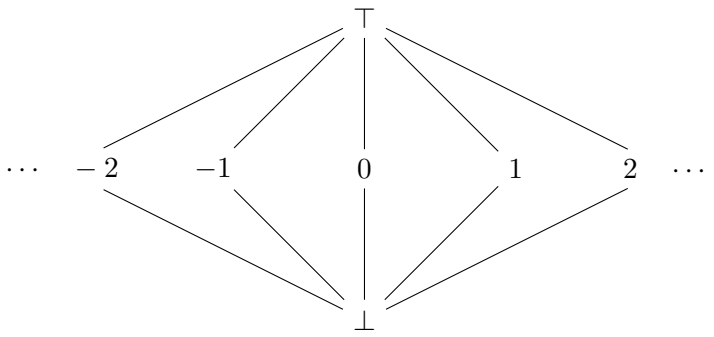
Konstrukcja kompilatorów

Projekt nr 6

Termin oddawania: 27 stycznia 2019

Piątym projektem na pracownię z konstrukcji kompilatorów jest zwijanie stałych oraz potrzebna tej optymalizacji analiza. W systemie SKOS znajdują się nowe źródła kompilatora, gdzie w katalogu «`source/mod_student`» zamieściliśmy szablony rozwiązania w plikach «`constant_folding.ml`» oraz «`constant_folding_analysis.ml`». W katalogu «`tests/pracownia4`» znajdą Państwo przygotowany przez nas zestaw testów, ten sam co do pracowni numer cztery.

Analiza przepływu danych Optymalizacja wymaga, aby dla każdego punktu programu wyliczyć informacje o wartościach rejestrów. Wiedza na temat rejestru jest reprezentowana przez kratę $\mathbb{Z}^{\top}_{\bot}$.



Krata zawiera w sobie zbiór liczb całkowitych oraz dwa wyszczególnione elementy $\bot$, $\top$. Liczby nie są ze sobą porównywalne. Liczba całkowita reprezentuje wiedzę, że rejestr tymczasowy w danym punkcie programu będzie zawierał w sobie taką wartość. Element największy $\top$ oznacza, że analiza nie potrafiła wyliczyć wartości rejestru. Element najmniejszy $\bot$ oznacza, że nie posiadamy wiedzy na temat danego rejestru.
Wiedza na temat punktu programu jest reprezentowana przez kratę funkcji $D = R \to \mathbb{Z}^{\top}_{\bot}$, gdzie R to zbiór rejestrów. Każda taka funkcja jest odwzorowaniem rejestru na wiedzę o nim.

$$
\begin{aligned}
\top_D &= \lambda x.\top \\
\bot_D &= \lambda x.\bot \\
f \sqcup_D g &= \lambda x.f(x) \sqcup g(x) \\
f \sqcap_D g &= \lambda x.f(x) \sqcap g(x)
\end{aligned}
$$

$$
f \sqsubseteq_D g \iff \forall x \in R.\ f(x) \sqsubseteq g(x)
$$

Niech l, l' oznaczają wierzchołki w grafie sterowania; $\mathrm{pred}(l)$ niech oznacza zbiór poprzedników 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; niech ι oznacza wartość ekstremalną; Równania dla analizy możemy sformułować w następujący sposób:

$$
\begin{aligned}
A_\circ(l) &= \bigsqcup_{l' \in \mathsf{pred}(l)} A_\bullet(l') && \text{gdy } l \neq \texttt{entry} \\
A_\circ(l) &= \iota && \text{gdy } l = \texttt{entry} \\
A_\bullet(l) &= f_l(A_\circ(l)) \\
\iota(x) &= \top && \text{gdy } x \text{ jest parametrem formalnym} \\
\iota(x) &= \bot && \text{gdy } x \text{ nie jest parametrem formalnym}
\end{aligned}
$$

Funkcje transferu dla wyszczególnionych wierzchołków `entry` oraz `exit` są identycznością. Funkcje transferu dla instrukcji «`I_Move`», «`I_Add`», «`I_Sub`», «`I_Mul`», «`I_Div`», «`I_Set`», «`I_Not`», «`I_Neg`», «`I_And`», «`I_Or`», «`I_Xor`» powinny zinterpretować działanie reprezentowane przez daną instrukcję. Funkcje transferu dla pozostałych instrukcji powinny uaktualniać wiedzę dla każdego rejestru modyfikowanego przez daną instrukcję. Uaktualniona wiedza dla każdego takiego rejestru powinna być elementem $\top$.
Przykładowo, funkcja transferu dla instrukcji dodawania jest zdefiniowana następująco. Zapis $f[x := y]$ oznacza funkcję, która odwzorowuje x na y, a dla pozostałych argumentów jest zgodna z f.

$$f_{\texttt{I\_Add(r, e1, e2)}}(d) = d\Big[r := [\![e_1]\!](d) \oplus [\![e_2]\!](d)\Big]$$

$$[\![r]\!](d) = d(r) \qquad \text{gdy } r \text{ jest rejestrem}$$
$$[\![c]\!](d) = c \qquad \text{gdy } c \text{ jest stałą}$$
$$c_1 \oplus c_2 = c_1 + c_2 \qquad \text{gdy } c_1, c_2 \text{ są stałymi}$$
$$v_1 \oplus v_2 = \top \qquad \text{gdy } v_1 \text{ albo } v_2 \text{ nie jest stałą}$$

W pewnym sensie optymalizacja uruchamia fragment kompilowanego programu. Stąd trzeba uważać czy podczas optymalizacji nie wykonamy nieprawidłowej operacji. Przykładowo, optymalizując program dzielący przez zero, kompilator nie powinien dać się nakłonić do tego samego.
Podczas wykładu interpretowaliśmy $\bot$ również jako „bełkot". Na pracowni, w celu łatwiejszego sprawdzania rozwiązań, przyjmijmy że interpretacje operatorów zawsze zwracają albo wyliczoną stałą albo $\top$.

