera código MSIL a partir de um arquivo PE (DLL ou EXE).

[10] *MSIL assembler*. Gera um arquivo PE (DLL ou EXE) a partir de código MSIL.

implementações. Entretanto, perde na portabilidade, pois requer que o novo ambiente seja distribuído junto com a linguagem, ou ainda que estas modificações sejam incorporadas a distribuição padrão, o CLR no caso da plataforma .NET . A CLI segue uma padronização, ECMA-335 [**4**], e a incorporação de novas características a este é dificultada, pois requer aprovação de um conselho de padronização.

O projeto F#[**29**], desenvolvido pela mesma equipe que criou a ILX, faz uso desta última como linguagem alvo do processo de compilação. ILX, por sua vez, é posteriormente traduzido para MSIL, de forma a preservar a compatibilidade com o ambiente padrão de .NET.

2.4 Mapeando Estruturas Funcionais em Ambientes OO

Para que seja feito o mapeamento de linguagens funcionais em um ambiente OO, como o .NET, faz-se necessário o desenvolvimento de técnicas e estruturas capazes de diminuir o *gap* semântico entre estes dois mundos. Nesta Seção tais técnicas estruturas serão apresentadas e discutidas.

2.4.1 Closures

Closures são estruturas essenciais para a representação de linguagens funcionais. Sendo assim o modelo adotado para a representação desta influenciará todo o restante do projeto. Podemos definir uma *closure* como uma função que armazena todas as variáveis utilizadas por ela, mas que foram definidas fora dela. Tais variáveis são definidas na teoria do cálculo lambda[**30**] como variáveis livres. Através do exemplo mostrado no Código 6 podemos observar com mais detalhes tais conceitos.

```
1  f1 :: Int -> t -> (Int -> Int)
2  f1 x y = let f2 k = x + k in f2
```

Código 6. Exemplo de closure

A função *f2* definida dentro da função *f1,* utilizando o comando *let,* faz uso da variável *x* definida fora de seu escopo, ou seja, *x* é uma variável livre da função *f2*. Ou seja, *f2* é uma *closure* que representa uma função que recebe um argumento *k* e faz uso de uma variável livre, a qual deve ser encapsulada dentro de sua representação. A função *f1* também pode ser considerada uma *closure*, só

que sem variáveis livres, o que faz sentido para uma representação única para todas as funções.

Em linguagens funcionais, além de representar funções, *closures* são comumente utilizadas para representar expressões não avaliadas, conhecidas como *thunks*. Em linguagens com mecanismo de avaliação preguiçosa (Seção 2.1.3) onde a avaliação das expressões é feita apenas uma vez e somente quando necessária, *closures* são utilizadas para representar a expressão a ser avaliada, armazenando suas variáveis livres e o valor resultante após a avaliação.

Closures são, normalmente, implementadas através de estruturas de dados especiais que contém um ponteiro para o código da função e o ambiente léxico da função (conjunto de variáveis livres)[**28**,**31**]. Esta abordagem é inviabilizada, ou ainda desestimulada, em ambientes com gerenciamento de memória, como o .NET, onde o uso de ponteiros embora permitido, gera código não verificável[11]. Ainda que, projetos como o ILX[**6**] tenham utilizado código não verificável para a construção de closures esta abordagem sofre de restrições de uso, uma vez que a execução de código não verificável requer permissões específicas e não pode se valer das garantias e funcionalidades fornecidas pela CLI. O próprio projeto ILX abandonou tal abordagem em implementações posteriores.

Uma alternativa ao uso de ponteiro em código verificável é o uso de estruturas conhecidas como *delegates*. *Delegate* é a versão orientada a objetos de ponteiro para função, que permite a chamada de métodos, tanto de instância como estático, de forma segura e verificável. Na implementação 1.0 da CLR havia problemas de desempenho, o que justificou a utilização de ponteiros na ILX, entretanto testes realizados demonstraram que tais problemas foram solucionados a partir da versão 2.0 fazendo com que chamadas a métodos utilizando *delegates* tenham desempenho semelhante a chamadas a métodos virtuais ou de interface [**21**].

[11] Código não verificável, no ambiente .NET, significa que o código não segue as restrições de segurança impostas pela CLI não sendo gerenciado diretamente pelo ambiente.

2.4.1.1 Projetando uma closure

Uma forma bastante direta de se representar closures em ambientes orientados a objetos é através da definição de uma classe abstrata *Closure* que possui um método *Invoke*, responsável pela execução da expressão. Neste modelo para cada closure deve ser criada uma nova classe que herda da classe *Closure*, armazena suas variáveis livres em campos da classe e sobrescreve o método *Invoke* de forma que ele execute o código correspondente a avaliação da closure. O Código 7 demonstra como criar uma nova closure estendendo a classe abstrata.

```
1  //Classe abstrata Closure
2  public abstract class Closure
3  {
4       public abstract object Invoke();
5  }
6
7  // Criando uma nova closure
8  class newClosure : Closure
9  {
10      // Campos representando variáveis livres
11
12      public override object Invoke()
13      {
14         //Código da closure
15      }
16 }
```

Código 7. Representação de closures utilizando uma classe abstrata

Para passagem de argumentos para a função *Invoke* poderia ser utilizado um *array* de objetos ou ainda uma pilha. F# [**29**] possui classes abstratas pré-definidas para até cinco argumentos e um valor de retorno, utilizando *generics*[**32**] para definição dos tipos. Funções com mais que cinco argumentos são tratadas utilizando aplicações parciais, mecanismo detalhado na Seção 2.5.4. Nemerle[**9**] utiliza mecanismo semelhante, entretanto possui classes abstratas pré-definidas para até vinte argumentos, além de permitir chamadas não *currificadas* utilizando para tanto uma tupla contendo todos os argumentos da função. É importante observar que embora hajam classes pré-definidas para cada nova closure definida deverá ser produzida uma nova classe que herde da classe correspondente, sobrescrevendo seu método *Invoke* e adicionando campos para suas variáveis livres.

Tanto F# como Nemerle são linguagens estritas, o que reduz o número de *closures* geradas, uma vez que, não são necessárias novas *closures* para representar computações não avaliadas. Entretanto, a geração de uma classe por *closure* em

linguagens funcionais não estritas, como Haskell, resultaria em uma grande quantidade de classes. Segundo Don Syme [**6**], estima-se que seja encontrado na biblioteca padrão do GHC uma *closure* por linha de código Haskell. Como na plataforma .NET a cada classe são associados metadados que necessitam ser carregados e checados durante a execução do programa, uma enorme quantidade de classes causariam uma queda no desempenho do código produzido.

Visando diminuir o número de classes geradas e conseqüentemente a queda de desempenho o projeto Haskell .NET [**5**] utilizou a abordagem da construção de classes pré-definidas para closures com *n* variáveis livres e adotou um mecanismo de pilha para a passagem dos argumentos. Neste, ao invés de ser gerada uma nova classe para representação de cada closure, todas as closures que possuem a mesma quantidade de variáveis livres serão representadas através de instâncias de uma mesma classe pré-definida no ambiente de execução da linguagem. O que diferencia as diversas instâncias da mesma classe será a função armazenada, correspondente ao código da closure. No projeto Haskell .NET para o armazenamento desta função é utilizada um *delegate* ao invés de um ponteiro ou método abstrato.

O Código 8 mostra como criar uma closure para representar a função *f2* mostrada no Código 6. Nas linhas 2 e 3 é criado o delegate que armazena a função com o código de *f2*. Como será mostrado na Seção 2.4.2.1, utilizando o modelo *push/enter* o delegate não armazena diretamente a função com o código correspondente a expressão, mas sim, uma função auxiliar. As linhas 6 e 7 são responsáveis por construir a closure que representa a função. Pode-se observar que a classe utilizada para representar a closure possui um tipo genérico, este tipo genérico representa o tipo da variável livre armazenada pela closure, que neste caso é instanciado como sendo do tipo inteiro. Na linha 10 é configurado o valor da aridade da função. Este valor, como será visto na Seção 2.4.2 é útil para definir se a aplicação da função é saturada ou não. Por último, na linha 13, o valor da variável livre é adicionado a closure.

```
1  //Delegate para a função
2  NonUpdCloFunction_1_FV<int> funcDelegate =
3      new NonUpdCloFunction_1_FV<int>(function);
4
5  //Criação da closure que recebe como argumento o delegate
```

```
6  NonUpdateableClosure_1_FV<int> closure =
7      new NonUpdateableClosure_1_FV<int>(funcDelegate);
8
9  //Configura a aridade da função
10 closure.arity = 1;
11
12 //Armazena o valor da variável livre
13 closure.fv1 = x;
```

Código 8. Representação de uma função utilizando closure e delegates

2.4.2 Mecanismo de aplicação de funções

A combinação de polimorfismo paramétrico, funções de alta ordem e aplicação parcial de funções gera um cenário onde em alguns momentos pode ser necessário efetuar a aplicação de uma função desconhecida em tempo de compilação. No Código 9, *f* representa uma função desconhecida, uma vez que não se sabe em tempo de compilação como se comportará tal função. Não é possível simplesmente aplicar *f* aos dois argumentos, pois não se pode afirmar quantos argumentos *f* espera receber e qual o retorno da aplicação. Esta pode ser uma função que recebe apenas um argumento, processa este e gera uma nova função que consumirá o argumento restante, ou mesmo, uma função que receba mais de dois argumentos e desta forma o resultado de *zipwith* é uma lista de funções.

```
1  zipWith :: (a->b->c)-> [a] -> [b] -> [c]
2  zipWith f [] [] = []
3  zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
```

Código 9. Exemplo de aplicação de uma função desconhecida

Para tratar a aplicação de funções desconhecidas em linguagens funcionais existem dois modelos: *eval/apply* e *push/enter*. A diferença básica entre os dois modelos é quem será o responsável por tratar em tempo de execução a aplicação da função, se a própria função chamada ou o código que faz a chamada. O uso de um destes mecanismos deve ser efetuado apenas para funções desconhecidas em tempo de compilação, caso contrário a função deve ser chamada normalmente, evitando assim um *overhead* desnecessário.

2.4.2.1 Modelo push/enter

No modelo push/enter a própria função será a responsável por, em tempo de execução, verificar a aridade[12] da função, o número de argumentos recebidos e decidir como deverá ser feita a aplicação da função. Neste modelo para cada função definida na linguagem duas funções devem ser geradas após a compilação. Uma, denominada *fast entry point* (FEP), contendo o código correspondente da função original e outra, *slow entry point* (SEP), com o código responsável por verificar a aridade e o número de argumentos, decidindo que atitude tomar. O processo executado pode ser resumido em duas etapas:

- **Push:** os argumentos passados para a função são empilhados (*push*) em uma pilha diferente da pilha de execução da CLR.
- **Enter**: é feita a chamada a função SEP que avalia a aridade da função e o número de argumentos presente na pilha e baseado nestas informações determina se o próximo passo será *a* ou *b*.
  a. Caso o número de argumentos presentes na pilha sejam suficientes, estes são desempilhados e a função FEP é executada retornando o valor da avaliação. Argumentos excedentes são mantidos na pilha para que possam ser consumidos posteriormente, provavelmente pelo retorno de FEP.
  b. Caso o número de argumentos presentes na pilha seja inferior à aridade, estes são desempilhados e utilizados para criar uma aplicação parcial que é retornada como valor da avaliação.

Haskell .NET utiliza esta abordagem criando pilhas diferentes para armazenar diferentes tipos de argumentos *boxing* e *unboxing*.

2.4.2.2 Modelo eval/apply

Neste modelo a responsabilidade sobre como tratar a chamada de uma função desconhecida fica a cargo do código que invoca a função (*caller*). Este código deve, primeiramente, avaliar (*eval*) a aridade e o número de argumentos e

[12] Aridade pode ser entendido como o número de argumentos que uma função espera receber para realizar sua funcionalidade.

então decidir qual a aplicação (*apply*) deve ser feita: chamar diretamente a função, caso o número de argumentos seja maior ou igual à aridade, ou criação de uma aplicação parcial a ser retornada, caso contrário.

Historicamente a grande maioria dos compiladores para linguagens funcionais *lazy* utilizam a abordagem push/enter, entretanto após estudos feitos por Marlow e Peyton Jones [**33**], que demonstraram uma ligeira vantagem do uso do modelo *eval/apply* em uma implementação do *Glasgow Haskell Compiler* (GHC), o modelo *eval/apply* tem ganhado espaço. Na plataforma .NET, ainda não existem estudos que apontem qual modelo apresenta melhor desempenho. Nesta plataforma, o uso do *eval/apply* teria como vantagem o uso direto da pilha da CLR como mecanismo de passagem de parâmetros, o que não é possível no modelo *push/enter* devido a restrições na manipulação direta da pilha impostas pela CLR. Entretanto, o modelo *eval/apply* pode gerar aplicações parciais desnecessárias, não geradas utilizando o *push/enter* [**33**]. F# e Nemerle são exemplos de utilização de *eval/apply* na plataforma .NET.

2.4.3 Representação de tipos algébricos

Na plataforma .NET não existe o conceito de tipos algébricos como em linguagem funcionais. O mais perto que há são as enumerações que permitem que se descreva um tipo através de um conjunto de constantes, entretanto enumerações não permitem o uso de argumentos. O uso de uma classe abstrata para representar um tipo algébrico e subclasses destas para representar as possíveis construções é uma das abordagens mais utilizadas em ambientes orientados a objetos [**34**,**35**,**36**]. Utilizando tal abordagem ListInt (Código 4) teria a seguinte representação em código C#.

```
1  public abstract class ListInt{}
2  public class Nil : ListInt {}
3  public class Cons : ListInt
4  {
5          public int val;
6          public ListInt list;
7  }
```

Código 10. ListInt C#

Variações polimórficas como a mostrada em List (Código 5) seriam facilmente traduzida utilizando para isto *generics*. Esta representação permite um mapeamento fácil e direto, entretanto peca quanto ao desempenho em operações de casamento de padrões. Operações estas bastante comuns na manipulação de tipos algébricos em linguagens funcionais. Tal queda de desempenho se deve ao fato do uso da instrução *isinst*[13] para testar se um objeto é da subclasse desejada.

O uso de um número inteiro *(tag)* para diferenciar os construtores de um tipo algébrico como proposto por Jones e Lester[**37**] fornece uma maneira de otimizar operações de casamento de padrões com tipos algébricos. Sendo assim, a classe abstrata passaria a ter um campo inteiro que armazenaria a *tag* e o construtor de cada subclasse deve preencher este campo com um valor diferente dos demais. Casamentos de padrões poderiam ser executados utilizando instruções *switch* sobre a *tag*, com mostrado a seguir:

```
1  switch (list.tag )
2  {
3       case tagNil:   // código correspondente a opção Nil
4       break;
5       case tagCons:  // código correspondente a opção Cons
6       break;
7       default:       // código correspondente a opção default
8       break;
9  }
```

Código 11. Casamento de padrão utilizando switch

2.5 Implementações Existentes

Tentativas de integração de linguagens funcionais a ambientes gerenciados tem sido feitas mesmo antes do surgimento da plataforma .NET. Projetos como Lambada[**26**], Pizza[**36**] são exemplos de tentativas de integração à *Java Virtual Machine* (JVM) que forneceram as bases para posteriores integrações com a plataforma .NET. Por ser multi-linguagens a plataforma .NET possui algumas características que favorecem esta integração, tais como um rico sistema de tipos e instruções que facilitam a implementação de outros paradigmas de linguagens, tais

[13] Instrução IL, sua correspondente em C# é is.

como *.tail* que permite descartar o frame de execução em algumas chamadas recursivas, evitando desta forma o estouro da pilha de execução.

Como o foco deste trabalho é a integração de linguagens funcionas à plataforma .NET, nesta Seção serão apresentados apenas projetos desenvolvidos para este ambiente, de forma a demonstrar como tais projetos tratam os problemas e desafios de mapear estruturas e características funcionais na plataforma .NET.

2.5.1 Hugs for .NET

Hugs98 for .NET[**25**] é uma extensão do interpretador Haskell, Hugs98, que provê uma boa interoperabilidade entre o mundo Haskell e o mundo do framework .NET. Esta extensão permite que sejam instanciados objetos .NET dentro de programas Haskell e, vice-versa, permitindo a chamada de funções Haskell a partir de qualquer linguagem provida pelo framework .NET. Com isto o *Hugs98 for .NET* incrementa o potencial dos programas Haskell permitindo que eles façam uso das funcionalidades presentes na biblioteca da plataforma .NET.

Para fazer a interoperabilidade entre Haskell e a plataforma .NET, *Hugs98 for .NET* usa uma abordagem conhecida como *bridge*. Nesta abordagem o código Haskell não é compilado dentro de um *assembly* .NET contendo código MSIL o qual seria gerenciado pelo ambiente de execução .NET. O que ele faz é interpretar as instruções lado a lado com o ambiente de execução .NET, provendo o código para ambos os mundos através de chamadas de um mundo ao outro, utilizando uma biblioteca FFI .

Esta abordagem possui uma série de características que comprometem seu desempenho, dentre elas:

- Durante a execução de um programa que possui código dos dois mundos são mantidos dois ambientes de execução: o interpretador Hugs e *runtime* .NET. Dentre outros custos, temos o de manter dois coletores de lixos, um em cada ambiente.

- Para acessar o modelo de objetos .NET é utilizada a API de Reflexão. Trabalhos, como Rail[**38**], que utilizaram esta API relatam que ela possui

baixo desempenho. Outro problema decorrente de se utilizar esta API é que os objetos construídos por ela são acessados como componentes COM[**39**], que possuem certo custo para seu uso.

- O acesso ao código Haskell é feito através de invocação de código não gerenciado, o que acarreta *overhead* na transição entre código gerenciado e código não-gerenciado.

Embora com esta abordagem, o *Hugs98 for .NET***,** consiga fazer uso das funcionalidades disponíveis na plataforma .NET em programas Haskell, inter-operando entre os dois mundos, ele está longe do ideal no quesito desempenho.

2.5.2 Mondrian

Mondrian[**35**,**7**] é uma linguagem funcional não estrita especificamente projetada para ambientes orientados a objetos, possuindo uma versão para a plataforma .NET. Pode ser visto como uma versão *light* de Haskell, contendo uma sintaxe mista entre Haskell e C#. Por ser uma linguagem criada especificamente para integração com ambiente OO, Java e .NET, possui comando nativos para criação de objetos, chamada a métodos e acesso a campos.

Quanto a sua implementação na plataforma .NET suas principais características são:

- Utiliza *push/enter* como modelo de aplicação de funções.
- Sua representação de *thunks* utiliza exceções, onde o consumo de uma closure não avaliada gera uma exceção que é tratada avaliando a expressão e retornando o resultado desta avaliação. Este valor é armazenado na closure para futuras chamadas.
- Sua compilação gera código C#, o qual é posteriormente compilado para código MSIL utilizando o compilador C# padrão da plataforma.

O mesmo projeto que construiu **Mondrian** desenvolveu, também, um compilador Haskell para .NET[**27**]. Este compilador usa o GHC, como *frontend,* o qual é responsável por fazer o *parser*, a checagem de tipos e otimizações do código Haskell, gerando uma saída no formato *GHC Core*. Utilizando uma

ferramenta, o código GHC Core é, então, transformado em *Mondrian Core* que através do compilador *Mondrian* gera código .NET.

2.5.3 Nemerle

Baseada em ML, Nemerle[**9**] foi projetada para ser uma linguagem funcional estaticamente tipada voltada para a plataforma .NET. Outro objetivo levado em consideração no seu projeto foi permitir o uso de construções típicas de linguagens imperativas e orientadas a objetos de forma a promover uma boa transição de programadores destes paradigmas para linguagens funcionais. Esta característica também facilita a interoperabilidade com a plataforma .NET. Dentre suas funcionalidades se destaca o suporte a meta-programação que permite estender a linguagem através de macros. Embora seja estrita, permite criação de expressões com avaliação preguiçosa através do uso da palavra reservada *lazy*.

Sua implementação na plataforma .NET faz uso das seguintes estratégias:

- Adota o modelo de aplicação *eval*/apply utilizando para isto classes pré-definidas para *n* argumentos de tipos genéricos.
- Funções quando utilizada como valor de alta-ordem são representadas utilizando classes específicas. Esta classe deve estender da classe correspondente ao número de argumentos, dentre as classes pré-definidas no ambiente, e sobrescrever o método *apply* com o código correspondente, geralmente uma chamada para a uma função estática.
- Caso a função tenha variáveis livres, é criada uma nova classe onde estas são armazenadas e uma instancia desta classe é adicionada a um campo da classe que representa a closure da função.
- Funções não utilizadas como valor de alta ordem e que não possuam variáveis livres não geram closures sendo representadas diretamente como funções estáticas.
- Tipos algébricos são representados utilizando mecanismo de herança e casamento de padrões através da verificação de tipos com uso da instrução *isinst*.

2.5.4 F#e ILX

Assim como *Nemerle, F#* [**6**] é uma linguagem da família ML especialmente desenvolvida para integração com a plataforma .NET. Ela pode facilmente interoperar com qualquer linguagem .NET, bem como suas bibliotecas de classe. Ela também permite integração com *Caml*[**40**], possibilitando a importação de bibliotecas desta para a plataforma.NET. Por ter sido desenvolvida tendo como foco a integração com .NET, **F#** possui suporte sintático e semântico para a maioria das construções presentes no mundo .NET.

F# utiliza a ILX como código destino de seu processo de compilação a qual é posteriormente convertida em código IL. Entretanto, diferentemente do descrito por Don Syme[**6**] no trabalho que apresenta a ILX e da versão baixada através do site do produto [**41**] o código gerado não faz uso de ponteiro para referenciar funções em sua representação de closure. O que demonstra que a ILX vem sendo evoluída em conjunto com o F#. Devido ao uso do ILX como código final as características aqui descritas, observadas através da utilização do compilador F#, provavelmente são providas pela versão atual da ILX e não diretamente pelo F#:

- Modelo de aplicação *eval*/apply, com classes pré-definidas para aplicações otimizadas de até cinco argumentos de tipos genéricos.
- Da mesma forma que Nemerle (Seção 2.5.3) funções de alta ordem estende de uma das classes pré-definidas sobrescrevendo o método *Invoke* com o código correspondente, geralmente com uma chamada para uma função estática.
- Utiliza mecanismos de *inline* de código evitando a criação de novas closures e desta forma diminuindo o número de classes geradas.
- Permite a execução de funções provenientes de outra linguagem como função de alta ordem, através de um mecanismo implementado utilizando delegates.
- Caso a função tenha variáveis livres é criada uma nova classe onde estas são armazenadas e uma instancia desta classe é adicionada a um campo da classe que representa a closure da função.

