Universidade Estadual de Maringá
Centro de Tecnologia
Departamento de Informática
Bacharelado em Ciência da Computação

Trabalho de Conclusão de Curso - TCC 2010

Tcl JIT

Relatório Técnico 1

Guilherme Henrique Polo Gonçalves
Orientador: Prof. Dr. Anderson Faustino da Silva

Guilherme Henrique Polo Gonçalves

Prof. Dr. Anderson Faustino da Silva

Maringá, 15 de julho de 2010

Tcl JIT

Guilherme Henrique Polo Gonçalves
Orientador: Prof. Dr. Anderson Faustino da Silva

Resumo

Linguagens de programação interpretadas tem trocado performance por maior expressividade, flexibilidade, dinamicidade, entre outros. Reescrita de trechos de código críticos em linguagens compiladas tem sido empregada como meio de reduzir o impacto da máquina virtual no tempo de execução de programas. Diante disso, um compilador JIT para a linguagem de programação Tcl é proposto como forma de diminuir a necessidade de tal reescrita de código. Um modo misto de execução é escolhido, fazendo com que execução de código de máquina, gerado por esse compilador dinâmico, e a interpretação pura se alternem. Procedimentos são definidos como os limites de inicio e término de compilação, e a compilação baseada em regiões é rapidamente mencionada. Simplicidade, manutenibilidade e flexibilidade são características que serão seguidas na construção dessa ferramenta. Algumas conclusões iniciais a respeito de máquinas virtuais baseadas em pilha e também sobre aquelas baseadas em registradores são feitas de acordo com o desenvolvimento do projeto até o momento.

Palavras-chave: Tcl, JIT, código de máquina

1 Introdução

Linguagens de programação interpretadas tem sacrificado performance em favor de um alto nível de abstração do funcionamento da máquina envolvida, possibilitando maior expressividade, facilidade de desenvolvimento, portabilidade, flexibilidade, dinamicidade, entre outros. Especificamente a linguagem de programação `Tcl` (*Tool Command Language*), teve como um de seus maiores objetivos a facilidade de incorporação (*embedding*) a programas que desejassem ter uma linguagem de comandos (OUSTERHOUT, 1989). Uma interface simples e extensível para aplicações em linguagem `C` era o fator motivante de seu uso. Programas inteiramente em `Tcl` eram vistos como pequenos scripts, muitos talvez de uma linha no máximo (OUSTERHOUT, 1989). Entretanto, aplicações em `Tcl` com milhares de linhas, como por exemplo exmh (WELCH, 1995), OpenACS (HERNÁNDEZ, 2005) ou mesmo a suíte de testes do GDB (*GNU Debugger*) (GILMORE; SHEBS, 2010), tem surgido e a reescrita de trechos críticos em `C` vem sendo aplicada para reduzir o impacto da máquina virtual no tempo de execução.

Outra forma de melhorar o desempenho de linguagens e, portanto, reduzir a necessidade de reescrita de código em linguagens compiladas, é através da utilização da compilação JIT (*Just-In-Time*) que realiza tradução de código sob demanda. No caso desse trabalho, estamos interessados na tradução de *bytecodes* da máquina virtual `Tcl` para código de máquina durante o tempo de execução. Informações acerca do programa em execução são coletadas conforme necessário para dirigir a compilação dinâmica. Portanto, linguagens tipicamente difíceis de serem analisadas e compiladas estaticamente, devido a uso de, por exemplo, tipagem dinâmica ou escopo dinâmico, ganham a oportunidade de melhoria de desempenho com uso de tal sistema de compilação.

O trabalho presente pretende melhorar o desempenho da linguagem `Tcl`, reduzindo o tempo de decodificação e interpretação de *bytecodes* através da implementação e implantação de um compilador JIT na mesma. Diversos trabalhos, como (DEUTSCH; SCHIFFMAN, 1984) para a linguagem `Smalltalk`, Jalapeño (ALPERN et al., 2000) para `Java`, Psyco (RIGO, 2004) para `Python` ou a implementação do SELF-93 (HÖLZLE, 1994), demonstraram resultados bastante significativos ao implantar tal método de compilação. Pretende-se ainda manter um nível de manutenibilidade e flexibilidade adequado, permitindo extensões e futuro desenvolvimento sobre esse trabalho inicial.

