Wiszące referencje.

Czy można wyeliminować to zagrożenie?

*Andrzej SALWICKI**

*Dombrova Research

W tej pracy przedstawimy groźne zjawisko – błąd wiszących referencji – jakie występuje w programowaniu obiektowym, np. w C++, Pascalu, C. W kolejnym artykule omówimy rozwiązanie pozwalające wyeliminować ten błąd. Zacznijmy od krótkiej ekspozycji problemów, jakie napotykamy podczas zarządzania pamięcią obiektów w każdym języku programowania obiektowego.

Obiekty są 1° tworzone, 2° współdzielone, 3° wykorzystywane i 4° ewentualnie stają się niepotrzebne.

Będziemy tutaj abstrahować od wielu szczegółów, takich jak rozmiar obiektu, jego typ, sposoby zarządzania odzyskiwaną pamięcią, itp.

Obiekty tworzymy, wykonując polecenie x←new(...), które tworzy nowy obiekt o i przypisuje go jako wartość zmiennej x. Np. $\mathsf{u} \leftarrow \underbrace{\mathsf{new}(\mathsf{Re} \leftarrow 3.0, \mathsf{Im} \leftarrow -4.5)}_{\text{nowy obiekt } o}$.

Obiekt o utworzony przez polecenie new może stać się wartością kilku zmiennych. Np. w ten sposób: y←x; ...; z←y. Ma wtedy miejsce współdzielenie (ang. *aliasing, sharing*) obiektu o przez zmienne x,y,z.

Współpracę z obiektem o, który jest wartością zmiennej x, można sprowadzić do trzech przypadków:

- *inspekcja* – ma miejsce wtedy, gdy odczytujemy wartość zmiennej zapisanej w obiekcie, np. trzeba wyznaczyć wartość wyrażenia x.attr.
- *uaktualnienie* – gdy przypisujemy nową wartość zmiennej w obiekcie x, np. x.at←17.
- *serwis* – gdy wykonujemy polecenie wykonania usługi zdefiniowanej w obiekcie, np. call x.NarysujOkrąg(p,12).

Wszystkie trzy przypadki powinny rozpoczynać się od sprawdzenia, czy wartością zmiennej x jest jakiś żywy obiekt, czy też specjalna wartość null.

Na koniec sytuacja, gdy program zaniecha współpracy z obiektem o i gdy przestaje on być wartością jakiejkolwiek zmiennej. Taki obiekt nazywany jest *śmieciem*. Mądry programista pozbędzie się śmieci, uwalniając pamięć zajętą przez te obiekty.

Jakie *problemy* stwarza zarządzanie pamięcią obiektów? Dwa główne zagrożenia dla obliczeń naszego programu to:

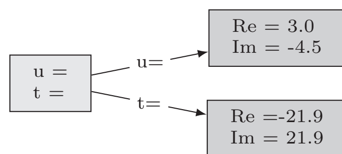
• wiszące referencje i • zaśmiecanie pamięci.

Najwięcej szkód wyrządza błąd wiszących referencji (ang. *dangling reference error*). Można go nazwać *cichym zabójcą*, ponieważ program, w którym usadowił się taki błąd, może wydawać się bezpieczny przez długie miesiące i lata, zanim użytkownicy zorientują się, że program działa nieprawidłowo. Koszty nieprawidłowego działania mogą okazać się bardzo wysokie. (Zdarzało się, że nieprawidłowe działanie programu powodowało nawet śmierć ludzi!) Koszty wykrycia błędu i zlokalizowanie miejsca w programie, w którym ten błąd wystąpił, też nie są bagatelne. Nie umiem podać rzetelnych statystyk. Na pewno jednak tylko finansowe straty w ostatnich trzydziestu latach to kwoty rzędu setek milionów euro. Czy można skonstruować taki program, który analizowałby programy i wykrywał miejsce wystąpienia błędu wiszących referencji w programie?

Niestety, taki algorytm nie istnieje. Wytłumaczenie jest proste: gdyby istniał taki algorytm A, to moglibyśmy skonstruować inny algorytm H, który wykrywałby, czy dowolny dany tekst programu zakończy obliczenia w czasie skończonym, czy też obliczenie będzie przedłużane bez ograniczenia (czyli program zapętli się). Jednakże – jak wiadomo – ten problem jest nierozstrzygalny.

