VII – Aplicação

7.1. Introdução

Nos capítulos anteriores, atingimos os objetivos de criar um modelo categórico de tradução entre linguagens de programação e, de a partir deste, desenvolver um critério formal que uma tradução deve atender para preservar a manutenibilidade. Mostramos que a motivação para este trabalho veio da aplicação prática de idéias e ferramentas que surgiram no meio acadêmico.

Este capítulo é o ponto de partida de mais um período deste ciclo academia-indústria-academia; nele buscamos aproximar as idéias teóricas desenvolvidas nesta tese da prática. Acreditamos que as aplicações práticas possam seguir diversas direções, mencionadas na seção 8.1.

Neste capítulo ficamos restritos apenas à “Analise de ferramentas de traduções existentes”. Mesmo dentro deste tópico, esse capítulo não é ambicioso, pois se limita a uma análise qualitativa simplificada, através de exemplos, cujo principal objetivo é mostrar como a idéia de “preservação de semântica de árvores” indica melhorias imediatas nas traduções implementadas por estas ferramentas.

7.2. Os tradutores estudados

Os tradutores escolhidos para estudo neste capítulo são os seguintes:

- P2C [Lauer & Wallwitz, 1989]. Este tradutor de Pascal para C foi bastante popular no início da década de 1990. No próprio Departamento de Informática da PUC houve uma publicação a respeito de sua aplicação [Guedes e Staa, 1993].
- F2C. Trata-se de um tradutor desenvolvido conjuntamente por pessoas da Bell Labs, Bellcore e Universidade Carnegie Mellon; seu código-fonte está disponível há mais de

10 anos na Internet e tem sido exaustivamente testado e melhorado desde então. Alguns artigos descrevem a utilização do tradutor [Angus e Stolzy, 1991] [Levy, 1995].

- ForC [FOR_C, 1999]. Trata-se de um tradutor comercial de Fortran para C produzido pela Cobalt Blue, Inc. Uma versão de demonstração e seu manual podem ser obtidos em www.cobalt-blue.com. É um produto de sucesso comercial indiscutível; a página da empresa na Internet lista mais de cem grandes clientes no mundo inteiro, entre empresas, governo e universidades.

7.3. Conclusões

Nesta seção serão apresentadas as conclusões obtidas durante a análise dos tradutores. Posteriormente serão apresentados os exemplos (traduções de trechos de programas) que justifiquem essas conclusões.

Ao iniciarmos esta análise tínhamos plena ciência de que os produtos analisados foram implementados por profissionais de alta qualificação e que passaram com sucesso pelos difíceis testes do mercado. Se as idéias desta tese pudessem ajudar a aprimorar o projeto de tais tradutores, com certeza seriam ainda mais úteis para profissionais que não possuam uma grande especialização na área.

Nos exemplos estudados neste capítulo, encontramos casos onde a preservação da semântica de árvores aponta uma alternativa melhor para a tradução implementada, provando assim a utilidade do conceito. Para nossa surpresa, em um grande número de casos, a preocupação com a manutenibilidade do código gerado levou os projetistas dos tradutores a soluções que parecem terem sido explicitamente projetadas com o objetivo de preservar a semântica de árvores. Em nossa opinião, esta situação valida o conceito, pois mostra que esses projetistas possuem um conceito intuitivo semelhante ao conceito

formalizado nesta tese. As principais constatações produzidas pela análise dos tradutores foram:

1. A experiência dos projetistas fez com que eles seguissem intuitivamente, em alguns casos, o critério de preservação da semântica de árvores no desenho da tradução. Os exemplos mostrados na próxima seção concentram-se nas construções que fazem parte das linguagens Pascal e Fortran; no entanto, a tradução das bibliotecas de fabricantes específicos encontra-se repleta de situações que também reforçam esta constatação.
2. Em diversas situações, o critério de preservação da semântica de árvores indica uma clara melhoria no projeto da tradução, mostrando que o conceito pode beneficiar até mesmo projetistas de alta qualificação, como os que implementaram as ferramentas estudadas.
3. Os projetistas das ferramentas de tradução dão ênfase à manutenibilidade dos programas gerados, até mesmo sacrificando a preservação da semântica em favor da manutenibilidade. Em alguns casos, a preservação da semântica tornaria o código gerado extremamente ineficiente, ou de difícil manutenção. Nestes casos, os projetistas da tradução preferem fazer traduções “aproximadas”, onde a semântica seria igual se certas condições fossem verdadeiras. Por exemplo, a tradução assume que a semântica dos tipos primitivos **integer** em Pascal e **int** em C são a mesma, a tradução da função **maxavail**() em Pascal, que retorna o maior bloco contíguo de memória livre não reproduz exatamente o seu comportamento, etc. Podemos listar dezenas de casos onde a tradução não preserva “estritamente” a semântica. O tradutor P2C, preocupando-se com a facilidade de uso, gera mensagens de aviso cada vez que uma tradução pode potencialmente alterar a semântica do programa (ou melhor, cada vez que os projetistas sabiam que estavam violando a semântica do programa original, muitas vezes não sabiam).

4. Em muitos casos a violação da semântica não traz nenhum beneficio em termos de eficiência ou manutenibilidade. Estas violações parecem não ter sido percebidas pelos implementadores das ferramentas, pois não há nenhuma menção a estas restrições na documentação. Isso mostra a grande dificuldade de avaliação informal da corretude de uma tradução, sendo assim um argumento a favor do uso de métodos formais durante o desenvolvimento de um tradutor, como por exemplo, prova de corretude das transformações a partir da especificação semântica da linguagem.
5. Encontramos diversos exemplos de soluções criativas, o que confirma a qualidade dos produtos.
6. Como era de se esperar, ficou claro que é muito mais fácil traduzir de Pascal para C do que de Fortran para C, principalmente a parte de definição de variáveis e a parte de Entrada/Saída, que é muito particular em Fortran. A dificuldade de preservação de semântica de árvores explica a degradação da manutenibilidade do código traduzido.
7. Em geral, pode-se dizer que os tradutores estudados devem o seu sucesso a uma abordagem prática ao problema da tradução. Isso significa que os sistemas são produtos bem acabados e fáceis de usar, que estão muito bem adaptados a extensões proprietárias da linguagem. O f2c é considerado como uma alternativa para compiladores Fortran 77 em diversos ambientes Unix. O ForC suporta as extensões do CDC FORTRAN-5, Data General (AOS) FORTRAN-77, HPUX-Precision, Architecture FORTRAN-77, VAX FORTRAN, IBM VS FORTRAN (FORTRAN-77), Microsoft FORTRAN v5.1, PDP FORTRAN-77, Prime FORTRAN-77, SGI FORTRAN-77, Sun FORTRAN-77, além de permitir a configuração por parte do usuário de “macros” para alguma versão não suportada da linguagem. No caso do P2C[1]

[1] Foi analisada a versão do P2C para Turbo Pascal 3/4/5.

observamos o mesmo em relação a um fabricante particular (Borland), o compilador dá suporte a todas as extensas bibliotecas não padrão desta versão.

7.4. Exemplos

Nesta seção apresentamos traduções de trechos de código que justificam as conclusões apresentadas na seção anterior.

Exemplo 7.1. Tradução do comando **with**.

Linguagem de entrada: Pascal

Linguagem de saída: C

Tradutor: P2C

Conclusões: 2, 3.

Comentários: O exemplo mostrado na figura 7.1 não preserva a semântica. Como observado anteriormente no capítulo 5, esta tradução preserva a semântica sob a condição de que o acesso à variável do comando **with** não produza efeito colateral. Nesse caso, o tradutor gera uma advertência para o usuário. Mesmo no caso em que haja preservação da semântica, não haverá preservação da semântica de árvores. A alternativa proposta no capítulo 5 é superior em todos os aspectos, pois sempre preserva a semântica e gera um código mais manutenível e mais eficiente.

PASCAL	C
...	...
WITH DISC[I].PROF DO	
BEGIN	
HORA := HORA + 4;	DISC[I].PROF.HORA = DISC[I].PROF.HORA + 4;
...	...
AULAGRAD := TRUE;	DISC[I].PROF. AULAGRAD = true;
END	
...	

