Metaprogramiranje predlošcima

Dragaš, Domagoj

Master's thesis / Diplomski rad

2021

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Science / Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:217:236505

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Domagoj Dragaš

METAPROGRAMIRANJE PREDLOŠCIMA

Diplomski rad

Voditelj rada:
prof. dr. sc. Mladen Jurak

Zagreb, 6. rujna 2021.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Sadržaj **iii**

Uvod **1**

1 Uvod u metaprogramiranje **2**
1.1 Metaprogramiranje vrijednosti 4
1.2 Metaprogramiranje tipova 6
1.3 Hibridno metaprogramiranje 8

2 Motivacija za predložne izraze **11**
2.1 Naivna implementacija 11
2.2 Problemi 13
2.3 Rezultati mjerenja 16

3 Predložni izrazi u implementaciji vektora **18**
3.1 Operandi 19
3.2 Klasa Vector 23
3.3 Operatori 25
3.4 Osvrt 26
3.5 Verifikacija i performanse 29

4 Predložni izrazi u implementaciji matrica **32**
4.1 Promjene u dosadašnjoj implementaciji 34
4.2 Operandi 38
4.3 Klasa Matrix 40
4.4 Operatori 44
4.5 Verifikacija i performanse 46

5 Eigen **50**
5.1 Matrice 51
5.2 Vektori 55

5.3 Usporedba Eigen-a i naše implementacije 56

6 Predložni izrazi kroz povijest **59**
6.1 Začeci ideje 59
6.2 Popularizacija tehnike 61

Bibliografija **63**

Uvod

Klase spremnika (eng. *container classes*) predstavljaju posebnu vrstu klasa dizajniranih da spremaju kolekciju nekakvih objekata. Takve klase imaju široko područje primjene, od biblioteka matrica u znastvenom računanju do reprezentacije grafova u modeliranju. Često su skupovi podataka koje instance takvih klasa spremaju ogromni veličinom stoga je nužno obratiti pozornost na efikasnost operacija nad tim podacima. Naime, ukoliko programer ne obrati pažnju na performanse, kao što ćemo pokazati u ovom radu, dolazi do nepotrebnog kopiranja unutar memorije. Ako su skupovi podataka preveliki ta kopiranja oduzimaju puno vremena pa samim time efikasnost algoritma neće biti zadovoljavajuća.

Tema ovog rada upravo opisuje jednu od tehnika u programskom jeziku C++ kojom se sprječava nepotrebno kopiranje, ali i smanjuje broj čitanja odnosno zapisavanja memorijskih lokacija. Navedenu tehniku nazivamo predložni izrazi (eng. *expression templates*). Tehnika spada pod metaprogramiranje pa prvo dajemo uvod u metaprogramiranje i metaprogramiranje predlošcima. Zatim stvaramo biblioteku vektora i pripadnih operacija naivnim pristupom i uočavamo propuste u samom izvođenju i efikasnosti. Probleme rješavamo pomoću predložnih izraza. Također pokazujemo kako tehniku iskoristiti u implementaciji matrica i matričnih operacija. Na kraju uspoređujemo učinkovitosti dobivenih implementacija sa Eigen bibliotekom koja također koristi predložne izraze, ali i brojne druge tehnike optimizacija.

Poglavlje 1

Uvod u metaprogramiranje

Metaprogramiranje se obično definira kao programerska tehinka u kojoj računalni programi imaju mogućnost tumačenja ostalih programa kao ulazne podatke. To znači da program može biti dizajniran da čita, generira, analizira ili čak mijenja sam sebe (ili druge programe) tijekom izvođenja. Kao jednostavan primjer metaprogramiranja izlažemo skriptu (program) u komandnoj liniji operacijskog sustava Unix [9]:

```
echo '#!/bin/sh' > program
for i in $(seq 992)
do
    echo "echo $i" >> program
done
chmod +x program
```

Primjer ilustrira generiranje novog programa. Nakon izvođenja, naša skripta generira novi program od 933 linija koda koji ispisuje sve prirodne brojeve od 1 do 932. Ovo naravno nije najefikasniji način za ispis brojeva, već samo primjer koji opisuje jednu od karakteristika metaprogramiranja.

Metaprogramiranje predlošcima je programerska tehnika u kojoj prevodilac koristi predloške (eng. *templates*) da bi generirao privremeni izvorni kod, koji se zatim spaja sa ostatkom izvornog koda i tek onda prevodi u objektni kod. Ova tehnika je procvjetala korištenjem predložaka u programskom jeziku C++, ali se koristi i u drugim programskim jezicima poput Curl, D, Nim, and XL. Kako preodlošci ne uzimaju programe kao vlastite ulazne podatke, strogo po definiciji, metaprogramiranje predlošcima nije podskup metaprogramiranja. Bez obzira na to, predlošci su dio programa koji se koriste za generiranje novog koda pa se dio samog programa može u neku ruku shvatiti kao vlastiti ulazni podatak. Stoga lako uočavamo sličnosti između definicija i jednostavno uvažamo metaprogramiranje predlošcima kao meta-programersku tehniku i zato u nastavku, radi jednostavnosti,

ne radimo razliku između dva pojma.

Formalnosti na stranu, postavlja se pitanje, zašto bi metaprogramiranje bilo poželjno? Kao i kod većina ostalih programerskih tehnika, cilj je postići što veću funkcionalnost sa što manje truda, gdje se pod trudom može smatrati broj linija koda, trošak održavanja itd. Osnovna karekteristika metaprogramiranja je da se neko korisničko izračunavanje događa tijekom kompilacije. Jedna od motivacija iza toga je zasigurno efikasnost jer na taj način žrtvujemo trajanje kompilacije radi bržeg izvođenja programa. Taj dio opisujemo u metaprogramiranju vrijednosti. Druga moguća motivacija je direktan utjecaj na asemblerski izlaz na koji također možemo ubrzati izvođenje programa, jednu od karakteristika hibridnog metaprogramiranja. I posljednje, ponekad je metaprogramiranje nužno kao što ćemo vidjeti u metaprogramiranju tipova.

Još napomenimo da se ne može bilo kakva upotreba predložaka u C++-u smatrati metaprogramiranjem. Strogo formalno, to ne bi trebao biti slučaj, ali prešutno zahtijevamo i još jedan uvjet. Naime, promotrimo ovu jednostavnu parametriziranu funkciju koja prima dva argumenta istog tipa i uspoređuje ih.

```
template <typename T>
bool compare(T x, T y){
    if(x == y)
        return true;
    return false;
}
```

Za poziv `compare(15,21);` naš prevodilac generira funkciju

```
int compare(int x, int y){
    if(x == y)
        return true;
    return false;
}
```

koju spaja sa ostatkom izvornog koda i koristi za evaluaciju izraza. I to je sve. Vidimo dakle da imamo isti efekt kao i da nismo parametrizirali funkciju `compare` odnosno da nemamo nikakvo računanje tijekom kompilacije. Stoga nekakav oblik računanja tijekom kompilacije je nužan uvjet za metaprogramiranje. U nastavku ukratko opisujemo tri glavna tipa metaprogramiranja. Za detalje pogledati u [6].

1.1 Metaprogramiranje vrijednosti

Kao što ime govori, metaprogramiranje vrijednosti je tehnika čiji cilj je dobivanje neke željene vrijednosti, npr. računanje drugog korijena, tijekom kompilacije. Dakle, računanje u najužem mogućem smislu. Kako se radi o metaprogramiranju račun mora biti isključivo obavljen tijekom kompilacije.

Prije C++14 standarda, računanje vrijednosti tijekom kompilacije nije bio osobit problem; sve se svodilo na rekurzivne pozive funkcijskih predložaka ili predložaka klase. Već znamo da se predlošci instanciraju tijekom kompilacije stoga nije teško uvidjeti samu ideju iza toga. Pokažimo kako bi to izgledalo za računanje potencije broja 3:

```
template <int N>
struct Pow3{
    enum { result = 3 * Pow3<N-1>::result};
};

template <>
struct Pow3<0>{
    enum { result = 1 };
};
```

Ako želimo izračunati potenciju broja četiri, to radimo pozivom `Pow3<4>::result`. Objasnimo ukratko kako i zašto znamo da će se to evaluirati tijekom kompilacije.

Najprije, prevodilac generira funkciju `Pow3<4>`. Prva i osnovna stvar koju trebamo uočiti je da je `result` ovdje enumerator, a enumeratori su cjelobrojne konstante; moraju se evaluirati tijekom kompilacije. Stoga se poziv `Pow3<4>::result` ili evaluira tijekom kompilacije ili rezultira kompilacijskom greškom. Nadalje, prevodilac `Pow3<4>::result` prevodi u `3* Pow3<3>::result`. Vidimo da se stvari nisu promijenile odakle smo krenuli. Ponovno koristimo enumerator odnsono cjelobrojnu konstantu koju prevodilac mora izračunati. Rekurzivnim pozivima dobivamo izraz `3* 3* 3* 3* Pow3<0>::result`. Na kraju, pošto je `Pow3<0>` puna specijalizacija danog predloška strukture, ona se uvijek preferira i rekurzija završava, a `Pow3<0>::result` se prevodi u konstantu `1`. Prevodilac je sada u stanju u potpunosti izračunati traženu potenciju broja 3.

Srećom, stvari se mijenaju nabolje dolaskom C++11 standarda, a pogotovo dolaskom C++14 standarda. Sada više nije potrebno koristiti rekurzivne pozive niti enumeracije radi izračunavanja vrijednosti tijekom kompilacije već sve to zamijenjuju takozvane `constexpr` funkcije. Takve funkcije nisu ništa drugo nego funkcije sa pridruženom ključnom riječi `constexpr` na početku signature što označava da se radi o konstantom izrazu. Konstantan izraz je izraz koji se može izračunati za vrijeme kompilacija i čija se vrijednost ne može promijeniti. Funkcije sa priključkom `constexpr` će se izvršiti za vrijeme kompilacije ako

se pozove sa argumentima koji su poznati za vrijeme kompilacije; inače se ponašaju kao i svake druge funkcije. Na taj način ne moramo imati dvije funkcije iste funkcionalnosti - jednu običnu, a drugu za izračunavanje za vrijeme kompilacije.

Ipak, postoje određena ograničenja `constexpr` funkcija u odnosu na obične funkcije. Parametri i povratna vrijednost moraju biti literalni tipovi. Također unutar tijela funkcije nisu dozvoljeni `try`-blokovi, `goto` naredbe i slično jer se tijelo funkcije izvršava za vrijeme kompilacije.

S tim na umu, implementacija funkcije za izračunavanje treće potencije tijekom kompilacije se svodi na standardan algoritam.

```
constexpr int Pow3(int n){
    int rez = 1;
    for(int i = 0; i<n; ++i)
        rez *= 3;
    return rez;
}

int main(){
    std::cout << Pow3(10); // 594049
    return 0;
}
```

U to se možemo i uvjeriti koristeći interaktivni prevodilac koji uz uobičajene usluge prikazuje korisniku i asemblerski izlaz prevedenog koda. Za ove potrebe koristimo godbolt [4].

Mogući asemblerski izlaz

```
1   main:
2           sub     rsp, 8
3           mov     esi, 59049
4           mov     edi, OFFSET FLAT:_ZSt4cout
5           call    std::basic_ostream<char, std::char_traits<char> >::operator<<(int)
6           xor     eax, eax
7           add     rsp, 8
8           ret
9   _GLOBAL__sub_I_main:
10          sub     rsp, 8
11          mov     edi, OFFSET FLAT:_ZStL8__ioinit
12          call    std::ios_base::Init::Init() [complete object constructor]
13          mov     edx, OFFSET FLAT:__dso_handle
14          mov     esi, OFFSET FLAT:_ZStL8__ioinit
15          mov     edi, OFFSET FLAT:_ZNSt8ios_base4InitD1Ev
16          add     rsp, 8
```

1.2 Metaprogramiranje tipova

Metaprogramiranje tipova se odnosi na računanje tipova tijekom prevođenja. Primamo nekakav tip podatka kao ulaz, a kao izlaz stvorimo tip koji nam odgovara. Zamislimo da imamo polje proizvoljne dimenzije i želimo dobiti tip koje to polje sprema. To je samo jedna od mogućih motivacija zašto bi stvaranje novih tipova uopće bilo korisno. Također možemo iz tipa koji je možda referenca dobiti isti taj tip ikakvih referenci, možemo za određene tipove stvoriti referencu na taj isti tip (s čime ćemo se susresti u poglavlju 3) itd. Većina tih stvaranja tipova ne uključuje proces računanja jer samo preko parcijalne specijalizacije predložaka odbacujemo ili prihvaćamo tip koji nam odgovara; ne koristimo rekurzivne pozive. Zato upravo izabiremo tzv. skidanje slojeva polja radi dobivanja sirovog tipa kao nekakav trijavalan primjer računanja tipova tijekom prevođenja. Za to će nam poslužiti predložak klase `RemoveExtents`.

```
// primarni predložak: ako T nije polje vraća T
template<typename T>
struct RemoveAllExtentsT {
    using Type = T;
};

// parcijalna specijalizacija za polja
template<typename T, std::size_t SZ>
struct RemoveAllExtentsT<T[SZ]> {
    using Type = typename RemoveAllExtentsT<T>::Type;
};

// parcijalna specijalizacija za polja
template<typename T>
struct RemoveAllExtentsT<T[]> {
    using Type = typename RemoveAllExtentsT<T>::Type;
};

template<typename T>
using RemoveAllExtents = typename RemoveAllExtentsT<T>::Type;
```

Prva stvar koju uočavamo je korištenje ključne riječi `using` za izračun tipova što sugerira izračunavanje tijekom prevođenja. Taj dio odmah opravdava metaprogramiranje iz naziva tehnike. `RemoveAllExtents` vraća sirovi tip kojeg polje sprema tako što instancira klase oblika `RemoveAllExtentsT<T[SZ]>` ili `RemoveAllExtentsT<T[]>` rekurzivnim pozivima sve dok ne dođe do tipa koji ne predstavlja polje. Znamo da prevodilac tijekom razrješivanja preopterećenja uvijek bira najspeciliziraniji predložak ukoliko se najbolji kandidat može dobiti instanciranjem više predložaka odnosno bira se onaj koji se

može pozvati na manjem skupu argumenata. To osigurava željeno ponašanje koda. Dakle, mi tijekom prevođenja instanciramo klasu upravo onim tipom koji nam treba. Predložak `RemoveAllExtents` koristimo na sljedeći način:

```
RemoveAllExtents<int[]> // vraća
RemoveAllExtents<int[5][10]> // vraća int
RemoveAllExtents<int[][10]> // vraća int
RemoveAllExtents<int(*)[5]> // vraća int(*)[5]
```

