

Trabalho Prático 3 - Tradução de código intermediário do front-end de SmallL para TAM

Hugo Araujo de Sousa
(2013007463)

Compiladores I (2017/2)
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG)

`hugosousa@dcc.ufmg.br`

***Resumo.** Este trabalho tem como objetivo o desenvolvimento de um tradutor da representação intermediária da linguagem SmallL, como definido no Trabalho Prático II da disciplina Compiladores I, para a máquina virtual TAM.*

1. INTRODUÇÃO

No Trabalho Prático II (TPII), foi desenvolvido um front-end completo para a linguagem SmallL. Dessa forma, todas as fases de análise léxica, sintática, semântica e geração de código intermediário foram completadas. O próximo passo, então, no processo de compilação de um programa escrito em SmallL é o de tradução do código intermediário gerado pelo front-end criado no TPII para alguma máquina alvo.

Como a etapa de tradução é dependente da máquina alvo, o primeiro passo é determinar para qual máquina o programa em representação intermediária deverá ser traduzido. Para esse trabalho, a máquina escolhida foi a máquina virtual TAM. Para informações sobre a máquina TAM, seu conjunto de instruções e registradores, ver [Watt et al. 2007].

2. METODOLOGIA

A primeira etapa no desenvolvimento do trabalho foi a de realizar uma pequena alteração do código do TPII, uma vez que o código intermediário gerado por esse não possui qualquer anotação sobre os tipos das variáveis. Sem essa informação, não é possível alocar espaço em memória para cada variável.

Logo, o TPII modificado imprime no cabeçalho do código intermediário gerado anotações de tipo das variáveis e o tradutor realiza a alocação de memória para as mesmas.

Em seguida, foi necessário identificar todos os tipos de quádruplas presentes no código intermediário. Esses tipos são apresentados a seguir:

1. **Salto condicional:** Desvios de código baseados em uma condição booleana. São representados por instruções "if" e "iffalse".
2. **Salto incondicional:** Desvios de código independentes de qualquer condição booleana. São representados por instruções do tipo "goto".
3. **Indexação de array como l-value:** Uma variável ou valor é copiado para um elemento de array.
4. **Indexação de array como r-value:** Um elemento de array é copiado para uma variável.

5. **Atribuição simples:** Valor número ou de uma variável é atribuído a outra variável.
6. **Atribuição aritmética:** Resultado de uma expressão aritmética é atribuída a uma variável.
7. **Atribuição unária:** Resultado de uma operação unária é atribuída a uma variável.

Com posse desses tipos, o processo de tradução se reduz a identificar quais são as instruções TAM necessárias para executar cada uma das quádruplas presentes no código intermediário. A metodologia usada usou como referência [Aho et al. 1986].

De maneira geral, a estrutura do algoritmo é a de ler todas as quádruplas, efetuar a tradução de cada uma e, por fim, efetuar o processo de Backpatching para obter os endereços das labels de desvio ainda não definidas.

3. CÓDIGO FONTE

O código fonte pode ser encontrado na pasta **src** no diretório do projeto no GitHub¹. A pasta **tp2-modified** contém o código fonte da versão modificada do TPII, como dito na Seção 2.

O tradutor foi implementado utilizando a linguagem Python 3 e o código foi separado em dois arquivos:

1. **quadruple.py:** Define a classe **Quadruple** que representa uma quádrupla.
2. **translator.py:** Implementa o tradutor propriamente dito.

4. EXECUÇÃO

Para executar o tradutor, primeiramente é necessário fornecer como entrada um arquivo em código intermediário de SmallL gerado através da versão modificada do TPII. Da raiz do diretório do projeto, fazemos:

```
cd tp2-modified/src
make
java main.Main < arquivo_entrada
```

Onde **arquivo_entrada** é o arquivo em SmallL para o qual deseja-se gerar código intermediário. Após a execução dos comandos, será gerado um arquivo **code.out** contendo as quádruplas correspondentes ao código de entrada.