Figura 7.1 Tradução do comando **with** no P2C.

Exemplo 7.2. Tradução de procedimentos e funções aninhadas.

Linguagem de entrada: Pascal

Linguagem de saída: C

Tradutor: P2C

Conclusões: 4, 5.

Comentários: O tradutor em questão oferece uma solução interessante para o problema de procedimentos (ou funções) aninhados em Pascal. É criada uma referência global para as variáveis de um procedimento exterior que são acessadas em procedimentos internos. No início da execução do procedimento externo, a referência global é apontada para o endereço da variável em questão na pilha. Esse esquema funciona satisfatoriamente com os exemplos de recursividade no manual do tradutor. No entanto, um detalhe passou despercebido dos projetistas: a necessidade de se restaurar a referência global ao final da ativação do procedimento externo; caso contrário, ao retornarmos para a ativação anterior e chamarmos o procedimento interno, a referência estará apontando para uma posição da pilha que já foi liberada. A figura 7.2 ilustra esse caso.

PASCAL	C
`function fat(n: integer):integer;`	`static SHORT *l_n;`
`  Function ene:integer;`	`static SHORT ene(VOID)`
`  Begin`	`{`
	`  SHORT r_ene;`
`  Ene := n;`	`  r_ene=*l_n;`
	`  return r_ene;`
`  end;`	`}`
`Begin`	`SHORT fat(SHORT n) {`
	`  SHORT r_fat;`
	`  L_n=&n;`
`if n=0 then`	`if (n==0)`
`  fat := 1`	`  r_fat=1;`
`Else`	`else`
`  fat:=fat(n-1)*ene;`	`  r_fat=fat(n-1)*ene();`
	`return r_fat;`
`end;`	`}`

Figura 7.2 Tradução de funções aninhadas no P2C[2].

[2] A ordem das funções no código C foi invertida para facilitar a comparação com o código Pascal.

Exemplo 7.3. Tradução de variáveis do tipo **string** e atualização de suas posições.

Linguagem de entrada: Pascal

Linguagem de saída: C

Tradutor: P2C

Conclusões: 3, 4, 7.

Comentários: O P2C traduz o tipo **string** de Pascal[3] para vetor do tipo **char** em C. O tradutor implementa as rotinas de manipulação de **string** de Pascal, ajusta o índice, e converte atribuições à posição zero de uma string em Pascal a um comando semanticamente equivalente em C. No entanto, como o problema de saber qual posição está sendo atualizada é não-computável, a tradução exibida na figura 7.3 não preserva a semântica em todos os casos. A opção de testar a posição atualizada levaria a um código correto, porém, ineficiente. Acreditamos que os projetistas do tradutor optaram por esta alternativa porque o código gerado possui um estilo apropriado para a linguagem C e é de fácil leitura. É improvável que os projetistas não tivessem consciência da não preservação da semântica neste caso; ainda assim, o tradutor não gera uma mensagem de advertência, o que pode tornar extremamente difícil a localização de um eventual erro no programa traduzido.

PASCAL	C
`...`	`...`
`readln(i);`	`scanf("%d\n",&i);`
`s:='ALO VOCE';`	`strasg(s,"ALO VOCE",sizeof(STRING);`
`s[i] := ' ';`	`s[i-1]=' ';`
`s[2*i] := 8;`	`s[2*i-1]=8;`
`writeln(s);`	`printf("%s\n",s);`
`...`	`...`

Figura 7.3 Tradução de **strings** no P2C.

[3] **Strings** são uma extensão ao Pascal introduzida pelo fabricante (Borland).

Exemplo 7.4. Tradução de constantes.

Linguagem de entrada: Pascal

Linguagem de saída: C

Tradutor: P2C

Conclusões: 3, 4.

Comentários: Constantes em Pascal são traduzidas para **#define** em C. O tradutor renomeia as constantes com o mesmo nome para evitar problemas de escopo. No entanto, o tradutor viola a semântica ao mover a declaração das constantes globais para um único arquivo ".h" no início do programa, como mostra a figura 7.4.

