Kompilator Javalette

Łukasz Czajka

30 czerwca 2008

Spis treści

1 Frontend

Parser zaimplementowany w pliku `parse.y` generuje drzewo składniowe używając funkcji `new_node()` z pliku `tree.h`. Moduł `tree.c` jest odpowiedzialny za sprawdzenie poprawności kontekstowej tego drzewa, oraz częściowo za generację z niego kodu pośredniego poprzez wywoływanie odpowiednich funkcji z modułu `quadr.h`. Plik `types.c` odpowiada za (bardzo prostą) analizę równoważności typów.

2 Kod pośredni

Struktury kodu pośredniego zdefiniowane są w pliku `quadr.h`, a operacje na nich w `quadr.c`.

Kod pośredni jest rodzjem kodu czwórkowego. Różni się tym od tego dla interpretera *iquadr*, że nie operuje na rejestrach, tylko na zmiennych, które są lokalne dla jednego wywołania funkcji.

Kod pośredni generowany jest od razu z podziałem na bloki bazowe, a etykiety utożsamiane są z początkami tych bloków.

Nie są rozróżniane zmienne tymczasowe od nietymczasowych. Generator kodu tego nie potrzebuje. Potrzebuje za to globalną informację o żywotności zmiennych.

3 Analiza żywotności

Globalna analiza żywotności zmiennych zaimplementowana jest w pliku `flow.c` w funkcji `analyze_liveness()`. Funkcja ta zakłada, że wcześniej wywołano `create_block_graph()` w celu stworzenia grafu przepływu z bloków bazowych. Ze względu na postać kodu pośredniego każdy wierzchołek w tym grafie może mieć co najwyżej dwa następniki.

Dla każdego bloku obliczamy zmienne żywe na początku bloku (`vars_at_start` w `basic_block_t`) oraz na końcu (`live_at_end`).

Algorytm analizy żywotności można by zakwalifikować jako nietrywialny, lecz jest on zupełnie standardowy, więc nie będę tutaj przepisywać książki. Nadmienię tylko, że wszystkie zbiory (in, out, gen, kill) zaimplementowałem jako drzewa czerwono-czarne.

Faza analizy żywotności oblicza także dla każdej zmiennej żywej na początku bloku odległość jej najbliższego użycia (NUD), definiowaną jako ilość instrukcji od początku bloku do pierwszego użycia tej zmiennej w bloku, lub minimum z NUD dla następników tego bloku w grafie przepływu.

4 Generacja kodu wynikowego

Kod wynikowy generowany jest z kodu pośredniego poprzez moduł `gencode.c` w połączeniu z odpowiednim obiektem `backend_t`, który udostępnia funkcje wirtualne[1] do wypisywania kodu dla odpowiednich konstrukcji. Moduł `gencode.c` bezpośrednio zajmuje się tylko alokacją rejestrów, śledzeniem ich zawartości oraz decydowaniem co kiedy przesłać do pamięci lub zachować na boku w innym rejestrze, a przy tym także analizą żywotności na poziomie jednego bloku bazowego (korzystając z informacji globalnych).

Standardowa metoda generacji kodu poprzez śledzenie zawartości rejestrów jest tutaj rozszerzona w ten sposób, że na końcu bloku *nie* odsyłamy zmiennych żywych do pamięci. Oczywiście trzeba jednak zapewnić, aby następnicy tego bloku w grafie przepływu wiedzieli, gdzie jest zmienna.

Sposób generacji kodu pośredniego oraz kolejność generacji kodu wynikowego dla bloków bazowych zapewniają, że w momencie generowania kodu dla bloku był już generowany kod dla co najwyzej jednego jego następnika. Ponadto każda etykieta (utożsamiana z początkiem jakiegoś bloku bazowego) jest celem co najwyżej jednego skoku. Można to zapewnić dodając zawsze odpowiednią liczbę pustych bloków. Te obserwacje uzasadniają poprawność poniższego algorytmu.

Zmiennym żywym na początku bloku, dla którego jeszcze nie generowaliśmy kodu, przypisywać będziemy lokacje na podstawie ich lokacji na końcu bloków-poprzedników w grafie przepływu.

