IZBIRA JEZIKA ZA PODATKOVNO VODENE RAČUNALNIKE

P. KOKOL, M. OJSTERŠEK, V. ŽUMER
TEHNIŠKA FAKULTETA MARIBOR
B. STIGLIC - ISKRA AVTOMATIKA LJUBLJANA

UDK: 681.3.06

POVZETEK - V članku opisujemo lastnosti programskih jezikov, ki so primerni za programiranje podatkovno vodenih računalnikov.

ABSTRACT - Language evaluation for data flow systems. In this paper properties of programming languages suitable for data flow systems programming are given.

1. UVOD

Večina današnjih računalnikov temelji na več kot trideset let stari von Neumannovi organizaciji. Ti računalniki so sekvenčni, opravijo eno operacijo na časovno enoto, imajo en sam procesni element, sekvenčno centralizirano kontrolno enoto, nizek strojni jezik in linearen pomnilnik s fiksno dolžino celic. Da se poveča hitrost takšnim računalnikom, je potrebno povečati hitrost posameznim enotam računalnika. Vendar maksimalna hitrost, omejena s hitrostjo svetlobe, ne bo zadoščala za kompleksne izračune, kakršne lahko pričakujemo v prihodnosti. Zato računalniki prihodnjih generacij temeljijo na arhitekturah, ki dovoljujejo paralelno delovanje in s tem omogočajo, da se mnogo operacij izvrši naenkrat. Od paralelnih arhitektur, ki veliko obetajo, so za nas zanimive podatkovno vodene arhitekture. Delovanje teh je asinhrono, kar pomeni, da vrstni red izvajanja operacij ni določen z nekim centralnim kontrolnim mehanizmom, ampak se operacije izvršijo takrat, ko imajo na voljo vse potrebne vhodne argumente. Podatkovno vodene programe predstavimo kot usmerjene grafe, katerih vozlišča so operacije, povezave pa predstavljajo pretok podatkov, zato jim pravimo grafi pretoka podatkov.

Dobre lastnosti podatkovno vodene arhitekture so:
a) asinhrono proženje,
b) velika možnost paralelnosti,
c) velja pravilo enkratne prireditve,
d) nadzorne ali kontrolne enote niso potrebne.

Slabosti podatkovno vodene arhitekture so:
a) da je potrebno pri vsaki operaciji s polji ali zapisi prenesti celoten podatek, kajti podatkovno vodeni modeli nimajo pomnilnikov, kamor bi te podatke spravljali,
b) da v primeru, ko algoritmi vsebujejo premalo paralelnosti, so te arhitekture manj učinkovite kot kontrolno vodene.

Zaradi naštetih slabosti podatkovno vodeni računalnik ni primeren za splošno namenske računalnike. Njegove dobre lastnosti pa kažejo, da je uporaben za vodenje procesov, robotiko in superračunalnike.

2. PROGRAMSKI JEZIKI

V zadnjih letih so bili razviti mnogi programski jeziki, ki omogočajo hkratno izvajanje programov (Ada, Modula, Concurent Pascal, Edison, razni dialekti Fortrana...). Slabost teh jezikov je, da so jezikovni konstrukti, ki omogočajo hkratno izvajanje programov, le pripomoček programerju, da pokaže prevajalniku, kje so možne paralelnosti. Ti jeziki so nenaravni, kajti prilagojeni so sistemom, na katerih se izvajajo, ne pa načinu, kako človek (programer) razmišlja. Če opazujemo razvoj programske opreme, na eni strani vidimo, da se je pojavila t.i. "softverska kriza", ki bi jo lahko rešili z uvedbo mnogo višjega programskega jezika, ki bi zadovoljeval množico zahtev t.i. "referential transparency" /7/. Na drugi strani, pa razvoj materialne opreme z uvedbo VLSI teži k vse večji paralelnosti in k vse bolj paralelnim arhitekturam. Tudi jeziki, ki bi ustrezali takšnim arhitekturam, bi morali zadovoljevati natanko enake zahteve kot so zahteve "referential transparency".