PASCAL	C
...	...
	#include "exemplo5.h"
Procedure p1;	VOID p1(VOID)
Const n = 3;	{
Begin	#define N 3
Writeln(n);	Printf("%d\n",N);
end;	#undef N
	}
Const n = 2;	
Procedure p2;	VOID p2(VOID)
Const n = 3;	{
Begin	#define N_1 3
Writeln(n);	printf("%d\n",N_1);
end;	#undef N_1
	}
Begin	SHORT main(VOID) {
	P2c_init(NULL,2048,0);
writeln(n);	Printf("%d\n",N);
end.	Return 0;
	}
...	...

Figura 7.4 Tradução de constantes no P2C.

Exemplo 7.5. Tradução de variáveis do tipo conjunto e operações sobre as mesmas.

Linguagem de entrada: Pascal

Linguagem de saída: C

Tradutor: P2C

Conclusões: 1.

Comentários: A tradução do tipo conjunto de Pascal mostra a preocupação intuitiva dos projetistas da tradução com a preservação da semântica de árvores. As diversas operações sobre conjuntos definidas em Pascal são implementadas em uma biblioteca, **p2cset**. Assim, a atribuição é traduzida para **setasg**, o operador **in** para a função **member**, e as operações sobre conjunto para a função **setop**, onde o primeiro parâmetro é a operação (“+”, “-”, “*”). A parte mais difícil é a tradução dos construtores do tipo conjunto. Devido à possibilidade de se criarem conjuntos tanto a partir de faixas de valores enumerados como de valores enumerados isolados, a sua tradução, **setgen**, possui um primeiro argumento que identifica se um dado parâmetro para a função é um valor isolado ou o limite de uma faixa. Por exemplo, o conjunto `[‘a’..‘h’,‘z’]` é traduzido para `setgen(“21” ,‘a’,‘h’,‘z’`. Nesse caso, a tradução é sensível ao contexto, pois o primeiro parâmetro depende dos demais. Porém, uma vez fixada a assinatura da função; a semântica de árvores é preservada. Dessa forma, toda a tradução de conjuntos procura preservar a estrutura sintática do programa original, seguindo a idéia de preservação de semântica de árvores. É interessante notar que a extensão do pré-processador de C proposta em [Garcia e Guedes, 1999] permitiria, neste caso, uma tradução livre de contexto de uma faixa de valores. A figura 7.5 mostra a tradução de um trecho de um analisador sintático, onde são inicializados os conjuntos de iniciadores usando-se o P2C.

PASCAL	C
...	...
procedure initialize;	VOID initialize(VOID)
Begin	{
FdefinitionPart := [const1, integer1, boolean1, proc1];	setasg(fdefinitionpart,setgen("1111", CONST1, INTEGER1, BOOLEAN1, PROC1));
FStatment := [skip1, read1, name1, write1, call1, if1, do1];	setasg(fstatment,setgen("1111111", SKIP1, READ1, NAME1, WRITE1, CALL1, IF1, DO1));
FstatmentPart := Fstatment;	setasg(fstatmentpart,fstatment);
FConstant := [numeral1, true1, false1, name1, integer1, boolean1];	setasg(fconstant,setgen("111111",NUMERAL1, TRUE1, FALSE1, NAME1, INTEGER1, BOOLEAN1));
FFactor := FConstant + [abrep1, nao1];	setasg(ffactor,setop("+",fconstant,setgen("11",ABREP1,NAO1)));
Fterm := FFactor;	setasg(fterm,ffactor);
FsimpleExpression := FFactor + [menos1];	setasg(fsimpleexpression,setop("+", ffactor, setgen("1",MENOS1)));
FprimaryExpression := FsimpleExpression;	setasg(fprimaryexpression, fsimpleexpression);
Fexpression := FprimaryExpression;	setasg(fexpression, fprimaryexpression);
FexpressionList := FExpression;	setasg(fexpressionlist, fexpression);
FguardedCommand := FExpression;	setasg(fguardedcommand, fexpression);
end;	}
...	...

Figura 7.5 Tradução de conjuntos no P2C.

Exemplo 7.6. Tradução de operações sobre o tipo **string**.

Linguagem de entrada: Pascal

Linguagem de saída: C

Tradutor: P2C

Conclusões: 1.