Um compilador JIT requer que as estruturas internas sejam suficientemente eficientes, caso contrário não se torna viável o uso de um compilador otimizador em tempo de execução. As escolhas a cerca de quais representações intermediárias utilizar, como estruturar os dados, quais otimizações aplicar e algoritmos para diversas fases da compilação,

devem ser feitas de forma a conseguir balancear baixo tempo de compilação com código gerado de alta qualidade. Ainda há o quesito de consumo de memória principal, que, apesar da crescente capacidade disponível, geralmente costuma ser um recurso escasso em dispositivos embarcados. Sabe-se que o interpretador `Tcl` está presente em roteadores da Cisco, pois vem incluso no Cisco IOS (*Internetwork Operating Systems*), mas esse trabalho não tem como foco tal tipo de dispositivo e, portanto, consumo de memória não será um dos pontos levados em consideração. Porém, entre os fatores complicadores consideramos também a manutenabilidade do sistema. Um nível muito alto de abstração, como dito no ínicio do texto, impediria a utilização de um programa de desempenho crítico e, por outro lado, um nível muito baixo dificultaria a correção/detecção de problemas e melhorias gerais. Uma alternativa para esse problema vem sendo aplicada através do projeto PyPy (RIGO; PEDRONI, 2006), onde um interpretador para uma linguagem qualquer é escrito em `RPython` (uma implementação mais restrita da linguagem `Python`) e o PyPy realiza a tradução do mesmo para a linguagem `C` incluindo (atualmente) juntamente um compilador JIT específico para arquitetura IA-32. Entretanto, um dos objetivos do trabalho discutido aqui é analisar como um sistema de tamanho reduzido compete com sistemas mais robustos. Não se tem a intenção de fornecer um ambiente de alto nível para construção de outras máquinas virtuais com ou sem compiladores JIT para `Tcl`, mas sim uma implementação específica e direta. A escolha da linguagem `C` reflete esse objetivo porque não adiciona novas dependências ao núcleo da linguagem `Tcl` além de ser considerada bastante eficiente.

Procura-se visar a simplicidade, de forma que o trabalho desenvolvido possa servir de base para expansão a novas arquiteturas e também para aplicação de técnicas de compilação diversas sob a `Tcl`. Na representação intermediária, quádruplas são escolhidas para formar blocos básicos e esses, por sua vez, são utilizados para definir grafos de fluxo de controle que servem de entrada a outra representação escolhida – SSA (*Static Single Assignment*). A arquitetura IA-32 foi escolhida como alvo, pois está presente em boa parte dos computadores de uso pessoal, porém a linguagem `Tcl` atualmente executa em várias outras arquiteturas. Nesse sentido, permitir portar para IA-64, ARM, e outras, sem tornar a tarefa demasiadamente complicada faz parte da simplicidade que se pretende práticar.

A seção 2 realiza uma revisão bibliográfica relevando trabalhos que, de alguma forma, buscaram melhorar a performance da `Tcl`. Também descrevemos brevemente alguns projetos de compiladores JIT que apresentam alguma semelhança com o nosso. Em seguida, na seção 3 é descrito uma proposta um pouco mais detalhada a respeito desse compilador. Na seção 4 alguns detalhes do que já foi desenvolvido são apresentados, incluindo a estrutura atual para quádruplas e blocos básicos. As demais seções se destinam a men-

cionar dificuldades encontradas até qui, além de descrever o que será feito a partir desse ponto.

2 Revisão Bibliográfica

A linguagem `Tcl` já obteve ganhos de performance em diferentes estudos feitos. Um deles, que atualmente faz parte da implementação da linguagem, descrito em (LEWIS, 1996), é a geração e a interpretação de *bytecodes*. Anterior a esse trabalho, foi demonstrado em (SAH, 1994) que o *parsing* do código realizado a todo momento para sua reinterpretação e também a conversão excessiva entre tipos de dados, pois a `Tcl` trata tudo como string, eram os grandes consumidores do tempo de execução. Com esse trabalho feito, a linguagem passou a utilizar representação dupla para os valores presentes na execução do programa. Uma representação é interna, possivelmente mais eficiente para se trabalhar. A outra é a típica representação em string que a linguagem sempre usou. Caso uma delas não esteja disponível, a outra é utilizada para recriar essa representação se necessário.

Um trabalho mais recente, descrito em (VITALE; ABDELRAHMAN, 2004), lida com a eliminação do *overhead* de decodificação dos *bytecodes*, introduzido pelo trabalho descrito anteriormente, fazendo uso de *templates* que contém as instruções em código nativo utilizadas para interpretar cada *bytecode*. Esse código é obtido através da compilação do próprio interpretador `Tcl` e cada *template* é copiado múltiplas vezes, numa área de memória alocada em tempo de execução, conforme a quantidade de cada *bytecode* gerado. Nesse trabalho o interpretador foi modificado de forma a sempre executar somente tal código formado por uma concatenação de *templates*, eliminando o *overhead* de decodificação. Demonstrou-se que em certos testes o desempenho da linguagem pode melhorar em até 60% com a aplicação dessa técnica. Esse trabalho é provavelmente o mais próximo, quando considerando somente a `Tcl`, do que se pretende produzir aqui. Ele não gera código, mas cópia código já gerado por um compilador estático e replica conforme necessário, fazendo os devidos ajustes, em tempo de execução. Por um lado o tempo de "compilação" é bastante baixo, porém, não dá espaço para técnicas de otimização e assim limita o potencial de melhoria de desempenho.