Com o código intermediário em mãos, podemos executar o tradutor. Para isso, da raiz do diretório do projeto, fazemos:

```
cd src
./translator.py arquivo_entrada arquivo_saida
```

Onde **arquivo_entrada** é o arquivo em código intermediário gerado no passo anterior e **arquivo_saida** é o arquivo onde o resultado da tradução, isto é, o código TAM correspondente ao código intermediário de entrada deve ser salvo. O tradutor também lista o arquivo de entrada na saída padrão.

¹<https://github.com/ha2398/compiladores1-tps/tree/master/tp3>

5. TESTES E RESULTADOS

Para ilustrar o funcionamento do tradutor, são apresentados dois exemplos de tradução, onde são mostrados primeiramente o código intermediário de entrada e o código TAM de saída.

5.1. code1.out

- Código intermediário:

```
1      int i
2      int j
3      float v
4      float x
5      [100] float a
6
7  L1:L3:      i = i + 1
8              t1 = i * 8
9              t2 = a [ t1 ]
10             if t2 < v goto L3
11  L4:        j = j - 1
12             t3 = j * 8
13             t4 = a [ t3 ]
14             if t4 > v goto L4
15  L6:        iffalse i >= j goto L8
16  L9:        goto L2
17  L8:        t5 = i * 8
18             x = a [ t5 ]
19  L10:       t6 = i * 8
20             t7 = j * 8
21             t8 = a [ t7 ]
22             a [ t6 ] = t8
23  L11:       t9 = j * 8
24             a [ t9 ] = x
25             goto L1
26  L2:
```

- Código TAM:

1	10	0	0	2	; PUSH 2
2	10	0	0	2	; PUSH 2
3	10	0	0	4	; PUSH 4
4	10	0	0	4	; PUSH 4
5	10	0	0	400	; PUSH 400
6	10	0	0	4	; PUSH 4
7	10	0	0	4	; PUSH 4
8	10	0	0	4	; PUSH 4
9	10	0	0	4	; PUSH 4
10	10	0	0	4	; PUSH 4
11	10	0	0	4	; PUSH 4
12	10	0	0	4	; PUSH 4
13	10	0	0	4	; PUSH 4
14	10	0	0	4	; PUSH 4
15	1	4	0	0	; LOADA 0[SB]
16	2	0	2	0	; LOADI (2)
17	3	0	0	1	; LOADL 1

18	6	2	0	8	; add
19	1	4	0	0	; LOADA 0[SB]
20	5	0	2	0	; STOREI(2)
21	1	4	0	0	; LOADA 0[SB]
22	2	0	2	0	; LOADI(2)
23	3	0	0	8	; LOADL 8
24	6	2	0	10	; mult
25	1	4	0	412	; LOADA 412[SB]
26	5	0	4	0	; STOREI(4)
27	1	4	0	412	; LOADA 412[SB]
28	2	0	2	0	; LOADI(4)
29	1	4	0	12	; LOADA 12[SB]
30	6	2	0	8	; add
31	2	0	4	0	; LOADI(4)
32	1	4	0	416	; LOADA 416[SB]
33	5	0	4	0	; STOREI(4)
34	1	4	0	416	; LOADA 416[SB]
35	2	0	4	0	; LOADI(4)
36	1	4	0	4	; LOADA 4[SB]
37	2	0	4	0	; LOADI(4)
38	6	2	0	13	; lt
39	14	0	1	14	; JUMPIF(1) 14[CB]
40	1	4	0	2	; LOADA 2[SB]
41	2	0	2	0	; LOADI(2)
42	3	0	0	1	; LOADL 1
43	6	2	0	9	; sub
44	1	4	0	2	; LOADA 2[SB]
45	5	0	2	0	; STOREI(2)
46	1	4	0	2	; LOADA 2[SB]
47	2	0	2	0	; LOADI(2)
48	3	0	0	8	; LOADL 8
49	6	2	0	10	; mult
50	1	4	0	420	; LOADA 420[SB]
51	5	0	4	0	; STOREI(4)
52	1	4	0	420	; LOADA 420[SB]
53	2	0	2	0	; LOADI(4)
54	1	4	0	12	; LOADA 12[SB]
55	6	2	0	8	; add
56	2	0	4	0	; LOADI(4)
57	1	4	0	424	; LOADA 424[SB]
58	5	0	4	0	; STOREI(4)
59	1	4	0	424	; LOADA 424[SB]
60	2	0	4	0	; LOADI(4)
61	1	4	0	4	; LOADA 4[SB]
62	2	0	4	0	; LOADI(4)
63	6	2	0	16	; gt
64	14	0	1	39	; JUMPIF(1) 39[CB]
65	1	4	0	0	; LOADA 0[SB]
66	2	0	2	0	; LOADI(2)
67	1	4	0	2	; LOADA 2[SB]
68	2	0	2	0	; LOADI(2)
69	6	2	0	15	; ge
70	14	0	0	71	; JUMPIF(0) 71[CB]
71	12	0	0	125	; JUMP 125[CB]
72	1	4	0	0	; LOADA 0[SB]

