Funktionell programmering inom mjukvaruprojekt

Lucas Fransman
Kandidatavhandling i datavetenskap
Handledare: Kristian Nybom
Fakulteten för naturvetenskaper och teknik
Åbo Akademi
30 mars 2021

Innehåll

1 Introduktion

...

2 Historia

De första digitala fullt funktionerbara datorerna byggdes under 1940-talet i stor del motiverat av andra världskriget. Forskare under andra världskriget behövde snabbare metoder för att lösa praktiska problem vilket ledde till utvecklingen av digitala datorer. Det här var maskiner som krävde flera tiotals kvadratmeter med fysiskt utrymme och vägde flera ton.

De första kommersiella datorerna var endast tillgängliga för större organisationer. Bland de första kommersiellt tillgängliga datorerna var UNIVAC:n som var designad för affärs- och administrativt bruk. UNIVAC:n lanserades år 1951 ursprungligen för 159,000 dollar, men priset steg gradvist upp till 1,500,000 dollar.

Uppfinningar så som transistorn och integrerade kretsen revolutionerade datorerna. Tills dess hade datorer varit uppbyggda av tusentals elektronrör som tog mycket utrymme och var opålitliga. Transistorn ledde till mindre, snabbare, och pålitligare datorer. Men det var fortfarande en utmaning att designa komplexa kretsar då hundratals separata komponenter behövdes fästas ihop med varandra. Det här var både dyrt och tidskrävande. Integrerade kretsen löste det här problemet genom att koppla alla komponenter på en enda yta var komponenterna är ihopkopplade med varandra genom en tunn metallyta.

Under 1970-talet började de första företagen lansera persondatorer menade för individuella hushåll [1]. En av de här företagen var Apple Computer, numera Apple som idag är världens värdefullaste bolag med ett marknadsvärde på mer än 2 biljoner dollar [2]. Idag i jämförelse med de första årtionden har datorernas beräkningskapacitet ökats med en faktor på flera miljoner. Datorn är inte bara standardutrustning i de flesta hemmen, utan också utanför i väskan eller fickan i form av en bärbar dator eller smarttelefon. Mjukvaruindustrins utveckling har naturligt följt efter hårdvarans utveckling. Mjukvarans roll i samhället har växt snabbt. För att kunna hålla med denna förfrågan har också processen samt verktygen för mjukvaruutveckling sett stora framsteg.

2.1 Mjukvaruutveckling

Metoden för mjukvaruutveckling under 1950- och 1960-talet var rätt enkel: Eftersom kod alltid kommer att innehålla en massa fel bör koden skrivas så snabbt som möjligt för att sedan snabbt kunna börja korrigera dem. Det var klart under slutet av 1960-talet att de dåvarande angreppssätten var bristfälliga och att det fanns ett brådskande behov för förändring. Natos vetenskapskommitté organiserade två konferenser som lyfte fram problem inom den dåtida mjukvaruindustrin så som budget- och tidsöverskridningar, samt problem inom kvalitet och pålitlighet för levererad mjukvara. Det här ledde till att mjukvaruutveckling blev sin egen disciplin då det insågs att mjukvaruutveckling skiljde sig en hel del från vetenskap och matematik. Mjukvaruutvecklare ansågs vara ingenjörer som istället för fysiska strukturer eller maskiner byggde mjukvara och behövde därför lära sig ingenjörs kunskaper för att bygga säkra produkter för samhället. [1]

Mjukvaruutveckling handlar inte strikt om programmering utan är ett systematiskt angreppsätt för utveckling och underhåll av mjukvara vilket innebär identifiering av krav för en mjukvaruprodukt, och därmed design och utveckling för en lösning som uppnår de här kraven. Det här omfattar metoder för att designa, utveckla, implementera och testa mjukvara samt ordentliga projekt-, kvalitets-, och konfigurationshanterings praktiker. För att förenkla den här processen har modeller utvecklats så som vattenfalls modellen under 1970-talet och spiral modellen under 1980-talet. Den agila modellen som används flitigt idag introducerades i början av 1990-talet. Den agila modellen påstods var mer responsiv till kundens behov än de då mera traditionell metoderna som vattenfalls modellen var förändring av krav kunde orsaka problem. [1]

