Weryfikacja Bajtkodu Javy

Patryk Czarnik

Wydział Matematyki, Informatyki i Mechaniki UW

11 grudnia 2007

Co teraz?

Weryfikacja bajtkodu w specyfikacji JVM

- 4.8 Constraints on Java Virtual Machine Code
  - 4.8.1 Static Constraints
  - 4.8.2 Structural Constraints
- 4.9 Verification of class Files
  - 4.9.1 The Verification Process
  - 4.9.2 The Bytecode Verifier
  - 4.9.3 Values of Types long and double
  - 4.9.4 Instance Initialization Methods and Newly Created Objects
  - 4.9.5 Exception Handlers
  - 4.9.6 Exceptions and finally

Własności statyczne (1)

Idea

Własności bloku kodu i każdej instrukcji lokalnie.

- Rozmiar tablicy kodu (>0, <65536)
- Poprawny format instrukcji
- Cele skoków – początki instrukcji
- Argumenty `ldc`, odwołania do pól i metod – zgodne typowo wpisy w tablicy stałych (*constant-pool*)
- Operacje na rejestrze nie wychodzą poza rejestr

Własności statyczne (2)

- Wywołania metod:
  - konstruktory tylko przez `invokespecial`
  - metoda `<clinit>` nigdzie nie wołana jawnie
- Instrukcja `new`:
  - odwołanie do pola *constant-pool*reprezentującego klasę (nie tablicę)
  - klasa nie abstrakcyjna
- Instrukcje `newarray`, `anewarray`, `multianewarray`:
  - nie więcej niż 255 wymiarów
  - `newarray` nie do referencji
  - `multianewarray` – nie więcej wymiarów niż opisuje typ

Własności strukturalne (1)

Idea

Zależności między instrukcjami. Mowa o wykonaniu programu.

- Instrukcja wywoływana tylko z odpowiednimi parametrami:
  - na stosie lokalnym
  - w rejestrze
- Dla danej pozycji programu jednakowa głębokość stosu lokalnego niezależnie od przebytej ścieżki
- Nierozdzielne pary w `long` i `double`
- Lokalny stos ograniczony, brak pobierania z pustego
- Dostęp do pól i elementów tablic – przypisaniowa zgodność typów
- `athrow` rzuca `Throwable`
- Sterowanie nie wychodzi poza blok kodu.

Własności strukturalne (2)

- Wywołania metod:
  - `invokespecial` do konstruktorów, metod w bieżącej klasie lub nadklasie
  - konstruktor wywołany na obiekcie niezainicjowanym, odpowiedni stan stosu
  - argumenty o zgodnych typach
- `_return` – typ zgodny z zadeklarowanym
- Wywołania zwykłych metod i dostęp do pól tylko na zainicjowanych obiektach
- Coś z `protected` ...
- Brak niezainicjowanych obiektów:
  - podczas skoku wstecz
  - w rejestrze wewnątrz kodu z zadeklarowaną obsługą wyjątku
- Konstruktor musi wołać konstruktor z tej samej lub nadklasy (z wyjątkiem klasy `Object`)
- Obsługa *subroutines* – typ `returnAddress`, instrukcje `jsr` i `ret`

Standardowy algorytm weryfikacji

- Opisany po angielsku w Specyfikacji JVM
- Zaimplementowany w Sunowskich JRE
- Fazy:
  1. ogólny format pliku class (*magic number*, atrybuty)
  2. analiza pliku class, ale nie kodu metod
  3. analiza kodu metod (w tym analiza przepływu danych)
  4. weryfikacja podczas wykonania
     (m.in. własności wymagające dostępu do innych klas)

Faza 2

- Analiza hierarchii klas:
  - klasy i metody `final` nie są nadpisywane
  - każda klasa (poza `Object`) ma nadklasę
- Analiza *constant-pool*:
  - tylko format

Faza 3

Dla każdej metody (osobno):

1. Podział kodu na instrukcje
2. Sprawdzenie instrukcji i parametrów
   - cele skoków (i ostatnia instrukcja)
   - granice bloków chronionych obsługą wyjątku
   - początki kodu obsługi wyjątków
   - indeksy zmiennych
   - referencje do *constant-pool*
3. Analiza przepływu danych (DFA)

DFA – struktury

Struktury danych

- Dla każdej instrukcji
  - instrukcja
  - typy stosu i rejestru
  - bit „zmieniony”

