prace naukowo-badawcze

Instytutu Maszyn Matematycznych

S. JARZĄBEK: System automatycznej konstrukcji optymalizatorów dla pewnej klasy języków pośrednich

A. ROWICKI: Realizacja maszynowa algorytmu planowania obliczeń dla systemów dwuprocesorowych

A. ROWICKI: Planowanie obliczeń zależnych podzielnych dla systemów dwuprocesorowych

A. ROWICKI: Planowanie obliczeń niezależnych podzielnych dla systemów wieloprocesorowych jednorodnych

Zjednoczenie Przemysłu Automatyki i Aparatury Pomiarowej "MERA"

I n s t y t u t M a s z y n M a t e m a t y c z n y c h

p r a c e n a u k o w o - b a d a w c z e
I n s t y t u t u
M a s z y n
M a t e m a t y c z n y c h

Nr [2]

w zeszycie zamieszczono

S. Jarząbek: System automatycznej konstrukcji optymalizatorów dla pewnej klasy języków pośrednich. 3

A. Rowicki: Realizacja maszynowa algorytmu planowania obliczeń dla systemów dwuprocesorowych. 83

A. Rowicki: Planowanie obliczeń zależnych podzielnych dla systemów dwuprocesorowych. 100

A. Rowicki: Planowanie obliczeń niezależnych podzielnych dla systemów wieloprocesorowych jednorodnych. 118

W a r s z a w a 1980

Adres Redakcji: Instytut Maszyn Matematycznych
Branżowy Ośrodek INTE
ul. Krzywickiego 34, 02-078 Warszawa
tel. 28-37-29 lub 21-84-41 w. 244

Prace naukowo-badawcze Instytutu Maszyn Matematycznych

SYSTEM AUTOMATYCZNEJ KONSTRUKCJI OPTYMALIZATORÓW DLA PEWNEJ KLASY JĘZYKÓW POŚREDNICH

Stanisław JARZĄBEK

Praca doktorska wykonana
pod kierunkiem prof. dr hab.
Zdzisława Pawlaka

Tematem pracy jest konstrukcja Systemu Optymalizacyjnego pozwalającego na automatyczną generację optymalizatorów dla pewnej klasy języków pośrednich kompilatorów, a także rozwiązania zagadnień teoretycznych powstałych w trakcie realizacji pomysłu. System Optymalizacyjny składa się z 2 części: Konstruktora i Optymalizatora Podstawowego. Idea działania jest następująca: w celu otrzymania optymalizatora dla konkretnego języka pośredniego, powiedzmy L, należy przygotować odpowiedni opis tego języka i podać go na wejście Konstruktora. Na podstawie opisu Konstruktor modyfikuje program Optymalizatora Podstawowego w taki sposób, że w wyniku powstaje optymalizator programów języka L.

Spis treści

str.

WSTĘP 5

Rozdział I. POJĘCIA WSTĘPNE 9

1. Oznaczenia 9
2. Podstawowe definicje z zakresu teorii grafów 9
3. Pojęcia dominacji i jego własności 11
4. Pętla i region w grafie 12

Rozdział II. JĘZYKI I PROGRAMY 14

1. Systemy programowania 14
2. Programy 18
3. Podział programu na bloki i graf programu 20

Rozdział III. JĘZYKI Z PROCEDURAMI I ICH PROGRAMY 22

1. Systemy programowania z procedurami 22
2. Programy z procedurami 25
3. Programy zupełne 26
4. Synonimy zmiennych 28
5. Definicja zmiennej i odwołanie do zmiennej w programie z procedurami 32

Rozdział IV. METODY OPTYMALIZACJI PROGRAMÓW 36

1. Przekształcenia optymalizacyjne programów 36
2. Usuwanie niepotrzebnych instrukcji z programu 39
3. Redukcja identycznych wyrażeń 40
4. Wynoszenie instrukcji poza pętle 41
5. Usuwanie prostych przypisań 43

Rozdział V. SYSTEM AUTOMATYCZNEJ KONSTRUKCJI OPTYMALIZATORÓW DLA JĘZYKÓW POŚREDNICH Z PROCEDURAMI 43

1. Wejście konstruktora systemu optymalizacyjnego 44
2. Optymalizator 48
3. Dyskusja wyniku i uwagi o implementacji systemu optymalizacyjnego 58

ZAKOŃCZENIE 61

DODATEK A. Przykład ilustrujący działanie systemu optymalizacyjnego 62

DODATEK B. Oszacowanie kosztu czasowego procesu optymalizacji dla optymalizatorów produkowanych przez system optymalizacyjny .. 78

LITERATURA 81

WSTĘP

Podstawowym narzędziem komunikowania się człowieka z maszyną cyfrową są języki programowania wysokiego rzędu. Mają one w porównaniu z językami wewnętrznymi wiele zalet: najważniejszą jest fakt, że są łatwe do nauczenia i może się nimi posługiwać prawie każdy, niekoniecznie matematyk. Ponadto programy pisane w tych językach odznaczają się większą czytelnością i przy stosunkowo niewielkich zmianach mogą być uruchamiane na maszynach różnych typów, wyposażonych w odpowiedni translator. Z kolei wadą języków wysokiego rzędu jest to, że ich programy są na ogół mniej efektywne od programów pisanych w językach wewnętrznych. Aby usunąć, albo przynajmniej zmniejszyć wagę tej niedoskonałości, programy języków wysokiego rzędu poddaje się w czasie translacji procesowi automatycznego zwiększania efektywności, tak zwanej optymalizacji.

Translator przewidujący fazę optymalizacji działa według następującego schematu: najpierw program zostaje poddany analizie syntaktycznej i semantycznej w wyniku czego powstaje program w tzw. języku pośrednim. Program ten stanowi przedmiot optymalizacji, po czym odbywa się generacja kodu. Fragment translatora, którego zadaniem jest optymalizacja programu nazywamy optymalizatorem.

Na ogół mówi się o dwóch rodzajach optymalizacji: zależnej od maszyny i niezależnej od maszyny. Optymalizacja zależna od maszyny wykorzystuje cechy konkretnej maszyny /na przykład liczbę jej rejestrów/ i zajmuje się między innymi rozsądnym przydziałem rejestrów zmiennym programu.

Nas będzie interesował wyłącznie drugi rodzaj optymalizacji, do którego zalicza się takie przekształcenia programów jak: wynoszenie instrukcji poza pętle, redukcja identycznych wyrażeń, usuwanie niepotrzebnych instrukcji.

Badania nad optymalizacją programów rozwijają się w trzech kierunkach. Pierwszy to próby formalnego ujęcia przekształceń optymalizujących. W pracach Aho i Ullmana [2] oraz Matwina [14] autorzy definiują szereg przekształceń dla modeli programów liniowych (tzn. programów bez pętli). Drugi kierunek obejmuje metody zbierania o programie informacji istotnych dla optymalizacji. Należy tu zwrócić uwagę na prace Cocke'a [9] i Allena [5], których wynikiem jest metoda analizy grafu programu opierająca się na tzw. interwałach, pozwalająca określić powiązania między definicjami zmiennych programu i odwołaniami do nich. Wreszcie ostatni kierunek to projektowanie optymalizatorów dla konkretnych translatorów i ich implementacja. Lowery i Medlock [13] opisują optymalizator zastosowany w translatorze OS/360 FORTRAN H, a Allen [4] omawia optymalizator innego translatora języka FORTRAN.

Omówimy teraz zasadniczy problem jaki podjęliśmy w niniejszej pracy.

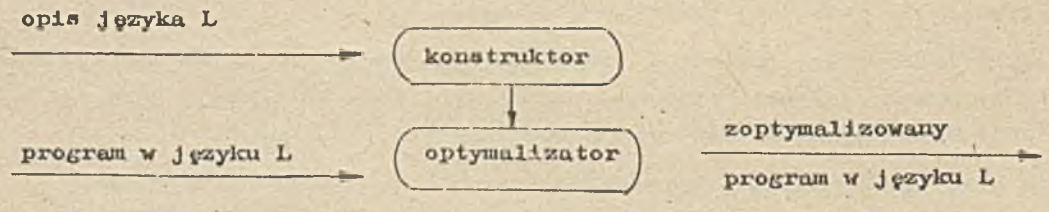
Spośród wielu rodzajów języków pośrednich stosowanych w translatorach (czwórki, trójki, zapis polski, itd) dla optymalizacji najdogodniejsze są języki czwórek (patrz Gries [11]). Taki język został wykorzystany np. w translatorze FORTRANu H, opisanym przez Lowery i Medlocka w [13]. Języki czwórek jakie spotykamy w różnych translatorach są podobne, ale nie identyczne. Przyczyny różnic

pomiędzy poszczególnymi ich wersjami są dwojakiego rodzaju: (1) brak sformalizowanej definicji języka ozwórek (assemblery nie wchodzą tutaj w rachubę, ponieważ dopuszczają operowanie na adresach, co uniemożliwia skuteczną optymalizację); (2) konstruktor pisząc translator konkretnego języka źródłowego na określoną maszynę dostosowuje język pośredni do potrzeb języka źródłowego i maszyny. W związku z tymi różnicami w dotychczasowych rozwiązaniach projektowano każdy optymalizator dla jednego translatora i jego języka pośredniego - bez możliwości wykorzystania tego optymalizatora w innych translatorach.

Można jednak zauważyć, że struktura optymalizatorów dla różnych języków ozwórek jest zasadniczo identyczna, że da się wydzielić i opisać fragmenty związane z możliwymi różnicami występującymi pomiędzy różnymi wersjami tych języków. To spostrzeżenie skłoniło nas do postawienia następującego problemu: czy można zbudować uniwersalny optymalizator dla programów napisanych w dowolnych językach ozwórek.

Niniejsza praca daje pozytywną odpowiedź na to pytanie, proponując jako rozwiązanie tzw. system optymalizacyjny.

System optymalizacyjny składa się z dwóch części: konstruktora i optymalizatora. Idea działania systemu jest następująca: wszystkie informacje związane z konkretnym językiem ozwórek, w formie specjalnego opisu, stanowią dane wejściowe konstruktora. Zadaniem konstruktora jest zmodyfikowanie optymalizatora w taki sposób, aby mógł on optymalizować programy pisane w tym konkretnym języku. Na schemacie można to wyrazić w następujący sposób:



Schemat systemu optymalizacyjnego

Językiem, w którym opisuje się konkretny język ozwórek (wejście konstruktora) jest PL/I. Opis ten zawiera jedną tablicę oraz deklaracje około 20 tak zwanych procedur standardowych. Konstruktor włącza ciąg instrukcji języka PL/I, tworzący opis konkretnego języka ozwórek (powiedzmy, L) do programu optymalizatora, umożliwiając mu wywołanie procedur standardowych oraz korzystanie z informacji zawartych w tablicy. Tak zmodyfikowany optymalizator jest w stanie optymalizować programy pisane w języku L.

Projekt systemu optymalizacyjnego powstał w toku prac prowadzonych pod kierunkiem doc. Jana Borowca nad Systemem Produkcji Translatorów, w Zakładzie Teorii Translatorów IMM. W przyszłości system optymalizacyjny zostanie włączony jako moduł do wspomnianego metatranslatora.

Oprócz aspektu teoretycznego cechą proponowanego tu rozwiązania jest jego użyteczność: zastosowanie systemu optymalizacyjnego pozwala konstruktorowi translatora uzyskać optymalizator kosztem napisania około 200 zdań języka PL/I (tyle wynosi na ogół opis języka pośredniego potrzebny dla konstruktora), podczas gdy

opracowanie od podstaw programu optymalizatora odpowiadającego złożonością naszemu, obejmuje około 3,5 tysiąca zdań języka PL/I.

Głównym wynikiem pracy (rozdział V) jest projekt systemu optymalizacyjnego i jego realizacja w postaci programów napisanych w języku PL/I, uruchomionych na maszynie IBM/370.

Celem teoretycznej części pracy jest wprowadzenie szeregu pojęć i przedstawienie - z ich pomocą - rozwiązań problemów, na jakie natrafiliśmy w trakcie realizacji systemu optymalizacyjnego. Zaczynamy od określenia modelu dla języków czwórek (tzn. zdefiniowania klasy języków, dla których przeznaczony jest system optymalizacyjny).

Następnie podajemy efektywną metodę znajdowania zbiorów zmiennych otrzymujących nowe wartości i zmiennych, których wartości są wykorzystywane w instrukcjach programu z procedurami. Problem wyznaczenia tych zbiorów wyłonił się w trakcie opracowywania systemu optymalizacyjnego. Dla programów nie zawierających procedur rekurencyjnych zagadnienie to zostało rozwiązane przez Allena [7]. Proponowane tu rozwiązanie (algorytm 2 z rozdziału III) jest wolne od tego ograniczenia i wzbogacone o analizę powiązań pomiędzy parametrami procedur wołanymi przez nazwę i odpowiadającymi im parametrami aktualnymi. Badanie tych powiązań prowadzące do określenia tzw. synonimów zmiennych (algorytm 1 z rozdziału III), jest istotne dla pełnego rozwiązania problemu.

Na tak przygotowanym gruncie definiujemy interesującą nas ze względu na system optymalizacyjny grupę przekształceń optymalizacyjnych. W dotychczasowych badaniach (Aho i Ullman [2], Matwin [14]) zajmowano się sformalizowaniem podobnych przekształceń jedynie dla tzw. programów liniowych (tzn. bez pętli i procedur). Ponieważ chcieliśmy przedstawić tu przekształcenia optymalizacyjne w takiej postaci, w jakiej stosowaliśmy je w systemie optymalizacyjnym, musieliśmy je zdefiniować dla programów pozbawionych tych ograniczeń, tzn. zawierających pętle i procedury (rozdział IV). Tą ogólnością należy też tłumaczyć fakt, że przekształcenia wyrażają się w tak skomplikowanej formie.

W niniejszej pracy postawiliśmy sobie za zadanie zrealizować i opisać system optymalizacyjny. Zamierzenie to okazało się na tyle obszerne, że z konieczności niektóre aspekty zagadnień związanych z optymalizacją musieliśmy w części teoretycznej pominąć. I tak, na przykład, nie podjęliśmy w pracy rozwiązania problemu poprawności zdefiniowanych przekształceń. Wprowadzenie aparatu semantycznego dla naszego modelu języków, na gruncie którego można by udowodnić, że przekształcenia zachowają równoważność i rzeczywiście optymalizują programy, jest sprawą niebanalną: rozpatrujemy bowiem języki z procedurami, z parametrami wołanymi przez nazwę i wartość. Ponadto trzeba by uwzględnić obecność zmiennych lokalnych. Temat ten, interesujący i niewątpliwie potrzebny, powinien zostać w przyszłości podjęty. W systemie optymalizacyjnym korzystaliśmy co prawda wyłącznie z przekształceń optymalizacyjnych dobrze znanych z praktyki, jednakże formalne uzasadnienie tych przekształceń i algorytmów zwiększyłoby zaufanie do systemu.

Plan pracy jest następujący: pierwszy rozdział zawiera podstawowe wiadomości z teorii grafów. W drugim rozdziale wprowadzamy model języków pośrednich, który w następnym rozdziale uzupełniamy procedurami. W trzecim rozdziale rozwiązu-

jemy ponadto problem definicji zmiennej i odwołania do zmiennej w programach z procedurami. Czwarty rozdział zawiera definicje przekształceń optymalizacyjnych dla programów. Piąty rozdział stanowi opis systemu optymalizacyjnego i zawiera uwagi o jego implementacji i dyskusję wyniku. Pracę zamyka dodatek, w którym podajemy przykład opisu konkretnego języka czwórek, na podstawie którego system optymalizacyjny buduje optymalizator oraz przykład działania uzyskanego optymalizatora dla programu tego języka.

ROZDZIAŁ I. POJĘCIA WSTĘPNE

1. Oznaczenia

Zbiory będziemy oznaczali dużymi literami, a ich elementy małymi (ewentualnie ze wskaźnikiem) . W szczególności przez N oznaczamy zbiór liczb naturalnych i przez $\emptyset$ zbiór pusty. Elementy zbioru N będziemy oznaczali literami i, j, k, l, m, n. Dla dowolnych liczb naturalnych n, m takich, że $n \leqslant m$, przez $<n,m>$ oznaczymy podzbiór liczb naturalnych $\{k \in N \mid n \leqslant k \leqslant m\}$.

Dla dowolnego zbioru A przez A^* oznaczymy zbiór wszystkich skończonych ciągów o elementach z A łącznie z symbolem pustym, który oznaczymy przez ε. Przez A^k, gdzie $k \in N$, będziemy oznaczali zbiór wszystkich ciągów długości k o elementach z A .

Dla dowolnych zbiorów A i B przez $A \cup B$, $A \cap B$ i $A \setminus B$ oznaczymy odpowiednio sumę, iloczyn i różnicę zbiorów A i B. Będziemy również używali zapisów $\bigcup_{i=1}^{n} A_i$ i $\bigcap_{i=1}^{n} A_i$ dla oznaczenia odpowiednio sumy i iloczynu zbiorów $A_1,\ldots A_n$.

Dla dowolnych zbiorów $A_1,\ldots A_n$ przez $A_1 \times \ldots \times A_n$ będziemy oznaczali produkt kartezjański zbiorów $A_1,\ldots,A_n$, tzn. zbiór wszystkich uporządkowanych n-tek $(a_1,\ldots,a_n)$ takich, że $a_i \in A_i$ dla $i \in <1,n>$.

Relacją w produkcie kartezjańskim $A_1 \times \ldots \times A_n$ nazwiemy każdy podzbiór tego produktu. W szczególności, gdy n=2, będziemy mówili o relacji binarnej.

Relację binarną Γ w produkcie $A \times B$ nazwiemy funkcją, jeśli dla każdego $a \in A$ istnieje dokładnie jeden element $d \in B$ taki, że $(a,b) \in \Gamma$.

Dla dowolnej relacji binarnej Γ w produkcie $A \times B$ zbiór $\{(a,b) \mid (b,a) \in \Gamma\}$ będziemy nazywali relacją przeciwną do Γ i oznaczali przez Γ^{-1}.

Niech $A = A_1 \times \ldots \times A_n$ będzie dowolnym produktem kartezjańskim. Dla $i \in <1,n>$ rzutem i-tej współrzędnej produktu A będziemy nazywali funkcję φ_{1i} odwzorowującą produkt A w zbiór A_i i określoną w następujący sposób:

$$\varphi_{1i}(a_1,\ldots,a_n) = a_i \quad \text{dla dowolnej n-tki } (a_1,\ldots,a_n) \in A.$$

2. Podstawowe definicje z zakresu teorii grafów

Teoria grafów znajduje zastosowanie w wielu dziedzinach nauki i w związku z tym posiada bogatą literaturę. W zależności od potrzeb wprowadza się różne definicje grafów, my posłużymy się następującą:

Definicja 1. Graf zorientowany jest to uporządkowana para $G = (X,\Gamma)$, gdzie X jest zbiorem elementów zwanych węzłami, a Γ jest relacją binarną w produkcie $X \times X$.

Graf o skończonej liczbie węzłów nazwiemy grafem skończonym i w dalszym ciągu pracy, mówiąc o grafie, zawsze będziemy mieli na myśli graf zorientowany skończony. Dla dowolnego grafu G przez X_G będziemy oznaczali zbiór węzłów grafu G, a przez Γ_G - relację opisującą graf.

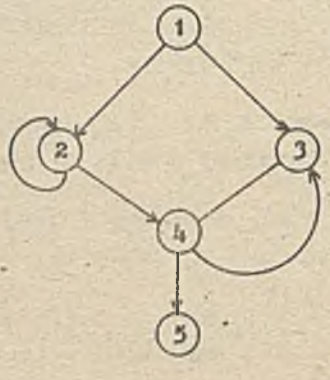
Definicja 2. Niech G będzie grafem. Dla dowolnej pary $(x,y) \in \Gamma_G$ powiemy, że węzeł x jest bezpośrednim poprzednikiem węzła y, natomiast o y powiemy, że jest bezpośrednim następnikiem węzła x.

Definicja 3. Drogą w grafie G, o początku x, końcu y i długości l, będziemy nazywali każdy niepusty ciąg węzłów $d = x_1,\ldots,x_l$ taki, że $x_1 = x$, $x_l = y$ oraz dla każdego $i \in \langle 1, l-1 \rangle$ węzeł x_i jest bezpośrednim poprzednikiem węzła x_{i+1}. Jeśli ten ciąg będzie różnoelementowy, to nazwiemy go drogą prostą. O każdym węźle z będącym elementem drogi d w grafie powiemy, że z leży na drodze d lub, że droga d przechodzi przez węzeł z.

Definicja 4. Dla dowolnych węzłów x i y grafu G powiemy, że x jest poprzednikiem węzła y oraz y jest następnikiem węzła x, jeśli istnieje droga w G o początku x, końcu y i długości większej niż 1.

Powyżej wprowadzone pojęcia zilustrujemy na przykładzie:

Przykład 1



Rys. 1

$$G = \left(\{1,2,3,4,5\}, \{(1,2), (1,3), (2,2), (2,4), (3,4), (4,3), (4,5)\} \right)$$

Węzeł 4 ma dwa bezpośrednie poprzedniki: 2 i 3 oraz dwa bezpośrednie następniki: 3 i 5. Poprzednikiem węzła 4 są wszystkie węzły grafu z wyjątkiem 5, a następnikami węzły 3,4 i 5.

Drogą w grafie G jest, na przykład, ciąg węzłów d=2,4,3. Początkiem drogi jest węzeł 2, końcem - węzeł 3, droga ma długość równą trzy. Jest to zarazem przykład drogi prostej w grafie.

Dotychczasowa definicja grafu jest dla naszych dalszych rozważań zbyt ogólna. Obecnie wprowadzimy bardziej dla nas dogodne pojęcie grafu spójnego. W literaturze można spotkać różne definicje grafu spójnego, my przyjmiemy następującą:

Definicja 5. Graf spójny jest to graf zorientowany $G = (X, \Gamma)$, w którym jest wyróżniony dokładnie jeden węzeł $\xi \in X$ spełniający warunki:

1) węzeł ξ nie ma poprzedników w G,
2) dla każdego węzła $x \in X$ różnego od ξ istnieje droga w G o początku i końcu x.

Grafy spójne będziemy przedstawiali jako uporządkowane trójki $G = (X, \Gamma, \xi)$, a wyróżniony węzeł ξ będziemy nazywali węzłem początkowym grafu.

Dla dowolnego spójnego grafu G przez X_G będziemy oznaczali zbiór węzłów grafu, węzeł początkowy przez ξ_G, a relację opisującą graf przez Γ_G. Wszystkie pojęcia wprowadzone w definicjach 2-4 przenoszą się bez żadnych zmian na grafy spójne. Graf z przykładu 1, w którym przyjmiemy jako początkowy węzeł 1, jest grafem spójnym.

W dalszym ciągu pracy, mówiąc o grafie, zawsze będziemy mieli na myśli graf spójny.

3. Pojęcie dominacji i jego właściwości

Obecnie omówimy pojęcie dominacji węzłów w grafie, bardzo ważne dla naszych dalszych rozważań. Jako pierwszy określił dominację węzłów Prosser [17] . Zastosowanie tego pojęcia w optymalizacji rozwijali następnie Gries [12] , Lowery i Medlock [13] , Aho i Ullman [3] .

Definicja 6. Dla dowolnych węzłów x i y grafu G powiemy, że węzeł y dominuje nad węzłem x (albo: y jest dominatorem x), jeśli $y \neq x$ i y leży na każdej drodze w G o początku ξ_G i końcu x.

Definicja 7. Dla dowolnych węzłów x i y grafu G powiemy, że y bezpośrednio dominuje nad węzłem x (albo: y jest bezpośrednim dominatorem x) , jeśli:

1) y dominuje nad x oraz
2) dla każdego $z \in X_G$ takiego, że $z \neq y$ z tego, że z dominuje nad x wynika, że z dominuje nad y.

W grafie z przykładu 1 węzeł 5 ma dwa dominatory: 1 i 4, przy czym 4 jest jego bezpośrednim dominatorem.

Pojęcia dominacji i bezpośredniej dominacji wyznaczają relacje binarne w produkcie $X_G \times X_G$, mianowicie $\{(x,y) \in X_G \times X_G \mid$ x jest dominatorem y$\}$ i $\{(x,y) \in X_G \times X_G \mid$ jest bezpośrednim dominatorem y$\}$. Relacje te będziemy nazywali odpowiednio relacją dominacji i relacją bezpośredniej dominacji.

Poniżej podajemy szereg właściwości relacji dominacji i bezpośredniej dominacji. Dowody właściwości znajdują się u Aho i Ullmana [3] i ponieważ są prawie natychmiastowe, więc nie będziemy ich przytaczali.
Niech G będzie grafem i x,y,z dowolnymi węzłami grafu G.

Właściwość 1. Węzeł początkowy ξ_G nie ma dominatora, natomiast sam dominuje nad każdym innym węzłem grafu.

Właściwość 2. Jeśli x dominuje nad y i y dominuje nad z, to x dominuje nad z.

Właściwość 3. Jeśli x dominuje nad y, to y nie dominuje nad x.

Właściwość 4. Jeśli x i y dominują nad z, to albo x dominuje nad y albo y dominuje nad x.

Właściwość 5. Każdy węzeł grafu z wyjątkiem początkowego ma dokładnie jeden bezpośredni dominator.

4. Pętla i region w grafie

Podstawowymi strukturami w grafie jakimi będziemy się posługiwali są pętle i regiony. Podobne do naszych określenia tych obiektów znajdują się u Griesa [11] oraz u Aho i Ullmana [3].

Definicja 8. Pętlą w grafie G nazwiemy każdy podzbiór R zbioru X_G taki, że dla dowolnych $x,y \in R$ istnieje droga $x_1,\ldots, x_n$ w G taka, że $n \geq 2$, $x_1 = x$, $x_n = y$ i $x_i \in R$ dla $i \in \langle 1,n \rangle$. Każdy węzeł $x \in R$ posiadający co najmniej jeden bezpośredni poprzednik poza pętlą będziemy nazywali wejściem pętli R, natomiast wyjściem pętli R - każdy węzeł $x \in R$ posiadający co najmniej jeden bezpośredni następnik nie należący do tej pętli.

Graf z przykładu 1 ma dwie pętle: $R_1 = \{2\}$ i $R_2 = \{3,4\}$. Pętla R_1 ma jako wejście i wyjście węzeł 2. Wejściami pętli R_2 są węzły 3 i 4, natomiast wyjściem - węzeł 4.

Właściwość 6. Węzeł początkowy grafu nie jest elementem żadnej pętli grafu.

Właściwość 7. Każda pętla grafu ma przynajmniej jedno wejście.

Powyższe właściwości wynikają natychmiast z definicji grafu i pętli.

Definicja 9. Regionem w grafie G będziemy nazywali każdą pętlę w G posiadającą dokładnie jedno wyjście.

Każdy węzeł x grafu G, dla którego istnieje region w G o wejściu x nazwiemy wejściem regionu w G.

Zanim podamy warunek konieczny i dostateczny jaki musi spełniać dowolny węzeł grafu, na to aby był wejściem regionu, udowodnimy dwa lematy:

Lemat 1. Niech $G = (X, \Gamma, \xi)$ będzie grafem i węzeł x wejściem regionu R w G. Węzeł x dominuje nad każdym innym elementem regionu R.

Dowód nie wprost. Przypuśćmy, że istnieje w R węzeł y różny od x taki, że x nie dominuje nad y. Wówczas istnieje droga w G o początku ξ i końcu y nie przechodząca przez x. Z tego wynika, że w R znajduje się węzeł różny od x posiadający bezpośredni poprzednik poza R, co jest sprzeczne z założeniem, że R jest regionem

c. k. d.

Lemat 2. Niech $G = (X, \Gamma, \xi)$ będzie grafem i x,y dowolnymi węzłami takimi, że x dominuje nad y. W grafie G istnieje droga $z_1,\ldots, z_n$ taka, że $z_1 = x$, $z_n = y$ i węzeł x dominuje nad wszystkimi węzłami z_i dla $i \in \langle 2,n-1 \rangle$.

Dowód. Weźmy dowolną drogę $z_1,\ldots,z_n$ o początku x i końcu y taką, że $z_i \neq x$, dla $i \in \langle 2,n-1 \rangle$. Przypuśćmy, że x nie dominuje nad pewnym węzłem z_k leżącym na

tej drodze. W takim razie istnieje droga w G o początku ξ i końcu z_k nie przechodząca przez x. Ale z tego wynika, że istnieje również droga o początku ξ i końcu y nie przechodząca przez x, co jest sprzeczne z założeniem, że x dominuje nad y

o. k. d.

Poniższe twierdzenie określa warunek konieczny i dostateczny na to, aby węzeł był wejściem regionu w grafie. Podobne twierdzenie zostało udowodnione przez Aho i Ullmana [3] . Zdecydowaliśmy się przytoczyć tu jego dowód ze względu na to, że w dowodzie konstruuje się tzw. maksymalne regiony, którymi będziemy się w dalszym ciągu posługiwali.

Twierdzenie 1. Niech $G = (X, \Gamma, \xi)$ będzie grafem. Dla dowolnego węzła $x \in X$, x jest wejściem regionu w G wtedy i tylko wtedy, gdy istnieje $y \in X$ taki, że $(y,x) \in \Gamma$ i spełniony jest jeden z warunków:

1) $x = y$,
2) x dominuje nad y.

Dowód konieczności. Niech węzeł x będzie wejściem regionu R w grafie G. Jeśli $R = \{x\}$, to $(x,x) \in \Gamma$ i spełniony jest warunek 1. Przypuśćmy teraz, że region R składa się z więcej niż jednego węzła. Wówczas musi istnieć taki węzeł $y \in R$, że $(y,x) \in \Gamma$ i $y \neq x$. Z lematu 1 otrzymujemy, że x dominuje nad y, a więc w tym przypadku spełniony jest warunek 2.

Dowód dostateczności. Jeśli $(x,x) \in \Gamma$, to x jest oczywiście wejściem regionu $R = \{x\}$. Przypuśćmy, że istnieje w R węzeł y taki, że x dominuje nad y. Aby zakończyć dowód wystarczy wskazać region w G o wejściu x. Rozważmy zbiór zdefiniowany w następujący sposób:

$$R = \{x\} \cup \{z \in X \mid x \text{ dominuje nad } z \ \&$$

istnieje droga $z_1, \ldots, z_n$ w G taka, że $z_1 = z$, $z_n = x$ i

$$x \text{ dominuje nad } z_i \text{ dla } i \in \langle 2, n-1 \rangle\}.$$

Zauważmy najpierw, że zbiór R jest więcej niż jednoelementowy, ponieważ oprócz węzła x należy do niego węzeł y. Powyższa uwaga w połączeniu z definicją zbioru R i lematem 2 pozwalają natychmiast stwierdzić, że R jest pętlą w G. Pozostaje wykazać, że węzeł x jest jedynym wejściem pętli R. Przypuśćmy, że wejściem pętli R jest węzeł $z \in R$ różny od x, tzn. istnieje węzeł $z' \in R$ taki, że $(z',z) \in \Gamma$. Ale to oznacza, że x nie dominuje nad z' (w przeciwnym przypadku z' musiałby należeć do R) i również x nie dominuje nad z, Ostatnie stwierdzenie jest sprzeczne z definicją zbioru R, a zatem żaden węzeł różny od x nie może być wejściem pętli R. Ponieważ każda pętla ma przynajmniej jedno wejście (właściwość 7) , więc musi nim być węzeł x

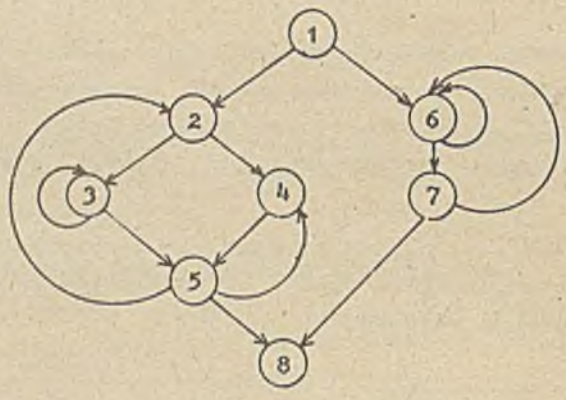
o, k. d.

Region zbudowany w drugiej części dowodu twierdzenia 1 będziemy nazywali <u>maksymalnym regionem w grafie z wejściem</u> x. Następujące dwie właściwości wynikają bezpośrednio ze sposobu konstrukcji maksymalnego regionu:

Właściwość 8. Dla dowolnego węzła x grafu G będącego wejściem regionu w G istnieje dokładnie jeden maksymalny region w G z wejściem x.

Właściwość 9. Dla dowolnych dwóch różnych maksymalnych regionów w G: R_1 i R_2 z tego, że $R_1 \cap R_2 \neq \emptyset$ wynika, że albo $R_1' \subset R_2$ albo $R_2 \subset R_1$.

Przykład 2.



Rys. 2

W grafie G przedstawionym na powyższym rysunku wejściami regionów są węzły 2, 3 i 6. Możemy wyróżnić następujące regiony w G:

$R_1 = \{\underline{2},3,4,5\}$, $R_2 = \{\underline{2}\}$, $R_3 = \{\underline{6}\}$, $R_4 = \{\underline{6},7\}$

(węzeł podkreślony jest wejściem danego regionu).

W grafie G są trzy regiony maksymalne, mianowicie: R_1, R_2 i R_4.

Rozdział II. JĘZYKI I PROGRAMY

Trudno jest mówić o optymalizacji programów nie nawiązując do języka, którego programy mają stanowić przedmiot optymalizacji. Toteż zanim przejdziemy do omawiania zagadnień związanych z optymalizacją, określimy rodzaj języków jakimi będziemy się zajmowali. Ze znanych języków pośrednich najbardziej dogodny do optymalizacji jest tzw. język czwórek (patrz Gries [11]) i on też znajduje często zastosowanie jako język pośredni w translatorach przewidujących fazę optymalizacji (Lowery i Medlock [13]). Instrukcje języka czwórek składają się z operacji, dwóch argumentów i jednego wyniku, przy czym każda instrukcja w programie może być poprzedzona etykietą.

W niniejszym rozdziale zdefiniujemy model języków stanowiący uogólnienie języka czwórek, przy czym ograniczymy się na razie do języków bez procedur. Na bazie tu omówionych pojęć w następnym rozdziale określimy języki z procedurami.

1. Systemy programowania

Szukając możliwie najprostszego modelu dla języków czwórek musieliśmy uwzględnić fakt, że ze względu na potrzeby optymalizacji powinny być dostępne następujące informacje o instrukcjach języka: zbiór zmiennych otrzymujących nowe wartości (tzw. definiowanych) i zbiór zmiennych, do których następuje odwołanie w

trakcie wykonywania instrukcji oraz określenie kolejności w jakiej mogą być wykonane instrukcje w programie.
Mając to na uwadze przyjęliśmy następującą definicję systemu programowania:

Definicja 10. Niech $\mathcal{B}$ będzie uporządkowaną piątką, tzw. bazą:

$$\mathcal{B} = (O, Z, S, E, U),$$

gdzie O, Z, S, E, U są przeliczalnymi zbiorami, których elementy będziemy nazywali odpowiednio operacjami, zmiennymi, stałymi, etykietami i instrukcjami.

Przez system programowania o bazie $\mathcal{B}$ będziemy rozumieli uporządkowaną piątkę:

$$S(\mathcal{B}) = (o, l_e, p_a, p_w, p_e)$$

w której:

1) $o : U \to O$ jest tzw. funkcją operacji,
2) $l_e : U \to E \cup \{\varepsilon\}$ jest tzw. funkcją etykiety,
3) $p_a : U \to (Z \cup S)^*$ jest tzw. funkcją pola argumentów,
4) $p_w : U \to Z^*$ jest tzw. funkcją pola wyników,
5) $p_e : U \to E^*$ jest tzw. funkcją pola etykiet.

Zwrot: "system programowania o bazie $\mathcal{B}$ "będziemy często zastępowali krótszym: "system programowania", jeśli wiadomo będzie jaką bazę mamy na myśli.

Dla dowolnej instrukcji $s \in U$ element $o(s)$ będziemy nazywali operacją instrukcji s, $l_e(s)$ - etykietą instrukcji s, ciąg $p_a(s)$ - polem argumentów instrukcji s, $p_w(s)$ - polem wyników instrukcji s. Uporządkowaną parę $(o(s), p_a(s))$ nazwiemy wyrażeniem instrukcji s.

Załóżmy, że dla dowolnych instrukcji s i s' systemu programowania S, mających identyczne operacje, długości pól argumentów, wyników i etykiet instrukcji s są równe odpowiednio długościom pól argumentów, wyników, etykiet instrukcji s'.

Instrukcję s mającą niepuste pola argumentów i wyników i puste pole etykiet nazwiemy instrukcją przypisania, a operację $o(s)$ - operacją przypisania lub krócej: przypisaniem. Instrukcję s o niepustym polu etykiet nazwiemy instrukcją sterującą, a operację $o(s)$ - operacją sterującą.

W dalszym ciągu bieżącego rozdziału mówiąc o systemie programowania będziemy mieli na myśli dowolny, ale ustalony system $S = S(\mathcal{B}) = (o, l_e, p_a, p_w, p_e)$, gdzie $\mathcal{B} = (O, Z, S, E, U)$, chyba że zostanie wyraźnie powiedziane, że chodzi o jakiś inny konkretny system.

Definicja 11. Instrukcję s systemu programowania S nazwiemy definicją zmiennej a, jeśli zmienna a należy do ciągu $p_w(s)$. Instrukcję s nazwiemy odwołaniem do zmiennej a, jeśli zmienna a należy do ciągu $p_a(s)$.

Mówiąc nieformalnie, wykonanie instrukcji polega na tym, że wartości zmiennych występujących w polu wyników zostają zmodyfikowane na podstawie wartości zmiennych i stałych z pola argumentów instrukcji. Pole wyników instrukcji reprezentuje zatem wszystkie zmienne, które mogą zmienić wartość w trakcie wykonania instrukcji, natomiast pole argumentów instrukcji stanowi wykaz zmiennych i stałych, których wartości są wykorzystane do wyliczeń związanych z wykonaniem instrukcji. Ponadto instrukcja może określać, które instrukcje programu mogą być wykonane następnie.

Jak to się dzieje dokładnie zostanie omówione w następnym paragrafie poświęconym programom, na razie powiemy tylko, że do tego celu służą etykieta instrukcji i pole etykiet instrukcji.

Za pomocą informacji zawartych w poszczególnych polach instrukcji można wyrazić wiele pojęć istotnych dla optymalizacji programów. Na bazie tych pojęć będziemy mogli zdefiniować wiele nietrywialnych, znanych z literatury i praktycznie sprawdzonych, przekształceń optymalizacyjnych, takich jak: redukcja identycznych wyrażeń, wynoszenie instrukcji poza pętlę, usuwanie zbędnych instrukcji z programu.

Celem naszym jest pokazanie jakie przekształcenia są wykonywane w konstruowanym Systemie Optymalizacyjnym, natomiast problemem znacznie przekraczającym zakres tej pracy jest formalne pokazanie, że opisane przekształcenia są semantycznie poprawne (tj. dają w wyniku program równoważny semantycznie programowi optymalizowanemu).

Przykład 3. Dla nadania większej jasności wprowadzonym pojęciom opiszemy pewien prosty system programowania. Bazę określimy w następujący sposób:

$$\overline{\mathcal{B}} = (\bar{Q}, \bar{Z}, \bar{S}, \bar{E}, \bar{U}), \quad \text{gdzie}$$

1) $\bar{Q} = \{:=, \ominus, +, -, *, /, \uparrow, =\uparrow, <\uparrow, \leq\uparrow, \text{NOP}, \text{PISZ}, \text{CZYT}\}$,

2) $\bar{Z}$ i $\bar{E}$ są zbiorami skończonych napisów literowo-cyfrowych zaczynających się od litery, np. a, ab, ca1,

3) $\bar{S}$ jest zbiorem skończonych napisów cyfrowych, które mogą być poprzedzone symbolem + albo -, np. 1, -124, +215,

4) zbiór instrukcji $\overline{U}$ zdefiniujemy jako podzbiór produktu kartezjańskiego:

$$(E \cup \{\varepsilon\}) \times \bar{Z}^* \times \bar{Q} \times (\bar{Z} \cup \bar{S})^* \times \bar{E}^*.$$

określony w następujący sposób:

$$\begin{aligned}
\bar{U} = {} & (\bar{E} \cup \{\varepsilon\}) \times \bar{Z} \times \{:=, \ominus\} \times (\bar{Z} \cup \bar{S}) \times \{\varepsilon\} \cup \\
& (\bar{E} \cup \{\varepsilon\}) \times \bar{Z} \times \{+, -, *, /\} \times (\bar{Z} \cup \bar{S})^2 \times \{\varepsilon\} \cup \\
& (\bar{E} \cup \{\varepsilon\}) \times \{\varepsilon\} \times \{\text{PISZ}\} \times (\bar{Z} \cup \bar{S}) \times \{\varepsilon\} \cup \\
& (\bar{E} \cup \{\varepsilon\}) \times \bar{Z} \times \{\text{CZYT}\} \times \{\varepsilon\} \times \{\varepsilon\} \cup \\
& (\bar{E} \cup \{\varepsilon\}) \times \{\varepsilon\} \times \{\uparrow\} \times \{\varepsilon\} \times \bar{E} \cup \\
& (\bar{E} \cup \{\varepsilon\}) \times \{\varepsilon\} \times \{=\uparrow, <\uparrow, \leq\uparrow\} \times (\bar{Z} \cup \bar{S})^2 \times \bar{E}^2 \cup \\
& (\bar{E} \cup \{\varepsilon\}) \times \{\varepsilon\} \times \{\text{NOP}\} \times \{\varepsilon\} \times \{\varepsilon\}
\end{aligned}$$

Niech $\bar{S} = \bar{S}(\overline{\mathcal{B}}) = (\bar{o}, \bar{l}_o, \bar{P}_a, \bar{P}_w, \bar{P}_o)$ będzie systemem programowania o bazie $\overline{\mathcal{B}}$, w którym

5) $\bar{o} \equiv \varphi_{13}$ (przypominamy, że φ_{il} oznacza rzut i-tej współrzędnej produktu)

6) $\bar{l}_o \equiv \varphi_{11}$

7) $\bar{P}_a \equiv \varphi_{14}$

8) $\bar{p}_w \equiv \varphi_{12}$

9) $\bar{p}_e \equiv \varphi_{15}$

Operacje :=, $\ominus$, +, -, * , / są przypisaniami i będziemy je nazywali kolejno: prostym przypisaniem, jednoargumentowym minusem, dodawaniem, odejmowaniem, mnożeniem i dzieleniem. Oto przykłady instrukcji przypisania systemu programowania $\bar{S}$ wraz z ich interpretacją:

$\mathcal{E}$, a, :=, b, $\mathcal{E}$ wartość zmiennej lub stałej z pola argumentów zostaje przypisana zmiennej z pola wyników.

Powyższa instrukcja jest definicją zmiennej a, odwołaniem do zmiennej b, wyrażeniem tej instrukcji jest para (:=,b) .

$\mathcal{E}$, c4, $\ominus$, a, $\mathcal{E}$ wartość zmiennej lub stałej z pola argumentów wzięta ze znakiem przeciwnym zostaje przypisana zmiennej z pola wyników.

et, b, +, a ; 25, $\mathcal{E}$ zostają dodane do siebie wartości elementów z pola argumentów i wynik umieszczony w zmiennej z pola wyników.

et1, c, -, a; b, $\mathcal{E}$ od wartości pierwszego elementu z pola argumentów zostaje odjęta wartość drugiego i wynik jest umieszczony w zmiennej z pola wyników.

$\mathcal{E}$, a, *, cd ;24, $\mathcal{E}$ zostają wymnożone wartości elementów z pola argumentów i wynik umieszczony w zmiennej z pola wyników.

$\mathcal{E}$, c, /, a; b, $\mathcal{E}$ wartość pierwszego elementu z pola argumentów zostaje całkowicie podzielona przez wartość drugiego i wynik jest umieszczony w zmiennej z pola wyników.

Operacje PISZ i CZYT są operacjami wejścia-wyjścia. Umożliwiają one wprowadzenie wartości zmiennej lub stałej na urządzenie zewnętrzne i wczytanie wartości na zmienną z urządzenia zewnętrznego.

Operacje $\uparrow$,$=\uparrow$,$<\uparrow$,$\leq\uparrow$ są operacjami sterującymi. Na poniższych przykładach podamy ich interpretację:

$\mathcal{E}$, $\mathcal{E}$, $\uparrow$, $\mathcal{E}$, et przekaż sterowanie do instrukcji s takiej, że et=$\bar{l}_c(s)$.

$\mathcal{E}$, $\mathcal{E}$, $=\uparrow$, a; b, et1; et2 jeśli wartości elementów z pola argumentów są identyczne, to przekaż sterowanie do instrukcji s takiej, że $\bar{l}_e(s)$= et1. W przeciwnym przypadku, przekaż sterowanie do instrukcji s takiej, że $\bar{l}_e(s)$=et2.

$\mathcal{E}$, $\mathcal{E}$,$<\uparrow$, a ; b, et1; et2 jeśli wartość pierwszego elementu z pola argumentów jest mniejsza od wartości drugiego, to przekaż sterowanie do instrukcji s takiej, że $\bar{l}_e(s) = et1$. W przeciwnym przypadku, przekaż sterowanie do instrukcji s takiej, że $\bar{l}_e(s) = et2$.

$\mathcal{E}$, $\mathcal{E}$,$\leq\uparrow$, a; b, et1; et2 podobnie jak w poprzedniej instrukcji z tym, że sprawdza się zachodzenie relacji mniejszości równości.

NOP jest operacją pustą. Z wykonaniem instrukcji o operacji NOP nie jest związana żadna akcja.

System programowania $\bar{S}$ często będziemy wykorzystywali do ilustrowania omawianych zagadnień. Dla zwiększenia przejrzystości zapisu instrukcji przyjmiemy konwencję, żeby symbole puste występujące w instrukcji zastępować po prostu wolnym miejscom. Na przykład, zamiast ε, ε, $\uparrow$, ε, et będziemy pisali:

, , $\uparrow$, , et

2. Programy

Wprowadzone w poprzednim paragrafie systemy programowania są jak gdyby generatorami instrukcji. Obecnie wyróżnimy niektóre ciągi instrukcji i nazwiemy je programami. Nasze pojęcie programu jest w pewnym sensie odpowiednikiem programu w maszynie Pawlaka [16] lub algorytmu Mazurkiewicza [15].

Przyjmiemy następującą definicję programu:

Definicja 12. Programem w systemie programowania S nazwiemy każdy skończony ciąg $\Pi = s_1, \ldots, s_n$ instrukcji tego systemu spełniający warunki:

1) $l_o(s_1) = \varepsilon$,

2) instrukcja s_n nie jest instrukcją sterującą tzn. $p_o(s_n) = \varepsilon$,

3) dla każdej instrukcji $s \in \Pi$ takiej, że $l_o(s) \in E$ istnieje instrukcja sterująca $s' \in \Pi$ taka, że $l_o(s)$ jest elementem ciągu $p_o(s')$,

4) dla każdej instrukcji sterującej $s \in \Pi$ i każdej etykiety e należącej do ciągu $p_o(s)$ istnieje dokładnie jedna instrukcja $s' \in \Pi$ taka, że $l_o(s) = e$.

Dla danego programu Π w systemie S zbiór etykiet programu Π będziemy oznaczali przez $E_s(\Pi)$, a zbiór zmiennych programu przez $Z_s(\Pi)$.

Zbiór wszystkich programów systemu S będziemy oznaczali przez $J(S)$ i nazwiemy go językiem systemu programowania S.