1.3 Hibridno metaprogramiranje

Sa metaprogramiranjem vrijednosti i metaprogramiranjem tipova možemo računati vrijednosti odnosno tipove tijekom prevođenja. Međutim metaprogramiranje nudi i više od toga. Metaprogramiranjem možemo programski utjecati na asemblerski izlaz tijekom kompilacije i na taj način ubrzavamo izvođenje programa. To nazivamo hibridnim metaprogramiranjem. Ilustrirajmo tehniku jednostavnim primjerom. Pretpostavimo da nam je cilj izračunati skalarni umnožak dva objekta tipa `std::array`. Sjetimo se da je `std::array` predložak spremnika čija pojednostavljena deklaracija u datoteci zaglavlja bi mogla izgledati poput:

```
namespace std {
    template<typename T, size_t N> struct array;
}
```

gdje `N` označava broj elemenata tipa `T` u spremniku. Skalarni umnožak stoga možemo izračunati na sljedeći način:

```
template<typename T, std::size_t N>
auto dotProduct(std::array<T, N> const& x,
                std::array<T, N> const& y)
{
    T result{};
    for (std::size_t k = 0; k<N; ++k)
        result += x[k]*y[k];

    return result;
}
```

Neka izravna kompilacija `for` petlje će proizvesti instrukcije grananja (zbog provjeravanja uvjeta) koje na nekim računalima mogu dovesti do dodatnih troškova u efikanosti u usporedbi sa jednolinijskim izvršavanjima sljedećeg koda:

```
result += x[0]*y[0];
result += x[1]*y[1];
```

```
result += x[2]*y[2];
result += x[3]*y[3];
...
```

Srećom, moderni prevodioci će optimizirati petlju u najefikasniju formu pogodnu za ciljanu platformu. Čisto radi ilustracije tehnike preradit ćemo skalarni umnožak tako da izbjegnemo korištenje petlji i dobijemo željeni asemblerski izlaz. Takvu reformu petlji nazivamo odmotavanje petlji (eng. *loop unrolling*).

```
template<typename T, std::size_t N>
struct SkalProduktT {
    static inline T rezultat(T* a, T* b) {
        return *a * *b + DotProduct<T, N-1>::result(a+1,b+1);
    }
};

// parcijalna specijalizacija kao kraj rekurzije
template<typename T>
struct SkalProduktT<T, 0> {
    static inline T rezultat(T*, T*) {
        return T{};
    }
};

template<typename T, std::size_t N>
auto SkalProdukt(std::array<T, N> const& x,
                 std::array<T, N> const& y)
{
    return SkalProduktT<T, N>::rezultat(x.begin(), y.begin());
}
```

Ova nova implementacija se podvrgava radu predloška klase `SkalProduktT`. Predlošci nam omogućuju korištenje rekurzivnih instaciranja predloška klase sa parcijalnom specijalizacijom koja je zaslužna za kraj rekurzije. Primijetimo kako svaka instanca predloška `SkalProduktT` eksplicitno stvara sumu jednog člana skalarnog umnoška i implicitno sume skalarnog umnoška ostatka polja. Da ovo uistinu bude efikasno ključno je da prevodilac ne poziva funkciju već ekspandira tijelo funkcije na mjestu poziva. Napomenimo još da ovdje korištenje ključne riječi `inline`, sa stajališta samog standarda, ne igra nikakvu ulogu s obzirom da se metode definirane unutar strukture ili klase automatski tretiraju kao da su označene sa `inline`. Razlog leži u tome da eksplicitna instrukcija katkad forsira određene prevodioce (posebno Clang) da ekspandiraju tijelo metode na mjestu poziva. Da eksplicitno stavljanje ključne riječi `inline` ne potiče dodatno prevodilac za ekspanziju (moderni prevodioci većinom samostalno ekspandiraju tijelo kao dio optimizacije), tada općenito `static inline` funkcije (pa tako i metode) ne bi imale smisla. Naime, ključna riječ

`inline`, uz nagovještavanje ekspanzije tijela funkcije, služi i kao sredstvo za dopuštenje višestrukog ponavljanja definicije funkcije što bi inače kršilo tzv. pravilo jedne definicije (eng. *One Definition Rule*). S obzirom da ključna riječ `static` prisili funkciju na unutarnje vezivanje (eng. *internal linkage*), svaka se "instanca" funkcije u tranlacijskoj jedinici tretira kao posebna funkcija sa različitim adresama, ključna riječ `inline` postaje redundantna.

Sve u svemu, najbitnija stvar koju treba primjetiti da ovakav kod spaja računanje tijekom prevođenja sa efektom na asemblerski izlaz koji utječe na samo izvršavanje programa. Sve to opravdava ime hibridno metaprogramiranje. Kao što ćemo moći vidjeti u poglavlju 3, predložni izrazi također spadaju pod hibridno metaprogramiranje.

Poglavlje 2

Motivacija za predložne izraze

Tehnika je izvorno smišljena u svrhu implementacije operacija nad klasama numeričkih nizovima i u tom istom duhu ju motiviramo i ovdje. Klase numeričkih nizova omogućavaju numeričke operacije nad cijelim objektima. Na primjer, moguće je zbrojiti dva niza tako da rezultat sprema zbrojeve pojedinih elemanata nizova u odgovarajućem poretku. Slično, moguće je pomnožiti niz sa skalarom tako da se svaki element pomnoži sa tim skalarom. Poželjno je, a i prirodno, da se notacija poklapa sa klasičnom notacijom za zbrajanje i množenje ugrađenih tipova:

```
Array<double> x(1000), y(1000);
...
x = 1.2*x + x*y;
```

Za korisnike koji upravljaju sa velikim numeričkim nizovima, ključno je da se ovakvi izrazi evaluiraju što brže moguće. Međutim, postizanje tako nečega nije trivijalan zadatak. Srećom predložni izrazi rješavaju taj problem. Pokazat ćemo na koje probleme učinkovitosti nailazimo sa svojevrsnim izravnim pristupom rješavanju toga problema, a zatim ih i ilustrirati uspoređujući mjerenja sa petljom po komponentama.

2.1 Naivna implementacija

Krenimo sa naivnim pristupom. Prije svega, s obzirom da programski jezik C++ (a i ostali jezici) ne podržavaju samo jedan integralni tip odlučujemo se za korištenje predložaka. Na taj način ne moramo gomilati isti kod samo za različite tipove podataka. Nadalje, izlažemo samo bitnije dijelove naivne implementacije. Čitav kod se može pronaći pod [6].

```
template<typename T>
```

```
class SArray{ // SArray kao skraćeno od "Simple Array"
public:
    // kreiraj niz sa inicijalnom veličinom
    explicit SArray (std::size_t s)
        : storage(new T[s]), storage_size(s)
    {
        init();
    }

    // operator pridruživanja
    SArray<T>& operator= (SArray<T> const& orig) {
        if (&orig == this) // x = x slucajevi
           return *this;

        for (std::size_t idx = 0; idx<size(); ++idx)
            storage[idx] = orig.storage[idx];

        return *this;
    }

    // operator indeksiranja
    T const& operator[] (std::size_t idx) const {
        return storage[idx];
    }

private:
    T* storage;
};

// zbrajanje vektora po točkama
template<typename T>
SArray<T> operator+ (SArray<T> const& a, SArray<T> const& b)
{
    SArray<T> result(a.size());
    for (std::size_t k = 0; k<a.size(); ++k)
        result[k] = a[k]+b[k];

    return result;
}

// mnozenje vektora po točkama
template<typename T>
SArray<T> operator* (SArray<T> const& a, SArray<T> const& b)
{
    SArray<T> result(a.size());
    for (std::size_t k = 0; k<a.size(); ++k)
        result[k] = a[k]*b[k];
```

```
    return result;
}

// množenje skalara sa vektorom
template<typename T>
SArray<T> operator* (T const& s, SArray<T> const& a)
{
    SArray<T> result(a.size());
    for (std::size_t k = 0; k<a.size(); ++k)
        result[k] = s*a[k];

    return result;
}
```

2.2 Problemi

U implementaciji nismo koristili ništa novo niti neobično pa ju nema potrebe posebno objašnjavati. Mogli bismo puno više verzija i operatora napisati, ali ovo nam je bilo dovoljno da ukažemo na stvarne probleme sa ovakvim pristupom. Promotrimo što se događa sa primjerom danim na početku poglavlja.

```
    Array<double> x(1000), y(1000);
    ...
    x = 1.2*x + x*y;
```

Ispostavlja se da se sa danom implementacijom stvaraju dva velika problema:

1. Svaka upotreba operatora (osim operatora pridruživanja) stvara najmanje jednu kopiju niza. Dakle u našem primjeru tijekom izvršavanja stvaramo tri kopije vektora od 1000 elemenata. Stvaranje privremenih objekata često dominira vremenom potrebnom za operacije s malim nizovima osim ako se ne koriste brzi alokatori. Za doista velike nizove, privremeni objekti neće biti samo neprihvatljivi zbog učinkovitosti već i zbog potencijalnog manjka memorije da ih spremimo.

2. Svaka upotreba operatora zahtijeva dodatna čitanja odnosno zapisivanja elemenata novonastalih nizova. U gornjem primjeru se to svede na približno 6000 čitanja i 4000 pisanja unutar memorije ukoliko pretpostavimo da će prevodilac maksimalno optimizirati nepotrebna kopiranja. Naime, prije bismo mogli strahovati od dodatnog kopiranja prilikom vraćanja funkcija, odnosno operatora, po vrijednosti pa su tadašnji popularni prevoditelji počeli to optimizirati. Srećom, dolaskom standarda

C++17, jezik osigurava gotovo svaku moguću eliziju kopiranja. Pretpostavimo li zato maksimalnu eliziju, prevoditelj rastavlja naš izraz tako da stvara slijed petlji koje djeluju na privremenim objektima:

```
temp1 = 1.2*x; //petlja od 1000 operacija čitanja i pisanja
//uz stvaranje i uništavanje privremenog objekta

temp2 = x*y //petlja od 2000 čitanja i 1000 pisanja
//uz stvaranje i unistavanje privremenog objekta

temp3 = temp1 + temp2;//petlja od 2000 čitanja i 1000 pisanja
//uz stvaranje i uništavanje privremenog objekta

x = temp3; //petlja od 1000 operacija čitanja i pisanja
```

Ranije implementacije biblioteka numeričkih nizova susrele su se sa ovim problemom i poticali korisnike na korištenje operacija složenih pridruživanja. Glavna prednost takvih operacija pridruživanja je što se i argument i destinacija eksplicitno zadaju od strane korisnika pa nema potrebe za stvaranjem dodatnih objekata.

```
// zbrajanje i pridruživanje složenim operatorom pridruživanja
template<typename T>
SArray<T>& SArray<T>::operator+= (SArray<T> const& b)
{
    for (std::size_t k = 0; k<size(); ++k)
        (*this)[k] += b[k];

    return *this;
}

// množenje i pridruživanje složenim operatorom pridruživanja
template<typename T>
SArray<T>& SArray<T>::operator*= (SArray<T> const& b)
{
    for (std::size_t k = 0; k<size(); ++k)
        (*this)[k] *= b[k];

    return *this;
}

// množenje skalarom i pridruživanje
// složenim operatorom pridruživanja
template<typename T>
SArray<T>& SArray<T>::operator*= (T const& s)
```

```
    {
        for (std::size_t k = 0; k<size(); ++k)
            (*this)[k] *= s;

        return *this;
    }
```

Koristeći dane operatore izraz `x = 1.2*x + x*y` sada možemo preoblikovati kao:

```
SArray<double> x(1000), y(1000);
...
SArray<double> tmp(x);
tmp *= y;
x *= 1.2;
x += tmp;
```

Ovaj pristup se nije pokazao uspješnim iz sljedećih razloga.

- Notacija postaje nespretna, osobito za duže izraze.
- Još uvijek moramo stvoriti jedan privremeni objekt.
- Još uvijek imamo sveukupno 6000 operacija čitanja i 4000 operacija pisanja u memoriji.

Sve u svemu, ovakav pristup žrtvuje notaciju na račun efikasnosti. Za male nizove bi se tako nešto i isplatilo, ali to je maksimalna dobit. Više ne moramo stvarati onoliko privremenih objekata koliko imamo i operacija već nam je jedan dodatni objekt dovoljan. Ispostavlja se da možemo puno bolje od toga. Ideja je razviti jednu "idealnu" petlju koja obrađuje čitav izraz za svaki pojedini indeks, kao da ručno idemo mijenjati svaki pojedini element u petlji. Takvu petlju nazivamo petljom po komponentama. To bismo naravno mogli obavljati i samostalno unutar funkcije `main()`, ali sjetimo se da je naš cilj između ostaloga i koristiti prirodnu notaciju.

```
SArray<double> x(1000), y(1000);
...

// "idealna" petlja koju želimo razviti
for (int idx = 0; idx<x.size(); ++idx)
    x[idx] = 1.2*x[idx] + x[idx]*y[idx];
```

Sada više nema potrebe za privremenim nizovima, a i imamo minimalan broj čitanja i pisanja u memoriji. Svega 2000 operacija čitanja i 1000 operacija pisanja. To nam upravo omogućava tehnika predložni izrazi.

2.3 Rezultati mjerenja

Za kraj ovog poglavlja uspoređujemo brzine izvođenja. S jedne strane imamo kod s petljom po komponentama, a s druge strane imamo našu naivnu implementaciju. Pokazat ćemo koliko naša implementacija zaostaje u ovisnosti o veličini polja i dužini izraza.