- Funções não utilizadas como valor de alta ordem e que não possuam variáveis livres não geram closures sendo representadas diretamente como funções estáticas.
- Tipos algébricos são representados utilizando mecanismo de herança e casamento de padrões através da verificação de tipos com uso da instrução *isinst*.

A geração de código verificável, decorrente do abandono do uso de ponteiros, e outras características aqui apresentada demonstra um amadurecimento no projeto da ILX. A disponibilização desta nova versão facilitaria o surgimento de novas implementações de linguagens funcionais na plataforma .NET, bem como a interoperabilidade entre estas. O projeto ILX serviu como base para a prototipagem e testes da implementação de generics para a CLR, o que demonstra a importância deste dentro do projeto .NET sugerindo que novas características, tais como closures, possam vir a ser integradas em futuras versões da CLR.

2.5.5 Haskell .NET

O projeto Haskell .NET[**5**] faz alterações no compilador *Glasgow Haskell Compiler* (GHC)[**42**] criando um novo *backend* capaz de gerar código MSIL. Este *backend* tem como entrada uma representação intermediária do programa, produzido pelo *frontend* do GHC, na linguagem *Spineless Tagless G-Machine* (STG)[**28**,**43**]. Utilizar esta representação facilita o processo de compilação, pois toda a checagem de tipo fica a cargo do *frontend* e também se aproveita de otimizações feitas em etapas anteriores a sua produção.

Sua implementação possui inúmeras peculiaridades que objetivam otimizar o mapeamento de uma linguagem funcional não estrita, como haskell na plataforma.NET:

- Representa closures utilizando classes pré-definidas para *n* variáveis livres de tipo genéricos e delegates para fazer referência à função. A função referenciada pelo delegate corresponde ao *slow entry point*, o qual busca os argumentos na pilha de argumentos. Desta forma evita a geração de um

grande número de classes, como ocorre quando se utiliza a estratégia de uma classe por closure.

- De forma a permitir que tipos *unboxed* sejam passados como argumentos, sua implementação para a pilha de argumentos é divida em quatro pilhas correspondente aos tipos inteiro, *double*, object e closure. Diferentes valores são convertidos para o tipo que mais se aproxima.
- Para representação de tipos algébricos existem classes genéricas pré-definidas no ambiente capazes de representar construtores com até nove tipos variáveis.
- Utiliza um número inteiro como *tag* para identificar construtores e assim otimizar operações de casamento de padrões através de instruções *switch*.
- Com objetivo de evitar a criação de várias instâncias de valores comuns em tempo de execução o próprio ambiente de execução pré-instancia alguns valores booleanos e inteiros e os compartilha sempre que necessários.

O foco deste projeto foi dado à otimização do mapeamento das estruturas funcionais na plataforma .NET, desta forma, conversão de tipos e mecanismos que facilitassem a interoperabilidade com linguagens não funcionais, presentes no ambiente, não foram implementados.

2.6 Considerações Finais

Neste capítulo foram descritas algumas das principais construções características a linguagens funcionais, que ao mesmo tempo em que incrementam o poder de expressão destas dificulta a implementação em ambientes orientados a objetos como o .NET. Possíveis alternativas para o mapeamento de cada uma destas construções apresentadas e discutidas. Por fim, foram apresentados exemplos de implementações, explicitando a abordagem tomada por cada projeto. A **Erro! Fonte de referência não encontrada.** mostra um resumo das principais características encontradas nas implementações analisadas neste capítulo.

Tabela 1 - Comparação entre implementações

Projeto	Estratégia	Tipos Algébricos	Closures	Aplicação de Funções	Polimorfismo Paramétrico	Avaliação Preguiçosa	Otimizações
F#/ILX	• Compilação	• Classe abstrata para definir um tipo algébrico e subclasses para definir os construtores. • Casamento de padrões utiliza *isinst* • Faz otimizações para tipos sem parâmetros	• Uma classe por closure. Cada closure estende uma classe pré-definida para o número de argumentos específico. • Expressão é mapeada dentro da função Invoke	• Modelo eval/apply	• Utiliza *generics*	• Linguagem estrita, sem avaliação preguiçosa	• *Inline* de funções para evitar criação de novas closures • Não cria novas classes para funções não alta ordem
Haskell .NET	• Compilação	• Tipos pré-definidos utilizando *generics* e números inteiros como *tag* para uso em casamento de padrões	• Utiliza *delegates* para referenciar a função. Possui classes pré-definidas para *n* variáveis livres	• Modelo push/enter	• Polimorfismo através de herança do tipo Closure	• Utiliza closures para representar *thunks*. Estas closures são avaliadas apenas uma vez e só quando requeridas	• Compartilhamento de valores inteiros e booleanos • Simula atualizações *in-place*.
Hugs .NET	• Ponte (*Bridge*)	• Tipos são convertidos através de operações de *marshalling/unmarshalling*	• Não há conversão. Closures só existem no lado do Hugs	• Informação não encontrada	• Implementado separadamente	• Mecanismo interno do Hugs, do lado .NET não há avaliação preguiçosa	• Baixo desempenho. Sem otimizações específicas para integração com .NET
Mondrian	• Compilação	• Classe abstrata para definir um tipo algébrico e subclasses para definir os construtores • Casamento de padrões utiliza *isinst*	• Define uma classe por closure. Cada classe possui um método *Enter* que possui o código referente à expressão da closure. • Utiliza uma pilha de objetos para passar os argumentos	• Modelo push/enter	• Polimorfismo através de herança do tipo Object	• Utiliza closures que geram exceções quando são requisitadas pela primeira vez. A exceção é tratada através da execução da expressão e o retorno do valor resultante	• Biblioteca própria para execução de *threads* no ambiente .NET
Nemerle	• Compilação	• Classe abstrata para definir um tipo algébrico e subclasses para definir os construtores • Casamento de padrões utiliza *isinst*	• Uma classe por closure. Cada closure estende uma classe pré-definida para o número de argumentos específico • A expressão é mapeada dentro da função *Apply*	• Modelo eval/apply	• Utiliza *generics*	• Linguagem estrita, porém admite avaliação preguiçosa, desde que especificado explicitamente.	• Suporte a meta-programação que permite estender a linguagem através de macros • Não cria novas classes para funções que não são de alta ordem

3 PHOENIX FRAMEWORK

Phoenix[**44**] é um framework completo para construção de compiladores e de uma grande quantidade de ferramentas para análise, otimização e testes de programas. Sua estrutura é bastante flexível e está centrada na representação intermediária (IR) e na existência de diversos *readers* e *writers* que são capazes de ler e gerar código em diversos formatos. A função de um *reader* é ler de um formato específico (PE[14], MSIL, CIL[15]) e gerar uma representação intermediária a ser manipulada com o Phoenix. De forma contrária, um *writer* é o responsável por gerar um arquivo específico (PE, MSIL, COFF, etc.) a partir da representação intermediária.

Os compiladores atuais funcionam como caixas pretas, onde todo o processo interno é escondido do usuário e alterações em seu funcionamento não são permitidas. Tudo que o usuário pode fazer é fornecer o código fonte como entrada, passar algumas diretivas de compilação e aguardar a compilação do programa. Phoenix objetiva abrir esta caixa. Um compilador escrito utilizando Phoenix é formado por uma lista de fases, sendo cada fase responsável por uma etapa do processo de compilação. Através de um mecanismo, denominado *plugins*, Phoenix permite que seja alterado o comportamento do compilador acrescentando, retirando ou alterando fases. A existência de uma representação intermediária própria, bem como uma rica API para manipulação desta, facilitam a alteração do compilador e a construção de ferramentas de análise e otimização.

A Figura 2 dá uma visão geral da plataforma Phoenix, apresentando seus principais componentes: *readers*, *writers*, *Intermediate Representation* (IR), *Phases*, API e ferramentas (análise, instrumentação e otimização). Nela, podemos observar que o processo de manipulação da IR é feito durante as fases, utilizando ferramentas construídas com a API do framework.

[14] *Portable Executable* [**39**]. Padrão para arquivos executáveis do Windows.

[15] C/C++ *Intermediate Language*.

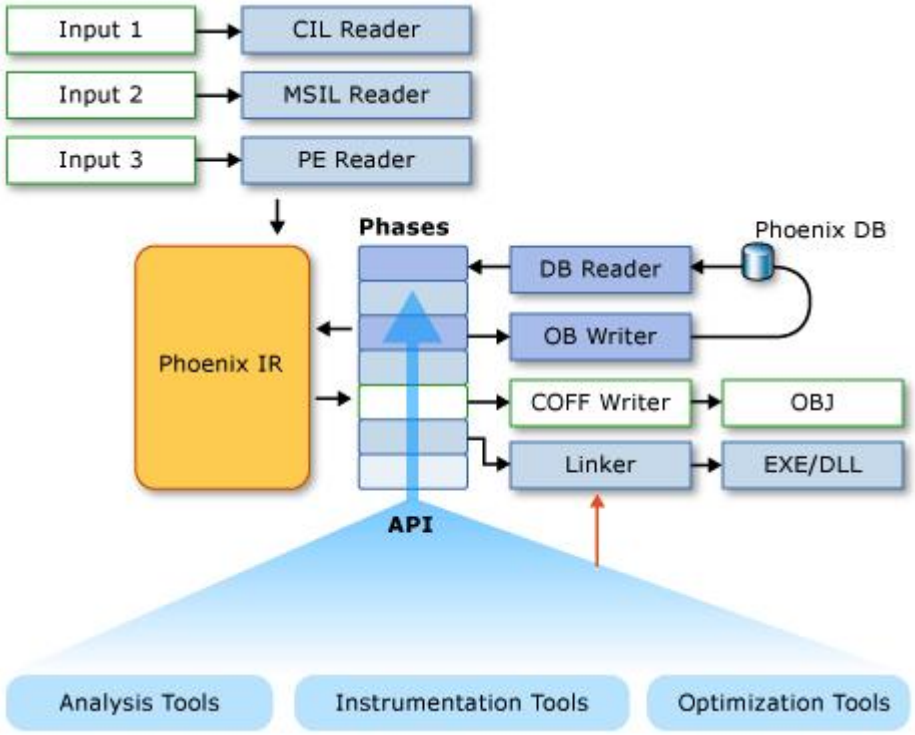

Figura 2. Visão geral da plataforma Phoenix. Adaptada da documentação do Phoenix[45].

Desta forma, Phoenix fornece um rico ambiente capaz de atender as necessidades tanto de pesquisadores como desenvolvedores. Aos pesquisadores é fornecida uma sólida infra-estrutura que suporta um modular reuso de código e o fácil redirecionamento para diferentes arquiteturas e linguagens. Assim, pesquisadores podem desenvolver novas ferramentas e elementos de compiladores sem o custo usual de ter de desenvolver uma nova infra-estrutura. Já desenvolvedores podem facilmente criar ferramentas para análise e otimização de seus programas, bem como, alterar o comportamento de programas já compilados sem ter que alterar diretamente o código.

3.1 Representação Intermediária (IR)

Phoenix utiliza uma representação intermediária fortemente tipada e linear para representar o fluxo de instruções de uma função. É sobre esta representação que é feita a manipulação de um programa utilizando a biblioteca de classes Phoenix. Para um programa ser reescrito utilizando o Phoenix, primeiramente, este deve ser convertido para a IR por um *reader* (*readers* para código nativo, MSIL e

AST[16] já são fornecidos pelo Phoenix, e outros podem ser escritos para formatos não suportados). Após a conversão a IR pode ser manipulada por uma ferramenta *Phoenix* e ao final do processo convertida novamente em um programa utilizando o *writer* específico. Desta forma, entender como é estruturada a IR é essencial para a construção de ferramentas e compiladores utilizando *Phoenix*.

A IR permite que uma função seja representada em diversos níveis de abstração, podendo representar uma função desde uma forma independente de máquina, alto nível, até uma forma dependente da máquina alvo, baixo nível, onde peculiaridades específicas como manipulação de registradores e pilha são descritas. Existem quatro níveis de representação providos por Phoenix, em ordem crescente de dependência: *high-level* IR (HIR), *mid-level* IR (MIR), *low-level* IR (LIR) e *encoded* IR (EIR).

A IR pode ser dividida em conjunto de conceitos básicos, cada um sendo representado por uma classe na API do Phoenix:

- Instruções e Operandos: representam respectivamente operações e recursos descritos através da IR.
- Tipos e Símbolos: conceitos básicos para definir o armazenamento e a referência dos dados manipulados.
- Unidades: são como containeres para o armazenamento dos demais elementos da IR.
- Classes Auxiliares (*Safety*, *Debug*, *Alias* e *Constant*): auxiliam na construção e manipulação da IR e na análise do código gerado.

As três primeiras categorias são essenciais para entender como construir um compilador utilizando o Phoenix e por isto serão detalhadas a seguir.

3.1.1 Instruções

Phoenix armazena a IR de uma função como uma lista de instruções duplamente ligadas, onde cada nó é uma instrução constituída de um operador (representado por um *opcode*) e duas listas de operandos: uma contendo os

[16] *Abstract Syntax Tree.*

operandos de origem e a outra com os de destino, como mostrado na Figura 3. Esta representação mostra de forma explícita todos os efeitos colaterais possíveis de uma instrução, uma vez que, todos os recursos lidos aparecem na lista de origem e todos os recursos potencialmente alterados estão especificados na lista de destino, favorecendo a análise destas instruções.

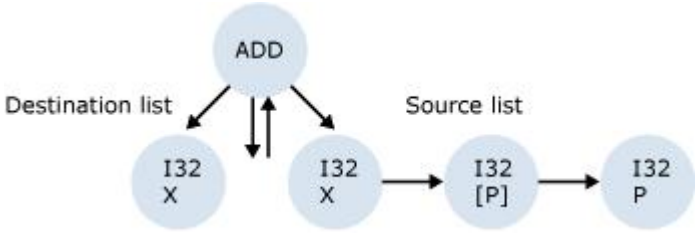

Figura 3. HIR da instrução x = add x, *p. Adaptada da documentação do Phoenix[45].

As instruções são classificadas em pseudo-instruções (label, pragma e data) e instruções reais (*value*, *call*, *compare*, etc.). Pseudo-instruções representam elementos tais como *labels* para fluxo de controle, *pragma* diretivas e alocação estática de dados. Embora pseudo instruções não sejam mapeadas para código de máquina elas são úteis para executar análise de código, passando diretivas para as unidades de compilação e identificando seções de dados.

- **LabelInstruction**: cria labels definidos pelo usuário e pontos de ligação para o fluxo de controle. Pode ser usado para determinar locais do código úteis para a criação de ferramentas de análise.
- **PragmaInstrucion**: representam diretivas e dados fornecidos pelo usuário. Pode ser usado para suprir informações do usuário para uma ferramenta ou compilador criado.
- **DataInstruction**: cria dados estaticamente alocados. Pode representar qualquer coisa que pode ser codificado em formato binário, tal como dados do programa ou instruções.

Instruções reais são as que modificam dados ou o fluxo de controle de um programa. Estas instruções são mapeadas diretamente para uma ou mais instruções de máquinas. São elas:

- **ValueInstruction**: operação aritmética ou lógica que produz um valor.

- **CallInstruction**: procedimento de invocação, direto ou indiretamente de uma função.
- **CompareInstruction**: instrução de comparação de dois operandos. Baseado neste resultado podem ser gerados códigos condicionais.
- **BranchInstruction** e **SwitchInstruction**: instruções de controle de fluxo para desvios condicionais, incondicionais e de múltiplas alternativas.
- **OutlineInstruction**: instrução para retirada do fluxo principal de instruções, tal como um bloco de assembly inline.

Embora o tipo de instrução restrinja os possíveis tipos de operação, o que realmente determina a operação a ser executada é o *opcode*. Por exemplo: para fazermos o cálculo de uma expressão utilizamos uma *ValueInstruction*, mas é através do *opcode* que determinamos se será realizada uma soma (*add*), subtração (*sub*) ou outra operação qualquer para a qual exista um *opcode* correspondente.

Para cada operação mapeada pelo Phoenix existe um *opcode* correspondente e este deve ser utilizado com a respectiva instrução. Na documentação do Phoenix[**45**] é fornecida uma lista com todos os *opcodes* existentes na IR.

3.1.2 Operandos

Operandos aparecem tanto na lista de origem quanto na de destino de uma instrução sendo que cada operando é associado a uma única instrução. Uma vez que, todos os efeitos das instruções são representados explicitamente, operandos refletem todos os potenciais recursos usados. O que inclui registros, alocações de memória e códigos condicionais. Cada operando possui um tipo abstrato associado a ele, este tipo abstrato é, posteriormente, mapeado para um tipo de máquina quando a instrução que o contém for transformada em uma instrução LIR ou EIR.

Existem diferentes tipos de operandos, cada um responsável por representar um determinado recurso. Por exemplo, para representar uma variável, seja ela temporária ou não, é utilizado um operador do tipo *VariableOperand*. Da mesma forma, existem operandos específicos para representar recursos armazenados na

memória (*MemoryOperand*), constantes (*ImmediateOperand*), *labels* (*LabelOperand*) e símbolos para funções (*FunctionOperand*). Tendo como exemplo a instrução descrita na Figura 3 "x" é referenciada utilizando um *VariableOperand* e "**p*" através um *MemoryOperand*.

3.1.3 Tipos

Phoenix possui um sistema de tipos bastante abrangente capaz de suportar todos os tipos descritos no *Common Language Runtime* (CLR)[**3**], incluindo tipos genéricos, bem como herança simples e múltipla (C++). Por ser a IR fortemente tipada para cada símbolo ou operando criado seu tipo deve ser especificado.

O sistema de tipos do Phoenix disponibiliza não só diferentes tipos como também a possibilidade de criar novos tipos e definir regras para a checagem destes tipos. Desta forma, um compilador ou ferramenta pode criar um conjunto de tipos e provê regras customizadas para sua checagem. É possível expressar tanto tipos de alto nível como tipos à nível de máquina, sendo permitida a checagem de tipos nos diversos níveis da representação intermediária (HIR, MIR e LIR).

A classe abstrata *Phx.Types.Type* é a classe base para todos os tipos suportadas por Phoenix, compartilhando propriedades e métodos utilizados por estes. Um sistema de tipos Phoenix é representado por um conjunto de tipos armazenados em um objeto *Phx.Types.Table* e um conjunto de regras prescritas por um objeto *Phx.Types.Check*. Assim, ao criarmos um compilador ou ferramenta deve ser criada uma única tabela de tipos, a qual deve ser compartilhada por toda a ferramenta. Vale observar que esta tabela é particular para uma arquitetura alvo uma vez que cada arquitetura possui sua própria representação de tipos. Certos tipos, tais como tipos primitivos, são disponibilizados como propriedades da tabela, sendo criados automaticamente quando Phoenix gera a instância da tabela.

Dentro do sistema de tipos Phoenix há uma classificação dos tipos, independentemente de estes serem padrão ou definidos pelo usuário devem se encaixar em uma das classes de tipos pré-existentes. Tais classes de tipos possuem atributos e métodos característicos de um determinado conjunto de tipos, facilitando a construção e a checagem de tipos. Desta forma Phoenix define

classes específicas para representar tipos primitivos, ponteiros, arrays, tipos variáveis, campos, tipos agregados e funções.

O tipo função é peculiar, pois diferentemente do que possa parecer ele não é utilizado para representar um tipo função como o existente em linguagens funcionais. Ele é utilizado para descrever um protótipo de uma função definindo sua assinatura, ou seja, os tipos de seus argumentos e de seu valor de retorno. Este tipo é essencial para a construção de uma função na IR. A tabela de tipos possui um método *GetFunctionType*, o qual facilita a criação de tipos função que possuam até quatro parâmetros. Para funções mais completas deve se utilizar a classe *FunctionTypeBuilder*. O Código 12 demonstra como criar um tipo para uma função que recebe um argumento do tipo inteiro e não retorna nenhum valor.

```
1  // Criando o tipo utilizando o método GetFunctionType
2  typeTable.GetFunctionType(CallingConventionKind.ClrCall,
3        typeTable.VoidType, typeTable.Int32Type, null, null, null);
4
5  // Criando o tipo utilizando FunctionTypeBuilder
6  FunctionTypeBuilder builder = FunctionTypeBuilder.New();
7  builder.Begin();
8  builder.CallingConventionKind = CallingConventionKind.ClrCall;
9  builder.AppendArgumentType(typeTable.Int32Type);
10 builder.AppendReturnType(typeTable.VoidType);
11 // Retorna o typo função criado
12 builder.GetFunctionType();
```

Código 12. Criação do tipo função

Tipos que possuem membros tais como classe, interfaces e estruturas são representados através da classe *AggregateType*. Para representar os diferentes tipos agregados são utilizados meta-propriedades que especificam as diferenças funcionais entre tipos diferentes. Ou seja, a combinação de meta-propriedades é que descrevem qual tipo esta sendo modelado diferenciando, por exemplo, uma interface de uma classe ou mesmo a representação de uma classe em linguagens diferentes como C++ e as linguagens .NET. O Código 13 demonstra como criar um tipo agregado que representa uma classe MSIL.

```
1  Phx.Name classTypeName = Phx.Name.New(lifetime, strClassTypeName);
2  Phx.Symbols.MsilTypeSymbol classTypeSym =
3      Phx.Symbols.MsilTypeSymbol.New(peModuleUnit.SymbolTable,
4         classTypeName, 0);
5
6  AggregateType classType =
7      AggregateType.NewDynamicSize(typeTable, classTypeSym);
8
9  // Configurando metapropriedades
10 classType.IsPrimary = true;
11 classType.IsSelfDescribing = true;
12
```

```
13 // Adição de métodos e campos.
14 classType.AddMethod(methodSymbol);
15 classType.AddField(fieldSymbol);
```

Código 13. Criando uma classe MSIL

A um tipo agregado podem ser adicionados campos e métodos. Campos são criados através da classe FieldType e possuem propriedades específicas como tamanho e deslocamento (*offset)*.

