**Plano de Curso - Doutorado em Ciência da Computação**

# Combinando Avaliação Parcial de Programas com Compilação Dinâmica

**Eliseu César Miguel**
eliseu@bcc.unifal-mg.edu.br

**Orientadora: Mariza Andrade da Silva Bigonha**
mariza@dcc.ufmg.br

**Co-orientador: Fernando Magno Quintão Pereira**
fpereira@dcc.ufmg.br

04/06/2010

## 1 Introdução

Compilação de código é o problema que consiste em transformar um programa escrito em uma linguagem de programação de alto nível, em uma cadeia de bits – zeros e uns – que serão lidos pelo processador de um computador [1]. Esta sequência de bits é denominada um programa em linguagem de máquina. Linguagens de programação são normalmente classificadas em duas categorias: *linguagens estaticamente compiladas* e *linguagens dinamicamente compiladas* [20]. Programas escritos em linguagens estaticamente compiladas são completamente lidos pelo compilador antes de serem transformados em programas de máquina. Exemplos de linguagens de programação deste tipo incluem C, C++, SML, Haskell além de uma vasta gama de outras linguagens de grande importância acadêmica e industrial. As linguagens dinamicamente compiladas também são comuns. Neste caso, programas são *interpretados*, isto é, ao contrário de serem diretamente traduzidos para linguagem de máquina, estes programas são lidos por um outro programa, o interpretador. O interpretador se encarrega de executar todas as ações previstas pelo programa interpretado no *hardware* alvo. O interpretador pode decidir, durante o processo de interpretação, compilar parte de um programa para linguagem de máquina. Este processo chama-se *compilação dinâmica*, ou compilação *just-in-time*. Dentre os exemplos de linguagens dinamicamente compiladas, citam-se: PHP; Perl; Python; JavaScrip; Prolog; Lua e Java. É importante notar que esta distinção entre linguagens de programação não é rígida. Java, por exemplo, é uma linguagem normalmente compilada dinamicamente, embora existam também sistemas que compilem Java estaticamente.

Linguagens dinamicamente compiladas como PHP, Java, Perl, Bash, JavaScript e Lua estão entre as mais populares do mundo. Java, por exemplo, é a segunda colocada no sítio *Tiobe*, que mede o índice de popularidade de linguagens de programação [14]. Em outro exemplo, PHP é a primeira colocada em buscas de emprego em programação no sítio *Craigslist* [2]. JavaScript [8], uma linguagem dinamicamente compilada, é utilizada por programadores em todo o mundo para a validação de formulários da Web. Linguagens tais como PHP, Perl e Bash são populares porque suas curvas de aprendizado tendem a ser mais suaves que as curvas de aprendizado de linguagens como C, C++ e Fortran. A popularidade de Java deve-se, em primeiro lugar, à sua portabilidade, e em segundo lugar, às poderosas abstrações para a programação orientada por objetos que a linguagem provê.

A despeito da grande popularidade, linguagens dinamicamente compiladas tendem a ser menos eficientes que linguagens estaticamente compiladas. Tal fato não está diretamente ligado às definições da linguagem propriamente dita, mas ao ambiente de execução onde tal linguagem é utilizada. Programas escritos nestas linguagens são interpretados antes de serem compilados. A interpretação é normalmente um processo mais lento que a execução de programas escritos em código de máquina. Tal lentidão deve-se ao fato da interpretação de cada instrução do programa alvo demandar a execução de dezenas, quando não centenas, de instruções da máquina real.

Melhorar o desempenho de linguagens dinamicamente compiladas pode ser visto, dado o exposto, como tarefa muito atraente tanto em nível de pesquisa como alvo prático. O desempenho de um programa em tal cenário é normalmente medido pelo tempo de compilação dinâmica somado ao tempo de execução do programa. O desenvolvimento de técnicas de geração dinâmica de código que reduzam o tempo de execução do programa sem comprometer o tempo gasto durante sua compilação pode ser visto como um grande avanço para as aplicações que fazem uso de compiladores dinâmicos atualmente, bem como mais um importante elemento na evolução da Web.

