Teoria Kompilacji

Kompilator okrojonej wersji języka C do bytecode'u - dokumentacja

Filip Buszko
Artur Orzeł
III rok Informatyka IEiT

1 Cel projektu

Celem projektu było utworzenie kompilatora okrojonej wersji języka C do bytecode'u maszyny wirtualnej Javy. Nie wystarczyło więc napisanie translatora języka C na Javę. Ważne było jak najwierniejsze odwzorowanie zachowań poszczególnych elementów języka.

2 Rewizja technologii

Za pozwoleniem prowadzącego, użyliśmy biblioteki **pycparser**, służącej do parsowania języka i generowania na podstawie źródła programu struktury drzewiastej (AST). Biblioteka ta wykorzystuje bibliotekę **PLY**, o której wspominaliśmy w wizji.
Dodatkowo, w trakcie projektu okazało się, że generowanie czystego bytecode'u jest bardzo żmudne. Wymaga ogromnego nakładu pracy - zbyt dużego w porównaniu do czasu, jaki mieliśmy na realizację projektu. Na domiar złego, dostępne nam źródła wiedzy nie opisują dostatecznie dokładnie struktury tego kodu. W związku z czym, za zgodą prowadzącego, zdecydowaliśmy się skorzystać z biblioteki **Jasmin** `http://jasmin.sourceforge.net/`, generującej pliki *.class z odpowiednio przygotowanych plików, które - ze względu na większą czytelność - dużo łatwiej wygenerować.

3 Ograniczenia dotyczące języka

Ze względu na niedobór czasu i napotkane trudności zdecydowaliśmy się jeszcze bardziej zmniejszyć zakres obsługiwanych elementów języka C:

- Typy danych - po przemyśleniu projektu stwierdziliśmy, że nie ma sensu obsługiwać wielu typów o podobnej semantyce, ale różnej dokładności (np. short, int, long). W bytecode'zie poszczególne operacje dla różnych typów prymitywnych są realizowane poprzez różne operacje. Ponadto, wykonywanie działań na wartościach o typach o różnej dokładności wymaga jawnego rzutowania z naszej strony. Z tego powodu, liczba przypadków koniecznych do rozpatrzenia urosłaby diametralnie. Dlatego też, typami obsługiwanymi są **int**, **float**, **char** oraz **struktury** i **tablice**.

- Struktury - pominęliśmy możliwość natychmiastowej inicjalizacji struktur na poziomie definicji typu oraz instancji, gdyż uznaliśmy te mechanizmy za niezbyt istotne oraz trudne w kontroli typów.

- Tablice - inicjalizacja na poziomie definicji jest możliwa jedynie dla tablic znaków, poprzez stałe znakowe.

- Enum - początkowo planowaliśmy zaimplementować odpowiednie części naszego programu obsługujące ten typ danych. Był to nawet jedyny element, który został stworzony w ramach generacji czystego kodu byte'owego. Jednak, po przemyśleniach doszliśmy do wniosku, że przeniesienie tego typu przy zachowaniu pełnej semantyki wymagałoby zupełnie innego podejścia - implementacji własnej klasy z odpowiednimi polami typu "int". Takie rozwiązanie byłoby zbyt czasochłonne, zaś zysk z jego tytułu byłby względnie niewielki, stąd z niego zrezygnowaliśmy.

- Switch - JVM wymaga występowania nagłówka "default", w związku z czym jest on tutaj również przez nas wymagany.

- Przesyłanie przez wartość - realizacja przesyłania tablic i struktur przez wartość wymagałaby od nas jawnego kopiowania poszczególnych pól dla obiektów o tych typach. Byłaby to dość czasochłonna w implementacji operacja, którą - ze względu na chęć ukończenia bardziej podstawowych elementów - zdecydowaliśmy się pominąć.

- Wskaźniki - jest to element, o którym kompletnie zapomnieliśmy w naszej wizji. W języku Java brak jest odpowiadającego im typu, bytecode również nie wspiera go w sposób naturalny. Stąd, w celu implementacji tego mechanizmu, musielibyśmy samodzielnie wygenerować odpowiednie klasy-wrappery, które trzymając odniesienia do rzeczywistych obiektów udawałyby wskaźniki. Rozwiązanie to zajęłoby nam jednak bardzo dużo czasu, dlatego najpierw odłożyliśmy jego realizację na koniec projektu, po czym z niego zrezygnowaliśmy z braku czasu.

