Konstrukcja kompilatorów

Projekt nr 5

Termin oddawania: 13 stycznia 2019

Piątym projektem na pracownię z konstrukcji kompilatorów jest analiza żywych zmiennych. W systemie SKOS znajdują się nowe źródła kompilatora, gdzie w katalogu «source/mod_student» zamieściliśmy szablon rozwiązania w pliku «live_variables.ml». W katalogu «tests/pracownia4» znajdą Państwo przygotowany przez nas zestaw testów, ten sam co do pracowni numer cztery.

Analiza przepływu danych Na pracowni posługujemy się dwupoziomową reprezentacją, gdzie mamy graf sterowania nad blokami podstawowymi. Stąd równania analizy przepływu danych musimy wyrazić również na poziomie bloków podstawowych. Dziedziną analizy żywych zmiennych jest zbiór rejestrów. Niech l, l' oznaczają wierzchołki w grafie sterowania; succ(l) niech oznacza zbiór następników wierzchołka l; niech $A_\circ(l)$, $A_\bullet(l)$ oznaczają wiedzę związaną z wierzchołkiem l odpowiednio na wejściu, wyjściu – zgodnie z kierunkiem analizy; niech f_l oznacza funkcję transferu dla wierzchołka l. Równania dla analizy żywych zmiennych możemy sformułować w następujący sposób:

$$A_\circ(l) = \bigcup_{l' \in \mathsf{succ}(l)} A_\bullet(l')$$
$$A_\bullet(l) = f_l(A_\circ(l))$$

Zwróćmy uwagę, że $succ(\texttt{exit}) = \emptyset$, stąd $A_\circ(\texttt{exit}) = \emptyset$.
Dla wyszczególnionych wierzchołków `entry`, `exit` funkcja transferu jest identycznością. Dla bloku podstawowego o identyfikatorze l składającego się z instrukcji $i_0, \cdots, i_n$ oraz terminatora t funkcja transferu jest zdefiniowana jako złożenie funkcji transferu przypisanych do poszczególnych instrukcji i terminatora.

$$f_l = f_{i_0} \cdot f_{i_1} \cdots f_{i_n} \cdot f_t$$

Proszę zwrócić uwagę, że analiza żywych zmiennych jest analizą działającą od tyłu. Stąd w powyższych równaniach ściągamy wiedzę z wierzchołków będących następnikami w grafie oraz złożyliśmy funkcje transferu dla instrukcji w sposób odwrotny do sterowania (funkcja transferu dla pierwszej instrukcji wykona się na końcu, a dla terminatora na początku).
Funkcje transferu dla poszczególnych instrukcji oraz terminatorów mają następujący schemat:

$$f_i(d) = (d \setminus \mathsf{kill}_i) \cup \mathsf{gen}_i$$

gdzie kill$_i$, gen$_i$ oznaczają odpowiednio zbiór tych rejestrów, które dana instrukcja/terminator modyfikuje[1],używa.

Implementacja Typy danych do reprezentowania wiedzy o wierzchołkach w grafie sterowania są zdefiniowane w module «Analysis» biblioteki «Xi_lib» – w podmodułach «Knowledge» oraz «BlockKnowledge». Pomocnicze definicje dla analizy żywych zmiennych znajdują się w podmodule «LiveVariables» modułu «Analysis_domain» tej samej biblioteki.
Podstawowym typem jest «Knowledge.t». Reprezentuje on wiedzę na wejściu oraz wyjściu jakiegoś punktu programu (instrukcji, terminatora, całego bloku). Pola «pre», «post» oznaczają odpowiednio wiedzę na wejściu, wyjściu.

```
1 type 'a t = { pre: 'a ; post: 'a }
```

Wiedza dla wierzchołka w grafie sterowania jest reprezentowana przez typ «BlockKnowledge.t».

[1] W przypadku terminatorów zbiór kill jest zawsze pusty.

```
type 'a t = Simple of 'a Knowledge.t
          | Complex of
            { block     : 'a Knowledge.t
            ; body      : ('a Knowledge.t * Ir.instr) list
            ; terminator: 'a Knowledge.t * Ir.terminator
            }
```