Przykład 4. Poniższy ciąg instrukcji jest programem w systemie programowania $\bar{S}$ zdefiniowanym w przykładzie 3:

```
s1                  , a, CZYT,    ,
s2                  , b, CZYT,    ,
s3                  , c,   + , a; b,
s4                  ,  , ≤ ↑, c; 5, et1; et2
s5                et1, c,  * , c; 2,
s6                  ,  , = ↑, c; 50, et1; et3
s7                et2, c , + , c; a,
s8                  ,  , PISZ, c,
s9                et3, c ,  / , c; b,
s10                 ,  , PISZ, c,
```

Program Π

Program Π ma następujące zbiory etykiet i zmiennych:

$$E_{\bar{S}}(\Pi) = \{et1, et2, et3\} \quad \text{i} \quad Z_{\bar{S}}(\Pi) = \{a, b, c\}.$$

Definicja 13. Niech $\pi = s_1,\ldots,s_n$ będzie programem w systemie programowania S. Obliczeniem w programie π o początku s i końcu s' będziemy nazywali dowolny niepusty ciąg $s_{i_1},\ldots,s_{i_l}$ instrukcji programu π taki, że $s_{i_1} = s$, $s_{i_l} = s'$ i dla dowolnego $k \in \langle 1, l-1 \rangle$ jest spełniony jeden z następujących warunków:

1) instrukcja s_{i_k} nie będąca instrukcją sterującą występuje bezpośrednio przed instrukcją $s_{i_{k+1}}$ w programie π,

2) instrukcja s_{i_k} jest instrukcją sterującą i etykieta $l_e(s_{i_{k+1}})$ należy do ciągu $p_e(s_{i_k})$.

Każde obliczenie o początku w pierwszej i końcu w ostatniej instrukcji programu π nazwiemy realizacją programu π.

W programie π z ostatniego przykładu następujące ciągi instrukcji są obliczeniami:

s_3, s_4, s_5, s_6, s_5

s_5, s_6, s_9, s_{10}

$s_1, s_2, s_3, s_4, s_5, s_6, s_9, s_{10}$

Ostatni ciąg jest realizacją programu π. Liczbę instrukcji występujących w obliczeniu nazwiemy długością obliczenia.

Definicja 14. Dla dowolnego programu π w systemie S i dowolnej instrukcji $s \in \pi$ powiemy, że s spełnia warunek spójności programu, jeśli s należy do pewnej realizacji programu π. Program π nazwiemy programem spójnym, jeśli każda instrukcja spełnia warunek spójności programu.

Zauważmy, że z dowolnego programu możemy otrzymać program spójny przez usunięcie wszystkich instrukcji nie spełniających warunku spójności. Program π z przykładu 4 jest programem spójnym.

Definicja 15. Niech $\pi = s_1,\ldots,s_n$ będzie programem w systemie S, a-dowolną zmienną programu π oraz s-definicją zmiennej a w programie π.

Powiemy, że definicja s zmiennej a osiąga instrukcję s' programu π, jeśli istnieje obliczenie $s_{i_1},\ldots,s_{i_l}$ w programie o długości większej niż jeden, takie że $s_{i_1} = s$, $s_{i_l} = s'$ i żadna instrukcja spośród $s_{i_2},\ldots,s_{i_{l-1}}$ nie jest definicją zmiennej a.

W programie π z przykładu 4 definicja s_1 zmiennej a osiąga wszystkie instrukcje programu z wyjątkiem s_1, definicja s_2 zmiennej b osiąga wszystkie instrukcje programu z wyjątkiem s_1 i s_2, z kolei definicja s_3 zmiennej c osiąga instrukcje s_4, s_5 i s_7.

Obecnie określimy dla dowolnego programu π rodzinę relacji binarnych, które będziemy nazywali relacjami definicji-odwołania w programie π.

Definicja 16. Niech $\pi = s_1,\ldots,s_n$ będzie programem w systemie programowania S oraz a-dowolną zmienną programu π. Powiemy, że instrukcja s programu π pozostaje w relacji $\Delta_\pi(a)$ z instrukcją s' programu π, jeśli s jest definicją zmiennej

a, s' jest odwołaniem do zmiennej a i definicja s zmiennej a osiąga instrukcję s' programu $\mathcal{T}$.

Relacje definicji-odwołania odgrywają ważną rolę w optymalizacji programów. Dla programu $\mathcal{T}$ z przykładu 4 następujące zbiory tworzą relacje definicji-odwołania:

$$\Delta_{\mathcal{T}}(a) = \{(s_1, s_3), \quad (s_1, s_7)\}$$

$$\Delta_{\mathcal{T}}(b) = \{(s_2, s_3), \quad (s_2, s_9)\}$$

$$\Delta_{\mathcal{T}}(c) = \{(s_3, s_4), \quad (s_3, s_5), (s_3, s_7), (s_5, s_6), (s_5, s_5), (s_5, s_9), (s_9, s_{10})\}$$

3. Podział programu na bloki i graf programu

Każdemu programowi można przyporządkować graf opisujący sterowanie programu. Ze względu na znaczenie, grafy programu były badane przez wielu autorów, np. Gries [11], Allen [4], Aho i Ullman [3], Lowery i Medlock [13]. Informacje uzyskane z analizy tego grafu są bardzo istotne z punktu widzenia optymalizacji programu. Zanim dokładnie określimy co będziemy rozumieli pod pojęciem grafu programu, pokażemy w jaki sposób można podzielić program na tzw. bloki.

Definicja 17. Niech $\mathcal{T} = s_1, \ldots, s_n$ będzie programem w systemie programowania $\mathcal{S}$. Instrukcję s_i programu $\mathcal{T}$ nazwiemy wejściem bloku programu $\mathcal{T}$, jeśli s_i jest pierwszą instrukcją programu lub $l_e(s_i) \neq \varepsilon$. Instrukcję s_j programu $\mathcal{T}$ nazwiemy wyjściem bloku programu $\mathcal{T}$ w jednym z trzech następujących przypadków:

1) s_j jest ostatnią instrukcją programu $\mathcal{T}$,

2) s_j jest instrukcją sterującą,

3) $l_e(s_{j+1}) \neq \varepsilon$.

Blokiem programu $\mathcal{T}$ nazwiemy każdy ciąg $s_i, \ldots, s_j$ kolejnych instrukcji programu $\mathcal{T}$ taki, że

1) s_i jest wejściem bloku programu $\mathcal{T}$,

2) s_j jest wyjściem bloku programu $\mathcal{T}$,

3) dla żadnego $k \in \langle i, j-1 \rangle$ instrukcja s_k nie jest wyjściem bloku programu $\mathcal{T}$.

Korzystając z powyższej definicji możemy każdy program podzielić na bloki w sposób jednoznaczny. Zbiór wszystkich bloków programu $\mathcal{T}$ będziemy oznaczali przez $B(\mathcal{T})$. Wracając do przykładu 4, w programie $\mathcal{T}$ wyróżnimy następujące bloki: $x_1 = \{s_1, s_2, s_3, s_4\}$, $x_2 = \{s_5, s_6\}$, $x_3 = \{s_7, s_8\}$, $x_4 = \{s_9, s_{10}\}$.

Opierając się na pojęciu bloku programu zdefiniujemy teraz graf programu i następnie wykażemy, że graf programu spójnego jest grafem spójnym.

Definicja 18. Niech $\mathcal{T}$ będzie programem w systemie $\mathcal{S}$. Grafem programu $\mathcal{T}$ nazwiemy graf $G = (B(\mathcal{T}), \Gamma)$, gdzie relacja Γ jest określana w następujący sposób: dla dowolnych bloków $x, y \in B(\mathcal{T})$ para $(x, y) \in \Gamma$ wtedy i tylko wtedy, gdy spełniony jest jeden z warunków:

1) etykieta instrukcji wejściowej bloku y jest elementem pola etykiet instrukcji wyjściowej bloku y,

2) instrukcja wejściowa bloku y występuje bezpośrednio po instrukcji wyjściowej bloku x w programie $\mathcal{T}$.

Właściwość 10. Dla dowolnego spójnego programu $\mathcal{T}$ w systemie S , graf programu $\mathcal{T}$ jest grafem spójnym.

Dowód. Niech $\mathcal{T} = s_1,\ldots,s_n$ będzie spójnym programem i $G = (B(\mathcal{T}), \Gamma)$ grafem programu $\mathcal{T}$. Wykażemy, że blok ξ, którego wejściem jest pierwsza instrukcja programu $\mathcal{T}$ jest węzłem początkowym grafu G. Z definicji programu i ze sposobu w jaki określiliśmy relację dla grafu programu wynika, że ξ nie ma poprzedników w G. Niech teraz x będzie dowolnym blokiem programu $\mathcal{T}$ różnym od ξ. Musimy pokazać, że istnieje droga w G o początku ξ i końcu x. Ponieważ, z założenia, każda instrukcja programu $\mathcal{T}$ spełnia warunek spójności programu, więc w szczególności dla instrukcji s będącej wejściem bloku x, istnieje obliczenie w $\mathcal{T}$ o początku s_1 i końcu s. Z definicji obliczenia oraz z określenia relacji Γ dla grafu programu otrzymujemy, że w grafie G istnieje droga o początku ξ i końcu x

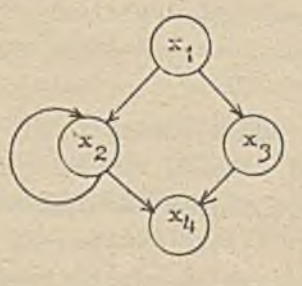
c. k. d.

Ponieważ w dalszym ciągu będziemy się zajmowali wyłącznie grafami programów spójnych, więc zgodnie z poprzednio przyjętymi oznaczeniami, graf programu $\mathcal{T}$ będziemy oznaczali przez

$G_{\mathcal{T}} = (X_{\mathcal{T}}, \Gamma_{\mathcal{T}}, \xi_{\mathcal{T}})$,gdzie $\xi_{\mathcal{T}}$ jest blokiem zawierającym pierwszą instrukcję programu $\mathcal{T}$.

Zbudujemy graf dla programu $\mathcal{T}$ z przykładu 4. W programie wyróżniliśmy cztery bloki: $x_1 = \{s_1, s_2, s_3, s_4\}$, $x_2 = \{s_5, s_6\}$, $x_3 = \{s_7, s_8\}$ i $x_4 = \{s_9, s_{10}\}$. Graf programu $\mathcal{T}$ ma następującą postać:

$$G_{\mathcal{T}} = (\{x_1, x_2, x_3, x_4\}, \{(x_1, x_2), (x_1, x_3), (x_2, x_2), (x_2, x_4), (x_3, x_4)\}, x_1)$$



Rys. 3

Na zakończenie wprowadzimy jeszcze pojęcia pętli i dominacji instrukcji w programie, mające ścisły związek z pojęciami pętli i dominacji bloków w grafie programu.

Definicja 19. Niech $\mathcal{T}$ będzie dowolnym programem w systemie programowania S. O instrukcjach s i s' powiemy, że s dominuje nad s' jeśli s należy do każdego obliczenia o początku w pierwszej instrukcji programu $\mathcal{T}$ i końcu s'.

Właściwość 11. Dla dowolnych instrukcji s i s' programu $\mathcal{T}$ s dominuje nad s' wtedy i tylko wtedy, gdy jest spełniony jeden z warunków:

1) s i s' znajdują się w tym samym bloku programu, przy czym s poprzedza s',

2) w grafie programu $\mathcal{T}$ blok, w którym znajduje się instrukcja s dominuje nad blokiem zawierającym instrukcję s'.

Definicja 20. Pętlą w programie $\mathcal{T}$ w systemie S nazwiemy każdy podzbiór L instrukcji programu $\mathcal{T}$ taki, że dla dowolnych $s, s' \in \mathcal{T}$ istnieje obliczenie $s_1, .. s_n$ takie, że $n \geq 2$, $s_1 = s$, $s_n = s'$ i $s_i \in L$ dla $i \in \langle 1, n \rangle$.

Wejściem pętli L w programie $\mathcal{T}$ nazwiemy każdą instrukcję $s \in L$ taką, że istnieje instrukcja s' programu $\mathcal{T}$ nie należąca do L i taka, że $l_e(s) \in p_e(s')$. Analogicznie wyjściem pętli nazwiemy każdą instrukcję $s \in \mathcal{T}$ taką, że istnieje instrukcja s' programu $\mathcal{T}$ nie należąca do L i taka, że $l_e(s') \in p_e(s)$.

Pętlę z jednym wejściem nazwiemy regionem w programie.

Własciwość 12. Niech $\mathcal{T}$ będzie programem w systemie programowania S i $G_{\mathcal{T}}$ grafem programu $\mathcal{T}$.

1) Każdej pętli (regionowi) w programie $\mathcal{T}$ odpowiada dokładnie jedna pętla (region) w grafie $G_{\mathcal{T}}$ i odwrotnie.

2) Jeśli instrukcja s jest wejściem (wyjściem) pętli w programie $\mathcal{T}$ to s jest pierwszą (ostatnią) instrukcją bloku będącego wejściem (wyjściem) pętli w grafie $G_{\mathcal{T}}$ i odwrotnie.

W programie $\mathcal{T}$ z przykładu 4 jest jedna pętla $L = \{s_5, s_6\}$. Pętla ta ma jedno wejście, instrukcję s_5, a zatem jest regionem. Wyjściem pętli L jest instrukcja s_6. Pętli L w programie $\mathcal{T}$ odpowiada pętla $R = \{x_2\}$ w grafie $G_{\mathcal{T}}$ programu $\mathcal{T}$.

Rozdział III. JĘZYKI Z PROCEDURAMI I ICH PROGRAMY

W tym rozdziale zajmiemy się uzupełnieniem dotychczas rozpatrywanych języków o procedury. Dopuścimy dwa rodzaje wołania parametrów: przez wartość i przez nazwę.

Programy z procedurami są trudniejsze do analizy. Podstawowy problem jaki tu napotykamy jest następujący: dla każdej instrukcji programu, w szczególności dla instrukcji wywołania procedury, należy określić zbiór zmiennych otrzymujących nowe wartości (definiowanych) i zbiór zmiennych, których wartości są wykorzystywane (tzn. zmiennych, do których ma miejsce odwołanie). W pracy Allen [7] została podana metoda rozwiązania tego zagadnienia dla programów bez procedur rekurencyjnych. Nasze rozwiązanie (algorytm 2) nie nakłada tego ograniczenia i jest ponadto wzbogacone o analizę powiązań parametrów wołanych przez nazwę z odpowiadającymi im parametrami aktualnymi, której wynikiem jest wyznaczenie tzw. synonimów (algorytm 1).

1. Systemy programowania z procedurami

Obecnie uzupełnimy systemy programowania zdefiniowane w poprzednim rozdziale o procedury, dołączając niezbędne do tego celu operacje i instrukcje, takie jak: nagłówek procedury, wołanie procedury i inne.

Definicja 21. Niech $\mathfrak{B} = (Q, Z, S, E, U)$ będzie bazą. Systemem programowania z procedurami o bazie $\mathfrak{B}$ będziemy nazywali uporządkowaną czwórkę:

$S^P(\mathfrak{B}) = (S(\mathfrak{B}), \{PROC, KON, WOL, MPROC, DKL\}, (p_n, p_r))$, w której

1) $S(\mathfrak{B}) = (o, l_e, p_a, p_w, p_o)$ jest systemem programowania w myśl definicji 10 z rozdziału II.

2) PROC, KON, WOL, MPROC, DKL są wyróżnionymi operacjami ze zbioru Q, które nazwiemy kolejno: nagłówkiem procedury, operacją końca procedury, wołaniem procedury, marką procedury i deklaracją

3) $p_n : U \to Z \cup \{\mathcal{E}\}$ jest tzw. funkcją nazwy taką, że

$p_n(s) \in Z$ jeśli $o(s) \in \{PROC, KON, WOL, MPROC\}$

oraz

$p_n(s) = \mathcal{E}$ w pozostałych przypadkach

4) $p_r : U \to (Z \times \{1,2\})^* \cup (Z \cup S)^*$ jest tzw. funkcją parametrów, dla której

$p_r(s) \in (Z \cup S)^*$ jeśli $o(s) =$ WOL

$p_r(s) \in (Z \times \{1,2\})^*$ jeśli $o(s) \in \{PROC, MPROC\}$

$p_r(s) \in Z^*$ jeśli $o(s) =$ DKL

$p_r(s) = \mathcal{E}$ w pozostałych przypadkach

Zwrot: "system programowania z procedurami o bazie $\mathfrak{B}$" będziemy często zastępowali zwrotem krótszym "system programowania z procedurami", jeśli wiadomo będzie jaką bazę mamy na myśli.

Dla dowolnej instrukcji s systemu S^P, $p_n(s)$ nazwiemy polem nazwy instrukcji s, natomiast ciąg $p_r(s)$ polem parametrów instrukcji s.

Instrukcje o operacjach PROC, KON, WOL i MPROC są związane z procedurami i, jak to wynika z punktu 3) definicji, mają niepuste pola nazwy. Zmienną z tego pola będziemy uważali za nazwę procedury, do której odnosi się dana instrukcja. Instrukcję o operacji PROC będziemy nazywali instrukcją nagłówka procedury. Pole parametrów tej instrukcji, o ile nie jest puste, jest ciągiem uporządkowanych par: zmienna - cyfra ze zbioru $\{1,2\}$. Każdą zmienną a, występującą w polu parametrów w postaci $(a,1)$ nazwiemy parametrem formalnym wołanym przez wartość (w skrócie p.f.w.), natomiast zmienną występującą w postaci $(a,2)$ nazwiemy parametrem formalnym wołanym przez nazwę (p.f.n.).

Instrukcję o operacji KON nazwiemy instrukcją końca procedury.

Instrukcję o operacji MPROC nazwiemy instrukcją marki procedury.

Instrukcja o operacji WOL jest instrukcją wołania procedury. Pole parametrów tej instrukcji zawiera tzw. parametry aktualne.

Instrukcja o operacji DKL jest instrukcją deklaracji. Pole parametrów tej instrukcji jest listą zmiennych wprowadzonych do programu (deklarowanych). Rola wyżej opisanych instrukcji zostanie wyjaśniona w trakcie omawiania procedur i programów w systemach z procedurami.

Przykład 5. System programowania $\bar{S}$ zdefiniowany w przykładzie 3 rozdziału II rozszerzymy obecnie do systemu programowania z procedurami. Niech $\tilde{\mathfrak{B}} = (\tilde{Q}, \tilde{Z}, \tilde{S}, \tilde{E}, \tilde{U})$ będzie bazą, w której

1) $\tilde{Q} = \bar{Q} \cup \{PROC, KON, WOL, MPROC, DKL\}$,

2) $\tilde{Z} = \bar{Z}$, $\tilde{S} = \bar{S}$, $\tilde{E} = \bar{E}$,

3) zbiór instrukoji systemu $\bar{S}^P$ określimy jako podzbiór produktu kartezjańskiego:

$$(\tilde{E} \cup \{\varepsilon\}) \times ((\tilde{Z} \cup \tilde{S})^* \cup (\tilde{Z} * \{1,2\})^*) * \tilde{Q} \times \tilde{Z}^* \tilde{E}^*$$

w następujący sposób:

$$\begin{aligned} \tilde{U} = \bar{U} \cup \{\varepsilon\} \times (\tilde{Z} \times \{1,2\})^* \times \{PROC, MPROC\} \times \tilde{Z} \times \{\varepsilon\} \cup \\ (\tilde{E} \cup \{\varepsilon\}) \times \{\varepsilon\} \times \{KON\} \times \tilde{Z} \times \{\varepsilon\} \cup \\ (\tilde{E} \cup \{\varepsilon\}) \times (\tilde{Z} \cup \tilde{S})^* \times \{WOL\} \times \tilde{Z} \times \{\varepsilon\} \cup \\ \{\varepsilon\} \times \tilde{Z}^* \times \{DKL\} \times \{\varepsilon\} \times \{\varepsilon\} \end{aligned}$$

Niech teraz $\tilde{S}^P = \tilde{S}^P(\tilde{\mathfrak{B}}) = (\tilde{S}(\tilde{\mathfrak{B}}), \{PROC, KON, WOL, MPROC, DKL\}, p_n, p_r)$ będzie systemem programowania z procedurami o bazie $\tilde{\mathfrak{B}}$ określonym w następujący sposób:

4) $\tilde{S}(\tilde{\mathfrak{B}}) = (\tilde{o}, \tilde{I}_e, \tilde{p}_a, \tilde{p}_w, \tilde{p}_e)$ jest systemem programowania $\tilde{S}(\tilde{\mathfrak{B}})$ z przykładu 3, zmodyfikowanym następująco:

4a) $\tilde{o} \equiv \varphi_{|3}$ (przypominamy, że przez $\varphi_{|i}$ oznaczamy rzut i-tej współrzędnej produktu kartezjańskiego),

4b) $\tilde{I}_e \equiv \varphi_{|1}$,

4c) $\tilde{p}_a(s) = \varphi_{|4}(s)$ jeśli $\tilde{o}(s) \notin \{PROC, MPROC, WOL, DKL\}$

$\tilde{p}_a(s) = \varepsilon$ w przeciwnym przypadku,

4d) $\tilde{p}_w(s) = \varphi_{|2}(s)$ jeśli $o(s) \notin \{PROC, MPROC, WOL\}$

$\tilde{p}_w(s) = \varepsilon$ w przeciwnym przypadku,

4e) $\tilde{p}_e \equiv \varphi_{|5}$,

5) $\tilde{p}_n(s) = \varphi_{|4}(s)$ jeśli $o(s) \notin \{PROC, KON, MPROC, WOL\}$

$\tilde{p}_n(s) = \varepsilon$ w przeciwnym przypadku

6) $p_r(s) = \varphi_{|2}(s)$ jeśli $o(s) \notin \{PROC, WOL, MPROC, DKL\}$

$p_r(s) = \varepsilon$ w przeciwnym przypadku

Oto parę przykładów instrukoji systemu $\tilde{S}^P$:

, a;b;c;d, DKL, ,

Instrukoja deklarująca zmienne a,b,c, i d.

, a,1 ; b,2 ; c,1, PROC, pr1,

Instrukoja nagłówka procedury o nazwie pr1 z trzema parametrami formalnymi, z których pierwszy i ostatni są wołane przez wartość, a drugi jest wołany przez nazwę.

, x;y;z; WOL, pr1,

Instrukcja wołania procedury o nazwie pr1 z parametrami aktualnymi x,y i z.

, , KON, pr1,

, (a,1); (b,2); (c,1), MPROC, pr1.

Instrukcje końca procedury o nazwie pr1 i marki procedury o nazwie pr1 z parametrami formalnymi.

Do systemu $\tilde{S}^P$ należą ponadto wszystkie instrukcje systemu z przykładu 3.

Zauważmy, że z każdym systemem programowania z procedurami $S^P(\mathcal{B})$ jest związany pewien prosty system programowania $S(\mathcal{B})$ według def. 10 z rozdziału II. System ten będziemy nazywali <u>prostym systemem programowania związanym z systemem programowania z procedurami</u> $S^P(\mathcal{B})$.

Przejdziemy teraz do omówienia programów z procedurami.

2. Programy z procedurami

Rozpoczniemy od wprowadzenia pojęcia procedury i następnie określimy program jako ciąg procedur spełniający odpowiednie warunki. Bardzo istotny jest fakt, że każda procedura może być traktowana jako program w pewnym systemie programowania według definicji 10 z rozdziału II. Pozwoli nam to bez trudu przenieść na procedury pojęcia takie jak: graf, obliczenie i inne wprowadzone w rozdziale II dla programów bez procedur.

Definicja 22. <u>Procedurą</u> w systemie programowania z procedurami S^P nazwiemy każdy ciąg $\pi = s_1,\ldots,s_n$ instrukcji tego systemu taki, że

1) s_1 i s_n są odpowiednio instrukcjami nagłówka i końca procedury o tej samej nazwie, tzn. $o(s_1) = \text{PROC}$, $o(s_n) = \text{KON}$ i $p_n(s_1) = p_n(s_n)$,

2) $o(s_i) \in \{\text{PROC}, \text{KON}\}$ dla $i \in \langle 2, 2-1 \rangle$,

3) ciąg π jest programem w prostym systemie związanym z S^P.

Ciąg $s_2,\ldots,s_{n-1}$ będziemy nazywali <u>częścią właściwą</u> procedury π. O zmiennej występującej w polu nazwy instrukcji nagłówka procedury π powiemy, że jest <u>nazwą procedury</u> π.

Definicja 23. <u>Programem</u> z <u>procedurami</u> w systemie programowania z procedurami S^P nazwiemy ciąg procedur w tym systemie $\Pi = \pi_1,\ldots,\pi_n$, w którym π_1 jest procedurą bez parametrów formalnych.

Wyróżnioną procedurę π_1 nazwiemy <u>procedurą główną</u> programu Π.

Zbiory zmiennych i etykiet programu Π w systemie S^P oznaczymy odpowiednio przez $Z_{S^P}(\Pi)$; $E_{S^P}(\Pi)$. Zbiór wszystkich programów w systemie S^P będziemy oznaczali przez $\mathcal{J}(S^P)$ i nazwiemy go językiem z procedurami systemu programowania z procedurami S^P.

Aby pojęcia procedury i programu z procedurami stały się bliższe intuicji zilustrujemy je przykładem.

Przykład 6. Następujące ciągi instrukcji stanowią procedury w systemie $\tilde{S}^P$ zdefiniowanym w poprzednim przykładzie:

```
,      ,  PROC  ,  proc 1  ,
, i;j;k,  DKL   ,          ,
,(a,2);(b,2);(c,2), MPROC , proc 2,
,   i    ,  CZYT  ,          ,
,   k    ,  CZYT  ,          ,
,   j    ,   +    ,  i; 2    ,
, i;j;k  ,  WOL   ,  proc 2   ,
,        ,  KON   ,  proc 1   ,
```

π_1

```
,(a,2);(b,2);(c,2), PROC  , proc 2 ,
,     k          ,  /    , b; c    ,
,(d,2);(e,2);(f,1), MPROC , proc 3 ,
, b;c;j          , WOL   , proc 3 ,
,                , KON   , proc 2 ,
```

π_2

```
,(d,2);(e,2);(f,1), PROC  , proc 3 ,
,   l; m         , DKL   ,         ,
,    d           ,  +    , e; f    ,
, k; l; m        , WOL   , proc 2  ,
,                , KON   , proc 3  ,
```

π_3

Procedura π_1 o nazwie proc1 jest bezparametrowa. Procedura π_2 o nazwie proc2 ma trzy parametry formalne wołane przez nazwę i procedura π_3 o nazwie proc 3 ma dwa parametry wołane przez nazwę i jeden wołany przez wartość.

Ciąg $\pi = \pi_1, \pi_2, \pi_3$ jest programem w systemie $\tilde{S}^p$ z π_1 jako procedurą główną.

Opiszemy teraz nieformalnie w jaki sposób można interpretować programy z procedurami. Zaczniemy od omówienia wykonania instrukcji wołania procedury. Część właściwa wołanej procedury jest zmieniana w następujący sposób: parametry formalne procedury wołano przez nazwę są zastępowane w procedurze odpowiadającymi im parametrami aktualnymi, natomiast na parametry formalne wołane przez wartość są podstawiane wartości odpowiadających im parametrów aktualnych. Następnie zostaje wykonana część właściwa procedury zgodnie z poprzednio przyjętymi zasadami. Zmodyfikowaną, na skutek wołania, procedurę będziemy nazywali wersją procedury.

Wykonanie programu polega na wykonaniu części właściwej procedury głównej programu.

Warto jeszcze wspomnieć, że instrukcje marki procedury służą do nadania pewnego porządku procedurom programu. Miejsce wystąpienia takiej instrukcji w programie można interpretować jako miejsce deklaracji danej procedury.

3. Programy zupełne

Pojęcie programu jakim się posługiwaliśmy dotychczas jest zbyt ogólne.

Obecnie sformułujemy dodatkowe warunki, jakie powinny spełniać programy, określając tzw. programy zupełne. W tym celu wprowadzimy następujące definicje: niech $\Pi = \Pi_1, \ldots, \Pi_n$ będzie programem z procedurami w systemie programowania z procedurami S^P.

Definicja 24. Powiemy, że procedura Π_i bezpośrednio obejmuje procedurę Π_j, jeśli w procedurze Π_i występuje instrukcja marki procedury Π_j. Procedura Π_i obejmuje procedurę Π_j, jeśli istnieje ciąg procedur $\Pi_{i_1}, \ldots, \Pi_{i_m}$ programu Π taki, że $\Pi_{i_1} = \Pi_i$, $\Pi_{i_m} = \Pi_j$ i Π_{i_k} bezpośrednio obejmuje procedurę $\Pi_{i_{k+1}}$ dla $k \in \langle 1, n-1 \rangle$.

Definicja 25. Niech s będzie dowolną instrukcją wołania procedury Π_j w programie Π. Powiemy, że zmienna a odpowiada zmiennej b w instrukcji s, o ile a jest i-tym parametrem aktualnym w instrukcji s, natomiast b jest i-tym parametrem formalnym instrukcji nagłówka procedury Π_j. O dowolnych zmiennych a i b powiemy, że zmienna a odpowiada b w programie Π, jeśli istnieje instrukcja wołania procedury w programie Π, w której a odpowiada b.

W programie z przykładu 6 procedura Π_1 obejmuje procedury Π_2 i Π_3, przy czym bezpośrednio obejmuje procedurę Π_3. Procedura Π_2 bezpośrednio obejmuje procedurę Π_3. W tym samym programie zmienna i odpowiada parametrowi a, zmienna j - parametrowi b, zmienna k - parametrowi c, itd.

Zmienne deklarowane w danej procedurze (tzn. będące parametrami instrukcji deklaracji należącej do tej procedury) oraz parametry formalne procedury będziemy nazywali zmiennymi lokalnymi dla tej procedury. Zmienne nie lokalne występujące w danej procedurze będziemy nazywali zmiennymi zewnętrznymi tej procedury. Każde wystąpienie zmiennej w programie na liście parametrów instrukcji deklaracji lub instrukcji nagłówka procedury będziemy nazywali wprowadzeniem zmiennej do programu.

Definicja 26. Niech $\Pi = \Pi_1, \ldots, \Pi_n$ będzie programem z procedurami w systemie z procedurami S^P. Program Π nazwiemy programem zupełnym, o ile spełnia następujące warunki:

1) każda zmienna programu jest wprowadzona do programu dokładnie jeden raz,

2) zmienne lokalne dla danej procedury programu Π występują wyłącznie w tej procedurze i procedurach przez nią obejmowanych,

3) dla dowolnych instrukcji s i s' programu Π o operacjach ze zbioru $\{MPROC, PROC, WOL\}$ i identycznych nazwach ($p_n(s) = p_n(s')$) pola parametrów tych instrukcji mają identyczną długość,

4) nazwy procedur w ciągu $\Pi_1, \ldots, \Pi_n$ nie powtarzają się

5) każda procedura programu Π z wyjątkiem procedury głównej jest bezpośrednio obejmowana przez dokładnie jedną procedurę w Π, różną od niej samej; procedura główna nie jest obejmowana przez żadną procedurę w programie Π,

6) dla każdego $i \in \langle 1, n \rangle$ procedura Π_i może być wołana wyłącznie w procedurze Π_j bezpośrednio obejmującej Π_i oraz w procedurach obejmowanych przez Π_j.

Pokrótce skomentujemy znaczenie wprowadzonych warunków. Pierwszy warunek gwarantuje, że każda zmienna w programie zupełnym oznacza dokładnie jeden obiekt. Drugi punkt mówi o zakresie ważności zmiennych, trzeci zapewnia sensowną postać instrukcji odnoszących się do procedur, a czwarty gwarantuje, że w programie

nie występują dwie procedury o tej samej nazwie. Piąty warunek mówi o tym, że procedury mogą być deklarowane tylko raz w programie, a szósty określa zakres ważności procedur, w zależności od miejsca deklaracji.

Program $\mathcal{T}$ z przykładu 6 spełnia wszystkie warunki powyższej definicji, a zatem jest programem zupełnym.

W dalszym ciągu będą nas interesowały wyłącznie programy zupełne i zawsze mówiąc o programie z procedurami będziemy mieli na myśli program zupełny.

4. Synonimy zmiennych

W instrukcjach programu bez procedur nie mieliśmy problemu z określeniem, które zmienne są definiowane i do których następuje odwołanie: informacje te były zawarte bezpośrednio w polu wyników i polu argumentów instrukcji. Obecność procedur i parametrów formalnych wołanych przez nazwę komplikuje sytuację: parametr formalny wołany przez nazwę może bowiem być zastępowany w kolejnych wersjach procedury przez różne parametry aktualne, a także przez inne parametry formalne wołane przez nazwę. Zatem pojawienie się takiego parametru w polu wyników (lub argumentów) instrukcji może oznaczać definicję (lub odwołanie do) nie tylko tego parametru ale i innych zmiennych programu, tzw. synonimów. Celem tego paragrafu jest określenie pojęcia zbioru synonimów i podanie metody wyznaczenia tego zbioru dla każdej zmiennej programu z procedurami. Wyniki bieżącego paragrafu posłużą nam następnie do sprecyzowania pojęć definicji zmiennej i odwołania do zmiennej w instrukcjach programu z procedurami.

Na wstępie zauważmy, że pojęcie odpowiadania zmiennych w programie (definicja 25 str. 23) wyznacza pewną relację binarną w zbiorze zmiennych programu:

$$ODP_{\mathcal{T}} = \{(a,b) \mid a,b \in Z_{SP}(\mathcal{T}) \;\&\; a \text{ odpowiada } b \text{ w programie } \mathcal{T}\}$$

Nas będzie interesował pewien podzbiór tej relacji, mianowicie:

$$\overline{ODP}_{\mathcal{T}} = \{(a,b) \mid (a,a) \in ODP_{\mathcal{T}} \;\&\; b \text{ jest p.f.n}\}$$

oraz przechodnie i zwrotne domknięcie relacji $\overline{ODP}_{\mathcal{T}}$:

$$\overline{ODP}^{*}_{\mathcal{T}} = \{(a,b) \mid a,b \in Z_{SP}(\mathcal{T}) \;\&\; \exists a_1,\ldots,a_n \; (a = a_1 \;\&\; b = a_n \;\&\; \forall\, i \in \langle 1,n-1\rangle \; (a_1,a_{i+1}) \in \overline{ODP}_{\mathcal{T}})\}$$

Dla dowolnych zmiennych a i b takich, że $(a,b) \in \overline{ODP}^{*}_{\mathcal{T}}$ powiemy, że a jest prostym synonimem b . Warto tu zaznaczyć, że każda zmienna jest swoim własnym prostym synonimem.

Definicja 27. Niech $\mathcal{T} = \mathcal{T}_1,\ldots,\mathcal{T}_N$ będzie programem w systemie programowania z procedurami S^P. O dowolnych zmiennych a i b powiemy, że b jest synonimem a w programie $\mathcal{T}$ w jednym z następujących przypadków:

1) a jest prostym synonimem b,
2) b jest prostym synonimem a,
3) istnieje c takie, że c jest prostym synonimem a i b jednocześnie.

Pojęcie synonimu wyznacza pewną relację w zbiorze zmiennych programu:

$$SYN_{\mathcal{T}} = \{(a,b) \mid a \text{ jest synonimem } b \text{ w programie}\}$$

Relacja ta jest zwrotna i symetryczna ale nie jest przechodnia (warunek 3 definicji) .

Sens pojęcia synonimu zilustrujemy następującym przykładem:

Przykład 7

Rozważmy następujący program:

```
,                    , PROC,proc1,
, i:j                , DKL ,     ,
,(a,2); (b,2)        , MPROC,proc2,
                       :
, J ; J              , WOL  ,proc2,
                       :
                     , KON  ,proc1,
                       Π1

,(a,2); (b,2)        , PROC, proc2,
, k                  , DKL ,     ,
, k ; i              , WOL , proc2,
                       :
                     , KON,, proc2,
                       Π2
```

Przypatrzmy się najpierw instrukcji wołania procedury proc2 występującej w procedurze proc1. Zmienna j jest synonimem parametrów a i b (warunek 1 definicji 27) , natomiast parametry a i b są synonimamy zmiennej j (warunek 2 tej samej definicji). Z kolei parametr a jest synonimem parametru b i odwrotnie: b jest synonimem a, ponieważ obydwa parametry mają wspólny odpowiadający im parametr aktualny, mianowicie j (warunek 3 definicji 27).

Zwróćmy jeszcze uwagę na pewien istotny fakt. Ze względu na rekurencyjne wołanie procedury proc2, parametr a jest zastępowany w kolejnych wołaniach przez kolejne "edycje" lokalnej zmiennej k. Może to sugerować, że w tym przypadku parametr a ma nieskończony zbiór synonimów. Do naszych celów nie jest jednak istotne rozróżnianie poszczególnych edycji zmiennej k. W takim przypadku stwierdzamy jedynie, że k jest synonimem a, co zostało formalnie uwzględnione w definicji synonimu.

Dla dowolnej zmiennej a przez SYN (a) będziemy oznaczali zbiór wszystkich synonimów tej zmiennej. Zbiór ten jest zawsze skończony, ze względu na skończoną liczbę zmiennych zadeklarowanych w programie.

Przystępując do podania algorytmu wyliczającego zbiory synonimów dla zmiennych programu, musimy powiedzieć parę słów o notacji jaką się posłużyliśmy przy opisie tego algorytmu i następnych. Bardzo często w algorytmach będziemy konstruowali zbiory i w związku z tym będzie nam potrzebna operacja przypisania określona na zbiorach. Umówimy się, że zapis typu: $A \leftarrow B$ oznacza działanie, w wyniku którego nazwa występująca po lewej stronie zostaje związana ze zbiorem wymienionym po prawej stronie. Dopuścimy także możliwość wystąpienia po prawej stronie przypisania skończonej sumy, iloczynu oraz różnicy zbiorów

z oczywistą interpretacją wyniku, np. $A \leftarrow B \cup C$, $A \leftarrow A \cap B$.

Następujący algorytm pozwala dla danego programu $\pi = \pi_1, \ldots, \pi_n$ w systemie programowania z procedurami S^p i dla każdej zmiennej programu π znaleźć zbiór synonimów tej zmiennej.

Algorytm 1

Wejście: program $\pi = \pi_1, \ldots, \pi_n$ w systemie programowania S^p.
Wyjście: zbiory synonimów SYN(a) dla wszystkich zmiennych programu π.

Metoda:

1. Wykonaj $SYN(a) \leftarrow a$ dla wszystkich zmiennych a programu oraz $SYN_1(a) \leftarrow \emptyset$ dla zmiennych a będących p.f.n.

2. Dla wszystkich zmiennych a, b programu π, takich, że a odpowiada b w π i b jest p.f.n. wykonaj:

$$SYN(a) \leftarrow SYN(a) \cup \{b\},$$
$$SYN_1(b) \leftarrow SYN_1(b) \cup \{a\}.$$

3. Dla wszystkich zmiennych a programu π wykonaj:

3.1. Dla wszystkich zmiennych $b \in SYN(a)$ wykonaj:
$SYN(a) \leftarrow SYN(a) \cup SYN(b)$.

4. Wykonywanie kroku 3. powtarzaj dopóty, dopóki powoduje to dołączanie nowych elementów do któregoś ze zbiorów.

5. Dla wszystkich zmiennych a programu π będących p.f.n. wykonaj:

5.1. Dla wszystkich zmiennych $b \in SYN_1(a)$ będących p.f.n. wykonaj:
$SYN_1(a) \leftarrow SYN_1(a) \cup SYN_1(b)$.

6. Wykonywanie kroku 5, powtarzaj dopóty, dopóki powoduje to dołączanie nowych elementów do któregoś ze zbiorów,

7. Dla wszystkich zmiennych a programu π będących p.f.n. wykonaj:
$SYN(a) \leftarrow SYN(a) \cup SYN_1(a)$.

8. Dla wszystkich zmiennych a programu π będących p.f.n. wykonaj:

8.1. Dla wszystkich zmiennych b programu π będących p.f.n. i takich, że $SYN_1(a) \cap SYN_1(b) \neq \emptyset$ wykonaj:
$SYN(a) \leftarrow SYN(a) \cup \{b\}$.

Opis algorytmu 1. Pierwszy krok inicjalizuje zbiory synonimów zmiennych - $SYN(a)$ oraz zbiory pomocnicze - $SYN_1(a)$. W drugim kroku są przeglądane wszystkie instrukcje wołania procedury w programie i dla każdej zmiennej a powstaje zbiór:

$$SYN(a) = \{b \in Z_{S^p(\pi)} \mid (a,b) \in ODP_\pi\},$$

natomiast dla zmiennych a będących p.f.n. powstają dodatkowo zbiory:

$$SYN_1(a) = \{b \in Z_{S^p(\pi)} \mid (b,a) \in ODP_\pi\}$$

Kroki trzeci i czwarty budują kompletne zbiory SYN a dla zmiennych a nie

będących p.f.n., natomiast dla zmiennych będących p.f.n. konstruują zbiory odpowiadające warunkowi 1) definicji 27.

W krokach piątym, szóstym i siódmym dla zbiorów SYN(a) zostają dołączone zmienne spełniające warunek 2) definicji 27 i w ostatnim kroku powstają pełne zbiory synonimów dla zmiennych będących p.f.n.

Algorytm kończy działanie w skończonym czasie, ponieważ w każdym programie mamy do czynienia ze skończonym zbiorem zadeklarowanych zmiennych i skończoną liczbą instrukcji.

Przykład 8

Za pomocą algorytmu 1 znajdziemy zbiory synonimów dla zmiennych programu $\mathcal{T}$ z przykładu 6.

SYN	1	2	3,4	7	8
i	i	a			
j	j	b	d		
k	k	a,a	e		
l	l	b	d		
m	m	c	e		
a	a			i,k	a,e
b	b	d		i,l	e
c	c	e		k,m	e
d	d			b, i, l	
e	e			c, k, m	a,b
f	f				

Rys. 4

SYN_1	2	5,6
a	i,k	
b	j,l	
c	k,m	
d	b	j,l
e	c	k,m

Rys. 5

Tabelki przedstawione na rysunku 4 i 5 przedstawiają jak w kolejnych krokach algorytmu 1 powstają zbiory, odpowiednio SYN (a) i SYN(a) dla zmiennych programu $\mathcal{T}$. Każdy wiersz tabelki z rysunku 4 zawiera synonimy zmiennej, którą wiersz jest oznaczony, (cyferki na kolumnach pokazują, które części zbiorów synonimów powstały w kolejnych krokach stosowania algorytmu 1). Podobnie jest zbudowana tabelka z rysunku 5.

Opierając się na pojęciu synonimu zmiennej rozwiążemy teraz nasz zasadniczy problem, tzn. określimy zbiory zmiennych definiowanych i zmiennych, do których ma miejsce odwołanie w instrukcjach programu z procedurami.

5. Definicja zmiennej i odwołanie do zmiennej w programie z procedurami

Przypuśćmy, że mamy dany program z procedurami i w pewnej procedurze Π_i tego programu znajduje się instrukcja wywołania procedury Π_j. Wówczas interesuje nas, które zmienne aktywne w miejscu wywołania otrzymują nowe wartości (tzn. są definiowane) i do których zmiennych następuje odwołanie w wyniku wykonania tej instrukcji. W bieżącym paragrafie określimy pojęcia definicji zmiennych i odwołania do zmiennych w instrukcjach programu z procedurami i podamy algorytm wyznaczania zbiorów tych zmiennych. Zauważmy, że wywołanie procedury Π_i może spowodować wywołanie szeregu innych procedur programu. Fakt, że pojęcia definicji zmiennej i odwołania do zmiennej są tak złożone wynika z dwóch powodów. Po pierwsze nie nakładamy żadnych ograniczeń na postać programu i dopuszczamy procedury rekurencyjne. Po drugie, uwzględniamy konsekwencje wynikające z istnienia parametrów wołanych przez nazwę i przez wartość.

Określone tu pojęcia definicji i odwołania do zmiennych programu będą odgrywały podstawową rolę przy omawianiu problemów optymalizacji.

Na wstępie przyjmiemy następujące definicje: niech $\Pi = \Pi_1, \ldots, \Pi_n$ będzie programem w systemie programowania S^P. Dla dowolnej procedury Π_i programu Π przez $LOK(\Pi_i)$ oznaczymy zbiór zmiennych lokalnych w procedurze Π_i. Dla dowolnej instrukcji s procedury Π_i programu Π przez $AKT(s)$ oznaczymy zbiór zmiennych aktywnych w procedurze Π_i:

$$AKT(s) = \{ a \in Zsp(\Pi) \mid a \in LOK(\Pi_i) \text{ lub istnieje procedura } \Pi_j \text{ taka, że } a \in LOK(\Pi_j) \text{ i } \Pi_i \text{ obejmuje } \Pi_j \}.$$

Definicja 28. O zmiennej a programu Π będącej p.f.n. (tzn. parametrem formalnym wołanym przez nazwę) powiemy, że jest modyfikowana w programie Π, jeśli a jest prostym synonimem zmiennej b i istnieje instrukcja s programu Π taka, że $b \in p_w(s)$.

O zmiennej a programu Π, będącej p.f.n. powiemy, że jest wykorzystywana w programie Π, jeśli istnieje ciąg zmiennych $a_1, \ldots, a_n$ taki, że $a_1 = a$, a_i odpowiada a_{i+1} w Π, dla $i \in \langle 1, n-1 \rangle$ i spełniony jest jeden z warunków:

1) a_i są p.f.n. dla $i \in \langle 1, n-1 \rangle$ i a_n jest p.f.n. (tzn. parametrem formalnym wołanym przez wartość)

2) $n \geq 1$, są p.f.n. dla $i \in \langle 1, n \rangle$ i istnieje instrukcja s taka, że $a_n \in p_a(s)$.

Wprowadzone pojęcia zilustrujemy na programie Π z przykładu 6:

$LOK(\Pi_1) = \{i,j,k\}$, $LOK(\Pi_2) = \{a,b,c\}$, $LOK(\Pi_3) = \{d,e,f,l,m\}$
$AKT(s) = \{i,j,k\}$ dla $s \in \Pi_1$
$AKT(s) = \{i,j,k,a,b,c\}$ dla $s \in \Pi_2$
$AKT(s) = \{i,j,k,a,b,c,d,e,f,l,m\}$ dla $s \in \Pi_3$

$\{b,d\}$ jest zbiorem modyfikowanych p.f.n. w programie Π, natomiast $\{b,c,e\}$ jest zbiorem wykorzystywanych p.f.n. w Π.

Pojęcia definicji zmiennej i odwołania do zmiennej wprowadzimy osobno dla wołania procedury i pozostałych instrukcji.

Definicja 29. Niech Π będzie programem w systemie z procedurami SP i s niech będzie dowolną instrukcją programu Π taką, że $o(s) \neq WOL$.

Powiemy, że s jest definicją każdej zmiennej należącej do zbioru $\bigcup_{a \in P_w} (s)\, SYN(a) \cap AKT(s)$.

Ponadto powiemy, że instrukcja s jest odwołaniem do każdej zmiennej należącej do zbioru: $\bigcup_{a \in P_a} (s)\ SYN(a) \cap AKT(s)$.

W bardziej skomplikowany sposób określa się pojęcia definicji i odwołania dla instrukcji wołania procedury.

W tym wypadku zmienna może być definiowana bądź przez instrukcję występującą w wołanej procedurze (w procedurze wołanej w wołanej procedurze, itd) bądź "za pośrednictwem parametru formalnego wołanego przez nazwę". To ostatnie pojęcie określimy nieco dokładniej: niech s będzie dowolną instrukcją wołania procedury w programie Π. Powiemy, że s jest definicją (odwołaniem do) zmiennej za pośrednictwem p.f.n., o ile a jest synonimem pewnego aktualnego parametru takiego, że odpowiadający mu w s parametr formalny jest modyfikowanym (wykorzystywanym) p.f.n. Ponadto powiemy, że s jest odwołaniem do zmiennej a za pośrednictwem p.f.w. (parametru formalnego wołanego przez wartość), o ile a jest synonimem pewnego aktualnego parametru takiego, że odpowiadający mu w s parametr formalny jest p.f.w.