2.1. Paralelnosti v programskih jezikih

Vidimo, da tako razvoj materialne kot razvoj programske opreme zahteva uvedbo jezikov, ki bi omogočali enostavno (blizu človekovemu načinu razmišljanja) in učinkovito izražavo paralelnosti. Zato si poglejmo, na kakšne načine se paralelnosti pojavljajo v programskih jezikih /14/:
a) eksplicitna paralelnost - pomeni, da je program naravno paralelen (inherentna paralelnost), ali da so za paralelnost dodani posebni jezikovni konstrukti (forall zanke, cobegin itd.),
b) lokalna paralelnost - pomeni, da se paralelnost odkriva v majhnih delih programa (interproceduralno) kot so bloki, zanke itd.,
c) globalna paralelnost - pomeni, da je za odkrivanje paralelnosti potrebna neka globalna intraproceduralna analiza, ki je nepraktična in neučinkovita,
d) neodkriljiva paralelnost - ta nastane zaradi odvisnosti med podatki ali zaradi neučinkovite predstavitve programa. Postopkovni programski jeziki z globalnimi spremenljivkami, stranskimi efekti in nestrukturiranimi jezikovnimi konstrukti (GO TO, aritmetični if stavek) vsebujejo v glavnem neodkriljive paralelnosti.

Paralelnost lahko odkrivamo na treh nivojih:
a) na algoritemskem nivoju (npr. program v zelo visokem programskem jeziku ali posebni paralelni algoritmi) odkrivamo paralelnosti, kadar programski jezik omogoča zadostno abstrakcijo algoritma in vsebuje veliko eksplicitnih paralelnosti,
b) na nivoju izvornega jezika (npr. program v visokem programskem jeziku) odkrivamo paralelnosti s po-

sebnim predprocesorjem. Te paralelnosti zaradi slabe reprezentacije ne moremo odkriti na algoritemskem nivoju,

c) v času izvajanja (npr. program v strojnem kodu) odkrivamo paralelnosti dinamično. To postane nepraktično, kadar se mora paralelno izvesti mnogo instrukcij in torej ni primerno za moderne VLSI arhitekture, ki omogočajo hkratno izvajanje več tisoč instrukcij.

2.2. Lastnosti programskih jezikov

Programske jezike lahko razdelimo, kot so prikazani na sliki 1.

Programi pisani v postopkovnih jezikih imajo naslednje pomembne lastnosti:

a) Uporabljajo model s stanji, kar onemogoča učinkovito paralelno izvajanje in zahteva uporabo arhitekture s stanji.
b) Uporabljajo kompleksne nematematične strukture, s čimer otežkočijo ali celo onemogočijo odkrivanje paralelnosti ter optimizacijo, preslikavo in verifikacijo programov.
c) Uporabljajo globalno okolje, kar povzroča velike odvisnosti med podatki, s čimer omejujejo paralelno izvajanje in dekompozicijo programov.
d) Izvajajo operacije z eno besedo podatka na časovno enoto, tudi če mora biti ista operacija izvedena z mnogo besedami (von Neumannovo ozko grlo).
e) Preslikavajo "pomnilnik" v "pomnilnik", zaradi tega so nefleksibilni in jih je težko sestavljati v obsežnejše programe.
f) Povzročajo veliko semantično vrzel med koncepti visokih programskih jezikov in koncepti obstoječih arhitektur.
g) Omogočajo samo deterministično programiranje, zaradi tega jih je težko uporabljati za reševanje problemov umetne inteligence, ki zahteva heuristično obdelavo podatkov.

V nasprotju s postopkovnimi jeziki pa nepostopkovni jeziki vsebujejo mnogo lastnosti, ki omogočajo izražava paralelnosti, enostavno verifikacijo programov, sestavljanje programov v kompleksnejše programe idr.

Lastnosti posameznih jezikov so podane v tabeli 1, kjer imajo okrajšave naslednji pomen:

S	model s stanji
N	programski jezik vsebuje nestrukturirane konstrukte
G	programski jezik uporablja globalno okolje
P	paralelnost
SE	možnost sestavljanja manjših programov v večje
PR	postopkovni jezik
NPR	nepostopkovni jezik
V	možnost matematične verifikacije programov
R	repetativen - vsebuje konstrukte za ponavljanje (npr. zanke)
ND	omogoča nedeterministično programiranje.

Značilnost definicijskih jezikov je t. i. pravilo enkratne povezave. Zaradi tega pravila so definicijski jeziki zelo primerni za verifikacijo programov in za programiranje podatkovno vodenih računalnikov.

Logični jeziki omogočajo paralelno in logično programiranje. Predvideni so za programiranje računalnikov pete generacije in tudi za podatkovno vodene računalnike.

Objektni jeziki uporabljajo kot osnovo t.i. sporočilo /objekt - model in ne bolj znan operator/ operand - model. V objektnih jezikih definiramo objekte ali razrede objektov (ti imajo skupne lastnosti). Objekti vsebujejo tako deklaracijo (opis) podatkov kot postopke za obdelavo teh podatkov. Če hočemo izvršiti kakšno operacijo, pošljemo določenemu objektu sporočilo, ta izvede ustrezno akcijo ali pošlje sporočilo drugemu objektu.