Inicjalizacja

- Dla pierwszej instrukcji:
  - pusty stos,
  - typy rejestru z sygnatury metody (parametry)
  - „zmieniony” = 1
- Dla pozostałych instruckji:
  - typ stosu i rejestrów nieustalony
  - „zmieniony” = 0

DFA – struktury

Struktury danych

- Dla każdej instrukcji
  - instrukcja
  - typy stosu i rejestru
  - bit „zmieniony”

Inicjalizacja

- Dla pierwszej instrukcji:
  - pusty stos,
  - typy rejestru z sygnatury metody (parametry)
  - „zmieniony” = 1
- Dla pozostałych instruckji:
  - typ stosu i rejestrów nieustalony
  - „zmieniony” = 0

DFA – w pętli

1. Weź instrukcję, dla której „zmieniony” = 1
   - brak – sukces weryfikacji
2. „zmieniony” := 0
3. Sprawdź istnienie i typy argumentów instrukcji
   - niezgodności – porażka weryfikacji
4. Zamodeluj zmiany powodowane przez instrukcję
   - brak miejsca na stosie – porażka weryfikacji
5. Dla każdej możliwej następnej instrukcji
   (uwzględniając następną, cele skoków i obsługę wyjątków)
   - złącz (*merge*) stan po instrukcji ze stanem przy docelowej instrukcji (dla wyjątków z czyszczeniem stosu)
   - „zmieniony” := 1

DFA – złączanie stanów

- Jeden ze stanów nieokreślony – wstawiamy drugi
- Rozmiar stosu musi się zgadzać
- Złączanie pojedynczych typów na stosie:
  - różne rodzaje – porażka weryfikacji
  - dwie referencje – wstawiany kres górny (?)
- Złączanie pojedynczych typów w tablicy:
  - różne rodzaje – wpisany typ „NIE WOLNO CZYTAĆ!”
  - dwie referencje – wstawiany kres górny

DFA – dodatkowe trudności

- Typy `long` i `double`:
  - operacje na parach komórek z zachowaniem kolejności
- Tworzenie obiektów i konstruktory:
  - oznaczanie faktu trwania inicjalizacji – specjalny typ
- Obsługa `finally` (*subroutines*)

Problemy z oryginalną specyfikacją

- *Subroutines* – trudne, a mało przydatne
- Niekonsekwencje w opisie:
  - sprawdzenia podobnego rodzaju: niektóre podczas weryfikacji, inne podczas wykonania
  - sprzeczność w opisie dostępu do niezainicjowanych obiektów
- Własności strukturalne:
  - własność o dobrym typie argumentów,
  - doprecyzowana potem dla różnych (ale nie wszystkich) instrukcji
- Własności niepotrzebnie strukturalne:
  - zgodność typu `_return` z sygnaturą metody,
  - nie wychodzenie poza koniec kodu

Problemy z oryginalną specyfikacją

- Własności bez wpływu na bezpieczeństwo typowe:
  - rozmiar stosu niezależny od ścieżki
  - brak niezainicjowanych obiektów przy skoku wstecz
  - brak rekursji itp. dla *subroutines*
  - `invokespecial` nie do zwykłych metod
- Relacje między typami referencji
  - problem z wersją klasy i *class loaders*
  - kres górny dla interfejsów (wielodziedziczenie) standardowy algorytm bierze `Object`
- Obsługa inicjalizacji obiektu (działa, ale nieelegancka)

Formalizacje z dodatkowymi pomysłami

- Zbiory zamiast pojedynczych typów referencji:
  Allen Goldberg, Zhenyu Quian (niezależnie)
  - istnieje kres górny (zbiór), nie trzeba brać `Object`
- Lekka weryfikacja (Rose&Rose, JavaCard, KVM):
  - pełna weryfikacja *off-device*
  - zapisane wyliczone informacje o typie w miejscach łączenia sterowania
  - jednoprzebiegowa weryfikacja *on-device*
- Wielowariantowa weryfikacja (chyba X.Leroy):
  - pozwala na więcej w przypadku *subroutines*
  - każde wejście do *subroutines* weryfikowane niezależnie
  - możliwa rekurencja

Duże formalizacje

- G.Klein&T.Nipkow(&M.Wildmoser)
  - formalizacja w HOL/Isabelle
  - lekka weryfikacja
  - język μ*Java*
  - subroutines i inicjalizacja obiektów
- S.Freund&J.Mitchell
  - system typów (chyba tylko na papierze)
  - wybrany „reprezentatywny” podzbiór instrukcji
  - subroutines i inicjalizacja obiektów

Co teraz?

