Relatório do trabalho
Compilador para Tiger-0

Pedro Miguel Ribeiro Carvalho, up201805068

Introdução

Este relatório contém a explicação da implementação do compilador para um subconjunto da linguagem *Tiger*, o subconjunto *Tiger-0*. *Tiger* é uma pequena linguagem definida no livro *Modern Compiler Implementation in ML* por *Andrew Appel* (*Cambridge University Press, 1998*). Este relatório define um subconjunto de *Tiger*, chamado *Tiger-0*. Segue-se assim uma pequena descrição deste subconjunto. O *Tiger-0* é uma pequena linguagem imperativa com inteiros, strings, arrays, estruturas básicas de fluxo de controlo e funções . As diferenças mais notáveis para a linguagem *Tiger* definida no livro de *Appel* são: *Tiger-0* não permite definições de funções encapsuladas e registo de tipos, e suporta apenas arrays inteiros.

program :
    let decl-list in expr-seq
decl-list :
    decl
    decl-list decl
decl :
    var-decl
    fun-decl
fun-decl :
    function id(type-fieldsopt) = expr
    function id(type-fieldsopt):type-id = expr
type-fields :
    type-field
    type-fields , type-field
type-field :
    id : type-id

expr:
    integer-constant
    string-constant
    lvalue
    expr binary-operator expr
    -expr
    lvalue := *expr*
    id (*expr-list*$_{opt}$)
    (expr-seq$_{opt}$ *)*
    if *expr* then *expr*
    if *expr* then *expr* else *expr*

while *expr do expr*
for *id* := *expr* to *expr* do *expr*
break
let *var-decl-list* in *expr-seq* end

lvalue :
id

expr-seq :
expr
expr-seq ; *expr*

expr-list :
expr
expr-list , expr

var-decl-list:
var-decl
var-decl-list var-decl

var-decl :
var id := expr

programa que transforma qualquer fórmula proposicional em forma normal conjuntiva (**CNF**) ou em forma normal disjuntiva (**DNF**) a pedido do utilizador. Para compilar e executar o programa é bastante simples, basta compilar o ficheiro **to_cnf_dnf.pl** e de seguida executar uma das seguintes querrys:

transform(A, D, cnf).
transform(A, D, dnf).

Em que no argumento **A** é suposto ser passada a fórmula que se pretende transformar. Para o argumento **D** pode ser ser uma qualquer variável para guardar o resultado da transformação, contudo também podemos introduzir uma fórmula para verificar se a transformação de **A** corresponde a **D**. E por último **cnf** ou **dnf** para que tipo de formula pretendemos transformar.

Algoritmo

Para passar de uma fórmula proposicional qualquer para uma **CNF** ou **DNF** temos de seguir o seguinte algoritmo:

1. Remover equivalências e implicações;
2. Remover negações, isto é, colocar todas as negações o mais perto dos literais possível;
3. Aplicar a propriedade distributiva:
    a. Para **CNF**, aplicar a distributividade sobre disjunção;
    b. Para **DNF**, aplicar a distributividade sobre conjunção.

Explicação da implementação

Em primeiro lugar, foi necessário definir os operadores das operações lógicas, para isso foi usado o predicado **op(+Precedence, +Type, :Name).** Em que **Precedence** é um inteiro entre 0 e 1200. Quando é 0 remove a declaração existente e quanto maior for o número maior é a precedência do operador perante outros. **Type** define se o operador é **infixo**, **sufixo** ou **prefixo**. Para além de permitir explicar a relação de precedência dos argumentos do operador, isto é, se os argumentos podem ter precedência menor ou menor ou igual ao operador. O argumento **Name** é simplesmente o nome do operador.

Assim sendo, os operadores foram definidos da seguinte maneira:

Negação -> **op(200, fy, ~).**
Conjunção -> **op(0, yfx, (/\)).**
Conjunção -> **op(400, xfy, (/\)).**
Disjunção -> **op(0, yfx, (\/)).**
Disjunção -> **op(500, xfy, (\/)).**
Implicação -> **op(600, xfy, =>).**
Equivalência -> **op(700, xfy, <=>).**

De ressalvar que, inicialmente a ideia era usar como operador para disjunção “||**”**, contudo quando compilava recebia um erro “**end_of_file_in_quasi_quotation”**. Como tal, optei por usar “\/**”** para representar a disjunção. E por questão de coerência foi “/**”** para representar a conjunção.

Para a negação e para a equivalência usei a definição dada nos slides das aulas teóricas. Removemos a associatividade à esquerda dos operadores já definidos “/**”** e “\/**”** para definir novos operadores com associatividade à direita. Como os operadores de disjunção e conjunção em proposições lógicas se

assemelham aos operadores da soma e multiplicação das expressões aritméticas, usei o mesmo número de precedência para ambos. E finalmente a equivalência coloquei com a maior precedência uma vez que é o mais prioritário na transformação de uma fórmula normal para **CNF** ou **DNF**.

Como quer para transformar em **CNF** ou **DNF** os dois primeiros passos do algoritmo são iguais os predicados são reutilizados. O predicado **rewrite_connectives()** não faz nada mais que rescrever as fórmulas removendo as implicações e as equivalências. O predicado **negation()** também nada mais faz que aplicar as leis de Morgan e puxando as negações de fora para dentro, isto é, garantir que apenas os literais podem estar negados. Os predicados **cnf_distribute_recursive()** e **dnf_distribute_recursive()** simplesmente são predicados recursivos que chamam o predicado **cnf_distribute** ou **dnf_distribute,** que apenas aplicam a propriedade distributiva uma vez, garantindo assim que as fórmulas ficam na forma pretendida.