LISP-ovski in čisti funkcijski jeziki vsebujejo tri vrste struktur: procedure, izraze, spremenljivke in konstante. Glavne značilnosti funkcijskih jezikov so:
- ne vsebujejo ponavljalnih struktur, prireditvenih stavkov in spremenljivk,
- preslikujejo objekte v objekte (teh objektov ne smemo zamenjati z objekti v objektnih jezikih).

Čisti funkcijski jeziki so zelo primerni za programiranje podatkovno vodenih računalnikov kakor tudi za druge arhitekture prihodnosti kot so Magov /8/ računalnik, redukcijski računalnik idr.

Programiranje v redukcijskih jezikih je dosledno funkcijsko in temelji na enostavnih matematičnih konstruktih, ki predstavljajo strukturo binarnih dreves, iz katerih z rekurzivno uporabo sestavljamo kompleksnejše izraze. Redukcijske jezike uporabljamo predvsem za programiranje redukcijskih računalnikov.

Vsi trije tipi jezikov pretoka podatkov (JPP) so si zelo podobni v svojih osnovnih lastnostih. Vsi predstavljajo abstrakcijo grafa pretoka podatkov za neki algoritem. Razlika med grafičnim in visokim JPP je podobna kot razlika med opisom algoritma v psevdokodu in opisom algoritma z grafičnimi pripomočki, kot so npr. diagrami poteka. Kodirni JPP pa je s posebnimi instrukcijami opisan graf pretoka podatkov.

S stališča verifikacije, dokumentacije, preglednosti in obsežnosti (programa, ki obsega samo nekaj vrstic teksta v visokem JPP, bi obsegal več strani in množico povezav v grafičnem JPP) mislimo, da je iz trojice jezikov pretoka podatkov visoki JPP najboljši.

Visoki JPP vsebuje elemente (sintakso) postopkovnih jezikov in semantiko čistih funkcijskih ter definicijskih jezikov. Uporabljamo jih za programiranje podatkovno vodenih računalnikov in za avtomatično programiranje.

3. LASTNOSTI JEZIKA PRIMERNEGA ZA PODATKOVNO VODENI RAČUNALNIK (PVR)

Da bi jezik ustrezal podatkovno vodenim arhitekturam, mora ugoditi zahtevama:

I. Pretok podatkov se mora ugotoviti iz operacij programa.
II. Potek izvajanja mora biti enak pretoku podatkov.

Kadar programski jezik ustreza gornjima zahtevama, mora imeti naslednje lastnosti:

a) lokalnost efektov,
b) ni stranskih efektov,
c) pravilo enkratne prireditve,
d) posebni konstrukti za iteracije,
e) ni nestrukturiranih konstruktov,
f) ni zgodovinsko občutljiv,
g) ni globalnega okolja,
h) ni spremenljivk (imena so le označbe vrednosti)
i) primeren za arhitekture z asinhronim proženjem.

Zgornje lastnosti omogočajo:
- enostavnejšo verifikacijo programov,
- paralelnosti se ugotavljajo interproceduralno (lokalna paralelnost),
- paralelnosti se ugotavljajo na algoritemskem nivoju (program je zelo dobra abstrakcija algoritma),
- jezik je bolj razumljiv uporabnikom in predvsem začetnikom,
- graf pretoka podatkov je naravna preslikava programa,
- ustreza VLSI arhitekturam.

Vrednosti in imena

Jeziki, primerni za programiranje PVR morajo biti vrednostno orientirani, kar pomeni, da v programih uporabljajmo zgolj vrednosti, imena pa vam predstavljajo le označbe teh vrednosti.

Lokalnost efektov

V postopkovnih jezikih je mogoče ista imena za spremenljivke uporabljati v različne namene - v različnih delih programa.

Če ne bi bilo enakosti imen, bi te dele programa lahko izvajali paralelno. Odkrivanje takšnih paralelnosti je mogoče, vendar je enostavneje omejiti delovanje imen le na neko določeno območje programa in kontrolirati vhode in izhode tega območja. Še bolje je, če se izognemo globalnim imenom, proceduram dovolimo le dostop do svojih parametrov in strukturiramo programe v bloke. Lokalnost efektov torej onemogoča odvisnosti podatkov, ki so v različnih blokih.