- Działania na liczbach, instrukcja if-else, pętle, instrukcja switch - działają, zgodnie z wizją.

- Instrukcje skoku - działają jedynie instrukcja **break** wewnątrz instrukcji **switch** oraz instrukcja **return**.

4 Ogólny opis implementacji

Program składa się z trzech plików:

- bytecode_generator.py - znajduje się tutaj klasa BytecodeGenerator, która wykorzystując wzorzec projektowy Visitor analizuje strukturę drzewiastą wygenerowaną przez pycparsera i tworzy pewien zestaw obiektów reprezentujących to, co znajdowało się w pliku źródłowym (definicje struktur, funkcji, itp.) w formie przystępnej do wygenerowania plików dla Jasmina. Ponadto definiuje zestaw wyjątków rzucanych przy błędach napotkanych w trakcie analizy.

- jasmin_translator.py - klasa JasminTranslator w tym pliku odpowiada za generowanie plików *.j wykorzystywanych przez Jasmina do tworzenia plików wykonywalnych JVM.
- main.py - jest skryptem służącym do uruchamiania procesu kompilacji.

5 Inne szczegóły implementacyjne

- Napotkane definicje - są przechowywane jako listy bądź słowniki w klasie BytecodeGenerator, przechowujące zagnieżdzone struktury danych języka python reprezentujące kolejne instrukcje.
- Zmienne globalne - są deklarowane jako statyczne pola publiczne wygenerowanej klasy głównej.
- Struktury - są przetłumaczone na klasy z polami statycznymi, publicznymi.
- Funkcje - odpowiadają publicznym funkcjom statycznym w klasie wynikowej.
- Kontrola typów - dla każdej deklaracji przechowywane są odpowiednie typu danych. Ponadto, w specjalnym polu klasy BytecodeGenerator utrzymywana jest nazwa typu ostatnio wykorzystywanego elementu, dzięki czemu możliwe jest pobranie typu wyniku działania dla dowolnie skomplikowanych operacji, bez sprawdzania rodzaju wywołania.
- Instrukcja return - została zrealizowana poprzez utrzymywanie typu zwracanego przez aktualnie analizowaną funkcję, dzięki czemu można dokonać walidacji i odpowiedniego doboru instrukcji przy zwracaniu wartości.
- Automatyczne rzutowania typów liczbowych - w bytecode'zie wymagane jest ręczne wywoływanie instrukcji rzutujących. Stąd, zdecydowaliśmy się na implementację takiego mechanizmu tylko dla operatorów binarnych. W pozostałych przypadkach należy samodzielnie zadbać o zgodność typów poprzez uprzednie podstawianie wartości pod zmienne tymczasowe. Jest to mało eleganckie, ale dużo nam upraszcza.
- Wartości liczb określających liczbę zmiennych lokalnych oraz rozmiar stosu - nie znaleźliśmy pewnego źródła informacji, jak można pewnie obliczyć te wartości, w związku z czym dla uproszczenia przyjeliśmy stałe wartości: 1000.

6 Błędy zauważone w trakcie testów

Przygotowując dokładniejsze testy kompilatora, zauważyliśmy następujące problemy:

- tablice jednowymiarowe - okazuje się, że nasza implementacja zawiera znaczące błędy powodujące, że obsługiwane w sposób prawidłowy są jedynie tablice dwuwymiarowe
- brak implementacji obsługi pętli do-while - planowaliśmy ją wykonać, ale w pewnym momencie źle się zrozumieliśmy, stąd wynika brak obsługi tej pętli
- ze względu na brak obsługi wskaźników, należy uważać przy stosowaniu zmiennych tymczasowych, np. przy implementacji funkcji "swap"

7 Uruchomienie programu

Aby skompilować kod źródłowy należy przejść do katalogu domowego naszego programu i uruchomić skrypt **main.py** z następującymi parametrami:

- ścieżka do pliku źródłowego
- ścieżka do istniejącego katalogu docelowego, w którym powstaną pliki pośrednie *.j oraz pliki *.class
- nazwa wygenerowanej klasy głównej, zawierającej funkcję "main".