73	2	0	2	0	; LOADI (2)
74	3	0	0	8	; LOADL 8
75	6	2	0	10	; mult
76	1	4	0	428	; LOADA 428[SB]
77	5	0	4	0	; STOREI (4)
78	1	4	0	428	; LOADA 428[SB]
79	2	0	2	0	; LOADI (4)
80	1	4	0	12	; LOADA 12[SB]
81	6	2	0	8	; add
82	2	0	4	0	; LOADI (4)
83	1	4	0	8	; LOADA 8[SB]
84	5	0	4	0	; STOREI (4)
85	1	4	0	0	; LOADA 0[SB]
86	2	0	2	0	; LOADI (2)
87	3	0	0	8	; LOADL 8
88	6	2	0	10	; mult
89	1	4	0	432	; LOADA 432[SB]
90	5	0	4	0	; STOREI (4)
91	1	4	0	2	; LOADA 2[SB]
92	2	0	2	0	; LOADI (2)
93	3	0	0	8	; LOADL 8
94	6	2	0	10	; mult
95	1	4	0	436	; LOADA 436[SB]
96	5	0	4	0	; STOREI (4)
97	1	4	0	436	; LOADA 436[SB]
98	2	0	2	0	; LOADI (4)
99	1	4	0	12	; LOADA 12[SB]
100	6	2	0	8	; add
101	2	0	4	0	; LOADI (4)
102	1	4	0	440	; LOADA 440[SB]
103	5	0	4	0	; STOREI (4)
104	1	4	0	440	; LOADA 440[SB]
105	2	0	4	0	; LOADI (4)
106	1	4	0	432	; LOADA 432[SB]
107	2	0	4	0	; LOADI (4)
108	1	4	0	12	; LOADA 12[SB]
109	6	2	0	8	; add
110	5	0	4	0	; STOREI (4)
111	1	4	0	2	; LOADA 2[SB]
112	2	0	2	0	; LOADI (2)
113	3	0	0	8	; LOADL 8
114	6	2	0	10	; mult
115	1	4	0	444	; LOADA 444[SB]
116	5	0	4	0	; STOREI (4)
117	1	4	0	8	; LOADA 8[SB]
118	2	0	4	0	; LOADI (4)
119	1	4	0	444	; LOADA 444[SB]
120	2	0	4	0	; LOADI (4)
121	1	4	0	12	; LOADA 12[SB]
122	6	2	0	8	; add
123	5	0	4	0	; STOREI (4)
124	12	0	0	14	; JUMP 14[CB]
125	15	0	0	0	; HALT

