Ambientes específicos para Análise e Transformação de Software

António Silva[1] and Rui Brito[2]

Departamento de Informática,
Universidade do Minho, Portugal
[1]pg22820@alunos.uminho.pt,[2]ruibrito666@gmail.com

Resumo Existirão diferenças entre compilação e transformação? Para que servem alguns dos ambientes específicos para transformação de software? Estas são apenas algumas das perguntas que este pequeno artigo se propõem a ajudar a esclarecer, utilizando alguns exemplos de linguagens utilizadas por ambientes de transformação e regras possíveis de lhes serem associadas.

Keywords: Análise e Transformação de Software, FermaT, Cocinnelle, TXL, Compilação

1 Introdução

Grande parte do tempo despendido no desenvolvimento de um software, é utilizado na sua manutenção. Quer devido a uma manutenção corretiva (corrigindo erros até então desconhecidos), quer devido a uma manutenção adaptativa (adaptando ou introduzindo novas funcionalidades impostas pelo uso). Por isso foram surgindo linguagens mais direcionadas para a fácil manutenção, e mesmo os próprios IDEs de outras linguagens já existentes começaram a disponibilizar ferramentas que possibilitam uma manutenção mais rápida e fácil do código.

2 Compilação VS Transformação

Existem várias discussões sobre o que é realmente diferente entre a compilação e a transformação de software, uma vez que um compilador também transforma um código de entrada noutro de saída. Todavia uma das grandes diferenças entre um compilador e um transformador é o facto de o código compilado não necessitar de ser legível para o programador. Porque na realidade o programador por norma não vai fazer nada com esse código. Pelo contrário, o código resultante de uma transformação, seja de uma transformação de uma linguagem para outra, seja o resultado de um método por exemplo de *slicing* precisa de poder ser compreendido pelo programador, já que o mesmo terá que ser possivelmente mantido, e o programador lidará diretamente com o mesmo (ao contrário por exemplo do código máquina gerado pela maioria dos compiladores).
Assim é possível perceber que muitas otimizações feitas por compiladores, comprovadas formalmente que o resultado inicial é igual ao resultado final, não podem ser muitas vezes aplicadas em transformadores já que fariam com que o código perdesse a sua legibilidade.

3 Ambientes de transformação especializados

Existem vários ambientes de transformação especializados, como por exemplo:

- Coccinelle [1]
- Stratego/XT [2]
- TXL [3]
- DMS [4]
- ASF+SDF [5]
- FermaT [6]

[1] https://github.com/coccinelle/coccinelle
[2] http://www.strategoxt.org/
[3] http://www.txl.ca/
[4] http://www.semanticdesigns.com/Products/DMS/DMSToolkit.html
[5] http://www.meta-environment.org/
[6] http://www.cse.dmu.ac.uk/ mward/fermat.html

Devido à complexidade de muitas linguagens de programação, a grande parte dos ambientes é desenvolvida tendo por base uma linguagem própria, como por exemplo o FermaT que utiliza o WSL (*Wide Spectrum Language*). No entanto, alguns como o *Coccinelle* recebem diretamente código em linguagem C como entrada.

3.1 Transformações

Sabendo que muitas vezes a compilação, em termos de otimização pode não ser o pretendido, pois o programador pretende perceber o código gerado, as otimizações feitas, e talvez até recomeçar a partir daí, é obrigado a recorrer à transformação de software. Uma opção é fazer tudo manualmente (até recorrendo por vezes a funções de *search & replace*) ou utilizar um ambiente no qual seja possível produzir essas transformações.
Esses mesmos softwares recorrem a procedimentos automatizados que modificam as estruturas de dados, que serão posteriormente utilizadas pelos compiladores, como por exemplo as Árvores de Sintaxe Abstrata. No entanto, e tal como num compilador, os transformadores devem garantir que o estado final do programa será exatamente o mesmo que do programa original, com o a mesma entrada e atendendo ao pedido feito pelo programador na transformação.

3.2 FermaT

O Fermat é um sistema de transformação de programas, utilizado industrialmente voltado para a engenharia reversa, a compreensão do programa e migração entre linguagens de programação. O sistema está a ser usado largamente utilizado na indústria para traduzir módulos Assembler IBM 370 para programas equivalentes legíveis e de fácil manutenção em C e COBOL. O Fermat está disponível como software livre sob a GNU General Public License (GPL).
Um exemplo simples de transformação é a inversão os braços de uma instrução IF. Assumindo que estamos a utilizar uma linguagem WSL com o FermaT, e que temos o seguinte código:

```
total:= 0;
i := 0;
evens := 0;
noevens := 0;
odds := 0;
noodds := 0;
n := n0;
WHILE i <= n DO
  evenflag := A[i] MOD 2;
  evenflag := 0;
  IF FALSE
    THEN evens := evens + A[i];
         noevens := noevens + 1
    ELSE odds := odds + A[i];
         noodds := noodds + 1 FI;
  total := total + A[i];
  i := i + 1 OD;
IF noevens <> 0
  THEN meaneven := evens/noevens
  ELSE meaneven := 0 FI;
IF noodds <> 0
  THEN meanodd := odds/noodds
  ELSE meanodd := 0 FI;
mean := total/(n+1);
evendifference := ABS(meaneven -
    mean);
odddifference := ABS(meanodd-mean)
```

