Weryfikacja Oprogramowania

semestr zimowy 2014/2015

Grzegorz Herman

Informatyka Analityczna
tcs@jagiellonian

Warunki zaliczenia

Punkty

- 5 zadań na Satori po 4 punkty każde
- egzamin: 3 z 4 pytań, po 6 punktów każde

Ocena

- z ćwiczeń: Satori
- końcowa: wszystko

Progi

$\leqslant$50%	2.0
50–60%	3.0
60–70%	3.5
70–80%	4.0
80–90%	4.5
$>$90%	5.0

Bonus

- implementacja metody z wykładu – podwyższenie oceny

Plan ćwiczeń

C/C++

- podstawowe narzędzia
- testy w modelu „black-box”
- analiza programów wielowątkowych

Java

- unit testing
- mock objects
- pokrycie kodu, mutation testing
- język specyfikacji JML
- instrumentacja

Testowanie UI

- web-based UI
- desktop UI

Plan wykładów

Wstęp

- grafowe reprezentacje programów

Część 1: analiza dynamiczna

- techniki instrumentacji
- wykrywanie data races
- analiza wpływu

Część 2: jakość i generowanie testów

- mutation testing
- testy pokrywające ścieżkę/punkt
- generowanie testów strukturalnych

Plan wykładów (cd.)

Część 3: analiza statyczna

- wnioskowanie oparte o type inference
- analiza wskazywania
- przekroje

Część 4: model checking

- logika Hoare'a
- logiki temporalne
- algorytmy model checking

$\Rightarrow$

- wskazówki
- optymalizacje

Analiza statyczna

- nie uruchamia programu
- ogólne własności programu
- więcej informacji
- trudniejsza

Analiza dynamiczna

- uruchamia program
- konkretny przebieg programu
- mniej informacji
- (względnie) prostsza

$\Leftarrow$

- feedback

$\Rightarrow$

- wskazówki
- optymalizacje

Analiza statyczna

- nie uruchamia programu
- ogólne własności programu
- więcej informacji
- trudniejsza

Analiza dynamiczna

- uruchamia program
- konkretny przebieg programu
- mniej informacji
- (względnie) prostsza

$\Leftarrow$

- feedback

Wejście: kod źródłowy

Założenia

- język imperatywny
- pojedynczy wątek
- determinizm
- pojedyncza funkcja (analiza *intra*proceduralna)

Uproszczenia wstępne

- dekonstrukcja struktur wysokiego poziomu
- przepływ sterowania zamieniony na skoki warunkowe

```
int ten() {
  int i;
  for (i=0; i<10; ++i);
  return i;
}
```

⇒

```
  i = 0;
checkfor:
  if (i>=10) goto endfor;
  ++i;
  goto checkfor;
endfor:
  return i;
```

Wejście: kod źródłowy

Założenia

- język imperatywny
- pojedynczy wątek
- determinizm
- pojedyncza funkcja (analiza *intra*proceduralna)

Uproszczenia wstępne

- dekonstrukcja struktur wysokiego poziomu
- przepływ sterowania zamieniony na skoki warunkowe

```
int ten() {
  int i;
  for (i=0; i<10; ++i);
  return i;
}
```

$\Rightarrow$

```
  i = 0;
checkfor:
  if (i>=10) goto endfor;
  ++i;
  goto checkfor;
endfor:
  return i;
```

Control Flow Graph

Basic block

- liniowy ciąg instrukcji
- skoki „na zewnątrz” tylko z ostatniej
- skoki „do wewnątrz” tylko do pierwszej

Control Flow Graph $G = (V, E, s, t)$

- V – zbiór basic blocks
- $E \subseteq V \times V$ – możliwy przepływ sterowania
- $s \in V$ – instrukcja wejściowa
- $t \in V$ – instrukcja wyjściowa

Regularyzacja

- każdy $v \in V$ osiągalny z s
- t osiągalny z każdego $v \in V$

Control Flow Graph

Basic block

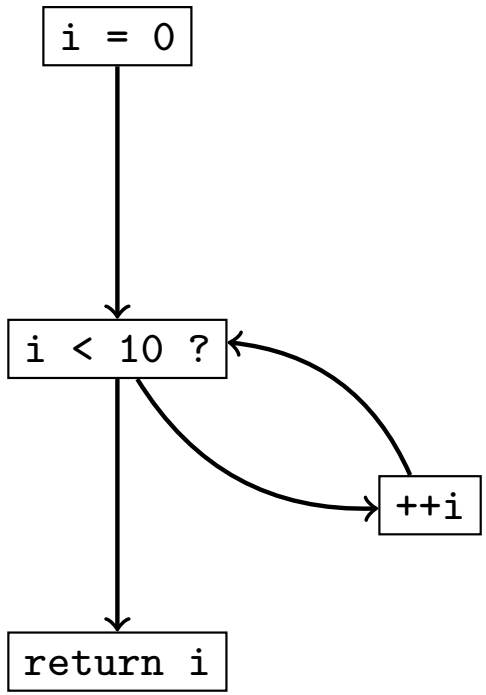
- liniowy ciąg instrukcji
- skoki „na zewnątrz” tylko z ostatniej
- skoki „do wewnątrz” tylko do pierwszej