2.2 Programmering

Det första datorprogrammet är generellt daterat tillbaka till 1843 då matematiker Ada Lovelace publicerade en algoritm för att beräkna Bernoullital vilket är en sekvens av rationella tal som ofta förekommer inom talteori. Programmet var avsett för Charles Babbages analytiska maskin som aldrig blev byggd men är den första ritningen på en Turingkomplett maskin. [3]

De första programmeringsspråken använde sig av maskinkod för att instruera datorn. De här språken referas som första generationens programmeringsspråk och användes för de första datorerna. följande utveckling var användning av assemblerkod på slutet av 1940-talet vilket användes för att representera maskininstruktioner på ett mera människovänligt sätt. Assemblerkoden översätts till maskinkod med hjälp av en assembler vilket är ett program som konverterar assemblerinstruktioner till maskinkod.[1] Assemblerkod var en klar förbättring men var fortfarande tidskrävande att skriva samt väldigt benägen för misstag. De första högnivå språken togs i bruk under 1950-talet. Det första var Fortran utvecklat av John Backus[1] designat för vetenskaps- och ingenjörsapplikationer. År 1959 introducerades COBOL vilket var utvecklat för applikationer inom finanssektorn. Följande stora utveckling kom tidigt under 1970-talet från Bell Labs där Dennis Ritchie utvecklade programmeringsspråket C. C var ursprungligen utvecklat för att skriva kerneln till UNIX operativsystemet som Ritchie tidigare hade utvecklat tillsammans med Ken Thompson men C blev snabbt populärt inom resten av industrin och är fortfarande idag bland de mest använda programmeringsspråken [5].

Objektorienterad programmering populariserades under 1980-talet av Bjarne Stroustrup genom programmeringsspråket C++ som var baserat på C och Simula. Simula var ett programmeringsspråk utvecklat under 1960-talet avsett för simulering. Simula var det första språket som introducerade objektorienterade koncept så som klasser, objekt, arv, och virtuella procedurer.

Funktionell programmering såg sin start redan i slutet av 1950-talet med programmeringsspråket Lisp utvecklat av Jhon McCarthy som tog inflytande från Alonzo Churchs notation från lambdakalkylen. Lisp grundlade många koncept som ses idag bland de mera traditionella programmeringsspråken så som trädbaserade datastrukturer, skräpsamling, dynamiska typer, rekursion, och funktioner av högre ordning.

[1]John Backus tog emot Turingpriset år 1977 för hans bidrag till design av praktiska högnivåprogrammeringssystem, särskilt genom arbetet med Fortran. [4]

3 Programmeringsparadigm

Ett programmeringsparadigm är ett sätt att klassificera programmeringsspråk baserat på deras egenskaper och funktionalitet. Varje programmeringsparadigm är definierad som en mängd av programmeringskoncept som bygger ihop ett enkelt kärnspråk. Programmeringsspråk kan implementera ett eller flera programmeringsparadigm, vilket ger en beskrivning på språket på en mera abstrakt nivå. Det finns färre programmeringsparadigm än programmeringsspråk. Från paradigmsynvinkeln kan olika språk som Java, C++, och Python ses som nästan identiska då de alla implementerar det objektorienterade paradigmet. [6]

Det förekommer många olika typer av problem då mjukvara ska byggas. Olika programmeringsparadigm fungerar bäst för att lösa specifika problem. Därför implementerar de flesta programmeringsspråken flera än ett programmeringsparadigm. [6] I det här kapitlet kommer jag att beskriva några programmeringsparadigm som implementerats av de mest använda programmeringsspråken för att se hurdan effekt de har på programmeringsspråken samt hur de skiljer sig från varandra.