Definicja 30. Niech $\Pi = \Pi_1, \ldots, \Pi_n$ będzie programem w systemie z procedurami i s niech będzie dowolną instrukcją wołania procedury Π_i w Π.

Powiemy, że instrukcja s jest definicją zmiennej a programu Π, o ile jest spełniony jeden z poniższych warunków:

1) s jest definicją a za pośrednictwem p.f.n.
2) istnieje ciąg $\Pi_{k_1}, \ldots, \Pi_{k_m}$ procedur programu Π taki, że
   a) $\Pi_{k_1} = \Pi_i$
   b) $\Pi_{k_j} \neq \Pi_{k_l}$ dla $j \neq l$,
   c) w procedurze Π_{k_j} istnieje instrukcja wołania procedury $\Pi_{k_{j+1}}$ dla $j \in \langle 1, m-1 \rangle$,
   d) $a \notin \bigcup_{j=1} LOK(\Pi_{k_j})$,

oraz istnieje instrukcja s' w procedurze Π_{k_m} taka, że jest spełniony jeden z warunków:

e) $o(s) \neq WOL$ i s' jest definicją zmiennej a
f) s' jest definicją a za pośrednictwem p.f.n.

Powiemy, że instrukcja s jest odwołaniem do zmiennej a programu Π, o ile jest spełniony jeden z poniższych warunków:

1) s jest odwołaniem do a za pośrednictwem p.f.n. lub p.f.w.
2) istnieje ciąg $\Pi_{k_1}, \ldots, \Pi_{k_n}$ procedur programu Π taki, że
   a) $\Pi_{k_1} = \Pi_i$,
   b) $\Pi_{k_j} \neq \Pi_{k_l}$ dla $j \neq l$;
   c) w procedurze Π_{k_j} istnieje instrukcja wołania procedury $\Pi_{k_{j+1}}$ dla $j \in \langle 1, m-1 \rangle$,

d) $a \in \bigcup_{j=k}^{m} LOK(\pi_{k_j})$,

oraz istnieje instrukcja s' w procedurze π_{k_m} taka, że jest spełniony jeden z warunków:

e) $o(s') \neq WOL$ i s' jest definicją zmiennej s,

f) s' jest odwołaniem do a za pośrednictwem p.f.n. lub p.f.w.

Dla każdej instrukcji s przez $DEF(s)$ i $ODW(s)$ będziemy oznaczali zbiory zmiennych odpowiednio: definiowanych przez s i tych zmiennych, dla których s jest odwołaniem. Dla instrukcji nie będącej wołaniem procedury zbiory powyższe wyrażają się prostymi wzorami:

$$DEF(s) = \bigcup_{b \in p_w(s)} SYN(b) \cap AKT(s)$$

$$ODW(s) = \bigcup_{b \in p_a(s)} SYN(b) \cap AKT(s)$$

Obecnie podamy algorytm znajdujący interesujące nas zbiory zmiennych dla instrukcji wołania procedury.

Niech $\pi = \pi_1, \ldots, \pi_n$ będzie programem w systemie programowania z procedurami S^P.

Algorytm 2.

Wejście: program $\pi = \pi_1, \ldots, \pi_n$ w systemie programowania S^P, zbiory $SYN(a)$, $LOK(\pi_i)$ i $AKT(s)$ dla zmiennych, procedur i instrukcji programu π.

Wyjście: zbiór zmiennych definiowanych - $DEF(s)$ i zbiór zmiennych, do których s jest odwołaniem - $ODW(s)$, dla każdej instrukcji wołania procedury w programie π.

Metoda:

1. Dla każdej procedury π_i programu π wykonaj:

$DEF(\pi_i) \leftarrow \{a \in Z_{SP}(\pi) \mid a \in DEF(s)$ dla pewnej instrukcji $s \in \pi_i$ takiej, że $o(s) \neq WOL\} \cup$

$\{a \in Z_{SP}(\pi) \mid$ w procedurze π_i istnieje instrukcja wołania procedury s taka, że s jest definicją a za pośrednictwem p.f.n.$\}$,

$ODW(\pi_i) \leftarrow \{a \in Z_{SP}(\pi) \mid a \in ODW(s)$ dla pewnej instrukcji $s \in \pi_i$ takiej, że $o(s) \neq WOL\} \cup$

$\{a \in Z_S(\pi) \mid$ w procedurze π_i istnieje instrukcja wołania procedury s taka, że s jest odwołaniem do a za pośrednictwem p.f.n. lub p.f.w.$\}$

2. Dla każdej procedury π_i programu π wykonaj:

2.1. dla każdego ciągu $\pi_{k_1}, \ldots, \pi_{k_m}$ procedur programu π takiego, że

a) $\pi_{k_1} = \pi_i$,

b) $\pi_{k_j} \neq \pi_{k_l}$ dla $j \neq l$,

c) w procedurze π_{k_j} istnieje instrukcja wołania procedury $\pi_{k_{j+1}}$ dla $j \in \langle 1, m-1 \rangle$,

wykonaj:

$$DEF(\mathcal{T}_i) \longleftarrow DEF(\mathcal{T}_i) \cup \{a \in Zsp(\mathcal{T}) \mid a \in DEF(\mathcal{T}_{k_m}) \text{ i } a \notin \bigcup_{j=1}^{m} LOK(\mathcal{T}_{k_j})\},$$

$$ODW(\mathcal{T}_i) \longleftarrow ODW(\mathcal{T}_i) \cup \{a \in Zsp(\mathcal{T}) \mid a \in ODW(\mathcal{T}_{k_m}) \text{ i } a \in \bigcup_{j=1}^{m} LOK(\mathcal{T}_{k_j})\}$$

3. Dla każdej procedury $\mathcal{T}_i$ i dla każdej instrukcji s wołania procedury $\mathcal{T}_i$ wykonaj:

$DEF(s) \longleftarrow DEF(\mathcal{T}_i) \cap AKT(s) \cup \{a \in Zsp(\mathcal{T}) \mid$ s jest definiowją a za pośrednictwem p.f.n. $\}$

$ODW(s) \longleftarrow ODW(\mathcal{T}_i) \cap AKT(s) \cup \{a \in Zsp(\mathcal{T}) \mid$ s jest odwołaniem do a za pośrednictwem p.f.n. lub p.f.w. $\}$

Algorytm 2 kończy działanie w skończonym czasie, ponieważ w każdym programie mamy do czynienia ze skończoną liczbą instrukcji i zmiennych. Ponadto istotny jest tu fakt, że w kroku 2 rozpatrujemy jedynie różnowartościowe ciągi procedur programu. W pierwszym kroku algorytmu 2 dla każdej procedury $\mathcal{T}_i$ programu powstaje zbiór zmiennych definiowanych przez instrukcje z procedury $\mathcal{T}_i$ nie będące wołaniem procedury, ewentualnie definiowanych przez instrukcje wołania procedury za pośrednictwem p.f.n. (podobnie zbiór zmiennych do których ma miejsce odwołanie).
W drugim kroku, do poprzedniego zbioru zostają dołączone zbiory zmiennych definiowanych przez procedury, które mogą być uaktywnione na skutek wołania procedury $\mathcal{T}_i$. Wreszcie w ostatnim kroku, dla każdej instrukcji wołania procedury zostaje zbudowany zbiór zmiennych definiowanych na skutek tego wołania (podobnie, zbiór zmiennych, do których ma miejsce odwołanie).

Dla ilustracji opisanej metody raz jeszcze powrócimy do programu z przykładu 6 i znajdziemy zbiory DEF(s) i ODW(s) dla instrukcji
s= , b;c;j, WOŁ,proc3, z procedury . W tym celu, zgodnie z algorytmem22, musimy najpierw zbudować zbiory pomocnicze $DEF(\mathcal{T}_2)$ i $DEF(\mathcal{T}_3)$, $ODW(\mathcal{T}_2)$ i $ODW(\mathcal{T}_3)$.

W pierwszym kroku powstają zbiory:

$DEF(\mathcal{T}_2) = \{k,o,a,e\} \cup \{b,d,j,l,c\} = \{k,o,a,e,b,d,j,l\}$
$ODW(\mathcal{T}_2) = \{b,d,j,l,e,c,k,m,a\} \cup \{c,e,k,m,a,j,b,d\} = \{b,l,j,l,e,c,k,m\}$
$DEF(\mathcal{T}_3) = \{d,b,j,l\} \cup \{l,b,d\} = \{d,b,j,l\}$
$ODW(\mathcal{T}_3) = \{e,c,k,m,a,b,f\} \cup \{l,b,d,m,c,e\} = \{e,c,k,m,a,b,f,l,d\}$

W drugim kroku algorytmu 2 zbiory $DEF(\mathcal{T}_3)$ i $ODW(\mathcal{T}_3)$ zostają uzupełnione:

$DEF(\mathcal{T}_3) = DEF(\mathcal{T}_3) \cup \{k\} = \{d,b,j,l,k\}$
$ODW(\mathcal{T}_3) = ODW(\mathcal{T}_3) \cup \{j\} = \{e,c,k,m,a,b,f,l,d,j\}$

Wreszcie w ostatnim kroku wyliczamy:

$AKT(s) = \{i,j,k,a,b,c\}$ i
$DEF(s) = \{d,b,j,l,k,\} \cap \{i,j,k,a,b,c\} \cup \{b,d,j,l,c\} \cap \{i,j,k,a,b,c\} =$
$= \{b,j,k\}$
$ODW(s) = \{e,c,k,m,a,b,f,d,j\} \cap \{i,j,k,a,b,c\} \cup$

$$\{o,e,k,m,a,j,b,d\} \cap \{i,j,k,a,b,o\} =$$
$$=\{o,k,a,b,j\} .$$

Jak pamiętamy procedura w systemie programowania z procedurami SP jest programem w systemie prostym związanym z SP. Obecnie, mając określone pojęcia definicji zmiennej i odwołania do zmiennej w programach z procedurami, możemy przenieść pojęcia takie jak obliczenie, relacja definicja-odwołanie, spójność, graf, wprowadzone w paragrafach 2 i 3 w rozdziale II dla programów na analogiczne dla procedur programu z procedurami. Ponieważ przeniesienie jest natychmiastowe, nie uważamy za celowe powtarzanie w tym miejscu wszystkich definicji. W dalszym ciągu będziemy się zatem posługiwali pojęciami takimi jak: obliczenie w procedurze, relacja definicja-odwołanie w procedurze, procedura spójna, graf procedury.

Rozdział IV. METODY OPTYMALIZACJI PROGRAMÓW

Celem optymalizacji jest dokonanie w programie zmian dających program równoważny w jakimś sensie i bardziej efektywny. Przez zwiększenie efektywności rozumie się zarówno skrócenie czasu potrzebnego na wykonanie programu, jak i zmniejszenie jego długości. W ogólności, gdy mamy do czynienia z programami zawierającymi pętle, nie da się rozstrzygnąć problemu znalezienia programu optymalnego dla danego. Istnieje jednak szereg transformacji, które zastosowane do programu pozwalają uzyskać lepszy jego wersję /patrz Allen [4] i [6]/. W niniejszym rozdziale omówimy transformacje optymalizujące dla poprzednio zdefiniowanych programów, zarówno z procedurami jak i bez. Aparat formalny jaki wprowadziliśmy w postaci systemów programowania z procedurami wraz z określeniem pojęć definicji zmiennej i odwołania do zmiennej w instrukcjach programu, pozwala na precyzyjne opisanie interesującej, znanej z praktyki klasy przekształceń optymalizujących. Warto przy tym dodać, że jest to pierwsza próba formalizacji tych przekształceń dla programów zawierających pętle i procedury. Dotychczasowe badania w tym kierunku (Aho i Ullman [2] i Matwin [14]) były prowadzone wyłącznie nad tzw. programami liniowymi (tzn. bez pętli i procedur) .

Niewątpliwie interesujące byłoby wzbogacenie wprowadzonego formalizmu o aparat semantyczny, który umożliwiłby udowodnienie, że omawiane przekształcenia rzeczywiście optymalizują programy i zachowują ich równoważność. Temat ten, o czym już mówiliśmy we wstępie, przekraczał zadania jakie sobie stawialiśmy w tej pracy i nie został podjęty.

1. Przekształcenia optymalizacyjne programów

Dwa typy operacji będą odgrywały ważną rolę w omawianych przekształceniach: operacja prostego przypisania i operacja pusta. Instrukcja o operacji prostego przypisania (zwana prostym przypisaniem) charakteryzuje się jednoelementowymi polami argumentów i wyników oraz pustym polem etykiet. Mówiąc nieformalnie, wykonanie tej instrukcji polega na tym, że zmiennej z pola wyników zostaje przypisana wartość zmiennej lub stałej z pola argumentów. Instrukcja o operacji pustej (zwana instrukcją pustą) ma puste pola argumentów, wyników i etykiet. Z jej wykonaniem nie jest związana żadna akcja. W dalszym ciągu będziemy zakładali, że wyżej wymienione typy operacji znajdują się w każdym systemie progra-

mowania bądź systemie programowania z procedurami.

W praktyce znane są przekształcenia, które nie zmieniają "istotnego znaczenia" programów pisanych w rzeczywistych językach pośrednich (patrz Allen [6]). Obecnie przejdziemy do formalnego zdefiniowania niektórych spośród tych przekształceń dla wprowadzonego przez nas modelu języków i programów. Będą to: usuwanie niepotrzebnych instrukcji z programu, redukcja identycznych wyrażeń, wynoszenie instrukcji poza pętlę oraz zamiana odwołań. Nie są to oczywiście wszystkie znane przekształcenia: mamy tu przede wszystkim na myśli wyliczanie wyrażeń (ang: folding) i redukcję mocy operatora.

Transformacje te wnikają bardziej szczegółowo w naturę konkretnego języka i nie są stosowane w stanowiącym przedmiot tej pracy systemie optymalizacyjnym, dlatego też je tu pominęliśmy.

Poniżej podamy przekształcenia programów osobno dla programów w systemach programowania według def. 10 z rozdziału II i dla programów w systemach programowania z procedurami. Jak zobaczymy, w obydwu przypadkach istota przekształceń pozostaje ta sama.

Niech $\mathcal{B} = (Q, Z, S, E, U)$ będzie bazą i $\mathcal{S}(\mathcal{B}) = S = (o, l_o, p_n, p_w, p_u)$ systemem programowania o bazie $\mathcal{B}$.

Przekształcenie T1. W wyniku przekształcenia z programu zostają usunięte instrukcje nie spełniające warunku spójności programu (rozdział II, paragraf 2).

Przekształcenie T2. $T2(\mathcal{T}) = \mathcal{T}'$ dla dowolnych programów $\mathcal{T}$ i $\mathcal{T}'$ takich, że $\mathcal{T} = s_1, \ldots, s_{i-1}, s_i, s_{i+1}, \ldots, s_n$ oraz

$\mathcal{T}' = s_1, \ldots, s_{i-1}, s_i', s_{i+1}, \ldots, s_n$, o ile są spełnione następujące warunki:

1) instrukcja s_i programu jest przypisaniem i dla każdej zmiennej $a \in p_w(s_i)$ instrukcja s_i nie pozostaje w relacji $\Delta_{\mathcal{T}}(a)$ z żadną instrukcją programu $\mathcal{T}$,

2) instrukcja s_i' programu $\mathcal{T}'$ jest instrukcją pustą, przy czym $l_o(s_i) = l_o(s_i')$.

Przekształcenie T3. $T3(\mathcal{T}) = \mathcal{T}'$ dla dowolnych programów $\mathcal{T}$ i $\mathcal{T}'$ takich, że $\mathcal{T} = s_1, \ldots, s_{i-1}, s_i, \ldots, s_n$ oraz

$\mathcal{T}' = s_1, \ldots, s_{i-1}, s', s_i, \ldots, s_n$,

gdzie s' jest przypisaniem definiującym wyłącznie zmienne nie należące do programu $\mathcal{T}$.

Przekształcenie T4. $T4(\mathcal{T}) = \mathcal{T}'$ dla dowolnych programów $\mathcal{T}$ i $\mathcal{T}'$ takich, że

$\mathcal{T} = s_1, \ldots, s_i, \ldots, s_{j-1}, s_j, s_{j+1}, \ldots, s_n$ oraz

$\mathcal{T}' = s_1, \ldots, s_i, \ldots, s_{j-1}, s_{j_1}, \ldots, s_{j_m}, s_{j+1}, \ldots, s_n$

o ile są spełnione warunki:

1) instrukcja s_i dominuje nad s_j w $\mathcal{T}$ oraz $o(s_i) = o(s_j)$, $p_u(s_i) = p_u(s_j)$,

2) dla każdej zmiennej a, będącej elementem $p_u(s_i)$ lub $p_w(s_i)$ i każdego

obliczenia $s_{i_1},\ldots,s_{i_l}$ w $\mathcal{T}$ takiego, że $s_{i_1} = s_i$, $s_{i_l} = s_j$ i $s_{i_k} \neq s_i$ dla $k \in \langle 2,l \rangle$, instrukcja s_{i_k} nie jest definicją zmiennej a dla $k \in \langle 1,l-1 \rangle$;

3) instrukcje $s_{j_1},\ldots, s_{j_m}$ programu $\mathcal{T}'$ są prostymi przypisaniami takimi, że $l_e(s_{j_1}) = l_e(s_j)$, $l_e(s_{j_k}) = \varepsilon$ dla $k \in \langle 2,m \rangle$, $p_w(s_{j_k})$ jest k-tym elementem ciągu $p_w(s_j)$ oraz $p_a(s_{j_k})$ jest k-tym elementem ciągu $p_w(s_i)$ dla $k \in \langle 1,m \rangle$.

Przekształcenie T5. $T5(\mathcal{T}) = \mathcal{T}'$ dla dowolnych programów $\mathcal{T}$ i $\mathcal{T}'$ takich, że

$\mathcal{T} = s_1,\ldots,s_i,s_{i+1},\ldots,s_{j-1},s_j, s_{j+1},\ldots,s_n$ oraz

$\mathcal{T}' = s_1,\ldots,s_i,s'_j,s_{i+1},\ldots,s_{j-1},s''_j, s_{j+1},\ldots,s_n$ i spełnione są warunki:

1) s_j jest przypisaniem, s''_j jest instrukcją pustą, przy czym $l_e(s''_j) = l_e(s_j)$ oraz instrukcja s'_j jest równa s_j za wyjątkiem etykiety, która jest symbolem pustym,

2) instrukcja s'_j dominuje nad s''_j w $\mathcal{T}'$,

3) instrukcja s''_j należy do każdego obliczenia w $\mathcal{T}'$ o początku s'_j i końcu s_n,

4) dla każdej zmiennej a należącej do pola wyników lub argumentów instrukcji s'_j i każdego obliczenia $s_{j_1},\ldots,s_{j_l}$ w $\mathcal{T}'$ takiego, że $s_{j_1} = s'_j$, $s_{j_l} = s''_j$ i $s_{j_k} \neq s'_j$ dla $k \in \langle 2,l \rangle$, instrukcja s_{j_k} nie jest definicją zmiennej a dla $k \in \langle 1,l-1 \rangle$;

5) dla każdej zmiennej a $p_w(s'_j)$ i każdego obliczenia $s_{j_1},\ldots,s_{j_l}$ w $\mathcal{T}'$ takiego, że $s_{j_1} = s'_j$, $s_{j_l} = s''_j$ oraz $s_{j_k} \neq s'_j$, $s_{j_k} \neq s''_j$ dla $k \in \langle 2,l-1 \rangle$ instrukcja s_{j_k} nie jest odwołaniem do zmiennej a dla $k \in \langle 2,l-1 \rangle$.

Przekształcenie T6. $T6(\mathcal{T}) = \mathcal{T}'$ dla dowolnych programów $\mathcal{T}$ i $\mathcal{T}'$ takich, że

$\mathcal{T} = s_1,\ldots,s_i,\ldots,s_{j-1}, s_j, s_{j+1},\ldots s_n$ oraz

$\mathcal{T}' = s_1,\ldots,s_i,\ldots,s_{j-1}, s_j, s_{j+1},\ldots, s_n$ o ile są spełnione warunki:

1) s_i jest prostym przypisaniem, s_j jest przypisaniem oraz $p_w(s_i) \in p_a(s_j)$

2) s_i dominuje nad s_j w $\mathcal{T}$,

3) s_i jest jedyną instrukcją w $\mathcal{T}$ taką, że $(s_i,s_j) \in \Delta_{\mathcal{T}}(p_w(s_i))$,

4) dla każdego obliczenia $s_{i_1},\ldots,s_{i_l}$ w $\mathcal{T}$ takiego, że $s_{i_1} = s_i$, $s_{i_l} = s_j$ i $s_{i_k} \neq s_i$ dla $k \in \langle 2,l \rangle$. zmienna $p_a(s_i)$ nie jest definiowana przez żadną instrukcję spośród $s_{i_2},\ldots,s_{i_{l-1}}$,

5) $l_o(s'_j) = l_e(s_j)$, $o(s'_j) = o(s_j)$, $p_w(s'_j) = p_w(s_j)$, $p_e(s'_j) = p_e(s_j)$ oraz ciąg $p_a(s'_j)$ powstał z $p_a(s_j)$ przez zastąpienie każdego wystąpienia zmiennej $p_w(s_i)$ na zmienną $p_a(s_i)$.

Przekształcenie T7. $T7(\mathcal{T}) = \mathcal{T}'$ dla dowolnych programów $\mathcal{T}$ i $\mathcal{T}'$ takich, że

$\mathcal{T} = s_1,\ldots,s_i,s_{i+1},\ldots,s_n$ oraz $\mathcal{T} = s_1,\ldots,s'_i,s'',s_{i+1},\ldots,s_n$ o ile są spełnione warunki:

1) s_i jest instrukoją przypisania,

2) s'_i jest instrukoją przypisania, w której $l_e(s'_i) = l_e(s)$, $o(s'_i) = o(s_i)$, $p_a(s'_i) = p_a(s_i)$, natomiast pole wyników instrukoji s' powstało z pola wyników instrukoji s_i przez zastąpienie jednego z jego elementów, powiedzmy a, przez zmienną t nie występującą w programie $\mathcal{T}$,

3) s' jest instrukoją prostego przypisania, w której $l_e(s') = \varepsilon$, $p_a(s') = t$ i $p_w(s') = a$.

Przechodzimy teraz do przekształceń programów z procedurami. W tym przypadku jednostką przekształcaną jest wprawdzie pojedyncza procedura, niemniej jednak musimy ją rozpatrywać w kontekście całego programu, do którego ta procedura należy.

Niech $\mathcal{B} = (Q, Z, S, E, U)$ będzie bazą i $S^P = S^P(\mathcal{B}) = (S\mathcal{B}, \{PROC, KON, WOL, MPROC, DKL\}, p_n, p_r)$ systemem programowania z procedurami o bazie $\mathcal{B}$. Ponadto, niech $\mathcal{T}$ i $\mathcal{T}'$ będą programami w systemie S^P takimi, że $\mathcal{T} = \mathcal{T}_1, \ldots, \mathcal{T}_{k-1}, \mathcal{T}_k, \mathcal{T}_{k+1}, \ldots, \mathcal{T}_n$ i $\mathcal{T}' = \mathcal{T}_1, \ldots, \mathcal{T}_{k-1}, \mathcal{T}'_k, \mathcal{T}_{k+1}, \ldots, \mathcal{T}_n$. Założymy, że przekształcenia dotyczą procedury $\mathcal{T}_k$ programu $\mathcal{T}$.

Przekształcenia T1 i T3-T7 pozostają zasadniczo bez zmian. Jedyna zmiana, czysto formalna, polega na zastąpieniu sformułowań: "program $\mathcal{T}$", "program $\mathcal{T}'$", "relacja $\Delta_{\mathcal{T}}$", odpowiednio na: "procedura $\mathcal{T}_k$", "procedura $\mathcal{T}'_k$", "relacja $\Delta_{\mathcal{T}_k}$". W przekształceniu T2 zachodzą bardziej istotne zmiany, w związku z tym przytoczymy je w całości:

Przekształcenie T2. $T2(\mathcal{T}_k) = \mathcal{T}'_k$ dla dowolnych procedur $\mathcal{T}_k$ i $\mathcal{T}'_k$ takich, że $\mathcal{T}_k = s_1, \ldots, s_{i-1}, s_i, s_{i+1}, \ldots, s_n$ oraz $\mathcal{T}_k = s_1, \ldots, s_{i-1}, s'_i, s_{i+1}, \ldots, \ldots, s_n$ o ile są spełnione następujące warunki:

1) instrukcja s_i procedury $\mathcal{T}_k$ jest przypisaniem i dla każdej zmiennej $a \in p_w(s_i)$ zachodzi:

a) jeśli a jest zmienną lokalną procedury $\mathcal{T}_k$ nie będącą p.f.n., to s_i nie pozostaje w relacji $\Delta_{\mathcal{T}_i}(a)$ z żadną instrukcją procedury $\mathcal{T}_k$,

b) jeśli a jest zmienną zewnętrzną procedury $\mathcal{T}_k$ lub p.f.n., to dla każdego obliczenia w $\mathcal{T}_k$: $s_{i_1}, \ldots, s_{i_l}$, takiego, że $s_{i_1} = s_i$, $s_{i_l} = s_n$, istnieje $m \in \langle 2, l \rangle$, że $a \in p_w(s_{i_m})$ i $a \notin p_a(s_{i_j})$ dla $j \in \langle 1, m \rangle$.

2) instrukcja s'_i procedury $\mathcal{T}'_k$ jest instrukoją pustą, przy czym $l_e(s_i) = l_o(s'_i)$.

Obecnie przechodzimy do zilustrowania zdefiniowanych przekształceń na przykładach programów systemu programowania $\bar{S}(\bar{\mathcal{B}})$ opisanego w przykładzie 3 rozdziału II. Wybraliśmy prosty system programowania (bez procedur) aby uniknąć skomplikowanych rachunków na jakie natrafilibyśmy w przypadku programów z procedurami. Podkreślamy jednak, że istota postępowania pozostaje taka sama w przypadku programów z procedurami.

2. Usuwanie niepotrzebnych instrukcji z programu

Rozważmy następujący program $\mathcal{T}$:

s_1: , a , CZYTAJ, ,

```
s2:        , b  , CZYTAJ,      ,
s3:        , c  ,    +  , a;b  ,
s4:    et1 , d  ,       , c;a  ,
s5:        , a  ,    +  , a;b  ,
s6:        ,    ,  ↑≤   , a;b,et1; et2,
s7:        , a  ,  :=    , 5    ,
s8:        , b  ,  -     , b;a  ,
s9:    et2 ,    , PISZ   , a    ,
s10:       ,    , PISZ   , b    ,
```

Program π

Instrukcje s_7 i s_8 programu π nie należą do żadnej realizacji programu, zatem nie spełniają warunków spójności i mogą być usunięte z programu.

W wyniku usunięcia z programu wszystkich instrukcji naruszających warunek spójności (przekształcenie T1) otrzymujemy program spójny. Instrukcje nie spełniające warunku spójności programu są usuwane na początku optymalizacji i w związku z tym, bez zmniejszenia ogólności możemy przyjąć, że przedmiotem następnych przekształceń optymalizujących są programy spójne.

Istnieje jeszcze inny rodzaj instrukcji zbędnych w programie: zauważmy, że wartość zmiennej d wyliczona w instrukcji s_4 programu π nie jest wykorzystana, mówiąc ściślej instrukcja s_4 nie pozostaje w relacji $\Delta_{\pi}(d)$ z żadną instrukcją programu. Usuwanie takich instrukcji z programu opisuje przekształcenie T2. Zauważmy dalej, że w wyniku usunięcia z programu π instrukcji s_4, instrukcja s_3 staje się niepotrzebna. Po usunięciu z programu π wszystkich instrukcji niepotrzebnych, otrzymujemy następujący program π':

```
          , a  , CZYTAJ,      ,
          , b  , CZYTAJ,      ,
          ,    ,  NOP   ,     ,
    et1   ,    ,  NOP   ,     ,
          , a  ,   +    , a;b,
          ,    ,  ↑≤    , a;b, et 1; et2
    et2   ,    , PISZ   , a   ,
          ,    , PIS2   , b   ,
```

Program π'

3. Redukcja identycznych wyrażeń

Często w programie występują wielokrotnie instrukcje o identycznych wyrażeniach (tzn. identycznych operacjach i polach argumentów). W niektórych sytuacjach można zastąpić powtórne wyliczenie wyrażenia odwołaniem do zmiennej mającej odpowiednią wartość. Przekształcenie T4 zawiera opis takich sytuacji i podaje metodę wykonania redukcji.

Weźmy pod uwagę następujący program:

```
s1:        , a  , CZYTAJ,
s2:        , b  , CZYTAJ,
s3:        , c  ,   +    , a;b ,
```

```
s4:        ,  d  ,   +   , a;b ,
s5:   et1  ,  c  ,   *   , d;c ,
s6:        ,  d  ,   /   , d;c ,
s7:        ,  ε  ,   +   , a;b ,
s8:        ,     ,  ↑≤   , ε;5, et1; et2,
s9:   et2  ,     , PISZ  , d   ,
```

Program $\mathcal{T}$

Przyjrzyjmy się instrukcjom s_3 i s_4 programu $\mathcal{T}$ i zauważmy, że spełniają one warunki 1) i 2) wyszczególnione w przekształceniu T4: s_3 dominuje nad s_4, wyrażenia instrukcji s_3 i s_4 są identyczne, zmienne a, b i c nie są definiowane przez instrukcje żadnego obliczenia o początku s_3 i końcu s_4. Zatem w myśl przekształcenia T4 możemy instrukcję s_4 w programie $\mathcal{T}$ zamienić na następującą instrukcję s_4': et1,d, := ,c, . Wyżej opisane przekształcenie programu nosi nazwę redukcji identycznych wyrażeń. Przyjrzyjmy się jeszcze instrukcjom s_3 i s_7 programu $\mathcal{T}$. Instrukcje te nie spełniają tylko jednego spośród założeń wymienionych w punktach 1) i 2) przekształcenia T4, mianowicie zmienna c jest definiowana przez instrukcję s_5 należącą do obliczenia o początku s_3 i końcu s_7. W tym przypadku możemy przeprowadzić redukcję w następujący sposób: przed instrukcję s_3 programu wstawiamy instrukcję s' = , t , +, a;b, Pozwala na to przekształcenie T3, ponieważ zmienna t nie występuje w programie $\mathcal{T}$ i s' jest przypisaniem. Zauważmy, że obecnie opierając się na instrukcji s' możemy zredukować wyrażenia w instrukcjach s_3 i s_7. Następujący program $\mathcal{T}''$ jest wynikiem dokonania redukcji identycznych wyrażeń w programie $\mathcal{T}$:

```
        ,  a  , CZYTAJ,     ,
        ,  b  , CZYTAJ,     ,
        ,  t  ,   +   , a;b ,
        ,  o  ,   :=  ,  t  ,
        ,  d  ,   :=  ,  o  ,
   et1  ,  c  ,       , d;c ,
        ,  d  ,   /   , d;c ,
        ,  ε  ,   :=  ,  t  ,
        ,     ,       , ε;5 , et1; et2
   et2  ,     , PISZ  , d   ,
```

Program $\mathcal{T}''$

4. Wyniesienie instrukcji poza pętle

W każdym bardziej złożonym programie istnieją fragmenty wykonywane wielokrotnie, tzw. pętle. Ważnym przekształceniem optymalizacyjnym jest przesuwanie instrukcji z pętli w miejsca programu wykonywane z mniejszą częstością.

Weźmy pod uwagę następujący program $\mathcal{T}$:

```
s1:        ,  a  , CZYTAJ ,    ,
s2:        ,  b  , CZYTAJ ,    ,
s3:        ,  ε  ,   /    , a;b,
s4:   et1  ,  o  ,   *    , a;n,
```

```
s5:        ,  d  ,   /    , o;ε,
s6:        ,  ε  ,   +    , a;b,
s7:        ,     , ↑ ≤    , o;d, et1, et2
s8:   et2  ,     , PISZ   , o  ,
s9:        ,     , PISZ   , d  ,
```

Program Π

W programie Π można wyróżnić jedną pętlę, mianowicie $L = \{s_4, s_5, s_6, s_7\}$. Warunki jakie musi spełniać instrukcja, aby można było ją przemieścić w programie oraz metoda przesuwania są podane w przekształceniu T5. Opierając się na tym wynosimy poza pętlę instrukcję o_4 i uzyskujemy następujący program Π':

```
      ,  a  , CZYTAJ ,    ,
      ,  b  , CZYTAJ ,    ,
      ,  ε  ,   /    , a;b,
      ,  o  ,        , a;b,
et1   ,     , NOP    ,    ,
      ,  d  ,   /    , o;ε,
      ,  ε  ,   +    , a;b,
      ,     , ↑ ≤    , o;d, et1; et2
et2   ,     , PISZ   , o  ,
      ,     , PISZ   , d  ,
```

Program Π'

Zauważmy, że operacja mnożenia w programie Π' jest wykonywana poza pętlą, co czyni program Π' bardziej efektywnym niż Π. Do żadnej innej instrukcji pętli L nie da się zastosować przekształcenia T5. Niemniej można, nieco okrężną drogą, wynieść poza pętlę instrukcję dodawania s_6 w następujący sposób:

Najpierw, zgodnie z przekształceniem T7 zamieniamy instrukcję s_6 na dwie instrukcje:

```
s'6   ,  t  ,   +    , a;b ,
s'    ,  ε  ,   =    , t   ,
```

Obocnie możemy wykonując przekształcenie T5 wynieść instrukcję s_6 poza pętlę. Wynikiem tych przekształceń jest następujący program Π'':

```
      ,  a  , CZYTAJ ,    ,
      ,  b  , CZYTAJ ,    ,
      ,  ε  ,   /    , a;b,
      ,  o  ,   *    , a;b,
      ,  t  ,   +    , a;b,
et1   ,     , NOP    ,    ,
      ,  d  ,   /    , o;ε,
      ,  ε  ,   :=   , t  ,
      ,     , ↑ ≤    , o;d, et1; et2
et2   ,     , PISZ   , o  ,
      ,     , PISZ   , d  ,
```

Program Π''

5. Usuwanie prostych przypisań

Rozpatrzmy następujący program $\mathcal{T}$:

```
s1:  ,  a  , CZYTAJ ,     ,
s2:  ,  b  , CZYTAJ ,     ,
s3:  ,  c  ,   :=  ,  a   ,
s4:  ,  d  ,   +   , c;b  ,
s5:  ,  b  ,   +   , c;d  ,
s6:  ,     , PISZ  ,  b   ,
s7:  ,     , PISZ  ,  d   ,
```

Program $\mathcal{T}$

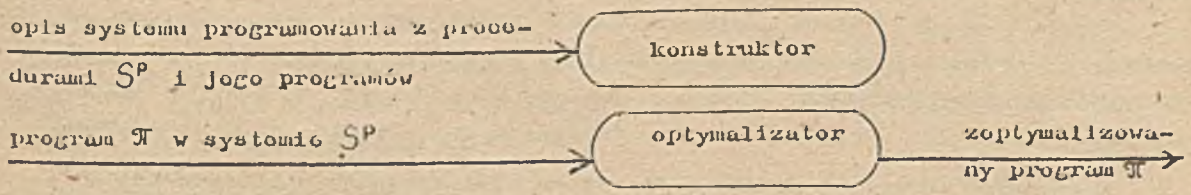
Zmienna c występująca w polach argumentów instrukcji s_4 i s_5 reprezentuje tę samą wartość co zmienna a. Nie zmieniając znaczenia programu można więc zamienić wystąpienia zmiennej c w polach argumentów instrukcji s_4 i s_5 na zmienną a. W ogólności, wszystkie sytuacje, w których możliwa jest podobna zamiana, są opisane w przekształceniu T6. Zauważmy, że po dokonaniu zamiany odwołań instrukcja s_3 programu staje się niepotrzebna i może być usunięta z programu:

```
,  a  , CZYTAJ ,     ,
,  b  , CZYTAJ ,     ,
,     ,  NOP  ,     ,
,  a  ,   +   , a;b  ,
,  b  ,   +   , a;d  ,
,     , PISZ  ,  b   ,
,     , PISZ  ,  d   ,
```

Rozdział V. SYSTEM AUTOMATYCZNEJ KONSTRUKCJI OPTYMALIZATORÓW DLA JĘZYKÓW POŚREDNICH Z PROCEDURAMI

Obecnie przechodzimy do omówienia głównego wyniku naszych badań. Celem naszym jest, przypomnijmy, skonstruowanie uniwersalnego optymalizatora mogącego optymalizować programy dowolnego systemu programowania z procedurami. Rozwiązanie jakie proponujemy, tzw. system optymalizacyjny składa się z dwóch części: konstruktora i optymalizatora. Idea działania systemu jest następująca: konstruktor wczytuje informacje związane z konkretnym systemem programowania - podane w formie specjalnego opisu - i na tej podstawie modyfikuje optymalizator tak, aby mógł on optymalizować programy tego systemu.

Na schemacie można to wyrazić w następujący sposób:



Schemat konstrukcji systemu optymalizacyjnego

Projekt systemu optymalizacyjnego powstał w toku prac prowadzonych pod kierunkiem doc. Jana Borowca nad metatranslatorem w Zakładzie Teorii Translatorów IMM w Warszawie. System optymalizacyjny został zaprogramowany w języku PL/I i uruchomiony na maszynie IBM/370. W przyszłości stanie się on modułem wspomnianego metatranslatora.

W kolejnych paragrafach omówimy opis systemu programowania potrzebny dla konstruktora oraz funkcje jakie spełnia optymalizator.

1. Wejście konstruktora systemu optymalizacyjnego

Metajęzykiem opisu wejścia konstruktora jest język programowania PL/I. Wejście konstruktora stanowi opis konkretnego systemu programowania, dla którego programów chcemy uzyskać optymalizator. Jest to ciąg instrukcji języka PL/I, na który głównie składają się deklaracje szeregu procedur standardowych i jednej tablicy. Procedury standardowe podają metodę dostępu do instrukcji programu i do poszczególnych pól instrukcji oraz umożliwiają przeprowadzenie zmian w programie i jego instrukcjach, natomiast tablica jest opisem zbioru operacji opisywanego systemu programowania. Nazwy procedur standardowych i tablicy są z góry ustalone i w opisie każdego systemu programowanie takie same. Dzięki temu konstruktor włączając opis do programu optymalizatora umożliwia mu korzystanie z informacji zawartych w tablicy i wykonywanie czynności realizowanych przez procedury standardowe.

Oprócz tych podstawowych i zawsze obecnych elementów, wejście konstruktora może obejmować deklaracje dowolnych innych obiektów języka PL/I, pod warunkiem, że ich nazwy zaczynają się od symbolu $. Ograniczenie to ma na celu uniknięcie kolizji nazw.

Ponieważ będziemy mówili jednocześnie o procedurach programów systemów programowania i o procedurach w języku PL/I, stanowiących opis systemu programowania, więc umówimy się te drugie nazywać procedurami standardowymi.

Zanim przejdziemy do szczegółowego przedstawienia poszczególnych składowych wejścia konstruktora, poczynimy dwie uwagi na temat rozpatrywanych systemów programowania:

1/ Będziemy zakładali, że zbiór operacji systemu jest skończony i elementy jego są ponumerowane kolejnymi liczbami naturalnymi w sposób różnowartościowy. W dalszym ciągu będziemy się odwoływali do operacji systemu przez podawanie ich numerów.

2/ Elementy zbiorów zmiennych, stałych i etykiet mogą mieć rozmaitą postać dla różnych systemów programowania. Aby zapewnić możliwość rozpoznawania elementów tych zbiorów założymy, że istnieje różnowartościowe przekształcenie tych zbiorów w zbiór napisów o długości od 1 do 256 znaków. Dzięki temu każdą zmienną, stałą lub etykietą dowolnego systemu można zidentyfikować przez podanie napisu oznaczającego ten obiekt. W dalszym ciągu napisy, o których mowa będziemy utożsamiali z elementami zbiorów zmiennych, stałych, etykiet systemu.

Przypuśćmy teraz, że zadany jest system programowania z procedurami SP, dla którego chcemy za pomocą systemu optymalizacyjnego otrzymać optymalizator. W tym celu musimy sporządzić i podać na wejście konstruktora opis systemu SP

i jego programów. Najpierw omówimy opis programów systemu S^P. Składa się na niego szereg procedur standardowych, które stanowią narzędzie umożliwiające pracę na programach systemu S^P. Załóżmy więc, że $\pi = \pi_1, .., \pi_n$ jest dowolnym programem systemu S^P. Poniższe procedury standardowe stanowią opis programów systemu S^P. Dla każdej procedury opiszemy parametry formalne i omówimy jej znaczenie posługując się przy tym terminami języka PL/I.

WCZYTAJ_PROC: PROC (I,P) ;

I jest parametrem typu FIXED BINARY, P jest parametrem typu POINTER.

Znaczenie: procedura sprowadza do pamięci operacyjnej procedurę programu π o numerze zadanym przez parametr I. Na wyjściu P wskazuje na pierwszą instrukcję sprowadzonej procedury. Jeśli I jest numerem nie istniejącej procedury, tzn. wartość I leży poza przedziałem $<1,n>$, to P otrzymuje wartość NULL.

WYPISZ_PROC : PROC (I) :

I jest parametrem typu FIXED BINARY.

Znaczenie: procedura powoduje wypisanie z pamięci operacyjnej na urządzenie zewnętrzne procedury programu π o numerze zadanym parametrem I.

DAJ_NAST_INST : PROC (P,Q) ;

P i Q są parametrami typu POINTER.

Znaczenie: procedura określa następstwo instrukcji w programie π . Na wejściu P wskazuje instrukcję z procedury π_i znajdującej się w pamięci operacyjnej. Na wyjściu Q wskazuje następną względem P instrukcję. Jeśli P wskazuje ostatnią instrukcję procedury π_i , to Q wraca z wartością NULL.

DAJ_POP_INST : PROC (P,Q) :

P i Q są parametrami typu POINTER.

Znaczenie: procedura określa następstwo instrukcji w programie π . Na wejściu P wskazuje instrukcję z procedury π_i znajdującej się w pamięci operacyjnej. Na wyjściu Q wskazuje poprzednią względem P instrukcję. Jeśli P wskazuje pierwszą instrukcję procedury π_i to Q wraca z wartością NULL.

GEN_ INST : PROC (P,I) ;

P jest parametrem typu POINTER, I jest parametrem typu FIXED BINARY.

Znaczenie: procedura generuje nową instrukcję i wstawia ją w określone miejsce programu π . Na wejściu P wskazuje instrukcję programu π , a I jest numerem pewnego elementu zbioru operacji. Procedura powoduje wygenerowanie instrukcji o operacji I i wstawienie tej instrukcji do programu π bezpośrednio po instrukcji P.

WYRZ_INST : PROC(P) ;

P jest parametrem typu POINTER.

Znaczenie: procedura umożliwia usunięcie określonej instrukcji z programu π . Na wejściu P wskazuje instrukcję programu π . W wyniku działania procedury zostaje usunięta z programu π instrukcja P.

PRZESUN_INST : PROC(P,Q) ;

P i Q są parametrami typu POINTER.

Znaczenie: procedura powoduje przesunięcie instrukcji w programie $\mathcal{T}$. Na wejściu P i Q wskazują pewne instrukcje programu $\mathcal{T}$. W wyniku działania procedury instrukcja P zostaje usunięta z miejsca, w którym się znajduje i wstawiona za instrukcję Q.

GEN_ET : PROC (E);

E jest parametrem typu CHAR (256).

Znaczenie: procedura pozwala uzyskać nową etykietę, nie występującą w programie $\mathcal{T}$. Na wyjściu E oznacza tę właśnie etykietę.

GEN_ZM : PROC(B) ;

B jest parametrem typu CHAR(256).

Znaczenie: procedura pozwala uzyskać nową zmienną, nie występującą w programie $\mathcal{T}$. Na wyjściu B oznacza tę właśnie zmienną.

Obecnie omówimy elementy wejścia konstruktora stanowiące opis systemu programowania z procedurami. Rozpoczynamy od tablicy opisu operacji systemu:

```
1  STR_OP_INST  (IL_OP+1)  ,
   2  TAK_NIE     BIT  (11) ,
   2  IL_ARG      FIXED BINARY,
   2  IL_WYN      FIXED BINARY,
   2  IL_ETYK     FIXED BINARY
```

Wymiar tablicy jest o jeden większy od liczby operacji systemu, dodatkowe miejsce jest rezerwowane dla potrzeb optymalizacji. Operacji o i-tym numerze odpowiada i-ty wiersz w tablicy STR_OP_INST. Wektor bitowy o nazwie TAK_NIE opisuje typ operacji, i tak - jedynka na kolejnych pozycjach tego wektora oznacza, że dana operacja jest:

1) operacją sterującą,
2) operacją pustą,
3) przypisaniem,
4) prostym przypisaniem,
5) operacją wejścia-wyjścia,
6) deklaracją,
7) nagłówkiem procedury,
8) symbolem końca procedury,
9) wołaniem procedury,
10) marką procedury.

Ostatni bit jest wypełniony w trakcie optymalizacji.

Zmienne IL_ARG, IL_WYN i IL_ETYK podają odpowiednio długość pola argumentów, wyników i etykiet instrukcji o danej operacji.

Oprócz tablicy operacji opis systemu programowania tworzą następujące procedury standardowe:

DAJ_OP : PROC (P,I) ;

P jest parametrem typu POINTER, I jest parametrem typu FIXED BINARY.

Znaczenie: procedura rozpoznaje operację instrukcji. Na wejściu P wskazuje pewną instrukcję. Na wyjściu I ma wartość równą numerowi operacji instrukcji P.

DAJ_ET : PROC (P,E);

P jest parametrem typu POINTER, E jest parametrem typu CHAR (256) .

Znaczenie: procedura rozpoznaje etykietę instrukcji. Na wejściu P wskazuje pewną instrukcję. Na wyjściu E oznacza etykietę instrukcji P (założymy, że etykiecie pustej odpowiada pusty napis) .

DAJ_ARG : PROC (P,A) ;

P jest parametrem typu POINTER, A jest tablicą o elementach typu CHAR (256).

Znaczenie: procedura rozpoznaje pole argumentów instrukcji. Na wejściu P wskazuje pewną instrukcję. Na wyjściu elementy tablicy A zawierają zmienne i stałe z pola argumentów instrukcji P.

DAJ_WYN : PROC (P,A) ;
DAJ_PET : PROC (P,A) ;
DAJ_PAR : PROC (P,A) ;

Procedury DAJ_WYN, DAJ_PET i DAJ_PAR spełniają analogiczną rolę dla pola wyników, pola etykiet i pola parametrów instrukcji.

DAJ_NAZ : PROC (P,B) ;

P jest parametrem typu POINTER, B jest parametrem typu CHAR (256) .

Znaczenie: procedura rozpoznaje pole nazwy instrukcji. Na wejściu P wskazuje pewną instrukcję. Na wyjściu B ma wartość będącą elementem z pola nazwy instrukcji P.

WST_ET : PROC (P,E) ;

P jest parametrem typu POINTER, E jest parametrem typu CHAR (256) .

Znaczenie: procedura umożliwia zmianę etykiety instrukcji. Na wejściu P wskazuje pewną instrukcję, a E oznacza etykietę. Na wyjściu w instrukcji P została wstawiona etykieta zdana przez E.

WST_ARG : PROC (P,B,I) ;

P jest parametrem typu POINTER, B jest parametrem typu CHAR (256) i I jest parametrem typu FIXED BINARY.

Znaczenie: procedura umożliwia zmianę pola argumentów instrukcji. Na wejściu P wskazuje pewną instrukcję, B oznacza zmienną lub stałą systemu programowania, a I pewną liczbę. Na wyjściu I-ta pozycja pola argumentów instrukcji P została zamieniona na element zadany przez B.

