Weryfikacja funkcyjności metod Javy

Sławomir Rudnicki

Niezawodność systemów współbieżnych i obiektowych

16 marca 2011

Wprowadzenie

Dla danej metody w Javie sprawdzamy pewne własności wprowadzone przez adnotacje

```
@Pure
public Integer getUniqueNumber() {
    ...
}

@Function
public Integer getSum(List<Integer> x) {
    ...
}
```

Czysta funkcyjność

Metoda czysto funkcyjna (ang. *pure*):

- nie posiada efektów ubocznych, czyli:
- nie modyfikuje stanu obiektów

Czysta funkcyjność

Czy ta metoda jest czysto funkcyjna?

```
@Pure
public Integer getSum(List<Integer> list) {
    Integer s = 0;
    Iterator<Integer> it = list.iterator();
    while (it.hasNext()) {
        s += it.next();
    }
    return s;
}
```

Czysta funkcyjność

Metoda czysto funkcyjna (ang. *pure*):

- nie posiada efektów ubocznych
- nie modyfikuje stanu obiektów istniejących na stercie w momencie jej wywołania

Po co nam czysta funkcyjność?

- Metody czysto funkcyjne mogą być używane w specyfikacji i asercjach:

```
public class Heap {
    public Integer insert(Integer n) {
        ...
        assert(isHeapOK());
    }

    @Pure Boolean isHeapOK() {...};
}
```

Po co nam czysta funkcyjność?

Metody czysto funkcyjne:

- mogą być wywoływane współbieżnie
  - w szczególności niemożliwe są wyścigi o dane (*data race*)
  - w weryfikacji opartej o przeszukiwanie przestrzeni stanów można pominąć analizę przeplotów pomiędzy dwiema metodami czysto funkcyjnymi
- nie naruszają niezmienników systemu
- mogą być wywoływane bez szkody dla integralności pozostałej części systemu, nawet jeśli są niezaufane.

Funkcyjność deterministyczna

Metoda deterministycznie funkcyjna:

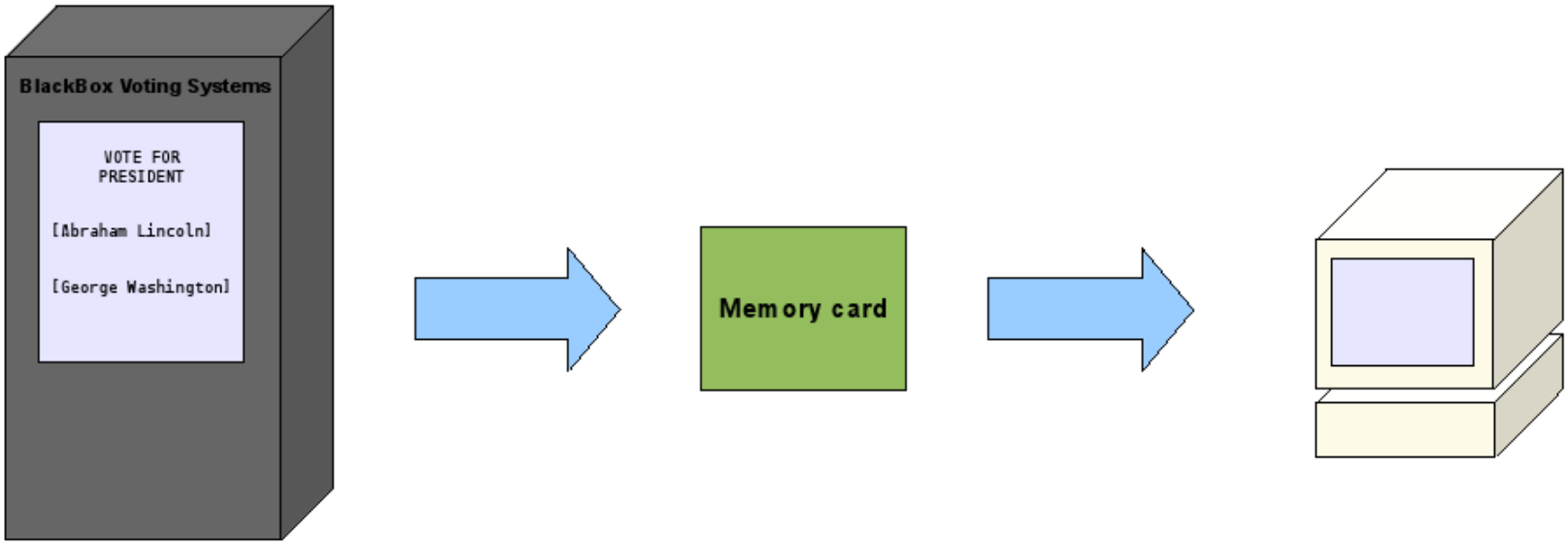
- jest czysto funkcyjna
- wywołana z tymi samymi argumentami, zawsze daje ten sam wynik

Po co nam funkcyjność deterministyczna?