Optymalizacja Optymalizacja dla każdej instrukcji powinna wziąć wynik analizy na wejściu do danej instrukcji i wykonać podstawienia na bazie tej wiedzy.
Dla każdego terminatora «`T_Branch`» optymalizacja powinna zinterpretować warunek skoku i zamienić terminatora na skok «`T_Jump`», gdy warunek wyliczy się do stałej.

Implementacja Kratę D reprezentujemy za pomocą mapy «`Int32.t option Ir.RegMap.t`». Pomocnicze definicje znajduja się w podmodule «`ConstantFolding`» modułu «`Analysis_domain`» w bibliotece «`Xi_lib`». Niech m będzie mapą w OCamlu. Niech zapis $x \notin dom(m)$ oznacza, że mapa nie ma klucza x. Niech zapis $m(x) = v$ oznacza, że mapa odwzorowuje klucz x na wartość v. Mapa m reprezentuje funkcję $f \in D$ według następujących zasad:

Implementacja	Teoria
$x \notin dom(m)$	$f(x) = \bot$
$m(x) =$ `None`	$f(x) = \top$
$m(x) =$ `Some(c)`	$f(x) = c$

Przypomnijmy, że parametrami formalnymi procedury są pierwsze rejestry tymczasowe. Procedura mająca dwa parametry reprezentuje je przez rejestry «`tmp0`» oraz «`tmp1`». W szablonie rozwiązania znajduje się już użycie funkcji zwracającej liczbę formalnych parametrów analizowanej procedury. Trzeba posłużyć się jej wynikiem w celu skonstruowania wartości ekstremalnej ι.

Debugowanie Aby zobaczyć wyniki statycznej analizy musimy uruchomić kompilator z opcją «`--extra-debug`».
Wyniki analizy żywych zmiennych zostaną zapisane do plików z rozszerzeniem «`.cfa.xdot`». Można je oglądać za pomocą programu «`xdot`».

Dla ciekawskich Optymalizujące kompilatory często stosują o wiele bardziej zaawansowany wariant powyższego algorytmu zwany Sparse Conditional Constant Propagation [1, 2]. Jedną z ważnych różnic w stosunku do wariantu prostego jest operowanie na postaci pośredniej SSA. Drugą jest używanie o wiele bardziej wyrafinowanej statycznej analizy opierającej się o abstrakcyjną interpretację, a nie analizę przepływu danych. Mimo dużego znaczenia i praktycznego i teoretycznego, abstrakcyjna interpretacja nie jest omawiana

w podręcznikach dedykowanych kompilatorom. Podczas studiów będą mogli Państwo zapoznać się z nią na przedmiocie Analiza Programów Komputerowych.
Nasz kompilator uruchamia każdą optymalizację raz. Praktycznym podejściem jest uruchamiać zestaw optymalizacji w pętli. Często rezultat działania poprzednich iteracji odblokowuje możliwość dalszych transformacji programu.
Jedną z fundamentalnych optymalizacji brakujących w naszym kompilatorze jest *Common Subexpression Elimination* (CSE). Ta optymalizacja posługuje się analizą przepływu danych aby wyliczyć dla każdego punktu programu informacje, jakimi instrukcjami rejestry zostały wyliczone. Na podstawie tej wiedzy wykrywa te instrukcje, które wyliczają coś już wyliczonego. Możemy zastanowić się jak taka optymalizacja powinna sobie radzić z instrukcjami operującymi na pamięci. Problemem jest tutaj wykrycie, kiedy wyliczona wiedza się deaktualizuje. W przypadku rejestrów jest to zwykle proste, gdy pamiętamy że rejestr r został wyliczony przez instrukcję $i(a, b)$ i właśnie rozpatrywana instrukcja modyfikuje a lub b, to wiedza o r idzie do kosza. W przypadku pamięci problem jest bardziej złożony z powodu możliwego aliasingu – ten sam wskaźnik może być zapisany w wielu rejestrach i komórkach na stercie. Bezpiecznie w takim przypadku jest założyć że dowolny zapis do pamięci dezaktualizuje całą wiedzę o stercie. W artykule [3] znajdą Państwo informacje o tym jak do tego problemu podchodzi kompilator gcc. W artykule [4] znajdą Państwo przykład zaawansowanej optymalizacji stosowanej w kompilatorze llvm. Jest to przykład optymalizacji celujący w programy wykonujące dużo obliczeń numerycznych i korzystających z takich rozszerzeń języka C jak OpenMP.

Literatura

[1] Sparse Conditional Constant Propagation - Wikipedia

[2] M. N. Wedgman, F.K. Zadeck. Constant Propagation with Conditional Branches. TOPLAS 1991.

[3] D. Novillo. Memory SSA – A Unified Approach for Sparsely Representing Memory Operations. GCC Developer's Summit 2007.

[4] T. Grosser. H. Zheng. R. Aloor. A. Simbürger. A. Größlinger. L. Pouchet. Polly - Polyhedral Optimization in LLVM. CGO 2011