WST_WYN : PROC (P,B,I) ;
WST_PET : PROC (P,B,I) ;

Procedury WST_WYN i WST_PET spełniają analogiczną rolę dla pola wyników i pola etykiet instrukcji.

CZY_STAL : PROC (B);

B jest parametrem typu CHAR (256).

Znaczenie: jest to procedura funkcyjna i rozróżnia czy dany napis oznacza zmienną czy stałą systemu programowania. Na wejściu wartość zmiennej B jest zmienną lub stałą systemu programowania. Procedura wraca z wartością "prawda" jeśli B oznacza stałą i z wartością "fałsz" jeśli B oznacza zmienną.

IL_PAR : PROC (P,I) ;

P jest parametrem typu POINTER, I jest parametrem typu FIXED BINARY.

Znaczenie: procedura pozwala ustalić długość pola parametrów instrukcji. Na wejściu P wskazuje pewną instrukcję. Na wyjściu parametr I podaje długość pola parametrów instrukcji P.

ROZDZ_WOL : PROC (P,A) ;

P jest parametrem typu POINTER, A jest tablicą o elementach typu FIXED BINARY.

Znaczenie: procedura rozpoznaje rodzaj wołania parametrów procedury. Na wejściu P wskazuje instrukcję nagłówka procedury. Na wyjściu i-ty element tablicy A zawiera 1, jeśli i-ty parametr formalny procedury jest wołany przez wartość, a 2 jeśli i-ty parametr formalny procedury jest wołany przez nazwę.

Jak widzimy, powyższy opis zawiera informacje związane z konkretną reprezentacją instrukcji i programów w danym systemie programowania z procedurami. Optymalizator, dzięki konstruktorowi otrzymuje więc mechanizm pozwalający na skuteczne badanie i przekształcenie programów danego systemu programowania.

2. Optymalizator

Opis konkretnego systemu programowania sporządzony w języku PL/I, w postaci tablicy operacji i standardowych procedur zostaje włączony przez konstruktor do programu optymalizatora. Tak zmodyfikowany optymalizator może optymalizować programy tego systemu programowania.

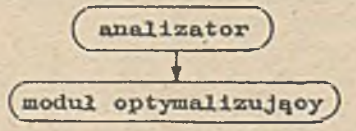
Przypuśćmy, że konstruktor na podstawie opisu wyprodukował optymalizator dla programów pewnego systemu programowania z procedurami:

$S^P = S^P(\mathcal{B}) = (S(\mathcal{B}),\ \{PROC, KON, WOL, MPROC, DKL\},\ p_n, p_r)$, gdzie

$\mathcal{B} = (Q, Z, S, E, U)$ oraz $S(\mathcal{B}) = (o, l_e, p_a, p_w, p_e)$

Omówimy działanie optymalizatora dla dowolnego programu $\mathcal{T} = \mathcal{T}_1, .. \mathcal{T}_n$ w tym systemie.

Optymalizator składa się z dwóch części logicznych: analizatora i modułu optymalizującego.



Zadaniem analizatora jest zebranie informacji o programie $\mathcal{T}$ istotnych dla optymalizacji. Możemy wyróżnić następujące fazy w działaniu analizatora:

1) zbudowanie zbiorów zmiennych lokalnych i zmiennych aktywnych dla procedur programu,

2) wyliczenie zbiorów synonimów dla zmiennych programu,

3) określenie dla każdej instrukcji programu zbioru zmiennych definiowanych przez instrukcję i zbioru zmiennych, do których instrukcja jest odwołaniem,

4) zbudowanie grafów dla poszczególnych procedur programu,

5) wyliczenie relacji bezpośredniej dominacji dla grafów procedur programu,

6) utworzenie list regionów dla grafów procedur programu.

Poniżej omawiamy jak są realizowane poszczególne zadania.

ad.1 Zbiór zmiennych lokalnych dla procedury π_i programu π jest wyliczalny według wzoru:

$$LOK(\pi_i) = \{a \in Z_{SP}(\pi) \mid \text{istnieje w } \pi_i \text{ instrukcja deklaracji } s \text{ taka, że } a \in p_r(s) \text{ lub } a \text{ jest parametrem formalnym procedury } \pi_i\}.$$

Zbiór zmiennych aktywnych dla dowolnej instrukcji s procedury π_i programu π zostaje obliczony ze wzoru:

$$AKT(s) = LOK(\pi_i) \cup \{a \in Z_{SP}(\pi) \mid \text{istnieje procedura } \pi_j \text{ programu } \pi \text{ taka, że } \pi_j \text{ obejmuje } \pi_i \text{ i } a \in LOK(\pi_j)\}$$

ad.2 Wyliczanie zbiorów synonimów dla zmiennych programu odbywa się zgodnie z algorytmem 1. opisanym w rozdziale III.4.

ad.3 Zbiory SYN (a) , LOK(π_i)i AKT(s) , które zostały poprzednio wyliczone pozwalają na znalezienie zbiorów DEF(s) i ODW(s) dla instrukcji programu. Dla instrukcji s, nie będącej wołaniem procedury, wspomniane zbiory są obliczane według wzorów:

$$DEF(s) = \bigcup_{a \in p_w(s)} SYN(a) \cap AKT(s),$$

$$ODW(s) = \bigcup_{a \in p_a(s)} SYN(a) \cap AKT(s),$$

Dla instrukcji wołania procedury, zbiory te są wyliczane według algorytmu 2, opisanego w rozdziale III.4.

ad.4 Każda procedura π_i programu π zostaje podzielona na bloki. Następnie z procedury π_i zostają usunięte instrukcje nie spełniające warunków spójności i powstaje graf dla procedury π_i .

Algorytm 3

Wejście: procedura π_i programu π

Wyjście: spójna procedura π_i , graf $G_{\pi_i} = (X_{\pi_i}, \Gamma_{\pi_i}, \xi_{\pi_i})$ procedury π_i

Metoda:

1. Wykonaj: $X_{\pi_i} \leftarrow \emptyset$, $\Gamma_{\pi_i} \leftarrow \emptyset$

2. Dla każdej instrukcji s z procedury π_i będącej wejściem bloku (patrz def.18 z § 3 rozdz.II) utwórz związany z tym blokiem węzeł i umieść go w zbiorze X_{π_i} . Węzeł zawierający instrukcję nagłówka procedury π_i zaznacz jako węzeł początkowy grafu, ξ_{π_i} .

3. Dla dowolnych węzłów x i y ze zbioru X_{π_i} wykonaj: jeśli instrukcje s i s' będące odpowiednio instrukcją wyjściową węzła x i wejściową węzła y spełniają jeden z następujących warunków:

a) $l_e(s') \in p_e(s)$,

b) $p_e(s) = \xi$, $l_e(s') \neq \xi$ i s' występuje bezpośrednio po s w procedurze π_i,

to dołącz parę (x,y) do relacji Γ_{π_i}.

4. Dla każdego węzła $x \in X_{\pi_i}$ takiego, że nie istnieje droga w G_{π_i} o początku ξ_{π_i} i końcu x, wykonaj:

4.1. Usuń z procedury blok odpowiadający węzłowi x.

4.2. Wykonaj: $X_{\pi_i} \leftarrow X_{\pi_i} \setminus \{x\}$

4.3. Dla każdej pary $(y,z) \in \Gamma_{\pi_i}$, w której y = x lub z = x podstaw:

$$\Gamma_{\pi_i} \leftarrow \Gamma_{\pi_i} \setminus \{y,z\}$$

Komentarz: ponieważ procedura zawiera skończoną liczbę instrukcji, więc algorytm kończy działanie w skończonym czasie. Powyższy algorytm oprócz budowy grafu procedury π_i wykonuje właściwie pierwsze przekształcenie optymalizacyjne: usuwa z procedury instrukcje nie spełniające warunku spójności. W dalszym ciągu będziemy zatem mieli do czynienia ze spójnymi procedurami i, w związku z tym, ze spójnymi grafami procedur.

Ponieważ istnieje wzajemnie jednoznaczna odpowiedniość między blokami procedury a węzłami grafu tej procedury więc często, dla wygody, będziemy utożsamiali te pojęcia. Zgodnie z tym, zamiast mówić np: s jest instrukcją bloku odpowiadającego węzłowi x grafu, będziemy mówili, że s jest instrukcją węzła x.

ad.5 Dla każdego grafu G_{π_i} ($i \in \langle 1,n \rangle$) zostaje wyliczona relacja bezpośredniej dominacji (patrz def. 7 z rozdz.I.3), według następującego algorytmu zapożyczonego od Aho i Ullmana [3]:

Algorytm 4.

Wejście: graf $G_{\pi_i} = (X_{\pi_i}, \Gamma_{\pi_i}, \xi_{\pi_i})$

Wyjście: relacja $DOM(\pi_i) \subset (X_{\pi_i} \setminus \{\xi_{\pi_i}\}) \times X_{\pi_i}$, gdzie dowolna para $(x,y) \in DOM(\pi_i)$ wtedy i tylko wtedy gdy węzeł x jest bezpośrednim dominatorem węzła y.

Metoda:

1. Podstaw $DOM(\pi_i) \leftarrow \emptyset$.
2. Dla każdego węzła x różnego od ξ_{π_i} dołącz do $DOM(\pi_i)$ parę (ξ_{π_i}, x).
3. Dla każdego węzła x grafu G_{π_i} wykonaj:

3.1. Dla każdego węzła y takiego, że nie istnieje droga w grafie G_{π_i} o początku ξ_{π_i} i końcu y nie przechodząca przez x wykonaj: jeśli istnieje z takie, że $(z,x) \in DOM(\pi_i)$ i $(z,y) \in DOM(\pi_i)$ to parę (z,y) w $DOM(\pi_i)$ zastąp przez (x,y).

Komentarz: kroki 1 i 2 inicjalizują relację $DOM(\pi_i)$. Przy każdej iteracji kroku 3 węzły otrzymują coraz "bliższe" dominatory, a po zakończeniu działania algorytmu $DOM(\pi_i)$ jest relacją bezpośredniej dominacji dla grafu G_{π_i}.

ad.6 W grafach $G_{\mathcal{P}_i}$ procedur programu $\mathcal{P}$ zostają wyizolowane maksymalne regiony (patrz uwaga po twierdzeniu 1 z rozdziału I.4.) Dla każdego grafu $G_{\mathcal{P}_i}$ powstaje lista maksymalnych regionów grafu, $LR(\mathcal{P})$. Kolejność występowania regionów na liście $LR(\mathcal{P}_i)$ jest taka, że dla dowolnych regionów R, R' $\in LR(\mathcal{P}_i)$ jeśli R występuje przed R' na liście, to R jest zawarty w R' lub R jest rozłączny z R'.

Algorytm 5

Wejście: graf $G_{\mathcal{P}_i} = (X_{\mathcal{P}_i}, \Gamma_{\mathcal{P}_i}, \xi_{\mathcal{P}_i})$ i relacja $DOM(\mathcal{P}_i)$
Wyjście: $LR(\mathcal{P}_i)$ - lista maksymalnych regionów grafu $G_{\mathcal{P}_i}$

Metoda:

1. Utwórz listę LW węzłów grafu $G_{\mathcal{P}_i}$ będących wejściami regionów w $G_{\mathcal{P}_i}$ w następujący sposób:

1.1. Podstaw $LW \leftarrow \emptyset$

1.2. Dla każdego węzła $x \in X_{\mathcal{P}_i}$ wykonaj:
jeśli istnieje węzeł $y \in X_{\mathcal{P}_i}$ taki, że $(y,x) \in \Gamma_{\mathcal{P}_i}$ i jest spełniony jeden z warunków:

a) $x = y$,
b) x dominuje nad y,
to wstaw węzeł x na listę LW.

1.3. Uporządkuj elementy listy LW tak, aby był spełniony warunek: jeśli węzeł x występuje na liście LW przed węzłem y, to y nie dominuje nad x.

2. Podstaw $LR(\mathcal{P}_i) \leftarrow \emptyset$

3. Dla kolejnych elementów x listy LW wykonaj:

3.1. Podstaw $R \leftarrow \{x\}$

3.2. Dla każdego węzła $y \in X_{\mathcal{P}_i}$ wykonaj:
jeśli istnieje droga $y_1,\ldots,y_n$ w grafie $G_{\mathcal{P}_i}$ taka, że $y_1 = y$, $y_n = x$ i x dominuje nad y_i dla $i \in \langle 1, n-1 \rangle$, to podstaw:

$$R \leftarrow R \cup \{y\}$$

3.3. Wstaw R na listę regionów $LR(\mathcal{P}_i)$

Komentarz: algorytm kończy działanie w skończonym czasie, ponieważ graf $G_{\mathcal{P}_i}$ zawiera skończoną liczbę węzłów. Zgodnie z twierdzeniem 1 z rozdziału I w pierwszym kroku algorytmu powstaje lista węzłów będących wejściami regionów w grafie $G_{\mathcal{P}_i}$. Ponadto druga część dowodu prawdziwości tego twierdzenia uzasadnia, że zbiory, które powstają w dwóch ostatnich krokach są maksymalnymi regionami w grafie $G_{\mathcal{P}_i}$. Porządek jaki został nadany elementom na liście LW w kroku 1.3 gwarantuje, że regiony na liście $LR(\mathcal{P}_i)$ zostają umieszczone we właściwej kolejności.

Oprócz izolacji regionów ta część analizatora wykonuje dwa dodatkowe zadania. Po pierwsze, są modyfikowane grafy procedur programu i w konsekwencji sam program do postaci, w której dla każdego węzła x będącego wejściem regionu, bezpośredni dominator y tego węzła jest jego bezpośrednim poprzednikiem i y ma dokładnie jeden bezpośredni następnik w postaci węzła x. Ta modyfikacja, mająca na celu ułatwienie i zwiększenie efektywności realizacji operacji wyniesienia instrukcji poza region, polega na dołączeniu dodatkowych węzłów do grafu i odpowiedniej zmianie procedur programu.

Po drugie, dla każdego regionu zostają wyróżnione węzły leżące na każdej drodze łączącej wejście regionu z dowolnym jego wyjściem.

Poniżej podajemy algorytmy realizujące te dwa zadania:

Algorytm 6

Wejście: procedura π_i, graf $G_{\pi_i} = (X_{\pi_i}, \Gamma_{\pi_i}, \xi_{\pi_i})$, relacja $DOM(\pi_i)$, maksymalny region R z wejściem x.

Wyjście: zmodyfikowana procedura π_i, zmodyfikowany graf G_{π_i}, zmodyfikowana relacja $DOM(\pi_i)$

Metoda:

1. Jeśli bezpośredni dominator węzła x jest bezpośrednim poprzednikiem węzła x i ma dokładnie jeden bezpośredni następnik, to koniec.

2. Dokonaj modyfikacji grafu G_{π_i} w następujący sposób:

2.1. Utwórz nowy węzeł z i podstaw:

$$X_{\pi_i} \leftarrow X_{\pi_i} \cup \{z\}$$

2.2. Podstaw $\Gamma_{\pi_i} \leftarrow \Gamma_{\pi_i} \cup \{(z,x)\}$ oraz dla każdego węzła y takiego, że $y \notin R(x)$ i $(y,x) \in \Gamma_{\pi_i}$ podstaw:

$$\Gamma_{\pi_i} \leftarrow \Gamma_{\pi_i} \setminus \{(y,x)\},$$
$$\Gamma_{\pi_i} \leftarrow \Gamma_{\pi_i} \cup \{(y,z)\}$$

3. Dokonaj modyfikacji relacji $DOM(\pi_i)$ w następujący sposób:

3.1. Dla węzła y takiego, że $(y,x) \in DOM(\pi_i)$ podstaw:

$$DOM(\pi_i) \leftarrow DOM(\pi_i) \setminus \{(y,x)\},$$
$$DOM(\pi_i) \leftarrow DOM(\pi_i) \cup \{(y,z)\}.$$

3.2. Podstaw $DOM(\pi_i) \leftarrow DOM(\pi_i) \cup \{(z,x)\}$.

4. Dokonaj modyfikacji ciała procedury π_i w następujący sposób:

4.1. Niech e będzie etykietą nie występującą w programie π. Wstaw bezpośrednio przed instrukcję wejściową węzła x instrukcję pustą oznaczoną etykietą e.

4.2. Niech s będzie instrukcją wejściową węzła x. Dla każdej instrukcji s' nie będącej instrukcją wyjściową węzła z regionu R i takiej, że $l_e(s) \in p_e(s')$ zamień każde wystąpienie etykiety $l_e(s)$ w $p_e(s')$ na etykietę e.

Komentarz: algorytm kończy działanie w skończonym czasie, ponieważ procedura π składa się ze skończonej liczby instrukcji, a graf G_{π_i} skończonej liczby węzłów.

Algorytm 7

Wejście: graf G_{π_i}, relacja $DOM(\pi_i)$, region R w G_{π_i} z wejściem x.
Wyjście: zbiór $ZW_{\pi_i}(R) = \{y \in R \mid y$ leży na każdej drodze łączącej wejście regionu R z dowolnym jego wyjściem$\}$.

Metoda:

1. Podstaw $ZW_{\pi_i}(R) \leftarrow \emptyset$

2. Dla każdego węzła y ∈ R takiego, że y dominuje nad każdym węzłem wyjściowym regionu R podstaw:

$$ZW_{\mathfrak{R}_i}(R) \leftarrow ZW_{\mathfrak{R}_i}(R) \cup \{y\}$$

Komentarz: algorytm kończy działanie w skończonym czasie, ponieważ region R składa się ze skończonej liczby węzłów. Algorytm wylicza rzeczywiście zbiory o żądanej własności, co wynika z faktu, że region ma tylko jedno wejście i z pojęcia dominacji węzłów. Elementy zbioru $ZW_{\mathfrak{R}_i}(R)$ będziemy nazywali węzłami zawsze wykonywanymi w regionie R.

Przejdziemy teraz do opisu części optymalizatora nazwanej modułem optymalizującym, której zadaniem jest wykonanie przekształceń optymalizujących na procedurach programu. Najpierw powiemy parę słów o ogólnej organizacji modułu. Rozpatrywane są kolejno procedury programu przechodząc do następnej dopiero po całkowitym zakończeniu optymalizacji danej procedury.

Przekształcenia optymalizacyjne są zestawione w dwie grupy. Pierwsza grupa obejmuje redukcję identycznych wyrażeń i operację wynoszenia instrukcji poza region. Wykonanie tych przekształceń przebiega w następujący sposób: na wstępie ma miejsce optymalizacja regionów procedury w kolejności ich występowania na liście $LR(\mathfrak{R}_i)$, tzn. rozpoczynając od regionów najbardziej zagnieżdżonych. Dla każdego regionu wykonuje się redukcja identycznych wyrażeń i odbywa się wynoszenie instrukcji poza region. Dzięki zachowaniu właściwej kolejności rozpatrywania regionów instrukcje wyniesione poza region stają się przedmiotem optymalizacji w następnym, szerszym regionie. Po zakończeniu optymalizacji regionów są redukowane identyczne wyrażenia w blokach nie należących do żadnego regionu.

Do drugiej grupy należą operacje usuwania niepotrzebnych instrukcji przypisania i zamiany odwołań.

Optymalizację rozpoczynają przekształcenia pierwszej grupy, po czym następują przekształcenia grupy drugiej. Ponieważ zastosowanie przekształceń każdej z grup może stworzyć nowe możliwości dla przekształceń pozostałej grupy, więc są one stosowane na zmianę dopóki jest to celowe.

Po tych uwagach ogólnych przechodzimy do omówienia algorytmów realizujących poszczególne zadania.

Następujący algorytm realizuje redukcję identycznych wyrażeń w regionie procedury:

Algorytm 8

Wejście: procedura $\mathfrak{R}_i$, graf $G_{\mathfrak{R}_i} = (X_{\mathfrak{R}_i}, \Gamma_{\mathfrak{R}_i}, \xi_{\mathfrak{R}_i})$, relacja $DOM(\mathfrak{R}_i)$, R - element listy regionów $LR(\mathfrak{R}_i)$.

Wyjście: procedura $\mathfrak{R}_i$, w której dokonano redukcji identycznych wyrażeń w obrębie instrukcji z regionu R.

Metoda:

1. Podstaw na x węzeł początkowy regionu R.
2. Jeśli x należy do innego regionu zawartego w R, to przejdź do kroku 6.
3. Podstaw na s ostatnią instrukcję węzła x.
4. Jeśli s jest instrukcją przypisania, to wykonaj:

4.1. Porównaj wyrażenia instrukcji s z wyrażeniami instrukcji poprzedzających s w bloku; jeśli istnieje instrukcja s' o identycznym wyrażeniu, (tzn. $C(s) = o(s')$ i $p_a(s) = p_a(s')$), to przejdź do kroku 4.5.

4.2. Podstaw z = x.

4.3. Podstaw na z węzeł bezpośrednio dominujący z; jeśli $z \notin R$, to przejdź do kroku 5.; jeśli z należy do regionu zawartego w R, to wykonaj jeszcze raz krok 4.3.

4.4. Jeśli w węźle z istnieje instrukcja s' mająca identyczne wyrażenie z wyrażeniem instrukcji s, to przejdź do kroku 4.5. W przeciwnym przypadku wróć do kroku 4.3.

4.5. Znajdź D - zbiór zmiennych definiowanych między instrukcjami s' i s procedury $\mathcal{T}_i$, określony wzorem:

$D = \{a \in Z_{SP}(\mathcal{T})$ istnieje instrukcja s" w $\mathcal{T}_i$ taka, że

a) $a \in DEF(s")$ oraz

b) instrukcja s" należy do pewnego obliczenia w $\mathcal{T}_i$ o końcu s, w którym instrukcja s' występuje dokładnie jeden raz na początku tego obliczenia$\}$.

4.6. Jeśli jakaś zmienna z pola argumentów instrukcji s występuje w zbiorze D, to przejdź do kroku 5.

4.7. Dla każdego $j \in \langle 1,1 \rangle$, gdzie 1 jest długością pola wyników instrukcji s' wykonaj:

4.7.1. Jeśli j-ta zmienna z pola wyników instrukcji s' (powiedzmy: a) należy do zbioru D, to wygeneruj nową zmienną (nazwijmy ją: t i wstaw ją w j-te miejsce pola wyników instrukcji s' oraz wstaw w procedurze $\mathcal{T}_i$ po instrukcji s' instrukcję prostego przypisania s", w której $l_o(s") = \xi$, $p_a(s") = t$, $p_w(s") = a$

4.7.2. Wstaw w procedurze $\mathcal{T}_i$ po instrukcji s instrukcję prostego przypisania s"', w której: $l_o(s"') = \xi$, $p_a(s"') = t$, $p_w(s"')$ jest równe j-tej zmiennej z pola wyników instrukcji s.

4.8. Usuń z procedury $\mathcal{T}_i$ instrukcję s.

5. Jeśli s nie była pierwszą instrukcją węzła x, to podstaw na s poprzednią instrukcję węzła x i wróć do kroku 4.

6. Jeśli istnieje w R węzeł jeszcze nie badany, to podstaw go na x i wróć do kroku 2. W przeciwnym przypadku zakończ algorytm.

Komentarz: algorytm kończy działanie w skończonym czasie, ponieważ w regionie znajduje się skończona liczba instrukcji. Natomiast nie jest oczywiste w jaki sposób można wyznaczyć zbiór D w kroku 4.5, dlatego też zanim przejdziemy do omówienia algorytmu wyjaśnimy tę sprawę.

Gdy instrukcje s' i s należą do tego samego węzła, wówczas zbiór D tworzą wszystkie zmienne definiowane przez instrukcje położone między s' i s.

Załóżmy teraz, że s' i s występują w różnych węzłach, odpowiednio z i x. W tym przypadku zbiór D tworzą zmienne definiowane przez:

a) instrukcje węzła z występujące po s'

b) instrukcje węzła x występujące przed s oraz

c) instrukcje należące do węzłów wyznaczonych przez następujący

Algorytm 8a) (pomocniczy)

1. Podstaw $D_B \leftarrow \emptyset$.

2. Podstaw $D_B \leftarrow D_B \cup \{ y \in G_{\mathcal{T}_i} \mid y$ jest bezpośrednim poprzednikiem jakiegoś węzła z D_B, $y \notin D_B$ i $y \neq z \}$.

3. Powtarzaj krok 2 dopóki powoduje to dołączanie nowych węzłów do zbioru D_B.

Algorytm pomocniczy jest efektywny, ponieważ graf $G_{\mathcal{T}_i}$ ma skończoną liczbę węzłów.

Przejdziemy teraz do omówienia działania algorytmu redukującego identyczne wyrażenia:

badane są instrukcje kolejnych węzłów regionu R.
Wyrażenie instrukcji s (tzn. operacja i pole argumentów) występującej w węźle x jest porównywane z wyrażeniami instrukcji poprzedzających s w węźle x i z wyrażeniami instrukcji występujących w węzłach regionu R dominujących nad x (kroki 4.1-4.4 algorytmu) . W wypadku znalezienia identycznego wyrażenia, na przykład w instrukcji s', jest wyliczany zbiór zmiennych definiowanych między instrukcjami s' i s (zbiór D w kroku 4.5.) . Redukcja jest możliwa jeśli żadna zmienna z pola argumentów instrukcji s nie występuje w zbiorze D. Sposób przeprowadzenia redukcji zależy od zmiennych z pola wyników instrukcji s': dla każdej zmiennej a z pola wyników s' i należącej do zbioru D jest generowana nowa zmienna, która zastępuje a w polu wyników tej instrukcji i za instrukcję s' zostaje wstawiona do procedury $\mathcal{T}_i$ odpowiednia instrukcja prostego przypisania (krok 4.7.1) . Natomiast instrukcja s zostaje usunięta z procedury $\mathcal{T}_i$ i jej miejsce zajmują odpowiednie instrukcje prostego przypisania (kroki 4.7.2 i 4.8.) . Algorytm w trakcie działania wykorzystuje przekształcenia T4 i T7.

2) Wynoszenie instrukcji poza region

Algorytm 9

Wejście: procedura $\mathcal{T}_i$, graf $G_{\mathcal{T}_i} = (X_{\mathcal{T}_i}, \Gamma_{\mathcal{T}_i}, \xi_{\mathcal{T}_i})$, relacja $DOM(\mathcal{T}_i)$, R - element listy regionów $LR(\mathcal{T}_i)$, zbiór $ZW_{\mathcal{T}_i}(R)$.

Wyjście: procedura $\mathcal{T}_i$, w której wyniesiono instrukcje poza region R z węzłów zawsze wykonywanych w R.

Metoda:

1. Podstaw na x węzeł początkowy regionu R,

2. Jeśli $x \notin ZW_{\mathcal{T}_i}(R)$ lub x jest elementem innego regionu zawartego w R, to przejdź do kroku 5.

3. Podstaw na s pierwszą instrukcję węzła x.

4. Jeśli s jest instrukcją przypisania to wykonaj:

4.1. Jeśli w regionie R istnieje instrukcja będąca definicją zmiennej z pola argumentów instrukcji s, to przejdź do kroku 5.

4.2. Znajdź zbiór zmiennych określony następującym wzorem:

$D = \{ a \in Z_{SP}(\mathcal{T}) \mid$ istnieje instrukcja s'' w $\mathcal{T}_i$ takim, że

a) a ∈ ODW (s") oraz

b) instrukcja s" należy do pewnego obliczenia w π_i o końcu s, początku w pierwszej instrukcji węzła wejściowego regionu R, która to instrukcja występuje w tym obliczeniu dokładnie jeden raz).

4.3. Dla każdego $j \in \langle 1,l \rangle$, gdzie l jest długością pola wyników instrukcji s wykonaj: niech a oznacza j-tą zmienną z pola wyników instrukcji s. Jeśli s nie jest jedyną definicją zmiennej a w regionie R lub a ∈ D, to wygeneruj nową zmienną (powiedzmy: t) i wstaw ją zamiast a w j-te miejsce pola wyników instrukcji s. Następnie wstaw w procedurze π_i bezpośrednio po instrukcji s instrukcję prostego przypisania s' taką, że $l_e(s') = \varepsilon$, $p_a(s') = t$, $p_w(s') = a$.

4.4. Przesuń instrukcję s na koniec węzła bezpośrednio dominującego węzeł wejściowy regionu R.

5. Jeśli instrukcja s nie była ostatnią instrukcją węzła x, to podstaw na s następną instrukcję i przejdź do kroku 4.

6. Jeśli istnieje w R węzeł jeszcze nie badany, to podstaw go na x i przejdź do kroku 2. W przeciwnym wypadku zakończ algorytm.

Komentarz: algorytm kończy działanie w skończonym czasie, ponieważ na region składa się skończona liczba instrukcji. Konstrukcja zbioru D w kroku 4.2. jest analogiczna do tej, którą omówiliśmy w związku z poprzednim algorytmem.

Działanie algorytmu wynoszenia instrukcji poza region jest następujące: badane są kolejno instrukcje przypisania z węzłów zawsze wykonywanych w regionie R. Instrukcja s, o ile żadna zmienna z jej pola argumentów nie jest definiowana przez instrukcje regionu R, zostaje wyniesiona poza R w następujący sposób: dla każdej zmiennej z pola wyników instrukcji s sprawdza się, czy istnieje w R, inna od s definicja tej zmiennej i czy w R ma miejsce odwołanie do tej zmiennej przed instrukcją s (kroki 4.2. i 4.3) . Jeśli przynajmniej jedno z tych pytań ma odpowiedź pozytywną, to zostaje wygenerowana nowa zmienna, która zajmuje miejsce rozpatrywanej zmiennej w polu wyników instrukcji s i ponadto zostaje wstawiona odpowiednia instrukcja prostego przypisania do procedury π_i bezpośrednio po instrukcji s (krok 4.3.). Z kolei instrukcja s zostaje przesunięta w procedurze π_i na koniec bezpośredniego dominatora wejścia regionu R. W algorytmie korzysta się z przekształceń T7 i T5.

3) Redukcja identycznych wyrażeń w blokach położonych poza regionami przebiega w podobny sposób jak to zostało opisane w algorytmie 8 z tą różnicą, że wyrażeń identycznych szuka się jedynie wśród instrukcji bloków nie należących do żadnego regionu procedury.

4) Usuwanie niepotrzebnych instrukcji przypisania

Algorytm 10

Wejście: procedura π_i, graf G_{π_i}, relacja DOM(π_i)
Wyjście: procedura π_i, z której usunięto niepotrzebne instrukcje przypisania.

Metoda:

1. Podstaw na x węzeł początkowy grafu
2. Podstaw na s ostatnią instrukcję węzła x.

3. Jeśli s jest instrukcją przypisania, to dla każdej zmiennej $a \in p_w(s)$ wykonaj:

3.1. Jeśli $a \in LOK(\mathcal{T}_i)$ i a nie jest p.f.n. oraz istnieje w procedurze $\mathcal{T}_i$ różnowartościowe obliczenie: $s_{i_1},\ldots,s_{i_l}$ takie, że

a) $s_{i_1} = s$,

b) s_{i_1} jest odwołaniem do zmiennej a,

c) s_{i_m} nie jest definicją zmiennej a dla $m \in \langle 2, l-1 \rangle$, to przejdź do kroku 5.

3.2. Jeśli $a \notin LOK(\mathcal{T}_i)$ lub a jest p.f.n. oraz istnieje obliczenie $s_{i_1},\ldots, s_{i_l}$ w $\mathcal{T}_i$ takie, że

a) $s_{i_1} = s$,

b) s_{i_l} jest instrukcją końcową procedury $\mathcal{T}_i$,

c) $a \in p_w(s)$ dla $m \in \langle 2, l \rangle$,

to przejdź do kroku 5.

4. Usuń instrukcję s z procedury $\mathcal{T}_i$.

5. Jeśli instrukcja s nie była pierwszą instrukcją węzła x, to podstaw na s poprzednią instrukcję i przejdź do kroku 3.

6. Jeśli są w grafie $G_{\mathcal{T}_i}$ jeszcze nie badane węzły, to podstaw na x kolejny węzeł i wróć do kroku 2.

7. Jeśli w czasie wykonywania kroków 1-5 została usunięta jakaś instrukcja z procedury $\mathcal{T}_i$, to wróć do kroku 1. W przeciwnym przypadku zakończ algorytm.

Komentarz: algorytm kończy działanie w skończonym czasie, ponieważ graf $G_{\mathcal{T}_i}$ składa się ze skończonej liczby węzłów, a każdy węzeł ze skończonej liczby instrukcji. Problemy istnienia obliczeń, o których mowa w krokach 3.1. i 3.2. dają się łatwo rozwiązać trawersując węzły grafu $G_{\mathcal{T}_i}$ według następników.

Usuwanie niepotrzebnych instrukcji przebiega w następujący sposób: rozpatruje się po kolei instrukcje przypisania występujące w procedurze $\mathcal{T}_i$. Dla każdej zmiennej a z pola wyników instrukcji s sprawdza się, czy istnieje w procedurze odwołanie do zmiennej a związane z instrukcją s relacją definicji-odwołania $\Delta_{\mathcal{T}_i}(a)$ (krok 3.1.). Jeśli dla wszystkich zmiennych z pola wyników instrukcji s nie istnieją takie odwołania, to instrukcja zostaje usunięta z procedury $\mathcal{T}_i$ (przekształcenie T2). Algorytm pracuje w pętli, ponieważ usunięcie jednej instrukcji z procedury może uczynić niepotrzebnymi inne instrukcje. Przyjęta kolejność badania instrukcji w blokach z dołu do góry ma na celu osiągnięcie minimum powtórzeń koniecznych do usunięcia wszystkich niepotrzebnych instrukcji z procedury.

5) Zamiana odwołań zmierzająca do eliminacji instrukcji prostego przypisania.

Algorytm 11

Wejście: procedura $\mathcal{T}_i$, graf $G_{\mathcal{T}_i}$, relacja $DOM(\mathcal{T}_i)$
Wyjście: procedura $\mathcal{T}_i$, w której dokonano zamiany odwołań.

Metoda:

1. Podstaw na x węzeł początkowy grafu

2. Podstaw na s pierwszą instrukcję węzła x.

3. Jeśli s jest instrukcją prostego przypisania, to wykonaj:

3.1. dla każdej instrukcji s' będącej odwołaniem do zmiennej $p_w(s)$ i takiej, że $(s,s') \in \Delta_{\pi_i}(p_w(s))$ wykonaj:

3.1.1. Jeśli w polu argumentów instrukcji s' występuje synonim zmiennej $p_w(s)$ różny od $p_w(s)$, to przejdź do kroku 4.

3.1.2. Jeśli s nie dominuje nad s', to przejdź do kroku 4.

3.1.3. Jeśli s' jest instrukcją wołania procedury, to przejdź do kroku 4.

3.1.4. Znajdź D - zbiór zmiennych definiowanych między instrukcjami s i s', analogiczny do zbioru D wyliczanego w algorytmie 8. Jeśli $p_a(s)$ lub $p_w(s)$ należy do zbioru D, to przejdź do kroku 4.

3.1.5. Zamień każde wystąpienie zmiennej $p_w(s)$ w polu argumentów instrukcji s' na zmienną $p_a(s)$.

4. Jeśli s nie jest ostatnią instrukcją węzła x to podstaw na s następną instrukcję i wróć do kroku 3.

5. Jeśli istnieje w grafie G_{π_i} węzeł jeszcze nie rozpatrywany, to podstaw na x kolejny węzeł i przejdź do kroku 2. W przeciwnym wypadku zakończ algorytm.

Komentarz: algorytm kończy działanie w skończonym czasie, ponieważ procedura składa się ze skończonej liczby instrukcji. Zamiana odwołań przebiega w następujący sposób: dla każdej instrukcji prostego przypisania w procedurze π_i dokonuje się próby zamiany odwołań do zmiennej definiowanej przez tę instrukcję. Warunki w jakich możliwa jest taka zamiana są sprawdzane w krokach 3.2.1. - 3.1.4., a podstawą jej dokonania jest przekształcenie T6.

W przypadku skutecznej zamiany odwołań instrukcja prostego przypisania może być usunięta z programu jako niepotrzebna. Faktycznie algorytm 11 działa w połączeniu z poprzednim algorytmem usuwającym niepotrzebne instrukcje.

3. Dyskusja wyniku i uwagi o implementacji systemu optymalizacyjnego

Najczęściej stosowanymi językami pośrednimi w translatorach przewidujących fazę optymalizacji są języki ozwórek. Języki ozwórek, jakie spotykamy w różnych translatorach różnią się między sobą i dlatego w dotychczasowych rozwiązaniach optymalizator zawsze był nierozłącznie związany z konkretnym translatorem i jego językiem pośrednim.

Nowością naszej propozycji jest pozbawienie optymalizatora tego ograniczenia. Przedstawiony system optymalizacyjny może optymalizować programy dowolnego języka ozwórek, pod warunkiem dostarczenia odpowiedniego opisu języka. Opracowanie od podstaw optymalizatora odpowiadającego złożonością naszemu wymaga napisania programu obejmującego około 3,5 tysiąca zdań języka PL/I. Korzystając z systemu optymalizacyjnego można uzyskać optymalizator kosztem napisania 20 krótkich procedur (ok. 10 zdań każda) i utworzenia tablicy opisu operacji. Zastosowanie systemu pozwala zatem konstruktorowi translatorów na znaczną oszczędność pracy.

Niezależnie od tego, system optymalizacyjny jest użyteczny przy automatycznej konstrukcji translatorów, w tzw. metatranslatorach. Mamy tam bowiem do czy-

nienia z wieloma językami źródłowymi i użytkownik metatranslatora ma możliwość definiowania własnych języków pośrednich. Włączenie systemu optymalizacyjnego do metatranslatora pozwala otrzymywać za jego pomocą translatory produkujące optymalny kod - bez konieczności pisania oddzielnych optymalizatorów.

Omówimy teraz pokrótce funkcje jakie spełnia optymalizator uzyskany za pomocą naszego systemu w porównaniu z optymalizatorem zastosowanym w translatorze Fortranu H opisanym przez Lowery i Medlocka w [11]. Wybraliśmy ten optymalizator za podstawę dyskusji, ponieważ nie spotkaliśmy w literaturze komunikatu o zbudowaniu optymalizatora stosującego przekształcenia niezależne od maszyny w szerszym zakresie niż ma to miejsce właśnie u Lowery i Medlocka.

W obydwu przypadkach ma miejsce globalna analiza programu umożliwiająca skuteczną optymalizację. Analiza obejmuje budowę grafu programu, wyliczenie relacji dominacji dla jego węzłów i wyodrębnienie pętli z jednym wejściem. Nasz optymalizator przeprowadza dodatkowo analizę procedur programu (uwzględniając możliwość rekursji) , w wyniku której do optymalizacji zostają dopuszczone parametry formalne i aktualne oraz zmienne zewnętrzne dla procedury. Operacje redukcji identycznych wyrażeń, wynoszenia instrukcji poza pętlę, zamiany odwołań i wyrzucania zbędnych instrukcji z programu obydwa optymalizatory przeprowadzają w podobnych zakresach. Z przekształceń niezależnych od maszyny (a tylko takie nas interesują) optymalizator translatora Fortranu H wykonuje dodatkowo wyliczanie wyrażeń (ang. folding) i redukcję mocy operatora. W obecnej wersji nasz optymalizator nie przeprowadza tych kroków optymalizacyjnych, ponieważ wymagają one dodatkowych informacji semantycznych o instrukcjach języka. Istnieje jednak możliwość wzbogacenia opisu języka czwórek dostarczonego systemowi optymalizacyjnemu, co pozwoli na rozbudowanie optymalizatora tak, aby przeprowadzał również to przekształcenia. Obecnie jesteśmy w trakcie prac w tym kierunku i następna wersja systemu będzie obejmowała wszystkie najważniejsze metody optymalizacji.

Zajmiemy się teraz dyskusją poprawności i użyteczności przekształceń wykonywanych przez optymalizatory produkowane przez nasz system.

Do procesu optymalizacji włączyliśmy przekształcenia dobrze znane z praktyki i wielokrotnie sprawdzone. Są one jednocześnie na tyle proste, że ich poprawność wydaje się intuicyjnie oczywista. W naszym przypadku optymalizacja odbywa się jednak w szerszym zakresie niż to miało miejsce w dotychczasowych rozwiązaniach: różnica polega na sposobie traktowania instrukcji wołania procedury w programie oraz na tym, że dopuszczamy do optymalizacji parametry procedur. Musimy się zatem zastanowić, czy to rozszerzenie zakresu przekształceń nie wpływa na bezpieczeństwo ich stosowania.

Realizacja interesujących nas przekształceń optymalizujących wiąże się z koniecznością określenia, które zmienne są definiowane (tzn. otrzymują nowe wartości) i do których ma miejsce odwołanie w instrukcjach programu. Przeanalizujemy najpierw, jak ta sprawa była traktowana w dotychczasowych optymalizatorach nie badających zależności wynikających z obecności procedur.

Pojawiały się tu dwie trudności: jak określić definicję zmiennej i odwołanie do zmiennej dla instrukcji wołania procedury i co zrobić w przypadku zmiennych oznaczających te same obiekty (parametry formalne i aktualne wołane przez nazwę) .

Z pierwszym kłopotem radzono sobie przyjmując, że instrukcja wołania procedury definiuje i odwołuje się do wszystkich zmiennych programu i, aby uniknąć drugiego kłopotu, nie dopuszczano do optymalizacji parametrów procedur. Przyjęcie takich ograniczeń wpływa oczywiście ujemnie na zakres stosowania przekształceń.

Widzimy jednak co należy zrobić, aby te ograniczenia mogły być odrzucone: po pierwsze, trzeba zbadać powiązania pomiędzy parametrami procedur celem ustalenia synonimów, tzn. zmiennych mogących oznaczać te same obiekty w programie i, po drugie, trzeba dokonać analizy wołań procedur programu prowadzącej do określenia zbiorów zmiennych definiowanych i tych, do których następuje odwołanie w instrukcjach wołania procedury. W pracy podaliśmy efektywną metodę rozwiązania tych zadań i wykorzystaliśmy ją praktycznie w systemie optymalizacyjnym.

Dodatkowa analiza programu przed jego optymalizacją pozwala zatem stosować przekształcenia w rozszerzonym zakresie bez zmniejszenia ich bezpieczeństwa.

Jak już zaznaczyliśmy, w systemie optymalizacyjnym korzystamy wyłącznie z przekształceń dobrze znanych z praktyki. Sytuacje w programie, które te przekształcenia "poprawiają" często nie wynikają stąd, że program źródłowy (napisany ręcznie) był nieoptymalny, ale powstają w trakcie automatycznego procesu przekładania programu źródłowego na język pośredni (np. podczas zamiany elementu macierzy wielowymiarowej z programu źródłowego na odpowiedni element wektora z programu pośredniego). Fakt ten jest pewną gwarancją użyteczności przekształceń, z uwagi na to, że właśnie program pośredni stanowi obiekt optymalizacji. Nasze rozwiązanie zyskuje dodatkowy walor użyteczności dzięki rozszerzeniu zakresu stosowania przekształceń.

W ogólności jest bardzo trudno ocenić zysk jaki może przynieść optymalizacja. Trzy czynniki wydają się w każdym razie tu odgrywać rolę: ilość sytuacji w programie źródłowym, które optymalizator "poprawia", sposób przekładania programu źródłowego na program pośredni oraz dane programu. Interesujące byłyby wyniki badań statystycznych przeprowadzonych na dużej liczbie rzeczywistych programów.

Dyskusję zakończymy uwagami o implementacji systemu optymalizacyjnego.

System optymalizacyjny według przedstawionego projektu został zaprogramowany w języku PL/I i uruchomiony na maszynie IBM/370. System był testowany wszechstronnie na wielu przykładach. Konstruktor został sprawdzony na czterech wersjach języków pośrednich. Z kolei algorytmy realizujące poszczególne zadania analizatora i modułu optymalizacyjnego były przetestowane na szeregu programach zarówno o typowej jak i nietypowej budowie z punktu widzenia zabiegów optymalizacyjnych.

Realizacja systemu optymalizacyjnego przedstawia się następująco: konstruktor wczytuje opis systemu programowania z procedurami z urządzenia zewnętrznego, na przykład z taśmy magnetycznej, po czym włącza go w odpowiednie miejsce programu optymalizatora. Program optymalizatora ze względu na wielkość został podzielony na pięć niezależnie kompilowanych fragmentów. Trzy spośród nich są przeznaczone dla zadań analizatora i dwa wykonują optymalizację programu. Struktura informacji zbieranych przez analizator jest dosyć skomplikowana. Aby uniknąć dużej liczby przebiegów poszczególne zadania przeplatają się, w trakcie wykonywania jednego zadania są zbierane pomocnicze informacje dla następnego. Dla-

tego też przedstawiony niżej podział na funkcje kolejnych fragmentów nie jest zbyt ścisły. Ponieważ teraz będziemy mówili jednocześnie o programie realizującym optymalizator i o programie systemu programowania z procedurami stanowiącym wejście dla optymalizatora, więc w celu uniknięcia niejednoznaczności, umówimy się ten pierwszy nazywać o. programem, a drugi - s.p. programem.

Zadaniem pierwszego fragmentu o. programu jest podział procedur s.p. programu na bloki, zbudowanie grafów procedur i wyliczenie dla nich relacji bezpośredniej dominacji. Obejmuje on ok. 1000 rozkazów PL/I.

Drugi fragment znajduje zbiory synonimów dla zmiennych s.p. programu i wylicza zbiory zmiennych definiowanych i tych, do których następuje odwołanie w instrukcjach s.p. programu. Ta część składa się z około 500 rozkazów PL/I.

W trzecim fragmencie o. programu ma miejsce wyodrębnienie i utworzenie list maksymalnych regionów dla procedur s.p. programu, znalezienie węzłów zawsze wykonywanych w poszczególnych regionach i modyfikacja procedur i ich grafów. Algorytmy spełniające te zadania obejmują około 600 rozkazów PL/I.

W czwartym fragmencie odbywa się optymalizacja regionów i redukcja identycznych wyrażeń w węzłach nie należących do regionów, około 800 rozkazów PL/I.

W ostatniej części o. programu ma miejsce zamiana odwołań i usuwanie niepotrzebnych instrukcji z procedur s.p. programu, około 500 rozkazów PL/I.

ZAKOŃCZENIE

W niniejszej pracy przedstawiliśmy system automatycznej konstrukcji optymalizatorów dla pewnej klasy języków pośrednich. Interesującą nas klasę języków zdefiniowaliśmy jako uogólnienie języka czwórek, szczególnie dogodnego do optymalizacji.

W części teoretycznej opracowaliśmy metodę analizy procedur z dwoma rodzajami wołania parametrów: przez wartość i przez nazwę, pozwalającą określić, które zmienne otrzymują nowe wartości i do których zmiennych ma miejsce odwołanie w instrukcjach programu z procedurami. Ponadto zdefiniowaliśmy szereg przekształceń optymalizujących dla programów z procedurami.

Część praktyczna obejmuje projekt i realizację w postaci programów w języku PL/I systemu automatycznej konstrukcji optymalizatorów dla opisanego modelu języków z procedurami. Opracowany przez nas system na podstawie opisu konkretnego języka produkuje optymalizator programów tego języka. Idea automatycznej konstrukcji optymalizatorów ma swoje uzasadnienie w praktyce: wiele translatorów ma bowiem podobne języki pośrednie. Zamiast tworzyć optymalizatory oddzielnie dla każdego translatora, co jest zajęciem bardzo pracochłonnym, można za pomocą odpowiedniego systemu generować je automatycznie. Przedstawiona propozycja stanowi pierwszy etap pracy nad zagadnieniem. Dalsze badania powinny się rozwijać w kierunku włączenia do opisu języka pewnych właściwości semantycznych instrukcji, aby umożliwić wzbogacenie produkowanego przez system optymalizatora o takie przekształcenia, jak wyliczanie wyrażeń (ang: folding) i redukcja mocy operatora. Pożądane byłoby także wprowadzenie aparatu semantycznego, na gruncie którego można by udowodnić poprawność przekształceń optymalizacyjnych. Niezależnie od tego, ze względu na sprawę praktycznego wykorzystania systemu, jest wskazane wy-

różnienie i specjalne traktowanie zmiennych indeksowanych oraz wydzielenie klasy zmiennych podlegających optymalizacji (w rzeczywistych językach pośrednich nie wszystkie zmienne spełniają wymagania jakie stawia optymalizacja). Realizując niniejszą wersję systemu optymalizacyjnego świadomie zrezygnowaliśmy z uwzględnienia tych niezbyt istotnych dla samego pomysłu punktów, aby nie tracić z oczu spraw pierwszorzędnej wagi.