Bytecode proofing – idea

- Andreas Gal(i Chrisian W. Probst, i Michael Franz)
  Information and Computer Science, University of California, Irvine
- Motywacja:
  - kwadratowy pesymistyczny koszt standardowego algorytmu weryfikacji
  - możliwość ataku DoS
- Idea – przekształcenie programu

wejście	**wyjście**	**semantyka**
poprawne	poprawne	zgodna
niepoprawne	poprawne	nieokreślona
niepoprawne	brak	N/D

- Koszt liniowy

Fazy algorytmu *Bytecode proofing*

1. Odtworzenie grafu przepływu sterowania
2. Przekształcenie do postaci rejestrowej
3. Przekształcenie do postaci SSA
4. DFA (tanie dzięki SSA)

Odtworzenie grafu przepływu sterowania

- Podział programu na bloki zwn skoki
  - każdy potencjalny cel skoku początkiem bloku
  - każdy potencjalny początek skoku końcem bloku
  - komplikacje w przypadku *subroutines*
- Możliwe przepływy sterowania krawędziami grafu
- Pozwala przechowywać dodatkowe struktury i analizować złączenia tylko w istotnych punktach

Przekształcenie do postaci rejestrowej

1. DFA z jednym typem dla referencji (liniowe)
2. Podział rejestru na części zwn na typ:
   - rozmnożenie rejestru dla każdego z typów bazowych
   - instrukcje używają „swojej” części rejestru
3. Przechowywanie elementów stosu w rejestrze:
   - dodatkowe pozycje rejestru
   - każda instrukcja na stosie obudowana `load` i `store`
   - między takimi blokami pusty stos

SSA – *Single Static Assignment*

- Każda zmienna przypisywana dokładnie raz
- Sprawdzany brak odczytu przed przypisaniem
- Kolejny raz rozmnożenie zmiennych

Co teraz?

1. Weryfikacja bajtkodu
   - Zapewniane własności
   - Standardowy algorytm weryfikacji
   - Słabości oryginalnej specyfikacji
   - Formalizacje
2. Bytecode proofing
   - Idea
   - Algorytm
3. Co można zdziałać?
   - Formalizacja i wykorzystanie własności
   - Formalizacja algorytmu
   - Bytecode proofing

Formalizacja własności – idea

- Formalizacja własności statycznych i strukturalnych
- W dowodach działania programu założenie, że zachodzą
- Prawdopodobnie do dowiedzenia:
  - zweryfikowany program się nie zacina

Formalizacja własności – status

- Wiele własności wynika z formalizacji Bicolano
- Moja formalizacja własności statycznych
- Moja formalizacja własności strukturalnych:
  - typy argumentów instrukcji
  - rozmiar stosu (ograniczony, niezależny od ścieżki)

Formalizacja własności – nie zrobione

- Problemy:
  - Odwołania do *constant-pool*
    (wydają się zapewnione przez Bicolano)
  - Wyrażenie faktu (nie)zainicjowania obiektu
- Z braku czasu / determinacji / . . . :
  - Rozmiar programu
  - Konstruktory
  - `protected`
  - Zgodność przypisaniowa

Formalizacja algorytmu – idea

- Formalizacja w Coqu standardowego algorytmu:
  - raczej w wersji dla Bicolano
  - w szczególności bez *subroutines*
  - być może z poprawkami (*Coglio*)
- Wykorzystanie:
  - dowód własności statycznych i strukturalnych
  - zweryfikowany program się nie zacina (bezpośrednio lub pośrednio)

Formalizacja algorytmu – status

- Szczątkowa implementacja DFA („*state shadow*”)
- Elementy wspólne z *bytecode proofing*
- Pomysł:
  - parametryzacja / opisanie instrukcji JVM
    zwn. typy argumentów i wyniku
  - algorytm DFA dla „meta-instrukcji”
  - dzięki temu łatwość dodawania kolejnych instrukcji JVM

Formalizacja *Bytecode proofing* – idea

- Formalizacja w Coqu algorytmu *bytecode proofing*:
  - raczej w wersji dla Bicolano (bez *subroutines*)
- Dowód poprawności typowej wynikowego programu
- Dowód zachowania semantyki poprawnego programu:
  - jak wyrazić?
  - punkt odniesienia (własności / standardowy algorytm)

Formalizacja *Bytecode proofing* – status

Zrobione

- Rozdzielenie rejestru na rodzaje
- Zamiana stosu na rejestr (dla kilku instrukcji)

Nie zrobione

- Pełna zamiana stosu na rejestr
- DFA
- SSA