Outros trabalhos, para diferentes linguagens, se assemelham mais com a proposta aqui discutida. A busca por máquinas virtuais de alta performance tem, atualmente, se dirigido principalmente a linguagem `Java`. É comum a presença de compiladores JIT em máquinas virtuais para essa linguagem, cada um com diferentes características. A JUDO (CIERNIAK; LUEH; STICHNOTH, 2000), faz uso de compilação dinâmica com dois tipos de compiladores e coleta informações em tempo de execução. O primeiro desses compilado-

res é um mais simples, que gera código rapidamente, destinado a compilação de métodos invocados pela primeira vez. O segundo compilador é utilizado quando informações coletadas indicam que certos métodos são executados muito frequentemente e, portanto, estes podem se beneficiar com a aplicação de otimizações. Essa recompilação dinâmica é feita com o intuito de balancear o tempo gasto na compilação com o tempo efetivamente gasto na execução do programa. Esse sistema trabalha com a compilação de métodos por inteiro, assim como o trabalho proposto aqui. Enquanto isso, o trabalho discutido em (SUGANUMA; YASUE; NAKATANI, 2003) avalia a aplicação de compilação dinâmica a regiões de código, evitando a compilação de trechos raramente executados. Além disso, esse sistema utiliza um modo misto de execução, onde interpretação e execução de código nativo se alternam. Nesse ponto, nosso trabalho e aquele em (SUGANUMA; YASUE; NAKATANI, 2003) se assemelham.

Nos dois trabalhos sobre JIT mencionados acima não há uma descrição a cerca das representações intermediárias (IR) utilizadas. Porém, um outro trabalho apresentado sobre a JVM (*Java Virtual Machine*) CACAO (KRALL; GRAFL, 1997), descreve algo parecido com a nossa proposta. De forma semelhante com a "TVM" (*Tcl Virtual Machine*), a JVM tem uma arquitetura de pilha e a CACAO faz uma conversão para uma representação orientada a registradores com uso de poucas instruções. O artigo por Suganuma et al. (2004) vai além, exibindo a evolução de uma JVM desenvolvida na IBM onde, inicialmente, era utilizado uma IR baseado em pilha, porém mais compacta que a representação em *bytecodes* da `Java`, e que mais tarde passou também a se basear em registradores. Os autores argumentam que para conseguir balancear performance e tempo foi necessário, além de outros avanços, fazer uso de 3 representações: aquela que já existia (chamada de EBC – *Extended Bytecode*) e de mais duas baseadas em registradores. Cada uma delas recebe um conjunto de otimizações, sendo feito propagação e cópias de constantes, eliminação de código morto, eliminação de verificação de exceção, e algumas outras, enquanto que na forma de quádruplas. A última dessas aparece na forma SSA, com os nós sendo formados por quádruplas (assim como o compilador proposto aqui fará).

3 Proposta

Para chegar ao código de máquina final, esse projeto se propôs em primeiro momento estudar e definir qual subconjunto da linguagem poderia se beneficiar mais com tal técnica. O subsistema de Entrada/Saída, por exemplo, dificilmente ganharia em desempenho com compilação dinâmica uma vez que o tempo gasto para transferência e aguardo por dispositivos costuma ser muito maior que o tempo das outras operações envolvidas. Por outro

lado, operações aritméticas podem ser bastante favorecidas. O trabalho feito no Psyco, mostra melhoria de 109 vezes no tempo de execução para aritmética de inteiros e 10,9 vezes em aritmética de ponto flutuante (RIGO, 2004). Após essa primeira análise, ve-se que entre todos os *opcodes* definidos cerca de 100 deles são de uso geral, enquanto que o restante é dedicado a alguns poucos comandos pré-definidos. Além disso, por volta de 20 deles realizam a mesma tarefa mas trabalham com operandos de tamanhos diferentes (1 ou 4 bytes). Também encontramos que dois *opcodes* são obsoletos, reduzindo um pouco mais o subconjunto de trabalho.