- Metoda funkcyjna może być bezpiecznie wywoływana w środowisku wielowątkowym.
- W asercjach determinizm zapewnia powtarzalność błędów i poprawnych wykonań.
- Wykonanie metody funkcyjnej jest powtarzalne
  - Przydatne w systemach z historią działania.
  - Zapewnia ważne własności bezpieczeństwa.

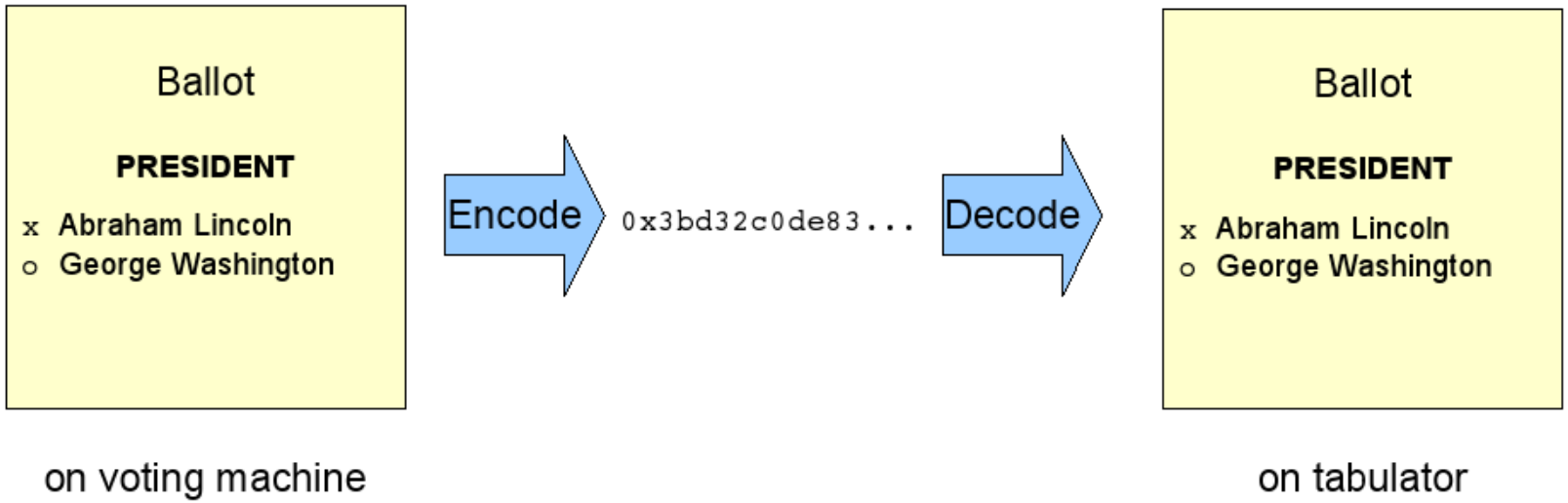
Po co nam funkcyjność deterministyczna?

Elektroniczny system głosowania



Voting machine — Ballot storage — Tabulator

Po co nam funkcyjność deterministyczna?



Jak zapewnić, że głos zostanie poprawnie przeliczony?