Kao sredstvo mjerenja koristimo odgovarajuće klase i metode iz biblioteke `chrono`. Za spremanje fiksnog vremenskog trenutka koristimo klasu `std::chrono::time_point`, a kao sat uzimamo `std::chrono::steady_clock`. Sat `steady_clock` reprezentira monotoni sat odnosno njegovi vremenski trenuci ne mogu se opadati kako stvarno vrijeme ističe i vrijeme između otkucaja ovog sata je konstantno. Zato je `steady_clock` pogodan za mjerenje vremena između dva trenutka; upravo ono što nama treba. Statička funkcija `std::chrono::steady_clock::now()` vraća trenutni broj otkucaja. Da bismo iz razlike broja otkucaja uzetih iz dva različita vremenska trenutka dobili stvarno proteklo vrijeme, tu razliku šaljemo klasi `std::chrono::duration`. Preciznije, stvaramo objekt tog tipa kojemu preko parametara predloška šaljemo integralni tip i mjernu jedinicu, a kao argument razliku broja otkucaja. Na kraju pozivamo funkciju `.count()` za dobivanje stvarnog vremena.

```
class Clock {
public:
    Clock() { start(); }
    void start();
    double stop();

private:
    std::chrono::steady_clock mClock;
    std::chrono::time_point<std::chrono::steady_clock> begin;
    std::chrono::time_point<std::chrono::steady_clock> end;
};

void Clock::start() {
    begin = mClock.now();
    return;
}

double Clock::stop() {
    end = mClock.now();
    std::chrono::duration<double, std::micro> mic = end - begin;
    return mic.count();
}
```

Mjerenja su podijeljena u tri skupine. Prvo smo uspoređivali rezultate mjerenja za

nizove od 1000 elemenata, zatim od 10 000 elemenata i na kraju od 100 000 elemenata. Za svaku skupinu smo 100 puta vršili izraz `x = 1.2*x + x*y` (odnosno odgovarajući pristup preko petlje) i sumirali vremena za svako izvršavanje. Na kraju dijelimo sa 100 da dobijemo prosječno vrijeme. Naredni rezultati mogu varirati u ovisnosti o mnogo parametara kao što su broj procesora i dretvi, operacijski sustav, prevoditelj, postavljene optimizacije prevoditelja itd. Ono što bi trebalo biti na svim platformama slično (uz iste optimizacije prevoditelja) su razlike u mjerenjima; kod s petljom po komponentama bi uvijek trebao biti brži, pogotovo za veće nizove. Analogno radimo i za izraz `1.2*x*(x+y+z) + 2.3*y*(x+y+z) + 3.4*z*(x+y+z)`. Za naša mjerenja koristili smo GNU g++ prevoditelj uz -O3 optimizaciju na operacijskom sustavu Windows 10 i Intel-ovom i5 procesoru 10. generacije sa 16 GB RAM-a. Optimizacija -O3 će se pokazati izuzetno bitnom u sljedećim poglavljima.

Tablica 2.1: Rezultati mjerenja za izračunavanje izraza `x=1.2*x+x*y`

Implementacija		
Veličina niza	Kod s petljom po komponentama	Naivna implementacija
1000	0 µs	0 µs
10 000	19.93 µs	179.53 µs
100 000	90 µs	768.19 µs

Tablica 2.2: Mjerenja za `1.2*x*(x+y+z) + 2.3*y*(x+y+z) + 3.4*z*(x+y+z)`

Implementacija		
Veličina niza	Kod s petljom po komponentama	Naivna implementacija
1000	0 µs	49.86 µs
10 000	19.69 µs	758.21 µs
100 000	189.57 µs	3640.48 µs

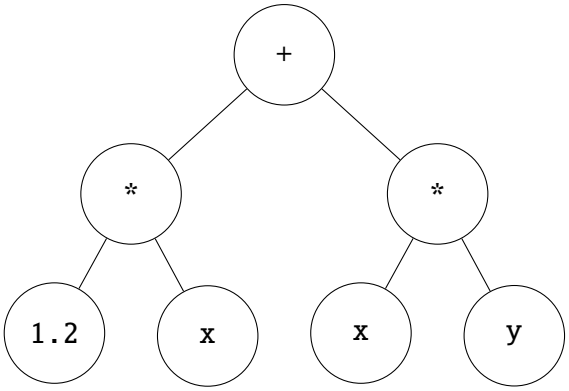
Poglavlje 3

Predložni izrazi u implementaciji vektora

Naš cilj je implementirati odgovarajuće operatore tako da njihova učinkovitost tijekom izvođenja bude približno identična učinkovitosti dobivenoj petljom po komponentama. Ključni problem u naivnoj implementaciji vektora bio je postupno izračunavanje izraza čime ne možemo izbjeći stvaranje privremenih objekata. Nekakva početna ideja je da prevoditelj tijekom čitanja izraza prvo obiđe čitav izraz prije obavljanja bilo kakvih naredbi izvođenja. Dakle lijena evaluacija. Tijekom čitanja prevodilac također mora i na neki način spremiti izraz kako bi ga mogao kasnije evaluirati. Stoga se prirodno nameće instaciranje predložaka klase svakim nailaskom na novu operaciju i na taj način spremiti odnosno iskodirati izraz u jednu instancu klase, točnije objekt te instance. Napomenimo da u narednom tekstu ne radimo razliku između pojma klase i pojma predloška klase radi jednostavnosti.

Pokušajmo ideju približiti na našem izrazu `x = 1.2*x + x*y`. Ovdje `1.2*x` ne znači stvaranje novog niza već objekta koji reprezentira množenje svakog elementa vektora `x` sa `1.2`. Taj objekt zato mora spremiti vektor `x`, a da izbjegnemo stvaranje privremenog objekta, to obavljamo korištenjem reference. Skalare ćemo ipak morati kopirati zbog razloga koji objašnjavamo tek kasnije, ali srećom te kopije ne utječu na performanse jer takvih kopiranja ima maksimalno jednako broju operacija u izrazu, a i objekti će biti mali veličinom. Uostalom, uz odgovarajuće optimizacije prevodilac će eliminarati stvaranje bilo kakvih objekata tijekom evaluacije. Isto tako, `x*y` instancira novi objekt koji reprezentira množenje svakog elementa vektora `x` sa odgovarajućim elementom vektora `y`. Konačno, oba ta novostvorena objekta spremamo u novi objekt koji reprezentira zbrajanje odgovarajućih rezultata koje ti objekti predstavljaju.

Sve to možemo ilustrirati u obliku binarnog stabla. Slijedeći gornje objašnjenje, naše binarno stablo izraza izgleda ovako.



Slika 3.1: Binarno stablo izraza `x = 1.2*x + x*y`

To nagovještava instaciranje klase sljedećeg oblika:

```
Add< Mult< Scalar, Vector >, Mult< Vector, Vector > >
```

Za to nam trebaju reprezentacije operanada - skalara `Scalar` i vektora `Vector` te reprezentacije operacija - zbrajanja `Add<Op1,Op2>` i množenja `Mult<Op1,Op2>`.

3.1 Operandi

U nastavku koristimo predložak klase `SVector` koji nije ništa drugo nego predložak `SArray` samo bez odgovorajućih aritmetičkih operatora i operatora pridruživanja. Klasa `SVector` služi samo kao sredstvo spremanja niza elemenata, dok je predložak klase `Vector`, s kojim se upoznajemo u ovom poglavlju, zaslužan sa izgradnju predložnih izraza. Dodavanjem dodatne klase `Vector` omogućavamo nesmetano spremanje bilo kakvih instanci predložaka (odnosno stabala izraza) koje reprezentiraju aritmetičke operacije nad vektorima, a ne samo sirove nizove, što nam je od osobite važnosti pri definiranju aritmetičkih operatora. Točnije, objekt klase `Vector` se stvara svakom prilikom kroz `return` naredbu u aritmetičkim operatorima gdje upravo spremamo dosadašnje stablo računanja. Dodatno, tom istom klasom služi se i korisnik dok se klasa `SVector` stvara samo pri inicijalizaciji vektora kojeg je stvorio korisnik i u potpunom je vlasništvu klase `Vector`.

Krećemo sa implementacijom predloška klase `Scalar`. Samu motivaciju zašto stvaramo posebnu klasu za skalar i pojedine detalje ne možemo još uvijek objasniti, stoga ju zasad uzimamo zdravo za gotovo.

```
    // reprezentant skalara
    template<typename T>
    class Scalar {
    private:
        const T& s; // vrijednost skalara
    public:

        // konstruktor
        Scalar (T const& v) : s(v) {}

        // operator potreban prilikom evaluacije izraza
        T const& operator[] (std::size_t) const {
            return s;
        }

        // uzimamo da skalari imaju veličinu 0
        std::size_t size() const {
            return 0;
        };
};
```

Nastavljamo sa implementacijom predloška klase `Mult`. Iz izgleda finalnog predloška koji kodira naš izraz (3) možemo vidjeti da svaka instanca te klase predstavlja množenje nekakvog izraza čiji tip označava prvi parametar predloška sa izrazom čiji tip označava drugi parametar predloška. Ti tipovi mogu predstavljati vektore, skalare, zbroj vektora i skalara, umnožak dvaju vektora itd.

```
1   #include "traits.h"
2
3   template<typename T, typename OP1, typename OP2>
4   class Mult {
5   private:
6       typename ExprRef<OP1> op1; // prvi operand
7       typename ExprRef<OP2> op2; // drugi operand
8   public:
9       Mult(OP1 const& a, OP2 const& b)  : op1(a), op2(b) { }
10
11      // služi za evaluaciju izraza kojeg drži
12      T operator[] (std::size_t idx) const {
13          return op1[idx] * op2[idx];
14      }
15
16      // maksimalna veličina
17      std::size_t size() const {
18          return op1.size()!=0 ? op1.size() : op2.size();
```

```
19          }
20      };
```

Parametri predloška označavaju redom tip elementa kojeg vektor sprema, tip prvog faktora i tip drugog faktora. Za početak objasnimo prvo na koji način spremamo operande odnosno što točno rade linije 3 i 4. Kako smo već obznanili ranije, prije bilo kakve evaluacije prevoditelj prvo prolazi čitavim izrazom i instancira odgovarajuće predloške. Zamislimo da unutar aritmetičkog izraza imamo nekakvo množenje skalarom, npr. `...1.2*x...`. Na tom mjestu, prevoditelj neće narediti nikakvo množenje već će instancirati klasu oblika `Mult<double, Scalar<double>, SArray<double> >` (pretpostavljamo da će korisnik paziti na tipove). Kao što ćemo moći vidjeti u nastavku, objekt tog tipa će se stvoriti argumentima `Scalar<double>(1.2)` i vektorom tipa `SArray<double>` unutar operatora množenja. Cilj je naravno što više novostvorenih objekata vezati referencom da izbjegnemo nepotrebna kopiranja, ali u ovom ćemo se slučaju ipak morati zadovoljiti kopiranjem. Naime, kako operator množenja neće vraćati objekt `Scalar<double>(1.2)` (što naravno ne bi imalo nikakvog smisla), taj objekt umire završetkom djelovanja operatora. Preciznije, objekt umire nakon naredbe `return`. Ukoliko bi klasa `Mult` vezala referencu na taj objekt, lako vidimo da sve pada u vodu jer bi referenca pokazivala na nedozvoljeni dio memorije nakon završetka operatora množenja. U ostalim slučajevima, tj. kada nemamo množenje skalarom, referenca prolazi. Razlog dajemo kasnije. Bilo kako bilo, u spas pristižu klase obilježja (eng. *traits classes*).

Klase obilježja donose informacije o tipovima. Ovisno o kontekstu to može značiti svašta, ali obično klase obilježja koristimo kada ne znamo unaprijed koji nam tip odgovara zbog parametrizacije funkcije ili klase. S takvim klasama barata prevoditelj i u ovisnosti o parametriziranom tipu, vraća tip koji nam treba. Ovdje konkretno želimo da nam `ExprRef<OP>` iz linija 3 i 4 vrati konstantnu referencu ukoliko `OP` nije skalar odnosno sirovi tip ukoliko se radi o skalaru. Tako nešto nije teško za postići.

```
template <typename T>
class Scalar;

template <typename T>
class Traits{
public:
    using ExprRef = const T&;
};

template <typename T>
class Traits<Scalar<T> >{
public:
    using ExprRef = Scalar<T>;
```

```
};

template <typename T>
using ExprRef = typename Traits<T>::ExprRef;
```

Jednostavno parcijalnom specijalizacijom definiramo specijalni slučaj u kojem vraćamo sirovi tip, dok za sve ostale slučajeve vraćamo konstantu referencu. Za naš slučaj nakon instanciranja predloška klase `Mult` za podizraz `1.2*x` linije 3 i 4 će izgledati:

```
    ...
    Scalar<double> op1;
    const SVector<double>& op2;
    ...
```

Klase obilježja velikim dijelom podsjećaju na metaprogramiranje tipova jer sve obavljamo tijekom kompilacije, a kao rezultat vraćamo tip. Jedina razlika ovdje je što nemamo nekakav oblik računanja već, neovisno o tipu, imamo samo jedno instanciranje predloška `Traits`.

Operator indeksiranja služi za evaluaciju izraza koja se izvršava tek pri pozivu operatora pridruživanja, dakle "kad se nalazimo na kraju izraza". Metoda `size()` nije od osobite važnosti. Služi prilikom konstrukcije vektora na kraju svake aritmetičke operacije pa vektori koji reprezentiraju stablo računanja imaju i to dodatno svojstvo koje se zapravo nigdje ne koristi. Jedina zapravo korist od te metode bila je prilikom ispravljanja grešaka (eng. *debugging*) aritmetičkih operacija. Njena netrivijalnost slijedi iz potencijalne egzistencije skalara u računu jer za veličinu skalara uzimamo vrijednost 0. Stoga unutar naredbe `return` provjeravamo imamo li skalar kao prvog operanda.