Konstruktor «Simple» jest używany, gdy dla danego wierzchołka w grafie sterowania interesuje nas tylko wiedza na wejściu oraz wyjściu całego bloku. W przypadku analizy żywych zmiennych powinien on być jedynie użyty dla wyszczególnionych wierzchołków entry, exit. Konstruktor «Complex» jest używany, gdy dla danego wierzchołka będącym blokiem podstawowym, chcemy zachować również wiedzę na wejściu/wyjściu poszczególnych instrukcji oraz terminatora. Zwróćmy uwagę, że typ zawiera w sobie kopie instrukcji oraz terminatora, czyli de facto zawiera on w sobie definicję danego bloku podstawowego. To *edukacyjne* uproszczenie na potrzeby następnego zadania na pracowni, by można było się wygodnie iterować po wynikach analizy i definicji bloku jednocześnie. Pole «block» to wiedza dla całego wierzchołka.
Funkcja transferu dla wierzchołka będącym blokiem podstawowym może przyjąć następujący schemat:

```
val transfer_basic_block: Ir.label -> domain BlockKnowledge.t -> domain BlockKnowledge.t


let transfer_basic_block l current_knowledge =
    (* weź terminator *)
    let t = ControlFlowGraph.terminator cfg l in
    (* weź listę instrukcji *)
    let instr = ControlFlowGraph.block cfg l in
    (* analiza działa od tyłu, stąd bierziemy wiedzę na wyjściu (post) z bloku *)
    let input = BlockKnowledge.post current_knowledge in
    (* policz wiedze dla terminatora i instrukcji, nazwijmy wyniki *)
    (* t_res oraz instr_res. Niech output oznacza wyliczoną wiedzę na wejściu do bloku *)
    ...
    (* skonstruuj reprezentację wiedzy dla bloku *)
    let block = Knowledge.make ~pre:output ~post:input in
    BlockKnowledge.make_complex ~block ~body:instr_res ~terminator:t_res
```

Przy implementacji należy zwracać uwagę na kierunek. Typy «Knowledge» oraz «BlockKnowledge» reprezentują wiedzę zgodnie z kierunkiem sterowania. To oznacza, że niezależnie od kierunku analizy pole «pre» reprezentuje wiedzę przed wykonaniem opisywanego punktu programu. W powyższym kodzie identyfikatory «input, output» są natomiast zgodne z kierunkiem analizy, stąd w przedostatnim wierszu «output» jest przekazany jako «pre», a wcześniej «input» jest wzięty z pola «post». Proszę zwracać na kierunek szczególną uwagę – łatwo tutaj o pomyłkę przy implementowaniu analizy działającej od tyłu, jaką właśnie jest analiza żywych zmiennych. Typy danych opisują wiedzę zgodnie z kierunkiem sterowania, a nie analizy, ponieważ użytkowników wyników statycznej analizy nie interesuje w jaki sposób działał algorytm je wyliczający. Interesuje natomiast stan wiedzy przed/po wykonaniem rozpatrywanego punktu programu.
Kompilator od analizy oczekuje dostarczenia hashtablicy mapującej wierzchołki grafu sterowania na wiedzę reprezentowaną przez typ «BlockKnowledge.t». Inicjalizacja takiej tablicy znajduje się już w szablonie rozwiązania.

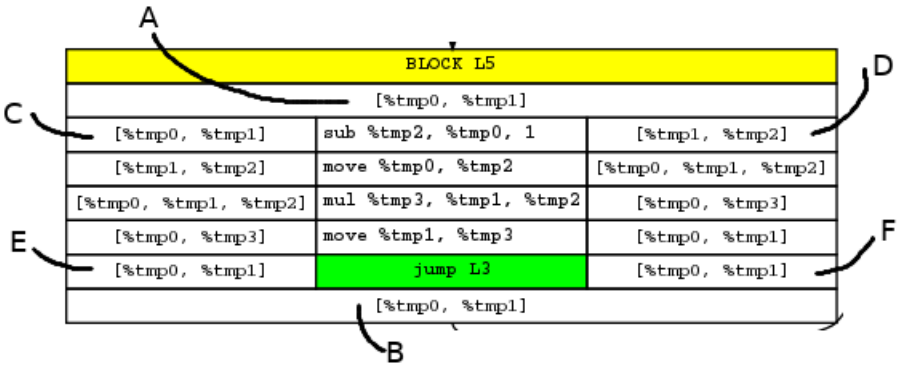
Zbiory kill, gen Zbiory kill, gen mogą być wyliczane za pomocą gotowych funkcji znajdujących się w module «Ir_utils» biblioteki «Xi_lib».

- «defined_registers_instr» zwraca listę definiowanych rejestrów – elementy zbioru kill dla instrukcji;
- «defined_registers_terminator» - zwraca pustą listę;
- «used_registers_instr» zwraca listę używanych rejestrów – elementy zbioru gen dla instrukcji;
- «used_registers_terminator» zwraca listę używanych rejestrów – elementy zbioru gen dla terminatora.

