PORADNIKI

Reverse Engineering od asemblera do formalnej specyfikacji przez transformacje programu

Streszczenie

System transformacji FermaT, oparty na badaniach prowadzonych przez ostatnie 16 lat na Uniwersytecie Durham ,Uniwersytet De Monfort i Software Migrations Ltd, jest przemysłową formalną transformacją z wieloma aplikacjami w rozumieniu programu i migracji językowej. Tu pokażemy studium przypadku, który używa zautomatyzowanych i ręcznych transformacji i abstrakcji dla konwersji kodu asemblerowego programu IBM 370 na specyfikację bardzo wysokopoziomej abstrakcji.

Słowa kluczowe: Assembler, Migracja, Zrozumienie, Metody formalne ,Abstrakcja, WSL, Szerokie Spektrum Języka, Transformacja Programu, System Legacy, Restrukturyzacja

Wprowadzenie

Jest ogromny zbiór systemów operacyjnych oprogramowania, które mają istotne znaczenie dla ich użytkowników, ale stają się coraz trudniejsze do utrzymania, wzocnienia i bycia na bieżąco z szybko zmieniającymi się wymaganiami. Dla wielu z tak zwanych klasycznych systemów istnieje możliwość odrzucenia systemu i ponowne napisanie od początku jest nieopłacalne ekonomicznie. Zatem pilną potrzebą dla tych stają się metody, które pozwalają rozwijać się w sposób kontrolowany.W szczególności klasyczne systemy asemblerowe mają wysokie koszty utrzymania, a migracja takich systemów do innych środowisk (np architektury klient-serwer) jest dużo trudniejsze niż do systemów języków wysokopoziomowych. System transformacji FermaT używa formalnie sprawdzonych transformacji programu, które chronią lub udoskonalają sematykę programu podczas zmiany jego formy. Te transformacje są stosowane do restrukturyzacji i uproszczenia klasycznych systemów i wyodrębniania wysokopoziomowej reprezentacji. Przez użycie właściwej sekwencji transformacji,wyodrębniona reprezentacja gwarantuje zrównoważenie logiki oryginalnego kodu. Metoda jest oparta na formalnym szerokimspektrum języka, zwanego WSL, z towarzyszeniem metody formalnej. Przez 16 lat zaprojektowano duży katalog sprawdzonych transformacji, razem z mechanicznym sprawdzaniem warunków stosowania. Ma to zastosowanie dla wielu projektów programistycznych, reverse engineeringu i zarządzania problemami.

Teoretyczne podstawy

Teoretyczna praca na której opiera się FeramT nie jest konserwacja oprogramowania ale na badaniach nad rozwojem języka w którym dowody równoważności dla transformacji programu mogą być osiągnięte taka łatwo jak to możliwe dla szerokiej gamy konstrukcji. WSL jest "Szerokim Spektrum Języka" jest używanym w naszych transformacjach programu, który obejmuje konstrukcje programistyczne niskopoziomowe i abstrakcyjne specyfikacje wysokopoziomowe wewnątrz pojedynczego języka. Ma to tę zaletę ,że nie ma konieczności rozróżniania między programowaniem i specyfikacją języka: cały rozwój transformacji programu od

specyfikacji abstrakcyjnej do szczegółowej implementacji może być zawarty w pojedynczym języku. Odwrotnie,cały proces reverse engineeringu , od transliteracji programu źródłowego do specyfikacji wysokopoziomowej, może być również zawarta w tym samym języku. Podczas tego pocesu, różne części programu mogą być wyrażone przez różne poziomy abstrakcji. Więc foram WSL jest idealnym narzędziem dla rozwoju metod dla rozwoju formalnego programu jak również formalnego reverse engineeringu (dla którego użyjemy terminu odwrócona inżynieria). Transformacja programu jest działaniem, które modyfikuje program na różne formy , które mają to samo zewnętrzne zachowanie (tj jest odpowiednikiem precyzyjnie zdefiniowanej semantyki denotacyjnej) Ponieważ zarówno programy i specyfikacje są częścią tego samego języka, transformacje mogą być używane do zademonstrowania ,ze dany program jest poprawnie zaimplementowany przy danej specyfikacji. Udoskonalenie jest działaniem , które modyfikuje program aby uczynić jego zachowanie bardziej określonym i/lub bardziej deterministycznym.Typowa implementacja niedeterministycznej specyfikacji będzie udoskonalona niż strikte równoważna. Przeciwieństwem udoskonalenia jest abstrakcja: mówimy ,że specyfikacja jesy abstrakcją programu, który ją implementuje. Większość konstrukcji w WSL, na przykład instrukcje if, pętle while, procedury i funkcje, są powszwechne w wielu językach programowania. Jednak są pewne funkcje powiązane z "poziomem specyfikacji" języka , które są niespotykane. Wyrażenia i warunki (formuły) w WSL są pobierane z logiki pierwszego stopnia: faktycznie, używana jest inifinitarna logika pierwszego stopnia, co pozwala na przeliczanie nieskończonych dysjunkcji i koniunkcji. To zastosowanie logiki pierwszego rzędu oznacza ,że instrukcje w WSL mogą objąć egzystencjalną i uniwersalną kwantyfikację zbiorów nieskończonych i podbne (niewykonywalne) operacje. Dwie listy operatorów również są używane w specyfikacji: dla funkcji jednoargumentowej f i listy L = $\langle a_1,...a_n\rangle$ odwzorowujemy operator * zdefiniowany :

$$f * L =_{DF} \langle f(a_1), f(a_2), \ldots, f(a_n)\rangle$$

Dla operatora binarnego g i nie pustej listy L zdefiniowany jest operator reduce / :

$$\begin{aligned} g/L &=_{DF} a_1 \qquad \text{if } n = 1 \\ &=_{DF} g(a_1, g/\langle a_2, \ldots, a_n\rangle) \quad \text{if } \ell(L) > 1 \end{aligned}$$