Control Flow Graph $G = (V, E, s, t)$

- V – zbiór basic blocks
- $E \subseteq V \times V$ – możliwy przepływ sterowania
- $s \in V$ – instrukcja wejściowa
- $t \in V$ – instrukcja wyjściowa

Regularyzacja

- każdy $v \in V$ osiągalny z s
- t osiągalny z każdego $v \in V$



Dominacja

u dominuje v

gdy każda ścieżka z s do v przechodzi przez u

u bezpośrednio dominuje v

gdy dodatkowo u nie dominuje żadnego innego dominatora v

bezpośrednie dominatory

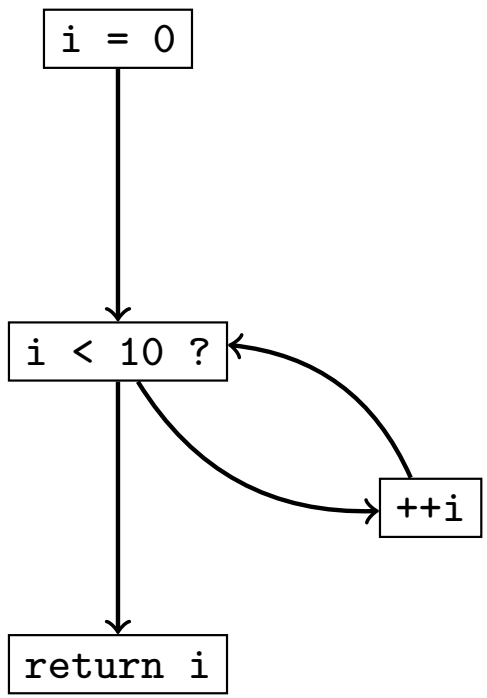
- tworzą drzewo o korzeniu w s
- można wyznaczyć w czasie prawie liniowym

$v \in$ dominance frontier(u)

- $u \rightsquigarrow w \longrightarrow v$
- u dominuje w
- u nie dominuje v

postdominacja

to dominacja od t po odwróconych krawędziach



Dominacja

u dominuje v

gdy każda ścieżka z s do v przechodzi przez u

u bezpośrednio dominuje v

gdy dodatkowo u nie dominuje żadnego innego dominatora v

bezpośrednie dominatory

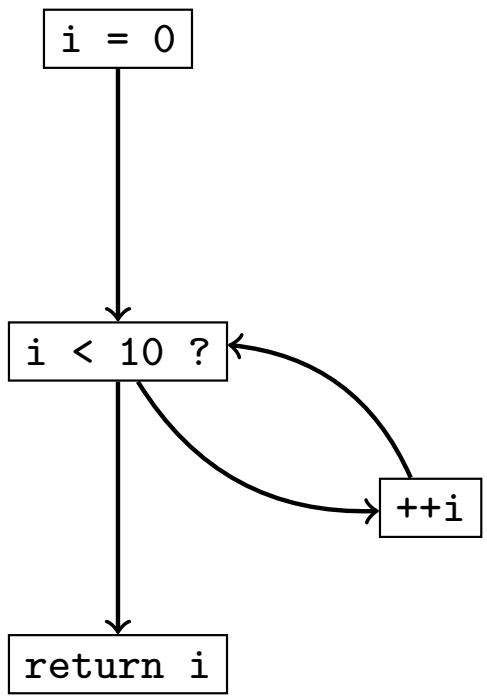
- tworzą drzewo o korzeniu w s
- można wyznaczyć w czasie prawie liniowym

$v \in$ dominance frontier(u)

- $u \rightsquigarrow w \longrightarrow v$
- u dominuje w
- u nie dominuje v

postdominacja

to dominacja od t po odwróconych krawędziach



Dominacja

u dominuje v

gdy każda ścieżka z s do v przechodzi przez u

u bezpośrednio dominuje v

gdy dodatkowo u nie dominuje żadnego innego dominatora v

bezpośrednie dominatory

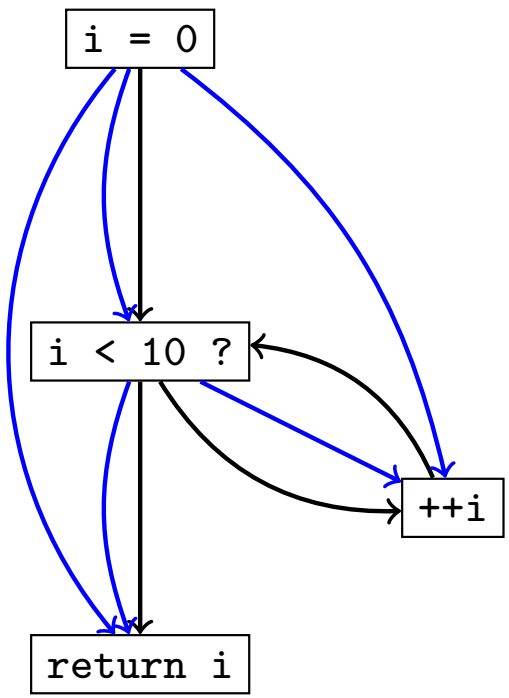
- tworzą drzewo o korzeniu w s
- można wyznaczyć w czasie prawie liniowym