Komentarza mogą wymagać niektóre instrukcje:

- «I_Call(ys, p, xs, ms)» - oznaczająca $ys = p(xs)$. Dodatkowy argument ms oznacza listę rejestrów (sprzętowych), dla których nie mamy gwarancji, że ich wartość zostanie zachowana przez wywołaną procedurę. Możemy parametr ms widzieć jako specjalny znacznik w kodzie pośrednim, mówiący co analiza żywych zmiennych powinna dodać do zbioru kill. Stąd lista zmodyfikowanych rejestrów jest równa $ys{+}{+}ms$.
- «I_StoreMem(a,i,v)» - oznaczającą mem$[a+i] = v$. Proszę zwrócić uwagę, że instrukcja modyfikuje pamięć, a nie rejestry. Stąd lista zmodyfikowanych rejestrów jest pusta. Podobnie z «I_StoreArray» czy innymi instrukcjami związanymi z pamięcią.
- «I_Def(ys)» - specjalna instrukcja oznaczająca, że rejestry ys powinny być traktowane jako zmodyfikowane. Możemy tę instrukcję widzieć jako fragment kodu pośredniego mówiący co analiza żywych zmiennych powinna wrzucić w zbiór kill w danym punkcie programu.
- «I_Use(ys)» - analogiczna instrukcja do poprzedniej, tylko mówi o zbiorze gen.

Debugowanie Aby zobaczyć wyniki statycznej analizy musimy uruchomić kompilator z opcją «--extra-debug». Wyniki analizy żywych zmiennych zostaną zapisane do plików z rozszerzeniem «.lva.xdot». Można je oglądać za pomocą programu «xdot». Możemy posłużyć się obrazkiem z tego programu w celu dodatkowego wyjaśnienia typu «BlockKnowledge.t».



Możemy powyższy obrazek potraktować jako bezpośrednią wizualizację wspomnianego typu, a dokładniej mówiąc konstruktora «Complex». Zaznaczyliśmy, na obrazku następujące punkty:

- A - wiedza na wejściu do bloku
- B - wiedza na wyjściu z bloku
- C - wiedza na wejściu do pierwszej instrukcji
- D - wiedza na wyjściu z pierwszej instrukcji
- E - wiedza na wejściu do terminatora
- F - wiedza na wyjściu z terminatora

Wracając do typu danych, mamy:

- wiedza w punktach (A,B) jest polem «block»;
- wiedza w punktach (C,D) jest wiedzą z pierwszego elementu listy z pola «body»;
- wiedza w punktach (E,F) jest wiedzą z pary z pola «terminator»;
- wiedza w punkcie A zawsze powinna być równa wiedzy w polu C; no bo wiedza na wejściu na bloku to ta sama wiedza, co wchodzi do pierwszej instrukcji;
- wiedza na wejściu każdej kolejnej instrukcji powinna być równa wiedzy na wyjściu z poprzedniej;
- wiedza na wejściu do terminatora (punkt E) zawsze powinna być równa wiedzy na wyjściu z ostatniej instrukcji; jeżeli blok nie ma instrukcji to jest równa wejściu do bloku (punkt A).
- wiedza na wyjściu z bloku (punkt B) zawsze powinna być równa wiedzy na wyjściu z terminatora (punkt F).