Para representar tipos variáveis, Phoenix disponibiliza a classe *VariableType*, a qual foi criada especificamente para representar tipos genéricos MSIL. Tipos variáveis são sempre associados a funções ou classes as quais definem o escopo dentro do qual ele pode ser acessado, sendo este escopo o tipo genérico ou método genérico que introduz o tipo variável.

3.1.4 Unidades

Unidades representam containeres lógicos para o armazenamento da IR. Além de outras unidades, estas unidades armazenam fluxos de instruções, tabelas de símbolos e variáveis inicializadas.

- **GlobalUnit** - Unidade de compilação mais externa, contém uma lista de objetos ProgramUnits. Criada quando inicializamos a infra-estrutura Phoenix, armazena, entre outras coisas, as tabelas de símbolos e de tipos globais.
- **ProgramUnit** - Unidade de compilação correspondente a uma imagem executável, podendo ser um arquivo EXE ou DLL. Contém uma lista de *AssemblyUnits* e uma lista de *ModuleUnits*. A razão para conter duas listas é que arquivos *Win32* não são formados por *assembly* e desta forma um objeto *ProgramUnit* pode conter diretamente módulos que não estejam dentro de assemblies.
- **AssemblyUnit -** unidade de compilação de um *assembly* do *Framework .NET*. Contém uma lista de objetos *ModuleUnits*. Menor unidade de re-uso, segurança e versionamento.
- **ModuleUnit –** coleção de funções (*FunctionUnits*), que normalmente representam um programa ou um arquivo fonte. Pode conter *DataUnits*.

- **PEModuleUnit** – tipo especial de *ModuleUnit* que representa um arquivo PE, pode ser um arquivo executável Windows (EXE) ou uma biblioteca de *link* dinâmico (DLL).
- **FunctionUnit** – representa uma função e com seu fluxo de instruções. Unidade alvo da maioria das transformações proporcionadas pela lista de fases.
- **DataUnit –** coleção de dados relacionados tal como um conjunto de variáveis inicializadas ou o resultado da codificação de *FunctionUnit*. Provê dados necessários para processar uma unidade.

Estas unidades podem ser aninhadas formando uma estrutura hierárquica, onde o a unidade mais externa é a *GlobalUnit* (Figura 4).

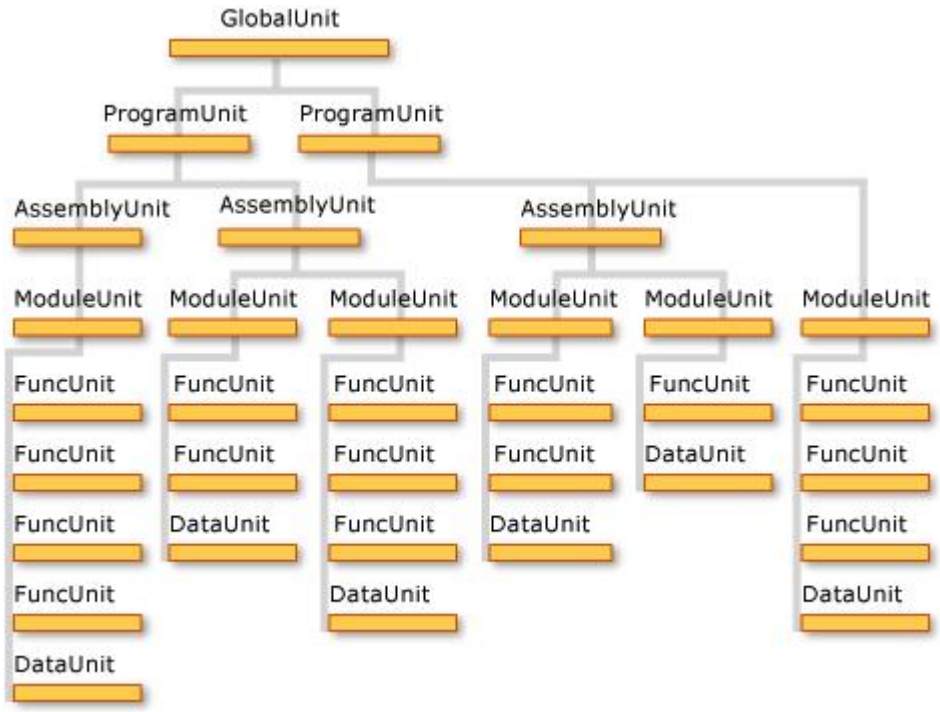

Figura 4. Hierarquia de unidades. Adaptada da documentação do Phoenix[45**].**

3.1.5 Símbolos

Símbolos Phoenix são associados a entidades tais como variáveis, *labels*, tipos, nomes de funções, endereços, entidades de metadados e módulos, fornecendo um nome para cada instância destes elementos. É o mecanismo através do qual tais entidades são referenciadas na IR. Estes símbolos são mantidos em tabelas que por sua vez são armazenados em unidades (Seção 3.1.4), devendo haver apenas

uma tabela de símbolos por unidade. Desta forma, a união de unidades e tabela de símbolos proporciona um controle sobre o escopo de um símbolo.

Para cada entidade a ser referenciada há um tipo correspondente e estes podem ser agrupados em:

- **Símbolos básicos** – símbolos que referenciam variáveis (locais e globais), funções, constantes, tipos, campos, *labels*, etc.
- **Símbolos que representam aspectos de módulos no formato PE** – módulos e variáveis importadas ou exportadas.
- **Símbolos para elementos de metadados da CLR** – assemblies, recursos, atributos, permissões, etc.

Uma tabela de símbolos não possui, por si só, nenhum mecanismo de busca. Para realizar uma busca numa tabela devemos associar a ela um mapeamento através de um objeto *Symbol.Map*, que permitirá fazer a busca na tabela utilizando como chave uma das propriedades do símbolo. Toda tabela possui pelo menos um mapeamento do tipo *IdMap*, o qual permite a busca na tabela através da propriedade *LocalId*, que é única para cada símbolo contido na tabela. *ExternIdMap* e *NameMap* são outros exemplos de mapeamento permitidos por Phoenix, sendo o último bastante útil pois permite a busca pelo nome do símbolo. A criação de uma tabela de símbolos e um mapeamento por nome pode ser observado no Código 14.

```
1  // Cria uma nova tabela de símbolos e associa a uma unidade
2  Phx.Symbols.Table funcSymTable =
3                  Phx.Symbols.Table.New(functionUnit, TABLESIZE, false);
4
5  // Cria um mapeamento por nome e o adiciona a tabela de símbolos
6  functionSymbolTable.AddMap(NameMap.New(funSymTable, TABLESIZE));
```

Código 14. Criação de tabela de símbolos e adição de um mapeamento por nome

É importante ressaltar que o tamanho tanto da tabela de símbolos como do mapeamento são fixadas no momento de sua criação, devendo estes ser grandes o suficiente para armazenar todos os símbolos que a ferramenta venha a necessitar ou deve ser feito um esquema que proporcione a expansão de seus tamanhos através da criação de uma nova tabela e novo mapeamento, de maior capacidade, e a cópia dos símbolos. O tamanho do mapeamento deve ser igual ou superior ao da tabela, para que este possa mapear corretamente todos os elementos desta.

3.1.5.1 Proxy

Proxy é um símbolo especial que permite que um mesmo símbolo apareça em mais de uma tabela de símbolo. Por exemplo, uma variável estática que é definida dentro de uma função usa um *proxy* para indicar que é tanto, logicamente, um membro do escopo da função como, fisicamente, uma variável global.

Um exemplo de quando se deve utilizar um *proxy* é quando uma instrução em uma *FunctionUnit* faz referência a uma variável global. Sabendo-se, que os operandos de uma instrução só podem referenciar símbolos na tabela de símbolos da unidade da função, para acessar uma variável global será necessário criar um *proxy* para esta variável na tabela de símbolos da função.

3.2 Fases e Plugins

Fases e *plugins* são estruturas que trabalham em conjunto, permitindo alterar o comportamento de ferramentas e compiladores, construídos com o Phoenix, sem que seja necessário alterar o código fonte destes.

Phoenix utiliza o conceito de *fases* para o processo de transformação de sua representação intermediária. Desta forma, um programa Phoenix é constituído por uma lista de fases, onde cada fase é responsável por uma característica específica do processo de compilação: transformação da IR, geração de código, otimização, alocação de registradores, etc. Uma fase atua sobre uma unidade, geralmente uma FunctionUnit, a qual representa uma função armazenando todos os símbolos e fluxo de instruções que compõem esta.

Plugins são módulos externos criados utilizando código gerenciado e armazenado em arquivos *dll*, os quais podem ser adicionados a programas construídos utilizando o Phoenix. Através deste mecanismo é possível modificar a lista de fases que compõe um programa Phoenix substituindo, alterando ou inserindo fases. Esta funcionalidade permite a modificação destes programas após sua compilação sem alterar seu código fonte.

A Figura 5 demonstra a utilização do *plugin MyPlugin.dll* que atua modificando o comportamento do compilador *cl* (compilador para código C/C++ construído utilizando o Phoenix). O compilador *cl* é dividido em dois módulos, o *frontend* (C1.exe) e o *backend* (C2.exe). O C2 é responsável pela geração de código final e foi construído utilizando o framework Phoenix. O *plugin* altera a lista de fases que compõem o *backend c2,* modificando assim seu funcionamento, o que pode ser refletido no programa gerado pelo compilador (App.exe).

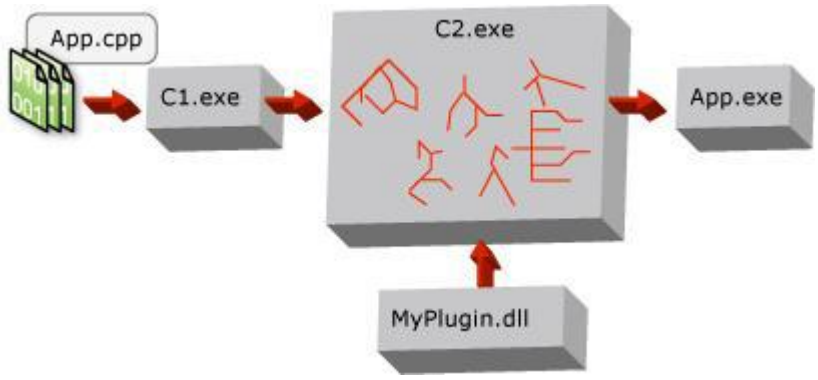

Figura 5. Funcionamento de um plugin Phoenix. Adaptada da documentação do Phoenix[45].

Com o uso de *plugins* fica fácil adicionar novas funcionalidades a um compilador. Para isto, basta identificar qual fase do processo de compilação proporciona representação e informações adequadas e através de um *plugin* inserir uma nova fase que execute a funcionalidade. O SDK[17] do Phoenix vem com um compilador C/C++ e um leitor de arquivos PE (PEReader), utilizando *plugins* é possível alterar o comportamento destes programas de forma a modificar o processo de compilação de códigos C/C++ ou obter informações de arquivos PE.

A construção de um *plugin* é bem simples, consistindo basicamente por duas etapas: construção de uma fase responsável por realizar a funcionalidade desejada e definição de uma posição na lista de fases onde esta será inserida. Para construir uma nova fase basta estender da classe *Phase*, criar um método construtor e sobrescrever o método *Execute* com o código correspondente ao trabalho a ser

[17] *Software Development Kit.*

realizado. O Código 15 demonstra a criação de uma fase (*MyPhase*), a qual descarrega o fluxo de instruções de uma função, fornecendo informações como *opcode* e operandos que compõem estas instruções.

```
1   public class MyPhase : Phx.Phases.Phase
2   {
3       public static MyPhase New(Phx.Phases.PhaseConfiguration config)
4       {
5           MyPhase phase = new MyPhase();
6           phase.Initialize(config, "Minha fase. Dump de instruções");
7           return phase;
8       }
9
10      protected override void Execute(Unit unit)
11      {
12          if (unit.IsFunctionUnit)
13          {
14              FunctionUnit funcUnit = unit.AsFunctionUnit;
15              foreach (Instruction instr in funcUnit.Instructions)
16              {
17                  instr.Dump();
18              }
19          }
20      }
21  }
```

Código 15. Construindo uma fase

Após a construção da fase, o plugin pode ser criado estendendo a classe *PlugIn* e sobrescrevendo os métodos *RegisterObjects* e *BuildPhases*, sendo este último o responsável por definir onde a nova fase será inserida. No Código 16 é exemplificada a construção de um plugin, o qual insere a fase *MyPhase* na lista de fases de um programa Phoenix após a fase de criação da IR. A implementação do método *RegisterObjects* é opcional, servindo para registrar controles que modificam o comportamento do *plugin*. É ainda necessário sobrescrever a propriedade *NameString* a qual deve retornar o nome do *plugin* criado.

```
1   public class MyPlugIn : Phx.PlugIn
2   {
3       public override void RegisterObjects() { }
4       public override void BuildPhases(
5           Phx.Phases.PhaseConfiguration config)
6       {
7           Phx.Phases.Phase encodingPhase;
8           Phx.Phases.Phase myPhase;
9           encodingPhase = config.PhaseList.FindByName("RaiseIR");
10          myPhase = MyPhase.New(config);
11          encodingPhase.InsertAfter(myPhase);
12      }
13      public override string NameString
14      {
15          get {return "MyPlugin"; }
16      }
17  }
```

Código 16. Construindo um Plugin

Utilizando o *pereader*, programa para leitura de arquivos PE fornecido junto com o SDK do Phoenix, é possível testar o funcionamento do *plugin* criado. Basta para isto executar o seguinte comando, descrito abaixo, substituindo <arquivoPE> por qualquer programa ou biblioteca .NET sobre o qual se deseja executar o *plugin*. Com isto é feita uma alteração na lista de fases do programa *pereader*, passando este a executar a fase *MyPhase*, logo após a fase *RaiseIR*.

pereader –plugin:myplugin.dll <arquivoPE>

A Figura 6 mostra o resultado obtido aplicando o *plugin*, com auxílio do *pereader*, sobre o clássico programa *HelloWorld*.

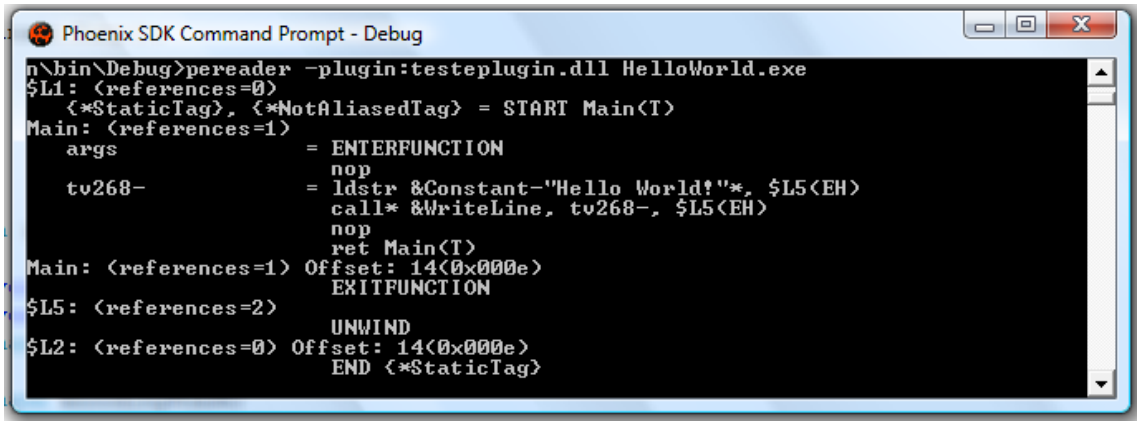

Figura 6. Dump HelloWorld

3.3 Gerando Código

O framework Phoenix é estruturado de forma a permitir a fácil geração de código para diversas arquiteturas (x86, x64 e CLR) por padrão e, também, facilitar a geração para novas arquiteturas através da *Grand Unified Retargeting Language* (GURL). A GURL é uma linguagem declarativa para descrição de instruções de máquinas utilizada, atualmente, apenas pela equipe de desenvolvimento interno do Phoenix. Entretanto, segundo Andy Ayers, gerente do projeto Phoenix, futuras versões do SDK devem incorporá-la.

O processo de geração de código é composto por diversas fases, com uma pequena variação para diferentes arquiteturas. Estas fases são responsáveis por transformar a representação intermediária de alto nível, adicionando

gradativamente informações da arquitetura alvo, até que seja gerada a representação codificada (EIR). A EIR pode então ser escrita em um arquivo utilizando o *writer* (PE ou COFF[18]) correspondente ao formato de arquivo a ser gerado.

3.3.1 Gerando código MSIL

Diferentemente da IR onde as instruções recebem diretamente os operandos sobre os quais a operação deve ser executada, as instruções MSIL operam utilizando uma pilha de execução. Phoenix faz esta conversão automaticamente durante o processo de transformação da HIR para EIR.

A fase *StackAllocation* é a responsável por fazer esta transformação nos operandos de forma que eles referenciem posições na pilha MSIL. Neste processo são utilizados pseudo-registradores que representam locais específicos na pilha. Por exemplo, o Código 17 demonstra como seria a representação de uma subtração em alto nível em baixo nível para uma máquina .NET. Na representação HIR **SR0** e **SR1** são pseudo-registradores onde a numeração representa a posição na pilha, sendo zero seu topo.

```
1  // Representação HIR
2  A.i32 = Subtract B.i32, C.i32
3
4  // Representaçõ LIR MSIL
5  T1.i32(SR0) = ldsfld B.i32
6  T2.i32(SR0) = ldsfld C.i32
7  T3.i32(SR0) = sub T1.i32(SR1), T2.i32(SR0)
8  A.i32 = stsfld T3.i32(SR0)
```

Código 17. Transformação HIR para LIR em máquina .NET

Além desta transformação a fase *StackAllocation* é responsável por:

- Calcular o tamanho máximo da pilha, informação esta necessária para a construção do cabeçalho de um método em código MSIL.
- Alocar espaço para variáveis locais e temporárias
- Gerar metadados com informações relacionadas às variáveis.

A geração automática de código MSIL pelo Phoenix permite que todas as otimizações feitas na IR sejam repassadas de forma consistente ao código final.

[18] *Common Object File Format.*

Desta forma, técnicas de otimização e ferramentas de análise podem ser criadas sem se preocupar em que arquitetura serão utilizadas.

3.4 Análise e Otimização

Phoenix fornece diversas bibliotecas que facilitam a criação de ferramentas de análise e otimização de programas. Estas bibliotecas tanto podem ser utilizadas dentro de fases do processo de compilação como na construção de novas ferramentas focadas na análise e otimização.

- **DataFlow** – implementa técnicas de análise de fluxo de dados que operam sobre a IR, tais como: *liveness* e *reaching definitions*.
- **Graphs** – fornece uma infra-estrutura para a construção de grafos que podem ser utilizados para representar fluxos de controle ou dados. Os grafos são direcionados (cada aresta possui um nó de origem e um de destino) e cada nó pode ser ligado a outro por mais de uma aresta.
- **Static Single Assignment (SSA)** – possui um conjunto de classes que facilitam a criação de representações SSA de um programa, bem como a análise e otimização baseada nestas representações. Dependências são modeladas utilizando um grafo SSA, onde as dependências são representadas como arestas entre operandos da IR.
- **Alias** – utilizado para rastrear o uso de memória feito pelas variáveis de um programa e modificações ocorridas nestas áreas decorrentes da execução das instruções de um programa.

O manual do Phoenix[**45**] fornece diversos exemplos práticos de como utilizar estas bibliotecas.

3.5 Considerações Finais

Os conceitos aqui apresentados dão uma visão geral de como construir um compilador utilizando o Phoenix e sua representação intermediária. Para tanto, inicialmente, é definida a hierarquia de módulos, a começar pela *GlobalUnit* a qual

conterá a tabela de símbolos globais e a tabela contendo os tipos a serem utilizado pelo compilador. Cria-se uma *ModuleUnit*, ou uma *PEModuleUnit*, caso se deseje gerar um arquivo PE, na qual serão adicionadas as *FunctionUnits* que representarão as funções presentes no programa a ser compilado. As variáveis criadas, utilizando símbolos e tipos correspondentes, deverão ser armazenadas no devido escopo, definido através da união entre tabela de símbolos e hierarquia de unidades. As instruções que compõem o programa poderão então fazer uso destas variáveis através dos operandos. Para finalizar, estas unidades serão submetidas a uma lista de fases responsáveis por tornarem a representação intermediária mais próxima da máquina alvo e por fim gerar o código.

Por fim, plugins e um conjunto de bibliotecas de análise de código como DataFlow, Graph, SSA e Alias fornecem uma rica infra-estrutura para análise e otimização do código gerado.

4 PROJETO E IMPLEMENTAÇÃO

O compilador aqui proposto busca, com auxílio da ferramenta Microsoft Phoenix, criar uma implementação de um compilador de uma linguagem funcional para a plataforma .NET que facilite o estudo e o desenvolvimento de novas técnicas de mapeamento de linguagens funcionais nesta plataforma. Neste capítulo serão descritos detalhes da implementação do compilador, bem como problemas e decisões de projetos.

4.1 Objetivos

Este projeto visa, com auxílio da ferramenta Microsoft Phoenix, criar uma implementação de um compilador de uma linguagem funcional .NET, que facilite o estudo e o desenvolvimento de técnicas de mapeamento de linguagens funcionais nesta plataforma. Com esta implementação objetiva-se, além de demonstrar a viabilidade de tal abordagem, desenvolver uma representação de um ambiente que contemple estruturas capazes de mapear características comuns a diversas linguagens funcionais na plataforma .NET. Com base nestes objetivos, ficam claros os seguintes requisitos:

- Gerar código MSIL a partir de uma linguagem representativa que contemple características mais relevantes de uma linguagem funcional.
- Compilar um *prelúdio* básico contendo funções necessárias para execução dos aplicativos selecionados para fazer a avaliação de desempenho.
- Facilitar a análise e otimização das estruturas responsáveis pelo mapeamento das características funcionais na plataforma .NET.