## 1.1 Motivação

A principal motivação para este projeto é o surgimento da técnica de compilação dinâmica conhecida como *compilação de trilhas*, ou *trace compilation* [5, 9, 11, 10]. Inicialmente a compilação dinâmica era realizada sobre funções. Um programa é composto por diversas *funções*, isto é, sub-programas que recebem parâmetros de entrada e os utilizam para gerar um valor de saída. Durante a interpretação de um programa, o interpretador se encarrega de inferir quais são as funções mais utilizadas e, a partir desta informação, ele compila tais funções para código de máquina. Denomina-se, neste texto, o modo de compilação baseado em funções por *compilação dinâmica tradicional* [1]. A compilação de trilhas trouxe uma nova proposta de geração de código, diferente da compilação dinâmica tradicional, e abriu diversas oportunidades de pesquisa, algumas das quais tornaram-se nossa intenção abordar neste projeto.

É nossa intenção disponibilizar as técnicas e algoritmos de compilação desenvolvidos durante este projeto à comunidade de *software livre* . Tanto o compilador de JavaScript usado pela Fundação Mozilla, o *TraceMonkey*, quanto o interpretador da linguagem Lua [1] criado na PUC-Rio possuem licenças de código aberto. Nós acreditamos que este projeto é relevante porque estas linguagens, principalmente JavaScript, são utilizadas por milhares de programadores em todo o mundo. Portanto, um aumento na eficiência destes ambientes de programação tem impacto real na indústria de informática.

## 1.2 Objetivo

O objetivo deste projeto é aumentar a eficiência de linguagens dinamicamente compiladas, tais como Java, JavaScript, Lua, PHP e Ruby. Para isso, pretende-se projetar, implementar e testar novas otimizações de código específicas para tais ambientes de execução. Esperamos, desta forma, contribuir para que programadores possam continuar desfrutando de linguagens de programação flexíveis e expressivas, e que sejam, ao mesmo tempo, eficientes.

## 1.3 Definição do Problema

Com foco na compilação dinâmica e com o surgimento da compilação de trilhas, técnicas tradicionais utilizadas na compilação estática podem oferecer bons recursos na otimização de códigos parcialmente compilados.

Como exemplo, a avaliação parcial de programas, técnica de geração automática de código com objetivo de aumentar a eficiência de programas em relação ao tempo de execução [16], discutida na Seção 2.2, pode ser melhor explorada em compiladores dinâmicos. Isso porque

[1] http://www.lua.org/

em tempo de execução, muitas das entradas dinâmicas do programa já podem ser manipuladas quando se deseja compilar um trecho de código. Além disso, a compilação de trilhas muda o foco da compilação dinâmica tradicional que traduzia funções inteiras para código de máquina. Nesse caso, uma função torna-se geral e é invocada com parâmetros conhecidos somente após sua compilação.

Conhecendo, por exemplo, o valor de uma variável de controle de um laço de repetição, a técnica de *loop unrolling* [13], discutida na Seção 2.3, pode oferecer a geração de um código de máquina com o desdobramento de todos os testes no fluxo da repetição, diferindo de sua aplicação tradicional quando apenas se aproveitava o paralelismo entre as instruções [13].

Assim, pretende-se investigar e desenvolver técnicas para otimização de compiladores baseados em compilação dinâmica. Acredita-se que a compilação de trilhas pode ser aplicada com êxito em compiladores para linguagens como, por exemplo, JavaScript e Lua. Espera-se, também, que a combinação de compilação de trilhas com avaliação parcial de programas e outras técnicas de otimização permita o desenvolvimento de compiladores dinâmicos mais eficientes.

## 1.4 Solução Proposta

Para cumprimento do Plano de Trabalho, no que se refere à pesquisa, seguem-se indicações das linhas de investigação pretendidas:

- Investigar e dominar o estado da arte em compilação de trilhas afim de identificar a abordagem teórica mais apropriada na busca de melhorias de desempenho em compiladores dinâmicos como os das linguagens *JavaScript* e *Lua*, gerando produtos de código livre que contribuam para a comunidade de software livre.