3.1 Imperativ programmering

Med imperativ programmering beskrivs datorberäkningar med programtillstånd och kommandon som förändrar dessa tillstånd. Ett program består av en mängd kommandon som exekveras av datorn, vilka beskriver explicit hur programmet ska exekvera. Exekveringen av ett kommando leder generellt till en ändring i programmets tillstånd. [1]

Både maskinkod och assemblerspråk är imperativa, men de flesta imperativa språk som används idag är högnivåspråk. Imperativa högnivåspråk använder sig av variabler för att lagra data i minnet. I ett imperativt språk är det tillåtet att tilldela värden åt en variabel och under exekveringens gång modifiera värdet. Detta leder

till en ändring i programtillståndet. [1]

```
1 bool innehållerTvå(std::array<int> lista) {
2     for(int i = 0; i < lista.size(); i++) {
3         if(lista[i] == 2)
4             return true;
5     }
6     return false;
7 }
```

Ovanför är ett exempel på en imperativ procedur som kontrollerar om en lista med heltal innehåller en tvåa. Proceduren itererar över listan och kontrollerar för varje iteration ifall elementet är två. I exemplet ovan finns en variabel i som håller reda på vilket element i listan vi är på under en iteration. För varje iteration ökas värdet inuti i med ett för att vi ska kunna hämta nästa element från listan. Det här är ett exempel på ett tilldelningskommando vilket tar ett värde från minnet, utför en operation på värdet, och lagrar resultatet tillbaka i minnet. Detta leder till en förändring i programtillståndet.

3.2 Deklarativ programmering

Med deklarativ programmering beskrivs vad ett program ska göra istället för hur. Om vi implementerar samma procedur *innehållerTvå* som i exemplet från imperativ programmering på ett deklarativt sätt ser vi en tydlig skillnad.

```
1 bool innehållerTvå(std::array<int> lista) {
2     return std::any_of(std::begin(lista), std::end(lista), 2);
3 }
```

Här används en C++ biblioteksfunktion *any_of* som går igenom listan och kontrollerar ifall något av heltalen är en tvåa. I det imperativa exemplet beskriver vi explicit rad för rad hur proceduren ska utföras medan i det deklarativa exemplet har vi endast en rad som beskriver vad proceduren ska göra. Båda procedurerna ger samma resultat och är kompilerbara från och med C++11 [2].

[2] C++11 är en version av C++. Talet på slutet berättar vilket år versionen släpptes.

Deklarativ programmering kan ses som ett sorts abstraktions lager. Funktionaliteten finns färdigt definierad, oftast inuti ett bibliotek inom ett programmeringsspråk. Ett typiskt exempel på ett deklarativt språk är SQL som är ett standardiserat språk för att hämta och modifiera data från en relationsdatabas.

3.3 Objektorienterad programmering

Den traditionella strukturen på ett program består ofta av en samling av procedurer vilka i sig kan innehålla en mängd underprocedurer. Det finns ingen direkt gräns hur djupt den här kedjan med procedurer och underprocedurer kan gå.

Med objektorienterad programmering så strukturerar vi ett program istället med en samling av objekt. Ett objekt kan ses som en självständig entitet med en egen roll eller ansvarsområde. För att skapa ett objekt behövs en klass som definierar egenskaper och beteende.

```
1  class Koordinat {
2      int x;
3      int y;
4      Koordinat(int x, int y): x(x), y(y) {} // Konstruktor
5
6      void flytta(int x_rörelse, int y_rörelse) {
7          x = x + x_rörelse;
8          y = y + y_rörelse;
9      }
10 }
```

Ovan är ett exempel på en klass som definierar en tvådimensionell koordinat. En koordinat är här definierad som två heltal x och y. Vi har också en konstruktor som används då vi skapar ett objekt baserat på en klass. I vårt fall kräver vår konstruktor två stycken heltal som argument vilka används för att initialisera x och y värdena i *Koordinat* objektet. För att beskriva beteende används metoder. I vårt exempel har vi en metod *flytta* som tillåter oss att flytta på koordinaten.

```
1  Koordinat k = Koordinat(2, 2);  // k.x = 2, k.y = 2
2  k.flytta(-1, 2);                // k.x = 1, k.y = 4
```

Här skapar vi med konstruktorn ett *Koordinat* objekt k med både x och y initialiserade som två. Sedan flyttar vi k ett steg bakåt och två steg uppåt med *flytta* metoden. Här ser vi en av de fundamentala principerna för objektorienterad programmering; enkapsulering. Enkapsulering tillåter oss att gömma objekts inre attribut och implementationsdetaljer bakom ett gränssnitt som är lätt att använda.