Predložak klase `Add` se obavlja na sasvim analogan način kao i za `Mult` stoga ju posebno ne objašnjavamo. Jedina razlika ovdje je što kod stvaranja objekta tipa `Add` nemamo slučaj kada je lijevi ili desni operand skalar pa nema potrebe za korištenjem klase obilježja.

```
#include "traits.h"

template<typename T, typename OP1, typename OP2>
class Add {
private:
    const OP1& op1; // prvi operand
    const OP2& op2; // drugi operand
public:
    Add(OP1 const& a, OP2 const& b)  : op1(a), op2(b) { }

    // služi za evaluaciju izraza kojeg drži
    T operator[] (std::size_t idx) const {
```

```
            return op1[idx] + op2[idx];
        }

        // maksimalna veličina
        std::size_t size() const {
            return op1.size()!=0 ? op1.size() : op2.size();
        }
    };
```

3.2 Klasa Vector

Rekli smo već da klasa `Vector` služi kao spremište bilo sirovog vektora bilo nekakvog stabla ili podstabla koje reprezentira računanje. Nju koristi i sam korisnik, ali i aritmetički operatori. Također, ona služi i kao početak evaluacije izraza odnosno njen operator pridruživanja "započinje" računanje. Dajemo pojednostavljeni oblik klase tj. samo one dijelove potrebne za funkcioniranje predložnih izraza. Izostavljeni dijelovi služe samo kao dodatna svojstva npr. inicijaliziranje preko inicijalizacijske liste, ispis vektora itd. Potpuna verzija se može pronaći pod [2]. Deklaracija glasi:

```
template<typename T, typename Rep = SVector<T>>
class Vector;
```

Tip Rep predstavlja tip stvarnog vektora ili ugniježđene instance predložaka koji kodiraju izraz. Kako drugi parametar predloška nije bitan korisniku, taj tip ima dodijeljenu vrijednost (tip) `SVector<T>` i ta vrijednost se koristi samo kada korisnik stvara novi vektor.

```
template<typename T, typename Rep = SVector<T>>
class Vector{
public:
    // konstrukcija vektora zadanom veličinom
    explicit Vector (std::size_t s) :
        expr_rep(s), mSize(s) { }

     // konstrukcija vektora izrazom
    Vector (Rep const& rb) :
        expr_rep(rb), mSize(rb.size()) { }

    // operator pridruživanja istim tipom
    Vector& operator= (Vector const& b) {
        for (std::size_t idx = 0; idx < mSize; ++idx) {
            expr_rep[idx] = b[idx];
        }
```

```
        return *this;
    }

    // operator pridruživanja za različite tipove
    template<typename T2, typename Rep2>
    Vector& operator= (Vector<T2, Rep2> const& b) {
        for (std::size_t idx = 0; idx < mSize; ++idx) {
            expr_rep[idx] = b[idx];
        }
        return *this;
    }

    // veličina vektora
    std::size_t size() const {
        return mSize;
    }

    // operatori indeksiranja
    decltype(auto) operator[] (std::size_t idx) const {
        return expr_rep[idx];
    }

    T& operator[] (std::size_t idx) {
        return expr_rep[idx];
    }

    // vraćanje izraza kojeg vektor predstavlja
    Rep const& rep() const {
        return expr_rep;
    }

    Rep& rep() {
        return expr_rep;
    }
private:
    Rep expr_rep; // izraz kojeg čuva Vector
    std::size_t mSize; // veličina vektora
};
```

Klasa `Vector` u službi reprezentacije izraza stvara se isključivo unutar aritmetičkih operatora tako da za potpuno shvaćanje koda potrebo je pogledati definicije i operatora koje izlažemo kasnije. Ukratko objasnimo operator pridruživanja. Potpuni osvrt na to dajemo u odjeljku 3.4. gdje na izrazu `x = 1.2*x + x*y` koncizno opisujemo što se događa. Nakon što prevodilac prođe čitavim izrazom, instacirali su se svi potrebni predlošci za reprezentiranje izraza. Na kraju, kada prevodilac nailazi na operator pridruživanja, vidimo korištenje samo jedne petlje. Ta se petlja poslije kompilacije, nakon niza ekspanzija

`inline` metoda (operatora indeksiranja) krečući od podizraza `b[idx]`, pretvara u oblik koji parira učinkovitosti koju dobijemo ručno dobivenoj petlji po komponentatama. U to se uvjeravamo u odjeljku 3.5.

Nadalje, primjetimo da imamo dvije verzije operatora pridruživanja. Prva verzija obavlja pridruživanje vektore istog tipa dok druga verzija obavlja pridruživanje ukoliko vektore na desnoj strani reprezentira stablo. Uočimo da nismo mogli samo napisati parametriziranu verziju operatora jer bi tada u slučaju pridruživanja vektora istog tipa prevoditelj sam sintetizirao operator pridruživanja za iste tipove što nama očito ne odgovara.

Sada tek možemo vidjeti potrebu za stvaranjem posebne klase za spremanje skalara. Naime, za izraz recimo `x = 1.2*x`, objekt `b` unutar operatora pridruživanja će biti tipa `Vector<Mult<Scalar<double>, SVector<double> > >`. Izraz `b[idx]` poziva operator indeksiranja instance `Mult`. Kako pak taj operator vraća `op1[idx] * op2[idx]`, gdje su op1 i op2 redom tipa `Scalar<double>` i `SVector<double>`, vidimo da oba operanda moraju imati definiran vlastiti operator indeksiranja. Ugrađeni tipovi nemaju definiran operator indeksiranja pa smo se morali odlučili skalar reprezentirati vlastitom klasom. Alternativno smo mogli napraviti poseban predložak za umnožak vektora i skalara.

Još treba objasniti konstantu verziju operatora indeksiranja. Primjetimo da operator koristi deducirani povratni tip za razliku od tradicijalnog `const T&`. To radimo jer u slučaju da `Rep` reprezentira množenje odnosno `Mult` ili zbrajanje odnosno `Add`, njihovi operatori indeksiranja vraćaju izraze koji imaju desnu vrijednost pa na njih ne možemo vezati reference. Operator `decltype` na izrazu koji ima desnu vrijednost vraća tip bez reference. Ako pak `Rep` reprezentira sirovo polje odnosno `SVector`, tada njegov operator indeksiranja vraća konstantnu referencu (pogledati 2.1). Operator `decltype` na izrazu koji ima lijevu vrijednost vraća lijevu referencu (eng. *lvalue reference*).

3.3 Operatori

Da upotpunimo priču o predložnim izrazima preostaje nam samo pokazati kako bi izgledala implementacija operatora koji operiraju predložnim izrazima. Već smo nagovijestili da se svakom aritmetičkom operacijom instancira novi predložak koji predstavlja stablo dosadašnjeg izračunavanja. Novostvorenu instancu moramo spremiti u `Vector` kako bismo mogli nastaviti sa izračunavanjem ili završiti priču i pozvati operator pridruživanja. Ponavljamo, niti jedan od operatora ne vrši nikakvu evaluaciju već samo nadograđuje postojeće stablo izraza.

Moramo implementirati tri verzije operatora: zbroj dvaju vektora, umnožak dvaju vektora i umnožak vektora skalarom.

```
// zbroj dvaju vektora
```

```
template <typename T, typename R1, typename R2>
Vector<T, Add<T, R1, R2> >
operator+ (const Vector<T, R1>& a, const Vector<T, R2>& b) {
    return Vector<T, Add<T, R1, R2> >
        (Add<T, R1, R2>(a.rep(), b.rep()));
}

// umnožak dvaju vektora
template <typename T, typename R1, typename R2>
Vector<T, Mult<T, R1, R2> > operator*
(const Vector<T, R1>& a, const Vector<T, R2>& b) {
    return Vector<T, Mult<T, R1, R2> >
        (Mult<T, R1, R2>(a.rep(), b.rep()));
}

// umnožak vektora skalarom
template <typename T, typename R2>
Vector<T, Mult<T, Scalar<T>, R2> >
operator*(const T& s, const Vector<T, R2>& b) {
    return Vector<T, Mult<T, Scalar<T>, R2> >
        (Mult<T, Scalar<T>, R2>(Scalar<T>(s), b.rep()));
}
```

Sada tek uistinu možemo vidjeti potrebu za stvaranjem klase obilježja u klasi `Mult`. Pozivi metode `rep()` pri stvaranju privremenih objekata tipa `Add` i `Mult` vraćaju reprezentante čiji vijek trajanja odgovara vijeku trajanja čitavog izraza. Razlog tomu je što svi ovi parametri u aritmetičkim operatorima su ili vektori koje je stvorio korisnik ili vektori koje vraćaju baš ti isti operatori. Neovisno o tome dolaze li od korisnika ili kroz naredbu `return`, ti vektori žive do kraja izraza pa tako i njihove enkapsulirane varijable. Zato slobodno možemo vezati referencu na povratni objekt dobiven pozivom metode `rep()`.

Deklaracije ovih operatora su poprilično nezgrapne kao što možemo vidjeti, ali njihova funkcionalnost nije komplicirana za shvatiti. Na primjer, operator zbrajanja prvo stvara objekt tipa `Add` koji reprezentira i operator i operande, a zatim zamotava taj objekt u objekt tipa `Vector` kojeg potom može preuzeti drugi aritmetički operator ili operator pridruživanja ako se nalazimo na kraju izraza.

3.4 Osvrt

Prvi susret sa idejom iza tehnike predložni izrazi može biti obeshrabljujuć. Zato za kraj objašnjavanja dajemo potpuni osvrt što se događa sa konkretnim izrazom odnosno kako će naš prevoditelj prevesti taj izraz prevesti. Pretpostavimo da imamo sljedeće:

```
int main() {
    Vector<double> x(1000), y(1000);
    ...
    x = 1.2*x + x*y;
}
```

Pošto je argument predloška izostavljen iz definicija vektora `x` i `y`, parametar predloška se postavlja na dodijljenu vrijednost odnosno `SVector<double>`. Prilikom parsiranja izraza `1.2*x + x*y` prevoditelj najprije obavlja najljeviju operarciju množenja, što je ovdje umnožak sa skalarom. Razrješivanje preopterećenja zato izabire skalar-niz oblik množenja:

```
template <typename T, typename R2>
Vector<T, Mult<T, Scalar<T>, R2> >
operator*(const T& s, const Vector<T, R2>& b) {
    return Vector<T, Mult<T, Scalar<T>, R2> >
        (Mult<T, Scalar<T>, R2>(Scalar<T>(s), b.rep()));
}
```

Tipovi operanada su redom `double` i `Vector<double, SVector<double> >`. Povratni tip operatora je

```
Vector<double, Mult<double, Scalar<double>, SVector<double>>>
```

Rezultat je objekt tipa `Vector` čiji unutrašnji objekt tipa `Mult` nagovještava množenje svakog elementa vektora `x` skalarom `1.2`.

Potom slijedi druga operacija množenja; umnožak dvaju vektora `x*y`. Ovog puta koristimo niz-niz oblik množenja:

```
template <typename T, typename R1, typename R2>
Vector<T, Mult<T, R1, R2> > operator*
(const Vector<T, R1>& a, const Vector<T, R2>& b) {
    return Vector<T, Mult<T, R1, R2> >
        (Mult<T, R1, R2>(a.rep(), b.rep()));
}
```

Oba tipa operanda su `Vector<double, SVector<double> >` pa je povratni tip

```
Vector<double, Mult<double, SVector<double>, SVector<double>>>
```

Ovog puta zamotani `Mult` objekt nagovještava množenje dvaju vektora `x` i `y`.

Na kraju se izvršava operacija zbrajanja. Zbrajaju se dva vektora, a tipovi operanada su upravo ovi deducirani tipovi maloprije. Pozivamo operator

```
    template <typename T, typename R1, typename R2>
    Vector<T, Add<T, R1, R2> >
    operator+ (const Vector<T, R1>& a, const Vector<T, R2>& b) {
        return Vector<T, Add<T, R1, R2> >
            (Add<T, R1, R2>(a.rep(), b.rep()));
    }
```

T se zamjenjuje sa tipom double dok se tip R1 zamjenjuje sa

```
Mult<double, Scalar<double>, SVector<double>>
```

a tip R2

```
Mult<double, SVector<double>, SVector<double>>
```

što odgovara tipovima operanada. Stoga povratni tip završne operacija biva

```
    Vector<double,
    Add<double,
    Mult<double, Scalar<double>, SVector<double>>,
    Mult<double, SVector<double>, SVector<double>>>>
```

Tim tipom instanciramo operator pridruživanja klase Vector

```
template<typename T, typename Rep = SArray<T>>
    class Array {
    public:
        ...
        // operator pridruzivanja za razlicite tipove
        template<typename T2, typename Rep2>
        Vector& operator= (const Vector<T2, Rep2>& b) {
            for (std::size_t idx = 0; idx<b.size(); ++idx) {
                expr_rep[idx] = b[idx];
            }
            return *this;
        }
        ...
};
```

gdje je parametar b konstanta referenca na objekt kojeg vraća operator zbrajanja. Pažljivim praćenjem operatora indeksiranja ispostavlja se da za dani index idx računamo

```
(1.2*x[idx]) + (x[idx]*y[idx])
```

što je upravo ono što i želimo. Također primjećujemo da prilikom evaluacije aritmetičkog izraza mnogo malih `inline` funkcija pozivaju jedna drugu i da se mnogo malih objekata alociraju na stogu. Optimizatori moraju izvršiti potpunu ekspanziju `inline` funkcija i eliminaciju malih objekata da bi proizveli kod čija učinkovitost nije ništa lošija od učinkovitosti dobivenoj petljom po komponentatama. Kao što smo nagovijestili na kraju odjeljka 2.2, predložnim izrazima (uz pretpostavljene optimizacije prevoditelja) izbjegavamo stvaranje bilo kakvih privremenih objekata te završavamo minimalnim brojem čitanja i zapisivanja memorijskih lokacija (konkretno za izraz `x = 1.2*x + x*y` sada imamo 2000 čitanja i 1000 zapisivanja). Današnji moderni prevoditelji uglavnom nemaju problema sa takvim optimizacijama pa se tehnika pokazuje izuzetno korisnom.

3.5 Verifikacija i performanse

Za kraj moramo se uvjeriti da tehnika zaista radi i dobivamo li željene performanse. Za verifikaciju ispravnosti smo nadopunili klasu `Vector` tako da se može inicijalizirati preko inicijalizacijske liste. Također smo implementirali operator izlaza radi kompaktnijeg ispisa vektora. Te nadopune nisu teške za napraviti niti su bitne za ovaj rad pa njihove detalje izostavljamo iz rasprave. Za početak krećemo od lakših izraza, a završavamo najkompliciranijim.

```
int main(){
    Vector<double> x = { -12, 32.2, 54, 4 };
    Vector<double> y = { 2.12, 0.21, -23.1, -1 };
    Vector<double> z = { 76.2, -32, 13.122, 90.1 };

    // spremista rezultata
    Vector<double> prvi(4), drugi(4), treci(4), cetvrti(4);

    ////// prvi izraz
    prvi = x;
    std::cout << "Rezultat:     " << prvi;
    std::cout << "WolframAlpha: "
              << "(-12, 32.2, 54, 4)" << std::endl;

    ////// drugi izraz
    drugi = 1.2 * x + x*y;
    std::cout << "Rezultat:     " << drugi;
    std::cout << "WolframAlpha: "
              << "(-39.84, 45.402, -1182.6, 0.8)" << std::endl;
```

```
    ///// treci izraz
    treci = x*y*x + (-2.1)*z + z*x*y;
    std::cout << "Rezultat:     " << treci;
    std::cout << "WolframAlpha: "
              << "(-1793.27, 68.5524, -83755.5, -565.61)"
              << std::endl;

    ///// cetvrti izraz
    cetvrti = 1.2*z*(x+y) + 2.3*y*(x+z) + 3.4*x*(y+z);
    std::cout << "Rezultat:     " << cetvrti;
    std::cout << "WolframAlpha: "
              << "(-3785.85, -4724.82, -4911.6, 1319.69)"
              << std::endl;

    return 0;
}
```

Slijedi ispis gornjeg koda:

```
Rezultat:     [-12, 32.2, 54, 4]
WolframAlpha: (-12, 32.2, 54, 4)
Rezultat:     [-39.84, 45.402, -1182.6, 0.8]
WolframAlpha: (-39.84, 45.402, -1182.6, 0.8)
Rezultat:     [-1793.27, 68.5524, -83755.5, -565.61]
WolframAlpha: (-1793.27, 68.5524, -83755.5, -565.61)
Rezultat:     [-3785.84, -4724.82, -4911.59, 1319.69]
WolframAlpha: (-3785.85, -4724.82, -4911.6, 1319.69)
```

Ovim ispisom razumno je pretpostaviti da kod radi ispravno. Još preostaje obaviti mjerenja performansi. Uspoređujemo kod s komponentama, naivnu implementaciju i predložne izraze. Mjerenja su rađena na isti način kao i u odjeljku 2.3.