- Investigar técnicas de *alocação de registradores* e *loop unrolling* que se oferecem na literatura como base alternativa, em baixo nível, para que, combinados com técnicas de compilação de trilhas, ampliem as alternativas de otimização.

- Investigar e dominar técnicas de *avaliação parcial de programas* que surgem como alternativa para tratar trechos de códigos a serem compilados dinamicamente.

## 1.5 Equipe de Pesquisa

O grupo de pesquisa em compiladores do Laboratório de Linguagens de Programação do DCC/ UFMG vem, desde Março de 2009, realizando pesquisas na área de linguagens dinamicamente tipadas. Tais trabalhos já produziram resultados tanto no âmbito teórico, que permitiram a escrita de artigos técnicos, quanto no âmbito prático. A título de exemplo deste último, atualmente existe uma relação estreita entre pesquisadores do DCC/UFMG e os desenvolvedores da fundação Mozilla. O estudante de mestrado Marcos Rodrigo Sol já produziu, em sua pesquisa, códigos para o TraceMonkey, compilador JavaScript utilizado no Mozilla Firefox. Além disso, pesquisadores do DCC/UFMG foram os autores da análise que remove testes de overflow no TraceMonkey [2].

Como se sabe, tanto é amplo o escopo da pesquisa quanto o do desenvolvimento do trabalho que se propõe. Contar com a troca de experiências e dedicação para realizar tal proposta é essencial.

Assim, são citados, a seguir, os nomes dos componentes da equipe envolvidos no projeto que hora propomos:

- professora Mariza A. S. Bigonha;

- professor Fernando M. Q. Pereira;

[2] $https://bugzilla.mozilla.org/show_bug.cgi?id = 536641$

- Marcos Rodrigo Sol Souza (mestrando), e;
- Eliseu César Miguel (candidato ao doutorado).

Além destes, está previsto um estudante de iniciação científica da UFMG, a ser definido, aguardando que sejam cumpridos critérios de seleção para o Programa de Bolsa de Iniciação Científica desta Instituição.

### 1.6 Contribuições Pretendidas

Partindo de um conjunto de técnicas de otimização validadas para a compilação dinâmica, torna-se possível o desenvolvimento de um compilador dinâmico totalmente focado na compilação de trilhas, algo sugerido para o futuro da Web por Chang e outros em [5]. Tal desenvolvimento traria para a comunidade de software livre uma nova ferramenta que promete ser mais eficiente e inovadora.

Além disso, como a comunidade de software livre trabalha voluntariamente e em equipes espalhadas por vários países, mais que oferecer uma ferramenta, o conteúdo da codificação que se pretende gerar poderá ser usada como ponto de partida para um projeto que, como o TraceMonkey [15], fomente novas pesquisas na área de otimização de compilação dinâmica. Isso permitiria, ao longo do tempo, várias melhorias no compilador inicial. Como exemplo, pode-se citar pesquisas em compilações mais especializadas para arquiteturas distintas em busca de aumentar a otimização associando o código gerado pelo compilador de trilhas aos recursos conhecidos do hardware, como seu banco de registradores e seus canais de *pipeline*.

Finalmente, sabe-se que as estratégias e técnicas para a compilação de trilhas são propostas recentes. Assim, é possível que estas sejam melhoradas ou até mesmo que novas opções surjam em busca de se alcançar melhor desempenho associando o interpretador ao compilador de trilha.

## 2 Revisão da Literatura

O uso de compilação dinâmica não é novidade. A idéia surgiu na década de 60 e vários casos de uso dessa abordagem em sistemas de software são citados e contextualizados em [3]. Também, as técnicas de *loop unrolling* e alocação de registradores são fortemente consolidadas na área de computação e estudadas nos programas de graduação e pós graduação em Ciência da Computação, bem como descritas em livros clássicos como em [13].