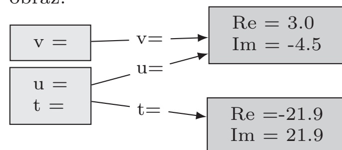
Na nasze potrzeby możemy przyjąć uproszczony obraz obiektów.

Obiekty są prostokątami (pojemnikami, pudełkami, ...). Zmienne występują tylko w obiektach. Wartością zmiennej jest albo liczba, albo obiekt.



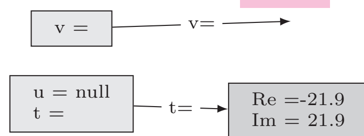
Zmienne t i u wskazują na dwa obiekty.

Po wykonaniu polecenia v←u mamy taki obraz.



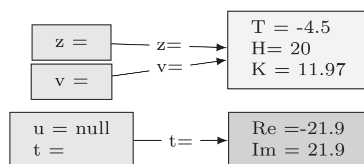
Zmienne u i v wskazują na ten sam obiekt. Wartości wyrażeń u.Im oraz v.Im są równe. Wykonanie polecenia v.Re ← 50 spowoduje, że od tej pory wartość wyrażenia u.Re będzie równa 50. W takim przypadku mówi się o *aliasingu* zmiennych u i v.

Po wykonaniu polecenia delete(u).



Od tej pory obiekt wskazywany przez zmienną u nie istnieje (u=null). A zmiennej v nie odpowiada żaden obiekt! Próba wyznaczenia wartości u.Re lub u.Im spowoduje zgłoszenie błędu. Wartości wyrażeń v.Re i v.Im są fałszywe. Obliczenie jest kontynuowane, bez ostrzeżenia o błędzie!! Na tym polega błąd DANGLING REFERENCE. Groźne jest to, że program nie sygnalizuje błędu. Programista może stracić miesiące na zrozumienie, co zaszło, i wykrycie takiego błędu. Koszty?

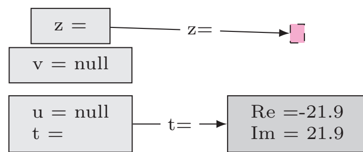
Co się dalej może zdarzyć? Sprzeczna interpretacja danych, gdy na zwolnionym miejscu pojawi się inny obiekt.



Widać to wyraźnie z rysunku: wartość v.Re jest nie tylko fałszywa numerycznie, typy usuniętego obiektu i obiektu z są różne.

Mając to na względzie, twórcy języka programowania Java postanowili: *ponieważ błąd wiszących referencji jest tak groźny i ponieważ nie jest możliwe skonstruowanie kompilatora, który by sygnalizował wystąpienie tego błędu przed uruchomieniem programu, to w języku Java instrukcji* `delete(u)` *nie będzie.* Jak postanowili, tak uczynili.

Inny błąd – *opóźnionej destrukcji* występuje, gdy zostanie wykonane polecenie `delete(v)`.



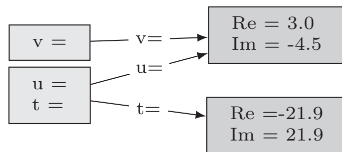
W efekcie usunięto poprawny i potrzebny obiekt `z`, zamiast wcześniej usuniętego obiektu `u`.

Ale czyniąc tak, naśladują pewnego króla Albanii.

> Rzecz działa się dawno, ponad sto lat temu. Pomiędzy stolicą Tiraną i portem Dürres istniała jedyna w kraju linia kolejowa, długa na 40 kilometrów i opadająca stromo w dół. Otóż jednego dnia król wydał takie rozporządzenie: *Ponieważ na linii kolejowej często zdarzają się katastrofy i ponieważ w katastrofach tych wykolejeniu ulega ostatni wagon, przeto zarządzam, by od dziś do pociągu nie doczepiać ostatniego wagonu.*

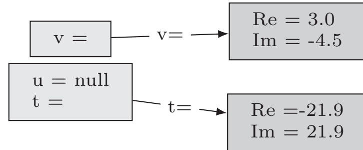
W przypadku Javy okazało się, że na programistę uwolnionego od potwora Charybdy czeka inny potwór Scylla (pamiętasz, Czytelniku, jak Odyseusz żeglował pomiędzy tymi potworami?). Okazało się, że programy w Javie uwolnione od zagrożenia błędem wiszących referencji wpadły w pułapkę *zaśmiecania pamięci* (ang. *memory leakage*).