Przyjęty przez nas model języków pośrednich jest jednym z możliwych. Wydaje się wysoce pożądane rozwiązanie problemu automatycznej konstrukcji optymalizatorów dla innych modeli języków.

DODATEK A. Przykład wykorzystania systemu optymalizacyjnego

Celem zilustrowania opisanej metody wykorzystamy system optymalizacyjny do otrzymania optymalizatora dla programów systemu programowania z procedurami $\tilde{S}^P$, który zdefiniowaliśmy w przykładzie 5. Następnie opiszemy działanie uzyskanego optymalizatora na przykładzie programu systemu $\tilde{S}^P$.

Nasze zadanie polega na sporządzeniu opisu systemu programowania $\tilde{S}^P$ według schematu podanego w rozdziale V i dostarczeniu go na wejście konstruktora systemu optymalizacyjnego. Na opis ten, który wykonujemy w języku PL/I, składają się: tablica operacji systemu $\tilde{S}^P$ oraz standardowe procedury.
Ponieważ omawiany przykład zrealizowaliśmy na maszynie, więc opis podamy w formie wydruków z maszyny:

```
DCL         $OB_PROG AREA(5000) BASED ($PP);     /* POLE O NAZWIE $OB_PROG SLUZY
                DO PRZECHOWANIA PROCEDURY W PAMIECI OPERACYJNEJ */

DCL         $POCZ OFFSET ($OB_PROG);  /*$POCZ WSKAZUJE POCZATEK PROCEDURY
                ZNAJDUJACEJ SIE W PAMIECI OPERACYJNEJ */

DCL       1 $INST_PROG BASED ($IPR) ,
            2 $OP FIXED BIN,
            2 $ETYK CHAR (256) ,
            2 $IL_PARM FIXED BIN,
            2 $NAZWA CHAR (256) ,
            2 $NAST_INST OFFSET ($OB_PROG) ,
            2 $POP_INST OFFSET ($OB_PROG) ,
            2 $PAR (10) ,
              3 $PARM CHAR (256) ,
              3 $RODZ  FIXED BIN;      /* STRUKTURA O NAZWIE $INST_PROG
OPISUJE WEWNETRZNA REPREZENTACJE INSTRUKCJI, KOLEJNE ELEMENTY TEJ STRUKTURY
OPISUJA:  OPERACJE, ETYKIETE, ILOSC PARAMETROW, NAZWE, ADRESY NASTEPNEJ I
POPRZEDNIEJ INSTRUKCJI W PROGRAMIE. PIERWSZE TRZY ELEMENTY TABLICY $PARM ZA-
WIERAJA POLE ARGUMENTOW, NASTEPNE TRZY POLE WYNIKOW I NASTEPNE POLE ETYKIET;
DLA INSTRUKCJI WOLANIA PROCEDURY, NAGLOWKA I MARKI PROCEDURY $PARM  ZAWIERA
PARAMETRY A $RODZ OKRESLA RODZAJ WOLANIA PARAMETROW; DLA INSTRUKCJI DEKLARA-
CJI $PARM ZAWIERA WYKAZ ZMIENNYCH DEKLAROWANYCH  */
```

```
DCL          ($ET, $ZM) FIXED DEC (2) ;     /* ZMIENNE $ET I $ZM I $ZM SLUZA DO
                  GENEROWANIA NOWYCH ETYKIET I ZMIENNYCH JĘZYKA * /

DCL           $LICZ FIXED BIN;   /* WARTOŚCIA ZMIENNEJ $LICZ JEST LICZBA PROCEDUR
                  PROGRAMU * /

DCL           $PLIK FILE RECORD ENV(F (501 6) REGIONAL (I) DIRECT;
              / * Z PLIKU $PLIK SA PODAWANE PROGRAMY SYSTEMU PROGRAMOWANIA * /

DCL           $TABLI FILE RECORD SEQUENTIAL;
              / * PLIK $TABLI ZAWIERA TABLICE OPISU OPERACJI * /

DCL           $C CHAR (5) , $P   POINTER, ($J,$JI)FIXED BIN;   / * ZMIENNE TE
                  MAJA CHARAKTER POMOCNICZY * /

DCL          1 STR_OP_INST (18) ,
               2 TAK_NIE BIT (II) ,
               2 IL-ARG FIXED BIN,
               2 IL-WYN FIXED BIN,
               2 IL-ETYK FIXED BIN;

WCZYTAJ_PROC: PROC ($I, $P) ;
  DCL $I FIXED BIN;
  DCL $P POINTER;
       IF $I > & $I < $LICZ THEN
    DO;
       ALLOCATE $OB_PROG;
       READ FILE ($PLIK) IN  ($OB_PROG) KEY ($I) ;
       $P=$PP;
       $POCZ=$PP;
    END;                                ELSE
       $P=NULL;
END WCZYTAJ_PROC;

WYPISZ_PROC:PROC ($I) ;
  DCL  $I  FIXED BIN;
        WRITE FILE ($PLIK) FROM ($OB_PROG) KEYFROM ($I) ;
        FREE $OB_PROG;
END WYPISZ_PROC;

DAJ_NAST_INST:PROC ($P,$Q);
  DCL $P POINTER;
  DCL $Q POINTER;
       IF $P-> $NAST_INST=NULLO THEN $Q=NULL;
                                ELSE $Q=$P — > $NAST_INST;
END DAJ_NAST_INST;

DAJ_POP_INST: PROC ($P,$Q) ;
  DCL $P POINTER;
  DCL $Q POINTER;
       IF $P-> $POP_INST=NULLO THEN $Q=NULL;
                               ELSE $Q=$P — > $POP_INST;
```

```
END DAJ_POP_INST;

GEN_INST:PROC ($P, $I) ;
  DCL $P POINTER;
  DCL $I FIXED BINARY;
        ALLOCATE $INST_PROG IN ($OB_PROG) ;
        $OP=$I;
        $NAST_INST=$P-> $NAST_INST;
        $POP_INST=$P;
        $P-> $NAST_INST=$IPR;
        IF $NAST_INST = NULLO THEN
  DO;
        $PI=$NAST_INST;
        $P1-> $POP_INST=$IPR;
  END;
END GEN_INST;

WYRZ_INST:PROC ($P) ;
  DCL $P POINTER;
        $IPR=$P;
        $PI=$POP_INST;
        $PI-> $NAST_INST=$NAST_INST;
        IF $NAST_INST =NULLO THEN
  DO;
        $P=$NAST_INST;
        $PI-> $POP_INST=$POP_INST;
   END;
        FREE $INST_PROG IN  ($OB_PROG) ;
END WYRZ_INST;

PRZESUN_INST:PROC ($P,$Q) ;
  DCL $P POINTER;
  DCL $C POINTER;
        $IPR=$P;
        $PI=$POP_INST;
        $P1-> $NAST_INST=$NAST_INST;
        IF  $NAST_INST  =NULLO THEN
  DO;
        $P=$NAST_INST;
        $P1-> $POP_INST=$POP_INST;
  END;
        $NAST_INST=$Q-> $NAST_INST;
        $POP_INST=$Q;
        $Q-> $NAST_INST=$IPR;
        $Q=$NAST_INST;
        $Q-> $POP_INST=$IPR;

END PRZESUN_INST;

GEN_ET:PROC ($E) ;
```

```
  DCL $E CHAR (256) ;
        $ET=$ET+1;
  ET    $C=$ET;
        $F='0';
        SUBSTR ($E,2,2) = SUBSTR ($C,4,2) ;
END GEN_ET;
GEN ZM:PROC ($B) ;
  DCL $B CHAR (256) ;
        $ZM=$ZM+1;
        $C=$ZM;
        $B='0';
        SUBSTR ($B,2,2) = SUBSTR ($C,4,2) ;
END GEN_ZM;
DAJ_OP:PROC ($P,$I) ;
  DCL  $P  POINTER;
  DCL  $I  FIXED BINARY;
        $I = $P-> $OP;
  END DAJ_OP;
DAJ_ET:PROC ($P,$E) ;
  DCL $P  POINTER;
  DCL $E  CHAR (256);
        $I = $P-> $ETYK;
END DAJ_ET;
DAJ_ARG:PROC ($P,$A) ;
  DCL  $P  POINTER;
  DCL  $A (*)CHAR (256) ;
    DO $J=I TO IL_ARG ($P-> $OP) ;
        $A ($J)=$P > $PARM ($J) ;
        END;
END DAJ_ARG;
DAJ_WYN:PROC ($P,$A);

  DCL $P POINTER;
  DCL $A (*) CHAR (256);
    DO $J=1 TO IL_WYN($P-> $OP);
       $A($J)=$P > $PARM ($J+3) ;
    END;
END DAJ_WYN;
DAJ PET:PROC ($P,$A)  ;
  DCL $P POINTER;
  DCL $A (*)CHAR (256) ;
    DO $J=1 TO IL_ETYK ($P-> $OP) ;
       $A ($J) =$P-> $PARM ($J+6) ;
    END;
END DAJ_PET;
```

```
DAJ_PAR:PROC ($P,$A) ;
  DCL  $P  POINTER;
  DCL  $A (*) CHAR (256) ;
    DO $J=1 TO $P- > $IL_PARM;
        $A ($J)= $P-> $PARM ($J);
    END;
END DAJ_PAR;

DAJ_NAZ:PROC ($P,$B) ;
  DCL  $P  POINTER;
  DCL  $B  CHAR (256) ;
        $B=$P -> $NAZWA;
END DAJ_NAZ;
        ET
WST_ET:PROC ($P,$E) ;
  DCL  $P  POINTER;
  DCL  $E  CHAR (256) ;
        $P-> $ETYK=$E;
END WST_ET;

WST_ARG:PROC ($P,$B,$I) ;
  DCL  $P  POINTER;
  DCL  $B  CHAR (256) ;
  DCL  $I  FIXED BINARY;
        $P-> $PARM ($I) =  $B  ;
END WST_ARG;

WST_WYN:PROC ($P,$B,$I) ;
  DCL  $P  POINTER;
  DCL  $B  CHAR (256) ;
  DCL  $I  FIXED BINARY;
        $P -> $PARM ($I+3) = $B;
END WST_WYN;

WST_PET:PROC ($P,$B,$I) ;
  DCL  $P POINTER;
  DCL  $B  CHAR (256) ;
  DCL  $I  FIXED BINARY;
        $P-> $PARM ($I+6) =  $B;
END WST_PET;

CZY_STAL:PROC ($B) RETURNS (BIT (1));
  DCL  $B  CHAR(256);
        IN VERIFY (SUBSTR ($B,1,1), '0123456789+ '  = 0  THEN
        RETURN ('1' B) ;                                 ELSE
        RETURN('0'B) ;
END CZY_STAL;

IL_PAR:PROC ($P,$I) ;
  DCL  $P  POINTER;
```

```
  DCL  $I  FIXED BINARY;
       $I=$P-> $IL_PARM;
END IL_PAR;

RODZ_WOL:PROC ($P,$N) ;
  DCL  $P  POINTER;
  DCL  $N (*) FIXED BIN;
       CALL IL_PAR ($P,$JE) ;
  DC   $J=1  TO  $J1;
       $N ($J)=$P -> $RODZ ($J);
  END;
EWD RODZ_WOL;
       READ FILE ($TABLI)  INTO (STR_OP_INST) ;
```

Na początku opisu znajdują się deklaracje zmiennych pomocniczych (nazwy rozpoczynające się od symbolu $). Następnie mamy zadeklarowaną tablicę opisu operacji STR_OP_INST, po której następują deklaracje standardowych procedur. Opis końcy instrukcja czytania pliku $TABLI powodująca wypełnienie tablicy STR_OP_INST.

Komentarz rozpoczniemy od przedstawienia tablicy opisu operacji STR_OP_INST:

Nazwa operacji	TAK_NIE	IL_ARG	IL_WYN	IL_ETYK
:=	'001100000000'B	1	1	0
⊖	'001000000000'B	1	1	0
+	'001000000000'B	2	1	0
-	'001000000000'B	2	1	0
/	'001000000000'B	2	1	0
↑	'100000000000'B	0	0	1
=↑	'100000000000'B	2	0	2
<↑	'100000000000'B	2	0	2
≤↑	'100000000000'B	2	0	2
NOP	'010000000000'B	0	0	0
PISZ	'000010000000'B	0	0	0
CZYT	'000010000000'B	0	1	0
PROC	'000000100000'B	0	0	0
KON	'000000010000'B	0	0	0
MPROC	'000000000010'B	0	0	0
DKL	'000001000000'B	0	0	0
WOL	'000000001000'B	0	0	0

Znaczenia poszczególnych pól zostały omówione wcześniej.

Opracowując treść standardowych procedur wchodzących w skład opisu musieliśmy ustalić jaka będzie wewnętrzna postać instrukcji i programów systemu programowania $\tilde{S}P$ oraz w jaki sposób programy te będą dostarczane do pamięci operacyjnej. Wewnętrzną postać instrukcji zadaliśmy zmienną pomocniczą $ INST_PROG i przyjęliśmy, że program systemu $\tilde{S}P$ jest dwukierunkową listą instrukcji i znajduję się

na urządzeniu zewnętrznym w zbiorze o nazwie $PLIK. Każdy zapis w tym zbiorze zawiera jedną procedurę programu systemu i jest oznaczony jednoznacznie kluczem. Instrukcja czytania zbioru $PLIK powoduje zatem wprowadzenie do pamięci operacyjnej określonej procedury programu systemu $\mathcal{SP}$, a instrukcja pisania - jej wyprowadzenie na zbiór $PLIK (patrz procedury standardowe WCZYTAJ_PROC i WYPISZ_PROC). W opisie mamy zadeklarowaną zmienną $OB_PROG, która służy do przechowywania wczytanej procedury programu systemu $\mathcal{SP}$ w pamięci operacyjnej.

Jeszcze jedna sprawa wymaga krótkiego omówienia, mianowicie sposób generowania nowych zmiennych i etykiet. Założyliśmy w tym celu, że zmienne i etykiety systemu programowania $\mathcal{SP}$ (tzn. elementy zbiorów $\bar{Z}$ i $\bar{E}$) nie zaczynają się od symbolu @. Nowe zmienne i etykiety są generowane jako napisy rozpoczynające się od symbolu @, co daje gwarancję uniknięcia kolizji nazw w programach systemu $\mathcal{SP}$ (patrz standardowe procedury GEN_ET i GEN_ZM oraz zmienne pomocnicze $ET i $ZM).

Wyżej przedstawiony opis został włączony przez konstruktor systemu optymalizacyjnego do programu optymalizatora, w wyniku czego otrzymaliśmy optymalizator dla programów systemu programowania $\mathcal{SP}$. Uzyskany optymalizator przetestowaliśmy na maszynie m.in. na następującym programie $\mathcal{T} = \mathcal{T}_1, \mathcal{T}_2$

```
                                PROC    PR1
        I       J       K       DCL
        L       M               DCL
        A,3     B,3     C,3     MPROC   PR2
        K                       CZYT
        L                       CZYT
        J       K       L       WOL     PR2
ET1     I                       :=        K
        L                       +         M       I
ET2     I                       +         M       K
        L                       +         I       L
        J                       +         M       K
                                <↑        L       020     ET2     ET3
ET3     L                       *         M       L
                                <↑        L       J       ET1     ET4
ET4                             PISZ      L
                                PI Z      J
                                KON
```

$\mathcal{T}_1$

```
        A,3     B,3     C,3     PROC    PR2
        X       W       Z       DCL
        A                               B       C
        X                       :=      A
        W                       :=      005
        A                       +       A       W
                                <↑      A       010     ET5     ET6
```

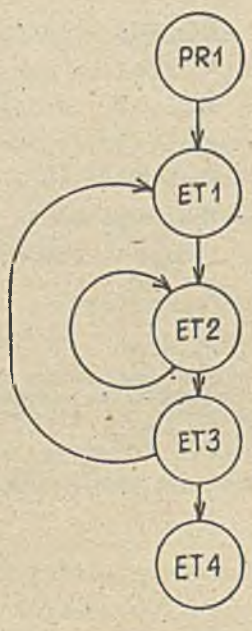
```
ET5     W          *        A    005
                   PISZ     W
                   <↑       A    B      ET5     ET7
ET6     Z          +        A    B
        Z          +        Z    001
        A          *        B    C
ET7                KON
```

$\mathfrak{P}_2$

Obecnie przedstawiamy w formie wydruków z maszyny wyniki działania najważniejszych faz optymalizatora dla programu $\mathfrak{P}$.

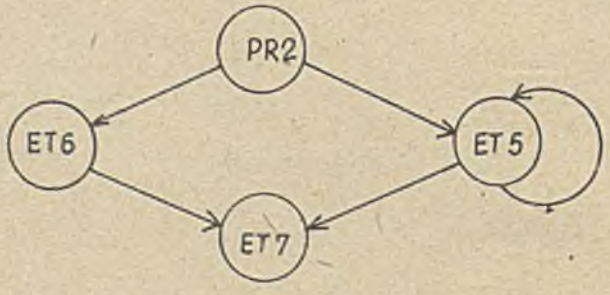
W części analizującej program zostają zbudowane grafy procedur $\mathfrak{P}_1$ i $\mathfrak{P}_2$:

```
WEZEL:                PR1
DOMINATOR:
NASTEPNIKI:           ET1
POPRZEDNIKI:

WEZEL:                ET2
DOMINATOR:            PR2
NASTEPNIKI:           ET2
POPRZEDNIKI:          PR2             ET3

WEZEL:                ET2
DOMINATOR:            ET1
NASTEPNIKI:           ET2             ET3
POPRZEDNIKI:          ET2             ET2

WEZEL:                ET3
DOMINATOR:            ET2
NASTEPNIKI:           ET2             ET4
POPRZEDNIKI:          ET2

WEZEL:                ET4
DOMINATOR:            ET3
NASTEPNIKI:
POPRZEDNIKI:          ET3
```



Graf G_{π_1} procedury π_1

WEZEL:	PR2	
DOMINATOR:		
NASTEPNIKI:	ET5	ET6
POPRZEDNIKI:		
WEZEL:	ET6	
DOMINATOR:	PR2	
NASTEPNIKI:	ET5	ET7
POPRZEDNIKI:	PR2	ET5
WEZEL:	ET6	
DOMINATOR:	PR2	
NASTEPNIKI:	ET7	
POPRZEDNIKI:	PR2	
WEZEL:	ET7	
DOMINATOR:	PR2	
NASTEPNIKI:		
POPRZEDNIKI:	ET3	ET6



Graf G_{π_2} procedury π_2

Blokom instrukcji odpowiadają w grafach węzły o nazwie będącej etykietą instrukcji wejściowej bloku lub nazwę procedury w wypadku bloku zawierającego nagłówek procedury.

Następnie zostają wyodrębnione regiony w grafach procedur programu $\mathfrak{P}$:

REGIONY GRAFU $G_{\mathfrak{P}_1}$: R1= (ET2) R2= (ET1, R1, ET3)

REGIONY GRAFU $G_{\mathfrak{P}_2}$: R3= (ET5)

Graf procedury $\mathfrak{P}_2$ i sama procedura wymagają modyfikacji ponieważ węzeł PR2, będący bezpośrednim dominatorem wejścia regionu R3 ma dwa bezpośrednie następniki: ET5 i ET6.
Wyniki modyfikacji są następujące:

WEZEL:	PR2		
DOMINATOR:			
NASTEPNIKI:	∂E1	ET6	
POPRZEDNIKI:			
WEZEL:	∂E1		
DOMINATOR:	PR2		
NASTEPNIKI:	ET5		
POPRZEDNIKI:	PR2		
WEZEL:	ET6		
DOMINATOR:	PR2		
NASTEPNIKI:	ET7		
POPRZEDNIKI:	PR2		
WEZEL:	ET5		
DOMINATOR:	∂E1		
NASTEPNIKI:	ET5	ET7	
POPRZEDNIKI:	∂E1	ET5	
WEZEL:	ET7		
DOMINATOR:	PR2		
NASTEPNIKI:			
POPRZEDNIKI:	ET5	ET6	

ZMODYFIKOWANY GRAF PROCEDURY $\mathfrak{P}_2$

A,3	B,3	C,3	PROC	PR2			
X	W	Z	DCL				
A			*	B	C		
X			:=	A			
W			:=	005			
A			+	A	W		
			< ↑	A	010	∂E1	ET6
∂E1			NOP				
ET5	W		*	A	005		

```
                                PISZ    W
                                <↑      A       B       ET5     ET7
ET6     Z                       +       A       B
        Z                       +       Z       001
        A                       *       B       C
ET7                             KON
```

ZMODYFIKOWANA PROCEDURA $\mathfrak{P}_2$

Jak widzimy do grafu został dołączony nowy wzeł: αE1, a do procedury instrukcja tworząca blok odpowiadający węzłowi αE1.

Po wstępnej fazie analizy programu następuje właściwa optymalizacja procedur programu $\mathfrak{P}$. Jako pierwsza jest optymalizowana procedura $\mathfrak{P}_1$. Oto wyniki przekształceń redukcji identycznych wyrażeń i wynoszenia instrukcji poza pętlę dokonanych w regionie: R1 = (ET2):

```
                                PROC    PR1
            I       J       K   DCL
            L       M           DCL
            A,3     B,3     C,3 MPROC   PR2
            K                   CZYT
            L                   CZYT
            J       K       L   WOL     PR2
    ET1     I                   :=      K
            L                   +       M       I
→   ET2     I                   +       M       K
            L                   +       I       L
        →   J                   :=      I
                                <↑      L   020     ET2     ET3
    ET3     L                   *       M       L
                                <↑      L       J   ET1     ET4
    ET4                         PISZ    L
                                PISZ    J
                                KON
```

Instrukcja przypisania została zastąpiona instrukcją prostego przypisania.

```
                                PROC    PR2
        I       J       K       DCL
        L       M               DCL
        A,3     B,3     C,3     MPROC   PR2
        K                       CZYT
        L                       CZYT
        J       K       L       WOL     PR2
ET1     I                       :=      K
        L                       +       M       I
```

```
→   I                    +       M       K
→   J                    :=      I
ET2 L                    +       I       L
                         <↑      L       020     ET2     ET3
ET3 L                    *       M       L
                         <↑      L       J       ET1     ET4
ET4                      PISZ    L
                         PISZ    J
                         KON
```

Instrukoja przypisania i prostego przypisania zostały przeniesione do bloku bezpośrednio dominującego wejście regionu R1.

Następnie te same przekształcenia zostają zastosowane do instrukoji regionu R2 = (ET1, ET2, ET3) .

```
                         PROC    PR1
    I       J       K    DCL
    L       M            DCL
    A,3     B,3     C,3  MPROC   PR2
    K                    CZYT
    L                    CZYT
    J       K       L    WOL     PR2
→ ∂ Z1                   +       M       K
ET1 I                    :=      K
    L                    +       M       I
→   I                    :=      ∂ Z1
    J                    :=      I
ET2 L                    +       I       L
                         <↑      L       020     ET2     ET3
ET3 L                    *       M       L
                         <↑      L       J       ET1     ET4
ET4                      PISZ    L
                         PISZ    J
                         KON
```

W tym przypadku, aby wynieść instrukoję poza region należało wygenerować nową zmienną: ∂ Z1.

Redukoja identycznych wyrażeń w blokach położonych poza regionami nie przynosi zmian i następnym przekształceniem jest usuwanie niepotrzebnych instrukoji i zamiana odwołań.

```
                         PROC    PR1
    I       J       K    DCL
    L       M            DCL
    A,3     B,3     C,3  MPROC   PR2
    K                    CZYT
    L                    CZYT
    J       K       L    WOL     PR2
```

```
      ∂ Z1                    +       M       K
ET1   I                       :=      K
      L                       +       M       I
————> I                       :=    ∂ Z1
————> J                       :=    ∂ Z1
→ET2  L                       +     ∂ Z1      L
                              <↑      L      020      ET2     ET3
ET3   L                       *       M       L
                              <↑      L       J       ET1     ET4
ET4                           PISZ    L
                              PISZ    J
                              KON
```

Wszystkie odwołania do zmiennej I związane relacją definicja-odwołanie z prostym przypisaniem I := ∂ Z1 zostały zamienione na odwołanie do ∂ Z1 i instrukcja prostego przypisania zostaje usunięta:

```
                              PROC    PR1
      I       J       K       DCL
      L       M               DCL
      A,3     B,3     C,3     MPROC   PR2
      K                       CZYT
      L                       CZYT
      J       K       L       WOL     PR2
      ∂ Z1                    +       M       K
ET1   I                       :=      K
      L                       +       M       I
————> J                       :=    ∂ Z1
→ET2  L                       +     ∂ Z1      L
                              <↑      L      020      ET2     ET3
ET3   L                       *       M       L
                              <↑      L       J       ET1     ET4
ET4                           PISZ    L
                              PISZ    J
                              KON
```

Podobnie dzieje się z prostym przypisaniem I := K :

```
                              PROC    PR1
      I       J       K       DCL
      L       M               DCL
      A,3     B,3     C,3     MPROC   PR2
      K                       CZYT
      L                       CZYT
      J       K       L       WOL     PR2
      ∂ Z1                    +       M       K
ET1   L                       +       M       K
      J                       :=    ∂ Z1
```

```
ET2     L                   +       @Z1     L
                            <↑      L       020     ET2     ET3
ET3     L                   *       M       L
                            <↑      L       J       ET1     ET4
ET4                         PISZ    L
                            PISZ    J
                            KON
```

W wyniku powtórnego wykonania przekształceń redukcji identycznych wyrażeń i wynoszenia instrukcji poza pętlę otrzymujemy kolejno programy:

```
                            PROC    PR1
        I       J       K   DCL
        L       M           DCL
        A,3     B,3     C,3 MPROC   PR2
        K                   CZYT
        L                   CZYT
        J       K       L   WCL     PR2
    @Z1                     +       M       K
    @Z2                     +       M       K
ET1     L                   :=      @Z2
        J                   :=      @Z1
ET2     L                   +       @Z1     L
                            <|      L       020     ET2     ET3
ET3     L                   *       M       L
                            <|      L       J       ET1     ET4
ET4                         PISZ    L
                            PISZ    J
                            KON


                            PROC    PR1
        I       J       K   DCL
        L       M           DCL
        A,3     B,3     C,3 MPROC   PR2
        K                   CZYT
        L                   CZYT
        J       K       L   WOL     PR2
    @Z1                     +       M       K
ET1     L                   :=      @Z1
        J                   :=      @Z1
ET2     L                   +       @Z1     L
                            <|      L       020     ET2     ET3
ET3     L                   *       M       L
                            <|      L       J       ET1     ET4
ET4                         PISZ    L
                            PISZ    J
                            KON
```

Ponowne zastosowanie przekształceń usuwania niepotrzebnych instrukcji i zamiany odwołań nie przynoszą zmian, więc na tym kończy się optymalizacja procedury π_1.

Przechodzimy do optymalizacji procedury π_2 :

	A,3	B,3	C,3	PROC	PR2			
	X	W	Z	DCL				
	A			*	B	C		
	X			:=	A			
	W			:=	005			
	A			+	A	W		
				<↑	A	010	∂E1	ET6
∂E1				NOP				
ET5	W			*	A	005		
				PISZ	W			
				<↑	A	B	ET5	ET7
ET6	Z			+	A	B		
	Z			+	Z	001		
	A			*	B	C		
ET7				KON				

Redukcja identycznych wyrażeń i wyniesienie instrukcji z pętli w obrębie regionu R3 = (ET5) dają w wyniku:

	A,3	B,3	C,3	PROC	PR2			
	X	h	Z	DCL				
	A			*	B	C		
	X			:=	A			
	W			:=	005			
	A			+	A	W		
				<↑	A	010	∂E1	ET6
∂E1				NOP				
——→	W			*	A	005		
ET5				PISZ	h			
				<↑	A	B	ET5	ET7
ET6	Z			+	A	B		
	Z			+	Z	001		
	A			*	B	C		
ET7				KON				

Redukcja identycznych wyrażeń w blokach położonych poza regionami przynosi następujące zmiany:

	A,3	B,3	C,3	PROC	PR2	
	X	h	Z	DCL		
——→	∂Z3			*	B	C
	A			:=	∂Z3	

```
        X                   :=      A
        W                   :=      005
        A                   +       A       W
                            <↑      A       010     ∂E1     ET6
∂E1                         NOP
        W                   *       A       005
ET5                         PISZ    W
                            <↑      A       B       ET5     ET7
ET6     Z                   +       A       B
        Z                   +       Z       001
------> A                   :=      ∂Z3
ET7                         KON
```

∂Z3 jest wygenerowaną zmienną

Przekształcenia usuwania niepotrzebnych instrukcji i zamiany odwołań powodują usunięcie dwóch instrukcji z bloku ET6:

```
        A,3     B,3     C,3     PROC    PR2
        X       W       Z       DCL
      ∂Z3                       *       B       C
        A                       :=      ∂Z3
        X                       :=      A
        W                       :=      005
        A                       +       A       W
                                <↑      A       010     ∂E1     ET6
∂E1                             NOP
        W                       *       A       005
ET5                             PISZ    W
                                <↑      A       B       ET5     ET7
ET6     A                       :=      ∂Z3
ET7                             KON
```

Zwróćmy uwagę, że instrukcja A := ∂Z3 nie może być usunięta ponieważ definiowaną zmienną jest parametr wołany przez nazwę.

Następnie na skutek zamiany odwołań zostaje usunięte proste przypisanie W := 005, a następnie proste przypisanie X := A :

```
        A,3     B,3     C,3     PROC    PR2
        X       W       Z       DCL
      ∂Z3                       *       B       C
        A                       :=      ∂Z3
------> A                       +       A       005
                                <↑      A       010     ∂E1     ET6
∂E1                             NOP
        W                       *       A       005
```

```
ET5                          PISZ      W
                             <↑        A        B        ET5      ET7
ET6     A                    :=        ∂Z3
ET7                          KON
```

Usunięte zostaje wreszcie proste przypisanie A := ∂ Z3, także po uprzedniej zamianie odwołań:

```
        A,3      B,3      C,3      PROC     PR2
        X        W        Z        DCL
      ∂Z3                         *        B        C
        A                         +        ∂Z3      005
                                  <↑       A        010      ∂E1      ET6
∂E1                               NOP
        W                         *        A        005
ET5                               PISZ     W
                                  <↑       A        B        ET5      ET7
ET6     A                         :=       ∂Z3
ET7                               KON
```

Powtórnie wykonane przekształcenia redukcji identycznych wyrażeń i wynoszenia poza pętlę nie przynoszą żadnych zmian, więc na tym kończy się optymalizacja procedury π_2 i optymalizacja całego programu.

DODATEK B. Oszacowanie kosztu czasowego procesu optymalizacji programów dla optymalizatorów produkowanych przez System Optymalizacyjny

Przypuśćmy, że otrzymaliśmy przy pomocy Systemu Optymalizacyjnego optymalizator dla programów pewnego systemu programowania z procedurami S^p. Niech $\Pi = \pi_1, \ldots, \pi_n$ będzie dowolnym programem systemu S^p. Oszacowanie złożoności czasowej procesu optymalizacji programu Π dokonamy w zależności od następujących wielkości charakteryzujących program:

l_{π_i} - długość (ilość instrukcji) procedury π_i

l_{Π} - długość programu Π ($l_{\Pi} = \sum_{i=1}^{n} l_{\pi_i}$)

m - liczba zmiennych zadeklarowanych w programie Π

n - liczba procedur programu Π

k_i - liczba węzłów w grafie procedury π_i

r_i - liczba regionów w grafie procedury π_i

w_i - liczba węzłów w regionach grafu procedury π_i

Rozpoczniemy od oszacowania kosztu czasowego potrzebnego na realizację zadań analizatora.

1) Zbudowanie zbiorów zmiennych lokalnych i aktywnych dla procedur programu Π oraz zbudowanie grafów procedur tego programu (algorytm 3).

Do wykonania tych czynności potrzebne jest jednokrotne przejrzenie instrukcji programu $\mathcal{T}$. Złożoność czasowa akcji związanych z każdą instrukcją nie zależy od długości analizowanego programu, a więc czas realizacji możemy wyrazić funkcją liniową długości programu:

$$O\ (l_{\mathcal{T}})$$

2) Wyliczenie zbiorów synonimów dla zmiennych programu $\mathcal{T}$ (algorytm 1) .

W algorytmie wyliczającym synonimy dla zmiennych programu przeglądane są wszystkie zmienne programu i akcja dla każdej ze zmiennych wymaga przejrzenia pozostałych zmiennych. Złożoność czasowa tego algorytmu wyraża się zatem funkcją kwadratową liczby zmiennych:

$$O\ (m^2)$$

3) Wyliczenie zbiorów zmiennych definiowanych i zmiennych, do których następuje odwołanie w instrukcjach programu .

Zasadniczą część tego zadania realizuje algorytm 2. W algorytmie tym badane są wszystkie możliwe ciągi wołań procedur takie, że nazwy procedur występujących w danym ciągu nie powtarzają się. Ciągów takich nie może być więcej niż $\sum_{j=1}^{n} \binom{n}{j} j!$, gdzie n jest liczbą procedur programu $\mathcal{T}$. Wielkość tę możemy oszacować w następujący sposób:

$$\sum_{j=1}^{n} \binom{n}{j} j! = \sum_{j=1}^{n} \frac{n!}{j!(n-j)!} \cdot j! = \sum_{j=1}^{n} \frac{n!}{(n-j)!} = n! \sum_{j=1}^{n} \frac{1}{(n-j)!} = n! \sum_{p=0}^{n-1} \frac{1}{p!} < n! \sum_{p=0}^{\infty} \frac{1}{p!} = en!$$

Akcja związana z każdym takim ciągiem nie zależy od żadnej z wyróżnionych wielkości charakteryzujących program, a zatem oszacowanie złożoności czasowej algorytmu 2 wynosi:

$$O\ (en!)$$

W tym miejscu niezbędny jest komentarz do tego bardzo "pesymistycznego" oszacowania. Górna granica oszacowania zostaje osiągnięta w przypadku programu, w którym każda procedura wywołuje wszystkie pozostałe procedury programu. W praktyce trudno sobie wyobrazić program o znacznej liczbie procedur (wtedy n jest duże) , w którym liczba rozpatrywanych ciągów wołań procedur byłaby choć zbliżona do granicy tego oszacowania .

Zadania, które omawiamy poniżej są wykonywane oddzielnie dla każdej procedury programu $\mathcal{T}$. Podamy oszacowania kosztu czasowego realizacji tych zadań dla jednej procedury, powiedzmy $\mathcal{T}_i$.

4) Wyliczenie relacji bezpośredniej dominacji dla grafu procedury $\mathcal{T}_i$ (algorytm 4) .

W algorytmie 4 przeglądane są wszystkie węzły grafu procedury $\mathcal{T}_i$. Akcja dla każdego węzła wymaga przejrzenia pozostałych węzłów, a zatem oszacowanie złożoności czasowej tego algorytmu wynosi

$$O\ (k_i^2)$$

5) Utworzenie listy regionów dla grafu procedury $\mathcal{T}_i$ (algorytm 5 i algorytmy uzupełniające 6 i 7).

W algorytmie 5 najpierw przeglądane są wszystkie węzły grafu procedury π_i w celu ustalenia listy wejść regionów. Akcja dla każdego węzła wiąże się z rozpatrzeniem wszystkich pozostałych węzłów grafu. Następnie przeglądane są jeszcze raz węzły będące wejściami regionów z wykonaniem takiej samej akcji. W sumie oszacowanie złożoności czasowej algorytmu 5 wynosi

$$O(k_i^2) + O(r_i \cdot k_i)$$

Koszt czasowy algorytmów 6 i 7 uzupełniających izolację regionów jest liniowo zależny od liczby regionów w grafie procedury π_i i od liczby węzłów w regionach tego grafu:

$$O(r_i) + O(w_i)$$

W sumie otrzymujemy następujące oszacowanie złożoności czasowej izolacji regionów w grafie procedury π_i :

$$O(k_i^2) + O(r_i \cdot k_i) + O(r_i) + O(w_i)$$

Przechodzimy teraz do oszacowania kosztu czasowego potrzebnego na realizację zadań modułu optymalizacyjnego.

6) Optymalizacja regionów procedury π_i (redukcja identycznych wyrażeń w regionach i wynoszenie instrukcji poza regiony) oraz redukcja identycznych wyrażeń w blokach położonych poza regionami (algorytmy 8 i 9) .

W trakcie łącznego działania algorytmów 8 i 9 dla kolejnych regionów procedury π_i oraz redukcji identycznych wyrażeń w blokach położonych poza regionami są przeglądane kolejne instrukcje procedury π_i . Akcja związana z każdą instrukcją s wymaga przejrzenia wszystkich instrukcji poprzedzających s w tym samym bloku oraz w blokach dominujących blok, do którego s należy. Wyliczymy zależność czasu wykonania tych przekształceń optymalizacyjnych od długości procedury π_i :

$$\sum_{j=1}^{l_{\pi_i}} j = \frac{l_{\pi_i} - 1}{2} \cdot l_{\pi_i} = \frac{l_{\pi_i}^2}{2} - \frac{l_{\pi_i}}{2}$$

Oszacowanie kosztu czasowego realizacji przekształceń wynosi zatem

$$O(l_{\pi_i}^2)$$

7) Usuwanie niepotrzebnych instrukcji z procedury π_i i zamiana odwołań w instrukcjach procedury π_i (algorytmy 10 i 11).

W algorytmach 10 i 11 przeglądane są kolejno wszystkie instrukcje procedury π_i . Akcja związana z każdą instrukcją s z bloku b wymaga przejrzenia wszystkich instrukcji występujących po s w tym samym bloku oraz z instrukcji znajdujących się w blokach leżących na drogach łączących b z blokiem zawierającym ostatnią instrukcję procedury π_i. Oszacowanie kosztu czasowego realizacji tych algorytmów będzie tu identyczne jak poprzednio:

$$O(l_{\pi_i}^2)$$

Sumując oszacowanie poszczególnych zadań otrzymujemy oszacowanie złożoności czasowej procesu optymalizacji programu π :

$$O(l_\pi) + O(m^2) + O(en!) + \sum_{i=1}^{n} (O(k_i^2) + O(k_i^2) + O(r_i \ldots k_i) + \\ + O(r_i) + O(w_i) + O(k_i^2) + O(r_i \ldots k_i) + O(r_i) + O(w_i) + O(l_\pi^2) + O(l_\pi^2))$$

Wobec faktu, że wielkości k_i, r_i, w_i są zawsze mniejsze od l_{π_i} otrzymujemy oszacowanie:

$$O(m^2) + O(en!) + \sum_{i=1}^{n} O(l_{\pi_i}^2)$$

Ponieważ $\sum_{i=1}^{n} l_{\pi_i} = l_\pi$ więc $\sum_{i=1}^{n} l_{\pi_i}^2 \leqslant l_\pi^2$.

Pozwala to nam oszacować ostatecznie koszt czasowy procesu optymalizacji programu π przez:

$$O(m^2) + O(en!) + l_\pi^2 .$$

Literatura

[1] AHO A., SETHI R., ULLMAN J.: A formal approach to code optimization, SIGPLAN Notices, 1970, t.5, s. 86-100

[2] AHO A., ULLMAN J.: Optimization of straight line programs, SIAM j. on Computing, 1972, t.1, nr 1, s. 1-19

[3] AHO A., ULLMAN J.: The theory of parsing, translation and compiling, Englewood, Cliffs, N.J.: Prentice Hall, 1973, t.II

[4] ALLEN F.: Program optimization, W: Annual Review in Automatic Programming, Elmsford, N.Y.: Pergamon Press, 1969, t.5, s.239-307.

[5] ALLEN F.: Control flow analysis, ACM SIGPLAN Notices, 1975, t.5, nr7, s. 1-19

[6] ALLEN F., COCKE J.: A cataloque of optimizing transformations, W Design and optimization of Compilers, Rustin ed., Prentice-Hall, Englewood Cliffs, N.J. 1972, s. 1-30

[7] ALLEN F.: Interprocedural data flow analysis, IFIP 74 North-Holland Publishing Company

[8] BACHMAN P.: A contribution to the problem of the optimization of programs, Proc. IFIP Conf.71 Amsterdam North Holland, 1972, s. 375-380

[9] COCKE J., SCHWARTZ J.: Programming Languages and their compilers, Courant Institute Notes, New York, 1970.

[10] COCKE J.: Global common subexpression elimination, ACM SIGPLAN Notices, 1970, t.5, nr 7, s. 20-24.

[11] GRIES D.: Compiler construction for digital computers, New York, Wiley, 1971.

[12] LETICZEWSKI A.: Equivalence and Optimization of Programs, Lecture Notes in Computer Science 5, Springer Verlag, International Symposium on Theoretical Programming, Eds. A. Ershov and V.A. Nepoumiaschy, 1974, s. 111-128.

[13] LOWERY E., MEDIOCK C.: Object code optimization, CACM, 1969, s. 13-22.

[14] MATWIN S.: Optymalizacja kodu w maszynach von Neumanna, Prace COPAN, 1975.

[15] MAZURKIEWICZ A.: Problemy przetwarzania informacji, Wydawnictwa Naukowo-Techniczne, Warszawa, 1974, t.2.

[16] PAWLAK Z.: Maszyny programowane, Algorytmy, 1969, nr 10, s. 5-19.

[17] PROSSER R.: Applications of Boolean matrices to the analysis of flow diagrams, Proc. Eastern Joint Comput. Conf., New York, 1959, s. 133-138.

[18] RASIOWA N.: Wstęp do matematyki współczesnej, PWN, Warszawa, 1968.

[19] TURSKI W.M.: Propedeutyka informatyki, PWN, Warszawa, 1972.

[20] Zakład Teorii Translatorów IMM: Projekt koncepcyjny metatranslatora, raport wewnętrzny, 1974.

Prace naukowo-badawcze Instytutu Maszyn Matematycznych

REALIZACJA MASZYNOWA ALGORYTMU PLANOWANIA OBLICZEŃ DLA SYSTEMÓW DWUPROCESOROWYCH

Andrzej ROWICKI

W pracy opisano realizacje maszynową algorytmu planowania obliczeń dla systemów dwuprocesorowych oraz podano wyniki działania tego algorytmu. Algorytm ten został zaprogramowany w języku PASCAL na maszynę ODRA 1304

1. OPIS PROGRAMU

Program planowania obliczeń dla systemów dwuprocesorowych jest oparty na algorytmie opisanym w pracy [1], który jest nieznaczną modyfikacją algorytmu podanego w pracy [2].

Program planowania obliczeń został napisany w języku PASCAL [3] zaimplementowanym na maszynę ODRA 1304.

Algorytm planowania obliczeń dla systemów dwuprocesorowych został skonstruowany przy następujących założeniach:

1. Procesory są identyczne.
2. Obliczenia składają się z niepodzielnych zadań o jednakowym czasie wykonania.
3. Obliczenia nie zawierają pętli oraz posiadają tylko jedno wyjście.

Jako model obliczenia przyjęto graf zorientowany nie zawierający obwodów i posiadający tylko jeden wierzchołek końcowy. Wprowadzony model interpretujemy w ten sposób, że każdemu wierzchołkowi grafu jest przyporządkowane tylko jedno zadanie. Przyporządkowanie wierzchołkom zadań jest takie, że orientacja grafu jest zgodna z następstwem wykonywania zadań obliczenia. Graf będący modelem obliczenia będziemy nazywali siecią obliczenia, w skrócie siecią.

W wyniku działania programu uzyskujemy ciąg par wierzchołków sieci oraz łączny czas wykonania obliczenia.

Uzyskany ciąg par wierzchołków sieci ma następującą interpretację: zadania odpowiadające wierzchołkom należącym do tej samej pary można wykonywać równocześnie oraz zadania odpowiadające wierzchołkom kolejnych par można wykonywać bezpośrednio po sobie.

Program zawiera 5 procedur i 3 funkcje, a mianowicie:

```
PROCEDURE GAMA   (I,J,M : INTEGER ; X : Z ; VAR Y : Z) ;
FUNCTION  MOC  (U : Z)  : INTEGER ;
FUNCTION  NI  (I : INTEGER)  : INTEGER ;
FUNCTION  GI  (I : INTEGER)  : INTEGER ;
PROCEDURE  OSOB  ( I : INTEGER ; VAR S1,S2 : INTEGER) ;
PROCEDURE DELTA  ( I : INTEGER ; VAR D1,D2 : Z)  ;
PROCEDURE RF   (I : INTEGER ; VAR R,T : INTEGER)  ;
PROCEDURE CZYTAJ ;
```

Procedury i funkcje zostały wymienione w kolejności występowania w programie. Bezparametrowa procedura CZYTAJ została wprowadzona ze względów formalnych, gdyż bez wydzielenia z programu części zawartej w procedurze CZYTAJ program był zaobszerny dla kompilatora i nie był w całości kompilowany. Pełny wydruk programu został podany w pracy [4] .

Algorytm planowania obliczeń nie miał żadnych ograniczeń na liczbę wierzchołków sieci. Przy realizacji maszynowej algorytmu wystąpiły ograniczenia na liczbę wierzchołków sieci wynikające z następujących przyczyn:

1. W języku PASCAL występują ograniczenia na liczność zmiennej zbiorowej (96 elementów) .

2. Maszyna ODRA 1304 posiada pamięć operacyjną 32 K.

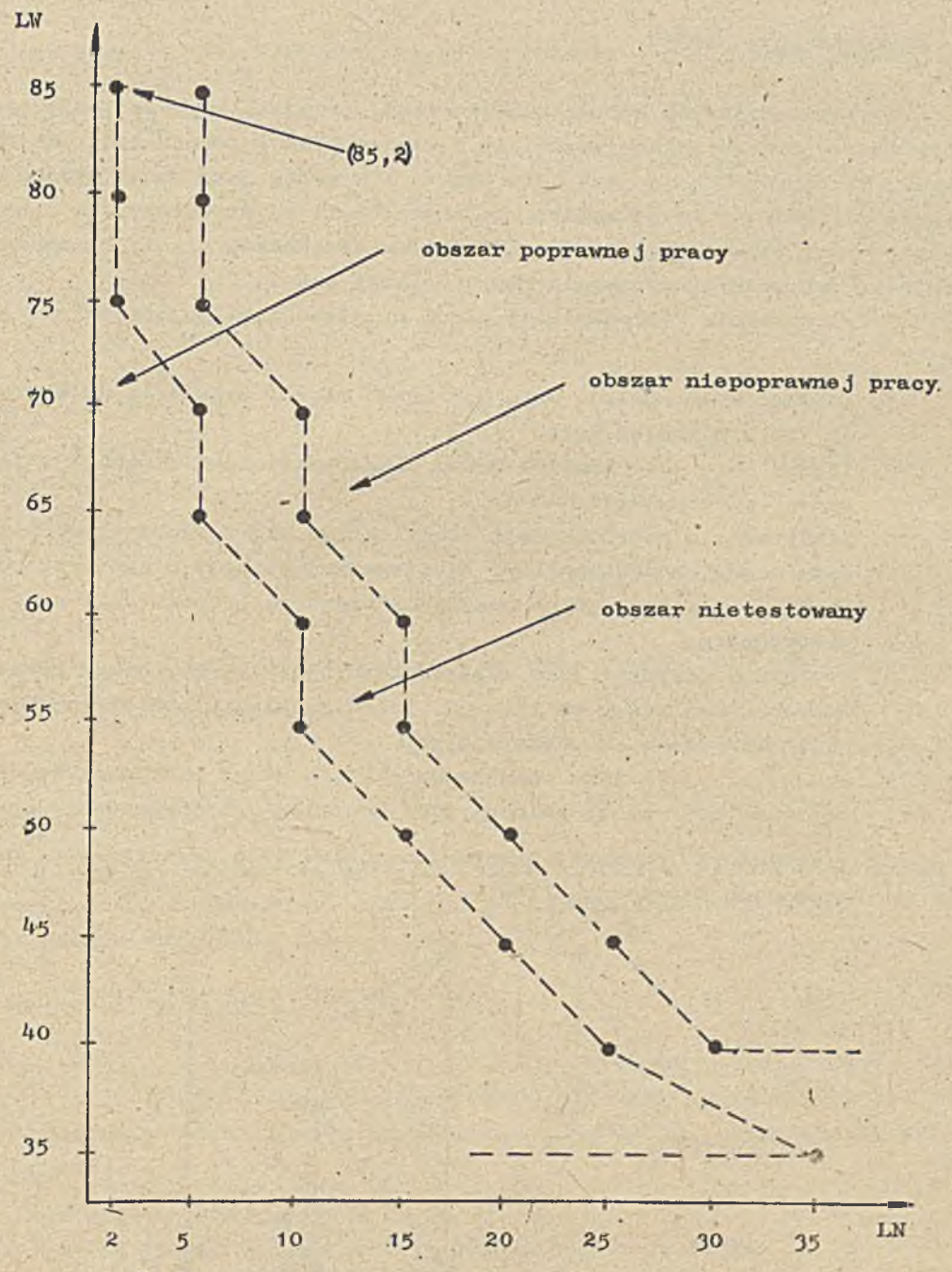
Powyższe ograniczenia narzucają ograniczenia na liczbę wierzchołków i następników sieci. Ograniczenia te wyznaczono na podstawie testowania obszaru poprawnej pracy programu. Wyniki badań przedstawiono na rysunku nr 1.