Na przykład ,jeśli fjest funkcją która zwraca liczby całkowite, a L jest nie pustą listą odpowiednich argumentów dla f, wtedy + / f * L jest wynikiem stosowania f dla każdego elementu L i podliczania wyników. Użyjemy również l(L) dla oznaczenia długości listy L i L[i..j] dla oznaczenia podlisty $\langle a_i,...,a_j\rangle$. Przez ostatnie 16 lat został stworzony język WSL, równolegle z projektowaniem teorii transformacji i metod dowodów. Przez ten czas język ewoluował od prostego i łatwego jądra języka do pełnego

i silnego języka programowania. Na "niskopoziomowym" końcu języka istnieje automatyczny tłumacz z Asemblera IBM na WSL, i z podzbioru WSL na C. Na "wysokopoziomowym" końcu, jest możliwe zapisanie abstrakcyjnej specyfikacji, podobnie do Z w VDM. Transformacje programów są używane do czerpania z różnych wydajnych algorytmów z abstrakcyjnej specyfikacji. Te same transformacje są używane w odwrotnym kierunku: używając transformacji wyprowadzamy zwięzłą abstrakcyjną reprezentację specyfikacji dla kilku wymaganych programów. Opisano również użycie FermaT dla migracji z programu asemblera do modularnego i zarządzalnego kodu C, używając czystej automatycznej transformacji bez żadnych ręcznych interwencji.O ile wiadomo, żaden z innych badaczy transformacji programów nie próbował stosować swoich metod na kodzie asemblerowym. Najbliżej była Cristina Cifuentes pracująca przy dekompilacji i translacji binarnej . Tu zajmiemy się procesem reverse engineeringu. Zaczniemy od kodu asemblerowego, użyjemy formalnych transformacji do abstrakcji równoważnej specyfikacji wysokopoziomowej programu

Przykład transformacji w FermaT

W tej części opiszę małą liczbę transformacji implementowanych w FermaT, które są użyte w tym studium przypadku. Jeśli S_1 i S_2 są dowolnymi instrukcjami WSL a Δ jest dowolną formułą zbioru przeliczalnego bez wolnych zmiennych, wtedy zapiszemy $\Delta \models S_1 \leq S_2$ oznaczając ,że S_2 jest udoskonaleniem S_1 ilekroć wszystkie formuły w Δ są prawdą .Jeśli $\Delta \models S_1 \leq S_2$ i $\Delta \models S_2 \leq S_1$ wtedy zapiszemy $\Delta \models S_1 \approx S_2$ i powiemy ,że S_1 i S_2 są odpowiednikami. Jeśli S_2 jest generowane z S_1 przez transformację, wtedy $\Delta \models S_1 \approx S_2$ gdzie Δ jest zbiorem warunków stosowania dla tej transformacji

Rozwinięcie w przód

Jeśli B jest dowolnym warunkiem a S_1, S_2 i S_3 są dowolnymi instrukcjami wtedy:

$$\Delta \models \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi; } S_3 \approx \text{if } B \text{ then } S_1;S_3 \text{ else } S_2;S_3 \text{ fi}$$

Pętle

Jak zwykle przy pętlach for i while istnieje zapis dla pętli nieograniczonej. Instrukcje tej formy do S od, gdzie S jest instrukcją, są "nieskończonymi" lub "nieograniczonymi" pętlami, które mogą być tylko zakończone przez wykonanie instrukcji w postaci exit(n) co powoduje ,że program się kończy po n pętlach. Użyjemy exit jako skrót dla exit(1). Uprościmy język odrzucając exit, które opuszcza blok lub pętlę inną niż pętla nieograniczona. Będziemy również naciskać aby n było liczbą całkowitą , a nie zmienną lub wyrażenie – to zapewnia ,że zawsze możemy określić cel exit.
Definicja 1 Globalen zastępowanie
Jeśli P(S,p) jest predykatem w instrukcji S i pozycją p wewnątrz

S,a S'(S,p) jest funkcją która zwraca instrukcję dla danej instrukcji S i p, wtedy wpływ zastępowania lub dodawania do instrukcji na pozycji p w S z S'(S,p) dla każdego p takiego ,że P(S,p) jest oznaczone

$$\mathbf{S}\left[\,\mathbf{S}'(\mathbf{S},p)\,/\,p \;\middle|\; \mathbf{P}(\mathbf{S},p)\,\right]$$