Tablica 3.1: Rezultati mjerenja za izračunavanje izraza `x=1.2*x+x*y`

Implementacija			
Veličina niza	Komponente	Naivna implementacija	Predložni izrazi
1000	0 µs	0 µs	0 µs
10 000	9.97 µs	89.75 µs	10 µs
100 000	89.76 µs	767.95 µs	79.78 µs

Tablica 3.2: Mjerenja za `1.2*x*(x+y+z) + 2.3*y*(x+y+z) + 3.4*z*(x+y+z)`

Implementacija			
Veličina niza	Komponente	Naivna implementacija	Predložni izrazi
1000	0 µs	49.86 µs	0 µs
10 000	19.69 µs	349.07 µs	20.22 µs
100 000	189.57 µs	3052.09 µs	169.55 µs

Kao što možemo vidjeti, predložni izrazi osiguravaju istu učinkovitost kao i kod sa petljom po komponenentama.

Poglavlje 4

Predložni izrazi u implementaciji matrica

Predložni izrazi sami po sebi ne rješavaju sve problematične situacije koje nastaju prilikom numeričkih operacija nad nizovima. Na primjer, predložni izrazi ne funkcioniranju na umnošku matrica i vektora u obliku

```
x = A*x;
```

gdje je `x` vektor veličine n i `A` matrica dimenzija $n \times n$. Problem nastaje jer predložnim izrazima mijenjamo elemente vektora `x` jedan za drugim. Kako je, na primjer, prvi element vektora `x` u našem izrazu potreban za izračunavanje drugog elementa ne smijemo koristiti promijenjeni element za evaluaciju drugog jer inače dobivamo pogrešan rezultat. Stoga imamo potrebu za stvaranjem privremenog objekta. Isti problemi nastaju i za umnoške matrica. Za izraze oblika

```
x = A*y;
```

takvih problema nema ako se `x` i `y` ne odnose na isti objekt u memoriji. Rješenje zato mora na određeni način biti svjesno odnosa operanada tj. njihovih adresa. S obzirom da potencijalni problemi nastaju samo prilikom matričnog množenja naše rješenje će samo paziti na taj slučaj. Naime, ukoliko izraz ne uključuje matrično množenje naša evaluacija neće biti ništa drukčija od dosadašnje. Ukoliko pak izraz uključuje matrično množenje tada ćemo još dodatno provjeravati nalazi li se adresa operanda s lijeve strane pridruživanja među adresama operanada desne strane. Ako je odgovor potvrdan, tada stvaramo privremeni objekt kao dodatan trošak. Napomenimo da će izrazi poput

```
x = A*y + x;
```

zahtijevati stvaranje privremenog objekta bez stvarne potrebe za to. Za dodatna ograničenja za takve slučajeve, nagađamo da rješenje postaje mnogo kompliciranije (ako je uopće izvedivo) pa se odlučujemo ne baviti time. Napomenimo još i da u narednom tekstu pod matričnim množenjem ne podrazumijevamo umnožak skalara i matrice jer taj umnožak nije ništa posebniji od umnoška skalara i matrice.

Problemi se javljaju i kod izraza koji uključuju samo matrice. Prva stvar koju treba imati na umu je korištenje pametne petlje pri prolaženju matričnim elementima. Ideja je kretati se u unutrašnjoj petlji onako kako je i matrica spremljena u memoriji. Ako će se matrica spremiti u priručnoj memoriji (eng. *cache*) tada to uvelike utječe na performanse jer pametnom petljom sprječavamo promašaje (eng. *cache misses*). Naime, ako idemo implementirati operator indeksiranja u matrica predložnim izrazima tada dobivamo sljedeći kod:

```
template <typename T2, typename Rep2>
Matrix& operator= (const Matrix<T2, Rep2>& b) {
    for (std::size_t idx = 0; idx < mRows; ++idx) {
        for (std::size_t idy = 0; idy < mColumns; ++idy) {
            expr_rep(idx, idy) = b(idx, idy);
        }
    }

    return *this;
}
```

Problem nastaje ako referenca `b` referira na objekt koji reprezentira umnožak dviju matrica. Tada on u sebi mora sadržavati nekakvu petlju koja se kreće po recima matrice na lijevoj strani umnoška odnosno po stupcima matrice na desnoj strani umnoška.

```
for(int id = 0; id < N; ++id){
    sum += mat1(idx, id) * mat2(id, idy);
}
```

Pretpostavimo da smo matricu spremili po recima (eng. *row-major*) i pretpostavimo da matrica živi u priručnoj memoriji. Prilikom pristupa elementu `mat(idx, id)` priručna memorija može učitati čitav segment memorije koji uključuje niz susjednih elemenata iz istog retka. Na taj način dobivamo na brzini izvođenja u idućim iteracijama petlje. Međutim, pristupamo li elementu `mat2(id,idy)` tada je izgledno da nećemo učitati element `mat2(id+1,idy)` ako je matrica dovoljno velika, a onda naravno ni ostale iz istog stupca. Tako gubimo na potencijalno većoj brzini izvođenja. Srećom taj problem se lako

rješava drukčijim rasporedom pristupa elementima u petlji. S obzirom na to da operator indeksiranja nećemo mijenjati jer se tehnika isplati ako npr. imamo samo zbrajanje matrica, uvijek ćemo umnožak matrica obavljati na samom mjestu poziva operatora množenja i rezultat spremiti u privremeni objekt. Gubitak performansi na to će biti relativno slab jer se većina vremena ionako oduzima na samom množenju koje zahtijeva tri petlje.

Kako umnožak obavljamo na licu mjesta i spremamo ga u privremeni objekt više ne moramo obraćati pozornost na dodatno stvaranje privremenog objekta kao što to trebamo raditi kod pridruživanja vektora. Naime, ako se matrica nalazi na lijevoj strani pridruživanja tada znamo da se na desnoj strani priduživanja u izrazu pojavljuju samo matrice. Jedine operacije koje implementiramo će biti zbroj i umnožak matrica te umnožak skalara i matrice. S obzirom da zbroj matrica i umnožak skalarom neće biti kritičani, a umnožak uvijek stvara novi objekt slobodno možemo pridružiti elemente bez dodatnih provjera.

Drugi problem kod matrica se također javlja pri njihovom umnošku. Promotrimo sljedeći izraz:

```
Matrix<double> M(size, size), N(size, size), P(size, size);
...
P = M * (M+N);
```

Kada bi ostavili lijenu evaluaciju za zbroj matrica možemo vidjeti da ćemo sve elemente zbroja više puta izračunavati zbog matričnog množenja. U takvim slučajevima se lijena evaluacija pokazuje prilično neefikasnom i stoga ćemo i u ovim situacijma račun obavljati na mjestu operacije množenja i rezultat spremiti u privremenu matricu.

4.1 Promjene u dosadašnjoj implementaciji

Na početku ovog poglavlja smo nagovijestili da ćemo morati uspoređivati adrese na desnoj strani pridruživanja sa adresom na lijevoj strani zbog potencijalnog umnoška matrice i vektora. Da ne bismo provjeravali adrese bez stvarne potrebe za to uvodimo dodatan parametar predloška za vektore:

```
template <typename T,
typename Rep = SVector<T>,
bool Mat_Mult = false>

class Vector;
```

Parametar `Mat_Mult` označava pojavljuje li se umnožak matrice i vektora u izrazu. Dodijeljenu vrijednost postavljamo na `false`. Vrijednost se postavlja na `true` samo ako pre-

voditelj nailazi na množenje matrice i vektora i u tom slučaju ostaje `true` sve do kraja prolaska po izrazu. Na taj način nećemo provjeravati adrese ako izraz ne uključuje množenje matrice i vektora i zadržavamo staru funkcionalnost.

Nadalje, više nemamo dvije verzije operatora pridruživanja već tri i to upravo radi spomenutog razloga. Ako je parametar `Mat_Mult` ostao `false` vektore pridružujemo na klasičan način kao i prije. Inače moramo provjeriti adrese.

```
template<typename T,
typename Rep = SVector<T>,
bool Mat_Mult = false>

class Vector{
public:
    // potrebno da korisnik nema pristup adresi vektora
    template <typename, typename, bool> friend class Vector;
    ...
    // operator pridruživanja istim tipom
    Vector& operator= (Vector const& b) {
        for (std::size_t idx = 0; idx < mSize; ++idx) {
            expr_rep[idx] = b[idx];
        }
        return *this;
    }

    // operator pridruživanja za različite tipove
    // nema matričnog množenja
    template<typename T2, typename Rep2>
    Vector& operator= (Vector<T2, Rep2> const& b) {
        for (std::size_t idx = 0; idx < mSize; ++idx) {
            expr_rep[idx] = b[idx];
        }
        return *this;
    }

    // operator pridruživanja za različite tipove
    // ima matričnog množenja
    template <typename T2, typename Rep2>
    Vector& operator= (const Vector<T2, Rep2, true>& b){
        if(b.check_addr(addr())){
            // pronašli smo podudaranje adresa
            T* temp = new T[mSize]; // privremeni objekt
            for (std::size_t idx = 0; idx < mSize; ++idx) {
                temp[idx] = b[idx];
            }
```

```
                for (std::size_t idx = 0; idx < mSize; ++idx) {
                    expr_rep[idx] = temp[idx];
                }
                return *this;
            }

            // ako nismo našli adresu ponašamo se kao i prije
            for (std::size_t idx = 0; idx < mSize; ++idx) {
                expr_rep[idx] = b[idx];
            }

            return *this;
        }

        // provjera nalazi li se adresa lijevog vektora
        // na desnoj strani pridruživanja
        bool check_addr(T* addr) const {
            return expr_rep.check_addr(addr);
        }
        ...

    private:
        Rep expr_rep; // izraz kojeg čuva Vector
        std::size_t mSize; // veličina vektora

        // vraća adresu sirovog vektora
        T* addr() const {
            return expr_rep.addr();
        }
    };
```

Možemo primjetiti da smo ovime dodatno zakomplicirali kod, ali ideja je sasvim jasna. Još preostaje objasniti metodu `check_addr(T*)`. Metodu poziva samo finalni vektor čitavog stabla izraza. Ona na pametan način iskorištava predložne izraze tako da kod svakog čvora stabla vraća zbroj vrijednosti `bool` lijevog i desnog djeteta. Listovi stabla će raditi stvarnu provjeru adrese i vratiti `true` ako se podudaraju, a zbroj će osigurati dovod iste vrijednosti na kraju. Uočimo da višestruko pojavljivanje iste adrese ne utječe na rezultat. Metoda se mora implementirati u svim dosadašnjim klasama i ima isto tijelo u klasama `Add`, `Mult` i novododanoj klasi `Mat_Mult` (ta klasa nema veze sa parametrom predloška iz vektora). Bitne su nam samo adrese vektora pa će metoda u klasi `Scalar` i novododanoj klasi `SMatrix` vraćati `false`, dok će se u klasi `SVector` jedino provjeravati podudaranje adresa. Primjetimo da metodu nećemo morati implementirati u klasi `Matrix` jer nema smisla da klasa `Vector` prima klasu `Matrix` u svom operatoru pridruživanja.

```
template <typename T, typename OP1, typename OP2>
class Mult{
public:
    ...
    bool check_addr(T* addr) const{
        return op1.check_addr(addr) + op2.check_addr(addr);
    }
    ...
};
```

```
template <typename T>
class SVector{
public:
    ...
    bool check_addr(T* addr) const{
        if(addr == storage)
            return true;
        return false;
    }
    ...
};
```

Promjene parametra predloška `Mat_Mult` se obavljaju kod operatora. Naleti li prevodilac na matrično množenje dodijeljenu vrijednost postavlja na `true`, a inače zbraja dodijeljene vrijednosti operanada. Ideja je slična kao i sa metodom `check_addr(T*)` samo što ona kreće od korijena i dolazi do svih listova stabla dok ovdje mijenjamo vrijednost prilikom konstrukcije stabla. Slučaj sa matričnim množenjem ćemo pokazati tek kasnije jer prvo trebamo definirati klasu `Matrix`.

```
// zbroj dvaju vektora
template <typename T, typename R1, typename R2, bool M1, bool M2>
Vector<T, Add<T, R1, R2> M1+M2>
operator+ (const Vector<T, R1, M1>& a, const Vector<T, R2, M2>& b)
{
    return Vector<T, Add<T, R1, R2>, M1+M2 >
        (Add<T, R1, R2>(a.rep(), b.rep()));
}

// umnožak dvaju vektora
template <typename T, typename R1, typename R2, bool M1, bool M2>
Vector<T, Mult<T, R1, R2>, M1+M2 > operator*
(const Vector<T, R1, M1>& a, const Vector<T, R2, M2>& b)
{
    return Vector<T, Mult<T, R1, R2>, M1+M2 >
```

```
        (Mult<T, R1, R2>(a.rep(), b.rep()));
}

// umnožak vektora skalarom
template <typename T, typename R2, bool M2>
Vector<T, Mult<T, Scalar<T>, R2>, M2>
operator*(const T& s, const Vector<T, R2, M2>& b)
{
    return Vector<T, Mult<T, Scalar<T>, R2>, M2 >
        (Mult<T, Scalar<T>, R2>(Scalar<T>(s), b.rep()));
}
```

Dodatne promjene objašnjavamo u narednim odjeljcima.

4.2 Operandi

Za početak pričamo o operandima predložnih izraza. Klase `Add` i `Mult` morat ćemo dodatno preraditi tako da mogu obavljati i matrične operacije. Preciznije, klasa `Add` će sada moći i reprenzentirati zbroj dviju matrica dok klasa `Mult` sada može reprezentirati množenje skalara i matrice. Za umnožak matrice i vektora posebno stvaramo klasu `Mat_Mult`, a za umnožak dviju matrica smo već objasnili proceduru na početku ovog poglavlja.

U klasu `Add` stoga samo dodajemo sljedeće metode:

```
// Matrix = Scalar + Matrix ili
// Matrix = Matrix + Matrix
T operator()(std::size_t row, std::size_t col) const{
    return op1(row, col) + op2(row, col);
}

std::size_t rows() const {
    return op2.rows();
}

std::size_t columns() const {
    return op2.columns();
}
```

Analogno radimo istu stvar za klasu `Mult`:

```
 // Matrix = Scalar * Matrix
T operator()(std::size_t row, std::size_t col) const{
    return op1(row, col) * op2(row, col);
```

```
    }

    std::size_t rows() const {
        return op2.rows();
    }

    std::size_t columns() const {
        return op2.columns();
    }
```

Metode `rows()` i `columns()` imaju istu ulogu kao i metoda `size()` iz odjeljka 3.1 tj. nisu bitne za funkcionalnost. Dodajemo ih samo radi potpunosti. Preopterećeni operatori funkcijskog poziva imaju pak istu ulogu kao i operatori indeskiranja, također kao i u poglavlju 3.1.