4.2 Arquitetura

O foco da implementação aqui proposta é dado à geração de código, análise e otimização (*backend*), desta forma preocupações quanto à análise léxica

e semântica do código são delegadas ao *frontend* a ser utilizado. O compilador desenvolvido tem como base a máquina abstrata *Spineless Tagless G-Machine* (STG)[**28**], a qual foi projetada para dar suporte a linguagens funcionais de alta ordem não estritas. Sua escolha se deve ao fato de fornecer estruturas semânticas simples capazes de representar as mais diversas construções características de uma linguagem funcional e por esta representação já ter sido amplamente testada e utilizada como formato intermediário em compiladores reais.

Como *frontend* será utilizada o *Glasgow Haskell Compiler* (GHC)[**42**], o qual é capaz de gerar, dentre outros formatos, código STG e CORE[19]. Embora internamente o GHC possua uma representação STG que contém informações sobre o uso e definição de tipos, o código gerado não as possui. Como tais informações são essenciais para uma implementação baseada em um ambiente fortemente tipado como .NET, o uso do código STG gerado foi descartado. Utilizar a representação STG interna, como feito em Haskell .NET, requer o uso de código Haskell o que dificultaria a abordagem proposta nesse trabalho que é utilizar framework Phoenix, uma biblioteca .NET, na construção do compilador. A alternativa encontrada foi o uso do arquivo CORE gerado, o qual mantém as informações de tipos necessárias. O uso da linguagem CORE seja como *backend* para novos compiladores [**27**,**46**] ou como alvo de transformações e otimizações[**47**,**48**] é bastante comum e tem seu uso sugerido pela equipe de desenvolvimento do GHC.

O uso do GHC como *frontend* não só garante que o código está correto como também permite a aplicação de uma série de otimizações, tais como *inlining*[**49**,**48**] e *strictness analysis*[**50**]. O processo de compilação do GHC (Figura 7) descrito por Peyton Jones et al. [**51**] pode ser resumido nos seguintes passos:

1. É feito o parser do código Haskell, gerando uma árvore sintática abstrata a qual em seguida tem seus tipos checados.
2. A árvore sintática é então simplificada (*desugaring*), gerando uma representação em linguagem CORE.

[19] *CORE* é uma pequena linguagem funcional produzida pelo compilador GHC que tem com intuito servir como linguagem alvo para novos *backends* e ferramentas de otimização que desejam utilizar o GHC como *frontend*. A definição da gramática e informações mais detalhadas sobre sua sintaxe é dada por Andrew Tolmach[**52**].

3. Otimizações opcionais, quando solicitadas através de linha de comando são feitas sobre a representação CORE.
4. A representação CORE é convertida para linguagem *Shared Term Graph* (STG).
5. A representação STG é convertida em uma representação interna denominada Abstract C, a qual pode gerar código C (quando solicitado código otimizado), ou código assembly.
6. Código nativo é então gerado utilizando um compilador C ou o Assembler.

O compilador aqui proposto, em destaque na Figura 7, não altera diretamente o GHC, ao invés disto utiliza como arquivo de entrada a representação CORE produzida utilizando a diretiva de compilação *-fext-core*.

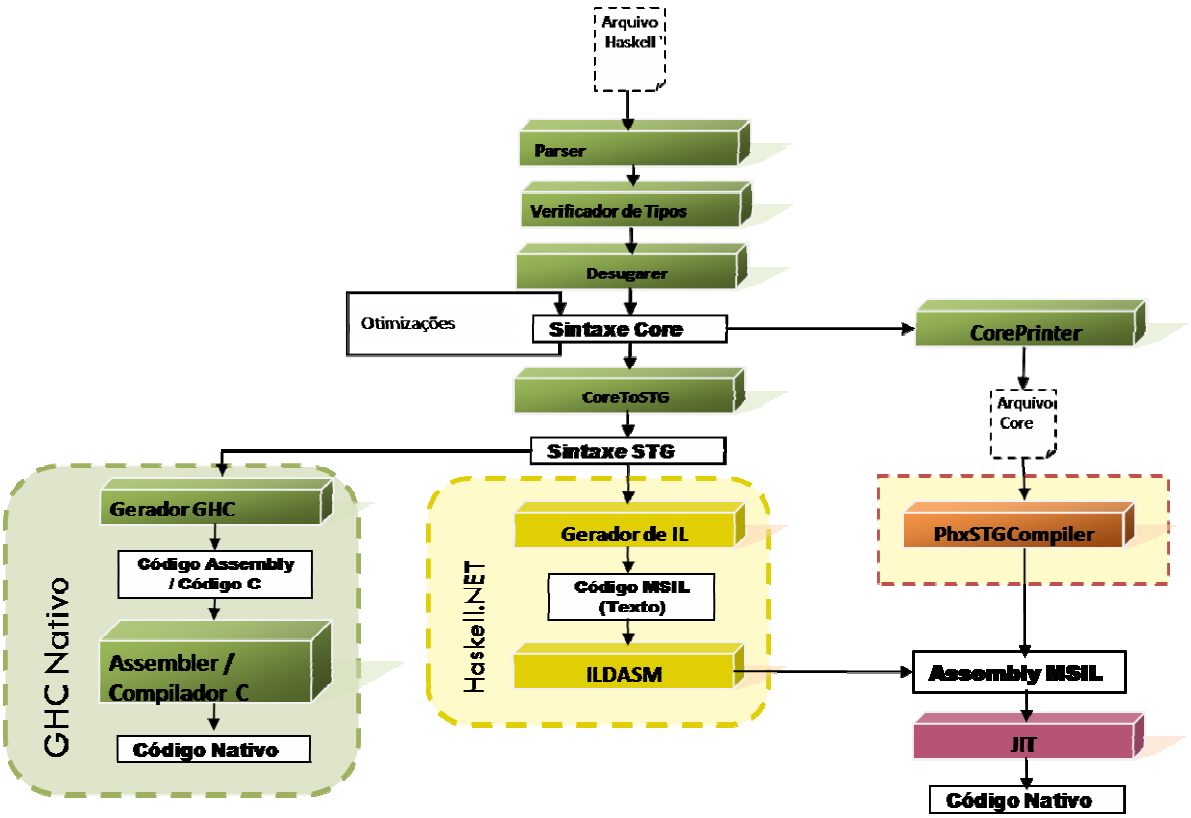

Figura 7. Inserção do PhxSTGCompiler

4.2.1 STG

A máquina STG fornece um conjunto de estruturas que, facilitam a representação de uma linguagem funcional de alto nível e que ao mesmo tempo são facilmente mapeadas para código nativo ou .NET. Seu modelo de execução é baseado na técnica conhecida como *graph reduction*, onde um programa é representado através de um grafo (neste caso uma árvore) e sua execução é feita reduzindo suas expressões a *Weak Head Normal Form* (WHNF)[20]. Os nós que compõem um grafo STG são os seguintes:

- *Progam* ou *module* – nó principal do grafo STG é composto por um conjunto *binds*.
- *Bind* – ligação entre uma variável, que identifica o *bind*, e uma abstração lambda (*lambda-form*).
- *Lambda-form* – representa uma função ou uma expressão atualizável. Explicita suas variáveis livres e argumentos.
- *Expression* – pode ser uma expressão binária sobre tipos primitivos, uma aplicação de funções e/ou construtores, uma expressão de casamento de padrões ou uma criação de binds locais através de uma instrução *let* ou *letrec*. Tais expressões são os alvos principais da redução.

Segundo Peyton Jones[**28**], criador da linguagem e da máquina STG, as principais características desta são:

- Todos os argumentos de funções e construtores são variáveis ou constantes. Esta restrição reflete a realidade operacional de chamadas de função onde seus argumentos devem ser preparados (seja construindo uma closure ou avaliando eles) antes da chamada. Esta restrição pode ser resolvida adicionando novas instruções *let* para a ligação de argumentos não triviais, como descrito na Seção 4.2.2.
- A aplicação de construtores e operadores primitivos (*built-in*) são sempre saturadas, ou seja, o número de argumentos esperado pelo construtor ou operador aplicado deve ser igual ao de argumentos fornecido.

[20] Termo criado por Peyton Jones[**37**] para explicitar a diferença entre *Head Normal Form* (HNF) e o que é produzido através da *graph reduction*.

- Casamentos de padrões são sempre executados através de expressões *case* e é permitido apenas padrões de um único nível.
- Existe uma forma especial de ligação (*binding*). Sua forma geral é:

  $$f = \{v_1,...,v_n\} \setminus \pi \{x_1,...,x_n\} \rightarrow e$$

  Através deste *binding* f é ligado a uma closure, que armazena as variáveis livres $v_1,...,v_n$ e a função $(\lambda x_1,...,x_n.e)$. O lado direito do *binding* é denominado *lambda-form* e é o único lugar onde uma abstração lambda pode aparecer. A *flag* π determina se a closure é atualizável, caso sua *flag* seja igual *u,* ou não atualizável caso seu valor seja *n*. O fato de a *lambda-form* permitir que as variáveis livres de uma abstração lambda sejam explicitadas faz com que não seja necessário o uso de técnicas de *lambda lifting*[21].
- Dá suporte a valores *unboxed*. Na STG, embora com algumas restrições, valores *unboxed* podem ser ligados a variáveis, passados como argumentos bem como serem retornos de uma função, armazenados e estruturas de dados, etc. Esta abordagem diminui o uso de *boxing*/*unboxing* durante operações de tipos primitivos.

4.2.2 Core to STG

A linguagem Core é facilmente traduzida para a STG de forma a ser utilizada na máquina abstrata STG. Algumas diferenças são apenas sintáticas, não necessitando grandes conversões, abaixo estão descritas apenas diferenças que exigiram modificações na máquina STG ou alguma análise prévia para identificação de informações relevantes.

1. Na STG os argumentos das funções devem ser atômicos (literais ou variáveis), diferentemente da linguagem Core, a qual permite que expressões sejam passadas como argumentos.
2. Aplicação de construtores e operadores primitivos tem de ser saturados. Embora a linguagem Core não possua nenhuma restrição quanto à

[21] *Lambda lifting* é uma técnica onde todas as definições locais de funções são elevadas para o nível definições globais transformando suas variáveis livres em argumentos extras [**87**].

aplicação não saturada destes elementos em sua especificação[**52**] é sugerido o uso de um pré-processador que torne tais aplicações saturadas.

3. Cada ligação *(bind*) é feita entre uma variável e uma *lambda-form*, a qual fornece explicitamente sua lista de variáveis livres. Core liga variáveis diretamente a expressões, sem se preocupar em explicitar suas variáveis livres.

A restrição *1* é resolvida, como proposto por Peyton Jones[**28**], adicionando novos *binds* através de uma instrução *let* responsável por ligar a expressão a uma variável a qual é utilizada para referenciar a expressão. Tomando como exemplo o Código 18, *testCore* é definido como a aplicação da função f1 que recebe uma expressão como argumento. Na STG isto não é permitido e por isto *testSTG* faz uso de uma expressão *let* a qual cria um *bind* ligando *t* à expressão *f2 2* e então aplica a função *f1* recebendo como argumento a variável ligada, no caso *t*.

```
1  testCore = f1 (f2 2)
2  testSTG = {} \u {} -> let t = f2 2 in f1
```

Código 18. Transformando uma expressão em um argumento atômico utilizando let

Para argumentos que correspondam à aplicação de operadores primitivos uma otimização pode ser conseguida utilizando expressões *case*, como definido em Peyton Jones e Launchbury [**53**]. Uma vez que tais aplicações resultam em tipos primitivos o qual não podem ser armazenados como thunks, a melhor abordagem é avaliar a expressão dentro de *case* e então retornar o resultado da avaliação através da alternativa *default* (Código 19). A mesma abordagem deve ser utilizada para aplicações de funções que retornam tipos *unboxed*.

```
1  testeSTG = {} \u {} ->
2        case 2+3 of var
3        {
4            default -> var
5        }
```

Código 19. Transformando uma expressão em um argumento atômico utilizando case

A forma direta para resolver a restrição 2 é utilizar o pré-processador Core, entretanto o pré-processador disponibilizado não condiz com a Core gerada pela atual versão do compilador GHC (6.8.2). Tim Chevalier, colaborador do projeto GHC, tem se esforçado em atualizar não só o pré-processador, como toda a linguagem Core gerada pelo GHC, de forma, a facilitar e ampliar o uso desta

linguagem. Entretanto, tais alterações só estarão presentes na próxima versão do GHC, ainda sem data prevista para lançamento. Uma possível alternativa é aplicar uma expansão-*n*, como sugerido por Peyton Jones[**28**], o que consiste em transformar aplicações não saturadas, de construtores ou operadores primários, em funções onde os valores fornecidos são considerados variáveis livres desta. A fórmula geral é dada abaixo, onde *c* é um operador interno ou um construtor de aridade *n* + *m*.

$$c\ \{e_1, ..., e_n\} \Rightarrow \lambda y_1 ... y_m\ .\ c\ \{e_1, ..., em, y_1, ..., y_m\}$$

Entretanto, para aplicar tal expansão é necessário que os módulos compilados guardem informações a respeito da aridade dos construtores, o que não era necessário para a compilação a partir da STG. A solução encontrada foi gerar para cada construtor uma função com código para aplicação do construtor, a qual guarda informações sobre sua aridade. Esta função não possui nenhuma variável livre e segue o mesmo modelo de avaliação de funções definidos na implementação do compilador, o que permite a geração de aplicações parcial quando aplicada a menos argumentos que o requerido. Para proporcionar melhor desempenho, a utilização desta técnica só é empregada quando observado o uso de aplicações não saturadas. Quando saturada, é feita a aplicação direta, criando um construtor ou aplicando a operação. Outro ganho obtido com esta conversão é permitir que construtores possam ser passados como parâmetros de uma função, uma vez que estes podem ser representados como uma função qualquer da linguagem.

O fato de não ter sido observada nenhuma aplicação não saturada de operadores primários na linguagem Core leva a crer que, na atual versão do GHC, tais aplicações são previamente expandidas. Desta forma, aplicações não saturadas de operadores primitivos não são tratadas na implementação aqui proposta.

Por fim, a transformação do lado direito dos *binds* em *lambda-forms* requer que duas operações sejam executadas: identificação das variáveis livres da expressão e adição da *flag* de atualização.

Uma variável é considerada livre se é mencionada no corpo de uma abstração lambda e não pertence nem ao seu conjunto de argumentos e nem ao

conjunto de *binds* globais do programa. Em nossa implementação tal identificação é feita ainda no *parser* da linguagem Core. Todas as variáveis referenciadas dentro da expressão, lado direito de um *bind,* são guardadas e posteriormente verificadas se pertencem ao conjunto de argumentos ou de *binds* globais, as que não correspondem são adicionadas ao conjunto de variáveis livres da *lambda-form*.

Quanto à *flag* de atualização, como descrito na própria definição da STG, é seguro configurar toda *lambda-form* como sendo não atualizável. Entretanto, tal atitude contradiz a definição da avaliação *lazy*, que diz que cada expressão deve ser avaliada somente quando necessária e apenas uma vez. Marcar toda *lambda-form* como não atualizável acarretaria em um gasto excessivo de processamento ao avaliar, desnecessariamente, uma mesma expressão mais de uma vez. Como definido pela STG, funções, aplicações parciais e construtores são consideradas não atualizáveis, sendo, apenas, *thunks* consideradas atualizáveis e mesmo estas, em alguns casos, podem ser não atualizáveis. Como regra geral, em nossa implementação consideramos *thunks* como sendo atualizável e separamos, ainda no *parser,* as expressões lambdas com e sem argumentos, sendo que as expressões com argumentos (funções e construtores) são sempre consideradas não atualizáveis. Já as sem argumentos são classificadas durante a compilação, onde se a expressão de for identificada como uma aplicação não saturada esta é tratada como uma closure não atualizável, caso contrário, será uma closure atualizável.

A fim de organizar e dividir melhor as responsabilidades, as transformações explicitadas nesta Seção deveriam ser delegadas a um pré-processador, o qual transformaria a linguagem Core numa STG enxertada com informações de tipos capaz de ser executada diretamente pelo compilador proposto. Entretanto, inicialmente, não foi cogitado o uso da linguagem Core como linguagem fonte. Esta só foi viabilizada na fase de integração com o compilador GHC, onde foi observado que a linguagem STG produzida não possui informações suficientes e a dificuldade em utilizar a representação STG interna em conjunto com o Phoenix. Com isto tal responsabilidade foi dividida entre o *parser* e o próprio compilador, cabendo ao primeiro a maior parte.

4.3 PhxSTGCompiler

O processo de compilação efetuado pelo *PhxSTGCompiler* pode ser observado na Figura 8. Inicialmente a linguagem Core fornecida pelo GHC é lida através de um *parser*, este gera uma representação abstrata do programa em forma de árvore a qual é convertida na representação intermediária IR, necessária para o uso do Phoenix. Utilizando uma lista de fases, construídas utilizando a API Phoenix, esta IR é sucessivamente manipulada e transformada em uma representação correspondente a requerida pela máquina alvo, neste caso a CLR. A última etapa deste processo de compilação corresponde à emissão do código final, a qual é feita através de um *writer* para arquivos PE, gerando uma biblioteca de link dinâmico (dll) ou arquivo executável (EXE).

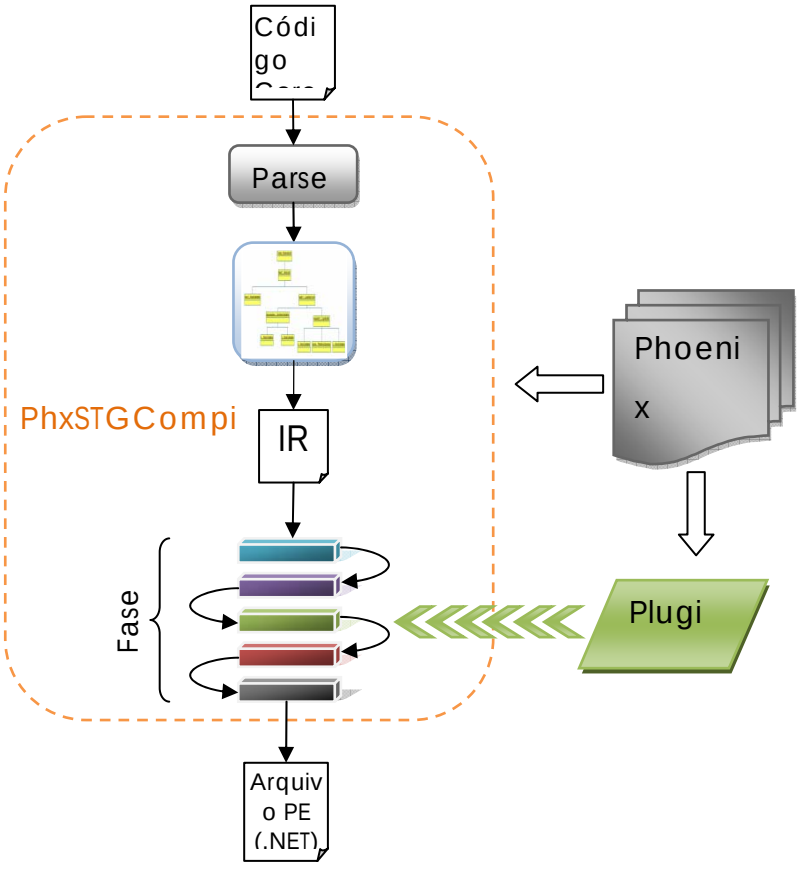

Figura 8. Processo de compilação

A implementação aqui proposta permite que seu processo de compilação seja alterado por programas externos, denominados *plugins*, os quais podem modificam a lista de fases do compilador. Este mecanismo será utilizado para produzir otimizações no código gerado, como demonstrado no Capítulo 5.

Internamente o *PhxSTGCompiler* é formado por um conjunto de classes responsáveis por representar estruturas de compilação, gerar a IR e pelo processo de compilação. Tais classes, representadas graficamente na Figura 9, são detalhadas a seguir:

- *Compiler*: responsável por inicializar e gerenciar a infra-estrutura Phoenix e as classes que compõem o compilador. Solicita o parser do arquivo fonte e a geração de código IR, o qual é então transformado em código MSIL através da execução da lista de fases definida no compilador. Ao final do processo de compilação emite o *assembly .NET*, podendo este ser um arquivo executável (EXE) ou biblioteca de classes (DLL).
- *CompilationEnvironment*: representa o ambiente de compilação, armazenando informações úteis ao processo de geração de código IR, tal como escopo e contagem de identificadores.
- *CompilationUnits*: coleção de classes que representam as estruturas básicas de compilação presentes na descrição da STG. Cada objeto desta classe armazena uma referência para um mesmo objeto da classe IRBuilder, compartilhado por todas as unidades do programa, a qual é utilizada para gerar o código IR. Todas as classes deste pacote herdam da classe CompilationUnit, unidade básica de compilação, que define um método abstrato o qual deve ser implementado em cada classe de forma a gerar, com auxílio do IRBuilder, a representação correspondente em código IR. Detalhes sobre a geração de cada uma das unidades pode ser observado no Apêndice A, de forma geral tais unidades podem ser classificadas em:
    - *BasicUnits*: unidades básicas de compilação (*module*, *bind*, *dataUnit* e *lambda-form*). Utilizam *Generate* para gerar seu código IR.
    - *ExpressionUnits*: representam as expressões disponíveis na máquina STG (*let*, *case* e aplicação de funções, construtores e operações sobre tipos primitivos). Disponibilizam o método *Evaluation,* responsável não só por gerar o código IR da expressão, como também retornar operando de destino da expressão.
    - *AtomUnits*: expressões atômicas (variáveis, construtores e tipos primitivos). Através do método *Evaluation* geram código IR,

quando necessário, e retornam um operando correspondente a sua representação na IR.

- *AlternativeUnits*: alternativas possíveis em uma expressão case. Podem operar sobre tipos algébricos ou primitivos. Possuem dois campos, um que armazena o valor da alternativa e outro para armazenar a expressão a ser executada caso seu valor seja selecionado. O código IR para a execução de sua expressão é gerado através do método *Evaluation*.

- *IRBuilder*: possui métodos responsáveis por gerar código IR, utilizando a API Phoenix. Disponibiliza um método *GetInstance,* o qual retorna sempre a mesma instância da classe, e deve ser utilizado sempre que se desejar obter uma instância desta classe. A utilização de uma única instância permite que informações sobre o código que está sendo gerado estejam sempre disponíveis aos métodos da classe.
- *Parser*: responsável por percorrer o arquivo fonte e gerar uma representação deste utilizando as unidades de compilação (*CompilationUnits*). Tal representação é semelhante a uma árvore onde cada nó é constituído por uma *CompilationUnit*.
- *Util*: possui funções que através de reflexão permitem obter informações de métodos e classes em bibliotecas .NET.