$v \in$ dominance frontier(u)

- $u \rightsquigarrow w \longrightarrow v$
- u dominuje w
- u nie dominuje v

postdominacja

to dominacja od t po odwróconych krawędziach



Dominacja

u dominuje v

gdy każda ścieżka z s do v przechodzi przez u

u bezpośrednio dominuje v

gdy dodatkowo u nie dominuje żadnego innego dominatora v

Bezpośrednie dominatory

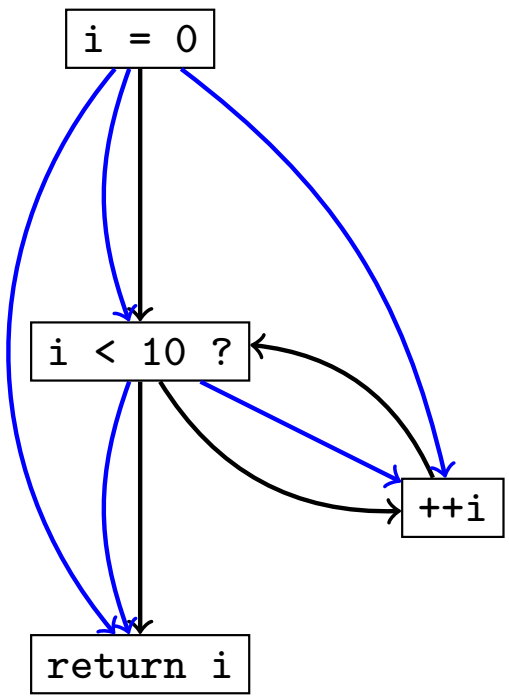
- tworzą drzewo o korzeniu w s
- można wyznaczyć w czasie prawie liniowym

$v \in$ dominance frontier(u)

- $u \rightsquigarrow w \longrightarrow v$
- u dominuje w
- u nie dominuje v

Postdominacja

to dominacja od t po odwróconych krawędziach



Dominacja

u dominuje v

gdy każda ścieżka z s do v przechodzi przez u

u bezpośrednio dominuje v

gdy dodatkowo u nie dominuje żadnego innego dominatora v

bezpośrednie dominatory

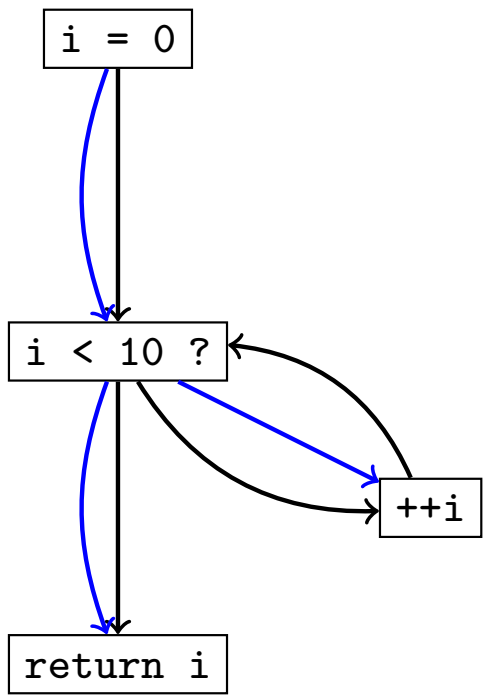
- tworzą drzewo o korzeniu w s
- można wyznaczyć w czasie prawie liniowym

$v \in$ dominance frontier(u)

