Kompilacja przez interpretację: metaprogramowanie w C-=-1

1st Adam Grabski
Wydział Elektroniki i Technik Informatycznych
Politechnika Warszawska
Warszawa, Polska
adam.grabski.stud@pw.edu.pl

***Abstract*—Statyczne metaprogramowanie to technika programistyczna umożliwiająca analizę, modyfikację lub generację kodu w czasie kompilacji. Współczesne, niskopoziomowe języki programowania wprowadzają wsparcie dla tych mechanizmów, w ograniczonym zakresie. W ramach tej pracy, zaproponowano nowy język programowania: C-=-1, zaprojektowany od początku z myślą o metaprogramowaniu, oraz nowe podejście do konstrukcji kompilatora: kompilacja przez interpretację.**

***Index Terms*—Metaprogramowanie, język programowania, kompilator, kompilacja**

I. Wstęp

Dwa trendy w rozwoju współczesnych języków programowania niskiego poziomu to wykonanie kodu w czasie kompilacja oraz wsparcie dla statycznego metaprogramowania. Mechanizmy takie jak system makr w Rust oraz szablony w C++ umożliwiają generowanie kodu oraz jego analizę w pewnym ograniczonym zakresie. Dają one niewielki wgląd w strukturę programu oraz wykorzystują składnię odrębną od reszty języka, co utrudnia ich efektywne wykorzystanie.

Te języki, umożliwiają też wykonywanie kodu w czasie kompilacji, jednak tutaj programista także spotyka się ze znacznymi ograniczeniami. C++ wprowadził tą możliwość jako pierwszy, tworząc funkcje constexpr w C++11. Na początku dawały dostęp wyłącznie do najbardziej podstawowych elementów języka, jendak z czasem te ograniczenia były znoszone. W C++20, funkcje constepxr mają już tylko jedno, poważne ograniczenie: nie mogą zwracać dynamicznie alokowanej pamięci. Rust podąża śladami C++, wprowadzając funkcje const.

Celem C-=-1 jest umożliwienie wykonywania oraz dowolnego modyfikowania kodu w czasie kompilacji. Osiągnięto to poprzez stworzenie interpretera tego języka a następnie udostępnienie mu programu jako modyfikowalnej struktury danych. W ten sposób, dowolny fragment kodu może być wykonany w czasie kompilacji oraz użytkownik może dowolnie modyfikować swój program, używając normalnej składni.

II. Struktura języka

Język C-=-1 bazuje na C++, z elementami składni Rust. Program może składać się z przestrzeni nazw, funkcji oraz typów takich jak klasy, typy enumeracyjne, atrybuty oraz interfejsy. Tak jak C++, C-=-1 jest językiem programowania ogólnego przeznaczenia, ze wsparciem dla programowania obiektowego.

A. System typów

W przeciwieństwie do C++, C-=-1 nie pozwala na dziedziczenie po klasach. Zamiast tego, wprowadza koncepcję interfejsu z języków takich jak C#. Interfejs może zawierać wyłącznie nagłówki metod, które implementująca klasa musi zdefiniować. Klasy i interfejsy mogą dziedziczyć po dowolnej ilości interfejsów.

Klasyczne dziedziczenie zostało usunięte aby uprościć model języka. Możliwość definiowania metod oraz pól na każdym poziomie hierarchi dziedziczenia, powodowałoby nieoczywiste zachowanie przy pobieraniu listy członków klasy. Na przykład, nie jest jasnym czy metoda zwracająca listę pól klasy, powinna uwzględniać pola z klasy bazowej. W ostatecznie wybranym modelu języka, wszystkie pola i metody muszą być zdefiniowane w typie konkretnym, natomiast wcześniej wspomniany problem jest ograniczony do iterowania po metodach interfejsów.

Atrybuty w języku C-=-1 są bardzo podobne do tych które można znaleźć w języku C#. Można nimi adnotować deklaracje typów, zmiennych oraz funkcji. Deklarując atrybut, użytkownik może zaimplementować szereg specjalnych metod, reagujących na użycie adnotowanego elementu. Na przykład dla atrybutu dołączanego do funkcji, użytkownik może napisać metodę która będzie wykonywana za każdym razem kiedy dana funkcja jest wywoływana w kodzie. Wewnątrz tej metody, można wykonywać dowolne transformacje na reprezentacji pośredniej.

III. Proces kompilacji

Projekt procesu kompilacji jest najważniejszą częścią tej pracy. W porównaniu z typowym kompilatorem, w którym wykonywanie kodu jest tylko niewielkim fragmentem całego procesu kompilacji, w C-=-1 interpreter jest głównym modułem kompilatora. Najogólniej, proces kompilacji programu składa się z następujących kroków:

1) Budowa reprezentacji pośredniej.
2) Wykonanie operacji metaprogramistycznych w kodzie.
3) Wykonanie metod atrybutów.
4) Wykonanie kodu w interfejsie backendu.

Reprezentacja pośrednia kodu użytkownika, jest budowana używając struktur danych interpretera. Utrudnia to znacząco pisanie kompilatora, ponieważ efektywnie usuwa z języka

statyczne typowanie. Ta decyzja została podjęta aby uniknąć konieczności konwersji pomiędzy reprezentacją pośrednią widzianą przez kompilator i kod użytkownika oraz synchronizowania tych dwóch struktur danych.

A. *Interpreter*

B. *Kompilator*

IV. Zalety nowego podejścia

V. Możliwa krytyka

References

[1] G. Eason, B. Noble, and I. N. Sneddon, "On certain integrals of Lipschitz-Hankel type involving products of Bessel functions," Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.

[2] J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[3] I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[4] K. Elissa, "Title of paper if known," unpublished.

[5] R. Nicole, "Title of paper with only first word capitalized," J. Name Stand. Abbrev., in press.

[6] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, "Electron spectroscopy studies on magneto-optical media and plastic substrate interface," IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].

[7] M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.