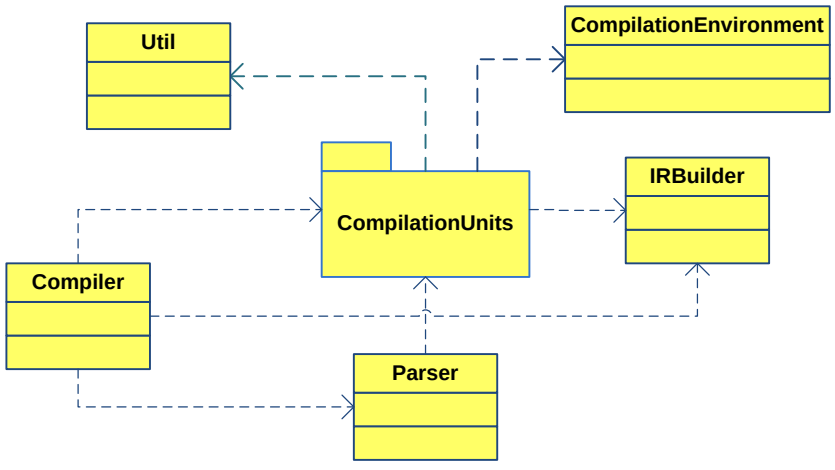

Figura 9. Arquitetura do compilador

Tendo como base o Código 20, uma representação da árvore gerada utilizando as unidades de compilação (objetos *CompilationUnits*) pode ser

observada na Figura 10. O processo de geração de código IR se inicia pelo nó raiz (*ModuleUnit*) o qual gera seu código e solicita aos nós filhos que façam o mesmo.

```
1  module Teste
2  func1 = {} \n {x,y} -> x+y
```

Código 20. Exemplo unidades de compilação

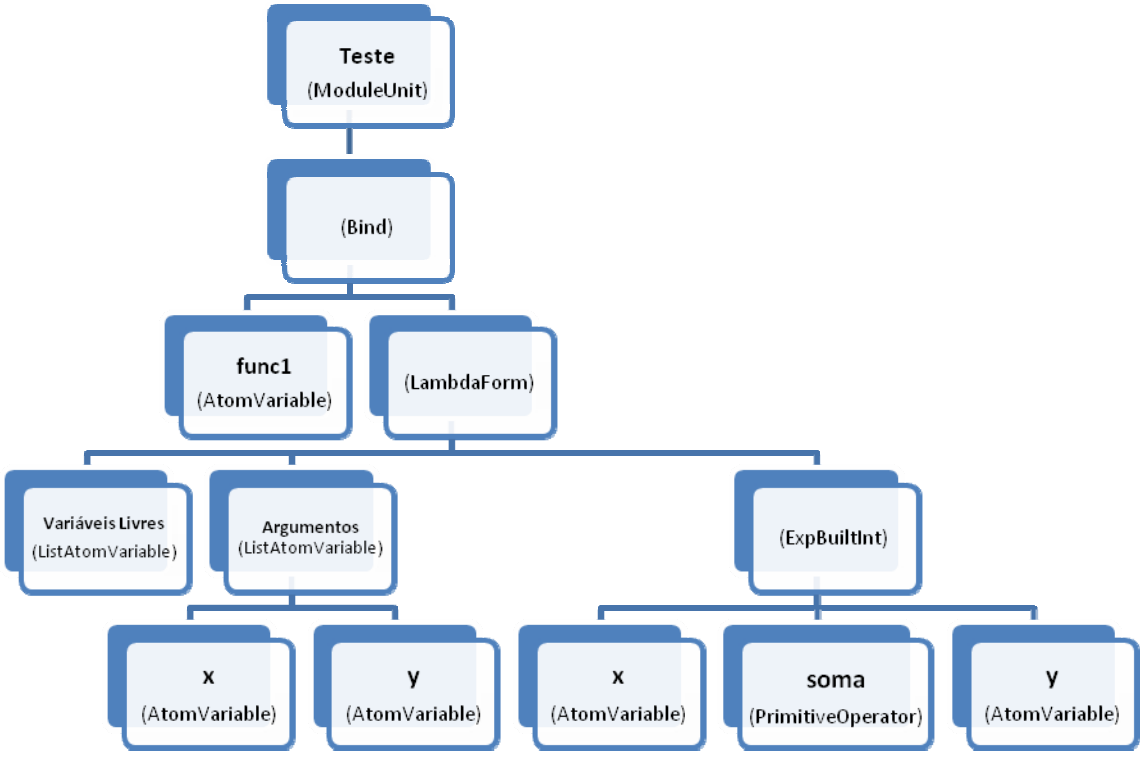

Figura 10. Árvore de compilação

4.3.1 Lista de fases

Efetuar a conversão da IR para código MSIL é um trabalho efetuado por uma lista de fases. Tais fases são responsáveis por gradativamente transformar uma IR de alto nível (HIR), independente da máquina alvo, para uma representação de baixo nível (LIR), dependente da máquina alvo, no caso em questão a CLR.

A lista de fases é construída dentro da classe *Compiler*, através do método *BuildPhaseList*. O mais usual é construir uma lista fases que opere sobre *FunctionUnits*, uma vez que estas unidades é que armazenam as listas de instruções. Entretanto, a

fim de permitir um maior controle sobre todo o código do compilador, neste projeto a lista de fases produzida opera também sobre a *ModuleUnit*. Para permitir que a lista de fases criadas operasse ao mesmo tempo sobre a *ModuleUnit* e sobre todas as *FunctionUnits* presentes nesta foi criada um tipo de lista de fases que opera especificamente sobre as *FunctionUnits*. Tal informação é importante para a construção de *plugins*, uma vez que, se estes desejarem operar sobre as *FunctionUnits*, deverão percorrer a primeira lista até encontrarem a outra lista e então atuar sobre esta.

A lista de fases criada pode ser observada na Figura 11. Ela é composta por três listas: a primeira que atua sobre *ModuleUnits*, a segunda que adentra a *ModuleUnit* e executa sobre as unidades existentes nesta e a terceira (*FuncUnitListPhaseList*) criada para selecionar apenas as *FunctionUnits*. Todas as fases padrão do compilador são adicionadas a esta última, pois elas atuam sobre as *FunctionUnits* transformando suas listas de instruções em código MSIL. Apenas a fase *VariableLocationPhase* não é implementada por padrão pelo Phoenix, esta foi codificada com objetivo de processar corretamente a assinatura das variáveis locais de um método, o que não era feito pelas fases fornecidas pelo Phoenix.

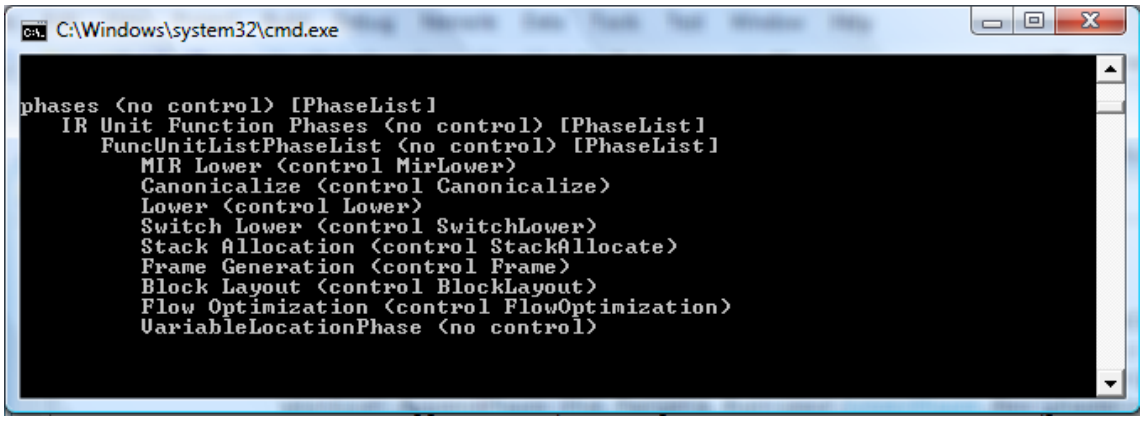

Figura 11. Lista de fases

Na fase de testes e otimizações (descrita na Seção 5) esta lista de fases é alterada, adicionando novas funcionalidades ao compilador, tanto diretamente como indiretamente, através de *plugins*.

4.3.2 Estratégia de compilação

Embora, utilizando o Phoenix não seja necessário manipular código .NET diretamente, e sim uma representação intermediária (IR), escolhas quanto à representação de cada uma das estruturas da linguagem devem ser feitas tendo em mente seu desempenho no código final. Aqui serão apresentadas quais estratégias foram utilizadas para a construção deste compilador, selecionada dentre as descritas na Seção 2.4.

Seguindo o modelo definido por Monteiro [**5**], o qual visa evitar a geração de um grande número de classes por programa, uma única classe é gerada por módulo, seja este um programa executável ou biblioteca de funções. Nesta abordagem para cada módulo compilado é gerado uma nova classe e o conjunto de *binds* presentes neste são compilados para funções estáticas e objetos de classes pré-definidas, os quais são armazenados em campos estáticos. Tais classes pré-definidas são utilizadas para representar *closures* com *n* variáveis livres, além de construtores com *n* argumentos.

Em linguagens funcionais *closures* são estruturas essenciais para a representação de objetos como funções e *thunks* na *heap*. Sendo assim, a forma como tal estrutura é definida influencia todo o restante do projeto do compilador. Na implementação aqui apresentada *closures* são construídas através de classes pré-definidas que utilizam *delegates* para referenciar a função correspondente a expressão e possui um conjunto de campos de tipos genéricos para armazenar as variáveis livres. Tendo como objetivo evitar a criação de uma classe por *closure*, estratégia utilizada por F# e Nemerle, é pré-definido um conjunto de classes para *n* variáveis livres, permitindo que novas *closures* sejam criadas através de novas instâncias da classe correspondente ao número de variáveis livres. O ambiente de compilação prevê a criação de closures com até nove variáveis livres. Embora nos testes realizados não tenha sido observado nenhum exemplo onde este número foi superado, *closures* com número superior a este são instanciadas utilizando uma classe especial onde as variáveis livres são armazenadas em um *array* de objetos do tipo *closure*. O uso desta classe deve ser evitado devido a custos no acesso aos valores do array e por não permitir o armazenamento de tipos *unboxed*. Uma representação das closures presentes nesta implementação pode ser observada na Figura 12.

O modelo de avaliação de funções adotado é o *push/enter*, o qual permite uma fácil representação de linguagens estritas na plataforma .NET. Embora, estudos realizados por Peyton Jones et al. [**33**] tenham demonstrado uma pequena vantagem a favor do modelo *eval/apply* na geração de código para uma linguagem estrita em ambientes não gerenciados, não foi encontrado nenhuma implementação que o mesmo ocorre no ambiente .NET ou em qualquer outro ambiente gerenciado. Dentre as implementações observadas apenas linguagens não estritas, como F# e Nemerle, implementam tal modelo na plataforma .NET. A implementação do modelo *eval/apply* na plataforma .NET permitiria o uso da pilha de argumentos da CLR como mecanismo de passagem de parâmetros, o que poderia acarretar um ganho no desempenho, entretanto aumentaria enormemente o número de classes pré-definidas pois seriam necessárias classes que combinassem um número *n* de argumentos a um número *m* de variáveis livres, o que resultaria em *n* x *m* classes.

Utilizando o modelo *push/enter* cada função definida é representada através de uma closure e dois métodos estáticos: *fast entry point* (FEP) e *slow entry point* (SEP). FEP possui o código real da função e é chamado sempre que todos os argumentos necessários estão presentes. SEP possui o código responsável por avaliar se todos os argumentos necessários à aplicação da função estão presentes na pilha, em caso positivo os desempilha e chama diretamente o FEP, caso contrário instancia uma aplicação parcial e armazena nesta os argumentos presentes na pilha. A closure instanciada referencia através de um *delegate* o método SEP o qual é executado através do método *Enter* presente na closure. A closure quando pertencente ao conjunto de *binds* globais do módulo é armazenada em um campo estático da classe e quando é instanciada através de uma expressão let é armazenada como variável local da função que engloba a expressão *let*.

Diferentemente de funções, *thunks* necessitam de apenas um método o qual armazena diretamente a expressão a ser executada. Esta expressão é avaliada apenas uma vez através do método *Enter* da closure, o qual verifica se a *closure* já foi avaliada, caso tenha sido retorna o valor armazenado, caso contrário chama a função referenciada pelo *delegate* e armazena o valor resultante para evitar futuras avaliações.

Tipos algébricos são representados utilizando classes pré-definidas, que herdam da classe *Pack*, e possuem *n* argumentos genéricos. A classe *Pack* possui um campo *tag*, o qual armazena um valor inteiro que é utilizado para identificar diferentes construtores. Para evitar que em um mesmo módulo existam dois objetos *Pack* com a mesma *tag*, este campo é preenchido utilizando o valor obtido através do método *GetHashCode* da string correspondente ao nome do construtor, o qual retorna um valor inteiro correspondente a *hash* do objeto. Casamento de padrões é implementado utilizando uma instrução *switch* que opera sobre a *tag* do construtor, o que é bem mais eficiente que através da verificação de tipos dos objetos. Na maioria dos casos novos construtores são instanciados diretamente, entretanto, em casos onde construtores são passados como argumento ou são aplicados parcialmente uma função responsável por gerar o construtor é criada e possíveis argumentos fornecidos são aplicados a esta.

Embora, a CLR permita a criação de funções polimórficas utilizando *generics* esta opção não foi utilizada para a representação de polimorfismo paramétrico no compilador aqui apresentado. Tal escolha se deve ao fato do GHC não permitir que tipos primitivos (*unboxed*) sejam utilizados como argumentos de funções polimórficas. Desta forma, o uso de *generics* não traria grandes benefícios, sendo tipos polimórficos representados através do uso da classe base *Closure*, a qual é a classe base para todos os demais tipos.

4.3.3 Ambiente de execução

Devido ao fato deste trabalho fazer parte do mesmo projeto, o ambiente de execução utilizado neste compilador segue, com algumas poucas alterações, o utilizado no projeto Haskell .NET. A descrição a seguir é fortemente baseada na feita por Monique Monteiro em sua dissertação: Integrando Haskell a Plataforma .NET[**5**], devendo esta ser consultada para um maior aprofundamento.

O ambiente de execução do PhxSTGCompiler consiste das classes pré-definidas que representam os diversos tipos de *closures* e das pilhas para passagem de parâmetros. Como mostrado no diagrama UML (Figura 12) a classe *Closure* é a classe base para a maioria das outras classes. Apenas *PAP* não herda de *Closure*, pois PAP por si só não representa um objeto manipulado diretamente pela STG,

devendo este ser associada a uma *closure* que representa uma função. *Closure* possui um método abstrato *Enter* o qual deve ser implementado por cada uma das classes que herdam desta com o código responsável por sua avaliação. As *closures* presentes no ambiente de compilação podem ser divididas em:

- Closures não atualizáveis (funções): mantém campos de tipos genéricos para o armazenamento de suas variáveis livres, um campo inteiro para o armazenamento da aridade e um campo *PAP* com valor *null*. Sua avaliação retorna uma chamada para o método SEP correspondente.
- Aplicações parciais: são *closures* não atualizáveis (funções) cujo campo *PAP* possui um objeto que armazena argumentos previamente recebidos. Sua avaliação, assim como de uma função, se dá através da chamada ao método SEP.
- Closures atualizáveis (*thunks*): expressões não avaliadas, as quais mantêm campos para o armazenamento de suas variáveis livres e um para armazenar o valor resultante de sua avaliação. Seu método *Enter* verifica se a closure já foi atualizada, em caso positivo apenas retorna o valor armazenado. Caso contrário é feita a avaliação e o valor resultante é armazenado.
- Construtores de dados: mantém campos genéricos para armazenar seus argumentos. Seu método *Enter* retorna ele próprio como resultado, uma vez que este se encontra na *Weak Normal Form* (WHNF), ou seja, na forma objetivada pela avaliação sob demanda[**37**].

No diagrama UML é possível observar os *delegates* responsáveis pela chamada dos métodos englobados por cada closure. Todos eles herdam da classe MultiCastDelegate e determinam a assinatura do método suportado. Existem delegates de dois tipos: *UpdCloFunction* utilizados para *closures* atualizáveis e *NonUpdCloFunction* para as não atualizáveis. Tal distinção se deve ao fato de *delegates* restringirem os métodos sobre os quais operam através de sua assinatura. Desta forma, um *delegate* do tipo *UpdCloFunction* suporta métodos que recebem como argumento um *UpdatableClosure* e retorna uma *closure* e um *NonUpdCloFunction* suporta métodos com um argumento do tipo *NonUpdatableClosure* retornando, também, uma *closure*. Assim como para as *closures* atualizáveis e não atualizáveis, são pré-definidos no ambiente variações destes para *n* variáveis livres.

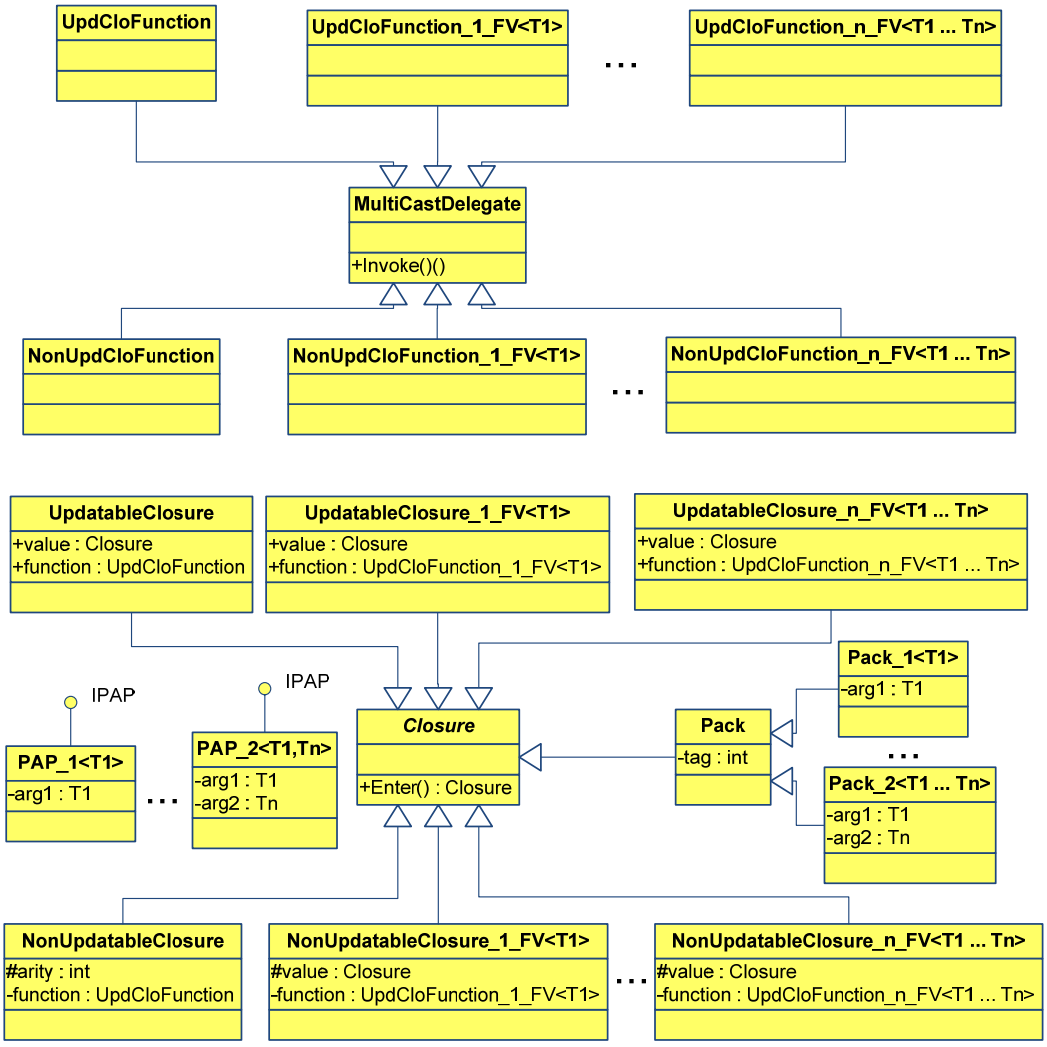

Figura 12. Ambiente de execução

Para a passagem de parâmetros, necessárias ao modelo *push/enter*, são utilizadas quatro pilhas que armazenam *closures*, inteiros, *double* e *object*. A razão para existência de mais de uma pilha é evitar operações de *boxed/unboxed* de tipos primitivos, permitindo que tipos primitivos sejam passados como parâmetros diretamente, otimização esta implementada pelo GHC seguindo a descrição dada por Peyton Jones et al.[**53**]. Outros tipos para os quais não haja pilha específica devem ser mapeados para uma das pilhas existentes, por exemplo: caracteres são armazenados na pilha de inteiros e tipos *float* na pilha de *double*.

4.4 Considerações Finais

Neste capítulo foi apresentada a arquitetura do PhxSTGCompiler, bem como, problemas e decisões de projetos enfrentados durante sua implementação. A idéia inicial de utilizar o código STG gerado pelo GHC como entrada se mostrou inviável e por isto uma alternativa foi apresentada: o uso da representação CORE. O uso desta representação requisitou que modificações fossem feitas tanto no parser como no próprio gerador de código.

A implementação atual possui um prelúdio reduzido, o qual é suficiente apenas para a compilação dos testes executadas no Capítulo 5. Outra restrição da implementação diz respeito à compilação direta a partir do código CORE. Por não estar disponível uma gramática atualizada da CORE gerada pela versão atual do compilador, eventualmente, foram necessárias intervenções manuais para que o código pudesse ser entendido pelo compilador.

O uso do Phoenix para geração do código final mostrou ser uma boa abordagem, pois permitiu que código .NET fosse gerado diretamente, sem a necessidade de manipulação de código MSIL. Pequenos problemas observados no código gerado pelo Phoenix serão discutidos na Seção 5.2.

