Konstrukcja kompilatorów

Projekt nr 2

Termin oddawania: 7 grudnia 2018

UWAGA! Przed przystąpieniem do zadania proszę zapoznać się ze stronami w systemie SKOS poświęconymi pracowni. Część ze stron została uaktualniona. Pojawiła się też nowa strona poświęcona językowi pośredniemu.

Trzecim projektem na pracownię z konstrukcji kompilatorów jest translacja abstrakcyjnego drzewa rozbioru programu na język pośredni. W systemie SKOS znajdują się nowe źródła kompilatora, gdzie w katalogu «source/mod_student» zamieściliśmy szablon rozwiązania w pliku «translator.ml». W katalogu «tests/pracownia3» znajdą Państwo przygotowany przez nas zestaw testów.
W przygotowanym szablonie znajdują się przygotowane funkcje, jakimi posługiwali się Państwo na wykładzie, do przydzielenia nowego rejestru, nowego bloku podstawowego, doklejenia instrukcji na koniec bloku, ustawienia terminatora.
Wszelkie niejasności powinni Państwo rozstrzygnąć obserwując jak radzi sobie nasza wtyczka.

Nasze podejście Nasze rozwiązanie posługuje się trzema funkcjami przy tłumaczeniu ciała funkcji:

- `translate_expression` - tłumaczy wyrażenie, zwraca informację jaka jest wartość wyrażenia (w przypadku literałów `EXPR_Int` lub `EXPR_Char`) lub nazwę tymczasowego rejestru, w którym będzie znajdowała się policzona wartość po wykonaniu właśnie wygenerowanych instrukcji;
- `translate_condition` - tłumaczy wyrażenie w specjalny sposób na użytek instrukcji warunkowych;
- `translate_statement` - tłumaczy komendę języka.

Na wykładzie został Państwu przedstawiony pewien protokół postępowania, pomocny przy zarządzaniu blokami podstawowymi, jakim posługują się funkcje tłumaczące. Spiszmy go:

- Funkcje tłumaczące dostają jako argument identyfikator bloku podstawowego, na koniec którego powinny doklejać wygenerowane instrukcje.
- Każda taka funkcja zwraca identyfikator bloku podstawowego, w którym powinna być dalej kontynuowana translacja.

Rozważmy przykład, tłumaczenie operatora odejmowania oraz literału znakowego.

```
let rec translate_expression env current_bb = function
    | Ast.EXPR_Char {value; _} ->
        current_bb, E_Int (Int32.of_int @@ Char.code value)
    | Ast.EXPR_Binop {lhs; rhs; op=Ast.BINOP_Sub; _} ->
        let my_reg              = allocate_register () in
        let current_bb, res_lhs = translate_expression env current_bb lhs in
        let current_bb, res_rhs = translate_expression env current_bb rhs in
        append_instruction current_bb @@ I_Sub (my_reg, res_lhs, res_rhs);
        current_bb, E_Reg my_reg
    ...
```

W przypadku tłumaczenia literału `EXPR_Char` nie musimy wygenerować żadnych instrukcji – wartość wyrażenia jest znana. Zwracamy ten sam identyfikator bloku podstawowego, co dostaliśmy, oraz wartość wyrażenia.
W przypadku tłumaczenia wyrażenia oznaczającego odejmowanie rekurencyjnym wywołaniom przekazujemy identyfikator obecnego bloku podstawowego `current_bb`, a one nam zwracają identyfikator bloku, który

powinniśmy traktować po wywołaniu jako obecny. W wyniku translacji podwyrażeń dostaliśmy też atomy `res_lhs`, `res_rhs` reprezentujące ich wartości. Do obecnego bloku doklejamy na koniec instrukcję odejmowania umieszczającą wynik w przydzielonym rejestrze. Ostatecznie zwracamy identyfikator obecnego bloku oraz informację o wyniku odejmowania.
Proszę zwrócić uwagę, że operator dodawania może oznaczać i dodawanie liczb i konkatenację tablic. Do mechanizmu translacji została przekazana hashtablica «`node2type`», którą uzupełniali Państwo w typecheckerze. Muszą się Państwo nią posłużyć aby pobrać typ węzła i na jego podstawie rozstrzygnąć co w danym miejscu programu operator dodawania robił.

Tłumaczenie warunków W naszym rozwiązaniu funkcja `translate_condition` odpowiada za kompilację wyrażeń użytych jako warunek. Obsługuje ona jedynie wyrażenia boolowskie (literały boolowskie, operatory logiczne, operatory porównania). Gdy nie zachodzi żaden z przypadków, który umiemy lepiej obsłużyć, to kompilujemy wyrażenie w sposób normalny, a następnie generujemy skok na podstawie wyniku. Funkcja bierze identyfikator obecnego bloku podstawowego oraz identyfikator bloku, gdzie sterowanie ma pójść gdy warunek jest fałszywy – nazwijmy ten blok alternatywnym. Zgodnie z konwencją zwracamy blok, gdzie powinna odbywać się dalej kompilacja. W przypadku tej funkcji założyliśmy że, obecny blok to ten gdzie pójdzie sterowanie gdy warunek będzie prawdziwy.