5.2. code3.out

- Código intermediário:

```
1      int n
2      int u
3      int result
4      int i
5      int t
6
7  L1:      n = 10
8  L3:      u = 0
9  L4:      result = 1
10 L5:      i = 2
11 L6:      iffalse i <= n goto L2
12 L7:      t = u + result
13 L8:      u = result
14 L9:      result = t
15 L10:     i = i + 1
16      goto L6
17 L2:
```

- Código TAM:

1	10	0	0	2	; PUSH 2
2	10	0	0	2	; PUSH 2
3	10	0	0	2	; PUSH 2
4	10	0	0	2	; PUSH 2
5	10	0	0	2	; PUSH 2
6	3	0	0	10	; LOADL 10
7	1	4	0	0	; LOADA 0[SB]
8	5	0	2	0	; STOREI(2)
9	3	0	0	0	; LOADL 0
10	1	4	0	2	; LOADA 2[SB]
11	5	0	2	0	; STOREI(2)
12	3	0	0	1	; LOADL 1
13	1	4	0	4	; LOADA 4[SB]
14	5	0	2	0	; STOREI(2)
15	3	0	0	2	; LOADL 2
16	1	4	0	6	; LOADA 6[SB]
17	5	0	2	0	; STOREI(2)
18	1	4	0	6	; LOADA 6[SB]
19	2	0	2	0	; LOADI(2)
20	1	4	0	0	; LOADA 0[SB]
21	2	0	2	0	; LOADI(2)
22	6	2	0	14	; le
23	14	0	0	46	; JUMPIF(0) 46[CB]
24	1	4	0	2	; LOADA 2[SB]
25	2	0	2	0	; LOADI(2)
26	1	4	0	4	; LOADA 4[SB]
27	2	0	2	0	; LOADI(2)
28	6	2	0	8	; add
29	1	4	0	8	; LOADA 8[SB]
30	5	0	2	0	; STOREI(2)
31	1	4	0	4	; LOADA 4[SB]

32	2	0	2	0	; LOADI (2)
33	1	4	0	2	; LOADA 2[SB]
34	5	0	2	0	; STOREI (2)
35	1	4	0	8	; LOADA 8[SB]
36	2	0	2	0	; LOADI (2)
37	1	4	0	4	; LOADA 4[SB]
38	5	0	2	0	; STOREI (2)
39	1	4	0	6	; LOADA 6[SB]
40	2	0	2	0	; LOADI (2)
41	3	0	0	1	; LOADL 1
42	6	2	0	8	; add
43	1	4	0	6	; LOADA 6[SB]
44	5	0	2	0	; STOREI (2)
45	12	0	0	17	; JUMP 17[CB]
46	15	0	0	0	; HALT

6. CONCLUSÃO

O trabalho em questão faz paralelo com um trabalho realizado para a disciplina Software Básico, onde um tradutor também deveria ser implementado. Dessa forma, uma certa familiaridade com o tema já existia. Entretanto, para esse trabalho, o tópico foi mais aprofundado, levantando muitas questões que antes não foram abordadas, como organização de memória, verificação de tipos, etc.

É importante ressaltar que a maior dificuldade foi a de estudar a fundo a máquina alvo TAM. Não havia nenhum conhecimento sobre a mesma e há pouca documentação online.

Alguns pontos levantaram muita dúvida durante a implementação, por exemplo, no manual da máquina vemos que somente há suporte para operações aritméticas com valores inteiros, o que é uma grande falha na especificação, uma vez que a SmallL permite o uso de variáveis de ponto flutuante.

De maneira geral, o trabalho pode ser concluído com sucesso e os testes se mostraram correspondentes ao que era desejado.

7. REFERÊNCIAS

- Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers principles, techniques, and tools*. Addison-Wesley, Reading, MA.
- Watt, D., Brown, D., and Sebesta, R. W. (2007). *Programming Language Processors in Java: Compilers and Interpreters AND Concepts of Programming Languages*. Prentice Hall Press, Upper Saddle River, NJ, USA.