A seguir, descrevem-se brevemente sobre compilação de trilhas, avaliação parcial de programas, *loop unrolling* e alocação de registradores. Bibliografias básicas para tais assuntos também são citadas.

### 2.1 Compilação de Trilhas

Durante a execução de um programa, suas instruções são enviadas ao processador do computador. Os laços de repetição, por exemplo, normalmente exigem que algumas instruções sejam reenviadas várias vezes para a execução. No caso da compilação estática, as instruções que pertencem ao escopo de algum laço de repetição são todas convertidas em instruções de máquina, permitindo que os desvios ocorridos direcionem a execução para um bloco de instruções em código binário já existente.

A compilação de trilhas permite evitar que todas instruções em um laço de repetição sejam compiladas durante o processo de compilação dinâmica. Fazendo uso do *grafo de fluxo de controle*, é possível identificar um caminho mais requisitado e gerar instruções de máquina direcionadas para o caminho escolhido. Isso permite que se economize esforço computacional deixando de compilar partes da iteração que em alguns casos deixariam de fazer parte da linha de execução. Assim, a compilação dinâmica de trilhas consiste em compilar somente os caminhos mais executados de um programa, deixando para o interpretador os menos executados.

A Figura 1(a) exibe um exemplo de um laço de repetição e a Figura 1(b) exibe seu grafo de fluxo de controle.

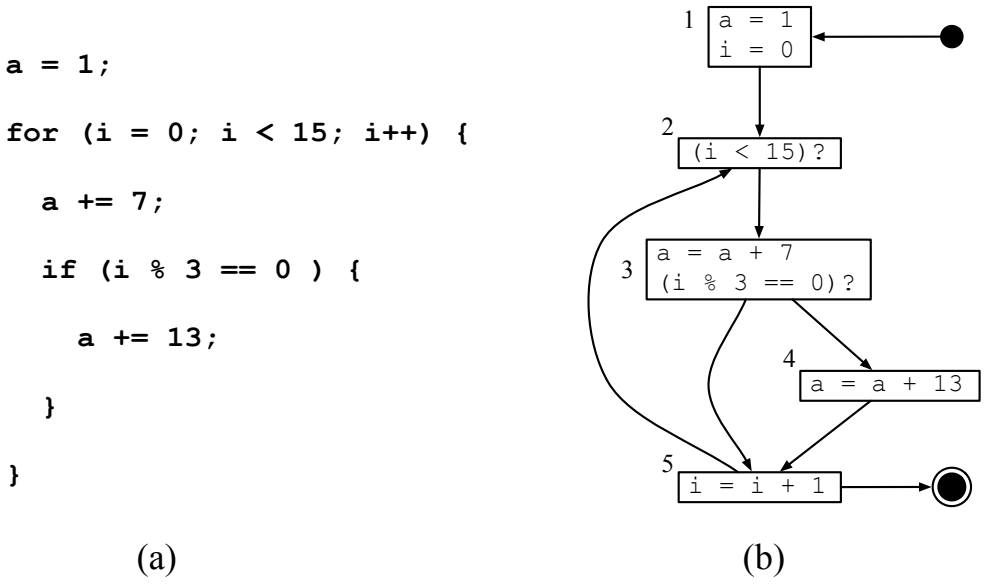


Figura 1: Um exemplo de programa (direita) com o respectivo grafo de fluxo de controle (esquerda).

Dois exemplos de trilhas que poderiam ser compiladas considerando o resultado da execução do exemplo na Figura 1 são as sequência definida pelos blocos [2,3,5] que será executada 10 vezes e [2,3,4,5], que ganha a atenção da execução 5 vezes.

## 2.2 Avaliação Parcial de Programas

A avaliação parcial de programas é uma técnica de geração automática de código com objetivo de aumentar a eficiência de programas em relação ao tempo de execução [16]. Fazendo uso de informações estáticas na compilação de um programa $P$, a avaliação parcial de programas permite gerar um programa especializado $P_e$ sobre os dados estáticos de modo que, ao receber em tempo de execução as entradas restantes, $P_e$ fornecerá a mesma computação de $P$. Dessa forma, pode-se alcançar grande ganho em eficiência na execução de $P$, pois sendo conhecida parte dos dados de entrada de $P$, as estruturas que dependem apenas desses dados podem ser previamente computadas e o programa especializado, $P_e$, conter apenas o código necessário para processar os dados ainda não conhecidos.