Objektorienterad programmering tillåter oss även att skapa klasshierarkier. Klasser kan ärva egenskaper och beteende från andra klasser och kan sedan bygga på den ärvda funktionaliteten. Det här tillåter oss att strukturera våra program på ett naturligt och effektivt sätt.

3.4 Funktionell programmering

Ett program skrivet med ett funktionellt programmeringspråk består av en mängd funktioner. Till skillnad från de funktioner vi är vana med från de mer traditionella språken strävar funktioner i funktionella programmeringsspråk att fungera som matematiska funktioner. Matematiska funktioner är ett samband mellan två olika mängder, en definitionsmängd och en värdemängd. Sambandet beskrivs genom en matematisk formel eller i vissa fall genom en tabell. [7]

Det här leder till att angreppssättet för att lösa problem med funktionella programmeringsspråk skiljer sig en hel del i jämförelse med imperativa programmeringsspråk. I imperativa programmeringsspråk sparas beräkningsresultat i minnet vilket representeras av en variabel inom programmet. Ett strikt funktionellt programmeringsspråk använder sig inte av variabler vilket befriar programmeraren att behöva tänka på förändrande programtillstånd. Utan variabler kan vi inte heller använda oss av iterativa konstruktioner som ofta ses inom imperativa språk utan repetition måste istället definieras med rekursion [8]. Utan programtillstånd är det garanterat att funktioner alltid beter sig på samma sätt och ger samma resultat för ett givet argument vilket leder till att göra resonemang över ett funktionellt program kan vara betydligt lättare än för ett imperativt program var möjliga förändringar i programtillstånd alltid måste tas i beaktande. [7]

I våra tidigare exempel med funktionen *innehållerTvå* itererades alla element i listan ända tills vi kom fram till en tvåa eller slutet på listan. Nedan är ett exempel på samma funktion men implementerat med Haskell vilket är ett funktionellt

programmeringsspråk.

```
1 innehållerTvå :: [Int] -> Bool
2 innehållerTvå [] = False
3 innehållerTvå x:xs = if x == 2 then True else innehållerTvå xs
```

På första raden definieras datatyperna för funktionen den sista datatypen berättar vilken datatyp returvärdet kommer ha i vårt fall *Bool* de som kommer före är datatyperna för funktions parametrarna i vårt fall har vi endast en *[Int]* vilket är en lista med heltal. Rad två och tre definierar beteende för funktionen. Ifall listan är tom är returvärdet falskt för det finns inga mer element att gå igenom. Annars kollar vi ifall första elementet i listan är en tvåa och isåfall är returvärdet sant om inte så anropar vi funktionen rekursivt med samma lista utan det första elementet som just granskades.

I exemplet används mönstermatchning vilket är en egenskap för funktionella programmeringsspråk. Ett mönster är ett schema som matchas med funktionsparametrarna. När en funktion anropas jämförs parametrarna med de givna mönstren i given ordning och matchas med första träff [8]. I vårt exempel hade vi två givna mönster ett för tomma listor och den andra för icke tomma listor.

```
1 ärAdmin :: String -> String -> Bool
2 ärAdmin "admin1" "lösenord1" = True
3 ärAdmin "admin2" "lösenord2" = True
4 ärAdmin "admin3" "lösenord3" = True
5 ärAdmin _ _ = False
```

Här ser vi lite tydligare hur mönstermatcning fungerar. Funktionen kollar om ett givet användarnamn och lösenord tillhör en administratör. I exemplet har vi tre stycken administratörer. Om det givna användarnamnet och lösenordet matchar nån av de givna mönstren så kommer funktionen att returnera sant om inte så kommer den att matcha med det sista mönstret var tecknet _ fungerar som ett jokertecken där vad som helst duger och returnera falskt.