Rozważmy przykład, tłumaczenie literałów oraz generyczny przypadek.

```
let rec translate_condition env current_bb alt_bb = function
    | Ast.EXPR_Bool {value=true; _} ->
        current_bb
    | Ast.EXPR_Bool {value=false; _} ->
        set_jump current_bb alt_bb;
        allocate_block ()
    ...
    | e ->
        let current_bb, res = translate_expression env current_bb e in
        let next_bb         = allocate_block () in
        set_branch COND_Ne res (E_Int (Int32.of_int 0)) current_bb next_bb alt_bb;
        next_bb
```

Jeżeli warunkiem jest literał `true` to nic nie musimy robić, kompilacja może dalej odbywać się w obecnym bloku. Jeżeli warunkiem jest literał `false` to zawsze musimy wykonać skok do alternatywnego bloku. W tym przypadku, zgodnie z protokołem, musimy przydzielić nowy blok podstawowy, w którym będzie odbywać się kompilacja kodu gdyby warunek był prawdziwy – wywołująca nas funkcja tego wymaga. Przydzielony blok będzie nieosiągalny i zostanie skasowany przez późniejsze fazy kompilatora.
W naszej implementacji funkcje `translate_expression` oraz `translate_condition` są wzajemnie rekurencyjne. Funkcja `translate_expression` podczas kompilacji wyrażeń boolowskich posługuje się funkcją do kompilacji warunków. Bloki, do których zostaną wykonane skoki, ustalają wartość wyrażenia na 0 lub 1.

Translacja komend Funkcja `translate_statement` w naszym rozwiązaniu nie zwraca żadnych dodatkowych informacji. Podąża tylko za ustalonym protokołem. Proszę zwrócić uwagę, że skoro możemy zwrócić tylko jeden identyfikator bloku podstawowego, to w przypadku kompilacji komend gdzie sterowanie się rozwidla (pętle lub komendy warunkowe) to jesteśmy odpowiedzialni za przydzielenie tego bloku podstawowego, gdzie sterowanie znów się łączy – jego identyfikator zwracamy.
Zwróćmy uwagę, że podczas translacji komend musimy zwrócić uwagę na inicjalizację tablic. Dla komendy «`STMT_VarDecl`» reprezentującej poniższy fragment programu musimy wygenerować kod co zaalokuje tablicę o odpowiednim rozmiarze.

```
xs:int[30];
```

Kolejność ewaluacji Podczas translacji języka źródłowego do pośredniej reprezentacji podejmujemy decyzje o kolejności ewaluacji. Ustalmy, że we wszystkich wyrażeniach i komendach ewaluujemy wszystko od lewej do prawej. To znaczy, że w operatorach binarnych jak $e_1 + e_2$ powinniśmy wygenerować kod co wpierw wyliczy e_1, a następnie e_2. Natomiast w wywołaniach liczymy rzeczywiste argumenty od pierwszego do ostatniego. Proszę zwrócić uwagę, że operatory logiczne wymagają specjalnej strategii ewaluacji (ang. *short-circuit evaluation*). Przykładowo, kod wygenerowany dla wyrażenia $e_1 \mid e_2$ powinien liczyć e_2 tylko, gdy e_1 wyliczył się do fałszu.

System typów Wygodne testowanie kompilatora wymaga, abyśmy osłabili jedną regułę z oficjalnego systemu typów języka Xi. Tych Państwa, którzy przenoszą rozwiązania z poprzednich pracowni proszę o zamianę lewej reguły na prawą:

$$\frac{\Gamma \vdash s_1 : \texttt{unit}, \Gamma_1 \quad \cdots \quad \Gamma_{n-1} \vdash s_n : R, \Gamma_n}{\Gamma \vdash \{s_1 \cdots s_n\} : R, \Gamma} \qquad \frac{\Gamma \vdash s_1 : \mathbf{R_1}, \Gamma_1 \quad \cdots \quad \Gamma_{n-1} \vdash s_n : R, \Gamma_n}{\Gamma \vdash \{s_1 \cdots s_n\} : R, \Gamma}$$

Nie chcemy sprawdzać czy wewnątrz bloku pojawił się koniec sterowania. Dzięki temu osłabieniu możemy pisać takie testy do języka Xi:

```
main():int {
    x:int = 33;
    if (x == 33) {
        return 100
    } else {
        return 10
    }
    return 99
}

//@PRACOWNIA
//@out Exit code: 100
```

Jeżeli program zwróci 99 to wiemy, że translacja komendy warunkowej ma poważną usterkę. Tego typu testy mogą być pomocne przy stawianiu pierwszych kroków podczas rozwiązywania zadania.