Se utilizarmos o seguinte comando (onde o ficheiro test-1.wsl contém o código anterior, e o test-2.wsl conterá o nosso resultado de saída):

```
dotrans test-1.wsl test-2.wsl
    Semantic\_Slice data=
    evendifference
```

Obteremos o seguinte resultado no ficheiro de saída:

```
evendifference := ABS(REDUCE("+",
    A[0..n0]) / (n0 + 1))
```

Neste caso o comando indicou ao FermaT que pretendíamos apenas garantir que o valor da final variável *evendifference* era igual em ambos os códigos para uma mesma entrada (neste caso o *Array* A). Deste modo, foram eliminadas todas as linhas que não interferiam com a nossa última (que é possível de verificar através do *Program Dependency Graph*).

3.3 Cocinnelle

Coccinelle (que em francês significa joaninha) é uma ferramenta para combinar e transformar o código fonte de programas escritos na linguagem de programação C. O Coccinelle foi inicialmente usado para ajudar na evolução do Linux, com suporte para mudanças de interfaces de programação de aplicações (APIs da biblioteca),

como renomear uma função, adicionando um argumento da função cujo valor é de alguma forma dependente do contexto, e reorganizar a estrutura de dados. A ferramenta está disponível gratuitamente sob uma licença open source. O código fonte para ser combinado e / ou substituído é especificado usando um padrão que é muito semelhante ao C (o Semantic Patch Language).
Por isso também é possível utilizar este tipo de ambientes para fazer um tipo de *refactoring* simples, como mudança de nome de funções. Podemos ver o seguinte exemplo utilizando o Cocinnelle, onde temos o código C de entrada:

```
#include <alloca.h>

    int
    main(int argc, char *argv[]) {
            unsigned int bytes =
                1024 * 1024;
            char *buf;

            /* allocate memory */
            buf = alloca(bytes);

            return 0;
    }
```

E o código do ficheiro .cocci:

```
@@ expression E; @@

    -alloca(E)
    +malloc(E)
```

Aplicando a transformação ao código teremos obteremos as seguintes diferenças:

```
@@ -7,7 +7,7 @@ main(int argc,
    char *argv[]){
              char *buf;

              /* allocate memory */
    -         buf = alloca(bytes);
    +         buf = malloc(bytes);

              return 0;
     }
```

Como podemos ver no exemplo acima os comentários não foram alterados, algo que poderia ser um pouco mais complexo de conseguir utilizando somente o *Find & Replace*.

3.4 TXL

O TXL é uma linguagem de programação de propósito específico originalmente concebido por Charles Halpern-Hamu e James Cordy na Universidade de Toronto em 1985. A sigla "TXL"originalmente significava "Turing eXtender Language"pela finalidade original da linguagem, que era a especificação e prototipagem rápida de variantes e extensões da linguagem de programação Turing, mas já não tem qualquer interpretação significativa.
O TXL moderno é projetado especificamente para criar, manipular e rapidamente prototipar descrições baseadas em linguagens, ferramentas e aplicações utilizando transformação do código fonte. É uma linguagem híbrida (funcional/baseada em regras) usando programação funcional de primeira ordem no nível mais elevado e reescrita de termos no nível inferior. A semântica e implementação de TXL formais baseiam-se em reescrita formal dos termos.
Cada programa TXL tem duas componentes: uma descrição das estruturas de origem para ser transformado, especificado como uma (possivelmente ambíguo) gramática independente do contexto (GIC) utilizando uma forma Backus-Naur estendida, e um conjunto de regras de transformação de árvores, especificados usando padrão/substituição de pares combinado com a programação funcional de primeira ordem. O TXL é projetado para permitir o controlo explícito do programador sobre a interpretação, aplicação, ordem e retrocesso tanto da análise e da reescrita das regras. O primeiro componente analisa a expressão de entrada numa árvore usando o padrão de correspondência. O segundo componente utiliza reescrita de termos de um modo semelhante ao Yacc para produzir a saída transformada.
O TXL é mais comummente usado na análise de software e engenharia de tarefas como recuperação de design, e em prototipagem rápida de novas linguagens de programação e dialetos.
Vejamos o exemplo de uma simples calculadora, com o seguinte código no ficheiro Calculador.txl:

```
define program
        [expression]
end define

define expression
        [term]
```

```
    |   [expression] [addop] [term
        ]
end define

define term
        [primary]
    |   [term] [mulop] [primary]
end define

define primary
        [number]
    |   ( [expression] )
end define

define addop
        '+
    |   '-
end define

define mulop
        '*
    |   '/
end define
```