Em paralelo ao requisito acima, foi iniciado o projeto e implementação de uma representação intermediária de baixo nível. Num primeiro momento sugerimos utilizar um conjunto reduzido de instruções (RISC), assim como é feito na GNU lightning (FREE SOFTWARE FOUNDATION, 2007), para construir uma árvore de instruções mais próximas da máquina alvo. Parte dessa decisão inicial ainda se mantém, o uso de algo parecido com RISC, mas partimos para uso de quádruplas seguida da representação SSA. A escolha inicial por árvores foi devido a existência de diversos métodos para seleção de instruções que trabalham sobre essa forma de representação (ERTL; CASEY; GREGG, 2006), mas trabalhar com quádruplas se mostrou simples. O sistema de compilação dinâmica precisa ser capaz de determinar quando iniciar e parar a construção dessas representações. Nesse projeto a intenção é fazer uso dos limites de um procedimento como pontos de inicio e parada. Ainda aqui, é importante que o sistema colete informações, como de tipos utilizados, necessárias de forma a simplificar o código gerado, para que não repliquemos os resultados do trabalho feito em (VITALE; ABDELRAHMAN, 2004).

Após essas etapas pretende-se aplicar pelo menos algumas das otimizações clássicas tais como remoção de código morto, propagação de constantes e movimentação de código.

Chegamos, assim, na fase de seleção de instruções seguida de alocação de registradores. Há diversos algoritmos, como *Maximal Munch*, seleção com uso de programação dinâmica ou NOLTIS (KOES; GOLDSTEIN, 2008) para seleção de instruções além de vários outros, por coloração de grafos ou mesmo através de uma varredura linear (POLETTO; SARKAR, 1999) (com ou sem (WIMMER; FRANZ, 2010) desconstrução da representação SSA), para alocação de registradores. Ainda não está decidido quais dos algoritmos serão aplicados nesse projeto, porém as metas são duas: que a implementação permita utilizar diferentes algoritmos e que o tempo gasto para execução deles seja compatível com um sistema que consome tempo de execução do programa.

Finalmente temos a geração de código. O foco desse trabalho é a arquitetura IA-32, que possui uma ampla quantidade de instruções, extensões e diversas formas de endereça-

mento. Porém, não pretendemos fazer uso de todos os recursos disponíveis de forma a tornar factível a criação do gerador. Idéias e trechos de trabalhos anteriores, como Harpy (GRABMÜELLER; KLEEBLATT, 2007) ou o *Tiny Code Generator* (incluído nas versões mais recentes do QEMU (BELLARD, 2005)), podem ser reutilizadas aqui. Ainda, de forma semelhante com a etapa anterior, a infraestrutura permitirá a implantação de geração de código para outras arquiteturas.

4 Desenvolvimento

Inicialmente a proposta descrevia o uso de uma forma de representação intermediária em árvore. A intenção era, em uma etapa mais adiante, fazer uso de um algoritmo, entre os diversos existentes, para seleção de instruções que se baseiam em árvores. Junto com esse fato, observa-se que o uso de otimizações durante a geração de código não havia sido explicitado no texto anterior. Porém, a partir da leitura de diferentes artigos, essas decisões iniciais foram repensadas. Sendo assim, a etapa de especificação do compilador JIT definiu o uso de uma representação intermediária inicial na forma de quádruplas que, em seguida, é convertida para a forma SSA.

A representação em SSA tem sido considerada por vários compiladores. Um dos motivos, de acordo com Cytron et al. (1991), é a possibilidade de se executar certas otimizações clássicas de maneira eficiente após a conversão para essa forma. Porém, para construir tal representação, precisamos fornecer como entrada o grafo de fluxo de controle (CFG – *Control Flow Graph*) de um trecho de código específico. Sendo assim, chegamos a nossa outra decisão: o uso de quádruplas para compor blocos básicos que representam os nós do CFG. Essa representação em quádruplas é relativamente simples de se construir, além de apresentar uma característica relevante para a otimização que realiza movimentação de código: mover uma quádrupla tende a requerer menor esforço do do que rearranjar uma representação em forma de árvore.

Atualmente, na implementação bastante inicial do compilador, temos a seguinte estrutura (em C) que representa uma quádrupla:

```
struct Quadruple {
  Value *dest;
  unsigned char instruction;
  Value *src_a, *src_b;
  struct Quadruple *next;
};
```