- $u \rightsquigarrow w \longrightarrow v$
- u dominuje w
- u nie dominuje v

postdominacja

to dominacja od t po odwróconych krawędziach



Dominacja

u dominuje v

gdy każda ścieżka z s do v przechodzi przez u

u bezpośrednio dominuje v

gdy dodatkowo u nie dominuje żadnego innego dominatora v

bezpośrednie dominatory

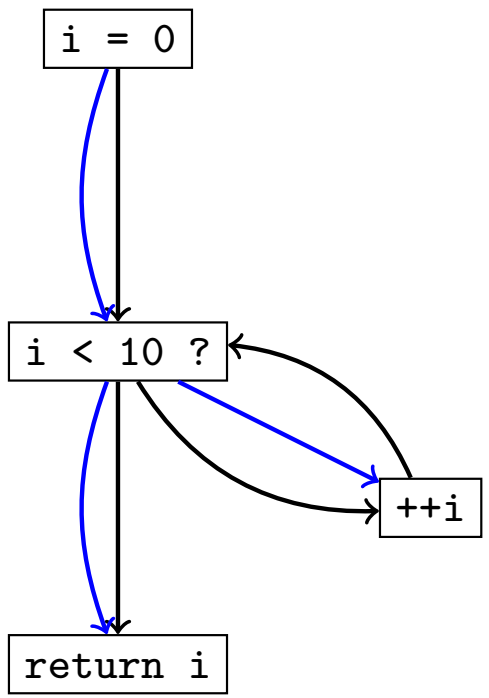
- tworzą drzewo o korzeniu w s
- można wyznaczyć w czasie prawie liniowym

$v \in$ dominance frontier(u)

- $u \rightsquigarrow w \longrightarrow v$
- u dominuje w
- u nie dominuje v

postdominacja

to dominacja od t po odwróconych krawędziach



Dominacja

u dominuje v

gdy każda ścieżka z s do v przechodzi przez u

u bezpośrednio dominuje v

gdy dodatkowo u nie dominuje żadnego innego dominatora v

bezpośrednie dominatory

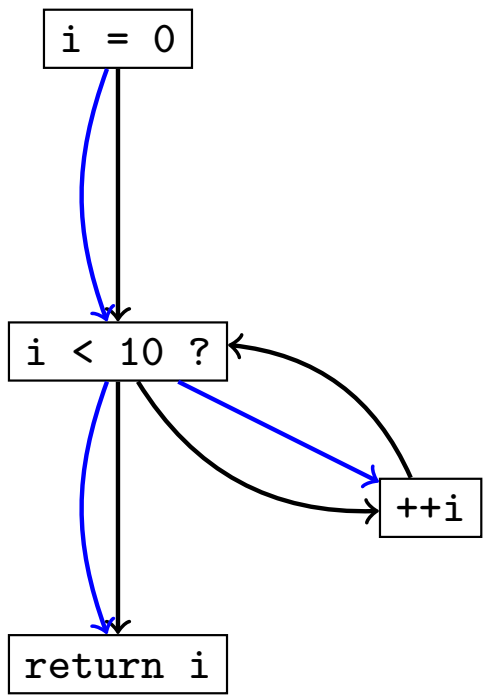
- tworzą drzewo o korzeniu w s
- można wyznaczyć w czasie prawie liniowym

$v \in$ dominance frontier(u)

- $u \rightsquigarrow w \longrightarrow v$
- u dominuje w
- u nie dominuje v

postdominacja

to dominacja od t po odwróconych krawędziach



Dominacja

u dominuje v

gdy każda ścieżka z s do v przechodzi przez u

u bezpośrednio dominuje v

gdy dodatkowo u nie dominuje żadnego innego dominatora v

Bezpośrednie dominatory

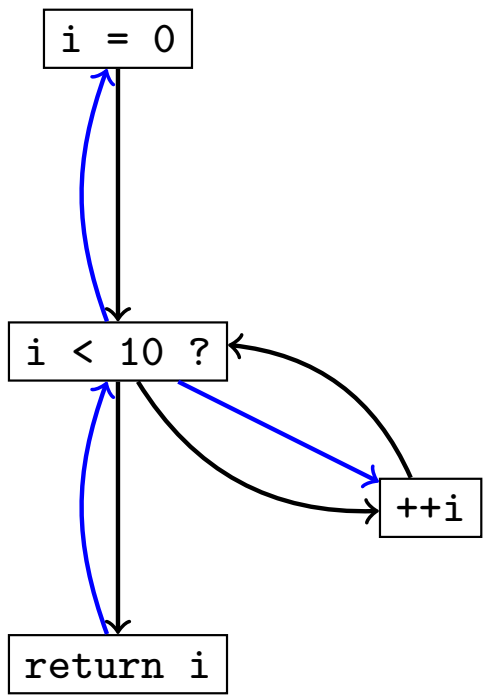
- tworzą drzewo o korzeniu w s
- można wyznaczyć w czasie prawie liniowym

$v \in$ dominance frontier(u)