Jeśli instrukcja na pozycji p w S jest instrukcją exit, wtedy jest zastępowana przez S'(S,p). W przeciwnym razie, S'(S,p) jest dołączane w sekwencji po instrukcji na pozycji p. Wewnątrz globalnego zastępowania użyjemy δ(S,p) oznacza głębokość elementu instrukcji. Jest to liczba dołączonych pętli do ... od otaczających ten element. Użyjemy τ(S,p) do oznaczenia wartości końcowej instrukcji. Jest to liczba dołączanych pętli wokół S które mogą być zakończone wykonaniem instrukcji na pozycji p w S. Jeśli instrukcja na pozycji p w S nie kończy S wtedy τ(S,p) = -1/ Na przykład, dowolne exit (n) ma wartość końcową n. Jeśli S zawiera exit (n) wewnątrz m zagnieżdżonych pętli (gdzie m ≤ n) wtedy wartość końcowa samej S, oznaczona τ(S,<>), będzie to przynajmniej n – m. Instrukcja S z wartością końcową zero nie może zakończyć dołączonych pętli, więc kolejna rzecz wykonywana po S będzie to kolejna instrukcja w sekwencji zawierającej S (jeśli jest jedynką). Taka instrukcja nazywana jest właściwą sekwencją. Jeśli S jest właściwą sekwencją wtedy

$$\Delta \vdash \textbf{do if } \mathbf{B} \textbf{ then exit fi; } \mathbf{S} \textbf{ od} \;\approx\; \textbf{while } \neg\mathbf{B} \textbf{ do } \mathbf{S} \textbf{ od}$$

W poniższej transformacji, globalne zastępowania są stosowane do prostej końcowej instrukcji S. To są instrukcje które są albo sekwencją, trybem warunkowym albo pętlą do... od i które będą się kończyć S jeśli są wykonywane. Na przykład w

```
if B then x := 1; y := 2 else exit fi
```

końcowe instrukcje to y := 2 i exit. Jeśli instrukcja jest dołączona w pętli do .. od, tylko exit będzie instrukcją końcową. Zwykle pomijamy parametry z δ i τ w globalnym zastępowaniu kiedy wynikają w sposób oczywisty z kontekstu

Definicja 2 Zwiększanie
Zwiększanie S o n (gdzie n jest dowolną nieujemną liczbą całkowitą) jest definiowane jako zwiększanie wszystkich prostych instrukcji końcowych w S. exit jest zwiększone przez zwiększanie jej parametrów, podczas gdy dowolne inne proste instrukcje są zwiększane przez dołączenie exit:

$$\mathbf{S} + n \;=_{\mathrm{DF}}\; \mathbf{S}\left[\mathbf{exit}(n+\delta)\,/\,p \,\middle|\, \tau \geqslant 0\right]$$

Na przykład:

if B then x:= 1; y:=2 else exit fi+2

= if B then x:=1; y:=2; exit(2) else exit(3) fi

podczas gdy:

do if B then x:= 1; y:= 2 else exit fi od +2
= do if B then x:=1; y:=2 else exit(3) fi od

Definicja 3 Częściowe zwiększanie
Notacja S + (n,m) gdzie m ≥ 0 oznacza zwiększenie instrukcji końcowej w S z wartością końcową m lub większą:

$$\mathbf{S} + (n, m) \;=_{\mathrm{DF}}\; \mathbf{S}\,[\mathbf{exit}(n + \delta)\,/\,p \mid \tau \geqslant m]$$

Zauważ ,że do S od + (n,m) =do S + (n,m+1) od

Absorpcja

Dla dowolnych instrukcji S_1 i S_2 :

$$\Delta \vdash \mathbf{S}_1;\, \mathbf{S}_2 \;\approx\; \mathbf{S}_1\,[\mathbf{S}_2 + \delta\,/\,p \mid \tau = 0]$$

Na przykład:

do if B then x:= 1; y:= 2 else exit fi od; z :=1
do if B then x:=1; y:=2 else z := 1;exit fi od

Ta transformacja może być stosowana na odwrót do "wyjmowania" kodu z pętli.

Fałszywa pętla

Możemy wstawić pętlę wokół dowolnej instrukcji, przez zwiększenie :

$$\Delta \vdash \mathbf{S} \;\approx\; \mathbf{do}\ \mathbf{S} + 1\ \mathbf{od}$$

(Jest to "fałszywa pętla" ponieważ ciało pętli może być wykonana tylko raz)

Pętla podwojona

Dowolna pętla może być skonwertowana do pętli podwójnej przez ostatnią transformację, lub zwiększenie ciała pętli:

$$\begin{aligned}\Delta \vdash \mathbf{do}\ \mathbf{S}\ \mathbf{od} \;&\approx\; \mathbf{do}\ \mathbf{do}\ \mathbf{S}\ \mathbf{od} + 1\ \mathbf{od}\\ &\approx\; \mathbf{do}\ \mathbf{do}\ \mathbf{S} + 1\ \mathbf{od}\ \mathbf{od}\end{aligned}$$

Generalnie możemy arbitralnie zdecydować czy lub nie zwiększać każdą końcową instrukcję w S z wartością końcową zero:

$$\Delta \vdash \textbf{do S od} \approx$$
$$\textbf{do do S}\,[\,\textbf{exit}(\delta+1)/p \mid \tau > 0 \vee \tau = 0 \wedge \Psi(\mathbf{S},p)\,]\ \textbf{od od}$$

Gdzie Ψ jest dowolnym warunkiem w S i p. Może to być połączone z odwróceniem absorpcji dla "izolowania" części ciała pętli. Na przykład:

$$\Delta \vdash \textbf{do S; if B then } \mathbf{S}_1 \textbf{ else } \mathbf{S}_2 \textbf{ fi od}$$
$$\approx \textbf{do do S} + (1,1);\ \textbf{if B then exit else } \mathbf{S}_2 + (1,1) \textbf{ fi od};\ \mathbf{S}_1 \textbf{ od}$$

