Tomasz Słojewski
nr albumu 300268

TKOM dokumentacja projektu

Zadanie O3

Zdanie polega na stworzeniu programu który optymalizuje kod programu w taki sposób, że wyciąga z pętli i przenosić przed pętle te instrukcje które są możliwe do przeniesienia bez wpływania na wynik działania kodu.
Kod może zawierać zagnieżdżone pętle. Jedna instrukcja może być przeniesiona poza kilka zagnieżdżonych pętli.

Język który będzie mógł zoptymalizowany to podzbiór języka **C**

Założenia

- Program jest napisany w języku C++
- Wszystkie instrukcje dostępne w implementowanym języku są wymienione niżej w sekcji „Instrukcje i wyrażenia dostępne w optymalizowanym kodzie”
- Optymalizator potrafi przenosić instrukcje przypisania zawierające stałą liczbę lub wyrażenia arytmetyczne złożone wyłącznie z operatorów arytmetycznych i liczb
- Optymalizator potrafi także przenosić niektóre deklaracje zmiennych
- Program jest aplikacją konsolową optymalizującą jeden plik którego nazwa jest argumentem wywołania aplikacji.
- Zoptymalizowany kod jest zapisywany do pliku, którego nazwa została podana jako argument wywołania aplikacji lub jeśli nie ma drugiego argumentu to kod jest wypisywany na standardowe wyjście
- Inicjacja każdej zmiennej odbywa się w oddzielnej instrukcji
- W części inicjacyjnej pętli for musi znaleźć się inicjacja zmiennej, przypisanie wartości lub może pozostać pusta operacja (średnik).
- W części aktualizacyjnej pętli for należy zmienić wartość zmiennej lub pozostawić tę część pustą
- W pętli musi występować warunek logiczny
- Blok operacji przypisanych do pętli zaczyna się od ‘{’ i kończył ‘}’
- Ze względu na możliwe niejednoznaczności niedopuszczane są wyrażenia zawierające 3 takie same znaki operatorów: (+++ lub ---) np. ‘a+++b’.
- Nie można też w jednej operacji jednocześnie dokonać modyfikacji zmiennej pre oraz post inkrementacyjnej/dekrementacyjnej np. ‘--a++’
- Maksymalna długość pojedynczego tokena wynosi 64 znaki.

Zadania programu:

- Odczytywanie, analizowanie i optymalizowanie podanego kodu zgodnego z zdefiniowaną składnią.
- Zgłaszanie błędów wykrytych podczas analizy kodu
- Tworzenie zmiennych w trzech obsługiwanych typach
- Obsługa zakresu dostępności zmiennych.
- Używanie pętli for
- Kod po optymalizacji musi być równoznaczny z kodem oryginalnym.

Optymalizowane wyrażenia:

Program przenosi dwa rodzaje wyrażeń przed pętlę w której takie wyrażenie występuje.

- przypisanie wartości do zmiennej
  - Przenoszone są wszystkie możliwe do przeniesienia przypisania, które nie zmienią działania programu.
  - Warunki które muszą być spełnione, aby takie wyrażenie zostało przeniesione:
    - przypisanie składa się jedynie w pojedynczego elementu będącego liczbą lub z wyrażenia arytmetycznego zawierającego jedynie liczby i operatory (bez zmiennych)
    - zmienna nie została wcześniej użyta w bloku z którego ma zostać przeniesiona ani w operacji for do której blok należy
    - wartość zmiennej nie została zmieniona po żadnym użyciu jej
    - zmienna została zdefiniowana w nadrzędnym bloku lub na jeszcze niższym poziomie
- definicja zmiennej
  - Warunki które muszą być spełnione, aby takie wyrażenie zostało przeniesione:
    - nazwa zmiennej nie może być użyta na głębokości na której znajduje się definicja więcej niż jeden raz (jedyne wystąpienie to to w definicji)
    - nazwa zmiennej nie może być użyta w bloku nadrzędnym

Instrukcje które mogą być przenoszone:

- zmienna = liczba;
  Przykłady:
  - a = 5;
  - a = 3.14;
- zmienna = wyrażenie złożone z samych liczb;
  - a = 5 * 3;
  - a = 12 * 3 + 56 / 5;
- typ danych zmienna = wyrażenie z samych liczb;
  - int a = 5;
  - int b = 12 * 5;

Instrukcje i wyrażenia dostępne w optymalizowanym kodzie:

- pętla for
  - for(*przypisanie_wartości_iteratorowi*; *warunek_logiczny*; *aktualizacja_iteratora*) {*zbiór_instrukcji*}
- instrukcja przypisania
  - *nazwa_zmiennej = liczba*;
  - *nazwa_zmiennej = wyrażenie*;
- wyrażenie arytmetyczne
  - *liczba operator nazwa_zmiennej*
  - *liczba1 operator liczba2*
  - *nazwa_zmiennej operator liczba*
  - *nazwa_zmiennej1 operator nazwa_zmiennej2*
- dostępne operatory arytmetyczne
  - mnożenie: *
  - dzielenie: /
  - dodawanie: +
  - odejmowanie: -
- operatory logiczne
  - mniejsze: <
  - większe: >
  - równe: ==
- inkrementacja
  - *zmienna*++;

    - ++*zmienna*;
- dekrementacja
    - *zmienna*--;
    - --*zmienna*;
- odwołanie do indeksu w tablicy (każdy indeks w tablicy traktowany jest jako zmienna)
    - *nazwa_tablicy*[*numer_indeksu*]
- inicjacja zmiennej
    - typ nazwa_zmiennej;
    - typ nazwa_zmiennej = wyrażenie;

Obsługiwane typy danych:

- int – maksymalnie 9 cyfr
- long – maksymalnie 12 cyfr
- double – maksymalnie 4 cyfry przed przecinkiem i 15 po przecinku

Gramatyka

program = {operation};

operation = loop | ([variable, (incrementalOperator | assigment) | initiation | preIncrementation], ";");

loop = "for(", [initiation | (variable, assigment)], ";",condition,";",[preIncrementation| (variable, incrementalOperator|assigment)],")","{",{operation},"}";

initiation = type, variable, [assigment];

assigment = "=", arithmeticExpression;

condition = arithmeticExpression, relationOperator, arithmeticExpression;

arithmeticExpression = primaryExpression, {additiveOperator|multiplicativeOperator, primaryExpression};

primaryExpression = (variable, [incrementalOperator]) | number |preIncrementation;

preIncrementation = incrementalOperator , variable;

variable = variableName, ("[", arithmeticExpression, "]");

variableName = sign, {sign| digit};

number = ("0"|digitWithoutZero,{digit}),[".",digit,{digit}];

digit = "0" | digitWithoutZero;

digitWithoutZero = "1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9";

sign = letter | "_";

letter = "a"|….|"z"|"A"|….|"Z";

relationOperator = "<" | ">" | "==";

additiveOperator = "+" | "-";

multiplicativeOperator = "*" | "/";

incrementalOperator = "++" | "–";

type = "int" | "long" | "double";

Przykłady optymalizacji kodu:

Przed optymalizacją	Po optymalizacji
1. Proste przeniesienie przypisania	

int a;
int b[100];
int c[100];
for (int i=1; i<100; i++) { a=5; b[i] = c[i] *a; }
	int a;
	int b[100];
	int c[100];
	a=5;
	for (int i=1; i<100; i++) { b[i] = c[i] *a; }

2. Przeniesienie o dwa poziomy	

int i;
int j;
int a;
int b[20];
for (i=1; i<20; i++){
for (j=1; j<20; j++){a = 5; b[j] = a * i;
}}
	int i;
	int j;
	int a;
	int b[20];
	a = 5;
	for (i=1; i<20; i++){
	for (j=1; j<20; j++){b[j] = a * i;
	}}

3. Przeniesienie przypisania, gdy zmienna jest użyta w instrukcji for	

int i;
int j;
int a;
int b[20];
int c[20];
for (i=1; i<a; i++){
for (j=1; j<20; j++){a = 5; b[j] = a * i;
}
a = 5 * b[i];
c[i] = a – 10;
}
	int i;
	int j;
	int a;
	int b[20];
	int c[20];
	for (i=1; i<a; i++){
	a = 5;
	for (j=1; j<20; j++){b[j] = a * i;
	}
	a = 5 * b[i];
	c[i] = a – 10;
	}

4. Niemożność przeniesienia przypisania, ponieważ zmieniłby to działanie programu	

int a = 3;
int b[20];
for (int i=1; i<20; i++){b[i] = 2 * a; a = 5; a = a + 1;}
	int a = 3;
	int b[20];
	for (int i=1; i<20; i++){b[i] = 2 * a; a = 5; a = a + 1;}

5. Niemożność przeniesienia zmiennej która jest używana i zmieniana jest jej wartość	

int i;
int a;
int b[20];
int c;
for (i=1; i<20; i++){a = 5; c = 10; b[i] = 2 * a + c; a = a + 1;}
	int i;
	int a;
	int b[20];
	int c;
	c = 10;
	for(i = 1; i < 20; i++){
	a = 5;
	b[i] = 2 * a + c;
	a = a + 1;
	}