Comentários: A tradução de operações sobre o tipo **string** de Pascal segue a mesma linha da tradução do tipo conjunto. As operações sobre **strings** são implementadas em uma biblioteca, **p2cstr**, e são traduzidas por funções cujos nomes começam com "STR". Semelhantemente ao construtor de conjuntos, a soma de **strings** possui um primeiro parâmetro que é o número de **strings** somadas. A figura 7.6 mostra um exemplo de tradução que consta do manual do tradutor. Apesar do claro esforço dos tradutores em preservar a mesma estrutura do código Pascal, há pequenos detalhes que não preservam a

semântica de árvores; como a necessidade de se colocar **sizeof(…)** ao final da atribuição de **strings**; nesse caso, o tipo das variáveis **s1**, **s2** e **s3** que estava parametrizado em um único ponto no programa original, encontra-se espalhado em diversos pontos do programa gerado, o que pode prejudicar uma futura manutenção. É claro que a solução do tradutor é melhor do que resolver o tamanho do tipo em tempo de tradução.

PASCAL	C
...	...
CONST c0 = ‘a’;	#define C0 ‘a’
c1 = ‘bcd’;	#define C1 “bcd”
c2 = ‘efg’;	#define C2 “efg”
TYPE TSTR = String[80];	typedef CHAR TSTR[81];
VAR s1, s2, s3: TSTR;	TSTR s1, s2, s3;
s4: String[5];	typedef CHAR T_S4[6];
	T_S4 s4;
i,j:Integer;	SHORT i,j;
BEGIN	SHORT main(VOID) {
	P2c_init(NULL,2048,0);
s1:= c0+c1+c2;	strasg(s1,strsum(3,ctos(C0),C1,C2), sizeof(TSTR));
If (s1<’A’) OR (s1>=c1) then	If((strcmp(s1,”A”)<0) || (strcmp(s1,C1)>=0))
Writeln(‘Error 1’);	Fputs(“Error 1\n”, stdout);
s4 := s1;	strasg(s4,s1,sizeof(T_S4));
s2 := s1;	strasg(s2,s1,sizeof(T_S4));
Delete(s2,Pos(c1,s1),Length(c1));	strdel(s2,strpos(s1,C1)+1-1, strlen(C1));
s3:=’’;	strasg(s3,””,sizeof(TSTR));
Insert(c2,s3,7);	strins(s3,C2,6,sizeof(s3));
Insert(c0,s3,1);	strins(s3,ctos(C0),0,sizeof(s3));
i:=length(s3);	i=strlen(s3);
Str(i:1,s3);	sprintf(s3,”%1d”,i);
Val(s3,i,j);	j= val(s3,”%d”,&i);
	return(0);
END.	}
...	...

Figura 7.6 Tradução de **strings** no P2C.

Exemplo 7.7. Tradução de funções-comando.

Linguagem de entrada: Fortran.

Linguagem de saída: C.

Tradutores: F2C e ForC.

Conclusões: 1 (ForC), 2 (F2C).

Comentários: As chamadas funções-comando em Fortran são pequenas funções escritas em uma única linha. As figura 7.7 e 7.8 mostram as traduções geradas pelo F2C e pelo ForC, respectivamente. Nesse caso, está absolutamente claro que a solução adotada pelo F2C, a saber, a expansão inline das funções-comando, não preserva a semântica de árvores do programa. A solução do tradutor ForC – traduzir as funções-comando para macros – preserva a semântica de árvores.

Fortran	C
...	...
LINHA(K) = ((K-1)/3)+1	
COL(K) = K-((K-1)/3)*3	
...	
IAUXL = LINHA(L)	iauxl = (*l –1)/3 +1;
IAUXC = COL(L)	iauxc = (*l – (*l –1) / 3 * 3);
...	...

Figura 7.7 Tradução de funções-comando pelo F2C.

Fortran	C
...	...
LINHA(K) = ((K-1)/3)+1	#define LINHA(k) ((long)((((k)-1)/3) + 1))
COL(K) = K-((K-1)/3)*3	#define COL(k) ((long)((k)-(((k)1)/3)*3))
...	
IAUXL = LINHA(L)	iauxl = LINHA(*l);
IAUXC = COL(L)	iauxc = COL(*l);
...	...