## 2.3 *Loop Unrolling*

O *loop unrolling*, originalmente concebido para maximizar a quantidade de paralelismo em um programa [17], é uma técnica antiga e bem consolidada. Em seu livro, Hennessy e Patterson [13] descrevem o uso de *loop unrolling* na fase de compilação para tirar proveito do paralelismo existente entre os canais do *pipeline* de execução no hardware.

Com a avaliação parcial de programas, essa técnica pode aumentar ainda mais a eficiência que oferece. Isso porque, conhecendo os valores que controlam o fluxo de execução da repetição, é possível gerar códigos automáticos com a avaliação parcial de programas desenrolando o laço de forma mais precisa.

### 2.4 Alocação de Registradores

Alocação de registradores é um problema que os compiladores enfrentam quando se faz necessário encontrar espaço na área de armazenamento real do hardware para alocar valores usados em um programa [1]. Duas possíveis soluções para tal problema são: ou variáveis são associadas aos registradores ou são mapeadas na memória. Apesar de oferecerem menor tempo de acesso para leitura e escrita se comparados com a memória, os registradores são normalmente em quantidade limitada nas arquiteturas de computadores, enquanto a memória é virtualmente ilimitada. Assim, é importante que os compiladores maximizem a utilização dos registradores priorizando os dados mais acessados para obtenção de maior eficiência na execução de programas.

## 3 Trabalhos Relacionados

Recentemente, o surgimento da compilação por trilhas trouxe novas opções na forma de se proceder a compilação dinâmica de programas. Ao contrário da compilação dinâmica tradicional que basiamente atua em módulos como funções, a compilação de trilhas oferece alternativa para definir trechos de códigos, aqui chamados de trilhas, que são requisitados com maior frequência durante a execução de um programa. Agora, não mais se faz necessário que o trecho compilado seja, por si só, um módulo. Como exemplos desses trechos, temos os *loops* que, em seções críticas, podem melhorar o desempenho na execução do programa se compilados.

Em 2004, Psyco [18] publica a utilização de compilação de trilhas para a linguagem dinâmica Python. Neste caso, não houve a preocupação em identificar *loops* em específico. Ao contrário, os *loops*, nesta proposta, são transformados em uma recursão comum antes da execução de todas operações.

Os resultados com compilação de trilhas começam a ganhar mais destaques em meados de 2006 quando trabalhos de Andreas Gal e outros [12, 11] descrevem técnicas e estruturas para se implementar a compilação por trilhas.

Ainda em 2006, em sua tese de doutodo, Andreas Gal [9] faz uso de compilação de trilhas aplicadas na compilação da linguagem Java propondo nova solução para se obter uma verificação mais eficiente de *bytecode* dessa linguagem.

Mais recentemente, Gal e outros [10] apresentam e avaliam uma abordagem de compilação de trilhas para linguagens dinâmicas, quando também são descritas soluções e estruturas que permitem: fazer o controle no fluxo de execução das trilhas compiladas; identificar uma nova trilha a ser compilada; e abortar a execução de uma trilha caso ocorra uma exceção que exija redirecionamento na execução para o interpretador. Um caso desses pode ocorrer quando um desvio direciona a execução para um trecho não compilado.

Em relação à avaliação parcial de programas, diversas variações desta técnica já foram utilizadas, com diferentes graus de sucesso, em otimização de código na compilação dinâmica. A avaliação parcial é especialmente útil no contexto de um compilador dinâmico, uma vez que alguns dos valores manipulados pelo programa são conhecidos. Neste caso, a avaliação parcial é chamada *especialização por necessidade* [18]. Como exemplo de compilador que se beneficia da especialização por necessidade pode-se citar o Psyco para linguagem Python [18], além de compiladores de Matlab [7, 6] e Maple [4].