Odwrócenie pętli

Jeślo S_1 jest właściwą sekwencją wtedy

$$\Delta \vdash \textbf{do } \mathbf{S}_1;\ \mathbf{S}_2 \textbf{ od} \approx \mathbf{S}_1;\ \textbf{do } \mathbf{S}_2;\ \mathbf{S}_1 \textbf{ od}$$

Generalnie, dla dowolnych instrukcji S_1 i S_2 :

$$\Delta \vdash \textbf{do } \mathbf{S}_1;\ \mathbf{S}_2 \textbf{ od} \approx \textbf{do } \mathbf{S}_1;\ \textbf{do } \mathbf{S}_2;\ \mathbf{S}_1 \textbf{ od} + 1 \textbf{ od}$$

Rozwijanie pętli

Możemy rozwinąć pierwszy krok pętli:

$$\Delta \vdash \textbf{do S od} \approx \mathbf{S}[\textbf{do S od} + \delta + 1 / p \mid \tau = 0]\ [\textbf{exit}(\tau + \delta - 1)/p \mid \tau \geqslant 1]$$

gdzie RHS zawiera dwa kolejne globalne zastąpienia w S. Generalnie możemy wstawić kopię całej pętli, z pewnymi zwiększonymi instrukcjami końcowymi ciała pętli, po pewnych instrukcjach końcowych w ciele pętli. Niech S' będzie sformowane z S przez zwiększenie wybranych instrukcji końcowych z wartością końcową zero:

$$\mathbf{S}' = \mathbf{S}\,[\textbf{exit}(\delta+1)/p \mid \tau = 0 \wedge \Phi(\mathbf{S},p)\,]$$

gdzie Φ jest dowolnym warunkiem. Wtedy:

$$\Delta \vdash \textbf{do S od}$$
$$\approx \textbf{do S}\,[\textbf{do S}' \textbf{ od} + \delta + 1/p \mid \tau = 0 \wedge \Psi(\mathbf{S},p)\,]\ [\textbf{exit}(\tau+\delta-1)/p \mid \tau \geqslant 1]\ \textbf{od}$$

gdzie Ψ jest dowolnym warunkiem

Modelowanie asemblera w WSL

Budowanie użytecznych modeli naukowych koniecznie wymaga odrzucenia pewnych informacji: innymi słowy, użyteczny model musi być niedokładny lub przynajmniej wyidealizowany do pewnego

zakresu.Na przykład "gazy idealne", "nieściśliwe płyny" czy "cząsteczki piłki bilardowej" są wszystkie użytecznymi modelami które czerpią swoją przydatność poprzez abstrahowanie od kilku szczegółów w prawdziwym świecie. W przypadku modelowania języka programowania, takiego jak Asembler, jest teoretycznie możliwe posiadać doskonały model tego języka który poprawnie przechwytuje zachowania wszystkich programów asemblerowych. Pewne funkcje Asemblera, takie jak rozgałęzienia do adresów rejestrów, samomodyfikujący kod itd, oznaczałyby ,że taki model zapisywałby cały stan komputera wliczając w to wszystkie rejestry, pamięć, przestrzeń dyskową i zewnętrzne urządzenia i "interpretował" ten stan jako każdą instrukcję do wykonania (Rozważmy wpływ ładowania jakichś danych z pliku dyskowego do pamięci, w operacjach arytmetycznych w dowolnym miejscu w danych a potem rozgałęzienie do początku bloku danych!) Niestety taki model jest nieprzydatny dla reverse engineeringu czy też celów migracyjnych. To co konieczne to praktyczny model dla programów asemblerowych, który jest odpowiedni dla reverse engineeringu, i jest dość wystarczający dla działania ze wszystkimi konstrukcjami programistycznymi które prawdopodobnie napotkamy.

Asembler i tłumaczenie na WSL

Celem tłumacza asemblera na WSL jest wygenrowanie kodu WSL, któego modele jak nakdokładniej to możliwe zachowują oryginalny moduł asemblera, nie martwiąc się zbytnio o rozmiar, wydajność i złożoność kodu wynikowego .Zazwyczaj, surowa translacja WSL modułu asemblerowego będzie trzy do pięciu razy większa niż plik źródłowy i ma bardzo wysoką złożoność cyklomatyczną McCabe'a (zwykle w setkach, często w tysiącach). Jest tak, częściowo, ponieważ każde "rozgałęzienie do rejestru" instrukcji rozgałęzienia wysyła do podprogramu, który z kolei zawiera rozgałęzienia do każdego możliwego punktu powrotu. Dodatkowo każda instrukcja która ustawia flagę "warunku kodu" będzie tłumaczona na kod WSL, który przypisuje właściwą wartość do specjalnej zmiennej cc (do emulacji kodu warunkowego): czy lub nie kod warunkowy jest potem testowany. Jednak silnik transformacji FermaT zawiera pewne bardzo silne transformacje dla uproszczenia kodu WSL, usuwając redundacje,śledząc wysyłany kod itd. W większości przypadków FermaT może automatycznie rozszyfrować zawiły kod "rozgałęzienia do rejestru" i "rozgałęzienie i zapis" dla wyodrębnienia niezależnych, jednowejściowych jednowyjściowych procedur i wyeliminować procedurę wysyłania. Dodatkowo FermaT może prawie zawsze wyeliminować zmienną cc przez skonstruowanie właściwych instrukcji warunkowych.