Introduzindo também as seguintes regras de transformação (também no mesmo ficheiro):

```
rule main
    replace [expression]
        E [expression]
    construct NewE [expression]
        E [resolveAddition] [
            resolveSubtraction]
          [resolveMultiplication]
              [resolveDivision]
          [resolveParentheses]
    where not
        NewE [= E]
    by
        NewE
end rule

rule resolveAddition
    replace [expression]
        N1 [number] + N2 [number]
    by
        N1 [+ N2]
end rule

rule resolveSubtraction
    replace [expression]
        N1 [number] - N2 [number]
    by
        N1 [- N2]
end rule

rule resolveMultiplication
    replace [term]
        N1 [number] * N2 [number]
    by
        N1 [* N2]
end rule

rule resolveDivision
    replace [term]
        N1 [number] / N2 [number]
    by
        N1 [/ N2]
end rule

rule resolveParentheses
    replace [primary]
        ( N [number] )
    by
        N
end rule
```

E utilizarmos o comando:

```
txt -Dapply input.Calculator
```

Onde o input.Calculator é um ficheiro com o seguinte texto:

```
1 + 2 * (3 + 4) * 5
```

Obteremos como resultado 71, com uma descrição do que foi acontecendo no processo de transformação (indicado no parâmetro Dapply).

```
TXL v10.3 (8.3.03) (c)1988-2003
    Queen's University at Kingston
Compiling Txl/Calculadora.Txl ...
Parsing input.Calculator ...
Transforming ...
3 + 4 ==> 7 [resolveAddition]
(7) ==> 7 [resolveParentheses]
1 + 2 * (3 + 4) * 5 ==> 1 + 2 * 7
    * 5 [main]
2 * 7 ==> 14 [
    resolveMultiplication]
14 * 5 ==> 70 [
    resolveMultiplication]
1 + 2 * 7 * 5 ==> 1 + 70 [main]
1 + 70 ==> 71 [resolveAddition]
1 + 70 ==> 71 [main]
71
```

Neste outuput podemos ver claramente as transformações a ocorrerem, na substituição expressões de modo a obtermos um número no resultado final, que neste caso é o 71.
Na realidade a transformação que está a ocorrer pode parecer a muitos um "engodo", já que podem argumentar que uma calculadora não é

na realidade um programa de transformação. Mas na realidade se olharmos para a maneira como nos próprios efetuamos este cálculos, reparamos que recorremos a transformações. É simplesmente uma perspetiva para se olhar para o problema e resolvê-lo. Mesmo na resolução, de equações, ou até sistemas de equações as transformações que fazemos são muitas vezes mais evidentes, não sendo simples "somas e subtrações".

4 Conclusão

Podemos ver nos vários exemplos como as linguagens de transformação podem ser bastante poderosas, cobrindo múltiplos objetivos, quer seja simples *refactoring* de código fonte, quer seja transformações mais complexas como *slicing*, ou inclusivamente criar programas mais complexos como o exemplo apresentado pelo TXL, da criação de uma calculadora.
Também podemos ver que a transformação de software têm bastante ímpeto na indústria onde muitas vezes é necessário que uma aplicação relativamente grande desenvolvida numa linguagem de programação descontinuada ou para um sistema que irá deixar de ter suporte, possa continuar a ser executada, já que os custos financeiros de refazer completamente a aplicação, o tempo despendido, e a necessidade de garantias de essas novas aplicações funcionariam de forma semelhante às primeiras, seriam incomportáveis e relativamente superiores a criar regras de transformação.

Referências

1. Program Transformation (Wikipédia), `http://en.wikipedia.org/wiki/Program_transformation`
2. Coccinelle (Wikipédia), `http://en.wikipedia.org/wiki/Coccinelle_(software)`
3. TXL (Wikipédia), `http://en.wikipedia.org/wiki/TXL_(programming_language)`
4. DMS (Wikipédia), `http://en.wikipedia.org/wiki/DMS_Software_Reengineering_Toolkit`
5. ASF+SDF Meta Environment (Wikipédia), `http://en.wikipedia.org/wiki/ASF\%2BSDF_Meta_Environment`
6. FermaT (Wikipédia), `http://en.wikipedia.org/wiki/FermaT_Transformation_System`
7. A Practical Introduction to TXL, `http://www.txl.ca/docs/TXLintro.pdf`
8. Semantic patching with Coccinelle, `http://lwn.net/Articles/315686/`
9. FermaT (Program-Transformation.Org), `http://www.program-transformation.org/Transform/FermaT`