Moramo i nadograditi klasu `Skalar` koja sada igra ulogu i u množenju matrice skalarom.

```
    template <typename T>
    class Scalar{
    public:
        ...
        const T& operator() (std::size_t row, std::size_t col) const{
            return s;
        }
        ...
    private:
        const T& s;
};
```

Još preostaje obraditi klasu `Mat_Mult` koja predstavlja množenje matrice i vektora.

```
    template <typename T, typename OP1, typename OP2>
    class Mat_Mult {
    public:
        explicit Mat_Mult(const OP1& a, const OP2& b)
            : op1(a), op2(b) { }

        //// Vector = Mat * Vector
        T operator[] (std::size_t idx) const {
            T sum = 0;
            for (std::size_t idy = 0; idy < op2.rows(); ++idy) {
                sum += op1(idx,idy) * op2[idy];
            }
            return sum;
        }
```

```
        std::size_t size() const {
            return op2.size();
        }

        std::size_t rows() const {
            return op1.rows() != 0 ? op1.rows() : op2.rows();
        }

        std::size_t columns() const {
            return op2.columns();
        }

        bool check_addr(T* addr) const{
            return op1.check_addr(addr) + op2.check_addr(addr);
        }

    private:
        ExprRef<OP1> op1;
        ExprRef<OP2> op2;
    };
```

Čim imamo množenje matrice i vetkora znamo da će na lijevoj strani pridruživanja biti vektor pa klasa `Mat_Mult` nema potrebu preopteretiti operator funkcijskog poziva sa dva argumenta.

4.3 Klasa Matrix

Ideja ostaje ista. S jedne strane imamo klasu koja služi samo za spremište sirove matrica koju nazivamo `SMatrix` dok s druge strane imamo klasu `Matrix` koju koristi korisnik, ali i mi za implementaciju predložnih izraza. Kao i kod vektora, odlučili smo se na spremanje matrice tako da čuvamo adresu jednodimenzionalnog dinamičkog niza koji alociramo u konstruktoru klase `SMatrix`. Za to moramo i čuvati dva dodatna objekta tipa `unsigned int` koji daju informacije o veličini retka i stupca da bismo mogli pristupati elementima matrice. Kao alternativu za čuvanje matrice smo mogli

1. alocirati dvodimenzionalno polje na stogu. Takav način zahtijeva čuvanje podataka o dimenziji matrice parametrima predloška i onemogućuje kasnija proširavanja koja bi uključivala mijenjanje dimenzija od strane korisnika. Takvim pristupom bi dobili nešto na učinkovitosti, ali izgubili na mogućim proširivanjima funkcionalnosti.

2. uz alociranje memorije za elemente matrice dodatno alocirati memoriju za spremanje adresa redaka pa spremati objekt tipa `T**` umjesto `T*`. Taj pristup ipak zahtijeva malo više memorije nego što je stvarno potrebno.

Klasa `SMatrix` izgleda ovako:

```
    template <typename T>
    class SMatrix {
    public:
        explicit SMatrix(std::size_t row, std::size_t col)
            : storage(new T[row*col]), mRows(row), mColumns(col) {
            init();
        }

        const T& operator()(std::size_t row, std::size_t col) const {
            return storage[row * mColumns + col];
        }

        T& operator() (std::size_t row, std::size_t col) {
            return storage[row * mColumns + col];
        }

        const T& operator[] (std::size_t idx) const {
            return storage[idx];
        }

        std::size_t rows() const { return mRows; }

        std::size_t columns() const { return mColumns; }

        T* addr() const {
            return storage;
        }

        bool check_addr(T* addr) const{
            if(addr == storage)
                return true;
            return false;
        }

    protected:
        void init();

    private:
        T* storage;
        std::size_t mRows;
        std::size_t mColumns;
};
```

Kako matricu u memoriji reprezentiramo kao jednodimenzionalan niz tada za dohvaćanje

elementa `storage(i,j)` koristimo formulu `[i * mColumns + j]` gdje `mColumns` označava broj stupaca. Također smo morali implementirati i operator indeksiranja radi množenja matrica i vektora.

U nastavku opisujemo klasu `Matrix`:

```
template
<typename T, typename Rep = SMatrix<T>>

class Matrix {
public:

    explicit Matrix(std::size_t row, std::size_t col)
        : expr_rep(row,col), mRows(row), mColumns(col) {  }

    // sada nam je bitan konstruktor kopije
    template <typename T2, typename Rep2>
    Matrix(const Matrix<T2, Rep2>& b)
        : expr_rep(b.rows(),b.columns()), mRows(b.rows()),
        mColumns(b.columns())
    {
        for (std::size_t idx = 0; idx < mRows; ++idx) {
            for (std::size_t idy = 0; idy < mColumns; ++idy) {
                expr_rep(idx, idy) = b(idx, idy);
            }
        }
    }

    // operator pridruživanja istim tipom
    Matrix& operator= (const Matrix& b) {
        for (std::size_t idx = 0; idx < mRows; ++idx) {
            for (std::size_t idy = 0; idy < mColumns; ++idy) {
                expr_rep(idx, idy) = b(idx, idy);
            }
        }
        return *this;
    }

    // operator pridruživanja za različite tipove
    template <typename T2, typename Rep2>
    Matrix& operator= (const Matrix<T2, Rep2>& b) {
        for (std::size_t idx = 0; idx < mRows; ++idx) {
            for (std::size_t idy = 0; idy < mColumns; ++idy) {
                expr_rep(idx, idy) = b(idx, idy);
            }
        }
```

```
            return *this;
        }

        std::size_t rows() const {
            return mRows;
        }

        std::size_t columns() const {
            return mColumns;
        }

        decltype(auto) operator()
        (std::size_t row, std::size_t col) const {
            return expr_rep(row,col);
        }

        T& operator() (std::size_t row, std::size_t col) {
            return expr_rep(row, col);
        }

        const Rep& rep() const {
            return expr_rep;
        }

        Rep& rep() {
            return expr_rep;
        }

    private:
        Rep expr_rep;
        std::size_t mRows;
        std::size_t mColumns;
};
```

Jedina novost koju ovdje imamo za razliku od vektora je konstruktor kopije. Njega ćemo koristiti u operatoru množenja za stvaranje privremenog objekta kada jedan od faktora reprezentira izraz, a ne sirovu matricu. Taj izraz sigurno ne uključuje umnožak matrica jer u tom slučaju također stvaramo privremeni objekt i ”brišemo” odgovarajući dio stabla. Stoga jedine operacije u tom izrazu mogu biti umnožak matrice skalarom ili zbrajanje matrica. Obe te operacije se ponašaju točkovno bez utjecaja ostalih elemenata pa se isplati koristiti lijena evaluacija.

4.4 Operatori

Implementacija operatora će se sada znatno zakomplicirati zahvaljujući umnošku matrica. Zbrajanje i umnožak skalarom i vektorom obavljamo na standardan način dok umnožak matrica dijelimo na čak četiri slučaja. Slučajeve dijelimo prema tipovima operanada. Ako su oba operanda sirove matrice tada nema potrebe za stvaranjem privremenih objekata za te matrice i možemo odmah iskoristiti pametnu petlju po recima. U ostalim slučajevima barem jedna matrica predstavlja stablo podizraza. Kako je već objašnjeno na početku poglavlja ovog poglavlja, tada je pametnije odmah evaluirati taj podizraz da sprječimo višestruka računanja istih elemenata.

```
// Matrix + Matrix
template <typename T, typename R1, typename R2>
Matrix<T, Add<T, R1, R2> > operator+
(const Matrix<T, R1>& a, const Matrix<T, R2>& b)
{
    return Matrix<T, Add<T, R1, R2> >(Add<T, R1, R2>(a.rep(), b.↩
       rep()));
}

// Scalar * Matrix
template <typename T, typename R2>
Matrix<T, Mult<T, Scalar<T>, R2> >
operator* (const T& s, const Matrix<T, R2>& b)
{
    return Matrix<T, Mult<T, Scalar<T>, R2> >
        (Mult<T, Scalar<T>, R2>(Scalar<T>(s), b.rep()));
}

// Matrix * Vector
template <typename T, typename R1, typename R2, bool M2>
Vector<T, Mat_Mult<T, R1, R2>, true> operator*
(const Matrix<T, R1>& a, const Vector<T, R2, M2>& b)
{
    return Vector<T, Mat_Mult<T, R1, R2>, true>
        (Mat_Mult<T, R1, R2>(a.rep(), b.rep()));
}
```

```
// Matrix(stablo) * Matrix(stablo)
template <typename T, typename R1, typename R2>
Matrix<T> operator*
(const Matrix<T, R1>& a, const Matrix<T, R2>& b)
{
```

```
        Matrix<T> new_a(a);
        Matrix<T> new_b(b);
        Matrix<T> result(a.rows(), b.columns());

        for(int idx = 0; idx < a.rows(); ++idx){
            for(int idz = 0; idz < a.columns(); ++idz){
                for(int idy = 0; idy < b.columns(); ++idy){
                    result(idx,idy) += new_a(idx,idz)*new_b(idz,idy);
                }
            }
        }

        return result;
    }

    // Matrix(sirova) * Matrix(stablo)
    template <typename T, typename R2>
    Matrix<T> operator*
    (const Matrix<T, SMatrix<T>>& a, const Matrix<T, R2>& b)
    {
        Matrix<T> new_b(b);

        Matrix<T> result(a.rows(), b.columns());
        for(int idx = 0; idx < a.rows(); ++idx){
            for(int idz = 0; idz < a.columns(); ++idz){
                for(int idy = 0; idy < b.columns(); ++idy){
                    result(idx,idy) += a(idx,idz) * new_b(idz,idy);
                }
            }
        }

        return result;
    }

    // Matrix(stablo) * Matrix(sirova)
    template <typename T, typename R1>
    Matrix<T> operator*
    (const Matrix<T, R1>& a, const Matrix<T, SMatrix<T>>& b)
    {
        Matrix<T> new_a(a);

        Matrix<T> result(a.rows(), b.columns());
        for(int idx = 0; idx < a.rows(); ++idx){
            for(int idz = 0; idz < a.columns(); ++idz){
                for(int idy = 0; idy < b.columns(); ++idy){
                    result(idx,idy) += new_a(idx,idz) * b(idz,idy);
                }
```

```
            }
        }

        return result;
    }

    //// Matrix(sirova) * Matrix(sirova)
    template <typename T>
    Matrix<T> operator*
    (const Matrix<T, SMatrix<T> >& a, const Matrix<T, SMatrix<T> >& b)
    {
        Matrix<T> result(a.rows(), b.columns());

        for(int idx = 0; idx < a.rows(); ++idx){
            for(int idz = 0; idz < a.columns(); ++idz){
                for(int idy = 0; idy < b.columns(); ++idy){
                    result(idx,idy) += a(idx,idz) * b(idz,idy);
                }
            }
        }

        return result;
    }
```

Primjetimo da u prvoj verziji operatora množenja stvaramo čak tri privremena objekta. Također obratimo pozornost na izbor elemenata pri samom množenju. Elementi u unutarnjoj petlji su ili konstantni ili se kreću po recima matrice čime dobivamo na efikasnosti. Ideja je da vanjska petlja predstavlja redak matrice dok sa dvije unutarnje obavimo sve potrebne promjene za fiksni redak. Po matricama se krećemo tako da se ne fokusiramo na fiksni element rezultatne matrice već fiksiramo element lijevog faktora i svako malo mijenjamo odgovarajući stupac istog retka.

4.5 Verifikacija i performanse

Kao i prije, moramo se uvjeriti funkcionira li čitava ova priča i koliko brzo.

```
int main() {

    Vector<double> x = { -12, 32.2, 54, 4 };
    Vector<double> y = { 2.12, 0.21, -23.1, -1 };
    Vector<double> z = { 76.2, -32, 13.122, 90.1 };

    Matrix<double> m1 = {{37.47, -5.626, -29.3, 13}, // 4x4
```

```
                    {-51.4, -73.9, 9, 21.80},
                    {-20.59, -54.70, 39.402, -77.79},
                    {11.13, -12.13, 58.2, -42.98} };

    Matrix<double> m2 = { {4.75, 29}, {16.5, -7.7}, // 4x2
                          {2.48, -45},{-36.37, 5.127} };

    Matrix<double> m3 = { {-20.59, -4.7}, {-9.31, 28.48} }; // 2x2

    Vector<double> x1 = x;
    Vector<double> y1 = y;
    Vector<double> z1 = z;

    // spremiste
    Matrix<double> mat(4, 2);

    ////// prvi izraz
    x = m1 * x;
    std::cout << "Rezultat:     " << x;
    std::cout << "WolframAlpha: "
              << "(-2161, -1189.58, 302.288, 2446.73)" << std::endl;

    ////// drugi izraz
    y = (m1 + m1) * (y + y);
    std::cout << "Rezultat:     " << y;
    std::cout << "WolframAlpha: "
              << "(2968.34, -1416.75, -3550.14, -5121.57)"
              << std::endl;

    ////// treći izraz
    mat = (m1+m1) * (m2+m2) * (m3+m3);// (4x4)x(4x2)x(2x2) = (4x2)
    std::cout << "Rezultat:     \n" << mat;
    std::cout << "WolframAlpha: \n"
            << "[-111501, 590348, \n 458470, -192781,
            \n-142480, -607372,\n -76523.7, -610764]" << std::endl;

    ////// četvrti izraz

    z1 = 1.2 * (m1 + m1) * x1 + 2.3 * (m1 + m1) * y1 +
            3.4 * (m1 + m1) * z1;
    std::cout << "Rezultat:     " << z1;
    std::cout << "WolframAlpha: "
              <<   "(24217.3, -877.54, -46267.9, -12750.8)"
              << std::endl;

    return 0;
}
```

Slijedi ispis gornjeg koda:

```
Rezultat:     [-2161, -1189.58, 302.288, 2446.73]
WolframAlpha: (-2161, -1189.58, 302.288, 2446.73)
Rezultat:     [2968.34, -1416.75, -3550.14, -5121.57]
WolframAlpha: (2968.34, -1416.75, -3550.14, -5121.57)
Rezultat:
[-111501, 590349,
 458470, -192780,
 -142480, -607371,
 -76523.9, -610764]
WolframAlpha:
[-111501, 590348,
 458470, -192781,
-142480, -607372,
 -76523.7, -610764]
Rezultat:     [24217.3, -877.546, -46267.9, -12750.9]
WolframAlpha: (24217.3, -877.54, -46267.9, -12750.8)
```

Kod mjerenja malim slovom smo označili vektore, a velikim matricu. Veličina sada označava ili veličinu vektora ili broj redaka odnoso stupaca matrice (uzimamo kvadratne matrice radi jednostavnosti). Stoga veličinu smanjujemo jer na primjer, spremati matricu dimenzija 100 000×100 000 zauzima previše memorije. Kao prvu odnosno zadnju veličinu uzimamo broj 32 odnosno 320 da matrica ima sveukupno 1000 odnosno 100 000 elemenata.