4 Funktionell programmering inom mjukvaruprojekt

Mjukvarusystem har blivit en viktig del inom stora delar av industrin. Mjukvaruprojekt har blivit allt större och berör större delar av organisationer vilket leder till större risker för företag ifall någonting går fel. Vilket det ofta gör. Enligt Michael Blochs, Sven Blumbergs, och Jürgen Lartzs undersökning tillsammans med Oxfords universitet går i genomsnitt 45% av stora mjukvaruprojekt över budget, 7% överskrider tidtabellen, och 56% tillhandahåller mindre värde än förutspått. [9]

Processen för utveckling av mjukvara är komplex och innehåller många delar. Modeller för mjukvaruutveckling samt verktyg utvecklas konstant [10]. Vi kommer att fokusera oss på programmeringsspråk. Mera specifikt på var och hur vi kan dra nytta av egenskaper från funktionella programmeringsspråk.

4.1 Upprätthållbarhet

Ett mjukvaruprojekt kan ta flera år att utveckla och kan kräva upprätthållning under hela mjukvarans livstid. Att lätt kunna göra ändringar eller lägga till ny funktionalitet är viktigt. Det här kan dock bli svårt speciellt ifall det har skapats beroenden mellan kodmoduler vilket lätt skapas då ändringar eller ny funktionalitet läggs till som inte tagits i beaktande då systemet originellt utvecklats.

En viktig faktor för upprätthållbarhet är läsbarhet. Programmerare jobbar oftast i lag med flera människor och måste kunna förstå kod skriven av andra. Funktionella programmeringspråk är deklarativa vilket gör koden lättare att förstå då man inte behöver oroa sig över implementationsdetaljer [11].

Med funktionella program har vi inget programtillstånd. I imperativa språk sker förändring av programtillstånd genom att tilldela värden till variabler. I funktionella språk är all data oföränderlig vilket gör det lättare att göra resonemang över program då vi kan vara säkra att data inte förändras under exekveringens gång.

Kodåteranvändnig sker naturligt i funktionella programmeringsspråk. Funktionella program är uppbyggda av funktioner vilka är lätta att återanvända. Det här förenklar också att göra förändringar i koden då vi endast behöver göra förändringar

inuti en funktion istället för alla platser där funktionaliteten behövs.

Att testa mjukvaran är en viktig del av utvecklingsprocessen för att försäkra sig att strävad funktionalitet har uppnåtts och för att hitta eventuella buggar innan mjukvaran når användarna. Att bygga test för funktioner är enkelt eftersom vi kan vara säkra att funktioner alltid returnerar samma värde för givna argument. Vi behöver inte heller oroa oss över sidoeffekter eller förändrande programtillstånd.

4.2 Säkerhet

...

4.2.1 Typ system

...

4.3 Effektivitet

...

4.3.1 Oföränderlig data

Funktionella programmeringsspråk använder sig av oföränderlig data. Datastrukturer så som strängar och listor kan inte förändras i minnet efter att de har blivit deklarerade. Istället skapas alltid en ny datasruktur då vi vill göra en förändring på en redan existerande datastruktur [12]. Det låter väldigt ineffektivt att för varje modifiering behöva skapa en ny datasruktur. Speciellt för stora datastrukturer som kan innehålla tusentals element.

För att lösa det här problemet utnyttjar vi oss av den gamla datasrukturen och faktumet att den gamla datasrukturen inte kan förändras. Datastrukturen delas upp i noder och då en förändring görs behöver vi endast allokera minne för den noden där modifieringen skedde. För resten av datastrukuturen kan vi återanvända noderna från den gamla datastrukturen för de kommer inte att förändras under exekveringens gång. [13]

4.3.2 Lat utvärdering

Lat utvärdering är en utvärderings metod som implementeras av en stor del av de funktionella programmeringsspråken. De två främsta egenskaperna är att utvärderingen av ett uttryck skjuts upp ända fram tills resultatet behövs och första gången uttrycket evalueras sparas resultatet så följande gång kan det sökas upp istället för att återberäknas. [13]

Det här speciellt nyttigt då vi har att göra med stora datastrukturer eller om vi vill endast behandla en liten del av en datastruktur. [14]

```
1 let lista = [1, 2, 3...]
```

Här är en "oändlig" lista i Haskell som innehåller alla positiva heltal. Listan sparas inte i minnet utan evalueras då någonting behövs från den.