`assert(x == decode(y));`

`decode` musi być funkcyjna

Weryfikacja czystej funkcyjności

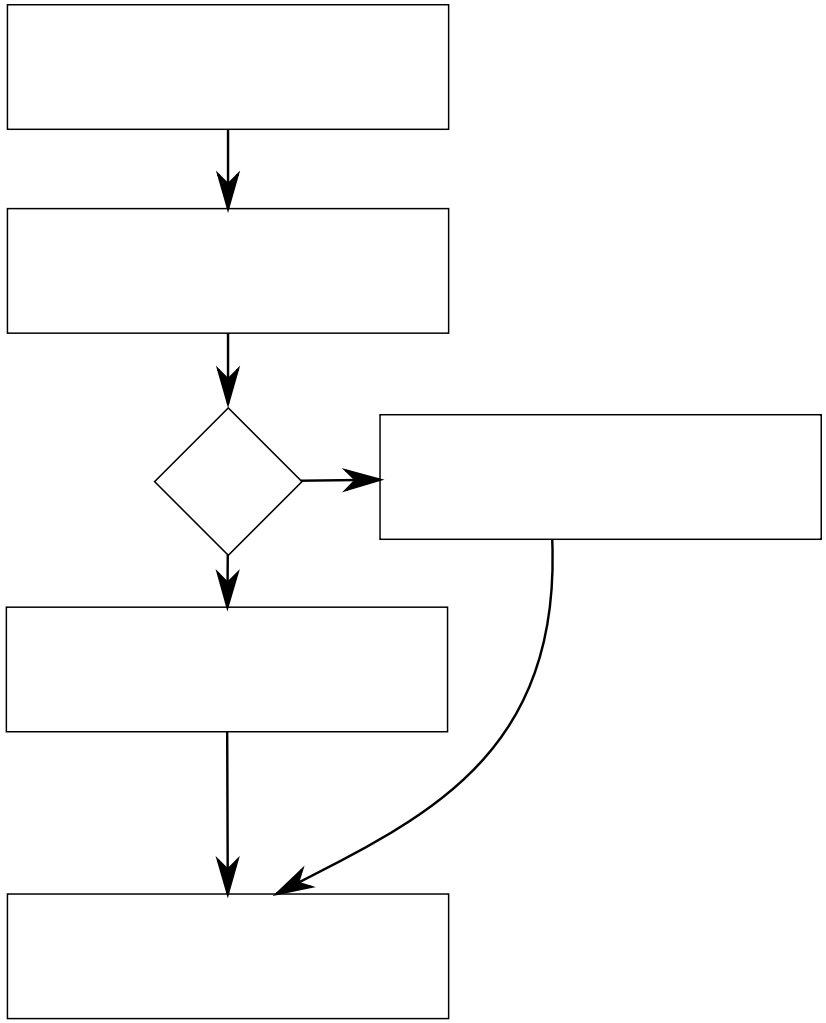
Metoda – **Analiza statyczna**

- dla każdego punktu sterowania budujemy strukturę wskaźników, która reprezentuje część sterty widoczną dla metody do tego punktu.

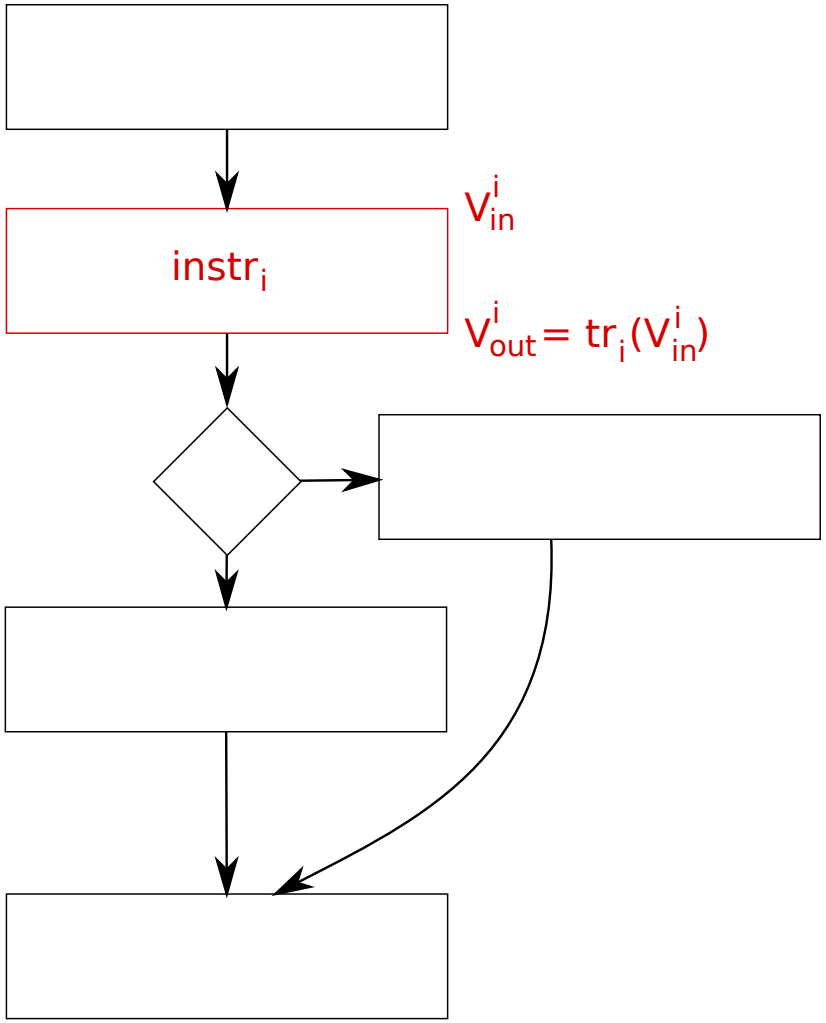
Podstawowe problemy:

- Śledzenie nowo utworzonych obiektów
  - Tylko one mogą być modyfikowane
- Wywołania innych metod
  - Jak zmieniają strukturę?