Obszaru poprawnej pracy dla liczby wierzchołków mniejszej niż 35 nie testowano, bowiem liczba następników wierzchołków sieci nie może być większa od liczby wierzchołków sieci.

Program wykrywa błędy powstałe z następujących przyczyn:

1. niepoprawny opis sieci,
2. niespełnienie ograniczeń na dopuszczalne struktury grafów.

W załączniku podano wynik działania programu dla pewnej wybranej sieci oraz sygnalizację błędów powstałych z wyżej wymienionych przyczyn.



Rys. 1. Wyniki testowania obszaru poprawnej pracy programu.

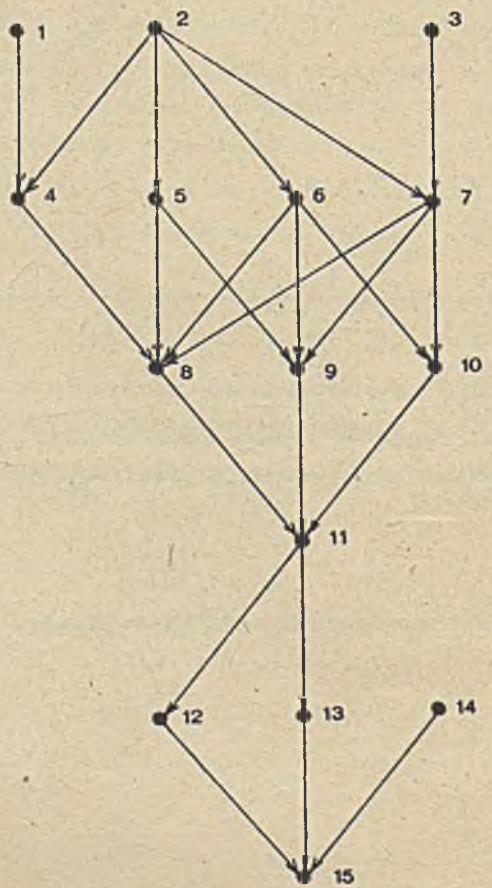
2. REPREZENTACJA DANYCH

Z wielu możliwych reprezentacji sieci, przyjęto reprezentację sieci za pomocą zbioru ciągów składających się z wierzchołka i jego bezpośrednich następników. Ta reprezentacja jest dość zwarta i wygodna jako punkt wyjściowy pracy algorytmu. Ponadto ma tę zaletę, że niezależnie od struktury sieci posiada zawsze tę samą liczbę ciągów równą liczbie wierzchołków sieci, co znacznie ułatwia kontrolę poprawności tworzenia reprezentacji sieci.

Przy tworzeniu reprezentacji sieci przyjęto następujące założenia techniczne:

1. nazwami wierzchołków mogą być tylko liczby całkowite dodatnie mniejsze od 100 i różne od zera.
2. liczba zero jest znakiem końca ciągu, natomiast liczba 100 jest znakiem końca reprezentacji sieci.
3. każdy ciąg z reprezentacji sieci składa się tylko z wierzchołka i jego bezpośrednich następników. Na pierwszej pozycji ciągu występuje wierzchołek, a na następnych pozycjach ciągu są bezpośrednie następniki tego wierzchołka.
4. przecinek przyjęto jako znak rozdzielający wyrazy ciągu /może nim być dowolny znak różny od liczby/. Przed znakiem końca ciągu /liczba 0/ należy umieścić znak rozdzielający.
5. spację przyjęto jako znak rozdzielający ciągi wierzchołków. Znak końca reprezentacji sieci /liczba 100/ powinien być poprzedzony spacją.

Przykładowo rozważymy prostą stosunkowo sieć przedstawioną na rysunku nr 2, zawierającą 15 wierzchołków.



Reprezentacja sieci przedstawionej na rysunku 2 jest następująca:

```
1 , 4 , 0
2 , 4 , 5 , 6 , 7 , 0
3 , 7 , 0
4 , 8 , 0
5 , 8 , 9 , 0
6 , 8 , 9 , 10 , 0
7 , 8 , 9 , 10 , 0
8 , 11 , 0
9 , 11 , 0
10 , 11 , 0
11 , 12 , 13 , 0
12 , 15 , 0
13 , 15 , 0
14 , 15 , 0
15 , 0
100
```

W wyniku działania programu uzyskujemy następujący ciąg par wierzchołków sieci (patrz str.91):

$<2,3>$, $<1,6>$, $<5,7>$, $<4,9>$, $<8,10>$, $<11,14>$, $<12,13>$, $<15,\emptyset>$

gdzie $\emptyset$ oznacza element pusty

Zadania odpowiadające wierzchołkom należącym do tej samej pary można wykonywać równocześnie, natomiast kolejność par wyznacza kolejność wykonywania zadań odpowiadających rozważanej parze.

3. EKSPLOATACJA PROGRAMU

Identyfikatorem programu jest " A P O S ". Program zawiera trzy stałe: RW, RN, PW. Stałe RW i RN narzucają ograniczenia na liczbę wierzchołków sieci i na maksymalną liczbę bezpośrednich następników wierzchołka. Ograniczenia te są następujące:

liczba wierzchołków sieci $\leq$ RW

maksymalna liczba bezpośrednich następników wierzchołka $<$ RN

Natomiast stałe RW i RN powinny spełniać ograniczenia związane z obszarem poprawnej pracy programu, które zostały przedstawione na rysunku 1.

Stała PW oznacza tryb pracy programu, tzn. jeżeli PW = 0, to uzyskujemy wydruk następujących informacji:

- parametry sieci /liczba wierzchołków, maksymalna liczba bezpośrednich następników wierzchołka/
- reprezentacja sieci
- łączny czas wykonania obliczenia
- rozłączny podział zbioru wierzchołków sieci na podzbiory wierzchołków Q (i)

W przypadku gdy stała PW = 1, to uzyskujemy pełny wydruk zawierający

oprócz informacji uzyskanych poprzednio /dla wartości stałej PW = 0/ informacje o wynikach pośrednich działania algorytmu /patrz [1] / takich jak:

- zbiory L (i)
- zbiory F (i)
- wartości funkcji f
- wierzchołki osobliwe s_i i s_i^*

Podzbiory Q (i) zbioru wierzchołków sieci mają następującą interpretację: zadania odpowiadające wierzchołkom ze zbioru Q (i) można wykonywać równocześnie, a zadania odpowiadające wierzchołkom należącym do zbioru Q (i+1) można wykonywać bezpośrednio po zadaniach odpowiadających wierzchołkom ze zbioru Q (i).

Program sygnalizuje błędy związane z błędnym opisem reprezentacji sieci oraz wykrywa błędne struktury, tzn. wykrywa grafy nie będące sieciami /zawierające pętle, niespójne/. Wykrycie błędu jest sygnalizowane odpowiednim komunikatem.

Komunikat "BŁĄD DANYCH" jest drukowany w następujących przypadkach:

1. niespełnienia ograniczeń na liczbę wierzchołków i następników,
2. niespełnienia ograniczeń na nazwy wierzchołków,
3. nadania tej samej nazwy różnym wierzchołkom,
4. podania kolejno po sobie dwóch znaków rozdzielających,
5. opuszczenia znaku rozdzielającego o ile to powoduje przekroczenie zakresu poprawnej pracy,
6. braku końca reprezentacji sieci w przypadku gdy liczba wierzchołków sieci jest równa wartości stałej RW.

Niesygnalizowanie opuszczenia znaku rozdzielającego może spowodować zmianę dopuszczalnej struktury grafu i w tym przypadku będzie odpowiednio sygnalizowane.

W przypadku wystąpienia błędu danych, jedynym wydrukiem jest wczytana reprezentacja sieci oraz jej parametry i komunikat "błąd danych", niezależnie od wartości stałej PW.

Komunikaty:

" SIEĆ NIE JEST SPÓJNA "

" SIEĆ ZAWIERA PĘTLĘ "

nie wymagają komentarza.

Komunikat "sieć nie jest spójna" nie oznacza, że sieć nie zawiera pętli. Jeżeli sieć nie jest spójna i zawiera pętle to również i w tym przypadku będzie drukowany komunikat "sieć nie jest spójna". Natomiast komunikat "sieć zawiera pętle" oznacza, że sieć jest spójna i zawiera pętle.

W tym miejscu wydaje się celowe zwrócenie uwagi na fakt, że pojęcie spójności ma nieco inne znaczenie dla sieci. Ponieważ sieć została zdefiniowana jako graf zorientowany spójny niezawierający obwodów, posiadający tylko jeden wierzchołek końcowy /patrz [1] /, to fakt wystąpienia więcej niż jednego wierzchołka końcowego będzie naruszeniem spójności sieci.

W przypadku wystąpienia błędów związanych z dopuszczalnymi strukturami grafów, wydruki są zależne od wartości stałej PW. Wydruki te są takie same jak poprzednio i są dodatkowo uzupełnione komunikatami określającymi typ błędów powodujących naruszenie dopuszczalnych struktur grafów.

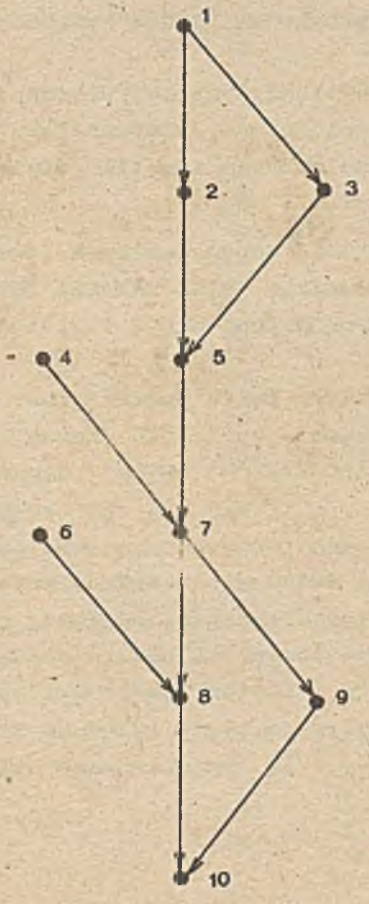
W przypadku równoczesnego wystąpienia błędów związanych z błędnym opisem

sieci i błędów związanych z przekroczeniem dopuszczalnych struktur grafów, sygnalizowane jest wówczas tylko wystąpienie błędów pierwszego typu.

W załączniku podano wynik działania programu dla sieci przedstawionych na rysunku 2 i 3 przy następujących wartościach stałych RW = 25, RN = 15, PW = 0 oraz PW = 1. Podano również wyniki testowania błędów dla sieci przedstawionej na rysunku 3. Pełny wydruk programu został podany w pracy [4] .

4. TESTOWANIE BŁĘDÓW

Dla sieci przedstawionej na rysunku 3 przeprowadzimy testowanie błędów powstałych w wyniku błędnego opisu reprezentacji sieci oraz związanych z przekroczeniem zakresu dopuszczalnych struktur grafu.



Rys.3. Sieć

Testowane będą następujące przypadki błędów związanych z niepoprawnym opisem reprezentacji sieci:

1. niespełnienie ograniczeń na nazwy wierzchołków /nadanie wierzchołkowi nazwy będącej liczbą większą od 100, patrz str.95/,

2. nadanie tej samej nazwy różnym wierzchołkom /nazwa 9 dwukrotnie stosowana, str.96/,
3. opuszczenie znaku rozdzielającego /ciąg 4,7 poprzedzający ciąg 5,7 nie ma znaku rozdzielającego, str.96/.

W przypadku nadania wierzchołkowi nazwy będącej liczbą większą od 100, w wydruku reprezentacji sieci temu wierzchołkowi jest nadawana nazwa będąca liczbą 0.

Nadanie tej samej nazwy kilku wierzchołkom nie powoduje zmian nazw wierzchołków w wydruku reprezentacji sieci.

W przypadku opuszczenia znaku rozdzielającego, błąd ten nie jest sygnalizowany jako błąd w danych, bowiem nie narusza ograniczeń zakresu poprawnej pracy programu. Natomiast będzie sygnalizowany jako błąd struktury sieci, ponieważ narusza spójność sieci. W tym przypadku będzie on sygnalizowany komunikatem "SIEĆ NIE JEST SPÓJNA".

Testowane będą następujące przypadki błędów związanych ze zmianą struktury sieci:

1. zawieranie pętli /pętla występuje między wierzchołkami 8 i 2, str.97/,
2. zawieranie wierzchołka izolowanego /wierzchołek 11 patrz str. 98/,
3. zawieranie wierzchołka izolowanego i pętli /wierzchołek izolowany 11, pętla między wierzchołkami 8 i 2/.

W przypadku wystąpienia zakłóceń dopuszczalnych struktur grafów, podanych w punkcie 1, będzie drukowany komunikat "SIEĆ ZAWIERA PĘTLE". Natomiast w przypadku wystąpienia zakłóceń podanych w punktach 2 i 3, będzie drukowany komunikat "SIEĆ NIE JEST SPÓJNA".

Jeżeli graf równocześnie zawiera pętle /patrz punkt 3/ i nie jest spójny, to jest tylko sygnalizowane naruszenie spójności /komunikat - "SIEĆ NIE JEST SPÓJNA"/. Natomiast komunikat "SIEĆ ZAWIERA PĘTLE" oznacza, że sieć jest spójna i zawiera pętle.

Przetestowano również przypadek równoczesnego występowania obydwóch typów błędów, (str.98) tzn. przypadek zawierania dwóch wierzchołków końcowych /wierzchołki 10 i 11 są wierzchołkami końcowymi/ oraz nadanie tej samej nazwy dwóm wierzchołkom /nazwa 9 dwukrotnie nadana/. W tym przypadku będzie sygnalizowany błąd pierwszego typu komunikatem "BLĄD W DANYCH", bowiem jeżeli występują błędy związane z błędnym opisem reprezentacji sieci oraz błędy związane z przekroczeniem dopuszczalnych struktur grafów, to wówczas są sygnalizowane tylko błędy pierwszego typu.

Literatura

[1] ROWICKI A.: A Note on Optimal Scheduling for Two-processor Systems. Information Processing Letters, 1975, t.4, nr 2, s. 27-30,

[2] ROWICKI A.: Algorytm planowania obliczeń dwuprocesorowych, Prace IMM, 1975, nr 2.

[3] WIRTH N.: The Programming Language Pascal, Acta Informatica, 1971, nr 1, s. 35-63.

[4] ROWICKI A.: Algorytm planowania obliczeń współbieżnych dla dwóch procesorów. Program w języku PASCAL 1900 - opis i tabulogram, IMM, Zakład Teorii Systemów Operacyjnych, Opracowanie nr 17, listopad, 1975, s.1-43.

Przykłady

LICZBA WIERZCHOŁKÓW LW = 15

MAKSYMALNA LICZBA NASTEPNIKOW LN - 1 = 4

ZBIORY NASTEPNIKOW WIERZCHOŁKOW SIECI

NUMER	WIERZCHOŁEK	NASTEPNIKI			
1	1	4			
2	2	4	5	6	7
3	3	7			
4	4	8			
5	5	8	9		
6	6	8	9	10	
7	7	8	9	10	
8	8	11			
9	9	11			
10	10	11			
11	11	12	13		
12	12	15			
13	13	15			
14	14	15			
15	15	WK			

WIERZCHOŁEK KOŃCOWY WK = 15
NUMER WIERZCHOŁKA WK K = 15

CZAS WYKONANIA OBLICZENIA T = 8

TABLICA ZBIORÓW Q (I)

NUMER	ELEMENTY ZBIORÓW Q (I)	
1	2	3
2	1	6
3	5	7
4	4	9
5	8	10
6	11	14
7	12	13
8	15	

LICZBA WIERZCHOŁKÓW LW = 15

MAKSYMALNA LICZBA NASTEPNIKOW LN - 1 = 4

ZBIORY NASTEPNIKOW WIERZCHOŁKOW SIECI

NUMER	WIERZCHOŁEK	NASTEPNIKI			
1	1	4			
2	2	4	5	6	7
3	3	7			
4	4	8			
5	5	8	9		
6	6	8	9	10	
7	7	8	9	10	
8	8	11			
9	9	11			
10	10	11			
11	11	12	13		
12	12	15			
13	13	15			
14	14	15			
15	15	WK			

WIERZCHOŁEK KOŃCOWY WK = 15
NUMER WIERZCHOŁKA WK K = 15

TABLICA POZIOMÓW

NUMER	POZIOM P (I)			
1	1	2	3	
2	4	5	6	7
3	8	9	10	
4	11			
5	12	13	14	
6	15			

LICZBA POZIOMÓW RZ = 6

TABLICA ZBIORÓW F (I)

NUMER	ELEMENTY ZBIORÓW			F (I)
1	1	2	3	6
2	4	5	7	9
3	8	10		
4	11	14		
5	12	13		
6	15			

WIERZCHOŁKI OSOBLIWE S1 S2

NUMER	S1	S2
1	1	6
2	4	9
4	11	14

WARTOŚCI FUNKCJI F (I)

NUMER	WARTOŚĆ FUNKCJI F (I)
1	2
2	2
3	1
4	1
5	1
6	1

CZAS WYKONANIA OBLICZENIA T = 8

TABLICA ZBIORÓW Q (I)

NUMER	ELEMENTY ZBIORÓW Q (I)	
1	2	3
2	1	6
3	5	7
4	4	9
5	8	10
6	11	14
7	12	13
8	15	

LICZBA WIERZCHOŁKOW LW = 10

MAKSYMALNA LICZBA NASTĘPNIKOW LN - 1 = 2

ZBIORY NASTEPNIKOW WIERZCHOŁKOW SIECI

NUMER WIERZCHOŁEK	NASTEPNIKI	
1	2	3
2	5	
3	5	
4	7	
5	7	
6	8	
7	8	9
8	10	
9	10	
10	WK	

WIERZCHOŁEK KOŃCOWY WK = 10

NUMER WIERZCHOŁKA WK K = 10

TABLICA POZIOMÓW

NUMER	POZIOM P (I)	
1	1	
2	2	3
3	4	5
4	6	7
5	8	9
6	10	

LICZBA POZIOMÓW RZ = 6

TABLICA ZBIORÓW F (I)

NUMER	ELEMENTY ZBIORÓW F (I)	
1	1	4
2	2	3
3	5	6
4	7	
5	8	9
6	10	

WIERZCHOŁKI OSOBLIWE S1 S2

NUMER	S1	S2
3	5	6

WARTOŚCI FUNKCJI F (I)

NUMER	WARTOŚCI FUNKCJI F (I)
1	1
2	1
3	1
4	1
5	1
6	1

CZAS WYKONANIA OBLICZENIA T = 6

TABLICA ZBIORÓW Q (I)

NUMER	ELEMENTY ZBIORÓW Q (I)	
1	1	4
2	2	3
3	5	6
4	7	
5	8	9
6	10	

LICZBA WIERZCHOŁKÓW LW = 10

MAKSYMALNA LICZBA NASTEPNIKOW LN - 1 = 2

ZBIORY NASTEPNIKOW WIERZCHOŁKOW SIECI

NUMER	WIERZCHOŁEK	NASTEPNIKI	
1	1	2	3
2	2	5	
3	3	5	
4	4	7	
5	5	7	
6	6	8	
7	7	8	9
8	8	10	
9	9	10	
10	10	WK	

WIERZCHOŁEK KOŃCOWY WK = 10

NUMER WIERZCHOŁKA WK K = 10

CZAS WYKONANIA OBLICZENIA T = 6

TABLICA ZBIORÓW Q (I)

NUMER	ELEMENTY ZBIORÓW	Q (I)
1	1 4	
2	2 3	
3	5 6	
4	7	
5	8 9	
6	10	

LICZBA WIERZCHOŁKÓW LW = 11

MAKSYMALNA LICZBA WIERZCHOŁKÓW LW - 1 = 2

ZBIORY NASTEPNIKÓW WIERZCHOŁKÓW SIECI

NUMER	WIERZCHOŁEK	NASTEPNIKI	
1	1	2	3
2	2	5	
3	3	5	
4	4	7	
5	5	7	
6	6	8	
7	7	8	9
8	8	10	
9	9	10	
10	10	WK	
11	0	WK	

WIERZCHOŁEK KOŃCOWY WK = 0

NUMER WIERZCHOŁKA WK K = 11

BŁĄD W DANYCH

LICZBA WIERZCHOŁKÓW LW = 11

MAKSYMALNA LICZBA NASTEPNIKÓW LN - 1 = 2

ZBIORY NASTEPNIKÓW WIERZCHOŁKÓW SIECI

NUMER	WIERZCHOŁEK	NASTEPNIKI	
1	1	2	3
2	2	5	
3	3	5	
4	4	7	
5	5	7	
6	6	8	
7	7	8	9
8	8	10	
9	9	10	
10	10	WK	
11	9	10	

WIERZCHOŁEK KOŃCOWY WK = 10
NUMER WIERZCHOŁKA WK K = 10

BŁĄD W DANYCH

LICZBA WIERZCHOŁKÓW LW = 9

MAKSYMALNA LICZBA NASTEPNIKÓW LN - 1 = 3

ZBIORY NASTEPNIKÓW WIERZCHOŁKOW SIECI

NUMER	WIERZCHOŁEK	NASTEPNIKI		
1	1	2	3	
2	2	5		
3	3	5		
4	4	7	5	7
5	6	8		
6	7	8	9	
7	8	10		
8	9	10		
9	10	WK		

WIERZCHOŁEK KOŃCOWY WK = 10
NUMER WIERZCHOŁKA K = 9

SIEĆ NIE JEST SPOJNA

CZAS WYKONANIA OBLICZENIA T = 3

TABLICA ZBIORÓW Q (I)

NUMER	ELEMENTY ZBIORÓW	Q(I)
1	6 7	
2	8 9	
3	10 6	

LICZBA WIERZCHOŁKÓW LW = 10

MAKSYMALNA LICZBA NASTEPNIKÓW LW - 1 = 2

ZBIORY NASTEPNIKÓW WIERZCHOŁKÓW SIECI

NUMER	WIERZCHOŁEK	NASTEPNIKI	
1	1	2	3
2	2	5	
3	3	5	
4	4	7	
5	5	7	
6	6	8	
7	7	8	9
8	8	2	10
9	9	10	
10	10	WK	

WIERZCHOŁEK KOŃCOWY WK = 10
NUMER WIERZCHOŁKA WK K = 10

SIEC ZAWIERA PETLE

CZAS WYKONANIA OBLICZENIA T = 2

TABLICA ZBIORÓW Q(I)

NUMER	ELEMENTY ZBIORÓW Q(I)	
1	9	7
2	10	10

LICZBA WIERZCHOŁKÓW LW = 11

MAKSYMALNA LICZBA NASTEPNIKÓW LW - 1 = 2

ZBIORY NASTEPNIKÓW WIERZCHOŁKÓW SIECI

NUMER	WIERZCHOŁEK	NASTEPNIKI	
1	1	2	3
2	2	5	
3	3	5	
4	4	7	
5	5	7	
6	6	8	
7	7	8	9
8	8	10	
9	9	10	
10	10	WK	
11	11	WK	

WIERZCHOŁEK KOŃCOWY WK = 11
NUMER WIERZCHOŁKA WK K = 11

SIEĆ NIE JEST SPÓJNA
CZAS WYKONANIA OBLICZENIA T = 1

TABLICA ZBIORÓW Q (I)

NUMER	ELEMENTY ZBIORÓW		Q(I)
1	11	10	

LICZBA WIERZCHOŁKÓW LW = 11

MAKSYMALNA LICZBA NASTEPNIKÓW LN - 1 = 2

ZBIORY NASTEPNIKÓW WIERZCHOŁKÓW SIECI

NUMER	WIERZCHOŁEK	NASTEPNIKI	
1	1	2	3
2	2	5	
3	3	5	
4	4	7	
5	5	7	
6	6	8	
7	7	8	9
8	8	2	10
9	9	10	
10	10	WK	
11	11	WK	

WIERZCHOŁEK KOŃCOWY WK = 11

NUMER WIERZCHOŁKA WK K = 11

SIEĆ NIE JEST SPÓJNA

CZAS WYKONANIA OBLICZENIA T = 1

TABLICA ZBIORÓW Q(I)

NUMER	ELEMENTY ZBIORÓW		Q(I)
1	11	10	

LICZBA WIERZCHOŁKÓW LW = 12

MAKSYMALNA LICZBA NASTĘPNIKÓW LN - 1 = 2

ZBIORY NASTEPNIKÓW WIERZCHOŁKÓW SIECI

NUMER	WIERZCHOŁEK	NASTEPNIKI	
1	1	2	3
2	2	5	
3	3	5	
4	4	7	
5	5	7	
6	6	8	
7	7	8	9
8	8	10	
9	9	10	
10	10	WK	
11	11	WK	
12	9	10	11

WIERZCHOŁEK KOŃCOWY WK = 11
NUMER WIERZCHOŁKA WK K = 11

BŁĄD W DANYCH

Prace naukowo-badawcze Instytutu Maszyn Matematycznych

PLANOWANIE OBLICZEŃ ZALEŻNYCH PODZIELNYCH DLA SYSTEMÓW DWUPROCESOROWYCH

Andrzej ROWICKI

W pracy podano algorytm planowania obliczeń dla systemów dwuprocesorowych, przy założeniu, że zadania są podzielne w dowolny sposób oraz, że struktura obliczenia jest grafem nie zawierającym pętli i posiadającym tylko jeden wierzchołek końcowy. Pokazano, że algorytm ten jest optymalny, tzn. zapewnia minimalny czas wykonania obliczenia.

1. WSTĘP

Rozważmy problem optymalnego planowania obliczeń dla systemów dwuprocesorowych. Zakładamy że są spełnione następujące założenia:

1. procesory są identyczne (mają tę samą szybkość działania),
2. obliczenia nie zawierają pętli i składają się z zadań podzielnych,
3. zadania są podzielne w dowolny sposób,
4. dany jest a priori czas wykonania zadań.

Przy tych założeniach skonstruujemy algorytm ustalający kolejność wykonywania zadań oraz dokonujący podziału zadań, tak by czas wykonania obliczenia był minimalny. Oczywiście, jeżeli zadanie uległo podziałowi, to dowolne jego części nie mogą być wykonywane równocześnie oraz części zadań są wykonywane w kolejności zgodnej z kolejnością ich podziału. Warunek ten oznacza, że zakładamy, iż żadne z zadań nie można podzielić na niezależne części. Skonstruowany algorytm będzie również wyznaczał minimalny czas wykonania obliczenia.

Jako model obliczenia przyjmujemy graf zorientowany opisany nie zawierający obwodów i mający tylko jeden wierzchołek końcowy. Model ten interpretujemy

w sposób następujący: każdemu wierzchołkowi grafu jest przyporządkowane tylko jedno zadanie obliczenia, przyporządkowanie zadań wierzchołkom grafu jest takie, że orientacja grafu jest zgodna z następstwem wykonywania zadań obliczenia.

W pracy [1] podano optymalny algorytm planowania obliczeń dla systemów dwuprocesorowych, przy założeniu, że obliczenia nie zawierają pętli i składają się z zadań podzielnych oraz, że czasy wykonania zadań mają wspólny podzielnik. Natomiast w pracy [2] podano optymalny algorytm planowania obliczeń dla systemów dwuprocesorowych, przy założeniu, że obliczenia nie zawierają pętli i składają się z zadań niepodzielnych o jednakowym czasie wykonania. Omówiono również rozszerzenie tego algorytmu na zadania podzielne, przy założeniu, że czasy wykonania zadań mają wspólną wielokrotność.

W niniejszej pracy podamy algorytm optymalnego planowania obliczeń spełniający warunki 1,2,3,4 i nie mający ograniczeń podanych w pracach [1] i [2] tzn., nie narzucamy żadnych ograniczeń na czasy wykonania zadań. Algorytm ten jest na tyle ogólny, że tkwią w nim potencjalne możliwości rozszerzenia na systemy wieloprocesorowe. Można go również traktować jako rozszerzenie algorytmu rozważanego w pracach [3], [4] i [5].

2. PODSTAWOWE POJĘCIA I DEFINICJE

Rozważmy graf opisany $G = \langle X, \Gamma, t, x_o \rangle$ gdzie:

X jest skończonym zbiorem wierzchołków

Γ jest relacją określoną na zbiorze X

t jest funkcją $(t : X \xrightarrow{w} R^+)$ przyporządkowującą każdemu wierzchołkowi liczbę rzeczywistą dodatnią

x_o jest wierzchołkiem końcowym, tzn. takim wierzchołkiem, dla którego zachodzi następująca relacja $\Gamma(x_o) = \emptyset$ /$\emptyset$ oznacza zbiór pusty/.

W dalszych rozważaniach wartość funkcji t dla argumentu x będziemy interpretowali jako czas wykonania zadania przyporządkowanego wierzchołkowi x.

Graf opisany G będziemy nazywali siecią, jeżeli jest on spójny, nie zawiera obwodów oraz posiada tylko jeden wierzchołek końcowy.

W dalszych rozważaniach ograniczymy się tylko do grafów będących sieciami. Sieci będziemy oznaczali literą S.

Niech $S = \langle X, \Gamma, t, x_o \rangle$ będzie dowolną siecią. Niech dla $i = 1,2,..,n$, $L'(i)$ będzie podzbiorem zbioru X zdefiniowanym w sposób następujący:

1^o $L'(1) = \{x_o\}$

2^o jeżeli $x \in Z_i$ oraz $t(x) = \min\limits_{y \in Z_i} \{t(y)\}$ to $x \in L'(i)$

gdzie Z_i jest zbiorem zdefiniowanym w sposób następujący:

$$Z_i = \left\{ x \in X : x \in X - \bigcup_{j=1}^{i-1} L'(j) \wedge \Gamma(x) \subset \bigcup_{j=1}^{i-1} L'(j) \right\}$$

Łatwo zauważyć, że jeżeli wartości funkcji t są jednakowe dla wszystkich wierzchołków sieci, to definicja zbioru $L'(i)$ /porównaj [4] i [5]/ przyjmuje następującą postać:

1^{o} $L'(1) = \{x_o\}$

2^{o} jeżeli $x \in X - \bigcup_{j=1}^{i-1} L'(j)$ oraz $\Gamma(x) \subset \bigcup_{j=1}^{i-1} L'(j)$ to $x \in L'(i)$

Niech p będzie liczbą naturalną spełniającą następującą relację $X = \bigcup_{j=1}^{p} L'(j)$, to wówczas dla $1 \leq i \leq p$ zbiór $L(i)$ zdefiniowany w sposób następujący:

$$L(i) = L'(p+1-i)$$

będziemy nazywali i-tym poziomem sieci S.

Łatwo się przekonać, że rodzina zbiorów $\{L(i)\}_{i=1,2,..,p}$ tworzy rozłączny podział zbioru wierzchołków X sieci S, tzn. że dla każdego $i,j \leq p$, jeżeli $i \neq j$ to $L(i) \cap L(j) = \emptyset$ oraz $\bigcup_{j=1}^{p} L'(j) = X$.

Sieć $S' = \langle X', \Gamma', t', x_o' \rangle$ nazywamy reduktom sieci $S = \langle X, \Gamma, t, x_o \rangle$, jeżeli

i) $X' \subset X$ i $\Gamma' \subset \Gamma$ oraz $t' \subset t$

ii) jeżeli $x \in X'$ to $\Gamma(x) \subset X'$

Łatwo zauważyć, że jeżeli sieć $S' = \langle X', \Gamma', t', x_o' \rangle$ jest reduktem sieci $S = \langle X, \Gamma, t, x_o \rangle$ to $x = x_o'$.

Redukt $S' = \langle X', \Gamma', t', x_o' \rangle$ nazywamy reduktem bezpośrednim sieci $S = \langle X, \Gamma, t, x_o \rangle$, jeżeli

i) $X - X' \neq \emptyset$

ii) $\Gamma(X - X') \subset X'$

Ciąg sieci $S_1,..,S_i,..,S_k$ nazywamy redukcją sieci S /piszemy R (S) /, jeżeli

i) $S_1 = S$

ii) dla każdego $i < k$, S_{i+1} jest reduktem bezpośrednim sieci S

iii) dla S_k nie istnieje redukt bezpośredni

Redukcję $R(S) = S_1,..,S_i,..,S_k$ nazywamy redukcją bezpośrednią sieci S, jeżeli dla każdego $i < k$ redukt S_{i+1} jest reduktem bezpośrednim sieci S_i.

Niech $S_i = \langle X_i, \Gamma_i, t_i, x_{oi} \rangle$ i $S = \langle X, \Gamma, t, x_o \rangle$ będą dowolnymi ustalonymi sieciami takimi, że $X \subset X_i$. Sieć S_i nazywamy bezpośrednim rozszerzeniem sieci S /piszemy $S \lesssim S_i$/, jeżeli istnieją takie dwa wierzchołki s_i i $s_i^{\oplus}$, że $s_i \in X \cap X_i$ oraz $s_i^{\oplus} \in X_i - X$, spełniające następujące warunki:

i) $\Gamma_i(s_i) = s_i^{\oplus}$ oraz $\Gamma(s_i) = \Gamma_i(s_i^{\oplus})$

ii) $t(s_i) = t_i(s_i) + t_i(s_i^{\oplus})$

iii) jeżeli $x \in X$ oraz $x \neq s_i$ to $\Gamma(x) = \Gamma_i(x)$

Natomiast wierzchołek s_i nazywamy wierzchołkiem osobliwym sieci S_i względem sieci S.

Jeżeli sieć S_i zawiera więcej niż jeden wierzchołek osobliwy, to sieć

S_1 nazywamy rozszerzeniem sieci S.

Należy podkreślić, że jeżeli sieć S_1 jest rozszerzeniem sieci S, to sieć S nie jest reduktem sieci S_1.

3. OGÓLNA IDEA ALGORYTMU

Omówimy teraz w sposób nieformalny ogólną ideę działania algorytmu planowania obliczeń. Aby nie zaciemniać ogólnego obrazu, zostaną pominięte pewne szczegóły nieistotne z punktu widzenia ogólnej zasady działania algorytmu. Mówiąc niezbyt precyzyjnie, istota algorytmu polega na wyszukiwaniu i odpowiednim grupowaniu zadań wzajemnie niezależnych, tak by łączny czas wykonania obliczenia złożonego z tych zadań był minimalny.

Algorytm działa przy przyjęciu założenia, że zadania są podzielne w dowolnych punktach czasowych. Jeżeli dokonamy podziału zadań, to struktura obliczenia ulega zmianie, co pociąga za sobą zmianę modelu obliczenia. Tak więc, w wyniku działania algorytmu uzyskujemy ciąg rozszerzeń sieci będących modelami obliczenia oraz ciąg reduktów tych rozszerzeń. Przy tworzeniu reduktów istotną rolę będą odgrywały pewne podzbiory wierzchołków F (i).

Przyjmiemy jeszcze pewne założenia upraszczające: wierzchołki sieci, którym są przyporządkowane zadania obliczenia będziemy identyfikowali z tymi zadaniami, a obliczenia będziemy identyfikowali z siecią będącą modelem tego obliczenia. Przyjęcie tego założenia nie zmniejsza ogólności przeprowadzanych tu rozważań.

Punktem wyjściowym działania algorytmu jest sieć $S_o = \langle X_o, \Gamma_o, t_o, x_{oo} \rangle$ będąca modelem obliczenia. Niech p oznacza liczbę poziomów sieci S_o. W początkowej fazie wykonywania pierwszego kroku algorytmu analizujemy kolejno po sobie wyrazy ciągu

$$(1) \quad L_o(1), \ldots, L_o(2), \ldots, L_o(n), \; L_o(n+1), \ldots, L_o(p_o)$$

aż znajdziemy taką liczbę naturalną n, że zachodzi następująca relacja

$$(2) \quad \Gamma_o\left(\bigcup_{j=1}^{n} L_o(j)\right) \cap L_o(n+1) \neq \emptyset$$

Następnie tworzymy zbiór wierzchołków M (0,n) określony w sposób następujący

$$(3) \quad M(0,n) = \bigcup_{1}^{n} L_o(j)$$

Jeżeli zbiór M (0,n) jest zbiorem jednoelementowym, to analizujemy w dalszym ciągu wyrazy ciągu (1), aż znajdziemy taką liczbę naturalną $n < m \leqslant p_o$, że zachodzi następująca relacja

$$(4) \quad \| L_o(m) - \left(\Gamma_o\left(\bigcup_{j=1}^{m-1} L_o(j)\right)\right) \| \geq 1$$

gdzie $\|X\|$ oznacza liczność zbioru X.

Jeżeli istnieje taka liczba naturalna $n < m \leqslant p_o$, że zachodzi relacja (4), to tworzymy pewien podzbiór D(0) wierzchołków sieci S_o określony w sposób następujący

$$(5) \quad D(0) = L_o(m) - \left(\Gamma_o\left(\bigcup_{j=1}^{m-1} L_o(j)\right)\right)$$

Jeżeli nie istnieje taka liczba naturalna $n < m \leq p_o$ spełniająca relację (4), to przyjmujemy, że zbiór $D(0)$ jest zbiorem pustym.

Natomiast jeżeli liczność zbioru $M(0,n)$ jest większa od jedności, to nie analizujemy powtórnie wyrazów ciągu (1).

Jeżeli $D(0)$ nie jest zbiorem pustym, to wybieramy ze zbioru $D(0)$ taki wierzchołek x_d, dla którego istnieje najmniejsze takie $1<i<p_o$, że zachodzi następująca relacja

$$x_d \in L_o(i) \cap D(0) \tag{6}$$

oraz uzupełniamy nim zbiór $M(0,n)$, tzn. tworzymy zbiór $M^{\oplus}(0)$ określony w sposób następujący

$$M^{\oplus}(0) = M(0,n) \cup \{x_d\} \tag{7}$$

W pozostałych przypadkach przyjmujemy, że $M^{\oplus}(0) = M(0,n)$.

Następnie w zbiorze $M(0,n)$ wyróżniamy pewien podzbiór X_s określony w sposób następujący

$$X_s = \{x \in M(0,n) : \Gamma_o(x) \cap L_o(n+1) = \emptyset\} \tag{8}$$

Jeżeli zbiór X_s nie jest zbiorem pustym, to dokonujemy podziału zadań ze zbioru X_s, tzn. tworzymy takie rozszerzenie $S_o^M = < X_o^M, \Gamma_o^M, t_o^M, x_o^M >$ sieci S_o by zbiór X_s był zbiorem wierzchołków osobliwych sieci S_o^M oraz był spełniony następujący warunek

$$\text{jeżeli } x \in X_s \text{ to } t_o^M(x) = \tau_o(x) + t_o(y) - \tau_o(y) \tag{9}$$

gdzie y jest dowolnym wierzchołkiem ze zbioru $M(0,n) - X_s$ a $\tau_o(x)$ sumą wartości funkcji t_o dla wierzchołków leżących na najdłuższej drodze w sieci S_o prowadzącej z wierzchołka x do wierzchołka x_{oo}.

Należy podkreślić, że zadania ze zbioru $M^{\oplus}(0)$ mają teraz nową interpretację zgodną z modelem S_o^M.

Natomiast jeżeli zbiór X_s jest zbiorem pustym, to nie dokonujemy podziału zadań ze zbioru $M(0,n)$, a sieć S_o dla wygody w dalszych rozważaniach będziemy oznaczali przez S_o^M.

Utworzenie zbioru $M^{\oplus}(0)$ i sieci S_o^M kończy pierwszą fazę wykonania kroku algorytmu.

Przystąpimy teraz do wyznaczania zbioru $F(1)$. Zbiór $F(1)$ określa zadania, które są wykonywane w pierwszym kroku algorytmu. Oczywiście, jeżeli $M^{\oplus}(0)$ jest zbiorem jednoelementowym, to

$$F(1) = M^{\oplus}(0) \tag{10}$$

oraz tylko jeden z procesorów będzie wykonywał zadania ze zbioru $F(1)$. Natomiast drugi z procesorów będzie oczekiwał na przydział zadania, które zostanie mu przydzielone w drugim kroku algorytmu.

Jeżeli zbiór $M^{\oplus}(0)$ jest zbiorem wieloelementowym oraz istnieje taki podzbiór $P(0)$ zbioru $M^{\oplus}(0)$, że

$$\sum_{x \in P(0)} t_o^M(x) = 1/2 \sum_{x \in M^{\oplus}(0)} t_o^M(x) \tag{11}$$

to niedokonujemy podziału zadań ze zbioru $M^{\oplus}(0)$ oraz zbiór $F(1)$ określamy w sposób następujący

(12) $F(1) = M^{\oplus}(0)$

Jeżeli nie istnieje podzbiór $P(0) \subset M^{\oplus}(0)$ spełniający warunek (11), to podzbiór P(0) wyznaczamy w ten sposób, by była spełniona następująca relacja

(13) $$\sum_{x \in P(0)} t_o^M(x) < 1/2 \sum_{x \in M^{\oplus}(0)} t_o^M(x)$$

Jeżeli liczność zbioru $M^{\oplus}(0) - P(0)$ jest większa od jedności, to dokonujemy podziału dowolnie wybranego zadania s_1 ze zbioru $M^{\oplus}(0) - P(0)$, tzn. tworzymy takie bezpośrednie rozszerzenie $S_1 = \langle X_1, \Gamma_1, t_1, x_{o1} \rangle$ sieci S_o^M by była spełniona następująca relacja:

(14) $$\sum_{x \in P(0) \cup \{s_1^{\oplus}\}} t_1(x) = 1/2 \sum_{x \in M^{\oplus}(0)} t_o^M(x)$$

oraz określamy zbiór F(1) w sposób następujący

(15) $$F(1) = M^{\oplus}(0) \cup \{ s_1^{\oplus} \}$$

Należy podkreślić, że w tym przypadku zadania ze zbioru F(1) mają interpretację zgodną z modelem S_1, a więc inną niż dla przypadku gdy zachodzi relacja (11).

Natomiast jeżeli zbiór $M^{\oplus}(0) - P(0)$ jest zbiorem jednoelementowym, to dokonujemy podziału jedynego zadania s_1 z tego zbioru, tzn. tworzymy takie bezpośrednie rozszerzenie $S_1 = \langle X_1, \Gamma_1, t_1, x_{o1} \rangle$ sieci S_o^M by była spełniona następująca relacja

(16) $$t_1(s_1) = \sum_{x \in P(0)} t_o^M(x)$$

oraz określamy zbiór F(1) w sposób następujący

(17) $$F(1) = M^{\oplus}(0)$$

Podobnie jak poprzednio zadania ze zbioru F(1) mają interpretację zgodną z modelem S_1.

Jeżeli zbiór F(1) jest zbiorem wieloelementowym, to działają równocześnie obydwa procesory wykonując zadania ze zbiorów P(0) i F(1) - P(0).

W wyniku wykonania pierwszego kroku algorytmu uzyskujemy nowy model obliczenia reprezentowany przez sieć $S_1^{\oplus} = \langle X_1^{\oplus}, \Gamma_1^{\oplus}, t_1^{\oplus}, x_{o1}^{\oplus} \rangle$, która w zależności od omawianych przypadków jest identyczna z siecią S_o^M lub z jej bezpośrednim rozszerzeniem S_1.

Przed przystąpieniem do wykonania drugiego kroku algorytmu tworzymy redukt S_1' sieci $S_1^{\oplus}$ określony w sposób następujący

(18) $$S_1' = \langle X_1^{\oplus} - F(1), \Gamma_1^{\oplus} / X_1^{\oplus} - F(1), t_1^{\oplus} / X_1^{\oplus} - F(1), x_o \rangle$$

Wykonanie drugiego kroku algorytmu przebiega w sposób podobny do wykonania pierwszego kroku algorytmu. Jako punkt wyjściowy działania drugiego kroku algorytmu przyjmujemy sieć S_1' określoną zależnością (18) zamiast sieci $S_o = \langle X_o, \Gamma_o, t_o, x_{oo} \rangle$ jak to miało miejsce poprzednio. Następnie analizujemy ciąg

(19) $$L_1'(1), \ldots, L_1'(n), L_1'(n+1), \ldots, L_1'(p_1')$$

poziomów sieci S_1' zgodnie z zasadami przyjętymi przy wykonywaniu pierwszego

kroku algorytmu oraz tworzymy zbiory $M'(1,n)$, $D'(1)$, $M^{\oplus}(1)$ przy założeniu, że są spełnione w zależności od sytuacji relacje (2), (3) , (4) , (5) , (6) , (7) dla sieci S_1'.

Następnie tworzymy rozszerzenie $S_1'^M$ sieci S_1' tak by były spełnione relacje (8) i (9) dla sieci S_1'. Po utworzeniu sieci $S_1'^M$ tworzymy zbiory $F(2)$ i $D'(1)$ tak, by były spełnione w zależności od sytuacji relacje (10), (11), (12), (13) , (14) , (15) dla sieci $S_1'^M$.

W wyniku wykonania drugiego kroku algorytmu uzyskujemy nowy model obliczenia $S_2^{\oplus}$, który w zależności od sytuacji jest identyczny z siecią $S_1^{\oplus}$ lub z jej rozszerzeniem.

Punktem wyjściowym działania trzeciego kroku algorytmu jest sieć S_2' będąca reduktem sieci $S_2^{\oplus}$, określona w sposób następujący

$$S_2' = \langle X_2^{\oplus} - \bigcup_1^2 F(i),\ \Gamma_1^{\oplus} / X_2^{\oplus} - \bigcup_1^2 F(i),\ t_2^{\oplus} / X_2^{\oplus} - \bigcup_1^2 F(i),\ x_o \rangle$$

Działanie algorytmu kończy się gdy uzyskamy redukt

$$S_k' = \langle \{x_o\},\ \Gamma_k^{\oplus} / \{x_o\},\ t_k^{\oplus} / \{x_o\},\ x_o \rangle$$

4. KONSTRUKCJA ALGORYTMU

Wprowadzone poprzednio pojęcia są niewystarczające do pełnego opisu algorytmu. Zachodzi konieczność wprowadzenia pewnych dodatkowych pojęć umożliwiających pełny opis algorytmu oraz pewnych pojęć umożliwiających zwartość opisu algorytmu.