Figura 1: Estrutura para uma quádrupla

O campo `instruction` tem tipo `unsigned char` pois atualmente representa um dos *bytecodes* da `Tcl` ou uma entre as instruções `JIT_INST_MOVE`, `JIT_INST_CALL`, `JIT_INST_GOTO`, `JIT_INST_JTRUE`, `JIT_INST_JFALSE`, `JIT_INST_ADD`, que podem ser representadas com um byte (todas elas tem um número associado menor que 256). O campo `next` existe para lidar de forma eficiente com casos de otimização que requerem a movimentação de código. Também temos os campos de tipo `Value`, permitindo, por enquanto, assumir valores inteiros, ponteiros para `Tcl_Obj` ou uma forma de registrador. Também temos uma estrutura simples para agrupar as quádruplas e formar os blocos básicos:

```
struct BasicBlock {
  int exitcount;
  struct Quadruple *quads, *lastquad;
  struct BasicBlock **exit;
};
```

Figura 2: Estrutura para um bloco básico

Como não há um campo "`next`" nessa abstração, guardamos o número de arcos, em `exitcount`, que saem de um bloco de modo a permitir o controle de sua travessia. O ponteiro `lastquad` é utilizado na construção do CFG, eliminando a necessidade de travessia das quádruplas presentes em um bloco para identificar se a última instrução é de desvio ou não. Atualmente, para visualizar os blocos básicos, há apenas a escrita de texto puro no terminal, sendo necessário utilizar algum programa para desenha-los. Em breve espera-se gerar código na linguagem `dot` ou fazer uso do aplicativo xfig para facilitar essa tarefa. Ainda assim é possivel analisar o que é gerado e, para isso, tomamos o código exemplo seguinte:

```
proc grayb {n} {
  for {set i 0} {$i < 2 ** $n} {incr i} {
    puts [expr {$i ^ ($i >> 1)}]
  }
}
```

Figura 3: Procedimento que exibe código Gray de números com até n bits

Após executar "`grayb 2`", obtemos o resultado esperado – `0 1 3 2` – mas também temos que a função `JIT_Compile` (nome segue o padrão de nomenclatura no fonte da `Tcl`) é, agora, chamada através da função `TclObjInterpProcCore` que foi levemente modificada. Essa última é invocada pela própria `Tcl` e fica responsável por chamar a função que realiza a interpretação de *bytecodes*, a `TclExecuteByteCode`. Pretende-se realizar a coleta de tipos durate a execução dessa última função mencionada, alterando-a de acordo

com as necessidades do compilador JIT. Enfim, ao término da função `JIT_Compile` o grafo de fluxo de controle exibido a seguir é obtido.

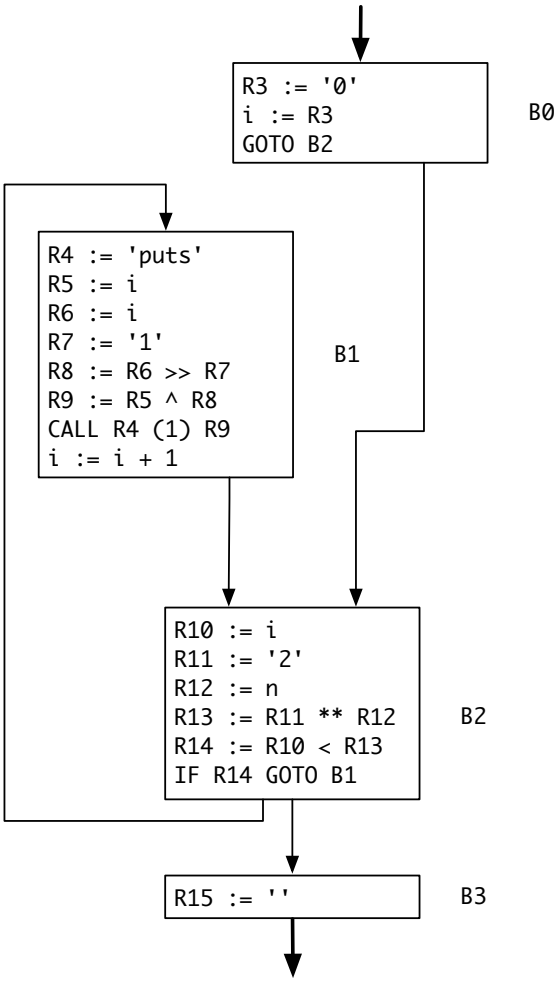

Figura 4: Blocos básicos e grafo de fluxo de controle construídos

Para se chegar nessa representação foi feita uma espécie de conversão de máquina virtual baseada em pilha para algo que deveria se assemelhar com uma máquina de infinitos registradores. Uma pilha temporária é utilizada para simplificar essa transição, sendo acessada e atualizada durante a construção de diversas instruções. Também são pré-alocados registradores virtuais para as variáveis locais. A implementação da `Tcl` disponibiliza uma lista dinâmica com todas essas variáveis locais, incluindo também os parâmetros formais da função, sendo necessário apenas atribuir índices a elas. O registrador `R1` está associado ao parâmetro `n`, mas nunca é acessado diretamente porque o programa exemplo não altera seu valor. Por outro lado, o `R2` que associa-se a variável local `i` é utilizado mas foi renomeado (de `R2` para `i`) no desenho acima para deixar mais claro seu uso.