Próbka programu

```
proc Management_Report ≡
  var SW1 := 0, SW2 := 0 :
    Produce_Heading;
    read(stuff);
    while NOT eof(stuff) do
      if First_Record_In_Group
        then if SW1 = 1
                then Process_End_Of_Previous_Group
             fi;
             SW1 := 1;
             Process_Start_Of_New_Group;


             Process_Record;
             SW2 := 1
        else
          Process_Record; SW2 := 1
      fi;
      read(stuff)
    od;
    if SW2 = 1 then Process_End_Of_Last_Group
    fi;
    Produce_Summary
  end.
```

Ten program jest prostym generatorem raportów, który odczytuje posortowane pliki transakcyje: każda transakcja zawiera nazwę pozycji i ilość odebraną bądź dystrybuowaną z magazynu. Program generuje raport pokazujący zmiany netto w stanie zapasów dla każdej pozycji w tym pliku transakcyjnym. Kod asemblerowy tego pseudo kodu został podany na końcu tego tekstu. Program zawiera kod samomodyfikujący: "pierwszy raz przez przełączanie" SW1 jest implementowane przez modyfikację gałęzi oznaczonej LAAA do NOP w istrukcji oznaczonej LAB i instrukcji Execute użytej do pobrania długości zmiennej

Automatyczna transformacja programu

Pierwszym etapem w procesie transformacji jest Tłumaczenie Danych. Ta transformacja używa restrukturyzowanego pliku danych do zmiany danych przedstawianych w programie. Początkowo wszystkie dane są dostępne z pamięci (przedstawiane jako tablica bajtów a) przez dodanie rejestru bazowego do przemieszczenia dla uzyskania adresu. Restrukturyzowany plik danych podaje rozkład wszystkich danych w pamięci, tak więc dokonując rozsądnych założeń o nie zachodzeniu DSECTS itd, FermaT może transformować program do odpowiedniego

programu gdzie dane są dostępne bezpośrednio przez zmienne i struktury. Na przykład rozważmy instrukcję "raw WSL":

$$!P\ \mathsf{mvc}(\mathsf{a}[\mathsf{db}(\mathsf{writem}, \mathsf{r3}), 3+1]\ \mathbf{var}\ \mathsf{a}[\mathsf{db}(\mathsf{wlast}, \mathsf{r3}), 3+1]);$$

Tu !P wskzuje zewnętrzną procedurę wywołującą procedurę mvc, któa implementuje instrukcję MVC (przenieś znaki). To przeniesienie podaje liczbę znaków z danego adresu źródłowego do danego adresu przeznaczenia. Funkcja db(x,y) po prostu zwraca x+y, przemieszczenie plus rejestr bazowy, więc adres źródłowy jest wrtitem +r3 a adres przeznaczenia to wlast +r3. Po przetłumaczeniu danych, te same nazwy sa używane jako rzeczywiste zmienne a rejestry bazowe są eliminowane. Ta instrukcja jest automatycznie transformowana na proste przypisanie:

wlast := wrec.writem;
W przypadku naszego prostego programu, jest tylko jedna ukryta struktura: wrec drukuje rekord , który zawiera pola writem, wrtype i wrqty plus pewne nienazwane wypełniacze.
Kolejnym etapem jest sterowanie przepływem restrukturyzacji: eliminowanie zbędnych etykiet i rozgałęzień, wprowadzenie pętli. Jest to wykonywane w szeregu przejść przez program, przy każdej iteracji program jest przeszukiwany pod kątem miejsc gdzie uproszczenie trransformacji może być zastosowane. Iteracja jest kontynuowana dopóki nie zostanie osiągnięta dalsza poprawa. Surowy WSL jest zapisany jako system akcji, zbiór bezparametrowych procedur (akcji) gdzie wykonywanie dowolnej akcji zawsze będzie prowadzić do albo wywołania innej akcji albo wywołania akcji specjalnej Z która kończy cały system akcji. Sam system akcji jest prostą instrukcją, więc akcje systemu mogą być zagnieżdżone jedna wewnątrz drugiej, ale system sub akcji nie może wywołać akcji z systemu głównego. Wtedy system analizuje pozostałe akcje określając które akcje mogą sformować ciało prostej procedury. Aby to zrobić używamy kontroli przepływu i analizy przepływu danych. Jeśli określa to ,że zbiór działań formuje procedurę , wtedy te działania są wyodrębniane jako system subakcji w ciele procedury. Po kontroli przepływu restrukturyzacji mamy analizę przepływu danych: w szczególności roszerzona postać propagacji stałych, która może propagować adresy powrotu przez wywołanie procedury. Jeśli napotkamy wywołanie dispatch,ze znaną wartością destination, wtedy może być to rozłożone i uproszczone. Ta sama transformacja działa również z warunkowy przypisaniem do kodu warunkowego (cc) aby usunąć referencje do cc gdy to możliwe. FermaT mógł wyodrębnić zbiór działań dla sformowania procedury endgroup, tak więc kod:

$$\mathsf{r10} := 112;\ \mathbf{call}\ \mathsf{endgroup}$$

staje się

$$\mathsf{r10} := 112;\ \mathsf{endgroup}();\ \mathbf{call}\ \mathsf{dispatch}$$

FermaT określa ,że ta wartość w r10 będzie skopiowana do destination przez ciało endgroup. Wewnątrz dispatch wartość w