Dla ciekawskich Język pośredni nie ma takiego samego statusu jak język programowania. Często jego postać tekstowa istnieje tylko na potrzeby programistów kompilatorów, aby mieli wgląd w działanie algorytmów. Jego reprezentacja wewnątrz kompilatora często jest też dostosowana na potrzeby operujących na nim algorytmów do statycznej analizy czy też optymalizacji. Przykładowo, na wykładzie oraz pracowni przedstawiliśmy Państwu dwu poziomową reprezentację, gdzie sterowanie jest reprezentowane przez graf sterowania nad blokami podstawowymi, w których zawarte są sekwencje instrukcji kodu trój-adresowego (lub SSA). Inna reprezentacja *tych samych języków* została wypracowana w [4, 7], gdzie spłaszczono wszystko do jednego poziomu. Nie ma bloków podstawowych oraz sekwencji instrukcji, a wszystko jest reprezentowane przez jeden graf. Taka reprezentacja umożliwiła autorom efektywniejszą implementację niektórych algorytmów. Stąd patrząc holistycznie na postać pośrednią możemy widzieć nie tylko język, ale także strukturę danych. Wspomnianą jednopoziomową grafową reprezentacją SSA posługuje się kompilator JIT wewnątrz maszyny wirtualnej Javy [5].
Klasyczny algorytm tłumaczący do SSA [2] nie jest trywialny i wymaga, aby kod był już reprezentowany w postaci nadającej się do statycznych analiz wymaganych przez mechanizm translacji. W tej dekadzie pojawiły się konkurencyjne metody translacji co potrafią bezpośrednio tłumaczyć AST do SSA [6].
Inną postacią pośrednią, niż dotąd omówione na wykładzie, jest *Program Dependency Graph* [1], gdzie program reprezentujemy przez graf gdzie, wierzchołkami są instrukcje lub bloki podstawowe. Krawędzie natomiast reprezentują zależności sterowania albo danych. Taka reprezentacja może posłużyć kompilatorowi

do wykrycia tych fragmentów programu, które można automatycznie zrównoleglić. Więcej o bezpośredniej reprezentacji zależności danych znajdą Państwo w rozdziale 9 książki [3].
W porównaniu do kodu trój-adresowego postacie pośrednie używane w kompilatorach języków funkcyjnych są w stanie swobodniej operować funkcjami oraz rozumieją zagadnienie zagnieżdżenia funkcji w funkcji. Na nich kompilator wykonuje potrzebne transformacje aby takich funkcji się pozbyć i umożliwić przetłumaczenie kodu na język maszynowy lub bardziej standardową postać pośrednią. W książce [9] znajdą Państwo opis języka pośredniego bazującego na kontynuacjach, a w artykułach [10, 11] znajdą Państwo opisy innych postaci pośrednich, takich jak A-NF lub Monadic.
Różne koncepcje z języków funkcyjnych pojawiają się w językach programowania głównego nurtu, dzisiaj nawet programiści C++ mogą posługiwać się funkcjami anonimowymi i funkcjami wyższego rzędu. W pracy [8] znajdą Państwo specjalną postać pośrednią, bazującą na pracach [9, 7], mającą ułatwić kompilatorom języków imperatywnych radzenie sobie z elementami programowania funkcyjnego.

Literatura

[1] J. Ferrante. K. J. Ottenstein. J. D. Warren. The Program Dependence Graph and Its Use in Optimization. TOPLAS 1987.

[2] R. Cytron. J. Ferrante. B. K. Rosen. M. N. Wegman. F. K. Zadeck. Efficiently Computing Static Single Assignment Form and The Control Dependence Graph. TOPLAS 1991.

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

[4] C. N. Click. Combining Analyses, Combining Optimizations. Phd 1995.

[5] M. Paleczny. C. Vick. C. Click. The java hotspot server compiler. JVM 2001.

[6] M. Braun. S. Buchwald. S. Hack. R. Leißa. C. Mallon. A. Zwinkau. Simple and Efficient Construction of Static Single Assignment Form. CC 2013.

[7] C. Click. M.Paleczny. A Simple Graph-Based Intermediate Representation. IR 1995.

[8] R. Leißa. M. Köster. S. Hack. A Graph-Based Higher-Order Intermediate Representation. CGO 2015.

[9] A. W. Appel. Compiling with continuations. Cambridge University Press New York, NY, USA ©1992.

[10] A. Sabry, M. Felleisen. Reasoning about Programs in Continuation-Passing Style. LFP 1992.

[11] E. Moggi. Notions of Computation and Monads. Information and Computation 1991.