Verifica-se que há diversas instruções de atribuição que recebem um valor que se parecem

com números, como `'0'` ou `'1'`, mas que estão sendo representados como strings. Na realidade esses valores são todos ponteiros para uma estrutura `Tcl_Obj`, podendo assumir qualquer tipo interno presente na linguagem. Nesse exemplo podemos assumir, e de fato assim ocorrerá, que todos são inteiros. No caso da string `'puts'` armazenada no registrador 4 tem-se que ela servirá como chave em uma tabela hash que possui como valor a respectiva função (ou não, gerando um erro). O número entre parênteses na instrução `CALL` está indicando que haverá 1 parâmetro na chamada e, no caso, este será o conteúdo de `R9`. A única instrução no bloco de saída B3 indica o valor de retorno de função – uma string vazia.

Haviam 55 *bytecodes* (não exibidos aqui) sendo utilizados para representar o código da figura 3, e agora 18 quádruplas são empregadas com o mesmo resultado. Nessa quantidade numérica superior destaca-se o uso de 9 bytes para representar a instrução `INST_START_CMD` da `Tcl`, destinando 4 bytes para contar a quantidade de instruções à frente que fazem parte desse comando e possibilitando ao interpretador respeitar limites impostos para certos recursos através de APIs específicas. Porém, apesar das quádruplas estarem em número inferior, se levarmos em conta o total de bytes contidos em cada quádrupla (de acordo com as estruturas acima) tem-se que o consumo de memória é bastante superior. Ao mesmo tempo é possível observar que muitas delas são passíveis a eliminação, ficando a cargo de otimizações em etapas futuras.

Um último esclarecimento a respeito da figura 4 precisa ser feito. Os *bytecodes* de desvio emitidos pela `Tcl` utilizam operandos que descrevem posições relativas (negativas ou positivas) à posição atual mas pode-se notar que na representação utilizada há somente desvios absolutos para blocos básicos. Para essa tarefa foi feito um mapeamento de cada *bytecode* para cada bloco básico antes de se construir o conteúdo deles assumindo que não haveriam mais quádruplas do que *bytecodes*.

4.1 Mapeamento de alguns *bytecodes* para quádruplas

Nessa subseção é realizado uma análise de como alguns *bytecodes* são convertidos para quádruplas e, em um caso específico, verifica-se um indicio de possível erro durante execução de código que poderia ter sido gerado pelo compilador JIT.

Na figura 3 não foi feito uso de variáveis globais, então começaremos analisando a saída gerada para um trecho que faz uso de tal recurso. O exemplo da figura 5 é bastante simples, porém incorreto na linguagem `Tcl`. É necessário preceder explicitamente a variável de seu escopo, no caso o escopo global é simbolizado por "`::`". Também é possível utilizar o comando `global` para indicar uma lista de variáveis como globais.

```
set x 4
proc y {} { puts $x }
y
```

Figura 5: Uso incorreto de variável global em Tcl

Ao executar tal código o interpretador gera uma exceção, mas, antes de exibir a descrição do problema, a função JIT_Compile é chamada. O resultado, ajustado para melhor visualização, é este:

Tabela 1: Saída produzida para procedimento da figura 5

Bytecode	**Registradores**
PUSH 'puts' LOAD_SCALAR x INVOKESTK 2 DONE	R2 := 'puts' R3 := 0x0 CALL R2 (1) R3

Os registradores 2 e 3 estão sendo atribuídos a endereços de memória estruturados como Tcl_Obj. Por esse motivo, fica claro que não há um endereço previsto para a variável x e, portanto, esse código certamente geraria um erro (assim como o interpretador gera em seguida) se fosse compilado para código de máquina.

Continuando ainda no exemplo da figura 5, detalhamos como o mapeamento de uma coluna para outra foi realizado no restante dessa seção. Começando pela instrução DONE vemos que nada equivalente aparece na coluna a direita. Isso é resultado da atual simplificação feita na outra representação, que, diferentemente da máquina virtual em execução, não verifica por exceções e tão pouco por código de retorno anormal – tais tarefas são realizadas em diversos trechos da função TclExecuteByteCode mas também são especificamente feitas na execução da instrução DONE.