Tablica 4.1: Rezultati mjerenja za izračunavanje izraza `x = (M+M)*(y+y);`

	Implementacija		
Veličina	Komponente	Naivna implementacija	Predložni izrazi
32	0 µs	0 µs	0 µs
100	9.97 µs	39.9 µs	9.97 µs
320	109.71 µs	329.12 µs	109.7 µs

Tablica 4.2: Mjerenja za `w = 1.2*M*x + 2.3*(M+N)*(3.4*y + 4.5*z);`

	Implementacija		
Veličina	Komponente	Naivna implementacija	Predložni izrazi
32	9.97 µs	9.97 µs	9.98 µs
100	39.88 µs	134.49 µs	29.91 µs
320	199.46 µs	892.67 µs	229.38 µs

Tablica 4.3: Rezultati mjerenja za izračunavanje izraza `P = M+M+N+N;`

Implementacija			
Veličina	Komponente	Naivna implementacija	Predložni izrazi
32	0 µs	0 µs	0 µs
100	19.95 µs	129.69 µs	10 µs
320	119.6 µs	728.31 µs	79.78 µs

Tablica 4.4: Rezultati mjerenja za izračunavanje izraza `P = (M+M)*(N+N);`

Implementacija			
Veličina	Komponente	Naivna implementacija	Predložni izrazi
32	19.93 µs	39.89 µs	29.92 µs
100	618.58 µs	678.19 µs	688.41 µs
320	18031.8 µs	18879.5 µs	18301.3 µs

Sudeći po mjerenjima, predložne izraze se općenito isplati implementirati i u matricama bez dodatnih optimizacija. Anomalija se pojavljuje kod mjerenja zadnjeg izraza. Rezultati tamo su približno jednaki neovisno o implementaciji. Razlog leži u tome što će u posljednjem izrazu sve implementacije raditi na sličan način. Za oba zbroja ćemo stvoriti po jedan privremeni objekt. Jedina zapravo razlika je što petlja po komponentama neće zahtijevati poseban privremeni objekt za množenje već ćemo umnožak moći direktno pridružiti rezultatnoj matrici.

Poglavlje 5

Eigen

Eigen je C++ biblioteka prvenstveno namijenjena za rad s objektima iz linearne algerbre. Podržava operacije nad matricama, vektorima i omogućuje znastvena računanja u numeričkoj matematici i geometriji. Osnovali su ju Benoît Jacob i Gaël Guennebaud u prosincu 2006. goodine.

Eigen pruža brojne mogućnosti. Podržava matrice svih veličina, od malih matrica fiksne veličine do proizvoljno velikih gustih matrica (matrica koje imaju relativno veliki broj ne-nul elemenata) ili proizvoljno velikih rijetkih matrica (matrica čiji elementi su većinom nule). Podržava sve standardne numeričke tipove uključujući integralne tipove, tip `std::complex` i omogućuje jednostavno proširivanje numeričkih tipova. Također podržava razne matrične dekompozicije i geometrijske operacije. Njen sustav nepodržanih modula odnosno korisničkih doprinosa pruža mnoga specijalizirana svojstva poput nelinearnih optimizacija, matričnih funkcija, kalkulatora s polinomima, brzih Fourierovih transformacija i mnogih drugih.

Eigen se odlikuje brzinom računanja. Osnovna karakteristika njegovih operacija su upravo predložni izrazi kojima na inteligentan način uklanja privremene objekte i omogućuje lijeno izračunavanje; baš kako smo pokazali u poglavljima 3 i 4. Matrice fiksne veličine su potpuno optimizirane jer se tada može izbjeći dinamička alokacija koja je puno skuplja od alokacije na stogu. Odmotava petlje kada to ima smisla odnosno kada uspije procijeniti da će time postići veću učinkovitost. Za velike matrice posebna pozornost se posvećuje prilagođenosti priručnoj memoriji *eng. cache-friendliness* da bi se ubrzalo čitanje iz memorije. Također vrši eksplicitnu vektorizaciju koda za određene skupove naredbi (eng. *instruction sets*) ako ih procesor podržava. Na taj način Eigen dodatno potiče prevoditelj na vektorizaciju jer moderni prevoditelji uglavnom mogu sami automatski vektorizirati kod bez potrebe eksplicitnog sustava. Objasnimo ukratko što bi to bila vektorizacija.

Vektorizacija koda se odnosi na specijalan slučaj automatske paralelizacije koda gdje se računalni program pretvara iz skalarne implementacije, koja istovremeno obrađuje jedan par operanda, u vektorsku implementaciju, koja obrađuje jednu operaciju na više parova operanada odjednom. Moderna računala tipično podržavaju vektorske operacije koje istovremeno obrađuju više skalarnih operacija kao što su na primjer četiri operacije zbrajanja. Na taj način iz obične petlje

```
for (i=0; i<n; i++)
    c[i] = a[i] + b[i];
```

vektorizacijom možemo dobiti kod nalik na

```
for (int i=0; i<16; i+=4)
    c[i:i+3] = a[i:i+3] + b[i:i+3];
```

gdje druga linija označava istovremeno zbrajanje i pridruživanje elemenata niza. Bitno je razlikovati vektorizaciju od odmotavanja petlji. Odmotavanjem petlje iz istog primjera možemo dobiti

```
for (i=0; i<n; i+=4){
    c[i] = a[i] + b[i];
    c[i+1] = a[i+1] + b[i+1];
    c[i+2] = a[i+2] + b[i+2];
    c[i+3] = a[i+3] + b[i+3];
}
```

Razlika ovdje je što odmotavanjem petlje ne uzrokujemo paralelizaciju koda već samo sprječavamo dodatne troškove u efikasnosti zbog instrukcija grananja.

Eigen je pouzdan i elegantan za korištenje. Implementirani algoritmi su pažljivo odabrani radi sigurnosti. Biblioteka je temeljito testirana preko vlastitih testova sa više od 500 izvršnih datoteka. Programsko sučelje je izuzetno prirodno zahvaljujući predložnim izrazima. Također Eigen ima dobru prevoditeljsku podršku jer su testovi između ostalog vršeni i na raznim prevoditeljima.

5.1 Matrice

Da bi koristili biblioteku Eigen potrebno je samo skinuti datoteke zaglavlja. Postoji i automatska instalacija kojom se uz dateoteke zaglavlja dobiva i dokumentacija skupa sa testo-

vima. Detaljna uputstva se mogu pronaći na službenoj stranici [3].

Stvaranje matrica koristeći Eigen je jednostavno:

```
#include <iostream>
#include <Eigen/Core>

using namespace Eigen;

int main(){
    Matrix3f A;
    Matrix4d B;
}
```

Eigen koristi nazivlja za njihove tipove podataka koje podsjećaju na OpenGL. Stvarno ime koje tip predstavlja praćeno je sufiksom koji označava njegovu veličinu i tip elemenata koje sprema respektivno. U gornjem primjeru, varijabla `A` je tipa `Matrix3f` - odnosno, matrica dimenzija 3×3 čiji elementi su tipa `float`. Varijabla `B` je matrica dimenzija 4×4 elemenata tipa `double`. Valja imati na umu da bilo kakva kombinacija dimenzija i tipova ne mora nužno funkcionirati, na primjer `Matrix2i` će stvoriti `int` matricu dok `Matrix5s` neće stvoriti `short int` matricu već će deklaracija vratiti grešku. To ne znači da mi ne možemo stvoriti matricu čiji su elementi tipa `short int`, samo Eigen zahtijeva malo drukčiju sintaksu:

```
    // 5x5 matrica tipa short
    Matrix <short, 5, 5> M1;

    // 20x75 matrix tipa float
    Matrix <float, 20, 75> M2;
```

Zapravo, na ovaj način su matrice i implementirane u biblioteci Eigen. Imena koje smo opisali na početku samo pružaju prirodniji način deklaracija određenih tipova. Eigen omogućuje i stvaranje matrica čije veličine nisu poznate za vrijeme kompilacije uvođenjem slova X na mjesto veličine u sufiksu, na primjer `MatrixXf` ili `MatrixXd`.

Sada kada smo stvorili naše matrice možemo im pridružiti i vrijednosti:

```
    // prvi način
    A << 1.0 f, 0.0 f, 0.0 f,
         0.0 f, 1.0 f, 0.0 f,
         0.0 f, 0.0 f, 1.0 f;
```

```
// drugi način
for(int i = 0; i<4; ++i){
    for(int j = 0; j<4; ++j){
        B(j,i) =  0.0;
    }
}
```

Prvi način prikazuje metodu inicijalizacije korištenjem zareza tako da korisnik navodi koeficijente redak po redak. Svi koeficijenti matrice moraju biti navedeni, inače ako se broj koeficijenata ne podudara sa punom veličinom matrice, prevoditelj vraća grešku. Stoga lako uočavamo da metoda postaje veoma nezgrapna za veće matrice.

Drugi način prikazuje pristup pojedinim koeficijentima matrice `B` korištenjem preopterećenog operatora funkcijskog poziva. Indeksiranje kreće od 0 i prvi indeks određuje broj retka dok drugi indeks broj stupca. Primjetimo da smo strukturirali petlje tako da se vanjska petlja se kreće po stupcima, a unutarnja po recima. Naravno, s obzirom da "inicijaliziramo" matricu na 0, vrijedilo bi i obratno. Razlog ove pomalo zbunjuće sintakse je taj što Eigen sprema matrice (ukoliko to ne promijenimo parametrom predloška) u poretku po stupcima. Preciznije, matrice se spremaju u memoriji tako da se prvo zapisuje prvi stupac, pa drugi, treći i tako dalje. Znamo već iz poglavlja 4 da je najefikasniji način kretanja po matrici u onom poretku u kojem je spremljena jer na taj način, ako se matrica spremila u priručnoj memoriji, smanjujemo broj promašaja (eng. *cache misses*) što oduzima značajan dio vremena. To je razlog zašto općenito preferiramo iteriranje vanjskom petljom po stupcima. Za naš primjer konkretno nismo trebali obraćati pažnju na to jer naša matrica nije velika i stane u prvu razinu (L1) priručne memorije.

Nadalje, Eigen pruža dodatne načine na koji možemo pridružiti vrijednosti elementima matrice.

```
// Postavlja koeficijente matrice na slučajno odabrane vrijednosti
// iz intervala [-1,1]
A = Matrix3f::Random ();

// Postavlja matricu B na jediničnu matricu
B = Matrix4d::Identity();

// Postavlja sve elemente na nulu
A = Matrix3f::Zero();

// Postavlja sve elemente na jedan
B = Matrix3f::Ones();

// Postavlja sve elemente na konstantnu vrijednost
B = Matrix4d::Constant(4.5);
```

5.1.1 Operacije

Uobičajeni aritmetički operatori su preopterećeni za rad s matricama:

```
Matrix4f M1 = Matrix4f::Random();
Matrix4f M2 = Matrix4f::Constant(2.2) ;
// Zbrajanje
// Veličine i tipovi koeficijenata se moraju podudarati
std::cout << M1 + M2 << endl;

// Mnozenje
// Veličine i tipovi koeficijenata se moraju podudarati
std::cout  << M1 * M2 << endl;

// Množenje skalarom i oduzimanje
std::cout  << M2 - Matrix4f::Ones() * 2.2 << endl;
```

Jednakost (==) i nejednakost (!=) su jedini relacijski operatori koji rade na matricama. Matrice se smatraju jednakima ako su odgovarajući koeficijenti isti.

```
std::cout << (M2 - Matrix4f::Ones() * 2.2 == Matrix4f::Zero());
```

Uobičajene matrične operacije su dane kao metode klase:

```
// Transponiranje
std::cout << M1.transpose() << std::endl;

// Invertiranje (#include <Eigen/Dense>)
// Generira NaNs ako matrica nije invertibilna
std::cout << M1.inverse() << std::endl;
```

Ponekad je zgodno operirati s matricama kao običnim nizovima. To se može postići pozivajući metodu `array()`:

```
// Kvadriramo svaki element matrice
std::cout << M1.array().square() << std::endl;

// Množimo dvije matrice kao sto množimo vektore
std::cout << M1.array() * Matrix4f::Identity().array()
          << std::endl;
```

```
    // Sada možemo iskoristiti sve relacijske operatore
    std::cout << M1.array() <= M2.array() << std::endl;
    std::cout << M1.array() > M2.array() << std::endl;
```

Primjetimo da posljednje operacije ne mijenjaju matricu koja ih poziva već samo vraćaju novu promijenjenu matricu.

5.2 Vektori

Vektor u biblioteci Eigen nije ništa drugo nego matrica sa jednim stupcem:

```
typedef Matrix <float, 3, 1> Vector3f;
typedef Matrix <double, 4, 1> Vector4d;
```

Posljedično, mnogi operatori i funkcije koje smo naveli kod matrica također rade i za vektore. Imenovanje tipova je također slično, ovdje sufiks označava jednodimenzionalnu veličinu za razliku od matrica.

```
    // Pridruživanje zarezom
    v << 1.0f, 2.0f, 3.0f;

    // Pristup koeficijentima
    std::cout << v(2) << endl;

    // Vektori do veličine 4 se mogu inicijalizirati u konstruktoru
    Vector3f w(1.0f, 2.0f, 3.0f) ;

    // Razne korisne statičke funkcije
    Vector3f v1 = Vector3f::Ones();
    Vector3f v2 = Vector3f::Zero();
    Vector4d v3 = Vector4d::Random();
    Vector4d v4 = Vector4d::Constant(1.8);

    // Aritmetički operatori
    std::cout << v1 + v2 << std::endl;
    std::cout << v4 - v3 << std::endl;

    // Množenje skalarom
    std::cout << v4 * 2 << std::endl;

    // Jednakost i nejednakost su ponovno jedini relacijski operatori
    std::cout << Vector2f::Ones()==Vector2f::Constant(1) << std::endl;
```

Kako su vektori samo jednodimenzionalne matrice, umnožak matrice i vektora funkcionira očekivano sve dok se unutarnje dimenzije i tipovi koeficijenata operanada podudaraju:

```
Vector4f v5 = Vector4f(1.0f, 2.0f, 3.0f, 4.0f);

// 4x4 * 4x1 - Radi!
std::cout << Matrix4f::Random() * v5 << endl;

// 4x1 * 4x4 - Greska pri kompilaciji!
std::cout << v5 * Matrix4f :: Random () << endl;
```

Očekivano, matrično množenje između dva vektora neće raditi jer se unutarnje dimenzije ne podudaraju. To jednostavno rješavamo metodom `transpose()`:

```
v1 = Vector3f::Random();
v2 = Vector3f::Random();
std::cout << v1 * v2.transpose() << std::endl;
```

Vektori imaju ugrađene funkcije za skalarni umnožak kao i za ostale uobičajene operacije nad njima u linearnoj algebri:

```
// Skalarni umnožak
cout << v1.dot(v2) << std::endl;

// Normiranje vektora
cout << v1.normalized() << std::endl;

// Vektorski produkt
cout << v1.cross(v2) << std::endl;
```

Konačno, operacije po točkama se mogu obaviti tako da objavimo biblioteci da želimo tretirati vektor kao običan niz:

```
std::cout << v1.array() * v2.array() << std::endl;
std::cout << v1.array().sin() << std::endl;
```

5.3 Usporedba Eigen-a i naše implementacije

Usporedimo performanse biblioteke Eigen sa našim implementacijama iz poglavlja 3 i 4. Naš fokus je bio isključivo na predložnim izrazima nad dinamički alociranim nizovima

stoga i prilikom korištenja biblioteke Eigen koristimo vektore i matrice alocirane na hrpi (eng. *heap*).