6. Przeniesienie definicji zmiennej	
for(int a = 0; a < 10; ++a){ for(int b = 0; b < 20; ++b){	int x = 0; for(int a = 0; a < 10; ++a){

int x = 0;
for(int c = 0; c < 10; ++c){
x = x + 2;
}
}
}
	for(int b = 0; b < 20; ++b){
	for(int c = 0; c < 10; ++c){
	x = x + 2;
	}
	}
	}

7. Błąd lexera – nieznany token	
int a = #;	LEXER ERROR: line 1; sign 9 #; Unknow token PARSER ERROR: line 1; sign 8 expected expression after assign operator Cannot optimize - error in code
8. Błąd parsera – oczekiwano średnik na koniec operacji	
int a = 10 a = 5;	PARSER ERROR: line 2; sign 0 expected semicolon after operation Cannot optimize - error in code
9. Błąd analizatora semantycznego – zdefiniowano zmienną, a nie tablicę	
int b = 10; int a = b[3];	ANALYZER ERROR: line 2; sign 9 Undefined variable name b[] Cannot optimize - error in code

Testy:

Do testowania powstały testy jednostkowe sprawdzające działanie poszczególnych modułów aplikacji, które znajdują się w pliku test/tester.cpp i są napisan eprzy pomocy biblioteki boost. Dodatkowo przykłady podane wyżej znajdują się w plikach folderu in, a kod wynikowy tych przykładów – w folderze out

Wykrywanie błędów:

Każdy z modułów zgłasza wykryte przez siebie błędy
Zgłoszone błędy wyświetlają:

- komunikat o typie zaistniałego błędu
- informację który z modułów wykrył ten błąd
- miejsce jego wystąpienia (linia i numer znaku)
- fragment kodu którego on dotyczy – w przypadków błędów lexera

Działanie programu:

Program został napisany przy użyciu języka C++.
Plik z kodem przeznaczony do optymalizacji jest przekazywany jako argument programu. Do projektu jest dołączony plik makefile który będzie umożliwiał kompilację przy pomocy polecenia *make*. Uruchamianie programu będzie dokonywane przy pomocy komendy ./optymalizator plik_wejściowy [plik_wyjściowy]
Kod po optymalizacji jest umieszczany w pliku wyjściowym o podanej nazwie lub wyświetlany na standardowym wyjściu.

Analiza kodu
Analiza kodu składa się z 3 etapów.

Analizator leksykalny:
Dzieli kod na tokeny. Wczytuje kolejne znaki, aż do stwierdzenia końca tokenu. Każdy z tokenów jest reprezentowany przez obiekt klasy Token.
Każdy z tokenów posiada pola: typ tokenu, wartość będącą typem całkowitym, zmiennopozycyjnym lub tekstowym oraz pozycję tokenu.
Tokeny mogą posiadać następujący typ:

- Słowo kluczowe - „for”
- Operator przypisania - „=’
- Operator addytywny - „+”, „-”
- Operator multiplikatywny - „*”, „/”
- Operator inkrementacyjny/dekrementacyny „++”, „--”
- Typ danych – „int”; „long”, „double”
- Operator relacyjny - „<”, „>”, „==”
- Nawias okrągły - „(”, „)”
- Nawias blokowy - „”{, „}”
- Liczba – „11.3”, „902”, „2321.42134”
- Koniec pliku
- Nieznany token
- Błędny token

Analizator składniowy:
Pobiera od lexera kolejne Tokeny, a następnie grupuje je w drzewo, jednocześnie sprawdzając, poprawność gramatyczną kodu, zgodną z gramatyką podaną powyżej.

Analizator semantyczny:

Pobiera od analizatora składniowego drzewo i analizuje je pod względem poprawności semantycznej. Analizator przechodzi po drzewie w głąb.
Analizator semantyczny sprawdza istnienie i zasięg zdefiniowanych zmiennych, Przechowywanych w tablicy symboli.

Optymalizacja:
Optymalizacja odbywa się poprzez przejście po drzewie, zbierając informacje o kolejnych występujących zmiennych, a następnie sprawdzeniu czy w momencie wystąpienia przypisania lub definicji spełnione są warunki przedstawione w punkcie „Optymalizowane wyrażenia”.

Tablica symboli:
Tablica symboli jest vectorem przechowującym informacje o zmiennych.
Na etapie analizy przechowuje informacje o nazwie zmiennej, poziomie na którym pojawia się jej definicja oraz o tym czy jest ona tablicą.
Na etapie optymalizacji kodu przechowujemy dodatkowo informację o tym czy zmienna została już użyta.