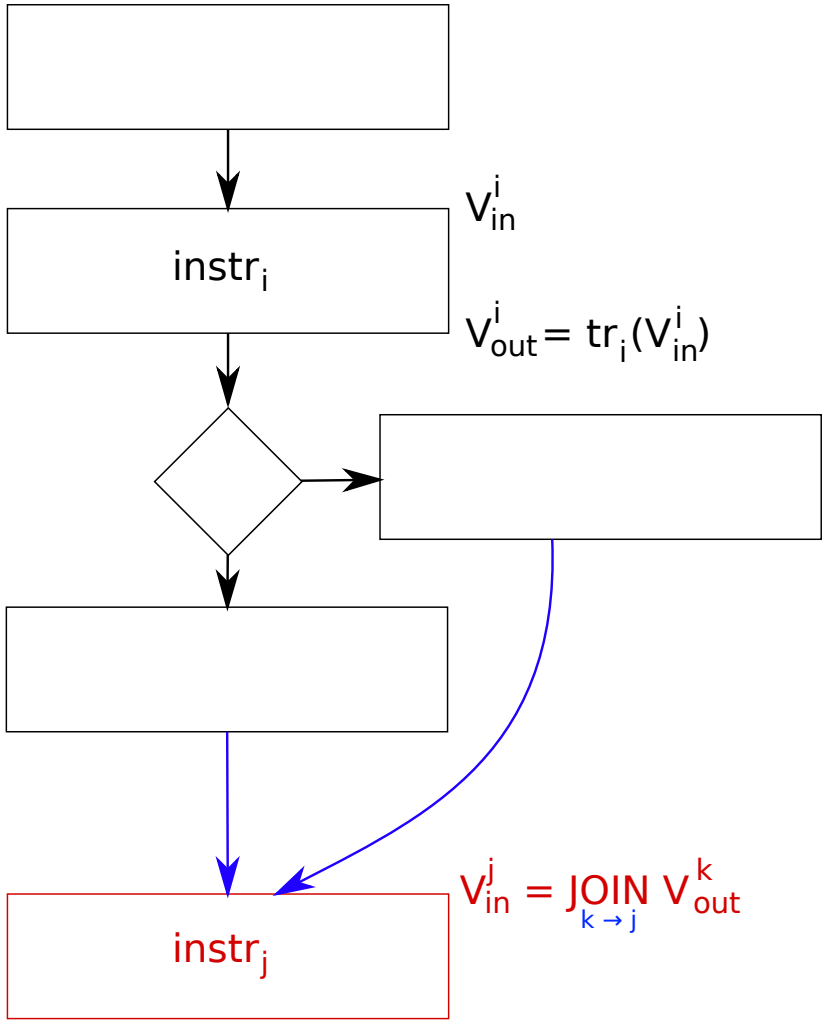
Analiza przepływu danych – ogólnie



Analiza przepływu danych – ogólnie



Analiza przepływu danych – ogólnie



Dla każdego punktu sterowania utrzymujemy:

1. graf wskaźników (ang. *points-to graph*)
2. zbiór pól, które *uciekają* z metody

$$G = \langle I, O, L, E \rangle$$

3. zbiór modyfikowanych pól

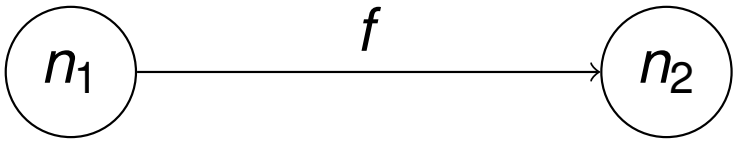
$$W = \mathcal{P}\left(Node \times Field\right)$$

Graf wskaźników: konwencje

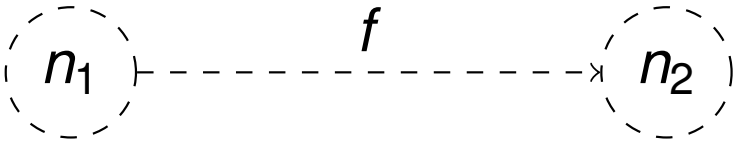
- Węzeł wewnętrzny – utworzony w metodzie

- Węzeł zewnętrzny – odczytany w metodzie

- Krawędź wewnętrzna – utworzona w metodzie:

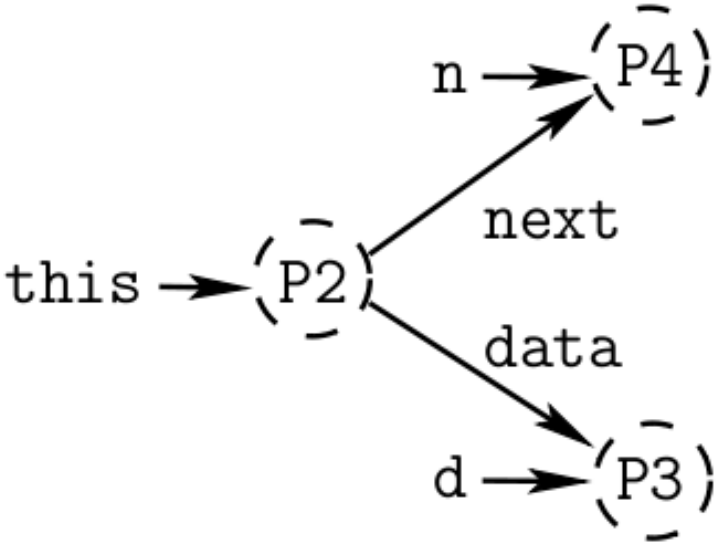


- Krawędź zewnętrzna – odczytana w metodzie:



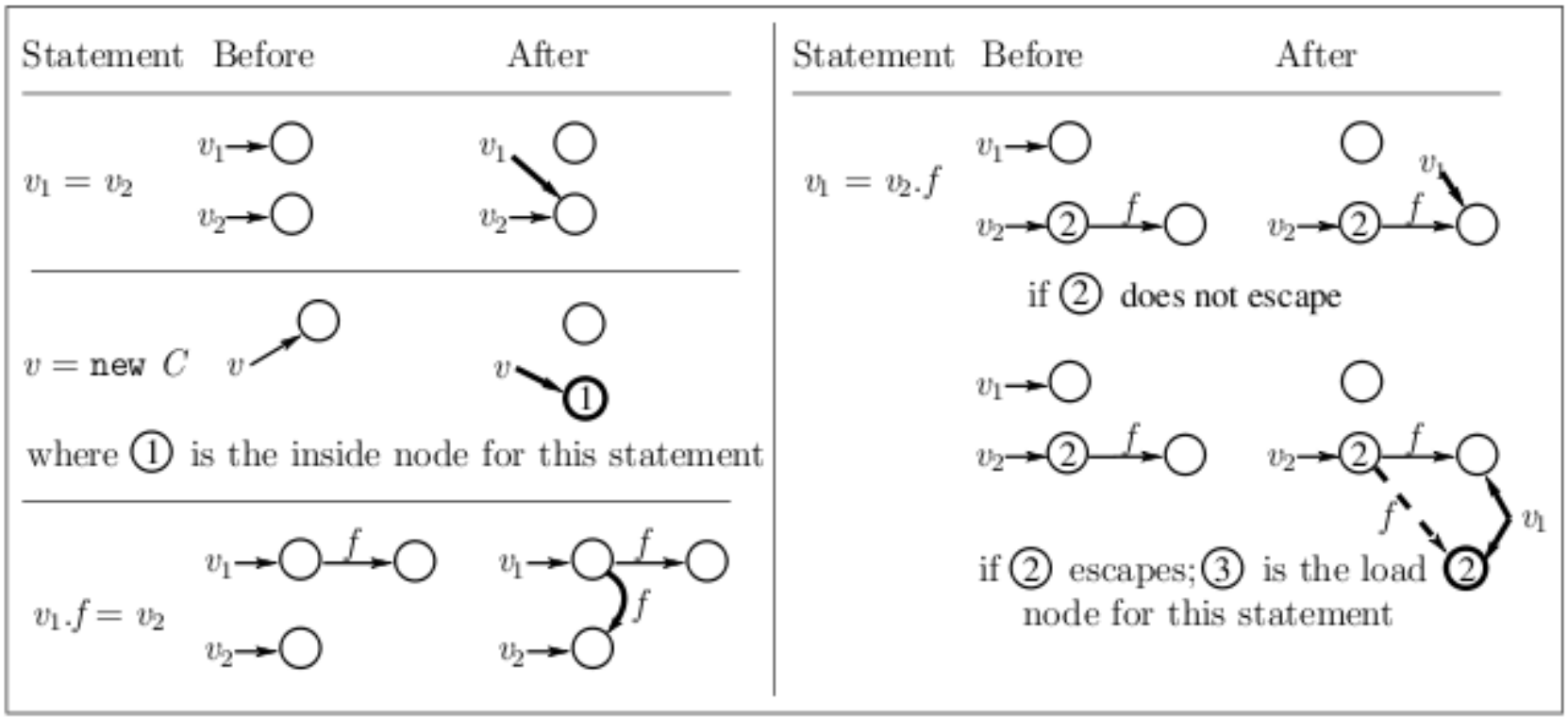
Przykładowy graf wskaźników

```
Cell(Object d, Cell n) {
      data = d;
      next = n;
}
```



$$W = \{\langle P2, \text{data}\rangle, \langle P2, \text{next}\rangle\}$$

Funkcja transferu



Wołanie metod

$$v_R = v_0.s(v_1, \ldots, v_j)$$

- metody nieanalizowalne
  - v_R jest nieznane,
  - v_i uciekają
- metody analizowalne
  1. mapowanie węzłów z grafu metody wołanej
  2. połączenie grafów
  3. uproszczenie grafu wynikowego