Dla ciekawskich Analiza żywych zmiennych to jedna z klasycznych analiz przepływu danych. Jej zastosowanie w algorytmie alokacji rejestrów mogli Państwo zobaczyć na poprzedniej pracowni – na jej podstawie budowany jest graf interferencji. Często wyniki tej analizy służą też jako pomoc przy wielu operacjach, a wyniki tej analizy są trzymane przez kompilator bezpośrednio w definicji bloku podstawowego. Innymi słowami, informacje o żywych zmiennych to absolutny fundament dla kompilatora.
Reprezentowanie dziedziny analizy przez zbiór może być w praktyce mało efektywne. Niektóre klasyczne analizy przepływu danych da się reprezentować za pomocą ciągu bitów. Mnogościowe operacje jak suma, przecięcie czy dodawanie/usuwanie elementu mogą być efektywnie realizowane za pomocą operacji bitowych AND, OR, NOT. Analizy używające takich reprezentacji występują pod nazwą Bitvector Data-Flow Analysis. Analiza żywych zmiennych jest właśnie taką analizą. Możemy łatwo sprawdzić (lub trzymać w informacjach o procedurze) numer największego użytego rejestru aby dowiedzieć się ile bitów potrzebujemy. Następnie uznać że rejestr jest reprezentowany przez bit o takim samym indeksie.
Analizy przepływu danych są jednym z podstawowych narzędzi jakimi posługuje się kompilator przy optymalizacjach, stąd ich opis znajdziemy niemalże w każdym podręczniku poświęconym kompilatorom. Istnieje też podręcznik w całości poświęcony tego typu analizom [4].
Metody rozwiązywania analizy przepływu danych polegające na przeliczaniu wiedzy dla wierzchołków w grafie sterowania nazywamy iteracyjnymi (ang. *Iterative Data-Flow Analysis*). Używa się przy nich różnych wariantów algorytmu chaotycznej iteracji lub algorytmu operującego na liście roboczej. Są proste w implementacji, łatwe do zrozumienia i rozumiemy co wpływa na ich wydajność [7].
Alternatywnymi technikami są tak zwane algorytmy eliminacyjne (ang. *elimination methods*) [6], które bazują na innym podejściu do sterowania. Bloki podstawowe są grupowane w tak zwane regiony, które są układane w hierarchiczną strukturę. Regiony reprezentują jakąś strukturę sterowania w kodzie pośrednim i mogą posłużyć aby zawęzić analizy czy też optymalizacje do konkretnego regionu. Techniki eliminacyjne biorą strukturę regionów dla danej procedury i wyprowadzają na jej podstawie funkcje transferu dla poszczególnych regionów. Dla grafów sterowania posiadających pewne własności techniki eliminacyjne potrafią być efektywne. Znane sa różne algorytmy dekompozycji programu na hierarchiczną strukturę regionów. Podstawowe z nich (takie jak *interval* czy też *structural analysis*) są opisane w książce Muchnika [1] w rozdziałach 7.6 oraz 7.7. Bazujące na nich techniki eliminacyjne są omówione w rozdziałach 8.6, 8.7, 8.8 tego samego podręcznika. Dobry opis znajduje się również w rozdziale 9.7 klasycznego podręcznika Dragonbook [2]. Kompilator llvm również potrafi budować strukturę regionów z grafu sterowania. Bazuje na podejściu przedstawionym w artykule [8]. Nie używa jednak regionów do analizy przepływu danych, lecz w samym artykule znajduje się komentarz na temat zastosowania do tego typu analiz.
Gdy są Państwo zaznajomieni już z podstawami statycznej analizy, to możemy ponownie skupić naszą uwagę na postaci pośredniej SSA i lepiej zrozumieć jej fundamentalne cechy. W tej postaci każdy rejestr ma dokładnie jedną definicję. Stąd widząc użycie jakiegoś rejestru, dokładnie wiemy gdzie on jest zdefiniowany. Takie połączenia między użyciami, a definicjami nazywane są *use-def*. Kompilator musi wyliczać takie informacje na potrzeby niektórych optymalizacji. W przypadku kodu trójadresowego wyliczenie tych informacji wymaga posłużenia się znaną już Państwu analizą *Reaching Definitions*. W przypadku SSA nic nie musimy wyliczać, ponieważ w tej postaci połączenia use-def są uproszczone (tylko jedna definicja) i wbudowane w samą postać pośrednią. Korzystając z tych cech da się przeformułować niektóre statyczne analizy tak, że nie wyliczają one wiedzy dla każdego punktu programu, a jedynie dla instrukcji definiujących rejestry. Prowadzi to do efektywniejszych implementacji. Te analizy, które da się sformułować w ten sposób nazywane są rzadkimi (ang. *Sparse Data-Flow Analysis*). Więcej o znaczeniu SSA dla statycznej analizy znajdą Państwo w podręczniku [5].
Obszerny przegląd podstawowych technik analizy programów znajduje się w podręczniku [3], który stanowi podręcznik do wykładu *Analiza Programów Komputerowych*. Tych Państwa, którzy chcieliby więcej dowiedzieć o statycznej analizie zachęcamy do zapisania się na ten przedmiot.

Literatura

[1] S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA ©1997.

[2] A. V. Aho. M. S. Lam. R. Sethi, J. D. Ullman. Compilers: Principles, Techniques, and Tools. Addison-Wesley Publishing Company , USA ©2007.

[3] F. Nielson. H.R. Nielson. C. Hankin. Principles of Program Analysis. Springer Publishing Company, Incorporated ©2010.

[4] U. Khedker. A. Sanyal. B. Karkare. Data Flow Analysis: Theory and Practice. CRC Press, Inc. Boca Raton, FL, USA ©2009.

[5] Lots of authors. Static Single Assignment Book. http://ssabook.gforge.inria.fr/latest/book.pdf.

[6] B. G. Ryder. M. C. Paull. Elimination Algorithms for Data Flow Analysis. CSUR 1986.

[7] K. D. Cooper. T. J. Harvey. K. Kennedy. Iterative Data-flow Analysis, Revisited. Technical Report 2004.

[8] R. Johnson. D. Pearson. K. Pingali. The Program Structure Tree: Computing Control Regions in Linear Time. PLDI 1994.