5.3.1 Vektori

Napomenimo da smo za potrebe mjerenja umjesto klase `VectorXd` koristili klasu `ArrayXd` radi izbjegavanja konstantog poziva metode `array()` u izrazu da tumači vektor kao niz.

Tablica 5.1: Rezultati mjerenja za izračunavanje izraza `z = 1.2*x + x*y;`

Implementacija			
Veličina niza	Komponente	Predložni izrazi	Eigen
10	0 µs	0 µs	0 µs
1000	19.94 µs	9.96 µs	9.99 µs
10 000	109.7 µs	79.78 µs	70.08 µs

Tablica 5.2: Mjerenja za `z = 1.2*x*(x+y) + 2.3*y*(y+z) + 3.4*z*(x+z);`

Implementacija			
Veličina niza	Komponente	Predložni izrazi	Eigen
10	0 µs	0 µs	0 µs
1000	29.94 µs	19.94 µs	29.91 µs
10 000	239.37 µs	209.44 µs	199.47 µs

Rezultati pokazuju da naša implementacija može konkurirati sa bibliotekom Eigen koristimo li samo vektore. Za veće izraze možemo vidjeti određene razlike, ali one nisu dovoljno velike da bi se zaključilo da je Eigen znatno brži.

5.3.2 Matrice

Tablica 5.3: Rezultati mjerenja za izračunavanje izraza `x = (M+M)*(y+y);`

Implementacija			
Veličina	Komponente	Eigen	Predložni izrazi
32	0 µs	0 µs	0 µs
100	9.96 µs	9.97 µs	9.97 µs
320	139.63 µs	90.03 µs	149.59 µs

Tablica 5.4: Mjerenja za `w = 1.2*M*x + 2.3*(M+N)*(3.4*y + 4.5*z);`

Implementacija			
Veličina	Komponente	Eigen	Predložni izrazi
32	0 µs	0 µs	0 µs
100	19.93 µs	9.97 µs	29.91 µs
320	219.42 µs	119.67 µs	229.37 µs

Tablica 5.5: Rezultati mjerenja za izračunavanje izraza `P = M+M+N+N;`

Implementacija			
Veličina	Komponente	Eigen	Predložni izrazi
32	0 µs	0 µs	0 µs
100	19.95 µs	9.96 µs	9.98 µs
320	109.71 µs	99.76 µs	109.71 µs

Tablica 5.6: Rezultati mjerenja za izračunavanje izraza `P = (M+M)*(N+N);`

Implementacija			
Veličina	Komponente	Eigen	Predložni izrazi
32	39.88 µs	19.98 µs	39.88 µs
100	618.58 µs	369.27 µs	688.41 µs
320	18031.8 µs	12775.9 µs	18301.3 µs

Kod matrica se pak lako vidi da Eigen postiže bolje performanse od naših implementacija. Razlike postaju očite čim ubacimo matrično množenje u izraz pogotovo ukoliko se radi o množenju dviju matrica. Eigen koristi dodatne tehnike i optimizacije među kojima ističemo eksplicitnu vektorizaciju. Eigen u dokumentaciji [3] navodi ručno iskorištavanje takozvanih *Streaming SIMD* skupova naredbi koje podržava i naš procesor. Pretpostavljamo da upravo ta optimizicija poboljšava njihove performanse kada je riječ o matričnim množenjima.

Poglavlje 6

Predložni izrazi kroz povijest

6.1 Začeci ideje

Predložne izraze razvili su neovisno Todd Veldhuizen i David Vandevvoorde. Tehnika je osmišljena još u vrijeme dok nismo mogli posebno parametrizirati članove klasa i čak se smatralo da tako nešto nikada neće biti dio C++ standarda. To je bio kamen spoticanja za implementacije koje bi koristile predložne izraze jer nismo mogli posebno parametrizirati operator pridruživanja koji bi kao argument primio čitavo stablo izraza ukodirano u jednom vektoru. Taj se problem rješavao korištenjem polimorfizma. Koristili bi operator konverzije u klasu `Copier` koja bi reprezentirala stablo, a prerađivala bi klasu `CopierInterface` preko koje bi vezali referencu na nju. Na taj način pametno izbjegavamo parametriziranje operatora pridruživanja.

```
template<typename T>
class CopierInterface {
public:
    virtual void copy_to(Vector<T, SVector<T>>&) const;
};

template<typename T, typename X>
class Copier : public CopierInterface<T> {
public:
    Copier(X const& x) : expr(x) {}

    virtual void copy_to(Array<T, SArray<T>>&) const {
        // implementacija petlje kod pridruživanja
        ...
    }
    private:
        X const& expr;
```

```
};

template<typename T, typename Rep = SVector<T>>
class Vector {
public:
    // operator pridruživanja
    Array<T, Rep>& operator=(CopierInterface<T> const& b) { b.↩
        copy_to(rep); };
    ...
};

template<typename T, typename A1, typename A2>
class A_mult {
    public:
    // operator konverzije
    operator Copier<T, A_Mult<T, A1, A2>>();
    ...
};
```

Ovo nadodaje dodatnu razinu kompleksnosti i dodatne troškove pri izvođenju naspram implementacije iz poglavlja 3, ali čak i s tim manama, performanse su se pokazale impresivne za to vrijeme.

Na ljeto 1994. Veldhuizen, kao redovni student i stažist u tvrtki Rogue Wave Software u Oregonu, radi na razvijanju C++ biblioteke za obrade slika. Osmišljava predložne izraza kao način optimizacije izraza koji obavljaju račune u tzv. algebri slika (na primjer, "zbrajanje" triju slika). Tvrtka objavljuje tehničko izvješće u svibnju 1994. godine i prezentira se na sastanku odbora za standardizaciju jezika C++. U rujnu 1994. godine Veldhuizen započinje rad na biblioteci koja s vremenom postaje veoma popularna. Riječ o biblioteci Blitz++. U siječnju iste godine daje govor o predložnim izrazima na svjetskoj konferenciji jezika C++.

Neovisno, Vandevoorde također dolazi do iste ideje. U veljači 1995. objavljuje dijelove koda o numeričkim nizovima, reprezentirani klasom `valarray<Troy>`, čija je implementacija bila zasnovana na predložnim izrazima. Tehniku naziva *template expressions*, ali nedugo zatim prihvaća Veldhuizen-ov naziv *expression template*. Kako oba autora osmišljavaju ideju u istom razdoblju neovisno jedan o drugom, i Veldhuizen i Vandevoorde dijele zaslugu. [7]

Standardna biblioteka sadrži predložak klase `valarray` koji se trebao koristiti za primjene koje bi opravdale korištenje tehnika poput preložnih izraza. Prvotno je `valarray` bio dizajniran sa namjerom da bi prevoditelji koji su pogodni za znanstvena računanja prepoznali taj tip i koristili značajne optimizacije za operacije nad nizovima. Međutim, to se nije pokazalo uspješnim. Takvih prevoditelja je relativno malo na tržištu, a i pro-

blem prepoznavanja tipa se otežao otkako je `valarray` postao predložak klase, a ne samo klasa. Nakon što je tehnika predložni izrazi otkrivena, Vandevoorde je odboru za standardizaciju jezika C++ predložio promjenu rada klase `valarray` koji bi sveobuhvatno koristio predložne izraze. Prijedlog je nažalost stigao prekasno tijekom procesa prve standardizacije jezika. Čitav dotadašnji tekst vezan za klasu `valarray` bi, prihvaćanjem tog prijedloga, trebao biti izmjenjen stoga prijedlog nije odobren. Za uzvrat, odbor je podesio par promjena u opisu klase kako bi se implementacija ipak mogla konstruirati korištenjem predložnih izraza. No, pokazalo se da je iskorištavanje tih promjena poprilično kompleksan posao. Do 2018. godine niti jedna takva implementacija nije bila poznata pa su operacije nad objektima tipa `valarray` općenito poprilično neučinkovite.

6.2 Popularizacija tehnike

6.2.1 Boost.proto

Predložni izrazi su se najprije primjenjivali za izraze nad tipovima koji su reprezentirali nekakve oblike nizova. Ali nekoliko godina nakon otkrića tehnike pojavile su se i nove primjene. Među najinovativnijima ističemo biblioteku Boost.proto autora Eric-a Nieblera.

Proto je razvojna cjelina (eng. *framework*) za izgradnju ugrađenih jezika za specifičnu domenu (DSEL) u jeziku C++. Pruža alate za izgradnju, transformaciju i izvršavanje predložnih izraza. Preciznije, Proto pruža

- strukturu podataka za stablo izraza;
- mehanizam za unapređivanje izraza;
- preopterećenja operatora za izgradnju stabla iz izraza;
- mehanizam za neposredno izvršavanje predložnih izraza;
- skup transformacija primjenjivih na stabla izraza.

Neformalno rečeno, Proto je biblioteka koja implementira predložne izraze umjesto programera za svakakve upotrebe. Biblioteka samostalno definira preopterećenja operatora i konstrukciju stabla izraza. Također samostalno izračunava izraz kodiran tim stablima. [5]

6.2.2 Blitz++

Blitz++ je biblioteka meta-predložaka za operacije nad nizovima u C++ čije se efikasnosti mogu usporediti sa Fortranovim implementacijama. Implementacije koriste predložne iz-

raze zahvaljujući kojim se optimizacije poput sažimanja petlji (eng. *loop fusion*), odmotavanja petlji i specijalizacije algoritama mogu automatski obavljati tijekom kompilacije bez oslanjanja na prevoditeljske optimizacije. Zanimljivost je da je njen tvorac upravo Todd Veldhuizen, jedan od dvojice začetnika tehnike predložnih izraza. [8]

6.2.3 Boost.spirit

Boost.spirit je skup biblioteka u jeziku C++ za parsiranje i generaciju izlaza implementiranih kao ugrađene jezike za specifičnu domenu koristeći predložne izraze i metaprogramiranje predloščima. Autori su Joel de Guzman i Hartmut Kaiser. Biblioteke omogućuju izradnju gramatike isključivo pisanih u C++-u. Specifikacije se mogu slobodno miješati sa drugim C++ kodom i zahvaljujući predloščima mogu se i neposredno izvršavati. [1]

Bibliografija

[1] J. de Guzman i H. Kaiser, *Boost.spirit*, http://boost-spirit.com/home/about-2/, 2009, [Online; accessed 04-September-2021].

[2] D. Dragaš, *Predložni izrazi*, https://github.com/domdrag/Diplomski_vs_code, 2021.

[3] B. Jacob i G. Guennebaud, *Eigen*, https://eigen.tuxfamily.org/, 2006.

[4] Godbolt M., *Compiler Explorer*, http://https://godbolt.org/, 2012.

[5] Eric Niebler, *Boost.Proto*, https://www.boost.org/doc/libs/1_62_0/doc/html/proto.html, 2008, [Online; accessed 04-September-2021].

[6] D. Vandevoorde, N. Josuttis i D. Gregor, *C++ Templates: The Complete Guide II*, Addison-Wesley, 2017.

[7] Todd Veldhuizen, *Who invented expression templates?*, https://web.archive.org/web/20041218004656/http://osl.iu.edu/~tveldhui/papers/priority.html#et, 2004, [Online; accessed 04-September-2021].

[8] ———, *Blitz++*, https://github.com/blitzpp/blitz/wiki, 2018, [Online; accessed 04-September-2021].

[9] Wikipedia, *Metaprogramming — Wikipedia, The Free Encyclopedia*, http://en.wikipedia.org/w/index.php?title=Metaprogramming&oldid=1032193891, 2021, [Online; accessed 01-September-2021].

Sažetak

U diplomskom radu obrađena je tehnika predložni izrazi u programskom jeziku C++ kojom se sprječava nepotrebno kopiranje kod operacija, ali i smanjuje broj čitanja odnosno zapisavanja unutar memorijskih lokacija. Tehnika spada pod metaprogramiranje i da bi ju čitatelj mogao u potpunosti razumjeti važno je prvo se upoznati sa metaprogramiranjem. Tehnika se općenito koristi u implementaciji klasa numeričkih nizova, ali se može i koristiti u raznim drugim primjenama. Teoretski, tehnika se može implementirati u bilo kojim situacijama koje bi uključivale operacije nad objektima.

Summary

This thesis describes the expression templates technique in C++ programming language which prevents unnecessary copying within operations, but also minimizes the number of readings that is, writings inside memory locations. The technique falls into metaprogramming so in order to completely grasp the concept of it, it is important for a reader to be familiar with metaprogramming in general. The technique is used in implementation of numeric arrays, but might also be used in other applications. Theoretically, we can implement it in any circumstances as long as we use some kind of operations on objects.

Životopis

Domagoj Dragaš rođen je 21.11.1997. godine u Zagrebu, Republika Hrvatska. Osnovnu školu pohađa u malom mjestu Vidovec nedaleko od Zagreba. Godine 2012. upisuje III. gimnaziju u Zagrebu, prirodoslovno-matematički smjer te se odlučuje na upis na Prirodoslovno-matematički fakultet u Zagrebu - Matematički odsjek 2016. godine. Uspješno završava preddiplomski studij 2019. godine te stiče zvanje Sveučilišnog prvostupnika matematike (univ.bacc.math). Svoje usavršavanje nastavlja na istom fakultetu i upisuje Diplomski studij Računarstvo i matematika.