Niech $S = \langle X, \Gamma, t, x_o \rangle$ będzie dowolną ustaloną siecią, a $S_1 = \langle X_i, \Gamma_i, t_i, x_{oi} \rangle$ dowolnym rozszerzeniem sieci S lub siecią identyczną z siecią S. Wyróżnimy pewien podzbiór $M(i,n)$ zbioru X_i określony w sposób następujący:

$$M(i,n) = \bigcup_{j=1}^{n} L_i(j) - \bigcup_{j=0}^{i} F(j)$$

gdzie $L_i(j)$ jest poziomem określonym dla sieci S_i, a $F(j)$ pewnym podzbiorem zbioru X_i. Zbiór $F(j)$ zostanie zdefiniowany w dalszych rozważaniach.

Zdefiniujemy teraz dwie funkcje n i g odgrywające zasadniczą rolę przy konstrukcji algorytmu. Funkcje te definiujemy w sposób następujący:

$$n(i) = \begin{cases} \text{najmniejsze takie } i < n < p_i \text{ że} \\ \Gamma_i(M(i,n)) \cap L_i(n+1) \neq \emptyset \\ i \quad \text{w przeciwnym przypadku} \end{cases}$$

$$g(i) = \begin{cases} \text{najmniejsze takie } n(i) < n \leq p_i \text{ że} \\ \| L_i(n) - (\Gamma_i(M(i,n-1)) \cup \bigcup_0^i F(j)) \| \geq 1 \\ n(i) \quad \text{w przeciwnym przypadku} \end{cases}$$

gdzie p_i jest liczbą naturalną spełniającą relację $\bigcup_{j=1}^{p_i} L_i(j) = X_i$, tzn. p_i wyznacza liczbę poziomów sieci S_i.

Niech $D(i) \subset X_i$ będzie zbiorem określonym w sposób następujący:

$$D(i) = L_i(g(i)) - (\Gamma_i(M(i,g(i)-1)) \cup \bigcup_{j=0}^{i} F(j))$$

Należy podkreślić, że istnieją przypadki gdy $D(i) = \emptyset$.

W zbiorze $D(i)$ wyróżnimy jeszcze pewien podzbiór $D^{\oplus}(i)$ określony w sposób następujący:

i) $\| D^{\oplus}(i) \| = 1$

ii) jeżeli $x \in D^{\oplus}(i)$ to $\tau_i(x) = \max\limits_{y \in D(i)} \{\tau_i(y)\}$

gdzie τ_i jest funkcją zdefiniowaną w sposób następujący:

$$\tau_i(x) = \begin{cases} t_i(x) & \text{jeżeli } x = x_{oi} \\ t_i(x) + \max\limits_{y \in \Gamma_i(x)} \{\tau_i(y)\} & \text{jeżeli } x \neq x_{oi} \end{cases}$$

W celu uzyskania bardziej zwartego opisu algorytmu wprowadzimy jeszcze pewne dodatkowe pojęcia.

Niech sieć $S_j = \langle X_j, \Gamma_j, t_j, x_{oj} \rangle$ będzie dowolnym rozszerzeniem sieci $S_i = \langle X_i, \Gamma_i, t_j, x_{oi} \rangle$. Niech X_s oznacza zbiór wierzchołków osobliwych sieci S_j. Sieć S_j nazywamy rozszerzeniem sieci S_i indukowanym przez zbiór $M(i,n(i))$, wtedy i tylko wtedy gdy są spełnione następujące warunki:

i) $X_s = \{ x \in M(i,n(i)) : \Gamma_i(x) \cap L_j(n(i)+1) = \emptyset \}$

ii) jeżeli $x \in X_s$ to $t_j(x) = \tau_i(x) - \tau_i(y) + t_j(y)$.

gdzie y jest dowolnym wierzchołkiem ze zbioru $M(i,n(i))$ spełniającym następującą relację:

$$\Gamma_i(y) \cap L_j(n(i)+1) \neq \emptyset$$

Łatwo zauważyć, że nie dla każdej sieci S_i istnieje rozszerzenie indukowane przez zbiór $M(i,n(i))$.

Rozszerzenie sieci S_i indukowane przez zbiór $M(i,n(i))$ będziemy w skrócie nazywali m-rozszerzeniem sieci S_i.

Celem uzyskania bardziej zwartego opisu, m-rozszerzenie sieci S_i lub sieć S_i o ile nie istnieje jej m-rozszerzenie, będziemy oznaczali przez S_i^M.

Niech $M^{\oplus}(i) \subset X_i$ będzie zbiorem zdefiniowanym w sposób następujący:

$$M^{\oplus}(i) = \begin{cases} M(i,n(i)) & \text{jeżeli } \| M(i,n(i)) \| > 1 \\ M(i,n(i)) \cup D^{\oplus}(i) & \text{jeżeli } \| M(i,n(i)) \| = 1 \end{cases}$$

a T i T_m funkcjami określonymi w sposób następujący:

$$T(i) = 1/2 \sum_{x \in M^{\oplus}(i)} \tau_i^M(x)$$

$$T_m(i) = \max_{x \in M^{\oplus}(i)} \{\tau_i^M(x)\}$$

W celu uproszczenia zapisu przyjmiemy następującą konwencję:

$$\sum_{x \in X} t_i(x) = t_i(X) .$$

Wyróżnimy jeszcze pewien podzbiór $P(i) \subset M^{\oplus}(i)$ zdefiniowany w sposób następujący:

$$P(i) = \left\{ x \in M^{\oplus}(i) \; : \; t_i^M(P(i)) \leq T(i) \text{ oraz nie istnieje takie } y \in M^{\oplus}(i) - P(i) \text{ że } t_i^M(P(i) \cup \{y\}) \leq T(i) \right\}$$

Łatwo zauważyć, że jeżeli zbiór $M^{\oplus}(i)$ jest zbiorem jednoelementowym to $P(i) = \emptyset$.

Opiszemy teraz proces tworzenia zbiorów $F(i)$ i przyporządkowywania sieci S jej rozszerzeń S_i. Proces ten będziemy nazywali algorytmem wstępnego porządkowania sieci S.

Definicja. Algorytm wstępnego porządkowania.

Proces tworzenia zbiorów $F(i)$ i rozszerzeń S_i sieci S definiujemy indukcyjnie w sposób następujący:

1. Krok podstawowy.

$F(0) = \emptyset$ oraz $S_o = S$

2. Krok indukcyjny

Przypadek 1.

Jeżeli $\| M^{\oplus}(i) \| = 1$ to $F(i+1) = M^{\oplus}(i)$ oraz $S_{i+1} = S_i$.

Przypadek 2.

Jeżeli $\| M^{\oplus}(i) \| > 1$ oraz

a) jeżeli $T(i) = T_m(i)$ to $F(i+1) = M^{\oplus}(i)$ oraz

$S_{i+1} = S_i^M$

b) jeżeli $T(i) < T_m(i)$ to $F(i+1) = M^{\oplus}(i)$ oraz

$S_i^M \subsetneq S_{i+1}$ i $\{s_{i+1}\} = M^{\oplus}(i) - P(i)$ oraz

$t_{i+1}(s_{i+1}) = t_i^M(P(i))$

c) jeżeli $T(i) > T_m(i)$ oraz

i) jeżeli $t_i^M(P(i)) = T(i)$ to $F(i+1) = M^{\oplus}(i)$ oraz

$S_{i+1} = S_i^M$

ii) jeżeli $t_i^M(P(i)) < T(i)$ to $F(i+1) = M^{\oplus}(i) \cup \{s_{i+1}^{\oplus}\}$

oraz $S_i^M \subsetneq S_{i+1}$ i $s_{i+1} \in M^{\oplus}(i) - P(i)$ oraz

$t_{i+1}(s_{i+1}) = t_i^M(P(i)) + t_i^M(s_{i+1}) - T(i)$

Niech ciąg $S_1, S_2, \ldots, S_k$ będzie ciągiem sieci przyporządkowanych sieci $S = \langle X, \Gamma, t, x_o \rangle$ w wyniku działania algorytmu wstępnego porządkowania. Sieć S_k będziemy nazywali końcowym rozszerzeniem sieci S jeżeli zachodzi następująca relacja:

$$X_k = \bigcup_1^k F(j)$$

a liczbę k będziemy nazywali rzędem rozszerzenia sieci S_k.

W dalszych rozważaniach końcowe rozszerzenie sieci S będziemy oznaczali przez $S^{\oplus}$.

Należy podkreślić, że jeżeli w wyniku działania algorytmu wstępnego porządkowania nie ulegnie podziałowi żadne z zadań obliczenia, to wówczas $S = S^{\oplus}$.

W wyniku działania algorytmu wstępnego porządkowania uzyskujemy rozłączny podział $\{F(i)\}_{i=1,2,..,k}$ zbioru wierzchołków $X^{\oplus}$ sieci $S^{\oplus}$. Jeżeli nie istnieje takie $1 \leqslant i \leqslant k$, że $s_i^{\oplus} \in F(i)$, to rozłączny podział $\{F(i)\}_{i=1,2,..,k}$ wyznacza grupy zadań składające się z zadań wzajemnie niezależnych. Tak więc, jeżeli $s_i^{\oplus} \in F(i)$, to zadania odpowiadające zbiorowi F(i) z wyjątkiem zadań przyporządkowanych wierzchołkom s_i i $s_i^{\oplus}$ można wykonywać równocześnie. Natomiast jeżeli $s_i^{\oplus} \notin F(i)$, to wszystkie zadania odpowiadająco zbiorowi F(i) można wykonywać równocześnie. Pozostaje jeszcze do rozwiązania problem przydziału zadań procesorom, tzn. problem takiego podziału zbiorów F(i) na dwa rozłączne zbiory, by łączny czas wykonania obliczenia był minimalny. Proces ten będziemy nazywali algorytmem podziału.

Definicja. Algorytm podziału.

Niech r będzie rzędem rozszerzenia sieci $S^{\oplus}$. W zbiorze $X^{\oplus}$, dla $1 \leqslant i \leqslant r$ oraz $j = 1,2$, wyróżnimy pewne podzbiory P(i,j) zdefiniowane w sposób następujący:

Przypadek 1.

Jeżeli $\|F(i)\| = 1$ to $P(i,j) = \begin{cases} F(i) & \text{jeżeli } j = 1 \\ \emptyset & \text{jeżeli } j = 2 \end{cases}$

Przypadek 2.

Jeżeli $\|F(i)\| > 1$

1. jeżeli $t_{i-1}(P(i-1)) = T(i-1)$ to

$$P(i,j) = \begin{cases} P(i-1) & \text{jeżeli } j = 1 \\ F(i) - P(i-1) & \text{jeżeli } j = 2 \end{cases}$$

2. jeżeli $t_{i-1}(P(i-1)) < T(i-1)$ oraz

i) jeżeli $T(i-1) < T_m(i-1)$ to

$$P(i,j) = \begin{cases} P(i-1) & \text{jeżeli } j = 1 \\ \{s_i\} & \text{jeżeli } j = 2 \end{cases}$$

ii) jeżeli $T(i-1) > T_m(i-1)$ to

$$P(i,j) = \begin{cases} P(i-1) \cup \{s_i^{\oplus}\} & \text{jeżeli } j = 1 \\ F(i) - (P(i-1) \cup \{s_i^{\oplus}\}) & \text{jeżeli } i = 2 \end{cases}$$

Niech P_j dla $j = 1,2$ będzie podzbiorem zbioru X określonym w sposób następujący:

$$P_j = \bigcup_{i=1}^r P(i,j).$$

Łatwo zauważyć, że zbiory P_1 i P_2 tworzą rozłączny podział zbioru $X^{\oplus}$. Zbiory P_1 i P_2 określają zadania, które będą przyporządkowane różnym procesorom. Jeżeli $P(i,j) \neq \emptyset$ dla $j = 1,2$, to zadania odpowiadające zbiorowi $P(i,1)$ są wykonywane przez pierwszy procesor, a zadania odpowiadające zbiorowi $P(i,2)$ są wykonywane przez drugi procesor. Jeżeli $P(i,1) \neq \emptyset$ oraz $P(i,2) = \emptyset$, to drugi procesor nie wykonuje żadnych zadań w czasie wykonywania przez pierwszy procesor zadań odpowiadających zbiorowi $P(i,1)$. Jeżeli $P(i,j) \neq \emptyset$ dla $j = 1,2$ to zadania odpowiadające zbiorom $P(i,j)$ mogą być wykonywane w dowolnej kolejności, o ile nie zachodzi następujący przypadek, że $s_i^{\oplus} \in P(i,1)$ oraz $s_i \in P(1,2)$. Jeżeli zachodzi powyższy przypadek, to zadanie odpowiadające wierzchołkowi s_i powinno być wykonane przed rozpoczęciem wykonywania zadania odpowiadającego wierzchołkowi $s_i^{\oplus}$. Oczywiście, najpierw powinny być wykonane zadania odpowiadające zbiorowi $P(i,j)$, a następnie zadania odpowiadające zbiorowi $P(i+1,j)$.

Czas wykonania $T(S)$ obliczenia złożonego z zadań przyporządkowanych wierzchołkom sieci S jest określony następującą zależnością:

$$T(S) = \sum_{1}^{r} T^{\oplus}(i)$$

gdzie $T^{\oplus}$ jest funkcją zdefiniowaną w sposób następujący:

$$T^{\oplus}(i) = \begin{cases} 1/2\ t_i(F(i)) & \text{jeżeli} \quad \|F(i)\| > 1 \\ t_i(F(i)) & \text{jeżeli} \quad \|F(i)\| = 1 \end{cases}$$

Z definicji funkcji $T^{\oplus}$ wynika, że po zakończeniu działania algorytmu wstępnego uporządkowania zadań, można już wyznaczyć łączny czas $T(S)$ wykonania obliczenia reprezentowanego przez sieć S, co oznacza, że do wyznaczenia czasu $T(S)$ nie jest wymagane zakończenie pełnego szeregowania zadań obliczenia Po zakończeniu pełnego szeregowania zadań obliczenia, tzn. po zakończeniu działania algorytmu wstępnego porządkowania i algorytmu podziału, dokonujemy przydziału procesorów. Inaczej mówiąc, łączny czas $T(S)$ wykonania obliczenia może być wyznaczony przed zakończeniem wykonania obliczenia.

5. OPTYMALNOŚĆ I INTERPRETACJA

Dowiedziemy teraz podstawowych własności algorytmu planowania obliczeń oraz jego optymalności. Podamy również interpretację podstawowych własności algorytmu.

Twierdzenie 1

Niech $S^{\oplus}$ będzie końcowym rozszerzeniem sieci S, a k rzędem rozszerzenia sieci $S^{\oplus}$. Niech graf R_i dla $1 \leq i \leq k$ będzie grafem określonym w sposób następujący:

$$(*) \quad R_i = \langle \bigcup_{i}^{k} F(j),\ \Gamma^{\oplus}/\bigcup_{i}^{k} F(j),\ t^{\oplus}/\bigcup_{i}^{k} F(j),\ x_o^{\oplus} \rangle$$

Jeżeli istnieje takie $1 \leq i < k$, że $s_i^{\oplus} \in F(i)$, to ciąg

$$(**) \quad R_f(S^{\oplus}) = R_1, \ldots, R_i, \ldots, R_k$$

nie jest redukcją bezpośrednią sieci $S^{\oplus}$ a jest tylko redukcją sieci $S^{\oplus}$. Natomiast jeżeli nie istnieje takie $1 \leq i < k$, że $s_i^{\oplus} \in F(i)$, to ciąg $(**)$ jest redukcją bezpośrednią sieci $S^{\oplus}$.

Dowód

Z definicji końcowego rozszerzenia sieci S wynika, że $R_1 = S^{\oplus}$. Ponieważ $R_k = < F(k), \Gamma^{\oplus}/F(k), t^{\oplus}/F(k), x_o^{\oplus} >$ to by dowieść twierdzenia trzeba pokazać, że dla $i < k$, jeżeli $s_i^{\oplus} \in F(i)$ to graf R_{i+1} jest reduktem sieci R_i, natomiast jeżeli $s_i^{\oplus} \notin F(i)$ to graf R_{i+1} jest reduktem bezpośrednim sieci R_i. Pokażemy teraz przez indukcję względem $i < k$, że graf R_{i+1} w zależności od sytuacji jest reduktem lub reduktem bezpośrednim sieci R_i.

1. Krok podstawowy. Niech i = 1. Na mocy (*) uzyskujemy

$$(1) \quad R_2 = < X^{\oplus} - F(1), \Gamma^{\oplus}/X^{\oplus} - F(1), t^{\oplus}/X^{\oplus} - P(1), x_o^{\oplus} >$$

Niech sieć $S_1 = < X_1, \Gamma_1, t_1, x_{o1} >$ będzie siecią przyporządkowaną sieci S w wyniku wykonania pierwszego kroku algorytmu wstępnego porządkowania. Jeżeli $s_1^{\oplus} \in F(1)$ to na mocy algorytmu wstępnego porządkowania sieć S_1 jest bezpośrednim rozszerzeniem sieci S^M oraz

$$(2) \quad F(1) = M^{\oplus}(0) \cup \{s_1\} \quad \text{i} \quad \Gamma_1(s_1) = s_1^{\oplus}$$

Natomiast jeżeli nie istnieje takie $i = 1$, że $s_1^{\oplus} \in F(1)$ to na mocy algorytmu wstępnego porządkowania sieć S_1 jest bezpośrednim rozszerzeniem sieci S^M lub jest siecią identyczną z siecią S oraz

$$(3) \quad F(1) = M^{\oplus}(0)$$

Ponieważ sieć S_1 jest bezpośrednim rozszerzeniem sieci S^M lub siecią identyczną z siecią S, to z (2) i (3) na mocy definicji zbioru $M^{\oplus}(i)$ uzyskujemy

$$(4) \quad \Gamma_1(H(1)) \subset X_1 - F(1)$$

gdzie zbiór H(1) jest zbiorem zdefiniowanym w sposób następujący

$$(5) \quad H(i) = \begin{cases} F(i) & \text{jeżeli } s_i^{\oplus} \notin F(i) \\ F(i) - \{s_i\} & \text{jeżeli } s_i^{\oplus} \notin F(i) \end{cases}$$

Ponieważ sieć $S^{\oplus}$ jest końcowym rozszerzeniem sieci S to na mocy (4) uzyskujemy

$$(6) \quad \Gamma^{\oplus}(H(1)) \subset X^{\oplus} - F(1)$$

Ponieważ $R_1 = S^{\oplus}$ oraz $\Gamma^{\oplus}(s_1) = s_1^{\oplus}$ to z (6) na mocy (5) i (1) uzyskujemy, że jeżeli $s_1^{\oplus} \notin F(1)$ to graf R_2 jest reduktem bezpośrednim sieci R_1, natomiast jeżeli $s_1^{\oplus} \in F(1)$ to graf R_2 jest reduktem sieci R_1 a nie jest reduktem bezpośrednim sieci R_1.

2. Krok indukcyjny. Załóżmy, że teza zachodzi dla $i = n < k$, tzn. że sieć

$$(7) \quad R_n = < \bigcup_n^k F(J), \Gamma^{\oplus}/\bigcup_n^k P(J), t^{\oplus}/\bigcup_n^k F(J), x_o^{\oplus} >$$

jest reduktem sieci R_{n-1} jeżeli $s_{n-1}^{\oplus} \in F(n-1)$ oraz jest reduktem bezpośrednim sieci jeżeli $s_{n-1}^{\oplus} \in F(n-1)$.

By dowieść tezy trzeba pokazać, że graf R_{n+1} jest reduktem sieci R_n jeżeli $s_n^{\oplus} \in F(n)$ oraz jest reduktem bezpośrednim sieci R_n jeżeli $s_n^{\oplus} \notin F(n)$.

Ponieważ

$$(8)\quad R_{n+1} = \langle \bigcup_{n+1}^{k} F(j),\ \Gamma^{\oplus} / \bigcup_{n+1}^{k} F(j),\ t^{\oplus} / \bigcup_{n+1}^{k} F(j),\ x_o^{\oplus} \rangle$$

oraz $\Gamma^{\oplus}(s_n) = s_n^{\oplus}$ to z (8) na mocy założenia indukcyjnego (7) wynika, że graf R_{n+1} w zależności od sytuacji jest reduktem sieci R_n lub reduktem bezpośrednim sieci R_n jeżeli jest spełniona następująca relacja:

$$(9)\quad \Gamma^{\oplus}(H(n)) \subset \bigcup_{n+1}^{k} F(j)$$

gdzie $H(n)$ jest zbiorem określonym przez (5).

Pokażemy teraz przez sprowadzenie do sprzeczności, że relacja (9) zachodzi. Załóżmy przeciwnie, tzn. że

$$(10)\quad \Gamma^{\oplus}(H(n)) \not\subset \bigcup_{n+1}^{k} F(j)$$

Ponieważ sieć $S^{\oplus}$ jest końcowym rozszerzeniem sieci S to z (10) na mocy (5) uzyskujemy

$$(11)\quad \Gamma^{\oplus}(F(n)) \cap \bigcup_{1}^{n} F(j) \neq \emptyset$$

Ponieważ sieć $S^{\oplus}$ jest końcowym rozszerzeniem sieci S to na mocy algorytmu wstępnego porządkowania oraz definicji zbioru $M^{\oplus}(i)$ uzyskujemy

$$(12)\quad \Gamma^{\oplus}(F(n)) \cap \bigcup_{1}^{n} F(j) = \emptyset$$

co jest sprzeczne z (11), a więc relacja (9) zachodzi, co oznacza, że jeżeli $s_n^{\oplus} \in F(n)$ to graf R_{n+1} jest reduktem sieci R_n a nie jest reduktem bezpośrednim, natomiast jeżeli nie istnieje takie $i = n$, że $s_n^{\oplus} \in F(n)$ to graf R_{n+1} jest reduktem bezpośrednim sieci R_n.

By dowieść twierdzenia trzeba jeszcze pokazać, że dla sieci R_k nie istnieje redukt.

Na mocy (12) uzyskujemy

$$(13)\quad \Gamma^{\oplus}(F(k)) \cap \bigcup_{1}^{k} F(j) = \emptyset$$

Ponieważ $S^{\oplus}$ jest końcowym rozszerzeniem sieci S to

$$(14)\quad F(k) = X^{\oplus} - \bigcup_{1}^{k-1} F(j)$$

Z (13) i (14) wynika, że dla sieci R_k nie istnieje redukt.

W dalszych rozważaniach, redukcję sieci $S^{\oplus}$ indukowaną przez zbiory F(i), tzn. ciąg

$$R_f(S^{\oplus}) = R_1, \ldots, R_i, \ldots, R_k$$

będziemy nazywali f-redukcją sieci $S^{\oplus}$.

Z twierdzenia 1 wynika, że jeżeli $x \in F(n)$ i $y \in F(m)$ oraz $n < m$ to nie istnieje droga z wierzchołka y do x i ponadto jeżeli $x, y \in F(i)$ i $s_i^{\oplus} \notin F(i)$ to nie zachodzą relacje $x \in \Gamma^{\oplus}(y)$ i $y \in \Gamma^{\oplus}(x)$.

W związku z powyższym rozłączny podział $\{F(i)\}_{i=1,2,\ldots,k}$ zbioru $X^{\oplus}$

sieci $S^{\oplus}$ można interpretować w sposób następujący: niech T_x oznacza zadanie przyporządkowane wierzchołkowi x.
Jeżeli $x,y \in F(i)$ oraz $s_i^{\oplus} \notin F(i)$ to zadania T_x i T_y można wykonywać równocześnie, natomiast jeżeli $s_i^{\oplus} \in F(i)$ to zadań T_{s_i} i $T_{s_i^{\oplus}}$ nie można wykonywać równocześnie, zadania $T_{s_i^{\oplus}}$ można wykonywać tylko po zakończeniu wykonania zadania T_{s_i}. Jeżeli $x \in F(n)$ i $y \in F(m)$ oraz $n < m$ to zadanie T_y powinno być wykonane po wykonaniu zadania T_x.

Niech $\prec$ będzie relacją częściowo porządkującą określoną na zbiorze $X^{\oplus}$, zdefiniowaną w sposób następujący:

$$x \prec y \iff x \in F(n) \wedge y \in F(m) \wedge n < m$$

Pokażemy teraz, że szeregowanie zadań odpowiadających zbiorom wierzchołków $X^{\oplus}$ rozszerzenia końcowego sieci S, oparte na relacji $\prec$ przy zachowaniu zasad przydziału procesorów indukowanych przez zbiory P(i,j), jest optymalne, tzn. że łączny czas wykonania obliczenia jest minimalny.

Twierdzenie 2

Niech $S' = \langle X', \Gamma', t', x_o' \rangle$ będzie takim rozszerzeniem sieci $S = \langle X, \Gamma, t, x_o \rangle$, że istnieje redukcja $R(S') = R_1', \ldots, R_n'$ spełniająca następujący warunek:

$$(*) \quad \begin{cases} \text{jeżeli } \| X_i' - X_{i+1}' \| = 2 \text{ to } \max_{x \in X_i' - X_{i+1}'} \{t'(x)\} = 1/2 \sum_{x \in X_i' - X_{i+1}'} t'(x) \\ \text{jeżeli } \| X_i' - X_{i+1}' \| > 2 \text{ to } \max_{x \in X_i' - X_{i+1}'} \{t'(x)\} \leq 1/2 \sum_{x \in X_i' - X_{i+1}'} t'(x) \end{cases}$$

Niech T' będzie funkcją określoną dla sieci S' w sposób następujący:

$$T'(i) = \begin{cases} 1/2 \sum_{x \in X_i' - X_{i+1}'} t'(x) & \text{jeżeli } \| X_i' - X_{i+1}' \| > 1 \\ t'(x) & \text{jeżeli } x = x_o' \text{ lub } \{x\} = X_i' - X_{i+1}' \end{cases}$$

Nie istnieje takie rozszerzenie S' sieci S, że

$$\sum_{1}^{n} T'(i) < T(S)$$

Dowód

Dowód twierdzenia przeprowadzimy przez sprowadzenie do sprzeczności. Załóżmy przeciwnie, tzn. że istnieje takie rozszerzenie S' sieci S spełniające warunek (*) oraz taka redukcja

(1) $R(S') = R_i', \ldots, R_i', \ldots, R_n'$

że zachodzi następujący warunek

(2) $\sum_{1}^{n} T'(i) < T(S)$

Niech $S^{\oplus}$ będzie końcowym rozszerzeniem sieci S, a ciąg

(3) $R_r(S^{\oplus}) = R_1, \ldots, R_i, \ldots, R_k$

będzie f-redukcją sieci $S^{\oplus}$.

Jeżeli zachodzi relacja (2) , to istnieje co najmniej jeden taki redukt R_{i+1}, że $\|F(i)\| = 1$. Niech R_{i+1} będzie pierwszym, w sensie numeracji elementów ciągu (3) , reduktem spełniającym następujący warunek

(4) $\|F(i)\| = 1$

Niech $F(i) = \{x_i\}$. Jeżeli zachodzi (4) to na mocy algorytmu wstępnego porządkowania uzyskujemy

(5) $M(i-1, n(i-1)) = \{x_i\}$ oraz $D(i-1) = \emptyset$

co oznacza, że istnieje taki wierzchołek $x \in \bigcup_1^{i-1} F(i)$, że $\Gamma^{\oplus}(x) \cap F(i) = \{x_i\}$ oraz nie istnieje wierzchołek $y \neq x$, taki że $y \in \bigcup_1^{i-1} F(i)$ oraz $\Gamma^{\oplus-1}(\Gamma^{\oplus}(y)) \subset \bigcup_1^{i-1} F(i)$, a więc dla sieci R_i nie istnieje taki bezpośredni redukt $R^{\oplus}_{i+1}$, że $\| \bigcup_1^{i-1} F(i) - X^{\oplus}_{i+1} \| > 1$. Jeżeli tak jest, to żadne rozszerzenie sieci S dotyczące wierzchołków $X - \bigcup_i^k F(i)$ nie powoduje zmiany wartości funkcji T(S). Mówiąc nieco dokładniej, niech S' będzie takim rozszerzeniem sieci S, że istnieje taka redukcja (1) oraz taki redukt R'_j należący do redukcji (1), że

(6) $X - X'_j = X - \bigcup_i^k F(i)$

Z (6) na mocy definicji funkcji T' i T uzyskujemy

(7) $\sum_1^{j-1} T'(n) \geqslant \sum_1^{i-1} T(n)$

Jeżeli redukt R_{i+1} jest jedynym reduktem sieci S spełniającym warunek (4) to na mocy (7) uzyskujemy

(8) $\sum_1^{n} T'(i) \geqslant T(S)$

Przeprowadzając podobne rozumowanie do kolejnych reduktów sieci $S^{\oplus}$ z ciągu (3) spełniających zależność (4) uzyskujemy, że dla każdego rozszerzenia S' sieci S spełniającego warunek (*) zachodzi następująca relacja

(9) $\sum_1^{n} T'(i) > T(S)$

co daje sprzeczność z (2).

Z powyższego twierdzenia wynika, że wartość funkcji T(S) wyznacza minimalny łączny czas wykonania obliczenia reprezentowanego przez sieć S. Natomiast z definicji funkcji T wynika, że łączny czas wykonania obliczenia można już wyznaczyć po zakończeniu działania algorytmu wstępnego porządkowania. Ponieważ do wykonania obliczenia wymagane jest zakończenie działania algorytmu wstępnego porządkowania i algorytmu podziału, to powyższy fakt oznacza, że łączny czas T (S) wykonania obliczenia można wyznaczyć przed zakończeniem wykonania obliczenia.

Algorytm wstępnego porządkowania działa przy założeniu, że zadania są podzielne. Punktem wyjściowym działania algorytmu są poziomy sieci będącej modelem obliczenia. Jeżeli dokonamy podziału zadań, to struktura obliczenia ulega

zmianie, co pociąga za sobą zmianę modelu obliczenia. W związku z tym powstaje konieczność wyznaczenia poziomów dla nowego modelu obliczenia. Pokażemy teraz, że wyznaczenie poziomów dla nowego modelu obliczenia jest sprawą trywialną.

Twierdzenie 3

Niech $S_i = \langle X_i, \Gamma_i, t_i, x_{oi} \rangle$ będzie siecią przyporządkowaną sieci $S = \langle X, \Gamma, t, x_o \rangle$ w wyniku wykonania jednego kroku algorytmu wstępnego porządkowania, a sieć $S = \langle X^M, \Gamma^M, t^M, x_o^M \rangle$ m-rozszerzeniem sieci S.

Jeżeli $S_i = S^M$ dla dla $1 \leqslant j \leqslant p$ $L_i(j) = L^M(j)$ oraz

$$(*) \qquad L^M(j) = \begin{cases} L(j) & \text{jeżeli } j \neq n(i) + 1 \\ L(j) \cup \Gamma^M(X_s) & \text{jeżeli } j = n(i) + 1 \end{cases}$$

gdzie X_s jest zbiorem wierzchołków osobliwych sieci S^M względem sieci S, a p oznacza liczbę poziomów sieci S.

Niech s_i będzie wierzchołkiem osobliwym sieci S_i względem sieci S^M a $s_i^{\oplus} \in X_i$ wierzchołkiem spełniającym następującą relację $\Gamma_i(s_i) = s_i^{\oplus}$.

Jeżeli $S^M \lneqq S_i$ oraz

i) jeżeli istnieje takie α że dla $x \in L(\alpha)$ zachodzi relacja
$\tau(x) = \tau(s_i) - t_i(s_i)$
to dla $1 \leqslant j \leqslant p$

$$(**) \qquad L_i(j) = \begin{cases} L^M(j) & \text{jeżeli } j \neq \alpha \\ L^M(j) \cup \{s_i^{\oplus}\} & \text{jeżeli } j = \alpha \end{cases}$$

ii) jeżeli istnieje takie α, że dla $x \in L(\alpha)$ i $y \in L(\alpha + 1)$ zachodzi relacja
$\tau(x) > \tau(s_i) - t_i(s_i) > \tau(y)$

to dla $1 \leqslant j \leqslant p+1$

$$(***) \qquad L_i(j) = \begin{cases} L^M(j) & \text{jeżeli } j \geqslant \alpha \\ \{s_i^{\oplus}\} & \text{jeżeli } j = \alpha + 1 \\ L^M(j-1) & \text{jeżeli } j > \alpha + 1 \end{cases}$$

Dowód

Z definicji m-rozszerzenia sieci wynika, że by dowieść relacji (*) wystarczy pokazać, że

$$(1) \qquad L^M(n(i) + 1) = L(n(i) + 1) \cup \Gamma^M(X_s)$$

Pokażemy teraz przez sprowadzenie do sprzeczności, że relacja (1) jest prawdziwa. Załóżmy przeciwnie, tzn. że

$$(2) \qquad L^M(n(i) + 1) \neq L(n(i) + 1) \cup \Gamma^M(X_s)$$

Z (2) na mocy definicji poziomu sieci oraz definicji funkcji τ wynika, że istnieje taki wierzchołek $x \in X_s$, że

(3) $\tau(x) - t^M(x) \neq \tau(y)$

gdzie y jest dowolnym wierzchołkiem ze zbioru $L(n(i)+1)$.

Natomiast na mocy definicji m-rozszerzenia sieci S uzyskujemy

(4) $$t^M(x) = \tau(x) - \tau(y)$$

Na mocy (3) i (4) uzyskujemy sprzeczność.

Jeżeli $s_1 \in L^M(m)$ to na mocy definicji poziomu sieci

(5) $$\Gamma^M(s_i) \subset \bigcup_{m+1}^{p} L^M(j)$$

Ponieważ $S^M \lneq S_i$ oraz s_i jest wierzchołkiem osobliwym sieci S_i względem sieci S^M, to z definicji bezpośredniego rozszerzenia sieci wynika, że istnieje dokładnie jeden taki wierzchołek $s_i^{\oplus} \in X_i$, że

(6) $$\begin{cases} t^M(s_i) = t_i(s_i) + t_i(s_i^{\oplus}) \\ \Gamma^M(s_i) = s_i^{\oplus} \quad \text{oraz} \quad \Gamma^M(s_i) = \Gamma_i(s_i^{\oplus}) \end{cases}$$

co oznacza, że przybywa co najmniej jeden poziom w sieci S_i.

Z (5) i (6) na mocy definicji poziomu sieci, definicji funkcji τ i liczby α wynika, że jeżeli $S^M \lneq S_i$ oraz jest spełnione założenie i), to zachodzi relacja (**), natomiast jeżeli jest spełniono założenie ii), to zachodzi relacja (***).

Z definicji algorytmu wstępnego porządkowania wynika, że w każdym kroku algorytmu może nastąpić podział zadań, co oznacza, że ulega zmianie model obliczenia. W związku z tym powstaje konieczność wyznaczenia poziomów dla sieci będącej nowym modelem obliczenia.

Natomiast z twierdzenia 3 wynika, że proces wyznaczania poziomów dla sieci będącej nowym modelem obliczenia sprowadza się, w zależności od sytuacji, do uzupełnienia tylko jednego poziomu następnikami wierzchołków osobliwych b_i do utworzenia nowego poziomu zawierającego tylko jeden następnik wierzchołka osobliwego i przenumerowania pewnej liczby poziomów.

Reasumując:

1. W wyniku działania algorytmów wstępnego porządkowania i podziału uzyskujemy uporządkowanie zadań obliczenia zapewniające minimalny czas wykonania obliczenia.
2. Do wyznaczenia minimalnego czasu wykonania obliczenia $T(S)$ wymagane jest tylko zakończenie działania algorytmu wstępnego porządkowania.

Literatura

[1] MUNTZ R.R., COFFMAN, Jr, E.G.: Optimal Preemptive Scheduling on Two-Processor Systems, IEEE Trans.on Comp., 1969, t. C-18, nr 11, s. 1014-1020.

[2] COFFMAN, Jr.E.G., GRAHAM R.L.: Optimal Scheduling for Two-Processor Systems, Acta Informatica, 1972, t.1, nr 3, s. 200-213.

[3] ROWICKI A.: Algorytm planowania obliczeń dwuprocesorowych, Prace IMM, 1975, t.17, nr 2, s. 65-83.

[4] ROWICKI A.: A Note on Optimal Scheduling for Two-Processor Systems, Information Processing Letters, 1975, t. 4, nr 2, s. 27-30.

[5] ROWICKI A.: On Optimal Scheduling for Two-Processor Systems, Bull. Acad. Polon. Sci., Sér. sci. math., astr. et phys., 1976, t.24, nr 4,s. 287-293.

Prace naukowo-badawcze Instytutu Maszyn Matematycznych

PLANOWANIE OBLICZEŃ NIEZALEŻNYCH PODZIELNYCH DLA SYSTEMÓW WIELOPROCESOROWYCH JEDNORODNYCH

Andrzej ROWICKI

W pracy podano dla systemów wieloprocesorowych optymalny algorytm planowania obliczeń, złożonych z zadań niezależnych o dowolnych czasach wykonania, przy założeniu, że zadania są podzielne w dowolny sposób. Zbadano podstawowe jego właściwości oraz podano warunki konieczne i dostateczne istnienia algorytmu. Rozważany algorytm zapewnia liczbę podziału mniejszą od liczby procesorów.

SPIS TREŚCI

1. UWAGI WSTĘPNE
2. OGÓLNA IDEA ALGORYTMU
3. PODSTAWOWE POJĘCIA I DEFINICJE
4. KONSTRUKCJA ALGORYTMU

1. UWAGI WSTĘPNE

Rozważymy zagadnienie optymalnego planowania obliczeń złożonych z n niezależnych zadań za pomocą m identycznych procesorów. Zakładamy, że zadania są podzielne oraz, że żadne zadanie nie może być wykonywane równocześnie przez dwa procesory. Ponadto zakładamy, że czas wykonywania zadań jest znany a priori. Jako kryterium optymalności przyjmujemy czas wykonania obliczenia, tzn. czas, po którym są wykonane wszystkie zadania obliczenia.

Podobne zagadnienia były rozważane w pracach [1] i [2]. W pierwszej pracy dopuszczono podział zadań oraz jako kryterium optymalności przyjęto minimalizację łącznej wartości strat. Natomiast w drugiej pracy założono, że zadania są niepodzielne, a jako kryterium optymalności przyjęto średni czas przepływu.

Niech $Z = \{z_1,\ldots,z_i,\ldots,z_n\}$ będzie zbiorem zadań niezależnych, a $t(z_i)$ oznacza czas wykonania zadania z_i. Niech $C(z)$ oznacza obliczenie złożone ze wszystkich zadań zbioru Z. Łatwo zauważyć, że jeżeli liczba zadań jest mniejsza od liczby procesorów, to w tym przypadku poprzez podział zadań nie można zoptymalizować czasu wykonania obliczenia. W tym przypadku czas wykonania obliczenia jest określony następującą zależnością:

$$t_o = \max_{x \in X} \{ t(x) \}$$

Jeżeli liczba procesorów jest mniejsza od liczby zadań obliczenia, to wówczas istnieje możliwość, poprzez odpowiedni podział zadań, zoptymalizowania czasu wykonania obliczenia. Optymalny czas wykonania obliczenia wynosi

$$t_o = \frac{\sum_{1}^{n} t(z_i)}{m}$$

Oczywiście optymalny czas obliczenia można uzyskać wtedy, gdy jest spełniony następujący warunek /patrz [1] /

$$\max_{x \in Z} \{ t(x) \} \leqslant \frac{\sum_{1}^{n} t(z_i)}{m}$$

Zajmiemy się teraz określeniem klasy obliczeń, dla której na drodze podziału można uzyskać optymalny czas wykonania obliczenia, tzn. czas, po którym są wykonane wszystkie zadania obliczenia.

Definicja 1. Podział m - regularny

Niech $Z = \{z_1,\ldots,z_i,\ldots,z_n\}$ będzie zbiorem zadań niezależnych, a $t(z_i)$ oznacza czas wykonania zadania z_i. Dla $m \leqslant n$, rozłączny podział P zbioru Z na podzbiory $P_o, P_1,\ldots,P_i,\ldots P_m$, takie że $\bigcup_{o}^{m} P_i = Z$ oraz jeżeli $i \neq j$, to $P_i \cap P_j = \emptyset$ /$\emptyset$ - oznacza zbiór pusty/, spełniający następujące warunki:

1° $\quad P_o = Z - \bigcup_1^m P_i$

2° $\quad$ jeżeli $\quad 0 < i < j \leq m \quad$ to $\quad t(P_i) \leq t(P_j) < \delta$

3° $\quad$ jeżeli $\quad x \in P_o \quad$ to dla $\quad 1 \leq i \leq m$

$t(x) \leq t(P_i) \quad$ oraz $\quad t(P_i \cup \{x\}) > \delta$

gdzie

$$\delta = \frac{\sum_1^n t(z_i)}{m}$$

$$t(P_i) = \sum_{x \in P_i} t(x)$$

nazywamy podziałem regularnym zbioru Z względem m, w skrócie podziałem m - regularnym zbioru Z.

Uwaga: Łatwo zauważyć, że istnieją takie przypadki, gdy $P_o = \emptyset$ dla $m < n$.

Definicja 2. Zbiór m - regularny

Jeżeli dla zbioru Z zadań niezależnych istnieje podział m - regularny, to zbiór Z nazywamy regularnym względem m, w skrócie m - regularnym.

Zbadamy teraz podstawowe właściwości zbiorów m - regularnych.

Własność 1

Niech $Z = \{z_1, \ldots, z_i, \ldots, z_n\}$ będzie zbiorem niezależnych zadań oraz

$$t_z = \max_{z_i \in Z} \{ t(z_i) \}$$

Warunkiem koniecznym i dostatecznym na to by zbiór Z był zbiorem m - regularnym jest następujący warunek:

$$t_z \leq \delta$$

Dowód

Dowód przeprowadzimy przez sprowadzenie do sprzeczności. Najpierw dowiedziemy dostateczność warunku, tzn. pokażemy, że zachodzi następująca implikacja

(1) Jeżeli $t_z \leq \delta$ to zbiór Z jest zbiorem m - regularnym.

Załóżmy, że tak nie jest, tzn. że zbiór Z nie jest zbiorem m - regularnym, co oznacza, że istnieje dla każdego rozłącznego podziału zbioru Z takie P_i dla $1 \leq i \leq m$, że $t(P_i) > \delta$. Ponieważ $t_z \geq t(P_i)$ to $t_z > \delta$, co jest sprzeczne z naszym założeniem /patrz (1) /.

Dowiedziemy teraz konieczności warunku, tzn. pokażemy, że zachodzi następująca implikacja

(2) Jeżeli zbiór Z jest zbiorem m - regularnym, to $t_z \leq \delta$

Załóżmy, że tak nie jest tzn., że $t_z > \delta$. Niech $P_o = \{x_1, \ldots, x_k\}$ oraz

$X = \{x_1,\ldots x_k, P_1,\ldots P_m\}$. Jeżeli $t_z > \delta$ to istnieje także $x \in X$ że $t(x) = t_z$, a więc $t(x) > \delta$, co oznacza, że zbiór Z nie jest zbiorem m - regularnym, a więc uzyskujemy sprzeczność z naszym założeniem.

Własność 2

Niech $\|Z\|$ oznacza liczność zbioru Z. Warunkiem koniecznym a niedostatecznym na to by zbiór Z był zbiorem m - regularnym jest następujący warunek

$$\|Z\| \geqslant m$$

Dowód

Najpierw pokażemy przez sprowadzenie do sprzeczności, że zachodzi następująca implikacja

(1) Jeżeli Z jest zbiorem m - regularnym to $\|Z\| \geqslant m$

Załóżmy, że tak nie jest tzn., że $\|Z\| < m$. Jeżeli $\|Z\| < m$ to na mocy definicji δ istnieje także $z_i \in Z$, że $t(z_i) > \delta$ co oznacza, że zbiór Z nie jest zbiorem m - regularnym, a więc uzyskujemy sprzeczność z naszym założeniem.

Pokażemy teraz przez podanie kontrprzykładu, że nie zachodzi następująca implikacja

(2) Jeżeli $\|Z\| \geqslant m$ to Z jest zbiorem m - regularnym

tzn., że warunek $\|Z\| \geqslant m$ nie jest warunkiem dostatecznym.

Rozważmy zbiór $Z = \{z_1,\ldots,z_n\}$ taki, że $t(z_i) = c$ dla $i \leqslant n-1$ oraz $t(z_n) = c\frac{m}{n-1}$. Łatwo się przekonać, że w tym przypadku $t(z_n) > \delta$, co oznacza, że zbiór Z nie jest zbiorem m - regularnym, a więc warunek $\|Z\| \geqslant m$ nie jest warunkiem dostatecznym.

Własność 3

Jeżeli podział $P = \{P_o, P_1,\ldots,P_m\}$ jest podziałem m - regularnym zbioru Z to $\|P_o\| \leqslant m-1$.

Dowód

Dowód przeprowadzimy przez sprowadzenie do sprzeczności. Przypuśćmy, że zachodzi następująca relacja $\|P_o\| > m-1$. Niech $\|P_o\| = m$ oraz $P_o = \{x_1,\ldots,x_m\}$. Na mocy definicji podziału m - regularnego uzyskujemy, że

$t(P_i) + t(x_j) > \delta$ co ostatecznie daje

$$(1)\quad \sum_1^m t(P_i) + t(P_o) > m \cdot \delta$$

Z drugiej strony na mocy definicji podziału m - regularnego uzyskujemy, że

$$(2)\quad \sum_1^m t(P_i) + t(P_o) = m \cdot \delta$$

co na mocy (1) daje sprzeczność.

Własność 4

Niech $P = \{P_o, P_1,\ldots,P_m\}$ będzie podziałem m - regularnym zbioru $Z = \{z_1,\ldots z_n\}$. Niech r oznacza liczbę podziałów zadań ze zbioru Z oraz niech

$P^{*} = P_1^{*},\ldots,P_m^{*}$ będzie ciągiem rozłącznych zbiorów utworzonych z elementów zbioru Z oraz z elementów będących wynikiem podziału zadań ze zbioru Z, spełniającym następujący warunek:

(*) $t(P_i^{*}) = \delta$ dla $1 \leq i \leq m$

Istnieje algorytm pozwalający utworzyć z elementów zbioru Z ciąg P^{*} na drodze podziału zadań, spełniający warunek (*) oraz następujący warunek

(**) $r < m$

Dowód

Pokażemy, że można dokonać takiego podziału zadań ze zbioru Z by były spełnione warunki (*) i (**).
Niech $\|P_o\| = k$ oraz $P_o = \{x_1,\ldots x_n\}$. Niech x_j^1 i x_j^2 oznaczają części, na które zostało podzielone zadanie $x_j \in P_o$. Ponieważ zbiór Z jest m - regularny, to zachodzą następujące relacje:

$$(1)\quad \sum_{1}^{k} \left(t(x_i^1) + t(P_i)\right) = k \cdot \delta$$

oraz

$$(2)\quad \sum_{1}^{k} t(x_i^2) + \sum_{k+1}^{m} t(P_i) = (m-k) \cdot \delta$$

Z (1) i (2) wynika, że dokonaliśmy k podziałów zadań. Natomiast z (2) wynika, że nie został rozwiązany problem uzupełniania zbiorów P_j dla $k+1 \leq j \leq m$ elementami ze zbioru $\{x_1^2,\ldots,x_k^2\}$ tak by była spełniona relacja (*). Pokażemy teraz przez sprowadzenie do sprzeczności, że przy liczbie podziałów zadań mniejszej od m-k można rozwiązać powyższy problem.