4.3.3 Rekursion

I funktionella programmeringsspråk har vi inga iterativa konstruktioner så som loopar som vi är vana med från imperativa programmeringsspråk. Som tidigare nämnts används rekursion för att sköta upprepningar. Rekursion är ett sätt att definiera en funktion var funktionen själv används i definitionen. Definitioner inom matematiken ges ofta rekursivt, ett välkänt exempel är Fibonaccisekvensen. [15]

$$f_0 = 0 \qquad f_1 = 1$$
$$f_n = f_{n-1} + f_{n-2}$$

I Fibonaccisekvensen är ett tal i sekvensen summan av de två föregående talen i sekvensen. De två första talen f_0 och f_1 är färdigt definierade, de kallas basfall och behövs för att en funktion ska kunna termineras. [15]

```
1 fibonacci :: Int -> Int
2 fibonacci 0 = 0
3 fibonacci 1 = 1
4 fibonacci n = fibonacci (n-1) + fibonacci (n-2)
```

Om vi jämför samma funktion implementerat i Haskell så ser vi att de ser nästan

identiska ut. Mönstermatchning är ett väldigt nyttigt verktyg då de kommer till rekursiva funktioner, speciellt för att definiera basfall. [15]

Ofta då vi använder oss av loopar vill vi iterera över en samling och göra någonting med elementen lagrade inuti samlingen. Det här görs även enkelt med rekursion, låt oss säga att vi vill räkna summan av alla tal i en lista.

```
1 summa :: [Int] -> Int
2 summa [] = 0
3 summa x:xs = x + summa xs
```

Här har vi en funktion *summa* som tar en lista med heltal som argument och returnerar ett heltal. Funktionen tar första heltalet från listan och adderar det med resultatet från samma funktion med samma lista utan det första heltalet. I det här fallet är en tom lista vårt basfall. Så funktionen fortsätter ända tills den får en tom lista som argument vilket är logiskt då vi vill gå igenom alla element.

$$
\begin{aligned}
&\mathit{summa}\ [1,2,3] \\
&\rightarrow 1 + \mathit{summa}\ [2,3] \\
&\rightarrow 1 + 2 + \mathit{summa}\ [3] \\
&\rightarrow 1 + 2 + 3 + \mathit{summa}\ [] \\
&\rightarrow 1 + 2 + 3 + 0 \\
&\rightarrow 6
\end{aligned}
$$

Här ser vi alla funktions anrop evaluerade om funktionen *summa* skulle anropas med en lista med heltalen ett, två, och tre inuti.

4.3.4 Sidoeffekter

...

4.4 Parallella system

Från 1986 till 2002 har prestandan på processorer ökat med 50% i medel per år. Det här betydde att mjukvaruutvecklare ofta kunde vänta på nästa generationen av

processorer för att åstadkomma ökad prestanda för deras applikationer. Sedan 2002 har ökningen i prestandan sjunkt till 20% per år i medel. Det här är en dramatisk förändring, med 50% prestanda ökning per år åstadkoms en prestanda ökning med en faktor på nästan 60 över 10 år. Medan med en 20% ökning per år åstadkoms en prestanda ökning med en faktor på sex över 10 år. [16]

Det här har lett till en förändring i processor design. Från 2005 har de flesta processor tillverkare att parallelism är bästa lösningen för att bibehålla en hög prestanda ökning. Istället för att försöka fortsätta utveckla snabbare monolotiska processorer har mera fokus lagts på att få flera processorer på en enda integrerad krets. [16]

Den här förändringen har stora konsekvenser på mjukvaruutveklings processen. Att öka på mängden processorer kommer inte automatiskt att öka prestandan på majoriteten av program som är huvudsakligen skrivna för processorer med endast en kärna. Sådana program är inte medvetna om att flera processorer är tillgängliga och kommer därför inte att se en stor prestanda ökning i jämförelse med en enkärnig processor. [16]