5 ANÁLISE E OTIMIZAÇÃO

Neste capítulo serão apresentadas avaliações de desempenho do compilador construído, bem como otimizações implementadas utilizando o mecanismo de *plugins* e a API de análise e manipulação de código do Phoenix. Ao final as otimizações que obtiveram melhores resultados serão adicionadas ao compilador e o código gerado por este será comparado ao gerado pelos compiladores Haskell .NET e GHC.

5.1 Metodologia

Na avaliação do desempenho dos programas gerados, pelo compilador aqui apresentado, foi utilizado um subconjunto dos programas presentes no *benchmark* NoFib[**54**]. Mais especificamente, um conjunto de programas pertencentes ao grupo Imaginário[22]. Embora na documentação do NoFib seja sugerido o uso dos programas pertencentes ao grupo dos Reais, uma vez que este possui programas mais complexos que representam problemas reais, esta opção foi descartada devido a restrições do *prelúdio* compilado nesta implementação. Contudo, os programas do grupo Imaginário, embora menos complexos, representam problemas específicos e facilmente escaláveis, permitindo não só a validação do processo de compilação como a descobertas de possíveis gargalos que venham a denegrir o desempenho dos programas gerados.

O NoFib sugere, para os programas do grupo Imaginário, dois possíveis valores de entrada, bem como, os respectivos resultados esperados. Um valor para uma execução mais demorada e outro para uma execução mais rápida. Entretanto estes valores não correspondem à capacidade de processamento das máquinas atuais, o que resultou em baixos tempos de execução, mesmo para o valor que gera um maior processamento. Desta forma, os valores de entrada utilizados para

[22] Os códigos dos exemplos utilizados podem ser obtidos através do endereço http://darcs.haskell.org/nofib/imaginary/

os testes aqui apresentados são diferentes dos sugeridos e foram selecionados de forma a evitar tempos de execução demasiadamente curtos, onde o tempo de inicialização e carga do programa predomine sobre o de execução. Para comparação com o Haskell .NET e com gerador de código nativo do GHC foram utilizados os seguintes valores:

- Tak: 12, 1, 25
- Primes: 4500
- Queens: 13
- Exp3: 9
- DigitsE1: 2000
- WheelSieve1: 400000
- WheelSieve2: 80000

Em alguns exemplos foi necessário o uso de valores diferentes destes, quando isto ocorrer o valor utilizado será especificado entre parênteses.

A Tabela 2 mostra a configuração do ambiente utilizado nos testes realizados.

Tabela 2. Configuração do Ambiente

Característica	Valor
Processador	Intel® Core™ 2 - 7200 (2 x 2.0 GHZ)
Memória	2 GB
Sistema Operacional	Windows™ XP Professional
Versão da CLR	.NET™ Framework 2.0
Versão do GHC	GHC 6.8.2

Para obtenção dos tempos de execução foi utilizado o comando *time* presente no utilitário *cygwin*[23]. *Os* valores apresentados correspondem à média obtida a partir da execução de cada exemplo 10 vezes, retirados possíveis *outliers*[24], tendo seu valor expresso em segundos. Os *outliers* são removidos, pois,

[23] Emula um ambiente Linux no sistema operacional Windows. Pode ser obtido através do endereço: http://www.cygwin.com/

[24] Valores destoantes do conjunto de dados. Para o cálculo é utilizada a fórmula: $\{x | Q_1 - 1{,}5 * (Q_3 - Q_1) \leq x \leq Q_3 + 1{,}5 * (Q_3 -$ onde e corresponde, repectivamente, ao primeiro e terceiro quartil.

possivelmente, representam momentos onde foi necessária alguma compilação em tempo de execução ou consumo de recursos do computador por algum outro processo. Nas comparações entre os valores (impacto da otimização), onde são apresentadas porcentagens, valores positivos indicam uma melhora, enquanto que, valores negativos indicam piora.

Para análise do consumo de memória foram testadas diversas ferramentas, tais como o *CLR Profiler*, *Performance Monitor*, *AQTime 5*, entre outras. Entretanto, com exceção do *Performance Monitor*, todas as ferramentas requisitaram um enorme tempo para análise do código, mesmo para valores de entrada pequenos. Em muitos casos todo o espaço em disco foi consumido antes que fosse retornado qualquer resultado. Isto se deve ao grande detalhamento das análises executadas por estas ferramentas e ao intenso consumo de memória feito pelos programas testados. O *Performance Monitor*, embora não tenha apresentado problemas quanto ao tempo de execução ou uso de recurso da máquina, mostra resultados que correspondem a uma média de um curto intervalo de tempo, atualizado constantemente durante a execução do programa, não refletindo o perfil completo do programa.

A solução encontrada para traçar um perfil do consumo de memória foi a construção de uma ferramenta específica, utilizando contadores de desempenho fornecidos pela plataforma .NET. Esta solução permitiu que apenas os recursos desejados fossem monitorados, evitando uma demora excessiva para análise dos programas. O código da ferramenta de análise pode ser observado no Apêndice B. Os valores monitorados correspondem ao número máximo de bytes alocados dinamicamente, porcentagem do tempo gasto com a coleta de lixo e número de coletas em cada geração.

5.2 Código .NET Gerado Com o Uso do Phoenix

Os primeiros testes com o conjunto de programas compilados demonstrou algumas deficiências do código gerado. Nesta Seção serão apresentadas as deficiências resultantes de restrições na geração de código .NET com o Phoenix. Após uma breve explanação sobre o porquê de cada problema será apresentado

uma alternativa para resolução deste e os resultados obtidos após a implementação da solução.

5.2.1 Variáveis temporárias

O uso de variáveis temporárias para o armazenamento de valores resultantes de operações é uma técnica comum no desenvolvimento de um compilador. Utilizando o Phoenix há uma série de instruções, denominada *ExpressionIntructions,* que fazem uso desta técnica, tendo como operando de destino uma variável temporária. Estas variáveis temporárias, entretanto, devem, durante a geração de código, ser removidas ou substituídas por variáveis reais. Como exemplo, uma soma de dois valores e o posterior armazenamento do resultado em uma variável local é feita, utilizando o Phoenix, através do Código 21. Neste, a remoção da variável temporária poderia ser feita passando como operando de retorno da instrução *instAdd* o operando *varOpZ*.

```
1  // Instrução soma dois operandos e retorna uma variável temporária
2  Instruction instAdd =
3      Instruction.NewBinaryExpression(FuncUnitListPhaseList,
4         Phx.Common.Opcode.Add, varOpX, varOpY);
5
6  // Armazena o valor da variável temporária na variável real
7  // representada por varOpZ
8  Instruction instStore =
9      Instruction.NewUnary(FuncUnit, Phx.Common.Opcode.Store, varOpZ,
10        instAdd.DestinationOperand);
```

Código 21. Variáveis temporárias

O código MSIL utiliza uma pilha para armazenamento temporário de valores, não necessitando de variáveis temporárias para executar tal função. Desta forma, o processo de geração de código ao transformar da representação HIR para LIR deveria remover as variáveis livres, substituindo seu uso pelo uso da pilha. Entretanto a geração de código MSIL padrão do Phoenix não realiza esta substituição, fazendo com que algumas variáveis temporárias que deveriam ser removidas acabem sendo promovidas a variáveis reais no código gerado. A não remoção destas variáveis resulta em um código sujo, cheio de instruções desnecessárias. O Código 22, mostra as instruções MSIL geradas para o exemplo anterior, sem a remoção da variável temporária que armazena o resultado da adição.

```
1  // Código gerado com variável temporária desnecessária
2  ldloc.0 // varOpx
3  ldloc.1 // varOpY
```

```
4   add
5   stloc.2 // variável temporária
6   ldloc.2
7   stloc.3 // varOpZ
```

Código 22. MSIL sem remoção de variáveis temporárias

Além de um maior consumo de memória, necessário para o armazenamento das variáveis temporárias, as instruções geradas para armazenamento e leitura destas variáveis intermediárias dificultavam, a implementação de algumas otimizações no código, como a inserção de instruções *tail* (Seção 5.3.1). Para minimizar tais problemas foi inserida uma nova fase no processo de compilação. Esta fase é responsável por identificar variáveis temporárias na representação LIR, bem como as instruções que a manipulam, e as removê-las. Com o uso desta nova fase as instruções *stloc.2* e *ldloc.2*, presentes no Código 22, seriam removidas, bem como a variável local correspondente.

Tabela 3. Impacto da remoção de variáveis temporárias

Programa	Número de variáveis removidas	S/ Fase de remoção		C/ Fase de remoção		Impacto da remoção	
		Tamanho	Tempo	Tamanho	Tempo	Tamanho	Tempo
Tak	32	6.144	17,71	5.632	17,70	8,33%	0,05%
Primes	123	9.728	16,37	9.216	16,33	5,26%	0,24%
Queens(12)	155	12.288	02,82	11.264	02,78	8,33%	1,42%
Exp3	443	22.016	32,58	19.968	32,64	9,30%	-0,20%
DigitsE1 (1500)	330	23.040	30,43	20.992	30,65	8,89%	-0,72%
WheelSieve1 (50000)	747	40.960	00,87	40.960	00,87	8,16%	0,03%
WheelSieve2 (50000)	626	35.328	12,20	31.232	12,17	8,13%	0,03%
Média						**8,02%**	**0,16%**

A Tabela 3 mostra o impacto da remoção das variáveis temporárias promovidas pela nova fase, inserida no processo de compilação. Em todos os exemplos foi observada uma redução do tamanho (em megabytes) do programa, que ficou em média 8,02% menor. Esta diminuição no tamanho se deve

principalmente à remoção de instruções que, desnecessariamente, liam e armazenavam valores nestas variáveis. Quanto ao tempo de execução, foram observadas pequenas variações para mais e para menos, ficando na média praticamente inalterado. Resultados mais precisos poderiam ser observados para valores de entrada maiores, principalmente para programas que fazem grande uso de memória, como *DigitsE1* e *Queens e WheelSieve 1 e 2*. Entretanto, neste estágio do desenvolvimento do compilador a questão do estouro da pilha de chamadas recursivas e vazamentos de memória ainda não haviam sido resolvidos por completo, ocasionando estouro da pilha para valores maiores que os utilizados.

A contribuição desta nova fase vai além da redução no tamanho do programa e do ganho de desempenho em alguns programas. Ela removeu instruções que dificultavam a identificação de chamadas recursivas e pontos de inserção para instruções *tail* (Seção5.3.1), essências para o controle da pilha de chamadas em programas funcionais.

5.2.2 Casamento de padrões aninhados

No compilador aqui proposto expressões de casamento de padrões são implementadas utilizando instruções *switches*. Tais instruções são úteis para selecionar entre diferentes blocos de instruções, entretanto estas foram desenvolvidas com foco em linguagens imperativas onde tais blocos são compostos por um conjunto de comandos os quais podem alterar estados das variáveis, mas não retornam um valor. Já em linguagens funcionais este bloco corresponde a uma expressão, a qual após sua avaliação retorna um valor. Embora, pensando diretamente na CLR seja possível passar este valor através da pilha de execução, utilizando o Phoenix, mais precisamente a HIR, esta opção não é valida, pois não é permitido o manuseio da pilha diretamente.

A solução encontrada para contornar tal restrição foi adicionar para cada expressão de casamento de padrões uma variável responsável por armazenar o resultado da avaliação das alternativas. Como apenas a alternativa selecionada é executada, o valor armazenado na variável corresponderá à expressão selecionada. Esta abordagem funciona bem para a maioria dos casos, entretanto ao observar o código gerado para o conjunto de programas de testes foi

observado que em casamento de padrões aninhados eram gerados desvios e alocações desnecessárias. Neste cenário era comum aparecer uma seqüência de instruções que armazenavam um valor em uma variável e em seguida fazia o desvio para outra seqüência, a qual armazenava o valor da variável anterior em uma nova e tornava a fazer um desvio, como observado no Código 23. Este conjunto de instruções redundantes também dificultava a identificação de pontos de inserção da instrução *tail*.

```
1          call       int32 sum(int32,int32)
2          stloc.1
3          br         IL_01d5
4             ...
5   IL_01d5:  ldloc.1
6          stloc.2
7             br       IL_01b8
8          ...
9   IL_01b8:  ldloc.2
10         ret
```

Código 23. Instruções desnecessárias em casamento de padrões aninhados

A fim de verificar o impacto da remoção destes desvios foi construído um plugin que percorre a lista de instruções a procura de instruções de desvio incondicional. Ao encontrar, é verificado se uma possível variável armazenada antes do desvio é re-armazenada após o desvio. Em caso positivo ele guarda a última variável armazenada e continua verificando se há novos desvios e armazenamentos. Ao final ele substitui a variável de destino da primeira instrução de armazenamento pela última variável guardada e apaga todas as instruções e variáveis percorridas no caminho. Como resultado desta transformação é obtido o Código 24, o qual não só é mais enxuto como permite a inserção de uma instrução *tail*, inviável no código anterior.

```
1       call       int32 sum(int32,int32)
2       ret
```

Código 24. Código após a remoção dos desvios e variáveis desnecessárias

A Tabela 4 mostra os resultados obtidos, comparando os tempos de execução de cada programa gerado com e sem a remoção dos desvios e variáveis desnecessárias. Embora, a inserção desta nova fase no processo de compilação tenha causado, diretamente, pouco impacto, apenas 5,43% na média, indiretamente o impacto foi bem maior. Ela permitiu que novos pontos para inserção de instrução tail ou de recursão fosse identificados, melhorando o tratamento de memória através das técnicas descritas na Seção 5.3. A pequena melhoria se deve a diminuição da quantidade de instruções, tendo sido observado

pouca ou nenhuma alteração no consumo de memória após a remoção das instruções de desvios.

Tabela 4. Remoção de desvios e variáveis desnecessárias

	Sem remoção	Com remoção	Impacto da remoção
Tak	17,70	17,86	-0,90%
Primes	16,33	16,32	0,10%
Queens(12)	02,78	02,44	12,27%
Exp3	32,64	32,53	0,35%
Digitse1	02,14	02,09	0,08%
Wheelsieve1(50000)	00,87	00,65	25,67%
Wheelsieve2(50000)	12,17	12,12	0,43%
Média			**5,43%**

Esta otimização demonstra como é fácil identificar padrões de códigos e alterá-los utilizando o *Phoenix*, o que abre um grande horizonte de possíveis otimizações a serem implementadas.

5.3 Análises e Otimizações

Após a resolução dos problemas ocasionados pela a conversão da IR para MSIL, discutidos na Seção anterior, novos *plugins* foram construídos para testar alternativas para resolução do estouro da pilha de chamada e para otimizar o código gerado.

5.3.1 Tail call

Como descrito na Seção 2.1, uma das principais características das linguagens funcionais é o uso de recursão ou invés de estruturas de repetição. Este forte uso da recursão faz com que a pilha de chamadas cresça excessivamente, o que acarreta não só em um grande consumo de memória, como também, na

possibilidade de um estouro da memória. Para evitar este estouro de memória é necessário utilizar algum mecanismo que descarte o frame de atualização de chamadas recursivas.

A CLR disponibiliza a instrução *tail* a qual descarta o frame de atualização de uma chamada a um método desde que esta seja precedida por uma instrução de retorno. Este seria o mecanismo ideal para solucionar o problema de chamadas recursivas, entretanto sua implementação na CLR requer melhorias, uma vez que seu uso penaliza o desempenho do programa[25]. Esta penalidade, entretanto, ocorre apenas na implementação da CLR para máquinas com arquitetura x86, na implementação para x64 este problema foi corrigido, o que acarreta ganho de desempenho ao utilizar a instrução *tail* nesta última. Detalhes sobre diferenças de implementação da instrução *tail* para máquinas x86 e x64 e explicações sobre o desempenho desta são fornecidas por Shri Borde[**55**].

A fim de avaliar o desempenho do uso da instrução *tail* em diferentes implementações da CLR, foi utilizado o Código 25[26], o qual foi executado, através de um loop, 10.000 vezes. Após a compilação foi feita uma cópia, a qual teve seu código IL alterado, sendo adicionada uma instrução *tail* antes da chamada recursiva. Os dois programas foram então executados tanto em um sistema x86 como em um x64. Como esperado no ambiente x86 houve uma grande queda no desempenho, com tempo de execução em média 79% maiores para o código com a instrução *tail*. Já para a máquina x64 o mesmo código apresentou uma melhora no desempenho com tempo de execução em média 44% menor.

```
1  static double OriginalFunction(double d, int k)
2  {
3      if (k > 1) return OriginalFunction(d * (k + 1) / k, k - 1);
4      else return d;
5  }
```

Código 25. Função recursiva para teste de tail-calls

No compilador a inserção desta instrução é feita através de duas fases, a primeira (*MarkTailCallPhase*) responsável por marcar as instruções de chamadas que devem ser modificadas e outra (*ApplyTailCallPhase*) responsável pela inserção.

[25] Explicações para esta queda no desempenho no uso da instrução tail. em sistemas x86 podem ser encontradas no endereço http://blogs.msdn.com/shrib/archive/2005/01/25/360370.aspx

[26] Retirado do endereço http://www.jelovic.com/weblog/e59.htm

Esta separação se faz devido a uma restrição do Phoenix, que só permite a inserção da instrução *tail* após a fase de construção da pilha (*StackAllocatePhase*). São marcadas para inserção da instrução *tail* todas as instruções de chamada a um método que precedam uma instrução de retorno.

Como o descarte da pilha de execução é essencial para o não estouro da pilha de chamadas em alguns programas funcionais a inserção da instrução *tail* é habilitada por padrão no compilador, entretanto ela pode ser desligada através da diretiva de compilação *–notail*, caso o usuário identifique que esta inserção seja desnecessária para o código a ser compilado.

A Tabela 3 mostra os tempos de execução antes e após a inserção de instruções *tail*. Diferente do esperado, apenas para *Tak,* Queens e WheelSieve1 a inserção da instrução *tail* causou um aumento do tempo de execução considerável, respectivamente: 54,63%, 12,39% e 12,89% mais lento. O grande aumento no tempo de execução de *Tak* se deve ao fato deste programa ser altamente recursivo e sem alocação dinâmica de memória, como pode ser observado na Tabela 6. Desta forma, não há ganho com a redução do tempo gasto em coleta de lixo, o qual poderia compensar o tempo perdido com o uso da instrução *tail*. Para os demais exemplos o tempo de execução diminui, devido à diminuição no tempo gasto com coletas de lixo, sendo que para DigitsE1, programa que requer muita memória, este ganho foi bastante expressivo.

Tabela 5. Impacto da inserção de instrução tail

Programa	Sem Tail	Com Tail	Impacto da Inserção
TAK	17,86	27,62	-54,63%
PRIMES	16,32	15,14	7,19%
Queens(12)	02,44	02,74	-12,39%
EXP3	32,53	30,71	5,60%
DigitsE1	02,09	36,75	40,82%
WheelSieve1(50000)	00,65	00,73	-12,89%
WheelSieve2(50000)	12,12	09,45	7,70%
Média			**-2,95%**

A fim de buscar uma explicação sobre o porquê do comportamento observado, foi traçado um perfil do uso de memória destes programas através da observação do comportamento do coletor de lixo, os resultados podem ser observados na Tabela 6. Para Tak e WheelSieve1 não ocorreu nenhum diminuição do tempo em coleta de lixo, uma vez que para estes foram realizadas poucas ou nenhuma coleta de lixo tanto antes como após inserção da instrução tail. Para os demais programas houve a diminuição do tempo gasto pelo coletor de lixo, o que ajudou a compensar as perdas impostas pela inserção da instrução *tail*. Como pode se observar o número de coleta de lixos praticamente não se alterou o que demonstra que a diminuição do tempo de coleta não se deve a uma diminuição do número de coletas, e sim a diminuição do número de frames a serem percorridos pelo coletor de lixo, promovida pela instrução *tail* ao descartar frames desnecessários.

Tabela 6. Informações sobre o coletor de lixo após a inserção de instruções tail

Programa	Sem Tail		Com Tail	
	% Tempo em coleta de lixo	Total de coletas	% Tempo em coleta de lixo	Total de coletas
Tak	0,0%	0	0,0%	0
Primes	63,9%	231	60,7%	231
Queens(12)	21,0%	65	17,9%	65
Exp3	70,2%	1644	64,6%	1644
DigitsE1	71,1%	2985	44,7%	3124
WheelSieve1(50000)	0,0%	3	0,0%	3
WheelSieve2(50000)	70,3%	159	50,1%	159

5.3.2 Desvios em chamadas recursivas

Como demonstrado anteriormente o uso de instruções *tail* não apresenta bom desempenho em implementações da CLR para sistemas x86. Outra técnica que pode ser utilizada para evitar que a pilha de chamadas estoure em chamadas recursivas é através da inserção de uma instrução de desvio incondicional para o

início da função. Esta técnica, entretanto, só pode ser utilizada para chamadas recursivas à própria função, não contemplando chamadas mutuamente recursivas, onde duas funções diferentes fazem chamadas recursivas entre si, como a mostrada no Código 26. Neste código a função *foo* chama *boo,* que por sua vez chama *foo*. Desta forma, instruções *tail* ainda são necessárias para este tipo de recursão.

```
1  foo = boo
2  boo = if ({-condição de parada-}) 1 else foo
```

Código 26. Chamadas mutuamente recursivas

Para promover esta otimização foi criado um *plugin*, o qual faz uso das marcações feitas na fase *MarkTailCallPhase* e substitui a fase *ApplyTailCallPhase* por uma nova que verifica se a instrução marcada corresponde a uma chamada a própria função na qual ela esta inserida e se este for o caso ao invés de adicionar uma instrução *tail* salva os argumentos e faz um desvio para o início da chamada. O código deste *plugin* pode ser conferido no Apêndice C.

O resultado da aplicação deste *plugin* pode ser observado na Tabela 7. O maior impacto foi observado nos programa *Tak* e *Queens* que tiveram seus tempos de execução drasticamente reduzidos, respectivamente, 45,48% e 30,19% menor. Outro programa beneficiado por esta substituição foi WheelSieve1 que obteve um tempo 25,23% menor. Os demais programas variaram pouco, obtendo variações menores que 1% para mais e para menos.