destination jest porównywana z offsetami wszystkich możliwych punktów powrotu. Offset 112 jest związany z etykietą lab, więc call dispatch może być zastąpione przez call lab. Sterowanie przepływem i transformacje restrukturyzacji przepływu danych są iterowane dopóki nie będzie możliwa dalsza poprawa.

```
begin
  f_laaa := 1;
  !P open(ddin_ddname, input var os);
  !P open(rdsout_ddname, output var os);
  wprt[1..17] := "MANAGEMENT REPORT";
  write1(); write1();
  wprt[1..20] := "ITEM NET CHANGE";
  write1(); write1();
  xsw1 := 0;
  do r0 := 0; r1 := 0; r15 := 0;
     !P get(ddin_ddname var os, r0, r1, r15, wrec);
     if !XC end_of_file(ddin_ddname)
       then exit(1) fi;
     if wrec.writem ≠ wlast
       then if f_laaa ≠ 1
                then endgroup() fi;
            f_laaa := 0;
            wlast := wrec.writem;
            wnet := !XF zap("hex 0x0C") fi;
     worka := !XF pack(wrec.wrqty, 2);
     if wrec.wrtype ≠ "R"
       then wnet := !XF sp(wnet, worka)
        else wnet := !XF ap(wnet, worka) fi;
     xsw1 := "hex 0xFF" od;
```

```
    if xsw1 = "hex 0xFF" then endgroup() fi;
    wprt[1..17] := "NUMBER CHANGED = ";
    !P ed(wchange[1..10] var workb);
    r4 := !XF address_of(workb); r1 := 9;
    do if a[r4, 1] ≠ " " then exit(1) fi;
        r4 := r4 + 1;
        r1 := r1 − 1;
        if r1 = 0 then exit(1) fi od;
    a[!XF address_of(wprt) + 17, r1 + 1]
      := a[r4, r1 + 1];
    write1();
    !P close(ddin_ddname var os);
    !P close(rdsout_ddname var os)
where
proc endgroup() ≡
    wprt[1..4] := wlast;
    wsign := "+";
    if wnet < "hex 0x0C" then wsign := "−" fi;
    wprt[8..17] := "hex 0x40206B2020206B202120";
    !P edmk(wnet[1..10] var wprt[8..17], r1);
    r1 := r1 − 1;
    a[r1, 1] := wsign;
    write1(); write1();
    wchange := !XF ap(wchange, "hex 0x1C") end,
proc write1() ≡
    !P put(rdsout_ddname, wprt var os);
    wprt := wspaces end
end
```

Abstrakcja a Specyfikacja

System FermaT może automatyczni transformować aplikacje bez interwencji człowieka. Kolejnym krokiem w procesie abstrahowania jest zmiana reprezentacji danych tak by pliki stały się listami. Rozwiniemy procedurę write1 i zastąpimy zap, ap i sp wywoływanymi przez ich aktalne działania. Abstrahujemy od układu pliku wyjściowego przez stworzenie listy elementów danych, które pojawią się w każdej lini danych wyjściowych i zostaną dodane do tej listy do tablicy output:

```
begin
  i := 0; f_laaa := 1;
  output := ⟨⟨"MANAGEMENT REPORT"⟩,
            ⟨"ITEM       NET CHANGE"⟩⟩;
  xsw1 := 0;
  do i := i + 1; wrec := input[i];
     if i ⩾ n then exit(1) fi;
     if wrec.writem ≠ wlast
       then if f_laaa ≠ 1
              then endgroup() fi;

            f_laaa := 0;
            wlast := wrec.writem;
            wnet := 0 fi;
     if wrec.wrtype ≠ "R"
       then wnet := wnet − wrec.wrqty
        else wnet := wnet + wrec.wrqty fi;
     xsw1 := "hex 0xFF" od;
  if xsw1 = "hex 0xFF" then endgroup() fi;
  output :=output ++
            ⟨⟨"NUMBER CHANGED = ", wchange⟩⟩;
where
proc endgroup() ≡
  output := output ++ ⟨⟨wlast, wnet⟩⟩;
  wchange := wchange + 1 end
end
```

Możemy pozbyć się przełączników xsw1 i f_laaa przez rozwinięcie pierwszego kroku do pętli do .. od i uproszczenie. Użyjemy wtedy pętli odwróconej do przeniesienia jakichś instrukcji na górę pętli:

```
i := i + 1; wrec := input[i];
if i ⩾ n
  then skip
   else wlast := wrec.writem;
        wnet := 0;
        do if wrec.wrtype ≠ "R"
             then wnet := wnet − wrec.wrqty
              else wnet := wnet + wrec.wrqty fi;
           i := i + 1; wrec := input[i];
           if wrec.writem ≠ wlast ∨ i ⩾ n
             then endgroup();
                  if i ⩾ n
                    then exit(1)
                     else wlast := wrec.writem;
                          wnet := 0 fi fi od fi;
```

Chcemy zwinąć dwie instrukcje LAST := wrec.writem; wnet:= 0 na górę pętli, więc skonwertujemy pętlę do pętli podówjnie zagnieżdżonej (podwajanie pętli) i wyjmiemy instrukcje z wnętrza pętli (wyjmowanie z pętli). Potem zastosujemy pętlę odwróconą. Możemy potem wyjąć również instrukcje zaczynając od endgroup () z wnętrza pętli:

```
i := i + 1; wrec := input[i];
if i ⩾ n
  then skip
   else do wlast := wrec.writem;
           wnet := 0;
           do if wrec.wrtype ≠ "R"
                then wnet := wnet − wrec.wrqty
                 else wnet := wnet + wrec.wrqty fi;
              i := i + 1; wrec := input[i];


           if wrec.writem ≠ wlast ∨ i ⩾ n
             then exit(1) fi od;
        endgroup();
        if i ⩾ nexit(1) fi od fi;
```