Att skriva program för parallella system medför en del utmaningar: Arbete måste fördelas i jämnlöpande delar för att de ska kunna exekveras parallellt. Ordningen uttryck evalueras kan ha effekt på resultatet. Delade minnesplatser måste vara låsta för att undvika konflikter som kan orsaka inkonsekventa resultat. Programmerare måste vara noggranna att inte skapa beroenden mellan arbeten som kan blockerar program exekveringen. Program måste vara skalbara för att utnyttja en ökande mängd parallella processorer. Arbetet måste balanseras mellan processorerna så jämnt som möjligt för att optimalt kunna utnyttja alla processresurser. [17]

Strikt funktionella programmeringsspråk har många fördelar då det kommer till parallell programmering. Den största fördelen är att pågrund av att funktioner inte innehåller sidoeffekter är det alltid säkert att utföra dem parallellt vilket gör fördelning av arbete i jämnlöpande delar lätt. Det har ingen betydelse i vilken ordning beräkningar utförs, resultatet kommer alltid vara det samma. Därmed kommer resultatet vara exakt samma som det skulle vara ifall programmet skulle körts sekventiellt. Försett att alla parallella beräkningar behövs för programmets resultat, om programmet termineras då det körs sekventiellt kommer det också att termineras då det körs parallellt. Det här möjliggör att programmet kan debuggas sekventiellt

vilket förlättar processen betydligt. [17]

På grund av ordningen in- och utdata operationer utförs är definierade i funktionella programmeringsspråkens implementation blir det inte problem med ordningen uttryck evalueras. En stor del eller all problem med låsandet av minnesplatser och synkronisering kan hanteras fullständigt inuti programmeringsspråkets implementation. beroenden mellan arbeten kan inte uppstå, data- och kontrollberoenden i fuktipnella programmeringsspråk försäkrar att det inte kan finnas olösta beroenden mellan arbeten. [17]

5 Sammanfattning

...

Referenser

[1] G. O'Regan, *Introduction to the history of computing: a computing history primer*. Springer, 2016.

[2] "Yahoo finance," https://finance.yahoo.com/quote/AAPL/, hämtat Mars 2021.

[3] T. J. Misa, "Charles babbage, ada lovelace, and the bernoulli numbers," in *Ada's Legacy: Cultures of Computing from the Victorian to the Digital Age*. Association for Computing Machinery, 2016, pp. 11–31.

[4] "Acm turing award," https://amturing.acm.org/.

[5] "Tiobe index for current month," https://www.tiobe.com/tiobe-index/, TIOBE, hämtat Mars 2021.

[6] P. Van Roy *et al.*, "Programming paradigms for dummies: What every programmer should know," *New computational paradigms for computer music*, vol. 104, pp. 616–621, 2009.

[7] R. W. Sebesta, *Concepts of programming languages*. Boston: Pearson,, 2012.

[8] M. Gabbrielli and S. Martini, *Programming languages: principles and paradigms*. Springer Science & Business Media, 2010.

[9] M. Bloch, S. Blumberg, and J. Laartz, "Delivering large-scale it projects on time, on budget, and on value," *Harvard Business Review*, pp. 2–7, 2012.

[10] C. Ryder, "Software measurement for functional programming," Ph.D. dissertation, Computing Lab, University of Kent, 2004.

[11] O. Torgersson, "A note on declarative programming paradigms and the future of definitional programming," *Das Winteroete*, vol. 96, no. 1996, p. 13, 1996.

[12] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, "Making data structures persistent," *Journal of computer and system sciences*, vol. 38, no. 1, pp. 86–124, 1989.

[13] C. Okasaki, *Purely functional data structures.* Cambridge University Press, 1999.

[14] M. L. Scott, *Programming Language Pragmatics (Third Edition)*, 2009.

[15] M. Lipovaca, *Learn you a haskell for great good!: a beginner's guide.* no starch press, 2011.

[16] P. Pacheco, *An introduction to parallel programming.* Elsevier, 2011.

[17] K. Hammond, "Why parallel functional programming matters: Panel statement," in *International Conference on Reliable Software Technologies.* Springer, 2011, pp. 201–205.