Załóżmy przeciwnie, tzn., że do uzupełnienia zbiorów P_j dla $k+1 \leq j \leq m$ elementami ze zbiorów $\{x_1^2,\ldots,x_k^2\}$ tak by była spełniona zależność (*) trzeba dokonać co najmniej m-k podziałów. Jeżeli trzeba dokonać m-k podziałów to oznacza, że zachodzi następująca relacja:

$$(3)\quad \sum_{1}^{k} t(x_i^2) + \sum_{k+1}^{m} t(P_i) > (m-k) \cdot \delta$$

Ponieważ zbiór Z jest m - regularny, to zachodzi relacja (2), a więc na mocy (3) uzyskujemy sprzeczność.

Z poprzednio przeprowadzonych rozważań oraz własność 1 wynika, że jeżeli zbiór zadań jest zbiorem m - regularnym, to tylko wtedy można uzyskać optymalny czas wykonania obliczenia. Jeżeli przyjmiemy interpretację, że liczba m oznacza liczbę procesorów, to własność 2 uściśla nasze intuicje związane z warunkami kiedy można uzyskać optymalny czas wykonania obliczenia. Własność 3 podaje związek zachodzący między liczbą procesorów a strukturą podziału.

Podział m - regularny można traktować jako wstępne grupowanie zadań do realizacji obliczenia i będzie on stanowił pozycję wyjściową do budowy algorytmu planowania obliczeń. W związku z powyższym powstaje pytanie, czy istnieją zbiory m - regularne, tzn. takie zbiory, dla których można utworzyć podział m - regularny. Odpowiedź na to pytanie jest pozytywna. Wynika ona z własności 1.

Jeżeli istnieją takie zbiory zadań, dla których $t_z \leq 6$ to te zbiory są m - regularne. Łatwo zauważyć, że istnieją takie zbiory zadań, dla których $t_z \leq 6$.

Pozostaje jeszcze zagadnienie utworzenia podziału m - regularnego. Z definicji podziału m - regularnego wynika, że nie trzeba stosować specjalnie wyrafinowanych metod, by utworzyć podział m - regularny. Jeżeli nie narzucamy dodatkowych warunków na podział, to wyznaczenie podziału m - regularnego dla zadanego zbioru jest sprawą trywialną.

Ponieważ zbiory m - regularne istnieją, to powstaje zagadnienie: ilu potrzeba dokonać podziałów zadań by zrealizować obliczenie złożone z zadań ze zbioru m - regularnego w czasie równym 6. Własność 4 określa potencjalne możliwości wykonania obliczenia, bowiem warunek 4 nie zapewnia sekwencyjności wykonania zadań podzielnych. Okazuje się, że można zbudować algorytm spełniający warunek 4 zapewniający poprawne wykonanie obliczenia. W dalszych rozważaniach zajmiemy się konstrukcją takiego algorytmu.

2. OGÓLNA IDEA ALGORYTMU

Omówimy teraz w sposób nieformalny ogólną ideę działania algorytmu planowania obliczeń. By nie zaciemniać ogólnego obrazu, zostaną pominięte pewne szczegóły nie istotne z punktu widzenia ogólnej zasady działania.

Zakładamy, że mamy dany zbiór $Z = \{z_1,\dots,z_n\}$ zadań niezależnych oraz zbiór $M = \{M_1,\dots,M_m\}$ identycznych procesorów. Zakładamy, że czas wykonania zadań mamy dany a priori. Ponadto zakładamy, że zadania są podzielne oraz, że żadne zadanie nie może być wykonywane równocześnie przez dwa procesory.

Przy konstrukcji algorytmu ograniczymy się tylko do zbiorów zadań m - regularnych, bowiem tylko w tym przypadku na drodze podziału zadań można uzyskać optymalny czas wykonania obliczenia. W związku z tym zakładamy, że mamy podział $P = \{P_o, P_1,\dots,P_m\}$ m - regularny zbioru Z oraz, że $m \leq n$.

Niech liczność zbioru P wynosi k oraz niech $P_o = \{x_1\dots,x_k\}$ będzie zbiorem uporządkowanym ze względu na następującą relację:
jeżeli $1 \leq i < j \leq k$ to $t(x_i) \leq t(x_j)$,
gdzie $t(x_i)$ oznacza czas wykonania zadania x_i. Punktem wyjściowym działania algorytmu będzie następujący zbiór:

$$S = \{x_1,\dots,x_k, P_o, P_1,\dots,P_j,\dots,P_m\}.$$

Liczba kroków algorytmu będzie zależna od liczby procesorów. W każdym kroku algorytmu będziemy dokonywali pewnych modyfikacji zbioru S zgodnie z ustalonymi regułami. Modyfikacje zbioru S będą określały stopień zaawansowania obliczenia. W związku z powyższym z każdym krokiem obliczenia będzie związany pewien stan obliczenia. Stan początkowy obliczenia będzie określany w sposób następujący:

$$S^{m+1} = \{\lambda^{m+1}(1)\dots, \lambda^{m+1}(k+1),\dots, \lambda^{m+1}(k+n)\}$$

gdzie λ^i dla $i=m+1$ jest funkcją określoną w sposób następujący:

$$\lambda^{m+1}(i) = \begin{cases} t(x_i) & \text{dla } 1 \leq i \leq k \\ t(P_{i-k}) & \text{dla } k < i \leq k+m \end{cases}$$

Przed przystąpieniem do wykonania obliczenia dokonujemy przydziału procesorów $M_1,\ldots,M_m$ zadaniom odpowiadającym kolejnym składowym stanu S^{m+1} w taki sposób by procesor M_i, dla $1 \leq i \leq m$, był przydzielony zadaniom odpowiadającym składowej $\lambda^{m+1}(i)$ stanu S^m.

Po wyznaczeniu dla stanu początkowego S^{m+1} średniego czasu działania procesorów, określonego w sposób następujący:

$$\delta_m = \frac{\sum_{1}^{k+m} \lambda^{m+1}(i)}{m}$$

wyznaczamy czas, po upływie którego zostanie przełączony procesor M_m. Czas ten jest określony następującą zależnością:

$$\delta_m^k = \delta_m - \lambda^{m+1}(k+m)$$

Po upływie czasu δ_m^k procesor M_m przydzielamy zadaniom odpowiadającym składowej $\lambda^{m+1}(k+m)$ stanu S^m, natomiast przydział pozostałych procesorów nie ulega zmianie. W dalszych krokach algorytmu przydział procesora M_m nie będzie ulegał zmianie. Po upływie czasu δ_m^k tzn., po wykonaniu pierwszego kroku algorytmu uzyskujemy nowy stan obliczenia:

$$S^m = \left\{ \lambda^m(1),\ldots, \quad \lambda^m(i),\ldots, \quad \lambda^m(k+m) \right\}$$

Natomiast funkcja λ^m jest określona w sposób następujący:

$$\lambda^m(i) = \begin{cases} \lambda^{m+1}(i) - \delta_m^k & \text{dla } 1 \leq i < m \\ \lambda^{m+1}(i) & \text{dla } m \leq i \leq k+m \end{cases}$$

W wyniku pierwszego kroku algorytmu problem wykonania zadań ze zbioru Z za pomocą m procesorów został sprowadzony do wykonania zadań określonych składowymi $\lambda^m(1),\ldots, \lambda^m(k+m-1)$ stanu S^m za pomocą m - 1 procesorów.

Postępując podobnie jak poprzednio wyznaczamy średni czas działania procesorów dla zadań określonych stanem S^m, w sposób następujący:

$$\delta_{m-1} = \frac{\sum_{1}^{k+m-1} \lambda^m(i)}{m-1}$$

oraz czas δ_{m-1}^k, po którym zostanie przełączony procesor M_{m-1}. Czas ten jest określony w sposób następujący:

$$\delta_{m-1}^k = \delta_{m-1} - \lambda^m(k+m-1)$$

Po upływie czasu δ_{m-1}^k przydzielamy procesor M_{m-1} zadaniom odpowiadającym składowej $\lambda^m(k+m-1)$ stanu S^m, przydział pozostałych procesorów nie ulega zmianie. Natomiast w dalszych krokach przydział procesora M_{m-1} nie będzie ulegał zmianie. Po wykonaniu drugiego kroku algorytmu, tzn. po upływie czasu $\delta_m^k + \delta_{m-1}^k$ uzyskujemy nowy stan obliczenia S^{m-1} określony w sposób następujący:

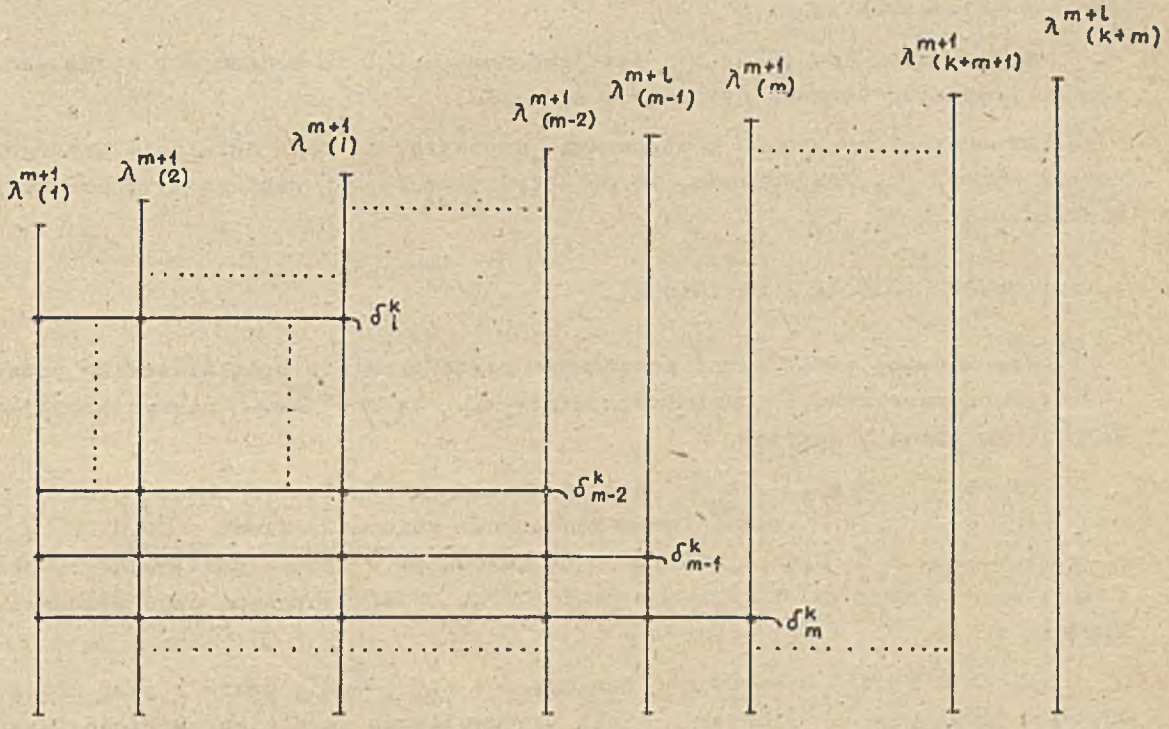
$$S^{m-1} = \left\{ \lambda^{m-1}(1), \ldots, \lambda^{m-1}(k+m) \right\}$$

oraz

$$\lambda^{m-1}(i) = \begin{cases} \lambda^{m}(i) - \delta^{k}_{m-1} & \text{dla } 1 \leqslant i \leqslant m-1 \\ \lambda^{m}(i) & \text{dla } m-1 < i \leqslant k+m \end{cases}$$

Działanie algorytmu kończy się, gdy problem wykonania zadań ze zbioru Z zostanie sprowadzony do zadań określonych składowymi $\lambda^{2}(1), \ldots, \lambda^{2}(k+1)$ stanu S^{2} za pomocą jednego procesora.

Schematycznie działanie algorytmu można przedstawić w sposób następujący:



Załóżmy, że został wykonany i-ty krok algorytmu, tzn. że został utworzony następujący stan obliczenia:

$$S^{m+1-i} = \left\{ \lambda^{m+1-i}(1), \ldots, \lambda^{m+1-i}(k+m) \right\}$$

Rozważmy przypadek gdy $\delta^{k}_{m-1} > \lambda^{m+1-i}(j)$ oraz $\delta^{k}_{m-1} \leq \lambda^{m+1-i}(j+1)$. W tym przypadku po wykonaniu kroku i-tego oprócz przydzielenia procesora M_{m-1} zadaniom odpowiadającym składowej $\lambda^{m+1-i}(k+m-1)$ stanu S^{m+1-i}, dokonujemy przełączania procesorów przyporządkowanych zadaniom odpowiadającym składowym $\lambda^{m+1-i}(1), \ldots, \lambda^{m+1-i}(j)$, na zadania odpowiadające składowym $\lambda^{m+1-i}_{j+1}, \ldots, \lambda^{m+1-i}(k+m-i)$ z zachowaniem porządku zgodnego z numeracją procesorów i składowych stanu S^{m}.

W przypadku gdy zbiór P_o jest zbiorem pustym, to dokonujemy tylko jednorazowego przyporządkowania procesorów zadaniom.

Przedstawiony sposób postępowania zapewnia, że czas działania każdego procesora wynosi δ_m, co oznacza, że po upływie czasu δ_m zostaną wykonane wszystkie zadania.

3. PODSTAWOWE POJĘCIA I DEFINICJE

Wprowadzimy teraz pewne podstawowe pojęcia, które będą stanowiły podstawę teoretyczną konstrukcji i działania algorytmu. Są one jednak niewystarczające do pełnego opisu algorytmu.

Niech $Z = \{z_1, \ldots, z_n\}$ będzie zbiorem m-regularnym zadań oraz $P = \{P_o, P_1, \ldots, P_m\}$ podziałem m-regularnym zbioru Z. Niech $\|P_o\| = k$ oraz $P_o = \{x_1, \ldots x_k\}$ będzie zbiorem uporządkowanym w sposób następujący: jeżeli $1 \leq i < j \leq k$ to $t(x_i) \leq t(x_j)$; gdzie $t(x)$ oznacza czas wykonania zadania x.

Niech δ będzie następującą funkcją $\delta : N \times N \xrightarrow{w} R^+$, gdzie N jest zbiorem liczb naturalnych, a R^+ zbiorem liczb rzeczywistych dodatnich. Wartość funkcji δ dla argumentów x, y spełniających następujące warunki: $0 \leq x \leq m-1$, $1 \leq y \leq m$ będziemy oznaczali w sposób następujący δ^{x}_{y}.

Niech sg i $\dot{-}$ będą funkcjami określonymi w sposób następujący:

$$sg(x) = \begin{cases} 1 & \text{jeżeli } x > 0 \\ 0 & \text{jeżeli } x \leq 0 \end{cases} \qquad x \dot{-} y = \begin{cases} x - y & \text{jeżeli } x \geq y \\ 0 & \text{jeżeli } x < y \end{cases}$$

Zdefiniujemy teraz dwie funkcje $p : R^+ \to Y(k+n+1)$ i $\lambda : Y(k+m) \to R^+$ gdzie $Y(n) = 1,2,\ldots,n$, wzajemnie skojarzone odgrywające zasadniczą rolę przy konstrukcji algorytmu.

Definicja 3. Funkcje p i λ

Funkcje p i λ definiujemy indukcyjnie w sposób następujący:

1. Krok podstawowy

$$p(\delta^{k}_{m}) = m$$

$$\lambda^{m}(i) = \lambda^{m+1}(i) - \delta^{k}_{m} \cdot sg(m+1-i)$$

gdzie $\lambda^{m+1}(i) = \begin{cases} t(x_i) & \text{dla } 1 < i \leq k \\ t(P_{i-k}) & \text{dla } k < i \leq k+m \end{cases}$

2. Krok indukcyjny

i) jeżeli $\lambda^{i+1}(n_i) \geq \delta_i^k$ dla $n_i = p(\delta_{i+1}^k) - i$ to

$p(\delta_i^k) = p(\delta_{i+1}^k) - 1$

$\lambda^i(j) = \lambda^{i+1}(j) - \delta_i^k \cdot sg(p(\delta_i^k) + 1 - j)$

ii) jeżeli $\lambda^{i+1}(n_i) < \delta_i^k$ dla $n_i = p(\delta_{i+1}^k) - i$ to

$p(\delta_i^k) = \mu(i) + i - 1$

$$\lambda^i(j) = \begin{cases} 0 & \text{dla } j < \mu(i) \\ \sum_{\alpha=0}^{q(i,j)} (\lambda^{i+1}(n_i + r(i,j) + \alpha \cdot i) - \delta_i^k \cdot sg\alpha) & \\ & \text{dla } \mu(i) \leq j \leq \mu(i) + i - 1 \\ \lambda^i(j+1) & \text{w pozostałych przypadkach} \end{cases}$$

gdzie:

$q(i,j)$ - część całkowita z dzielenia $(j - n_i)$ przez i

$r(i,j)$ - reszta z dzielenia $(j - n_i)$ przez i

Natomiast μ jest funkcją zdefiniowaną w sposób następujący

Definicja 4. Funkcja μ

Funkcję μ definiujemy w sposób następujący:

$$\mu(i) = \begin{cases} \text{najmniejsze takie } n_i < n \leq m + k \\ \text{że } \lambda^{i+1}(n) \geq \delta_i^k \\ m + k + 1 \quad \text{jeżeli nie istnieje takie } n_i < n \leq m + k \\ \text{że } \lambda^{i+1}(n) \geq \delta_i^k \end{cases}$$

Definicja 5. Funkcja δ

Dla $1 \leq i \leq m$ oraz $0 \leq k \leq m-1$ funkcję δ definiujemy w sposób następujący:

$$\delta_i^k = \beta_i - \lambda^{i+1}(k+i)$$

gdzie: $\beta_i = \dfrac{\sum_{1}^{k+1} \lambda^{i+1}(j)}{i}$

To trzy funkcje p, λ i δ stanowią podstawę teoretyczną konstrukcji i działania algorytmu planowania obliczeń. Można je interpretować w sposób następujący: funkcja λ określa stopień zaawansowania obliczenia, funkcja p określa przyporządkowanie procesorów określonym grupom zadań, natomiast funkcja δ określa czas przełączenia procesorów.

Przystąpimy teraz do zbadania podstawowych właściwości omawianych funkcji, interesujących z punktu widzenia konstrukcji algorytmu planowania obliczeń.

Twierdzenie 1

Jeżeli zbiór zadań jest m - regularny to dla $1 \leqslant i \leqslant m$ oraz $k = 0$

$$\delta_i^k = 0$$

Dowód

Dowód przeprowadzimy przez indukcję względem i.

1° Niech $i = m$. Ponieważ zbiór zadań jest m-regularny, to na mocy definicji podziału m - regularnego uzyskujemy

(1) $\lambda^{m+1}(i) = 6$ dla $1 \leqslant i \leqslant m$

Co na mocy definicji funkcji δ daje $\delta_m^o = 0$.

2° Załóżmy, że dla $i = n$ teza zachodzi tzn, że

(2) $\delta_n^o = 0$

Na mocy założenia indukcyjnego (2) z definicji funkcji λ wynika

(3) $\lambda^j(i) = \lambda^{m+1}(i)$ dla $1 \leqslant i \leqslant m$ oraz $n \leqslant j \leqslant m + 1$

Natomiast na mocy definicji funkcji δ z (2) i (3) uzyskujemy

(4) $\sum_1^{n-1} \lambda^n(i) = (n-1) \cdot \lambda^n(n)$

z (4) na mocy definicji funkcji δ uzyskujemy

(5) $\delta_{n-1}^o = \lambda^n(n) - \lambda^n(n-1)$

z (5) na mocy (3) i (1) uzyskujemy ostatecznie

(6) $\delta_{n-1}^o = 0$

Twierdzenie 2

Jeżeli zbiór zadań jest m - regularny to dla $1 \leqslant j \leqslant m$ oraz $0 < k \leqslant m-1$

$$\delta_j^k \geqslant 0$$

Dowód

Dowód przeprowadzimy przez indukcję względem j.

1° Ponieważ zbiór zadań jest m - regularny to na mocy definicji funkcji δ

(1) $\delta_m^k \geqslant 0$

2° Załóżmy, że teza zachodzi dla $j = \alpha + 1$ tzn, że

(2) $\delta_{\alpha+1}^k \geqslant 0$

Rozważmy przypadek gdy $\mu(\alpha) \neq 0$. Jeżeli $\mu(\alpha) \neq 0$ to na mocy definicji funkcji p istnieje taka liczba n_α, że

(3) $\begin{cases} \lambda^{\alpha+1}(n_\alpha) < \delta_\alpha^k \\ \lambda^{\alpha+2}(n_\alpha) \geqslant \delta_{\alpha+1}^k \end{cases}$

Natomiast na mocy definicji funkcji λ uzyskujemy

$$(4)\quad \lambda^{\alpha+1}(n_\alpha) = \lambda^{\alpha+2}(n_\alpha) - \delta^k_{\alpha+1} \cdot sg(p(\delta^k_{\alpha+1}) + 1 - n_\alpha)$$

z (3) i (4) na mocy założenia indukcyjnego (2) uzyskujemy ostatecznie

$$(5)\quad \delta^k_\alpha > 0$$

Rozważmy przypadek gdy $\mu(\alpha) = 0$. Jeżeli $p(\delta^k_\alpha) = \alpha$ to na mocy definicji funkcji p, $\mu(i) = 0$ dla $i \geqslant \alpha$. Jeżeli $\mu(i) = 0$ dla $i \geqslant \alpha$ to na mocy definicji λ uzyskujemy

$$(6)\quad \sum_1^{k+\alpha} \lambda^{\alpha+1}(j) = \sum_1^{k+\alpha} \lambda^{\alpha+2}(j) - (\alpha+1) \cdot \delta^k_\alpha$$

Natomiast na mocy definicji funkcji δ uzyskujemy

$$(7)\quad \begin{cases} \delta^k_{\alpha+1} = \dfrac{\sum_1^{k+\alpha+1} \lambda^{\alpha+2}(j)}{\alpha+1} - \lambda^{\alpha+2}(k+\alpha+1) \\ \delta^k_\alpha = \dfrac{\sum_1^{k+\alpha} \lambda^{\alpha+1}(j)}{\alpha} - \lambda^{\alpha+1}(k+\alpha) \end{cases}$$

Z (6) na mocy (7) uzyskujemy ostatecznie

$$(8)\quad \delta^k_\alpha = \lambda^{\alpha+2}(k+\alpha+1) - \lambda^{\alpha+1}(k+\alpha)$$

Ponieważ $\mu(\alpha) = 0$ oraz $p(\delta^k_\alpha) = \alpha$ to na mocy definicji funkcji λ uzyskujemy

$$(9)\quad \lambda^{\alpha+2}(k+\alpha+1) \geqslant \lambda^{\alpha+1}(k+\alpha)$$

Z (8) na mocy (9) uzyskujemy ostatecznie

$$(10)\quad \delta^k_\alpha \geqslant 0$$

Niech $p(\delta^k_\alpha) > \alpha$. Jeżeli $p(\delta^k_\alpha) > \alpha$ to na mocy definicji funkcji p istnieje takie $i_o > \alpha$, że $\mu(i_o) \neq 0$.

Niech i_o będzie największą taką liczbą, że $\mu(i_o) \neq 0$, to wówczas na mocy definicji funkcji λ uzyskujemy

$$(11)\quad \lambda^{i_o}(j) = 0 \quad \text{dla} \quad 1 \leqslant j \leqslant \mu(i_o)$$

Natomiast z (11) na mocy definicji funkcji λ uzyskujemy

$$(12)\quad \sum_1^{k+\alpha} \lambda^{\alpha+1}(j) \geqslant \sum_1^{k+\alpha} \lambda^{\alpha+2}(j) - (\alpha+1) \cdot \delta^k_\alpha$$

Z (12) na mocy (7) uzyskujemy ostatecznie

$$(13)\quad \delta^k_\alpha \geqslant \lambda^{\alpha+2}(k+\alpha+1) - \lambda^{\alpha+1}(k+\alpha)$$

Co oznacza, że

$$(14)\quad \delta^k_\alpha \geqslant 0.$$

Twierdzenie 3

Niech $P = \{P_0, P_1, \ldots, P_m\}$ będzie podziałem m - regularnym zbioru zadań. Jeżeli $\|P_0\| = k$ to dla $1 \leq i \leq m$,

$$p(\delta_i^k) < i + k$$

Dowód

Przez sprowadzenie do sprzeczności. Załóżmy przeciwnie, tzn, że istnieje takie i_0, że

$$(1)\quad p(\delta_{i_0}^k) \geq i_0 + k$$

Niech $p(\delta_{i_0}^k) = i_0 + k$ oraz i_0 będzie najmniejszą taką liczbą, że zachodzi relacja (1) tzn, że

$$(2)\quad p(\delta_i^k) = i + k \quad \text{dla} \quad i < i_0$$

Na mocy definicji funkcji p, z (2) wynika, że $\mu(i_0) = k + 1$ oraz $\mu(i) = 0$ dla $i < i_0$.

Jeżeli $\mu(i_0) = k + 1$ oraz $\mu(i) = 0$ dla $i < i_0$ to na mocy definicji funkcji λ uzyskujemy

$$(3)\begin{cases} \lambda^{i_0}(n) = 0 & \text{dla} \quad 1 \leq n \leq k \\ \lambda^{i_0}(n) \geq \lambda^{m+1}(n) - \sum\limits_{i_0}^{m} \delta_j^k \cdot sg(p(\delta_j^k) + 1 - n) & \\ & \text{dla} \quad k < n \leq m+k \end{cases}$$

Rozważmy przypadek gdy $\mu(i) = 0$ dla $i > i_0$.

Jeżeli $\mu(i) = 0$ dla $i > i_0$ to z (2) na mocy definicji funkcji λ uzyskujemy, że

$$(4)\quad \sum_1^k \lambda^{m+1}(n) < \sum_1^k \sum_{i_0}^m \delta_j^k \cdot sg(p(\delta_j^k) + 1 - n)$$

oraz

$$(5)\quad \sum_1^m \lambda^{i_0}(k+n) = \sum_1^m \lambda^{m+1}(k+n) - \sum_1^m \sum_{i_0}^m \delta_j^k \, sg(p(\delta_j^k) + 1 - k - n) +$$

$$+ \left(\sum_1^k \lambda^{m+1}(n) - \sum_1^k \sum_{i_0}^m \delta_j^k \, sg(p(\delta_j^k) + 1 - n) \right)$$

Ponieważ zbiór zadań jest m - regularny to

$$(6)\quad \sum_1^k \lambda^{m+1}(n) = m \cdot 6 - \sum_1^m \lambda^{m+1}(k+n)$$

Z (6) na mocy (3), (4) i (5) uzyskujemy ostatecznie

$$(7)\quad \sum_1^{k+m} \lambda^{m+1}(n) < m \cdot 6$$

Ponieważ

$$(8) \quad m \cdot 6 = \sum_{1}^{k+m} \lambda^{m+1}(n)$$

to na mocy (7) i (8) uzyskujemy sprzeczność.

By dowieść twierdzenia należy jeszcze rozważyć przypadek gdy $\mu(i) \neq 0$ dla $i > i_0$. Załóżmy, że istnieje taki ciąg liczb $\alpha_1, \ldots, \alpha_n, \ldots, \alpha_p$, że $\alpha_n > \alpha_{n+1} > i_0$ dla $n = 1, \ldots, i, \ldots, p-1$ oraz, że $\mu(\alpha_n) \neq 0$. Załóżmy, że α_1 jest największą taką liczbą, że $\mu(\alpha_1) \neq 0$, a α_p najmniejszą taką liczbą, że $\mu(\alpha_p) \neq 0$, tzn, że $\mu(i) = 0$ dla $i \neq \alpha_n$ oraz $i > i_0$. W tym przypadku na mocy (2) oraz definicji funkcji λ uzyskujemy

$$(9) \quad \sum_{1}^{m} \lambda^{i} e(k+1) = \sum_{1}^{m} \lambda^{m+1}(k+1) - \sum_{1}^{m} \sum_{i_0}^{m} \delta_j^k \cdot sg(p(\delta_j^k)+1-k-1) +$$

$$+ \Big(\sum_{1}^{\mu(\alpha_1)-1} \lambda^{m+1}(1) - \sum_{1}^{\mu(\alpha_1)-1} \sum_{\alpha_1}^{m} \delta_j^k \cdot sg(p(\delta_j^k)+1-1) \Big) +$$

$$+ \Big(\sum_{\mu(\alpha_1)}^{\mu(\alpha_2)-1} \lambda^{m+1}(1) - \sum_{\mu(\alpha_1)}^{\mu(\alpha_2)-1} \sum_{\alpha_2}^{m} \delta_j^k \cdot sg(p(\delta_j^k)+1-1) \Big) +$$

$$+ \ldots + \Big(\sum_{\mu(\alpha_{n-1})}^{\mu(\alpha_n)-1} \lambda^{m+1}(1) - \sum_{\mu(\alpha_{n-1})}^{\mu(\alpha_n)-1} \sum_{\alpha_n}^{m} \delta_j^k \cdot sg(p(\delta_j^k)+1-1) \Big) +$$

$$+ \ldots + \Big(\sum_{\mu(\alpha_p)}^{k} \cdot \sum_{i_0}^{m} \lambda^{m+1}(1) - \sum_{\mu(\alpha_p)}^{k} \sum_{i_0}^{m} \delta_j^k \cdot sg(p(\delta_j^k)+1-1) \Big)$$

oraz

$$(10) \quad \sum_{1}^{k} \lambda^{m+1}(1) < \sum_{1}^{k} \sum_{i_0}^{m} \delta_j^k \cdot sg(p(\delta_j^k)+1-1)$$

Na mocy (9) uzyskujemy

$$(11) \quad \sum_{1}^{m} \lambda^{i} e(k+1) \geq \sum_{1}^{m} \lambda^{m+1}(k+1) - \sum_{1}^{m} \sum_{i_0}^{m} \delta_j^k \cdot sg(p(\delta_j^k)+1-k-1) +$$

$$+ \Big(\sum_{1}^{k} \lambda^{m+1}(1) - \sum_{1}^{k} \sum_{i_0}^{m} \delta_j^k \cdot sg(p(\delta_j^k)+1-1) \Big)$$

Z (6) na mocy (3), (10) i (11) uzyskujemy ostatecznie

$$(12) \quad \sum_{1}^{k+m} \lambda^{m+1}(1) < m \cdot 6$$

Co na mocy (8) daje sprzeczność.

Twierdzenie 4

Jeżeli zbiór zadań jest m - regularny to dla $1 \leq i \leq m$

$$\lambda^{i+1}(k+i) + \sum_{i}^{m} \delta_{j}^{k} = \sigma$$

Dowód

Przeprowadzimy przez indukcję względem i.

1° Ponieważ $\sigma = \sigma_m$ to na mocy definicji funkcji δ dla i = m teza zachodzi.

2° Załóżmy, że teza zachodzi dla i = n, tzn, że

$$(1) \quad \lambda^{n+1}(k+n) + \sum_{n}^{m} \delta_{j}^{k} = \sigma$$

Niech i = n - 1. Pokażemy teraz przez sprowadzenie do sprzeczności, że zachodzi następujący związek

$$(2) \quad \lambda^{n}(k+n-1) + \sum_{n-1}^{m} \delta_{j}^{k} = \sigma$$

Załóżmy przeciwnie, tzn, że

$$(3) \quad \lambda^{n}(k+n-1) + \sum_{n-1}^{n} \delta_{j}^{k} \neq \sigma$$

Na mocy założenia indukcyjnego (1) oraz definicji funkcji δ z (3) wnioskujemy ostatecznie

$$(4) \quad \sigma_{n-1} \neq \lambda^{n+1}(k+n)$$

Ponieważ (patrz def. funkcji λ)

$$(5) \quad \lambda^{n}(j) = \lambda^{n+1}(j) - \delta_{n}^{k} \cdot sg(p(\delta_{j}^{k}) + 1 - j)$$

to z (4) na mocy twierdzenia 3 oraz definicji σ_{n-1} uzyskujemy

$$(6) \quad \sum_{1}^{k+n} \lambda^{n+1}(j) - n \cdot \delta_{n}^{k} \neq n \cdot \lambda^{n+1}(k+n)$$

z (6) na mocy definicji σ_n uzyskujemy

$$(7) \quad \delta_{n}^{k} \neq \sigma_{n} - \lambda^{n+1}(k+n)$$

Natomiast na mocy definicji funkcji δ uzyskujemy

$$(8) \quad \delta_{n}^{k} = \sigma_{n} - \lambda^{n+1}(k+n)$$

Co jest sprzeczne z (7).

Twierdzenie 5

Jeżeli zbiór zadań jest m - regularny, to zachodzą następujące związki:

1. $p(\delta_1^k) = k$ oraz $\delta_1^k = \sum_1^k \lambda^2(j)$

2. $\lambda^i(k+j) \geq 0$ dla $1 \leq i,j \leq m$

Dowód

Bezpośrednio z definicji funkcji δ wynika, że

(1) $\delta_1^k = \sum_1^k \lambda^2(j)$

Pokażemy teraz przez sprowadzenie do sprzeczności, że zachodzi relacja $p(\delta_1^k) = k$. Załóżmy przeciwnie, tzn

(2) $p(\delta_1^k) \neq k$

Z (2) na mocy twierdzenia 3 wynika, że wystarczy rozważyć następujący przypadek

(3) $p(\delta_1^k) < k+1$

Niech $p(\delta_1^k) = 1$, to wówczas na mocy definicji funkcji p i λ uzyskujemy

(4) $\lambda^2(i) \geq \delta_1^k$ dla $1 \leq i \leq m$

Co na mocy (1) daje sprzeczność.

Niech $p(\delta_1^k) = \alpha$ dla $1 < \alpha < k+1$, to wówczas na mocy definicji funkcji p i λ uzyskujemy

(5) $\begin{cases} \lambda^2(i) = 0 & \text{dla } 1 \leq i < \alpha \\ \lambda^2(i) \geq \delta_1^k & \text{dla } \alpha \leq i \leq k+m \end{cases}$

Co na mocy (1) daje sprzeczność.

Przystąpimy teraz do dowodu drugiej części twierdzenia. Na mocy twierdzenia 3 uzyskujemy

(6) $p(\delta_i^k) < k+i$

Rozważmy przypadek gdy $\mu(j) = 0$ dla $i \leq j \leq m$. W tym przypadku na mocy definicji funkcji λ uzyskujemy

(7) $\lambda^{i+1}(n) \geq \delta_i^k$ dla $1 \leq n \leq k+m$

Rozważmy przypadek gdy $\mu(j_o) \neq 0$ dla $1 < j_o < i$. Niech j_o będzie najmniejszą taką liczbą, że $\mu(j_o) \neq 0$. Jeżeli $j_o \leq i$ jest najmniejszą taką liczbą, że $\mu(j_o) \neq 0$, to na mocy definicji λ uzyskujemy

(8) $\lambda^{i+1}(j) \geq \delta_i^k$ dla $\mu(j_o) \leq j \leq m+k$

Na mocy definicji funkcji μ z zależności (6) uzyskujemy, że $\mu(J_o) \leq k$, co na mocy (8) i (7) oraz twierdzenia 2 ostatecznie daje

(9) $\lambda^i (k+J) \geq 0$ dla $1 \leq i,j \leq m$

4. KONSTRUKCJA ALGORYTMU

Wprowadzone w poprzednim rozdziale pojęcia stanowią podstawę teoretyczną działania i konstrukcji algorytmu. Są one jednakże niewystarczające do pełnego opisu algorytmu. Zajmiemy się teraz wprowadzeniem aparatu formalnego pozwalającego precyzyjnie opisać algorytm.

Załóżmy, że mamy dany m-regularny zbiór zadań $Z = \{z_1, z_2, \ldots, z_n\}$. Niech U' będzie dowolnym zbiorem oraz $U = U' \cup Z$. Zbiór U' interpretujemy jako zbiór wszystkich możliwych podziałów zadań ze zbioru Z.

Niech $t = U \to R^+$ (R^+ = zbiór liczb rzeczywistych dodatnich) będzie taką funkcją, że $t(\Lambda) = 0$, gdzie Λ oznacza element pusty zbioru U.

Wartość funkcji $t(x)$ interpretujemy jako czas potrzebny do wykonania zadania x.

Niech $\oplus$ będzie operacją złożenia określoną na zbiorze U, zdefiniowaną w sposób następujący:

1. jeżeli $x,y \in U$ to $x \oplus y \in U$
2. $x \oplus \Lambda = x$ oraz $\Lambda \oplus x = x$
3. $x \oplus y \neq y \oplus x$
4. $(x \oplus y) \oplus z = x \oplus (y \oplus z)$
5. $t(x \oplus y) = t(x) + t(y)$

Jeżeli $z = x \oplus y$, to w zależności od sytuacji, powiemy, że zadanie Z składa się z segmentów x i y, lub, że zadanie z zostało podzielone na segmenty x i y. Segment x jest pierwszym segmentem, a segment y drugim segmentem. Wykonanie zadania Z można interpretować jako wykonanie segmentów x i y kolejno po sobie.

Niech X i Y będą dowolnymi podzbiorami zbioru U. Zbiór Y nazywamy bezpośrednim rozszerzeniem zbioru X wtedy i tylko wtedy, jeżeli istnieją takie dwa segmenty $x,y \in U$, że $x \oplus y \in X$ oraz $Y = X \cup \{x,y\} - \{x \oplus y\}$.

Niech X będzie dowolnym podzbiorem zbioru U. Zbiór J(X) nazywamy zbiorem indukowanym przez zbiór X ze względu na operację $\oplus$ wtedy i tylko wtedy gdy:

i) $\ldots \cap X \subset J(X)$

ii) jeżeli $x_1, \ldots, x_k \in X - Z$ oraz $x_1 \oplus x_2 \oplus \ldots \oplus x_k \in Z$ to $x_1 \oplus x_2 \oplus \ldots \oplus x_k \in J(X)$

iii) jeżeli $x \notin X - Z$ oraz nie istnieją taki $y_1, \ldots, y_k \in X - Z$, że $x \in \{y_1, \ldots, y_k\}$ oraz $y_1 \oplus \ldots \oplus y_k \in Z$ to $x \in J(X)$

Zbiór J(X) interpretujemy w sposób następujący: jeżeli w zbiorze X są zawarte wszystkie segmenty pewnego zadania, to w zbiorze J(X) jest zawarte to zadanie, natomiast w zbiorze J(X) nie są zawarte segmenty tego zadania.

Opierając się na wprowadzonym aparacie formalnym, przystąpimy do opisu procesu tworzenia zbiorów zadań, które będą wykonywane tylko przez jeden z pro-

cesorów $M_1,\ldots,M_m$. Zasadniczą rolę przy tworzeniu zbiorów zadań będą odgrywały pewne podzbiory P_j^* i $Q(i,j)$ zbioru U.

Załóżmy, że mamy dany podział m-regularny $P = \{P_o, P_1, .., P_m\}$ zbioru zadań Z. Niech $\|P_o\| = k$ oraz $P_o = \{x_1,\ldots,x_k\}$ będzie zbiorem uporządkowanym ze względu na następującą relację:
jeżeli $1 \leqslant i < j \leqslant k$ to $t(x_i) \leqslant t(x_j)$.

Niech $P(Z) = \{P(1),..,P(i),..,P(m+k)\}$ będzie rozłącznym podziałem zbioru Z zdefiniowanym w sposób następujący:

$$P(i) = \begin{cases} \{x_i\} & \text{dla } i \leqslant k \\ P_{i-k} & \text{dla } i > k \end{cases}$$

Opiszemy teraz dla $j = 1,2...,m$ i $n = 1,2,\ldots,m+k$ proces tworzenia zbiorów P_j^* i $Q(j,n)$. Proces ten będziemy nazywali algorytmem podziału.

Definicja 6. Algorytm podziału

Niech $n_j = p(\sigma_{j+1}^k) - j$. Algorytm podziału definiujemy indukcyjnie w sposób następujący:

1. Krok podstawowy

$$P_m^* = P(m+k) \cup D(m,m)$$

$$Q(m,n) = E_{m,n}(P(n)) - D(m,n) \cap h(m,m+1-n)$$

2. Krok indukcyjny

$$P_j^* = Q(j+1,k+1) \cup J(\bigcup_{i=j}^{m} D(i,p(\sigma_i^k) + i - m))$$

i) jeżeli $\lambda^{j+1}(n_j) \geqslant \sigma_j^k$ to

$$Q(j,n) = E_{j,n}(Q(j+1,n)) - D(j,n) \cap h(j,p(\sigma_j^k) + 1 - n)$$

ii) jeżeli $\lambda^{j+1}(n_j) < \sigma_j^k$ to

$$Q(j,n) = \begin{cases} \emptyset & \text{jeżeli } n < \mu < j \\ E_{j,n}\left(\bigcup_{\alpha=0}^{q(j,n)} Q(j+1, n_j + r(j,n) + \alpha \cdot j)\right) - D(j,n) \cap h(j,p(\sigma_j^k) + 1 - n)) & \\ & \text{jeżeli } \mu(j) \leqslant n < \mu(j) + j \\ Q(j+1,n) & \text{jeżeli } \mu(j) + j \leqslant n \leqslant m + k \end{cases}$$

gdzie h jest funkcją zdefiniowaną w sposób następujący:

$$h(j,x) = \begin{cases} U & \text{jeżeli } 1 \leqslant x \leqslant j \\ \emptyset & \text{pozostałych przypadkach} \end{cases}$$

a $E_{j,n}(X)$ jest takim bezpośrednim rozszerzeniem zbioru $X \subset U$, dla którego istnieje taki podzbiór $D(j,n) \subset E_{j,n}(X)$ że $t(D(j,n)) = \sigma_j^k$. W przypadku gdy

$X = \emptyset$ to przyjmiemy, że $B_{j,n}(X) = \emptyset$.

Natomiast, podobnie jak poprzednio, $q(j,n)$ jest częścią całkowitą z dzielenia $(n - n_j)$ przez j, a $r(j,n)$ resztą z dzielenia $(n - n_j)$ przez j.

Bezpośrednio z definicji algorytmu podziału oraz twierdzeń 1,2,3,4,5 wynika następujące twierdzenie:

Twierdzenie 6

Jeżeli zbiór zadań jest m-regularny to dla $i,j = 1,2,\ldots,m$, $t(P_j^*) + 6$ oraz jeżeli $i \neq j$ to $P_i^* \cap P_j^* = \emptyset$.

Z twierdzenia 6 wynika, że jeżeli każdemu zbiorowi P_j przyporządkujemy tylko jeden procesor oraz jeżeli zadania ze zbioru $J(\bigcup_{i=j}^{m} D(i,p(\delta_i^k)+i-m))$ są wykonywane przed zadaniami ze zbioru $Q(j+1,k+j)$, to wówczas obliczenie złożone z zadań ze zbioru Z zostanie wykonane poprawnie i w optymalnym czasie oraz liczba podziałów zadań będzie mniejsza od m. Co więcej, zadania ze zbiorów $Q(j+1,k+j)$ i $J(\bigcup_{i=j}^{m} D(i,p(\delta_i^k)+i-m))$ mogą być wykonywane w dowolnym porządku.

Literatura

[1] Mc NAUGHTON R.: Scheduling with Deadlines and Loss Functions, Management Science, 1959, t.6, nr 1, s. 1-12.

[2] BRUNO J., COFFMAN E.G., Jr., SETHI R.: Algorithms for Minimizing Mean Flow Time.-Mathematical Aspects of Information Processing, IFIP Congress 74, Sztokholm, 1974, s. 504-510.

Streszczenia

Содержания

Contents

СИСТЕМА АВТОМАТИЧЕСКОЙ КОНСТРУКЦИИ ОПТИМАЛИЗАТОРОВ ДЛЯ НЕКОТОРОГО КЛАССА ПРОМЕЖУТОЧНЫХ ЯЗЫКОВ

С.Яжомбек

В статье описана Оптимизирующая Система, которая генерирует оптимизаторы для некоторого класса промежуточных языков, выступающих в трансляторах. Приведены также некоторые теоретические результаты, связанные с реализацией этой системы. Оптимализирующая Система состоит из двух частей: Конструктора и Базисного Оптимализатора. Идея работы Системы следующая: для получения оптимализатора некоторого промежуточного языка L необходимо составить соответствующие описание этого языка; это описание вводится в конструктор, который на его основе модифицирует Базисный Оптимализатор, в результате чего получается оптимализатор языка L.

AUTOMATIC CONSTRUCTION OF OPTIMIZERS FOR A CERTAIN CLASS OF INTERMEDIATE LANGUAGES

S. Jarząbek

In this paper an Optimization System is described which automatically generates optimizers for a certain class of compiler intermediate languages. Some theoretical problems connected with optimizer construction are discussed as well. The Optimization System consists of two parts: the Constructor and the Basic Optimizer. The idea of operation is as follows: in order to obtain an optimizer for a particular intermediate language a special description of that language must be input to the Con-

structor. The Constructor's task is to modify the Basic Optimizer program in such a way that an optimizer is created for programs written in the particular intermediate language.

МАШИННАЯ РЕАЛИЗАЦИЯ АЛГОРИТМА ПЛАНИРОВАНИЯ ВЫЧИСЛЕНИЙ ДЛЯ ДВУХПРОЦЕССОРНЫХ СИСТЕМ

А. Ровицки

В работе описана машинная реализация алгоритма планирования вычислений для двухпроцессорных систем и приведены результаты работы этого алгоритма. Этот алгоритм был запрограммирован на языке ПАСКАЛЬ для машины ОДРА 1304.

COMPUTER REALIZATION OF SCHEDULING ALGORITHM FOR TWO-PROCESSOR SYSTEMS

A. Rowicki

The paper describes the computer realization of scheduling algorithm for two-processor systems and presents the results of acting of this algorithm. This algorithm have been programmed in PASCAL for ODRA 1304 computer.

ПЛАНИРОВАНИЕ ЗАВИСИМЫХ РАСПРЕДЕЛЕННЫХ ВЫЧИСЛЕНИЙ ДЛЯ ДВУХПРОЦЕССОРНЫХ СИСТЕМ

А. Ровицки

В работе приведён алгоритм планирования вычислений для двухпроцессорных систем при предположении, что задачи разделяются любым способом, структура вычислений является графом, не содержащим петлю и имеющим только одну конечную вершину. Показано,

что этот алгоритм является оптимальным, м.е. обеспечивает оптимальное время выполнения вычисления.

PREEMPTIVE SCHEDULING OF DEPENDENT TASKS FOR TWO-PROCESSOR SYSTEMS

A. Rowicki

In this paper we give an algorithm for preemptive scheduling for two-processor systems provided that preemption times are completely unconstrained and computations submitted to the system contain no loops and have one output. It has been proved that this algorithm is optimal, that is, the execution time is minimized.

ПЛАНИРОВАНИЕ НЕЗАВИСИМЫХ РАСПРЕДЕЛЁННЫХ ВЫЧИСЛЕНИЙ ДЛЯ МНОГОПРОЦЕССОРНЫХ ОДНОРОДНЫХ СИСТЕМ

А. Ровицки

В работе приведён для многопроцессорных систем оптимальный ан алгоритм планирования вычислений, состоящих из независимых задач с любым временем выполнения при предположении, что задачи разделяются любым способом. Были исследованы его основные свойства и приведены необходимые и достаточные условия существования алгоритму. Рассматриваемый алгоритм обеспечивает число распределений, меньше, чем число процессоров.

PREEMPTIVE SCHEDULING OF INDEPENDENT TASKS FOR MULTIPROCESSOR SYSTEMS

A. Rowicki

In this paper we give an algorithm for optimal preemptive scheduling of independent computations for multiprocessor sys-

tems provided that preemption times are completely unconstrained. In the paper has been considered basic properties of the algorithm and a necessary and sufficient condition for existence of this algorithm. The algorithm considered gives the number of preemptions smaller than the number of processors needed.

Wyk.Sp."Tłumacz-Tekst",Al.Ujazd.26
U-101/A4/450/II- 03.80r.- TO

Druk okładki IMM zam. 57/80 n. 450.