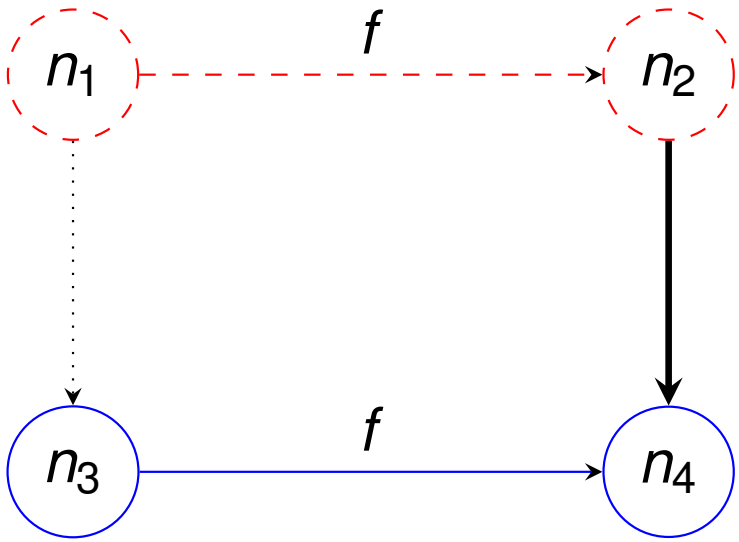
Wołanie metod: mapowanie węzłów

$$\mu' \subseteq \mathit{Node} \times \mathit{Node}$$

- dla węzła n z grafu wskaźników dla metody wołanej, $\mu'(n)$ zawiera odpowiadające mu węzły z grafu po wywołaniu metody.

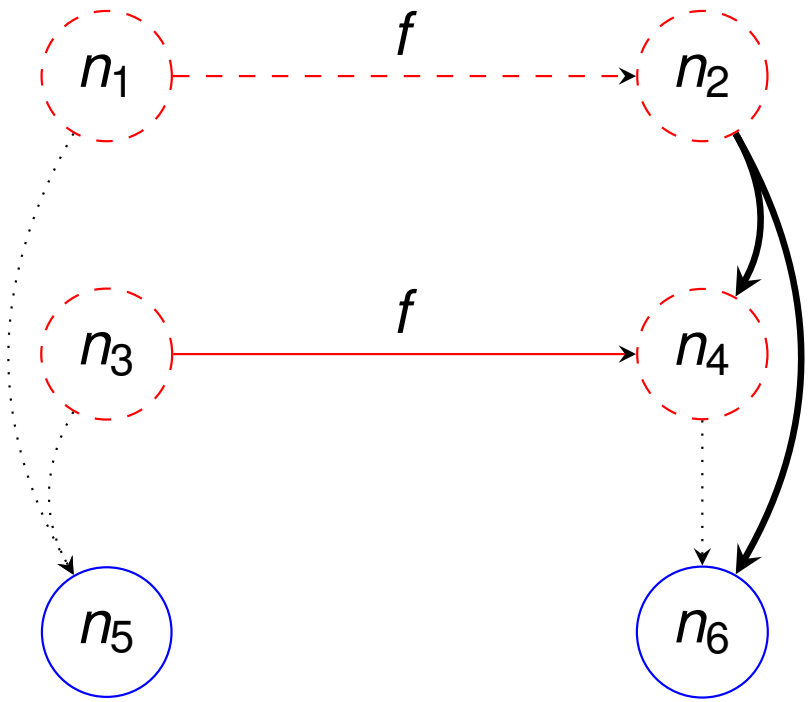
Wołanie metod: konstrukcja mapowania

1. $\forall i \in \{1 \ldots j\} \quad L(v_i) \subseteq \mu\left(n^P_{callee,i}\right)$
2. $$\frac{\langle n_1, f, n_2\rangle \in O_{callee} \quad \langle n_3, f, n_4\rangle \in I \quad n_3 \in \mu(n_1)}{n_4 \in \mu(n_2)}$$



Wołanie metod: konstrukcja mapowania

$$\textbf{3}\quad \frac{\langle n_1, f, n_2\rangle \in O_{callee} \quad \langle n_3, f, n_4\rangle \in I_{callee} \quad (\mu(n_1) \cup \{n_1\}) \cap (\mu(n_3) \cup \{n_3\}) \neq \emptyset}{\mu(n_4) \cup (\{n_4\} \setminus PNode) \subseteq \mu(n_2)}$$



Wołanie metod: konstrukcja mapowania

- iterujemy reguły 1-3 do uzyskania punktu stałego,
- uzupełniamy μ do μ':

$$\mu'(n) = \mu(n) \cup (\{n\} \setminus \mathit{PNode})$$

Wołanie metod: inne szczegóły

- Połączenie grafów:
  - dodajemy krawędzie jak w wołanej metodzie
  - uwzględniamy węzły *uciekające* z metody wołanej
  - do W dodajemy te węzły, na które mapują się pola modyfikowane w wołanej metodzie
- Uproszczenie grafu wynikowego:
  - usuwamy *schwytane* wierzchołki zewnętrzne