A instrução PUSH é mapeada fazendo uso de uma pilha temporária e também de um array de objetos literais. Um operando que segue a instrução PUSH é tomado como o índice nesse array, sendo possível obter o objeto esperado. O registrado criado nesse momento, R2 no exemplo, é inserido no topo da pilha temporária. A instrução LOAD_SCALAR é mapeada de forma similar, mas ela se baseia em obter o endereço da variável escalar de uma lista de variáveis locais criada pelo compilador da Tcl para o procedimento corrente. Nesse momento temos os registradores 2 e 3 nessa pilha e então a instrução INVOKESTK deve ser convertida. Seguida de um operando, chamado de objc aqui, indicando a quantidade total de argumentos – incluindo o nome da função – temos que os elementos $0 \ldots objc - 2$ da pilha a partir do topo são os parâmetros para a função na posi-

ção $objc - 1$ a partir do topo. Na representação utilizada não contamos o nome da função como um parâmetro, por isso entre parênteses vê-se o número 1 ao invés do 2.

5 Dificuldades Encontradas

A primeira dificuldade encontrada foi em relação a aprendizagem (parcial) do funcionamento da linguagem de programação `Tcl` – especificamente a versão 8.5.8 – que contém mais de 200 mil linhas de código `C`. Boa parte desse número de linhas não afeta diretamente a construção desse compilador, porém uma grande quantidade será ativada ao longo da execução do código gerado pelo compilador.

A linguagem faz uso de contagem de referências para realizar coleta de lixo e também compartilha muitos dos valores utilizados. Com isso, reutilizar objetos `Tcl_Obj` (atualmente no caso da função `JIT_Compile`) que foram construídos em partes distintas não é tão simples pois é necessário se ter certeza de que o objeto não será desalocado enquanto se está trabalhando com ele e, ao mesmo tempo, não se quer deixar objetos com contagem de referência superior a necessária. Compartilhar objetos economiza memória mas não simplifica o uso de objetos alheios, sendo necessário verificar se um objeto específico é compartilhado ou não antes de, dependendo do uso, duplicar o mesmo. Não são tarefas tão complexas mas tendem a ser pontos de erros obscuros em programas não tão curtos e não tão simples.

6 Próximos Passos

Até o momento pouco foi implementado, porém logo em seguida o que se pretende fazer é a coleta de tipos. Somente durante a interpretação os tipos numéricos são descobertos, a representação em *bytecodes* não possui instruções dedicadas para operandos na pilha com tipos diferenciados pois todos são inicialmente do tipo `Tcl_Obj` e, portanto, podem assumir valores de qualquer tipo. Sem essa coleta fica inviável a aplicação de até mesmo da otimização de propagação de constante seguido de empacotamento de constante pois, caso contrário, não sabemos quais os tipos dos valores em uso.

Seguindo a proposta atual é necessário realizar a construção da forma SSA após, ou em conjunto, com o passo anterior. Pretende-se seguir o trabalho de Cytron et al. (1991) aqui, obedecendo os algoritmos lá descritos. Também pretende-se aplicar nesse momento pelo menos algumas das otimizações que se beneficiam com a representação SSA, tais como propagação de constantes e eliminação de código morto. Ainda será avaliado a

possibilidade se realizar alocação de registradores enquanto nessa forma, o trabalho por Wimmer e Franz (2010) discute tal tarefa e será a referência base para essa decisão.

Do restante planeja-se seguir com as etapas previstas anteriormente, refinando-as conforme o projeto avança.

7 Conclusões

Compiladores JIT tem sido implementados em máquinas virtuais que demandam alta performance. Entre as linguagens de programação, `Java` se destaca ao ter recebido suficiente atenção ao ponto de empresas e pesquisadores desenvolverem diversas JVMs com compiladores dinâmicos que fazem uso de uma variedade de técnicas.

Escolhas adequadas para todas as partes de um compilador JIT tornam possível o uso de um compilador otimizador em tempo de execução. Esse texto teve maior foco nas representações intermediárias utilizadas (ou que serão utilizadas) nesse projeto. Quádruplas para formar blocos básicos e permitir a construção do grafo de fluxo de controle (CFG) foi mais discutida aqui, mas também foi mencionado a representação SSA que inicia-se com a entrada de um CFG. A SSA tem sido aplicada em diversos compiladores otimizadores, pois permite aplicar (ao menos) as otimizações de movimentação de código, propagação de constantes e eliminação de redundância parcial de forma eficiente.

A conversão de *bytecodes* de uma máquina de pilha para uma representação na forma de máquina de registradores exige atenção aos detalhes da semântica implementada na máquina virtual atual da linguagem. Uma vantagem visível é a capacidade de sintetização que uma representação baseada em registradores tem sobre uma que faz uso de pilha. Entretanto, um número reduzido de quádruplas não indica necessariamente menor consumo de memória do que uma quantidade superior de *bytecodes*.