W końcu zewnętrzna instrukcja if może być usunięta przez skonwertowanie pętli zewnętrznej do pętli while (jest to transformacja floop to while):

```
i := i + 1; wrec := input[i];
while i < n do
  wlast := wrec.writem;
  wnet := 0;
  do if wrec.wrtype ≠ "R"
       then wnet := wnet − wrec.wrqty
       else wnet := wnet + wrec.wrqty fi;
     i := i + 1; wrec := input[i];
     if wrec.writem ≠ wlast ∨ i ⩾ n
       then exit(1) fi od;
  endgroup() od;
```

Zauważ ,że po kodzie inizcjalizującym,niezmienne wrec = input[i] jest zawsze prawdą, a dla i > 1 wlast = input[i-1].writem jest również prawdą, ponieważ jest niezmienne wchange = l(output) - 2. Więc możemy usunąć te trzy zmienne z programu. Teraz program składa się z dwóch prostych zagnieżdżonych pętli, zewnętrznej pętli while iterujące po grupie rekordów i zakończonej wywołaniem endgroup(), podczas gdy wewnętrzna pętla do ... od iteruje po rekordach w grupie. Sugeruje to ,że zrestrukturyzowaliśmy dane blżej dopasowując je do struktury sterującej programu przez konwersję tablicy wejsciowej do listy list gdzie każda sublista składa się pojedynczej grupy elementów danych tak więc pętla zewnętrzna przetwarza sublisty raz w danym czasie a pętla wewnętrzna przetwarza elementy każdej sublisty. Kluczem do restrukturyzacji danych jest podział sekwencji wejściowej na sekcje , takie ,że pętla zewnętrzna przetwarza jeden segment na iterację. Łatwo to uzyskać funkcją split(p,B) która dzieli p na niepuste sekcje z sekacjmi dzielonymi między te pary elementów p gdzie B jest fałszywe. W naszym przypadku, warunek zakończenia w pętli wewnętrznej dostarcza właściwości przy której dzieli:

$$\textbf{funct}\ \mathsf{same\_item}(x, y) \equiv$$
$$x.\mathsf{writem} = y.\mathsf{writem}.$$

Wtedy nowa zmienna q jest wprowadzana z przypisaniem q := split(input,same_item). Indeksujemy listę q z dwoma zmiennymi k_1 i k_2 tak ,że $q[k_1][k_2]$ = input[i]. Aby to zrobić zachowamy niezmienność

$$: = +/(\ell * q[1\,..\,k_1 - 1]) + k$$

która razem z niezmiennością input = ++/q daje wymagany związek. Dodając te zmienne duchy do programu otrzymujemy

```
q := split(input, same_item);
i := 1; k1 := 1; k2 := 1;
while i < ℓ(input) do
  wnet := 0;
  do if input[i].wrtype ≠ "R"
       then wnet := wnet − input[i].wrqty
        else wnet := wnet + input[i].wrqty fi;
     i := i + 1;
     k2 := k2 + 1;
     if k2 > ℓ(q[k1]) then k1 := k1 + 1; k2 := 1 fi;
     if input[i].writem ≠ input[i − 1].writem
        ∨ i ⩾ ℓ(input) then exit(1) fi od;
     endgroup() od;
```

Możemy zastąpić referencje do konkretnych zmiennych input i i przez referencje do nowych zmiennych q, k_1 i k_2. Kluczowym punktem jest to ,że i < l(input) jeśli i tylko jeśli k_1 < l(q) i

$$\text{input}[i].\text{writem} \neq \text{input}[i-1].\text{writem}$$

jest prawdą kiedy mamy tylko przesunąć do nowej sekcji danych wejściowych: innymi słowy, precyzyjniej, kiedy k_2 = 1. Możemy usunąc konkretne zmienne z programu

```
q := split(input, same_item);
k1 := 1; k2 := 1;
while k1 < ℓ(q) do
  wnet := 0;
  do if q[k1][k2].wrtype ≠ "R"
       then wnet := wnet − q[k1][k2].wrqty
        else wnet := wnet + q[k1][k2].wrqty fi;
     k2 := k2 + 1;
     if k2 > ℓ(q[k1]) then k1 := k1 + 1; k2 := 1 fi;
     if k2 = 1 then exit(1) fi od;
  endgroup() od;
```

Teraz rdukujemy pętlę wewnętrzną do prostej pętli loop:

```
q := split(input, same_item);
k1 := 1;
while k1 < ℓ(q) do
  wnet := 0;
  for k2 := 1 to ℓ(q[k1]) step 1 do
    if q[k1][k2].wrtype ≠ "R"
      then wnet := wnet − q[k1][k2].wrqty
      else wnet := wnet + q[k1][k2].wrqty fi;
    k1 := k1 + 1;
    endgroup() od;
```

Możemy wyrazić zmianę wnet jako funkcję struktury:

```
funct change(s) ≡
  if s.wrtype ≠ "R" then −s.wrqty else s.wrqty fi.
```

Jasne jest ,że pętla wewnętrzna jest wyliczana jako suma danych wyjścowych change dla wszystkich struktur w subliście $q[k_1]$, więc możemy zwinąć pętlę wewnętrzną dla redukcji mapy działania:

```
q := split(input, same_item);
k1 := 1;
while k1 < ℓ(q) do
  wnet := +/change * q[k1];
  k1 := k1 + 1;
  endgroup() od;
```