- $u \rightsquigarrow w \longrightarrow v$
- u dominuje w
- u nie dominuje v

Postdominacja

to dominacja od t po odwróconych krawędziach



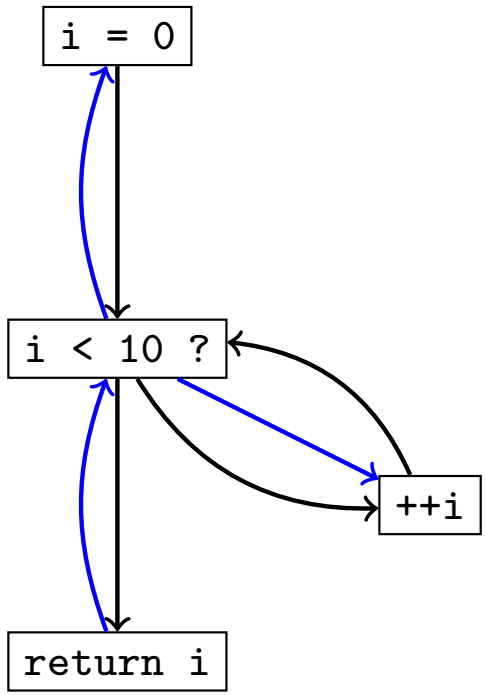
Control Dependence Graph

v control-depends on u

- $\exists u \longrightarrow w \rightsquigarrow v$
- v post-dominuje w (lub $v = w$)
- v nie post-dominuje u

Intuicja

- u ma przynajmniej 2 wyjścia
- jedno z nich zawsze prowadzi do v
- drugie nie



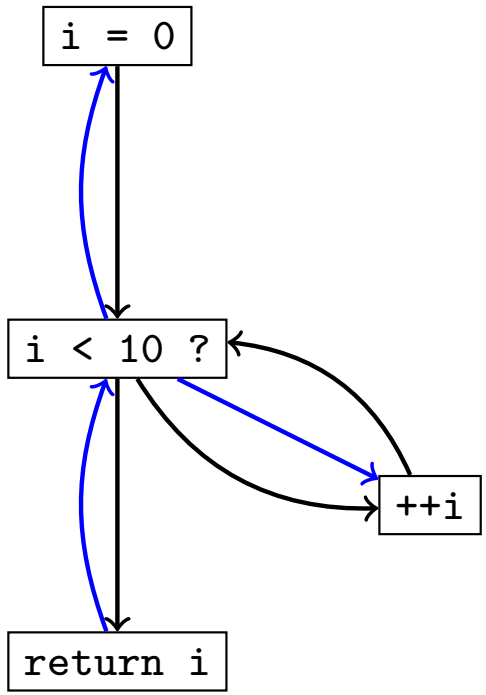
Control Dependence Graph

v control-depends on u

- $\exists u \longrightarrow w \rightsquigarrow v$
- v post-dominuje w (lub $v = w$)
- v nie post-dominuje u

Intuicja

- u ma przynajmniej 2 wyjścia
- jedno z nich zawsze prowadzi do v
- drugie nie



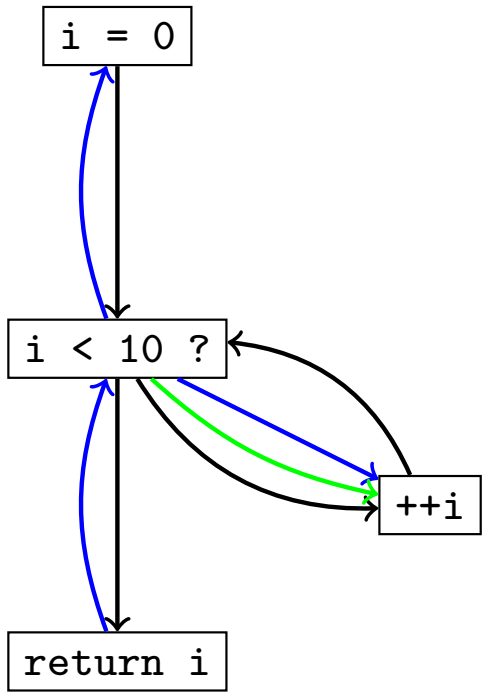
Control Dependence Graph

v control-depends on u

- $\exists u \longrightarrow w \rightsquigarrow v$
- v post-dominuje w (lub $v = w$)
- v nie post-dominuje u