Jak powstają obiekty śmieci?



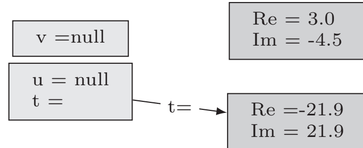
Twórcy Javy zaoferowali programistom narzędzie znane od dawna: *odśmiecacz* (ang. *garbage collector*). Algorytm ten uruchomiony podczas wykonywania programu potrafi zlokalizować wszystkie istniejące obiekty-śmieci i je usunąć. Odzyskaną pamięć można przeznaczyć na nowe obiekty. Koszt odśmiecania nie jest bagatelny, ale pozbywamy się zagrożenia wiszącymi referencjami (twierdzili) i dajemy Ci narzędzie pozwalające kontrolować koszt pamięciowy Twego programu. Ty określisz, co jest śmieciem, a my pozbędziemy się wszystkich śmieci.

Po wykonaniu polecenia `u←null`



Zapanowała euforia, ale trwała krótko. Niebawem okazało się, że mnożą się sytuacje podobne do trzeciego rysunku po lewej stronie. Programista wykonał polecenie `u←null`, ale zapomniał, że obiekt *o* jest nadal wskazywany przez zmienną `v`. Odśmiecacz nie ma podstaw, by go usunąć! Zresztą, nie zawsze błąd taki leży po stronie programisty. Programiści wykorzystują klasy napisane przez kogoś innego, a tam może ukrywać się polecenie podobne do `v ← u` z naszego przykładu.

Po wykonaniu `v←null`



Na tym obrazku pojawił się **śmieć**! Wskaż go!.

Podsumujmy sytuację:

Charybda	*Wiszące referencje* stanowią realne zagrożenie.	Nie istnieje algorytm wykrywania takiego błędu w kodzie źródłowym programu.
Scylla	Zjawisko *zaśmiecania pamięci* zagraża zmniejszeniem szybkości wykonywania obliczeń, a nawet zablokowaniem pracy programu, gdy zabraknie miejsca na kolejny nowy obiekt.	Odśmiecacz jest bezsilny, gdy w trakcie wykonywania programu obiekt już niepotrzebny nadal jest wartością jakiejś (zapomnianej) zmiennej.

Co pozostaje? Programiści Javy mogą korzystać z efemerycznych *słabych referencji*. Pojęcie to pojawiło się trzy lata później niż Java. Nie zyskało uznania i w najnowszej wersji Javy pozostał po nim tylko ślad. Programistom w C++ oferowane są różne narzędzia do monitorowania pracy programu (np. *debugging*).

Zobacz str. 374 w Java Language Specification v. 8

A przecież ponad 35 lat temu Antoni Kreczmar (1945–1996), profesor informatyki Uniwersytetu Warszawskiego, wynalazł *bezpieczny i tani* (w eksploatacji) system przekształcania błędów wiszących referencji w ostrzeżenia o ich obecności. Przypomnijmy sytuację na obrazku z poprzedniej strony, trzecim od góry. Błędem (groźnym!) jest korzystanie ze zmiennej `v`, ponieważ zmienna ta wskazuje na nieistniejący już obiekt. Natomiast próba

wykorzystania zmiennej u zakończy się ostrzeżeniem: *referencja wskazuje na null.* Ostrzeżony programista może łatwo odnaleźć błąd i (z pewnym trudem) go naprawić.