8 Referências

ALPERN, B. et al. The jalapeño virtual machine. *IBM Syst. J.*, IBM Corp., Riverton, NJ, USA, v. 39, n. 1, p. 211–238, 2000. ISSN 0018-8670.

BELLARD, F. Qemu, a fast and portable dynamic translator. In: *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2005.

CIERNIAK, M.; LUEH, G.-Y.; STICHNOTH, J. M. Practicing judo: Java under dynamic optimizations. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 35, n. 5, p. 13–26, 2000. ISSN 0362-1340.

CYTRON, R. et al. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, v. 13, n. 4, p. 451 – 490, 1991.

DEUTSCH, L. P.; SCHIFFMAN, A. M. Efficient implementation of the Smalltalk-80 system. In: *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*. Salt Lake City, Utah: [s.n.], 1984. p. 297–302.

ERTL, M. A.; CASEY, K.; GREGG, D. Fast and flexible instruction selection with on-demand tree-parsing automata. In: *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2006. p. 52–60. ISBN 1-59593-320-4.

FREE SOFTWARE FOUNDATION. *Using and porting GNU lightning*. 2007. `http://www.gnu.org/software/lightning/manual/lightning.html`. Acessado em 9 de maio de 2010.

GILMORE, J.; SHEBS, S. *Testsuite – GDB Internals*. 2010. `http://sourceware.org/gdb/current/onlinedocs/gdbint/Testsuite.html`. Acessado em 8 de maio de 2010.

GRABMÜELLER, M.; KLEEBLATT, D. Harpy: run-time code generation in haskell. In: *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell workshop*. New York, NY, USA: ACM, 2007. ISBN 978-1-59593-674-5.

HERNÁNDEZ, R. Openacs: robust web development framework. In: *12th Annual Tck/Tk Conference*. Portland, Oregon, USA: [s.n.], 2005.

HÖLZLE, U. *Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming*. Tese (Doutorado), Stanford, CA, USA, 1994.

KOES, D. R.; GOLDSTEIN, S. C. Near-optimal instruction selection on dags. In: *CGO '08: Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. New York, NY, USA: ACM, 2008. p. 45–54. ISBN 978-1-59593-978-4.

KRALL, A.; GRAFL, R. Cacao - a 64 bit javavm just-in-time compiler. In: . [S.l.]: ACM, 1997. p. 1017–1030.

LEWIS, B. T. An on-the-fly bytecode compiler for tcl. In: *TCLTK'96: Proceedings of the 4th conference on USENIX Tcl/Tk Workshop, 1996*. Berkeley, CA, USA: USENIX

Association, 1996.

OUSTERHOUT, J. K. *Tcl: An Embeddable Command Language*. [S.l.], Nov 1989. Disponível em: <http://www.eecs.berkeley.edu/Pubs/TechRpts/1989/5742.html>.

POLETTO, M.; SARKAR, V. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, ACM, New York, NY, USA, v. 21, n. 5, p. 895–913, 1999. ISSN 0164-0925.

RIGO, A. 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*. New York, NY, USA: ACM, 2004. p. 15–26. ISBN 1-58113-835-0.

RIGO, A.; PEDRONI, S. Pypy's approach to virtual machine construction. In: *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. New York, NY, USA: ACM, 2006. p. 944–953. ISBN 1-59593-491-X.

SAH, A. *TC: An Efficient Implementation of the Tcl Language*. [S.l.], Apr 1994. Disponível em: <http://www.eecs.berkeley.edu/Pubs/TechRpts/1994/5189.html>.

SUGANUMA, T. et al. Evolution of a java just-in-time compiler for ia-32 platforms. *IBM J. Res. Dev.*, IBM Corp., Riverton, NJ, USA, v. 48, n. 5/6, p. 767–795, 2004. ISSN 0018-8646.

SUGANUMA, T.; YASUE, T.; NAKATANI, T. A region-based compilation technique for a java just-in-time compiler. In: *In Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. [S.l.]: ACM Press, 2003. p. 312–323.

VITALE, B.; ABDELRAHMAN, T. S. Catenation and specialization for tcl virtual machine performance. In: *In IVME '04 Proceedings*. [S.l.]: ACM Press, 2004. p. 42–50.

WELCH, B. Customization and flexibility in the exmh mail user interface. In: *TCLTK '98: Proceedings of the 3rd Annual USENIX Workshop on Tcl/Tk*. Berkeley, CA, USA: USENIX Association, 1995.

WIMMER, C.; FRANZ, M. Linear scan register allocation on ssa form. In: *CGO '10: Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. New York, NY, USA: ACM, 2010. p. 170–179.