Stranski efekti

Odsotnost stranskih efektov je nujen pogoj za enakost med pretokom podatkov in potekom izvajanja. Medtem ko lokalnost efektov zahteva le nekatere manjše spremembe v jeziku, odsotnost stranskih efektov zahteva spremembo načina obdelave podatkov. Zaradi tega prepovemo globalna imena in dodamo pravilo, da procedura ne sme spremeniti ničesar - niti svojih vhodnih parametrov. S tem odstranimo enostavne stranske efekte, ostanejo pa stranski efekti, ki nastanejo zaradi obdelave sestavljenih podatkovnih struktur (npr. polja ali zapisi). Pri teh je namreč skoraj nemogoče ugotoviti, kakšne odvisnosti podatkov povzroči sprememba posameznega elementa sestavljene podatkovne strukture. V takšnem primeru je treba vzeti največjo možno odvisnost podatkov, s čimer zelo zmanjšamo možnost paralelnega izvajanja. Temu se izognemo, da obravnavamo sestavljene strukture enako kot skalarne, ter da v procedurah parametre kličemo z vrednostjo in ne z referenco. Posledica tega je, da moramo pri spremembi vrednosti posameznega elementa sestavljene strukture zgraditi celotno novo strukturo.

Pravilo enkratne prireditve

V prejšnjih razdelkih smo se odločili za vrednostno orientiran način programiranja. V tem načinu prireditveni stavek ne pomeni več prireditve vrednosti spremenljivki, ampak zgolj povezavo med imenom na levi in vrednostjo na desni strani prireditvenega stavka. S tem postane program enak sistemu matematičnih enačb, kajti kjerkoli v programu lahko zamenjamo izraz na desni z ustreznim imenom na levi strani, ne da bi s tem kakorkoli spremenili pomen programa. Postavimo še naslednji pravili:
P1: ime se mora povezati prej preden ga uporabimo,
P2: ime lahko povežemo samo enkrat.

S pravili P 1 in P 2 in vrednostno orientiranim načinom programiranja dosežemo, da v programih ni spremenljivk in da je prireditveni stavek kar definicija imena.

Iteracije

Zaradi pravil P 1 in P 2 se pojavijo problemi pri iteracijah, ki uporabljajo prireditvene stavke oblike:

I : = I op J

kjer so:

I ..... iteracijsko ime
J ..... ime
op ..... operator.

Ta stavek je v nasprotju s pravilom P 1, ker zaradi pravila P 2 I uporabljamo prej, preden ga definiramo. Vendar so stavki takšne oblike zaradi iteracij potrebni. Prepoved uporabe nestrukturiranih jezikovnih konstruktov nam omogoča, da z ustrezno prireditvijo while-do zanke omogočimo iteracije, ne da bi kršili ostale lastnosti.

Tako prirejena while-do zanka vsebuje:
(1) deklaracijo in inicializacijo iteracijskih vrednosti,
(2) test, če naj se zanka konča,
(3a) če se zanka konča, definicijo izhodnih vrednosti,
(3b) če se zanka ne konča, izraze, ki generirajo nove vrednosti iteracijskih imen.

Čeprav se iteracijske vrednosti spreminjajo, se to zgodi le v prehodu iz ene iteracije v drugo, tako da pravilo enkratne prireditve še vedno velja, četudi samo v sklopu ene iteracije.

Zgodovinsko občutljivo računanje

Ena pomembnih lastnosti, ki jih mora imeti programski jezik primeren za podatkovno vodene računalnike, je zgodovinska neobčutljivost. Zgodovinska neobčutljivost pomeni, da procedure ne vsebujejo spremenljivk, v katere bi shranile podatke med posameznimi klici procedur. To pomeni, da procedure nimajo zmožnosti "pomnjenja", kar v sklopu z ostalimi lastnostmi predstavlja veliko težavo pri računanju v realnem času.

Prepoved globalnega okolja in nestrukturiranih konstruktov

Uporaba globalnega okolja in nestrukturiranih konstruktov nam oteži ali celo onemogoča odkrivanje paralelnosti in analizo pretoka podatkov.

4. VREDNOTENJE JEZIKOV

V tem razdelku bomo ovrednotili jezike glede na lasnosti, ki jih mora imeti programski jezik za učinkovito programiranje podatkovno vodenega računalnika. Poleg navedenih lastnosti v razdelku 3 smo ocenjevali še dve pomembni splošni lastnosti:
- preglednost programa in
- možnost matematične verifikacije programov.

Pri vrednotenju jezikov upoštevamo naslednje ocene:

+ . jezik ima potrebno lastnost,
Ø jezik lahko ima potrebno lastnost, vendar se to od jezika ne zahteva,
- jezik nima potrebne lastnosti.