Dla każdej zmiennej v żywej na końcu aktualnego bloku B, sprawdź czy w którymś z następników B ta zmienna ma już przypisaną lokację. Ze względu na powyższe obserwacje może być co najwyżej jeden taki następnik. Jeśli nie, to przypisz obecne lokacje zmiennej do obu następników. Jeśli tak, to przenieś zmienną do jakiejś lokacji z tego następnika, oraz usuń pozostałe lokacje jeśli kod dla tego następnika nie był jeszcze generowany, lub przenieś zmienną do wszystkich lokacji, jeśli był. Nie jest wykluczone, że obecne lokacje i lokacje w następniku się pokrywają – wtedy jeśli kod dla następnika nie był generowany, to wystarczy usunąć te, które się nie powtarzają. Następnie uaktualniamy lokacje tej zmiennej w drugim następniku.

Może się potem okazać, że w pustych blokach, które dodaliśmy na etapie generacji kodu pośredniego trzeba będzie wygenerować jakieś przypisania.

Powyższa metoda nieco skomplikowała algorytm generacji kodu, ale pozwala na generowanie całkiem dobrego kodu, pod względem odwołań do pamięci, nawet bez globalnej alokacji rejestrów.

[1] No tak ściśle rzecz biorąc, to jest to struktura ze wskaźnikami do funkcji, bo pisałem to w C. Ale przecież liczą się idee a nie nazwy...

4.1 Alokacja rejestrów

Alokacja rejestrów zaimplementowana jest w funkcji `bellady_ra` w pliku `gencode.c`. Jest to strategia Bellady'ego wyboru rejestru dla którego średnia najbliższego użycia dla przechowywanych w nim wartości jest najmniejsza spośród rejestrów, których wybór spowoduje odesłanie do pamięci najmniejszej liczby zmiennych.

4.2 Dla interpretera iquadr

Generacja kodu dla interpretera *iquadr* zaimplementowana jest w pliku `quadr_backend.c`. Implementuje ona po prostu funkcje z `backend_t`.

4.3 Na intela i386

Dla kodu assemblera NASM jest podobnie jak w powyższym punkcie (plik `i386_backend.c`) z tym, że implementacja tych funkcji jest już nieco mniej trywialna.

5 Optymalizacje

5.1 Lokalna optymalizacja bloków bazowych

Optymalizacja bloków bazowych zaimplementowana jest w pliku `opt.cpp`. Obejmuje ona zwijanie stałych, eliminację podwyrażeń wspólnych oraz propagację kopii. Wykorzystywany jest standardowy algorytm budowy DAGa dla bloku bazowego, który można znaleźć np. w [1].

5.2 Pomijanie wskaźnika ramki

Pomijany jest wskaźnik ramki stosu. Adresowanie odbywa się od czubka stosu. Maksymalna wysokość stosu dla jednej funkcji jest wyliczana w trakcie generacji kodu, po czym funkcja `fix_stack()` z pliku `outbuf.c` poprawia odwołania do stosu uwzględniając tę informację.

6 Tablice

Zaimplementowałem jakąś podstawową wersję tablic. Nie mogą być one kopiowane ani przekazywane jako parametr. Mają stałą wielkość i nie mogą być wielowymiarowe. Kod dla nich generowany nie jest specjalnie optymalny.

7 Testy

- Tablice: `good004.jl`, `bad104.jl`, `bad105.jl`, `bad106.jl`.
- Optymalizacje: `good006.jl`, `good007.jl`. Można zobaczyć kod wyniowy. Najlepiej wychodzi dla interpretera *iquadr*.
- Pozostałe „feature'y" są raczej integralną częścią kompilatora, bez których działać on nie potrafi. W związku z tym ich działanie można zaobserwować na dowolnym z testów.

8 Znane niedociągnięcia

- Nie zdążyłem zaimplementować optymalizacji przez szparkę dla assemblera x86, więc mimo włączonych optymalizacji pojawia się trochę „głupich" instrukcji w stylu `sub esp,0` czy `fadd st1; fstp st0`, które mogłyby być w prosty sposób ulepszone.
- Przekazywanie parametrów w rejestrach też nie działa mimo, że jest taka opcja. Nie zdążyłem zdebugować.
- Opcja `-O2` nie działa.
- Używanie tablic pogarsza efekty optymalizacji bloków bazowych.

Literatura

[1] Aho, Sethi, Ullman, *Kompilatory*.