Przykład

```
interface Iterator {
      boolean hasNext();
      Integer next();
}
class ListItr implements Iterator {
   ListItr(Cell head) {
      cell = head;
   }
   Cell cell;
   public boolean hasNext() {
      return cell != null;
   }
   public Integer next() {
      Integer result = cell.data;
      cell = cell.next;
   }
```

Przykład

```
class List {
  Cell head = null;
  void add(Integer e) {
    head = new Cell(e, head);
  }
  Iterator iterator() {
    return new ListItr(head);
  }
}
class Cell {
  Cell(Integer d, Cell n) {
    data = d; next = n;
  }
  Integer data;
  Cell next;
}
```

Przykład

Weryfikujemy czystą funkcyjność następującej funkcji:

```
static Integer getSum(List list) {
   Integer s = 0;
   Iterator it = list.iterator();
   while(it.hasNext()) {
         Integer n = it.next();
         s += n;
   }
   return s;
}
```

Wnioski z grafu wskaźników

- **Czysta funkcyjność.** A – zbiór węzłów osiągalnych z parametrów przez krawędzie zewnętrzne, B – zbiór węzłów osiągalnych z E. Metoda jest czysto funkcyjna wtw:

$$A \cap B = \emptyset$$

oraz

$$\forall n \in A \neg \exists f \quad \langle n, f \rangle \in W$$

Wnioski z grafu wskaźników

- **Stałe parametry.** S_1 – osiągalne z parametru po krawędziach zewnętrznych. Parametr jest stały wtw.:

$$\forall n \in S_1 \neg\exists f \quad \langle n, f \rangle \in W$$

- **Bezpieczne parametry.** S_2 – osiągalne z parametru i węzła zwracanego. Parametr jest bezpieczny wtw. jest stały i nie istnieje krawędź wewnętrzna ze zbioru S_2 do zbioru S_1

Wnioski z grafu wskaźników

- **Modyfikowane pola** – z grafu wskaźników tworzymy automat nad alfabetem złożonym z identyfikatorów, który będzie akceptował wszystkie i tylko słowa postaci $v.f_1.f_2.\cdots.f_n$ takie, że wskazane pole jest modyfikowane w metodzie.

Metoda grafu wskaźników – podsumowanie

Zalety

- Dokładne odwzorowanie fragmentu sterty widocznego w metodzie
- Możliwość wnioskowania o innych własnościach metody
- Możliwość wnioskowania własności, a nie tylko ich dowodzenia.

Problemy

- Analiza nie jest modularna – do weryfikacji metody potrzebny jest kod źródłowy metod, które są z niej wywoływane.

JPure

Weryfikacja czystej funkcyjności sprawdzalna modularnie:

- Do weryfikacji danego pliku wystarczy jego kod źródłowy.
- Informacja o funkcyjności metod jest przekazywana pomiędzy plikami źródłowymi przy użyciu adnotacji Javy:

@Pure @Fresh @Local

Naiwne podejście

Sprawdzamy, czy metoda:

- nie modyfikuje bezpośrednio pól obiektów,
- nie woła metod, które nie są czysto funkcyjne,
- nie implementuje lub redefiniuje metody zadeklarowanej jako czysto funkcyjna, sama nie będąc czysto funkcyjną.

Dlaczego jest źle?

Dlaczego jest źle?

```
@Pure
public Integer getSum(List<Integer> list) {
	Integer s = 0;
	Iterator<Integer> it = list.iterator();
	while (it.hasNext()) {
		s += it.next();
	}
	return s;
}
```

Dlaczego jest źle?

Aby **getSum** była czysto funkcyjna, musielibyśmy wiedzieć, że:

- **l.iterator()** jest czysto funkcyjna i zwraca nowy obiekt,

  @Fresh Iterator iterator();

- **it.hasNext()** oraz **it.next()** modyfikują tylko obiekt **it**.

  @Local Integer next();

Adnotacje

- **@Local na polu:** pole jest składnikiem stanu wewnętrznego obiektu.
- **@Local na parametrze** (w szczególności na odbiorcy): tylko stan wewnętrzny parametrów oznaczonych @Local może być modyfikowany.
- **@Pure:** metoda jest czysto funkcyjna.
- **@Fresh:** metoda jest czysto funkcyjna i zwraca nowy obiekt.

Weryfikacja adnotacji

- @Pure:
  - metoda nie modyfikuje nielokalnych parametrów,
  - metoda woła inne metody poprawnie ze względu na lokalność jej parametrów.
- @Fresh: jw., poza tym metoda zwraca nowy obiekt.

Weryfikacja wewnątrzproceduralna przy użyciu uproszczonego grafu wskaźników, przy założeniu, że adnotacje poza właśnie weryfikowaną metodą są poprawne.