Intuicja

- u ma przynajmniej 2 wyjścia
- jedno z nich zawsze prowadzi do v
- drugie nie



Data Flow (Dependence) Graph

Definicja zmiennej x

to instrukcja ustawiająca x

Użycie zmiennej x

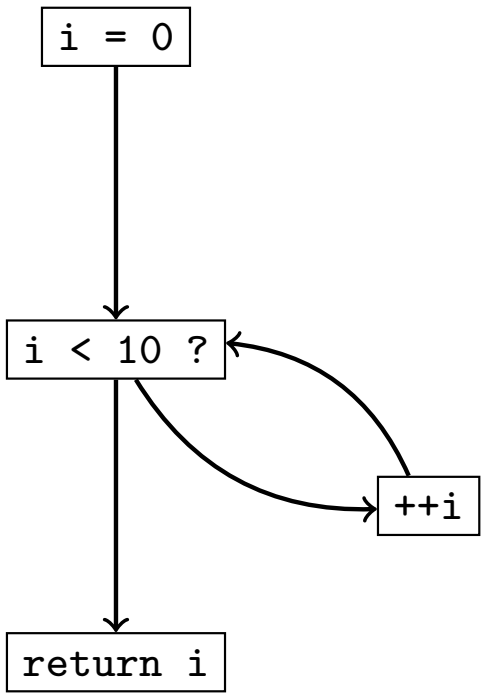
to instrukcja odczytująca x

Ścieżka wolna dla x

to ścieżka w CFG omijająca definicje x

Krawędź $u \to v$ w DFG

- u – definicja x
- v – użycie x
- $\exists u \rightsquigarrow v$ wolna dla x



Data Flow (Dependence) Graph

Definicja zmiennej x

to instrukcja ustawiająca x

Użycie zmiennej x

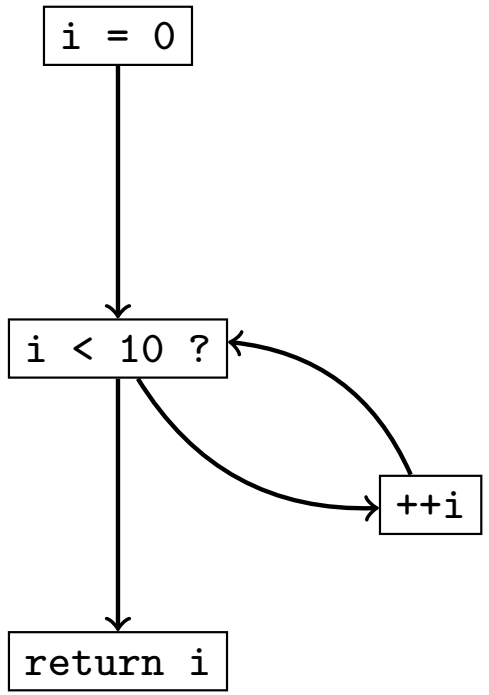
to instrukcja odczytująca x

Ścieżka wolna dla x

to ścieżka w CFG omijająca definicje x

Krawędź $u \to v$ w DFG

- u – definicja x
- v – użycie x
- $\exists u \rightsquigarrow v$ wolna dla x



Data Flow (Dependence) Graph

Definicja zmiennej x

to instrukcja ustawiająca x

Użycie zmiennej x

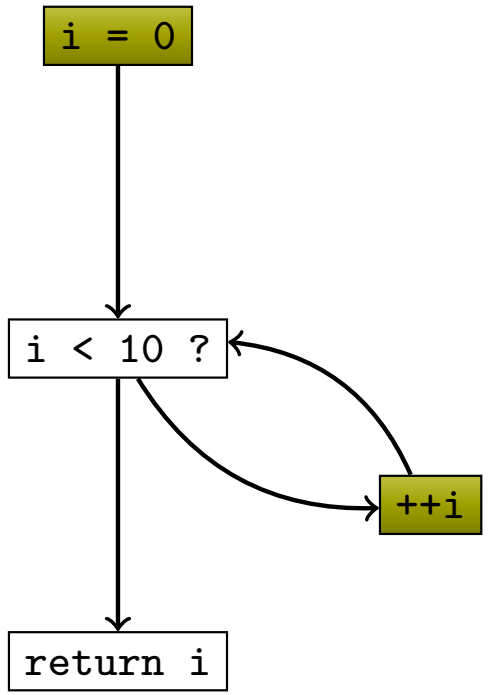
to instrukcja odczytująca x

Ścieżka wolna dla x

to ścieżka w CFG omijająca definicje x

Krawędź $u \to v$ w DFG

- u – definicja x
- v – użycie x
- $\exists u \rightsquigarrow v$ wolna dla x



Data Flow (Dependence) Graph

Definicja zmiennej x

to instrukcja ustawiająca x