É importante observar que a melhoria ocorre nos exemplos onde a inserção da instrução *tail*, Seção 5.3.1, gerou uma grande queda de desempenho. Desta forma, a substituição de instruções *tail* por instruções de desvio permite o tratamento de chamadas recursivas sem os efeitos colaterais no desempenho gerados pela instrução *tail*.

Tabela 7. Recursão através de desvio para o inicio da função

Programa	Com Tail	Com desvio	Impacto
Tak	27,62	15,06	45,48%
Primes	15,14	15,23	-0,56%
Queens	16,47	11,50	30,19%
Exp3	30,71	30,62	0,29%

DigitsE1	36,75	36,70	0,13%
WheelSieve1	12,93	09,67	25,23%
WheelSieve2	20,70	20,71	-0,04%
Média			**14,39%**

Por ter apresentado na média um bom desempenho (14,39%) e uma grande melhora (acima de 25%) para programas extremamente recursivos, como o *Tak, Queens* e *WheelSieve1*, o uso de desvios em chamadas recursivas para a própria função foi incorporado ao compilador. Entretanto, este pode ser desabilitado pelo usuário através da diretiva *–nobranchrecursion*.

5.3.3 Casamento de padrões com valores booleanos

Desvios condicionais, como instruções *if* em *haskell*, são traduzidas para a linguagem *Core* como instruções de casamento de padrões de valores booleanos. Uma vez que, na *Core*, valores booleanos são representados utilizando tipos algébricos, a avaliação da expressão condicional e a escolha da alternativa requerem uma série de operações, que degradam seu desempenho. O Código 27 mostra em C# como é feito tal mapeamento. Inicialmente, a condição é avaliada e de acordo com o resultado é gerado o construtor correspondente, linhas 1 a 5. Através de uma instrução *switch* a tag deste construtor é verificada e é feito o desvio para o código da alternativa correspondente, linhas 7 a 15.

```
1  //Avaliação da expressão e instanciação do Construtor correspondente
2  if (ExpCondição)
3      pack = RuntimeSystem.TRUE;
4  else
5      pack = RuntimeSystem.FALSE;
6  //Casamento de padrão utilizando o valor resultante da avaliação da condição
7  switch (((Pack) pack).tag)
8  {
9      case 0:
10         //Alternativa condição falsa
11         break;
12
13     case 1:
14         //Alternativa condição verdadeira
15         break;
16 }
```

Código 27. Representação de desvios condicionais com construtores para valores booleanos.

Este código pode facilmente ser otimizado, eliminando se o uso de um construtor para representar o valor booleano. Uma vez que a avaliação da expressão de comparação retorna sempre um valor inteiro (zero para falso e um para verdadeiro), é possível substituir a avaliação da *tag* na instrução *switch* pela avaliação da expressão condicional, resultando no Código 28.

```
1  switch (ExpCondição)
2  {
3      case 0:
4           //Alternativa condição falsa
5           break;
6
7      case 1:
8           //Alternativa condição verdadeira
9           break;
10 }
```

Código 28. Representação de desvios condicionais otimizada

Com auxílio de um *plugin*, tal otimização foi adicionada ao compilador. O impacto desta adição pode ser observado na Tabela 8. Três programas tiveram grande melhoria de desempenho: *Tak* (16,46%), *Queens* (11,77%) e *WheelSieve1* (18,21%). Para os demais o impacto foi pequeno, ou ainda irrelevante (variação menor que 0,1%).

Tabela 8. Impacto da remoção de construtores em desvios condicionais.

Programa	Com Tail	Com desvio	Impacto
Tak	15,06	12,58	16,46%
Primes	15,23	15,22	0,07%
Queens	11,50	10,15	11,77%
Exp3	30,62	30,64	-0,07%
DigitsE1	36,70	36,01	1,89%
WheelSieve1	09,67	07,91	18,21%
WheelSieve2	20,71	20,58	0,65%
Média			**7,00%**

5.4 Análise Final do Compilador

Após os testes utilizando *plugins*, descritos nas seções anteriores, foram selecionadas as otimizações que obtiveram melhores resultados, as quais foram

adicionadas como fases do compilador final. De modo a quantificar o desempenho do código gerado por este compilador, nesta Seção, este será comparado com outros compiladores Haskell.

5.4.1 Versus Haskell .NET

O compilador Haskell .NET[**5**], representou o modelo de compilação utilizado neste trabalho e é o único dentre os compiladores analisados na Seção 2.5 que gera código .NET a partir de uma linguagem funcional estrita. Desta forma, foi o melhor exemplo encontrado para mensurar a qualidade do código gerado pelo *PhxSTGCompiler,* tendo como base uma implementação anterior.

Como pode ser observado na Tabela 9 na média o PhxSTGCompiler obteve tempos 1,70% menores que o Haskell .NET. Tak e Queens obtiveram uma grande melhoria, acima de 20%, decorrente principalmente da otimização dos desvios condicionais, descrita na Seção 5.3.3. Apenas em dois exemplos: Exp3 (-2,2%) e DigitsE1 (-44,8%), ocorreu uma queda no desempenho, sendo esta bastante expressiva para o último.

Tabela 9. PhxSTGCompiler x Haskell .NET

Programa	Haskell .NET	PhxSTGCompiler	PhxSTGCompiler/Haskell .NET
Tak	15,10	12,58	20,00%
Primes	15,24	15,22	0,13%
Queens	13,54	10,15	33,45%
EXP3	29,97	30,64	-2,20%
DigitsE1	19,86	36,01	-44,83%
WheelSieve1	08,29	07,91	4,77%
WheelSieve2	20,70	20,58	0,60%
Media			**1,70%**

Buscando identificar o porquê de DigitsE1 ter tido uma queda de desempenho tão grande, foi feito um vasto estudo tanto no código Core como no STG gerados para este. Tal estudo demonstrou a ausência, no código Core, de

algumas otimizações e informações importantes. Foi observado que na STG de DigitsE1 algumas expressões ao invés de gerarem *thunks*, para avaliação posterior, eram avaliadas imediatamente através de expressões *case*. Este tipo de otimização era previsto na transformação Core para STG (Seção 4.2.2), mas apenas para tipos *Unboxed*, o que não é o caso. Outra informação, ausente, que demonstrou alta relevância foi a presença da *flag* de atualização */r* (*reentrant*) em alguns *binds*, que informa que a *closure* não necessitava ser atualizada. O uso desta última informação causou grande impacto no programa WheelSieve2.

Tais informações não são obtidas através da Core, desta forma, a fim de comprovar a importância destas, alterações manuais foram feitas no processo de compilação. Estas alterações não se estendem a compilação em geral sendo específicas para comprovar a influência destas informações em DigitsE1 e WheelSieve2. Como observado na Tabela 10, a grande queda no desempenho observado em DigitsE1, praticamente, não existe mais (-0,5%) e WheelSieve2 obteve uma grande melhoria, passando a ser 224% mais rápido que a versão compilada com o Haskell .NET.

Tabela 10. Compilação com informações ausentes na CORE

Programa	PhxSTGCompiler(Alt)	Haskell .NET	PhxSTGCompiler/Haskell .NET
DigitsE1	19,97	19,86	-0,5%
WheelSieve2	6,39	20,58	224,0%

Com estes novos valores, com exceção de Exp3, o PhxSTGCompiler obtém melhores resultados para todos os códigos analisados e passa a ser na média 39,95% mais veloz que o Haskell .NET, vide Tabela 11. Como mencionando na Seção 4.2.2, a Core gerada pelo GHC está sendo, atualmente, modificada. Espera-se que em futuras versões estas informações possam estar presentes, auxiliando na geração de um código mais veloz.

Tabela 11 - PhxSTGCompiler* x Haskell .NET. *Com alterações manuais

Programa	Haskell .NET	PhxSTGCompiler	PhxSTGCompiler/Haskell .NET

Tak	12,58	15,10	20,00%
Primes	15,22	15,24	0,13%
Queens	10,15	13,54	33,45%
EXP3	30,64	29,97	-2,20%
DigitsE1	19,97	19,86	4,77%
WheelSieve1	07,91	08,29	-0,5%
WheelSieve2	06,39	20,58	223,98%
Media			**39,95%**

5.4.2 Versus GHC nativo

Por ser considerado o compilador estado da arte para Haskell, a comparação com o gerador de código nativo do GHC é essencial para qualquer implementação de um compilador Haskell. A Tabela 12 resume os resultados desta comparação. Como esperado, os tempos obtidos com GHC foram menores, havendo uma diferença superior a uma ordem de magnitude apenas para *Primes* e *Exp3*. Para os demais a variação foi bem menor, sendo bastante semelhante para *Tak* (1,21) e praticamente igual para *WheelSieve2* (1,02).

Tabela 12. PhxSTGCompiler x GHC

Programa	PhxSTGCompiler	GHC	PhxSTGCompiler/GHC
Tak	12,58	10,43	1,21
Primes	15,22	00,46	33,08
Queens	10,15	03,29	3,09
Exp3	30,64	01,74	17,58
DigitsE1*	19,97	02,01	9,95
WheelSieve1	07,91	02,38	3,32
WheelSieve2*	06,39	06,26	1,02
Média			9,89

***Valores obtidos com modificações manuais explicadas na Seção anterior**.

A explicação para uma diferença tão grande para *Primes* e *Exp3* se deve ao alto tempo gasto com coleta de lixo realizados durante a execução destes

programas, onde a porcentagem de tempo gasto com coleta de lixo em relação ao tempo total de execução ficou acima de 55% (Tabela 13). Como o GHC tem um coletor de lixo especificamente criado para lidar com uma linguagem funcional estrita, em programas onde o consumo de memória exige um grande trabalho por parte do coletor de lixo o GHC tende a se destacar.

Tabela 13. Perfil do consumo de memória (PhxSTGCompiler)

Programa	Bytes alocados na Heap	% Tempo em coleta de lixo
Tak	0	0,00%
Primes	549.341.700	55,34%
Queens	729.656	18,66%
Exp3	12.138.600	65,29%
DigitsE1*	6.657.824	53,36%
WheelSieve1	33.458.150	11,48%
WheelSieve2*	**14.589.090**	**22,28%**

***Valores obtidos com modificações manuais explicadas na Seção anterior.**

A comparação com GHC demonstrou que com exceção dos programas onde o consumo de memória é algo crítico o compilador implementado é capaz de gerar programas com desempenho semelhante. Mostra ainda onde está o gargalo do mapeamento de uma linguagem funcional na plataforma .NET, gerenciamento de memória, apontando a direção para a qual futuras pesquisas nesta área devem ser voltadas.

5.5 Considerações Finais

Neste capítulo foram efetuadas diversas alterações no código utilizando o modelo de *plugins*, fornecido pelo Phoenix. Este recurso em conjunto com a biblioteca de manipulação de código IR demonstrou ser bastante útil, facilitando a identificação de padrões de códigos e sua manipulação. O conjunto de otimizações permitiu que o compilador final gerasse um código com desempenho satisfatório.

Ficou evidente que o maior problema no mapeamento de linguagens funcionais na plataforma .NET é o gerenciamento de memória. Por este motivo a maioria das otimizações realizadas tiveram como objetivo diminuir o consumo de memória dos programas gerados. Entretanto, melhorias maiores não foram possíveis devido à inviabilidade da manipulação do coletor de lixo da CLR e da opção por gerar código verificável.

6 CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho, após um amplo estudo de técnicas de implementações de linguagens funcionais na plataforma .NET, foi apresentada uma nova abordagem para construção de um compilador .NET para uma linguagem funcional. O compilador foi construído utilizando o framework Phoenix, que é uma ferramenta para construção de compiladores e de ferramentas para análise e otimização de código.

O uso do framework Phoenix, inicialmente, representou uma dificuldade a mais, pois por ser uma ferramenta recente, havia pouco material de referência para estudo. Entretanto, passado esta etapa inicial seu uso facilitou, bastante, a construção do compilador, ao abstrair o processo de geração de código .NET, evitando a manipulação de código MSIL diretamente. A maior contribuição do Phoenix, entretanto, foi permitir que uma série de otimizações fossem realizadas, melhorando o desempenho do código gerado.

O uso de *plugins* permitiu que diversas otimizações fossem testadas gradualmente, sem que fosse necessário alterar diretamente o código do compilador. O teste individual de cada otimização permitiu avaliar, isoladamente, o impacto de cada uma das otimizações e assim escolher uma melhor configuração para o compilador.

Por fim, a comparação com outros compiladores demonstrou que o código gerado possui um bom desempenho, produzindo códigos mais velozes que os gerados pelo Haskell .NET e valores satisfatórios quando comparado ao código gerado pelo GHC nativo.

6.1 Resumo das Contribuições

A seguir é descrito um resumo das principais contribuições deste trabalho:

- apresentação do estado da arte de implementações de linguagens funcionais na plataforma .NET, onde foram descritas as principais técnicas de mapeamento e comparação entre projetos reais;
- estudo detalhado do framework Phoenix, com descrição das principais funcionalidades, sempre que possível através de exemplos práticos;
- apresentação de uma nova abordagem para construção de um compilador funcional, capaz de gerar código .NET para programas Haskell. Este compilador além de validar a abordagem escolhida serve como ferramenta para auxílio de pesquisas na área de otimização de código, uma vez que novas técnicas podem facilmente ser incorporadas a ele;
- descrição e implementação de técnicas de otimização de código. Podendo algumas destas técnicas ser implementadas mesmo em compiladores não funcionais.

6.2 Limitações e Trabalhos Futuros

O prelúdio compilado representa apenas uma pequena parte do real. Apenas as funcionalidades básicas requeridas pelos exemplos utilizados nos testes foram contempladas. A compilação completa de toda a especificação Haskell 98 [**12**] e das bibliotecas do GHC, embora bastante trabalhosa, permitiria que qualquer programa Haskell pudesse ser diretamente compilado para .NET através desta implementação. Com a compilação completa do prelúdio, melhores testes poderiam ser efetuados utilizando os grupos *Espectral* e *Real* do NoFib. O que pode apontar novas possibilidades de otimização do código.

Neste projeto foi utilizada a versão de julho de 2007 do Phoenix SDK. Esta versão possui diversas limitações quanto à geração de código .NET. Algumas destas limitações foram contornadas, seguindo orientações obtidas através do fórum da ferramenta, outras foram contornadas utilizando *plugins*, como especificado na Seção 5.2. Pouco antes do fim deste trabalho uma nova versão do Phoenix foi lançada, trazendo, dentre outras novas funcionalidades, diversas correções e melhorias na geração de código MSIL. Foram feitas algumas tentativas de atualizar o código para esta nova versão do SDK, entretanto devido a restrições de tempo

esta atualização foi deixada de lado. A atualização para a nova versão por si só já corrige uma série de deficiências na geração de código MSIL e pode resultar na geração de códigos mais velozes.

O framework Phoenix fornece uma grande quantidade de facilidades para execução de análises e otimizações de código. As otimizações aqui implementadas utilizaram apenas uma parte destes recursos, o que já foi suficiente para um ganho considerável no desempenho. Dentre os diversos projetos que podem ser desenvolvidos utilizando o compilador aqui implementado em conjunto com a API Phoenix, são sugeridos:

- Adição de *labels* que permitam identificar trechos do código IR responsáveis pelo mapeamento das estruturas funcionais. A identificação destes trechos de código poderia ser utilizada para permitir que novas formas de mapeamentos fossem testadas utilizando o modelo de *plugin*s.

- Utilizar as bibliotecas de análise, tais como: Graph e Alias, para identificar trechos de código que executam tarefas desnecessárias ou passíveis de otimização. A identificação destes trechos de código pode ser adicionada a *plugins* que alterariam o funcionamento do compilador permitindo a otimização do código gerado.

Embora o PhxSTGCompiler permita chamadas a métodos estáticos escritos em outras linguagens .NET, como foi feito em algumas das bibliotecas do prelúdio, a implementação aqui proposta teve como objetivo principal melhorar o desempenho do mapeamento de estruturas funcionais no ambiente .NET, não investindo muito na interoperabilidade. Sem dúvida esta interoperabilidade foi facilitada uma vez que após mapeadas na CLR todas as linguagens compartilham o mesmo conjunto de tipos. Entretanto, a construção de bibliotecas que encapsulem as diferenças existentes entre as estruturas funcionais e as estruturas OO permitiria uma comunicação mais direta.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] J. Hughes, "Why Functional Programming Matters," in *Research Topics in Functional Programming*. Addison-Wesley Pub, 1989, vol. 32, pp. 98-107.

[2] J. Gough, *Compiling for the .NET Common Language Runtime (CLR)*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001.

[3] D. Box and T. Pattison, *Essential .NET: The Common Language Runtime*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[4] ECMA, "Standard ECMA-335: Common language infrastructure (CLI).," 2006.

[5] M. L. d. B. Monteiro, "Integrando Haskell à Plataforma .NET," Dissertação de Mestrado, 2006.

[6] D. Syme, "ILX: Extending the .NET Common IL for Functional Language Interoperability," *Electronic Notes in Theoretical Computer Science*, vol. 59, 2001.

[7] E. Meijer, N. Perry, and A. v. Yzendoorn, "Scripting .NET Using Mondrian," in *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, 2001, pp. 150-164.

[8] Y. Bres, B. P. Serpette, and M. Serrano, "Bigloo.NET: compiling Scheme to .NET CLR," *Journal of Object Technology*, vol. 3, pp. 71-94, 2004.

[9] M. Moskal, P. W. Olszta, and K. Skalski, "Nemerle: Introduction to a Functional .NET Language,"

[10] D. A. Watt, *Programming Language Design Concepts*. John Wiley & Sons, 2004.

[11] Y. Minamide, G. Morrisett, and R. Harper, "Typed Closure Conversion," in *Symposium on Principles of Programming Languages*, 1996, pp. 271-283.

[12] S. L. Peyton Jones, *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[13] J. Smith, N. Perry, and E. Meijer, "Mondrian for .NET," *Dr. Dobb's J.*, vol. 27, pp. 28-34, 2002.

[14] L. Augustsson and T. Johnsson, "The Chalmers lazy ML-compiler," *Comput. J.*, pp. 127-141, 1989.

[15] Microsoft Corporation. Microsoft .NET Framework. [Online]. Disponível em:

http://www.microsoft.com/net/

[16] ECMA International. [Online]. Disponível em: http://www.ecma-international.org/default.htm

[17] International Organization for Standardization. [Online]. Disponível em: http://www.iso.org/iso/home.htm

[18] S. Lidin, *Expert .NET 2.0 IL Assembler*. Apress, 2006.

[19] Microsoft Research. SSCLI (Rotor) Home Page. [Online]. Disponível em: http://research.microsoft.com/sscli/

[20] MONO. [Online]. Disponível em: http://www.mono-project.com/Main_Page

[21] G. A. Avelino, "Avaliação de Desempenho de Programas C# em Ambientes .NET - SSCLI 2.0, .NET 2.0 e .NET 3.0," Trabalho não publicado, 2006.

[22] Novell. NOVELL: Worldwide. [Online]. Disponível em: http://www.novell.com/home/index.html

[23] S. Finne, D. Leijen, E. Meijer, and S. L. Peyton Jones, "H/Direct: A Binary Foreign Language Interface for Haskell," in *ICFP'98*, 1998.

[24] GreenCard: A Haskell Foreign Function Interface Preprocessor. [Online]. Disponível em: http://www.haskell.org/greencard/

[25] S. Finne. Hugs98 for .NET. [Online]. Disponível em: http://www.galois.com/~sof/hugs98.net/

[26] E. Meijer and S. Finne, "Lambada, Haskell as a better Java," in *Proc. Haskell Workshop*, vol. 41, 2001, pp. 91-119.

[27] L. O´Boyle, "Making Haskell .NET Compatible,"

[28] S. L. Peyton Jones, "Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-Machine," *Journal of Functional Programming*, vol. 2, pp. 127-202, 1992.

[29] Microsoft F#. [Online]. Disponível em: http://research.microsoft.com/fsharp/

[30] H. Barendregt and E. Barendsen, "Introduction to Lambda Calculus," in *Aspenäs Workshop on Implementation of Functional Languages*, Göteborg, 1988.

[31] T. M. Breuel, "Lexical Closures for C++," in *C++ Conference*, 1988, pp. 293-304.

[32] A. Kennedy and D. Syme, "Design and implementation of generics for the .NET

Common language runtime," in *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, New York, NY, USA, 2001, pp. 1-12.

[33] S. Marlow and S. L. Peyton Jones, "Making a fast curry Push/enter vs eval/apply for higher-order languages," in *ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, 2004, pp. 4-15.

[34] O. Hunt, "The Provision of Non-Strictness, Higher Kinded types and Higher Ranked Types on an Object Oriented Virtual Machine," Dissertação de Mestrado, 2006.

[35] E. Meijer and K. Claessen, "The Design and Implementation of Mondrian," in *Haskell Workshop*, 1997.

[36] M. Odersky and P. Wadler, "Pizza into Java: translating theory into practice," in *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1997, pp. 146-159.

[37] S. L. Peyton Jones and D. R. Lester, *Implementing Functional Languages*. Prentice-Hall, Inc, 1992.

[38] B. Cabral, P. Marques, and L. Silva, "RAIL: Code Instrumentation for .NET," in *SAC '05: Proceedings of the ACM Symposium on Applied Computing*, 2005.

[39] Microsoft Corporation. COM: Component Object Model Technologies.

[40] INRIA. (2008,) The Caml language. [Online]. Disponível em: http://caml.inria.fr/

[41] Microsoft Corporation. Microsoft Research. [Online]. Disponível em: http://research.microsoft.com/projects/ilx/ilx.aspx

[42] S. Marlow. The Glasgow Haskell Compiler. [Online]. Disponível em: http://www.haskell.org/ghc/

[43] S. L. Peyton Jones and S. Marlow, "The STG Runtime System (Revised)," Yale University, 1999.

[44] M. Research. Microsoft Research. [Online]. Disponível em: http://research.microsoft.com/phoenix/

[45] "Phoenix Documentation," 2007.

[46] D. Stewart, "Multi-Paradigm Just-In-Time Compilation," Dissertação de Mestrado, 2002.

[47] S. L. Peyton Jones and A. L. M. Santos, "A transformation-based optimiser for Haskell," *Sci. Comput. Program.*, vol. 32, pp. 3-47, 1998.

[48] S. L. Peyton Jones and S. Marlow, "Secrets of the Glasgow Haskell Compiler Inliner," *Journal of Functional Programming*, vol. 12, pp. 393-434, 1999.