Figura 7.8 Tradução de funções-comando pelo ForC.

Exemplo 7.8. Tradução de *arrays* com dimensões ajustáveis.

Linguagem de entrada: Fortran.

Linguagem de saída: C.

Tradutores: F2C e ForC

Conclusões: 1.

Comentários: Fortran oferece *arrays* com dimensões ajustáveis. Nesses *arrays* as dimensões são conhecidas ao se invocar uma função, de acordo com os parâmetros desta. O tamanho máximo do *array* não pode ultrapassar o tamanho definido no ambiente global. Sendo assim, o código Fortran ilustrado nas figuras 7.9 e 7.10 não há alocação de espaço para o *array* R (como no caso dos *arrays* semidinâmicos em Algol). Nesse caso, a variável R precisa ter sido declarada no ambiente global com um tamanho maior ou igual ao que ocupará após a invocação da função. Ou seja, a declaração local simplesmente altera a forma de cálculo da posição de um determinado elemento em função do início do *array*, não reserva espaço na memória.

Neste exemplo, ambas as traduções preservam a semântica de árvores. No entanto, a solução do ForC nos parece muito mais legível. Observe que o F2C elimina a necessidade de se subtrair 1 de cada índice a cada acesso ao *array* R através de um recuo do início do *array*. No entanto, essa louvável tentativa de economizar tempo de execução, poderia perfeitamente ser feita sem introduzir novas variáveis e mantendo o *array* com duas dimensões. Além disso, observe que neste exemplo o ForC decrementou os índices em tempo de tradução.

Fortran	C
`...`	`...`
`SUBROUTINE SUB(R,I,J)`	`int sub_ (r__, i__, j)`
	`real *r__;`
	`integer *i__, *j;`
	`{`
	`r_dim1 = *i__;`
	`r_offset = r_dim1 + 1;`

`DIMENSION  R(I,J)`	`r__ -= r_offset;`
`R(2,3) = 4`	`r__[r_dim1 * 3 + 2] = (float) 4.;`
`RETURN`	`return 0;`
`END`	
`...`	`...`

Figura 7.9 Tradução de *arrays* com dimensões ajustáveis no F2C.

Fortran	C
`...`	`...`
`SUBROUTINE SUB(R,I,J)`	`void sub(float *r, long *i, long *j) {`
`DIMENSION  R(I,J)`	`#define R(I_,J_) (*(r+(I_)*(*i)+(J_)))`
`R(2,3) = 4`	`  R(2,1) = 4;`
`RETURN`	`  return;`
`END`	`#undef R`
	`}`
`...`	`...`

Figura 7.10 Tradução de *arrays* com dimensões ajustáveis no ForC.

Exemplo 7.9. Tradução de sub-rotinas com **entry**, i.e., pontos de entrada alternativos.

Linguagem de entrada: Fortran.

Linguagem de saída: C.

Tradutores: F2C e ForC

Conclusões: 1.

Comentários: Fortran oferece sub-rotinas com pontos de entrada alternativos, que funcionam como *headers* alternativos definidos em algum ponto da sub-rotina de forma que, quando invocados, iniciam a execução da rotina daquele ponto. A tradução desejada deve manter funções com o mesmo significado desses *headers* alternativos e, além disso, evitar a repetição desnecessária de código.

Por exemplo, uma sub-rotina **cosh(X)**, que calcula o co-seno hiperbólico de x, pode possuir um ponto de entrada **exp(X)**, que calcule e^X. Neste caso é desejável uma tradução que ofereça uma função com semântica igual a $\lambda x.\cosh(x)$ e uma segunda função com semântica igual a $\lambda x.e^X$.

As soluções oferecidas por ambos os tradutores atingem estes objetivos. A principal diferença entre elas é que o ForC define uma **struct** para a passagem de parâmetros, o que torna o programa gerado maior e menos legível. As traduções são ilustradas nas figuras 7.11 e 7.12.