Użycie zmiennej x

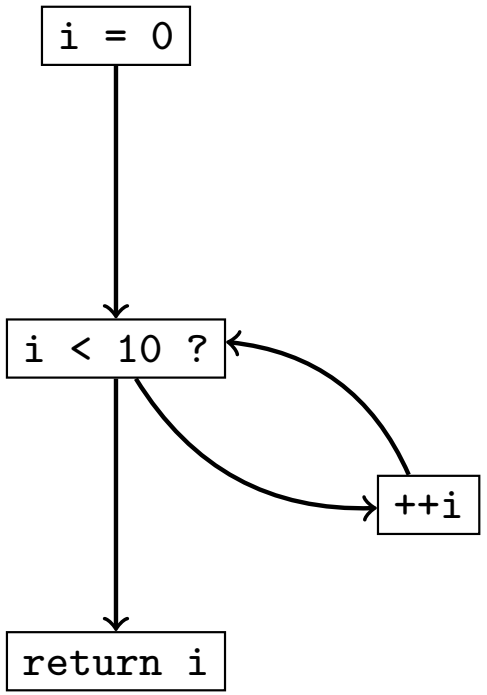
to instrukcja odczytująca x

Ścieżka wolna dla x

to ścieżka w CFG omijająca definicje x

Krawędź $u \to v$ w DFG

- u – definicja x
- v – użycie x
- $\exists u \rightsquigarrow v$ wolna dla x



Data Flow (Dependence) Graph

Definicja zmiennej x

to instrukcja ustawiająca x

Użycie zmiennej x

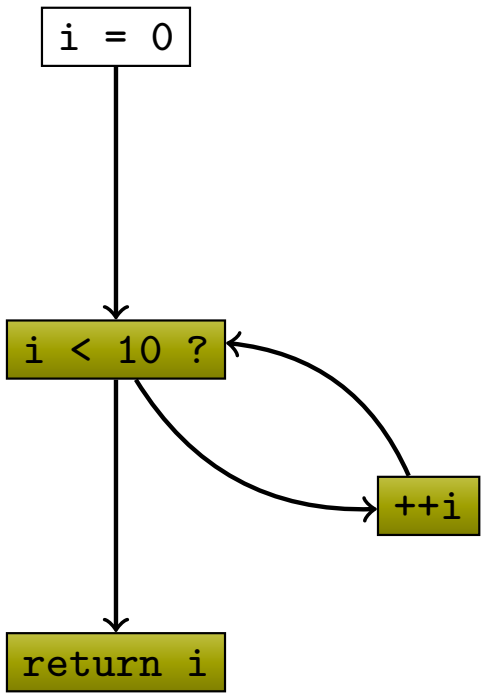
to instrukcja odczytująca x

Ścieżka wolna dla x

to ścieżka w CFG omijająca definicje x

Krawędź $u \to v$ w DFG

- u – definicja x
- v – użycie x
- $\exists u \rightsquigarrow v$ wolna dla x



Data Flow (Dependence) Graph

Definicja zmiennej x

to instrukcja ustawiająca x

Użycie zmiennej x

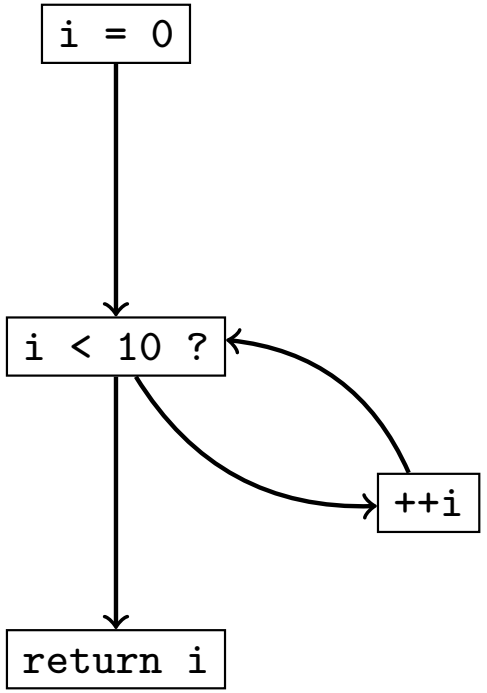
to instrukcja odczytująca x

Ścieżka wolna dla x

to ścieżka w CFG omijająca definicje x

Krawędź $u \to v$ w DFG

- u – definicja x
- v – użycie x
- $\exists u \rightsquigarrow v$ wolna dla x



Data Flow (Dependence) Graph

Definicja zmiennej x

to instrukcja ustawiająca x

Użycie zmiennej x

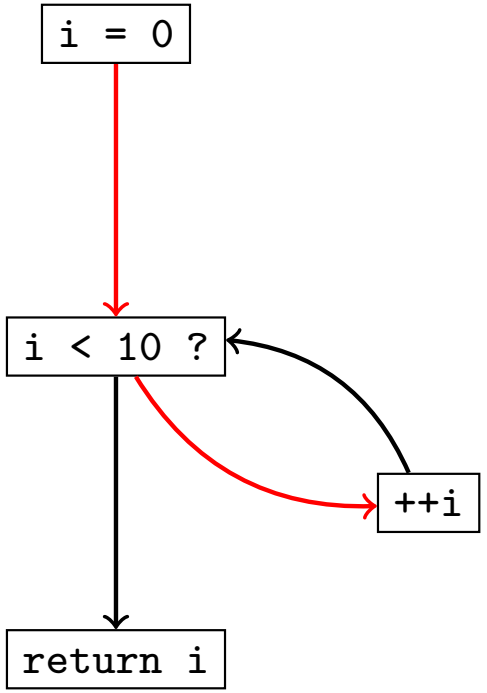
to instrukcja odczytująca x

Ścieżka wolna dla x

to ścieżka w CFG omijająca definicje x

Krawędź $u \to v$ w DFG