Procedura endgroup() po porstu dołącza element do listy output:

```
q := split(input, same_item);
k1 := 1;
while k1 < ℓ(q) do
  wnet := +/change * q[k1];
  output := output ++ ⟨⟨q[k1][1], wnet⟩⟩;
  k1 := k1 + 1;
  endgroup() od;
```

więc możemy zwinąć pętlę zewnętrzną do mapy działania.Takie wyodrębnienie wygląda inaczej niż oryginalny asembler (spójrz poniżej) ale oba programy są sematycznie ekwiwalentne i genereują identyczne pliki wyjściowe (kiedy dane wyjściowe ze specyfikacji są formatowane do wzorca asemblera)

Źródło asemblerowe

```
*****************************************
*  TST004A0  SAMPLE PROGRAM (MCDONALDS) *
*****************************************
*
         REGEQU
*
*        PRINT NOGEN
TST004A0 CSECT
         STM   R14,R12,12(R13)
         LR    R3,R15
         USING TST004A0,R3
         ST    R13,WSAVE+4
         LA    R14,WSAVE
         ST    R14,8(R13)
         LA    R13,WSAVE
*
         OPEN  (DDIN,(INPUT))
         OPEN  (RDSOUT,(OUTPUT))
*
         MVC   WPRT(17),=CL17'MANAGEMENT REPORT'
         BAL   R10,WRITE1
         BAL   R10,WRITE1
         MVC   WPRT(20),=CL20'ITEM      NET CHANGE'
         BAL   R10,WRITE1
         BAL   R10,WRITE1
*
         MVI   XSW1,0
LAA      EQU   *
         GET   DDIN,WREC
         CLC   WRITEM,WLAST
         BE    LAC
LAAA     B     LAB
         BAL   R10,ENDGROUP
LAB      MVI   LAAA+1,0
         MVC   WLAST,WRITEM
         ZAP   WNET,=P'0'
         BAL   R10,PROCGRP
         MVI   XSW1,X'FF'
         B     LAA
LAC      BAL   R10,PROCGRP
         MVI   XSW1,X'FF'
         B     LAA
*
LAD      CLI   XSW1,X'FF'
```

```
          BNE   LADA
          BAL   R10,ENDGROUP
LADA      EQU   *
          MVC   WPRT(17),=CL17'NUMBER CHANGED = '
          ED    WORKB,WCHANGE
          LA    R4,WORKB
          LA    R1,9
LADB      CLI   0(R4),C' '
          BNE   LADC
          LA    R4,1(R4)
          BCT   R1,LADB
LADC      EX    R1,WMVC1
*WMVC1    MVC   WPRT+17(1),0(R4)
          BAL   R10,WRITE1
*
          CLOSE DDIN
          CLOSE RDSOUT
*
          L     R13,WSAVE+4
          LM    R14,R12,12(R13)
          SLR   R15,R15
          BR    R14
*
PROCGRP   EQU   *
          ST    R10,WST10A
          PACK  WORKA,WRQTY
          CLI   WRTYPE,C'R'
          BNE   LBA
          AP    WNET,WORKA
          B     LBB
LBA       SP    WNET,WORKA
LBB       L     R10,WST10A
```

```
         BR    R10
*
ENDGROUP EQU   *
         ST    R10,WST10A
         MVC   WPRT(4),WLAST
         MVI   WSIGN,C'+'
         CP    WNET,=P'0'
         BNL   LCA
         MVI   WSIGN,C'-'
LCA      EQU   *
         MVC   WPRT+7(10),=X'40206B2020206B202120'
         EDMK  WPRT+7(10),WNET
         BCTR  R1,0
         MVC   0(1,R1),WSIGN
         BAL   R10,WRITE1
         BAL   R10,WRITE1
         AP    WCHANGE,=P'1'
         L     R10,WST10A
         BR    R10
*
WRITE1   EQU   *
         PUT   RDSOUT,WPRT
         MVC   WPRT,WSPACES
         BR    R10
*
WMVC1    MVC   WPRT+17(1),0(R4)
*
WSAVE    DC    18F'0'
WST10A   DS    F
WREC     DS    0CL80
WRITEM   DS    CL4
         DS    CL1
WRTYPE   DS    CL1
         DS    CL1
WRQTY    DS    CL3
         DS    CL70
WPRT     DC    CL80' '
WSPACES  DC    CL80' '
WLAST    DC    CL4'****'
WCHANGE  DC    PL4'0'
WNET     DC    PL4'0'
WORKA    DC    PL2'0'
WORKB    DC    XL10'40206B2020206B202120'
WSIGN    DC    CL1' '
XSW1     DC    X'00'
*
         LTORG
*
DDIN     DCB   DDNAME=DDIN,
               DSORG=PS,
               EODAD=LAD,
               MACRF=GM
RDSOUT   DCB   DDNAME=RDSOUT,
               DSORG=PS,
               MACRF=PM
*
         END
```

```
begin
  q := split(input, same_item);
  output :=header ++ process * q
                  ++ ⟨⟨"NUMBER CHANGED = ", ℓ(q)⟩⟩
where
funct same_item(x, y) ≡ x.writem = y.writem.
funct process(L) ≡ ⟨L[1], +/change * L⟩.
funct change(s) ≡
  if s.wrtype ≠ "R" then −s.wrqty else s.wrqty fi.
end
```