Wnioskowanie o czystej funkcyjności

pkg	#Methods	#Pure		#Local		#Fresh	
java.lang	1624	995	(61.2%)	103 / 599	(17.1%)	113 / 520	(21.7%)
java.util.prefs	202	75	(37.1%)	5 / 125	(4.0%)	25 / 80	(31.2%)
java.lang.management	130	105	(80.7%)	0 / 16	(0.0%)	34 / 60	(56.6%)
java.lang.instrument	15	15	(100.0%)	0 / 9	(0.0%)	3 / 5	(60.0%)
java.util.concurrent	525	142	(27.0%)	16 / 242	(6.6%)	15 / 164	(9.1%)
java.util.regex	371	181	(48.7%)	32 / 181	(17.6%)	24 / 70	(34.2%)
java.util	2151	647	(30.0%)	171 / 1043	(16.3%)	108 / 745	(14.4%)
java.util.concurrent.atomic	170	41	(24.1%)	8 / 80	(10.0%)	3 / 21	(14.2%)
java.util.concurrent.locks	98	39	(39.7%)	1 / 35	(2.8%)	8 / 15	(53.3%)
java.io	1017	374	(36.7%)	111 / 491	(22.6%)	22 / 153	(14.3%)
java.util.zip	255	131	(51.3%)	36 / 90	(40.0%)	6 / 23	(26.0%)
java.lang.annotation	17	10	(58.8%)	1 / 8	(12.5%)	2 / 10	(20.0%)
java.util.jar	134	39	(29.1%)	3 / 75	(4.0%)	8 / 44	(18.1%)
java.util.logging	238	49	(20.5%)	8 / 140	(5.7%)	2 / 69	(2.8%)
Total	6947	2843	(40.9%)	495 / 3134	(15.7%)	373 / 1979	(18.8%)

Joe-E

Weryfikacja funkcyjności deterministycznej:

- wyróżniamy podzbiór Javy, który jest deterministycznym językiem obiektowo-uprawnieniowym (ang. *object-capability language*),

Język obiektowo-uprawnieniowy

- Stan systemu jest w całości przechowywany w obiektach,
- Obiekty są dostępne jedynie przez referencję,
- Referencje mogą być przekazywane tylko:
  - jako argumenty metod,
  - przez stan współdzielonych obiektów,

Referencje modelują **uprawnienia** obiektów do korzystania z innych obiektów.

Język obiektowo-uprawnieniowy $\subseteq$ Java

Ten kod nie jest deterministyczny:

```
boolean randomBit() {
    return (new Object().hashCode() % 2) == 1;
}
```

...a ten ma efekty uboczne:

```
void protest() {
    System.err.println("No!");
}
```

Pozwalamy wołać tylko niektóre funkcje z biblioteki standardowej.

Joe-E

Weryfikacja funkcyjności deterministycznej:

- wyróżniamy podzbiór Javy, który jest językiem obiektowo-uprawnieniowym
- deterministyczną funkcyjność metody w takim języku zapewnia:
  - niemutowalność wszystkich jej parametrów i odbiorcy,
  - niemutowalność zmiennych statycznych.

Joe-E – implementacja

- Niemutowalność:
  - wprowadzana przez interfejs,
  - wymuszenie modyfikatora `final` na polach,
  - pola są typu prymitywnego lub są referencją do obiektu typu niemutowalnego,
  - konstruktory nie mogą wołać metod instancyjnych.
- Funkcyjność:
  - niemutowalność wszystkich parametrów

Joe-E w praktyce

Weryfikator Joe-E został użyty do sprawdzenia deterministycznej funkcyjności:

- biblioteki AES
  - szyfrowanie i deszyfrowanie
- implementacji maszyn do głosowania
  - serializacja i deserializacja
- parsera HTML

Joe-E – podsumowanie

Problemy z Joe-E

- Niemutowalność obiektu wymaga niemutowalności wszystkich obiektów z niego osiągalnych,
- Brak wyróżnienia stanu wewnętrznego i zewnętrznego obiektu,
- Niemutowalność definiowana dla klas: brak pojęcia obiektu tylko do odczytu,
- Kryterium czystości deterministycznej jest bardzo silne.

Wniosek:

Na pewno da się zrobić to lepiej!

Bibliografia

- A. Sălcianu, M. Rinard,
  Purity and side-effect analysis for Java programs,
  Proc. VMCAI, 2005, ss. 199–215.
- D. J. Pearce,
  JPure: A modular purity system for Java,
  Proceedings of the Conference on Compiler Construction, 2011
- M. Finifter, A. Mettler, N. Sastry, D. Wagner,
  Verifiable functional purity in Java
  Proc. CCS, ACM 2008, ss. 161–174.

Praca magisterska

Rozwój projektu i pracy można obserwować w publicznym repozytorium:

www.github.com/saf/Funcheck

```
git clone git://github.com/saf/Funcheck.git
```