Spodnjo mejo primernosti dobimo tako, da seštejemo lastnosti označene s +, zgornjo pa tako, da spodnji meji prištejemo lastnosti označene z Ø.

Tabela 2 vrednoti posamezne jezike z upoštevanjem lastnosti od a do i.

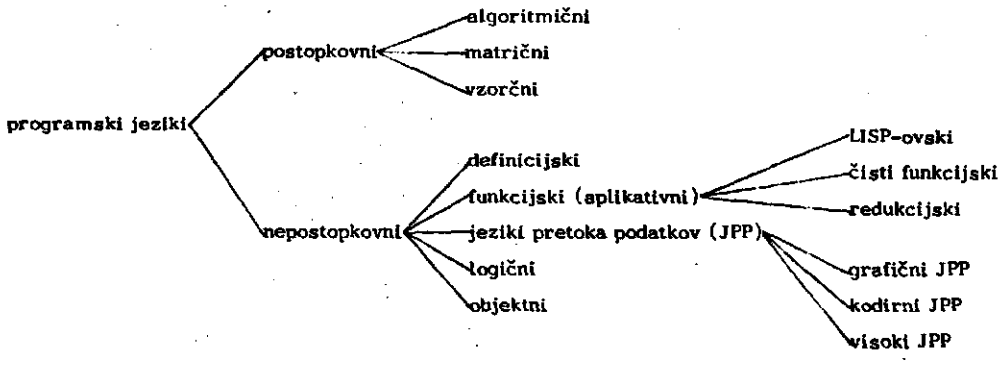
Vrednotenje logičnih in objektnih jezikov ni realno, ker se struktura njihovih predstavnikov (Prolog, Smalltalk) ne ujema popolnoma s shemo našega vrednotenja. Iz tabele vidimo, da čisti funkcijski jeziki, redukcijski jeziki in visoki JPP vsebujejo vse zahtevane lastnosti. Visoki JPP so po sintaksi jezika bližji postopkovnim jezikom kot funkcijski ali redukcijski jeziki in zaradi tega bolj sprejemljivi za uporabnike vajene postopkovnega stila programiranja.

5. ZAKLJUČEK

V članku smo na kratko opisali jezike, ki so primerni za programiranje podatkovno vodenih arhitektur. Programske jezike smo razdelili v več skupin in ocenili njihove lastnosti.

LITERATURA

1. Data - flow computers, IEEE Computer, Vol. 15, No. 2, Februar 1982.
2. Ackerman, W. B. Dennis J. B., VAL-A Value oriented Algorithmic Language, Preliminary Reference Manual, MIT Laboratory for CS, 1979.
3. Japanese Computer Technology Culture, IEEE Computer, Vol. 17, No. 3, Marec 1984.
4. Treleaven, P. C., i.dr., Data - Driven and Demand - - Driven Computer arhitecture, Computing Surveys, Vol. 14, No. 1, Marec 1982.
5. Lerner, J. E., Data Flow arhitecture, IEEE Spectrum, Vol. 21, No. 14,April 1984.
6. Kokol, P., Stiglic, B., Žumer, V., Pretvorba aplikativnega jezika v grafe pretoka podatkov, ETAN 1984, Split.
7. Turner, J. D., Sequin, C., Discusion on Very High Level Languages.
8. Backus, J., Function - level computing, IEEE Spectrum, Vol. 19, No. 8, August 1982.
9. Backus, J., Can Programming Be Liberated from the von Neumman Style?, Communications on the ACM, Vol. 21, No. 8, August 1978.
10. Cox, J. B., Message Object programming, An Evelutionary Change in Programing Technology, IEEE Software, Vol. 1, No. 1, January 1984.
11. O'Keete, R. A., Prolog compared with LISP?, SIGPLAN notices, Vol. 18, No. 5, Maj 1983
12. Fifth Generation Computer Systems, edited by T. Moto - oka, North Holland, Pub. Co., 1982.
13. Berkling, K. J., Reduction Languages for Reduction Machines, Proc. Second, Int. Symp. on Computer Architecture, April 1975.
14. Flynn, M. J., Hennessy, J. L. Parallelism and Representation Problems in Distributed Systems, IEEE Trans. on Computers, Vol. 29, No. 12, December 1980.
15. Hoare, C. A. R., Programing: Soucery or Science, IEEE Software, Vol. 1, No. 2., April 1984.
16. Kokol P., Aplikativni jeziki in njihova interpretacija v realnem času, Magistrska naloga, TF - Maribor.



Slika 1: Delitev programskih jezikov