Uma utilização da avaliação parcial em compiladores dinâmicos é na especialização de tipos de dados. Isso porque quando o compilador sabe que um determinado valor pertence a um certo tipo, ele pode usar este tipo diretamente, em vez de recorrer às técnicas populares e ineficientes como: *boxing* e *unboxing*. Apesar de antigo, este tipo de especialização foi usado recentemente em JavaScritp [10] e Matlab [6], o que contribui para validar os benefícios que a avaliação parcial de programas pode oferecer na compilação dinâmica.

Atualmente, o compilador dinâmico utilizado pelo Firefox para a linguagem JavaScritp, o SpiderMonkey, evolui ganhando novas opções de compilação baseadas em trilhas em um projeto chamado TraceMonkey [15]. Como exemplo, Rodrigo Sol [19], propõe otimizar códigos com-

binando a compilação de trilhas com avaliação parcial de programas. Assim, seus algoritmos poderão ser associados ao compilador do Mozilla contribuindo na busca de mais eficiência.

Apesar do recente projeto TraceMonkey permitir a compilação dinâmica de trilhas, nossa proposta pretende ampliar ainda mais as possibilidades de otimização de códigos para tais compiladores. Isso porque, ao contrário de basear-se em um compilador já existente e assim extendê-lo como ocorre entre o SpiderMonkey e o TraceMonkey, agora propõe-se projetar e desenvolver um compilador totalmente baseado na compilação de trilhas. Além de um grande diferencial, pode-se, com isso, evitar o uso de técnicas ultrapassadas e implementadas em compiladores atuais. Tal trabalho permitirá tirar maior proveito da compilação de trilhas, associando ao compilador concebido nesse paradigma as consagradas técncias de otimização citadas nesse plano.

A seguir, o cronograma previsto para o desenvolvimento do plano de trabalho.

# 4 Cronograma

As atividades a serem desenvolvidas durante o programa de doutorado, caso eu venha a ser admitido no programa, são:

- $1^o$ **Ano**. Este período é necessário para cursar os créditos que completam a carga horária exigida. As disciplinas que pretendo cursar são de extrema relevância no escopo do trabalho proposto. São elas:
  - Semântica Formal - (DCC-880) com 4 créditos;
  - Sistemas Operacionais - (DCC-816) com 4 créditos
  - Teoria das Linguagens - (DCC-874) com 4 créditos;
  - Compiladores - (DCC-886) com 4 créditos;
  - Tópicos em Compiladores - (DCC-888) com 4 créditos;

  Estas disciplinas, juntamente com as já cursadas no mestrado, permitirão uma base sólida para a qualificação, pesquisa e desenvolvimento do doutorado.

- $2^o$ **Ano**. No segundo período de 2011, a atenção deverá ser dada à qualificação. Reserva-se, para isso, um semestre. O primeiro período de 2012 será destinado ao envolvimento com o projeto de pesquisa, possibilitando a escrita da proposta e, consequentemente, sua defesa.

- $3^o$ **Ano**. Esta etapa deverá ser totalmente destinada à pesquisa e desenvolvimento do trabalho. Prevê-se, para este ano, a realização de um doutorado sanduíche durante seis meses. Uma possibilidade seria trabalhar sob a co-orientação do prof. Fabrice Rastello (ENS de Lion, França) e outra possibilidade seria trabalhar sob a co-orientação do prof. Jens Palsberg (USA), ambos com grande domínio e dedicação na área em que pretendo atuar.

- $4^o$ **Ano**. Finalmente, o último ano será importante para finalizar a pesquisa, escrever a tese, submissão de artigos para publicação e defesa da tese.

A Tabela 1 exibe as atividades previstas relacionando-as aos semestres em que devem ocorrer.