Fortran	C
...	...
	int sub_0_(n__, l, m, n)
	int n__;
	integer *l, *m, *n;
SUBROUTINE SUB(L, M, N)	{
	switch (n__) {
	case 1: goto l_entrada:
	}
L = 2	*l = 2;
M = M + L	*m += *l;
ENTRY ENTRADA(N)	l_entrada:
N = N + 1	++(*n);
RETURN	return 0;
END	}
	int sub_(l, m, n)
	integer *l, *m, *n;
	{
	return sub_0_(0, l, m, n);
	}
	int entrada_(n)
	integer *n;
	{
	return sub_0_(1, (integer *)0, (integer *)0, n);
	}
...	...

Figura 7.11 Tradução do “comando” **entry** no F2C.

Fortran	C
...	...
	struct {
	long int *l;
	long int *m;
	long int *n;
	} _sub;
	void /*FUNCTION*/ sub(long int *l, long int *m, long int *n)
	{
	_sub.l = l;
	_sub.m = m;
	_sub.n = n;
	sub(0);
	return;
	}
	void /*FUNCTION*/ entrada(long int *n)
	{
	_sub.n = n;
	sub(1);
	return;
	}
SUBROUTINE SUB(L, M, N)	_sub_(_entry_)
	int _entry_;
	{
	long int *l = _sub.l;
	long int *m = _sub.m;
	long int *n = _sub.n;
	/* ENTRY Jumps */
	switch(_entry_){
	Case 1: goto _entrada_;
	Case 0:
	Default: break;
	}
	/* Main ENTRY Point */
L = 2	*l = 2;
M = M + L	*m += *l;
ENTRY ENTRADA(N)	_entrada_: /*ENTRY Point*/
N = N + 1	*n += 1;
RETURN	return;
END	} /*end of function*/

Figura 7.12 Tradução do “comando” **entry** no ForC.

Exemplo 7.10. Tradução do "**go to** atribuído".

Linguagem de entrada: Fortran.

Linguagem de saída: C.

Tradutores: F2C e ForC

Conclusões: 1, 5 (F2C).

Comentários: A tradução dos comandos de controle de fluxo de Fortran para C é relativamente simples, se comparada com a tradução de comandos de entrada/saída e da alocação e manipulação de memória. Os diversos tipos de desvios e decisões oferecidos pelo Fortran são traduzidos por ambos os tradutores de forma bastante óbvia. Destacamos, no exemplo ilustrado nas figuras 7.13 e 7.14, o caso do **go to** atribuído, onde a variável de controle recebe primeiro um *label*, através do comando **assign**. Nesse caso, um detalhe muito simples permite que o código gerado pelo F2C seja muito mais eficiente. Simplesmente o tradutor tem o cuidado de gerar uma numeração seqüencial para os valores da variável de controle.

Fortran	C
...	...
40 ASSIGN 40 TO B	b = 0;
...	
GO TO B, (40,50,60)	switch ((int)b) {
	case 0: goto L40;
	case 1: goto L50;
	case 2: goto L60;
	}
...	...

Figura 7.13 Tradução do **go to** atribuído no F2C.

Fortran	C
...	...
40 ASSIGN 40 TO B	b = 40;
...	...
GO TO B, (40,50,60)	switch(b){
	case 40: goto L_40;

	`case 50: goto L_50;`
	`case 60: goto L_60;`
	`}`
`...`	`...`

Figura 7.14 Tradução do **go to** atribuído no ForC.

Exemplo 7.11. Tradução do **equivalence**.

Linguagem de entrada: Fortran.

Linguagem de saída: C.

Tradutores: F2C e ForC

Conclusões: 6.

Comentários: A tradução das declarações de variáveis é uma das partes difíceis da tradução de Fortran para C. A tradução dos comandos **equivalence**, **common** e **data** são, isoladamente, fáceis. No entanto, a combinação de dois ou mais dos mesmos torna a tradução ilegível e, em alguns casos, semanticamente errada. No exemplo ilustrado nas figuras 7.15 e 7.16, vemos que os tradutores tratam o comando **equivalence** de forma semelhante, sendo a tradução do ForC ligeiramente superior, pois usa o escopo de identificadores da linguagem C para implementar o escopo dos ponteiros, enquanto o F2C utiliza o pré-processador para atingir o mesmo efeito.