[49] M. Serrano, "Inline expansion: when and how?," in *Proceedings of the conference on Programming Languages, Implementation and Logic Programming*, 1997.

[50] P. Wadler and R. J. M., "Projections for Strictness Analysis," in *Functional Programming Languages and Computer Architecture*, vol. 274, 1987, pp. 385-407.

[51] S. L. Peyton Jones, C. V. Hall, K. Hammond, W. Partain, and P. Wadler, "The Glasgow Haskell compiler: a technical overview," in *Proceedings of UK Joint Framework for Information Technology (JFIT) Technical Conference*, 1993.

[52] A. Tolmach, "An External Representation for the GHC Core Language,"

[53] S. L. Peyton Jones and J. Launchbury, "Unboxed Values as First Class Citizens in a Non-Strict Functional Language," in *Proceedings of the Conference on Functional Programming and Computer Architectur*, Cambridge, Massachussets, USA, 1991, pp. 636-666.

[54] W. Partain, "The nofib Benchmark Suite of Haskell Programs," *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, no. Springer-Verlag, 1993.

[55] S. Borde. Shri Borde's WebLog. [Online]. Disponível em: http://blogs.msdn.com/shrib/archive/2005/01/25/360370.aspx

[56] D. Stutz, T. Neward, and G. Shilling, *Shared Source CLI Essentials*. O´Reilly, 2003.

[57] D. Wakeling, "A Haskell to java Virtual Machine Code Compiler," in , 1998, pp. 39-52.

[58] S. Thompson, *Haskell: The Craft of Functional Programming (2nd Edition)*. Addison Wesley, 1999.

[59] M. M. T, "The Haskell 98 Foreign Function Interface 1.0,"

[60] D. Syme, A. Granicz, and A. Cisternino, *Expert F\#*. APress, 2007.

[61] R. Pickering, *Foundations of F\#*. Apress, 2007.

[62] T. Parr, *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Bookshelf, 2007.

[63] A. Kennedy and D. Syme, "Pre-compilation for .NET Generics," Microsoft Research, Cambridge, U.K., 2005.

[64] A. V. Aho, R. Sethi, and J. D. Ullman, *Compiladores: Princípios, Técnicas e Ferramentas*. LTC, 1995.

[65] J. Launchbury and S. L. Peyton Jones, "Lazy Functional State Threads," *SIGPLAN Not.*, vol. 29, pp. 24-35, 1994.

[66] The NoFib Haskell Benchmark Suite. [Online]. Disponível em: http://einstein.dsic.upv.es/nofib/ingles/index_1024_en.php

[67] G. Bracha, N. Gafter, J. Gosling, and P. von der Ahé. Closure for the Java Programming Language. [Online]. Disponível em: http://www.javac.info/

[68] T. Dowd, F. Henderson, and P. Ross, "Compiling Mercury to the .NET Common Language Runtime," *Proceedings of Babel'01*, 2001.

[69] J. Hamilton, "Language integration in the common language runtime," *SIGPLAN Not.*, vol. 38, 2003.

[70] G. A. Avelino, "Análise Comparativa de Frameworks para Instrumentação de Código .NET," Trabalho não publicado, 2006.

[71] S. Smetsers, E. Nöcker, J. v. Groningen, and R. Plasmeijer, "Generating Efficient Code for Lazy Functional Languages," in *Proceedings of the Conference on Functional Programming and Computer Architecture*, 1991, pp. 592-617.

[72] S. C. Wray and J. Fairbairn, "Non-Strict Languages - Programming and Implementation," *The Computer Journal*, vol. 32, pp. 142-151, 1989.

[73] M. Tullsen, "Compiling Haskell to Java," in *IFL '97: Selected Papers from the 9th International Workshop on Implementation of Functional Languages*, 1996.

[74] M. Serrano and M. Feeley, "Storage use analysis and its applications," in *IFL '97: Selected Papers from the 9th International Workshop on Implementation of Functional Languages*, 1996, pp. 50-61.

[75] J. B. Rosser, "Highlights of the history of the lambda-calculus," in *LFP '82: Proceedings of the 1982 ACM symposium on LISP and functional programming*, 1982, pp. 216-225.

[76] N. Perry and E. Meijer, "Implementing Functional Languages on Object-Oriented Virtual Machines," *IEE Proceedings - Software*, vol. 151, pp. 1-9, 2004.

[77] I. Holyer and E. Spiliopoulou, "The Brisk Machine: A Simplified STG Machine," in *Implementation of Functional Languages, 9th International Workshop, {IFL}'97, St. Andrews, Scotland, {UK}, September 1997, Selected Papers, {LNCS} 1467*, 1999, pp. 20-38.

[78] R. Douence and P. Fradet, "A Systematic Study of Functional Language

Implementations," *ACM Transactions on Programming Languages and Systems*, vol. 20, pp. 344-387, 1998.

[79] F. H. de, H. P. de, R. M. Ferreira, and R. D. Lins, "An Action Semantics for STG,"

[80] D. Coutts, D. Stewart, and R. Leshchinskiy, "Rewriting Haskell Strings," in *Practical Aspects of Declarative Languages 8th International Symposium, PADL 2007*, 2007, pp. 50-64.

[81] J. Clements and M. Felleisen, "A Tail-recursive Machine With Stack Inspection," *ACM Trans. Program. Lang. Syst.*, vol. 26, pp. 1029-1052, 2004.

[82] K. Choi, H.-i. Lim, and T. Han, "Compiling Lazy Functional Programs Based on the Spineless Tagless G-Machine for the Java Virtual Machine," *Lecture Notes in Computer Science*, vol. 2024, 2001.

[83] Y. Bres, B. P. Serpette, and M. Serrano, "Compiling Scheme programs to .NET Common Intermediate Language," in *2nd International Workshop on .NET Technologies*, 2004.

[84] S. L. Peyton Jones and D. Lester, "A Modular Fully-lazy Lambda Lifter in HASKELL," *Software - Practice and Experience*, vol. 21, pp. 479-506, 1991.

[85] R. Ennals and S. L. Peyton Jones, "Optimistic evaluation: an adaptive evaluation strategy for non-strict programs," in *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, 2003, pp. 287-298.

[86] S. Finne, D. Leijen, E. Meijer, and S. L. Peyton Jones, "Calling Hell from Heaven and Heaven from Hell," in *ICFP '99: Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, 1999, pp. 114-125.

[87] T. Johnsson, "Lambda lifting: transforming programs to recursive equations," in *Functional programming languages and computer architecture. Proc. of a conference (Nancy, France, Sept. 1985)*, 1985.

[88] M. Monteiro, M. Araújo, R. Borges, and A. Santos, "Compiling Non-strict Functional Languages for the .NET Platform," *Journal of Universal Computer Science*, vol. 11, pp. 1255-1274, 2005.

[89] R. F. Massa, R. D. Lins, and A. L. M. Santos, "A back-end for GHC based on categorical multi-combinators," in *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, New York, NY, USA, 2004, pp. 1482-1489.

[90] S. Marlow, A. R. Yakushev, and S. L. Peyton Jones, "Faster laziness using dynamic

pointer tagging," in *ICFP '07: Proceedings of the 2007 ACM SIGPLAN international conference on Functional programming*, Freiburg, Germany, 2007, pp. 277-288.

APÊNDICE A - UNIDADES DE COMPILAÇÃO

As unidades de compilação representam estruturas presentes na máquina STG e contém código responsável por gerar sua respectiva representação IR. São divididas em unidades básicas, expressões, unidades atômicas e alternativas.

A Tabela 14 descreve classes que representam as unidades básicas de compilação, as quais correspondem aos nós principais da STG. Utilizam o método Generate para gerar seu código IR e delega às unidades que as compõem a geração de seus próprios códigos.

Tabela 14. Unidades básicas

Classe	Campos	Geração da IR (método Generate)
ModuleUnit	• name:String • binds:List<Bind>	Constrói a classe correspondente ao módulo e adiciona um método *.cctor* para inicialização dos campos estáticos correspondentes aos *binds* globais. Através de um loop é chamado o método *Generate* de cada um dos *binds* armazenados em sua lista.
Bind	• var:AtomVariable • lambda:LambdaForm	Verifica se a ligação (*bind*) é global ou local e cria o local correspondente para o armazenamento da closure. Caso seja global é criado um campo estático na classe e um método para sua inicialização, o qual é adicionado a uma lista de métodos de inicialização a serem chamados no método *.cctor* da classe. Caso o *bind* seja local a closure é armazenada como variável local da função FEP, criada a partir da *lambda-form*.
Lambda-Form	• freeVars:List<AtomVariable> • arg:List<AtomVarible>	Verifica a *flag* de atualização para identificar se é necessário criar a SEP e

	• flag:Bool • expression:Expression	decidir o tipo de closure a ser instanciado. A SEP é criada apenas quando o valor da flag for falso (closure não atualizável). O método *Generate* da *lambda-form* gera o esqueleto da FEP e chama o método *Evaluation* da expressão armazenada, a qual gera o código correspondente da função.
DataUnit	• listTvs:List<String> • listConsDef:List<ConstructorDefUnit>	Cria *lambda-forms* que instanciam construtores. Tais *lambda-form* são necessárias para aplicações parciais de construtores e passagem de um construtor como argumento de uma função. Após a criação das *lambda-forms*, estas são ligadas a variáveis através de um *bind* e cada um destes *binds* tem seu método *Generate* executado.

As unidades de compilação que representam expressões correspondem às expressões presentes na STG. Geram código IR através do método *Evaluation*, o qual além de gerar sua representação IR retorna um operando que armazena o resultado da expressão.

Tabela 15. Unidades de compilação que representam expressões

Classe	Campos	Geração da IR (método Evaluation)
ExpLet	• binds:List<Bind> • expression:Expression	Casa *bind* presente tem seu método *Generate* executado. Após a geração do código dos binds a expressão é avaliada, retornando o operando que armazena o resultado.
ExpLetRec	• binds:List<Bind> • expression:Expression	Semelhante a *ExpLet*, porém antes de efetuar o *bind* cada variável é adicionada a lista de variáveis livres da *lambda-form*, permitindo que a expressão possa fazer referência

		recursiva.
ExpCase	•expression:Expression •alts:List<Alternative>	Cria uma instrução *switch* onde para um *case* é gerado para cada objeto AlternativeUnit. Para tipos algébricos os valores presentes no construtor geram variáveis locais, para poderem ser acessados pela expressão. É criada uma variável para o armazenamento da expressão selecionada, a qual consiste no retorno da avaliação de uma expressão case.
ExpLiteral	•lit:AtomLiteral	Retorna a avaliação do AtomLiteral armazenado.
ExpApplication	•var:AtomVariable •args:List<Atom>	Utiliza *Reflection* e informações armazenadas no ambiente de compilação para decidir se a função pode ser chamada diretamente. Se a aridade é conhecida em tempo de compilação e a aplicação é saturada é gerado código para a chamada direta da função estática correspondente. Caso contrário os argumentos são empilhados na pilha de argumentos e é feita a chamado ao método *Enter* da closure correspondente.
ExpConstructor	•const:Constructor •args:List<Atom>	Utilizando as informações coletadas através de funções criadas a partir da DataUnit correspondente é verificado se a aplicação do construtor é saturada. Se a aplicação for satura é gerada a classe Pack correspondente ao construtor e seus argumentos, caso contrário a aplicação do construtor é tratada como uma aplicação parcial da função do construtor.
ExpBuiltIn	•op:PrimitiveOperator •args:List<Atom>	Gera uma instrução que aplica dois operandos e retorna o resultado da aplicação. A operação a ser executada é definida pelo *PrimitiveOperator*, o qual é

	mapeado para uma operação básica presente na CLR.

Unidades de compilação atômicas representam os elementos atômicos da STG. Estes elementos pode ser variáveis, literais, construtores ou ainda expressões entre parênteses. Seu método *Evaluation* retorna o operando correspondente a unidade atômica que pode ser uma variável ou uma constante. Não gera instruções IR diretamente, no geral, apenas retorna operandos criados em outras unidades de compilação.

Tabela 16. Unidades de compilação atômicas

Classe	Campos	Geração da IR (método Evaluation)
AtomVariable	• moduleName:String • varName:String • type:STGType	Faz a busca, primeiramente, na tabela de símbolos da função, caso não encontre nesta faz a busca na tabela de símbolos do módulo. A partir do símbolo localizado é retornado um *VariableOperand* que representa ou uma variável local da função ou um campo estático da classe, neste último caso quando o símbolo é localizado na tabela de símbolos do módulo (*bind* global).
AtomLiteral<T>	• value:T	Retorna um *ImmediateOperand* com o valor correspondente ao literal.
Constructor	• moduleName:String • constName:String	Utilizado apenas para armazenar o nome do construtor. Não é avaliado diretamente.
AtomExpression	• expression:Expression	Retorna a avaliação da expressão armazenada.

Esta última classe de unidades de compilação representa possíveis alternativas de uma expressão case. Não geram código IR diretamente, apenas armazenam a expressão, que caso selecionada, será executada.

Tabela 17. Unidades de compilação que representam alternativas

Classe	Campos	Geração da IR (método Evaluation)
AltPrimitive	• literal:Interger	Sua avaliação retorna avaliação da expressão armazenada.
AltAlgebraic	• const:Constructor • vars:List<Atomvariable> • expression:Expression	Armazena variáveis que devem ser preenchidas antes da avaliação da expressão. Sua avaliação retorna avaliação da expressão armazenada.
AltDefault	• expression:Expression	Sua avaliação retorna avaliação da expressão armazenada.

APÊNDICE B -PROFILER DE MEMÓRIA

O Código 29 cria uma ferramenta que gera o perfil de consumo de memória de um programa. Utiliza, para tanto, contadores de desempenho disponibilizados pelo framework .NET. São retornados cinco valores, que corresponde, respectivamente, ao máximo de memória *heap* alocado, a média das porcentagens de tempo gasto em coleta de lixo e o número de coletas realizadas nas gerações 0, 1 e 2.

```
1   class MemoryProfiler
2   {
3       static float maxMem;
4       static float totalGCTime;
5       static int numGCTime;
6       static int ger0;
7       static int ger1;
8       static int ger2;
9       private static PerformanceCounter gcTimerCounter;
10      private static PerformanceCounter heapBytesCounter;
11      private static PerformanceCounter ger0Counter;
12      private static PerformanceCounter ger1Counter;
13      private static PerformanceCounter ger2Counter;
14      static void Main(string[] args)
15      {
16          if (File.Exists(args[0]))
17          {
18              string instanceName = args[0].Remove(args[0].Length-4);
19
20              // Cria contadores
21              gcTimerCounter =
22                  new PerformanceCounter(".NET CLR Memory", "% Time in GC");
23              gcTimerCounter.InstanceName = instanceName;
24              heapBytesCounter =
25                  new PerformanceCounter(".NET CLR Memory", "# Bytes in all
26                     Heaps");
27              heapBytesCounter.InstanceName = instanceName;
28              ger0Counter =
29                  new PerformanceCounter(".NET CLR Memory", "# Gen 0
30                     Collections");
31              ger0Counter.InstanceName = instanceName;
32              ger1Counter =
33                  new PerformanceCounter(".NET CLR Memory", "# Gen 1
34                     Collections");
35              ger1Counter.InstanceName = instanceName;
36              ger2Counter =
37                  new PerformanceCounter(".NET CLR Memory", "# Gen 2
38                 Collections");
39              ger2Counter.InstanceName = instanceName;
40
41              //Cria thread para monitorar execução do programa
42              ThreadStart memoryOperation = new ThreadStart(GetMemoryCount);
43              Thread memoryThread = new Thread(memoryOperation);
44              memoryThread.Start();
45
46              //Executa o programa e espera pelo seu final
```

```
47              Process p = Process.Start(args[0]);
48              p.WaitForExit();
49              memoryThread.Abort();
50
51              //Imprime valores obtidos
52              Console.Write("{0:N0}\t", maxMem);
53              if (numGCTime > 0)
54              {
55                  Console.Write(totalGCTime / numGCTime + "\t");
56              }
57              else
58                  Console.Write("0");
59              Console.Write("{0:N0}\t", ger0);
60              Console.Write("{0:N0}\t", ger1);
61              Console.Write("{0:N0}\n", ger2);
62          }
63          else
64              Console.WriteLine("Programa não existe: " + args[0]);
65      }
66
67      static void GetMemoryCount()
68      {
69          while (true)
70          {
71              try
72              {
73                  float totalmemory = heapBytesCounter.NextValue();
74                  if (maxMem < totalmemory)
75                      maxMem = totalmemory;
76                  Thread.BeginCriticalRegion(); //Inicio operação unária
77                  totalGCTime += gcTimerCounter.NextValue();
78                  numGCTime++;
79                  Thread.EndCriticalRegion(); //Fim operação unária
80               //Armazena os ultimos valores para cada geração
81                  ger0 = (int)ger0Counter.NextValue();
82                  ger1 = (int)ger1Counter.NextValue();
83                  ger2 = (int)ger2Counter.NextValue();
84              }
85              catch (Exception){}
86              //Aguarda 10 milisegundos
87              Thread.Sleep(10);
88          }
89      }
90  }
```

Código 29. Ferramenta de profiler de memória.

APÊNDICE C -PLUGIN DE RECURSÃO ATRAVÉS DE DESVIOS

Plugin responsável por substituir chamadas recursivas por desvios incondicionais para o início da função. De forma a evitar redundância, no Código 30 é apresentado, apenas, o método Execute do plugin, o qual contém a parte funcional deste. Instruções de como construir o restante do plugin podem ser vistas na Seção 3.2.

A função *Execute* verifica se a instrução marcada se chama recursivamente (linhas 18 e 19), se este for o caso cria a instrução de desvio (linhas 53 a 102), caso contrário apenas adiciona uma instrução *tail* antes da chamada (linhas 23 a 43).

```
1  protected override void Execute(Phx.Unit unit)
2  {
3      if (!unit.IsFunctionUnit)
4          return;
5
6      Phx.FunctionUnit functionUnit = unit.AsFunctionUnit;
7      foreach (Phx.IR.Instruction instruction in
8            functionUnit.Instructions)
9      {
10         if (instruction is Phx.IR.CallInstruction)
11         {
12             TailCallExtensionObject extObj =
13                TailCallExtensionObject.Get(instruction);
14             if (extObj != null)
15             {
16                 //Verifica se função chamada tem o mesmo nome da função que a
17                 //contém
18                 if (instruction.AsCallInstruction.FunctionSymbol !=
19                     functionUnit.FunctionSymbol)
20                 {
21                     //Se não tiver o mesmo nome é inserido uma instrução tail
22
23                     Phx.IR.Instruction tailInstruction =
24                         Phx.IR.ValueInstruction.New(functionUnit,
25                            Phx.Targets.Architectures.Msil.Opcode.TAILPREFIX);
26
27                     instruction.InsertBefore(tailInstruction);
28
29                     //Remove a instrução que armazena o valor de
30                     //retorno da função
31                     if (instruction.Next.Opcode ==
32                        Phx.Targets.Architectures.Msil.Opcode.st)
33                         instruction.Next.Remove();
34
35                     //Busca a instrução de retorna da função
36                     Phx.IR.Instruction returnInstruction =
37                       instruction.Next;
38                     while (!returnInstruction.IsReturn)
39                         returnInstruction = returnInstruction.Next;
40                     instruction.InsertAfter(returnInstruction.Copy());
41
42                     //Desmarca a instrução
```

```
43                      instruction.RemoveExtensionObject(extObj);
44                  }
45                  else
46                      //Chama método responsável por gerar desvio
47                      InsereBranch(instruction, functionUnit);
48              }
49          }
50      }
51  }
52  //Método responsável por criar instruções de desvio
53  void InsereBranch(Instruction instruction,FunctionUnit functionUnit)
54  {
55      Operand varOp =
56          functionUnit.FirstEnterInstruction.DestinationOperandList;
57      Phx.Symbols.FunctionSymbol funcSym = instruction.FunctionSymbol;
58
59      //Cria lista com os argumentos passados à função
60      List<Operand> argsOpAux = new List<Operand>();
61      while (varOp!=null)
62      {
63          if (varOp.IsVariableOperand)
64          {
65              argsOpAux.Add(varOp);
66          }
67          varOp = varOp.Next;
68      }
69
70      //Inverte a lista de argumentos para que sejam armazenados corretamente
71      List<Operand> argsOp = new List<Operand>();
72      for (int i = 1; i <= argsOpAux.Count; i++)
73      {
74          argsOp.Add(argsOpAux[argsOpAux.Count - i]);
75      }
76
77      //Armazena os valores passados para a função
78      foreach (Operand op in argsOp)
79      {
80          if (op.IsVariableOperand)
81          {
82              Operand sourceOp =
83                  Operand.NewRegister(functionUnit, op.Type,
84                     Phx.Targets.Architectures.Msil.Register.SR0);
85              Instruction storeInstr =
86                  Instruction.NewUnary(functionUnit,
87                     Phx.Targets.Architectures.Msil.Opcode.st, op, sourceOp);
88              instruction.InsertBefore(storeInstr);
89          }
90      }
91      //Cria instrução de desvio p/ inicio da função
92      Instruction branchInstruction =
93          Instruction.NewBranch(functionUnit,
94             Phx.Targets.Architectures.Msil.Opcode.br,
95                functionUnit.FirstEnterInstruction.AsLabelInstruction);
96      instruction.InsertBefore(branchInstruction);
97
98      //Remove a instrução que armazena o valor de retorno da função
99      if (instruction.Next.Opcode == Phx.Targets.Architectures.Msil.Opcode.st)
100         instruction.Next.Remove();
101     instruction.Remove();
102 }
```

Código 30. Plugin que substitui recursão por desvios incondicionais.

Dissertação de Mestrado apresentada por Guilherme Amaral Avelino à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título, **"Integração de Linguagens Funcionais à Plataforma .NET Utilizando o Framework Phoenix"**, orientada pelo **Prof. André Luis de Medeiros Santos** e aprovada pela Banca Examinadora formada pelos professores:

Prof. Hermano Perrelli de Moura
Centro de Informática / UFPE

Prof. Ricardo Massa de Oliveira Lima
Departamento de Sistemas Computacionais / UPE

Prof. André Luiz de Medeiros Santos
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 8 de agosto de 2008.

Prof. FRANCISCO DE ASSIS TENÓRIO DE CARVALHO
Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.