Zadanie, jakie postawił sobie Kreczmar:

1) Zapewnić prawdziwość wszystkim formułom postaci

$$\underbrace{((f_1 = \cdots = f_k) \wedge f_1 \neq \mathbf{none})}_{\text{warunek początkowy}} \Rightarrow \underbrace{[\mathsf{kill}(f_i)]}_{\text{instrukcja}} \underbrace{(f_1 = \cdots = f_k = \mathbf{none})}_{\text{warunek końcowy}}$$

   Formułę tę należy czytać tak: *jeśli na pewien obiekt o wskazuje k zmiennych* $f_1, \ldots f_k$, *to po wykonaniu instrukcji* $\mathsf{kill}(f_i)$ *wszystkie zmienne przyjmują wartość* **none**, a miejsce zajmowane przez obiekt zostaje zwrócone do puli wolnych miejsc.
2) Wykonanie instrukcji kill ma zająć tyle samo czasu, niezależnie od liczby k zmiennych współdzielących obiekt.
3) Koszt sprawdzenia, czy obiekt jest żywy, ma być stały $O(1)$ i niewielki. Ma to znaczenie, ponieważ sprawdzenie takie dokonuje się przy każdym dostępie do obiektu.

Zajrzyj do hasła Antoni Kreczmar w Wikipedii.

Kreczmar nie tylko rozwiązał ten problem, ale także był bardzo ważnym współtwórcą języka programowania Loglan'82 i zaprogramował maszynę wirtualną tego języka.

Parę słów o projekcie badawczym Loglan

Prace nad językiem zostały podjęte w roku 1977 w Zakładzie Teorii Obliczeń Instytutu Informatyki UW. Rok później Zjednoczenie MERA, producent minikomputerów Mera 400, zawarło z nami umowę na stworzenie języka programowania obiektowego i współbieżnego Loglan oraz na zaprogramowanie kompilatora tego języka na komputery Mera 400. Otrzymaliśmy niezbyt dużą kwotę pieniędzy (rzędu 40 milionów ówczesnych złotych, z czego większą część pochłonął zakup 2 komputerów Mera 400), i co najważniejsze, dwa komputery do dyspozycji Instytutu. Prace zakończyły się powodzeniem: opublikowaliśmy raport zawierający definicję języka Loglan, przekazaliśmy kompilator Loglanu dla komputerów Mera 400, zorganizowaliśmy dwie konferencje międzynarodowe (1983, 1984), szkołę PTI nt. Loglanu w 1985, nawiązaliśmy owocną współpracę z Uniwersytetami w Kilonii i w Rzymie, ...

Wiele informacji o języku Loglan, jego kompilatorach dla Linuxa i Windows (działają do dziś!), o problemach badawczych sformułowanych i rozwiązanych w związku z Loglanem znajdziesz w repozytorium:
`lem12.uksw.edu.pl`

Pragnę podkreślić, że:

- W projekcie uczestniczyło kilkanaście osób z mojego Zakładu, wiele z nich to dziś profesorowie uczelni w Polsce, Niemczech, Kanadzie, USA, Meksyku, Szwecji, ... Niestety, nie ma dziś z nami Antoniego Kreczmara.
- Bardzo cenny wkład wnieśli doktoranci i studenci. Student Bolek Ciesielski (1988) wymyślił i zrealizował nowy protokół współpracy pomiędzy obiektami procesów – *alien call.* Jego odkrycie wciąż czeka na uznanie szerszej publiczności.
  Doktorant Oskar Świda (1996) zrealizował koncepcję łączenia loglanowskich maszyn wirtualnych w rozproszony, sieciowy klaster loglanowski.
  Z braku miejsca nie wspomnę o wielu innych osobach.
- Lista problemów badawczych, które sformułowano i rozwiązano, by uzyskać odpowiednią jakość języka i jego kompilatora, zawiera dziewięć pozycji. Tu omówiliśmy problem zapobiegania błędowi wiszących referencji.
- Loglan świetnie nadaje się jako platforma nauczania programowania, ponieważ zawiera prawie wszystkie mechanizmy programowania obiektowego i rozproszonego.
- Loglan jest dobrym punktem wyjścia do prowadzenia badań nad kolejnymi problemami, np. jak zarządzać obliczeniami w komputerze wieloprocesorowym. Żaden z obecnie stosowanych języków programowania nie ma odpowiednich do tego narzędzi.

Jak to rozwiązać?

Czy potrafisz odgadnąć, jak działa system zarządzania pamięcią obiektów zbudowany przez Antoniego Kreczmara? Pamiętasz wymagania? Spróbuj swych sił! Jeżeli wymyślisz **coś lepszego**, to – Autor obiecuje – otrzymasz nagrodę 50 EUR.