Fortran	C
`...`	`...`
	`/* Main program */ MAIN__()`
	`{`
	`/* System generated locals */`
	`static integer equiv_0[10], equiv_1[14];`
`INTEGER A, B, C, D, E`	`/* Local variables */`
`DIMENSION A(10)`	`static integer a[10];`
`DIMENSION B(10)`	`#define b (equiv_0)`
`DIMENSION C(10)`	`#define c__ (equiv_0)`
`DIMENSION D(10)`	`#define d__ (equiv_1)`

DIMENSION E(10)	#define e (equiv_1 + 4)
EQUIVALENCE (B, C)	
EQUIVALENCE (D(5), E(1))	
A(5) = 4	a[4] = 4;
B(5) = 5	b[4] = 5;
D(5) = 7	d__[4] = 7;
E(5) = 8	e[4] = 8;
END	} /* MAIN__ */
	#undef e
	#undef d__
	#undef c__
	#undef b
...	...

Figura 7.15 Tradução do **equivalence** no F2C.

Fortran	C
...	...
	main(int argc, char *argv[])
INTEGER A, B, C, D, E	{
DIMENSION A(10)	long int a[10];
	/*EQUIVALENCE translations*/
	double _e1[7], _e0[5];
DIMENSION B(10)	long int *const b = (long*)_e0;
DIMENSION C(10)	long int *const c = (long*)_e0;
DIMENSION D(10)	long int *const d = (long*)_e1;
DIMENSION E(10)	long int *const e = (long*)((char*)_e1 + 16);
EQUIVALENCE (B, C)	
EQUIVALENCE (D(5), E(1))	/*end of EQUIVALENCE */
	f77ini(argc,argv);
A(5) = 4	a[4] = 4;
B(5) = 5	b[4] = 5;
D(5) = 7	d[4] = 7;
E(5) = 8	e[4] = 8;
	clos_units();
	exit(0);
END	} /*end of function*/
...	...

Figura 7.16 Tradução do **equivalence** no ForC.

Exemplo 7.12. Tradução do **equivalence** com uma das variáveis em um bloco **common**.

Linguagem de entrada: Fortran.

Linguagem de saída: C.

Tradutores: F2C e ForC

Conclusões: 4, 6.

Comentários: Neste exemplo, vemos a tradução do comando **equivalence** com uma das variáveis em um bloco **common**. Em Fortran, o comando **equivalence** pode estender o tamanho de um bloco **common**, desde que esta extensão se dê após o final do bloco e não antes de seu início. Observe, no código abaixo, que a posição de memória G(3) está após o final do bloco **common**. Ambos os tradutores usam a sua tradução usual de **common** e **equivalence**, mas o ForC não estende o tamanho do bloco **common**; portanto, não aloca espaço para D(3).

Fortran	C
. . .	. . .
	union {
	struct {
COMMON D, E, F	integer d__, e, f;
	} _1;
	struct {
DIMENSION G(3)	doublereal eqv_pad[2];
	} _2;
	} _BLNK__;
	#define _BLNK__1 (_BLNK__._1)
	#define _BLNK__2 (_BLNK__._2)
. . .	. . .
EQUIVALENCE (E, G(1))	#define g ((integer *)&_BLNK__1 + 1)
D = 7	_BLNK__1.d__ = 7;
E = 8	_BLNK__1.e = 8;
F = 9	_BLNK__1.f = 9;
G(3) = 10	G[2] = 10;
. . .	. . .

Figura 7.17 Tradução de **equivalence** junto com **common** no F2C.

Fortran	C
. . .	. . .
	/*COMMON translations*/
COMMON D, E, F	struct t_bc {
	long int d, e, f;
	} bc;
	/*end of COMMON translations*/
. . .	. . .
DIMENSION G(3)	long int *const g = (long*)&bc.e;
EQUIVALENCE (E, G(1))	
. . .	. . .
D = 7	bc.d = 7;
E = 8	bc.e = 8;
F = 9	bc.f = 9;
G(3) = 10	g[2] = 10;
. . .	. . .

Figura 7.18 Tradução de **equivalence** junto com **common** no ForC.