Para contribuir no cumprimeto do plano de trabalho, algumas das disciplinas cursadas durante o mestrado poderão ser aproveitadas em um momento oportuno compondo parte dos créditos exigidos pela UFMG. São elas:(i) Engenharia de Software (4 créditos); (ii) Projeto e Análise de Algoritmos (4 créditos); (iii) Inteligência Artificial (4 créditos); (iv) Arquitetura e Organização de Computadores (4 créditos); (v) Redes de Computadores (4 créditos); e (vi) Tópicos em Otimização (Metaheurística e Decisão Multicritério).

| Atividades | Semestres | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Disciplinas | x | x | | | | | | |
| Estudo e realização da qualificação | | | x | | | | | |
| Estudo da bibliografia básica | | | x | x | x | | | |
| Escrita e defesa da proposta de tese | | | | x | | | | |
| Pesquisa e desenvolvimento do trabalho | | | | x | x | x | x | |
| Período Sanduíche | | | | | x | | | |
| Escrita da tese | | | | | | | x | x |
| Escrita de artigos | | | | | | | x | x |
| Defesa da tese | | | | | | | | x |

Tabela 1: Cronograma do Trabalho

# Referências


[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.

[2] M. Authors. Programming language popularity, 2009. http://langpop.com/.

[3] J. Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, 2003.

[4] J. Carette and M. Kucera. Partial evaluation of maple. In *PEPM '07: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 41–50, New York, NY, USA, 2007. ACM.

[5] M. Chang, E. Smith, R. Reitmaier, M. Bebenita, A. Gal, C. Wimmer, B. Eich, and M. Franz. Tracing for web 3.0: trace compilation for the next generation web applications. In *VEE*, pages 71–80. ACM, 2009.

[6] M. Chevalier-Boisvert, L. J. Hendren, and C. Verbrugge. Optimizing matlab through just-in-time specialization. In *CC*, pages 46–65. Springer, 2010.

[7] D. Elphick, M. Leuschel, and S. Cox. Partial evaluation of matlab. In *GPCE*, pages 344–363. Springer-Verlag New York, Inc., 2003.

[8] D. Flanagan. *JavaScript: The Definitive Guide*. O'Reilly, 4 edition, 2001.

[9] A. Gal. *Efficient Bytecode Verification and Compilation in a Virtual Machine*. PhD thesis, University of California, Irvine, 2006.

[10] A. Gal, B. Eich, M. Shaver, D. Anderson, B. Kaplan, G. Hoare, D. Mandelin, B. Zbarsky, J. Orendorff, J. Ruderman, E. Smith, R. Reitmair, M. R. Haghighat, M. Bebenita, M. Change, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *PLDI*, pages 465 – 478. ACM, 2009.

[11] A. Gal and M. Franz. Incremental dynamic code generation with trace trees. Technical Report 06-16, University of California, Irvine, 2006.

[12] A. Gal, C. W. Probst, and M. Franz. Hotpathvm: an effective jit compiler for resource-constrained devices. In *VEE*, pages 144–153, 2006.

[13] D. A. Hennessy, J. L. ; Patterson. *Arquitetura de Computadores. Uma abordagem Quantitativa*. Editora Campus, 2003.

[14] P. Jansen. Tiobe code, 2009. http://www.tiobe.com/.

[15] JavaScript. TraceMonkey-Mozilla Wiki, 2008. https://wiki.mozilla.org/JavaScript:TraceMonkey/.

[16] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1st edition, 1993.

[17] K. Kennedy and R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001.

[18] A. Rigo. Representation-based just-in-time specialization and the psyco prototype for python. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 15–26, New York, NY, USA, 2004. ACM.

[19] M. R. S. Souza. Otimizações de código para compiladores de trilhas - proposta de dissertação de mestrado. Master's thesis, Universidade Federal de Minas Gerais - UFMG, 2010.

[20] A. B. Webber. *Modern Programming Languages - A practical introduction*. Franklin Beedle and Associates, 1 edition, 2005.

Assinaturas:

Estudante: _____
Eliseu César Miguel

Orientadora: _____
Mariza Andrade da Silva Bigonha

Co-orientador: _____
Fernando Magno Quintão Pereira