- u – definicja x
- v – użycie x
- $\exists u \rightsquigarrow v$ wolna dla x



Data Flow (Dependence) Graph

Definicja zmiennej x

to instrukcja ustawiająca x

Użycie zmiennej x

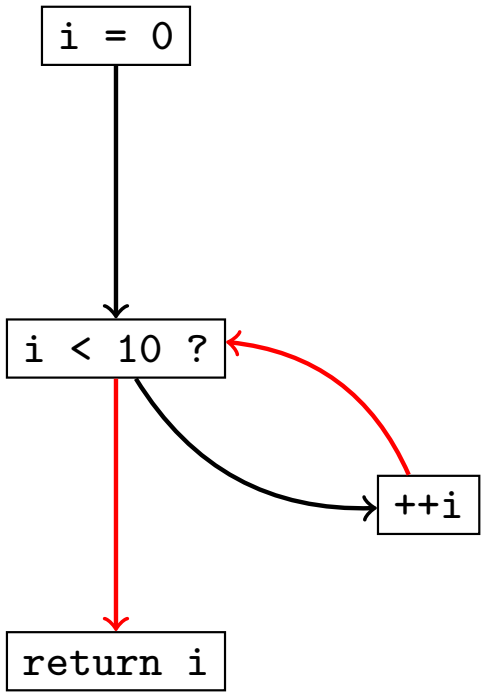
to instrukcja odczytująca x

Ścieżka wolna dla x

to ścieżka w CFG omijająca definicje x

Krawędź $u \to v$ w DFG

- u – definicja x
- v – użycie x
- $\exists u \rightsquigarrow v$ wolna dla x



Data Flow (Dependence) Graph

Definicja zmiennej x

to instrukcja ustawiająca x

Użycie zmiennej x

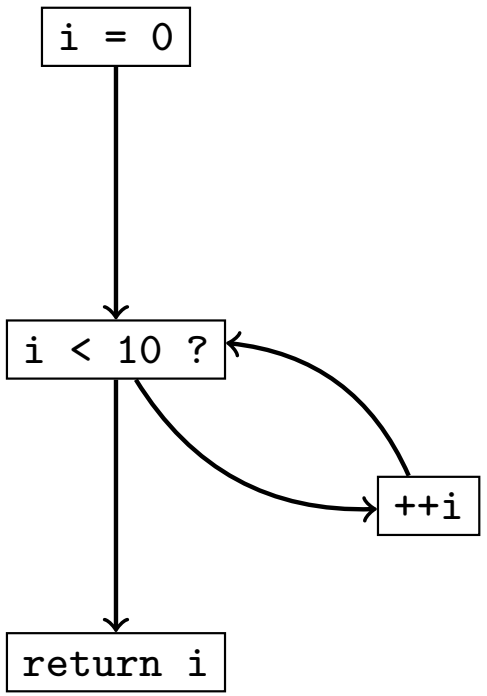
to instrukcja odczytująca x

Ścieżka wolna dla x

to ścieżka w CFG omijająca definicje x

Krawędź $u \to v$ w DFG

- u – definicja x
- v – użycie x
- $\exists u \rightsquigarrow v$ wolna dla x



Data Flow (Dependence) Graph

Definicja zmiennej x

to instrukcja ustawiająca x

Użycie zmiennej x

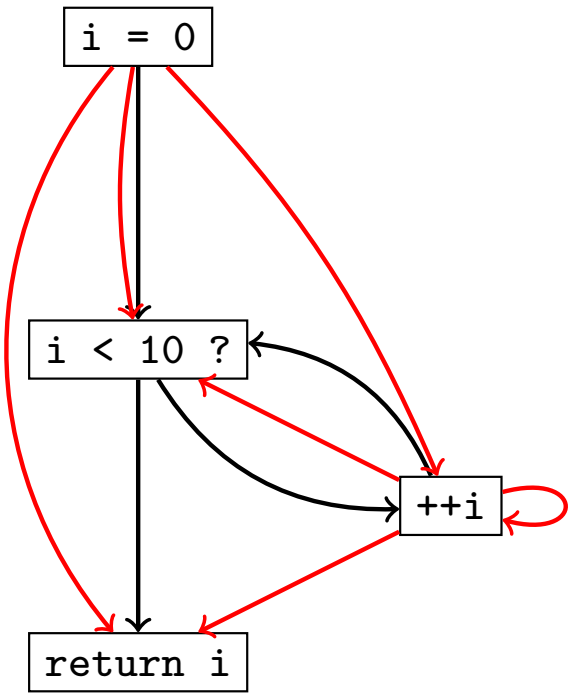
to instrukcja odczytująca x

Ścieżka wolna dla x

to ścieżka w CFG omijająca definicje x

Krawędź $u \to v$ w DFG

- u – definicja x
- v – użycie x
- $\exists u \rightsquigarrow v$ wolna dla x



Data Flow (Dependence) Graph

Definicja zmiennej x

to instrukcja ustawiająca x

Użycie zmiennej x

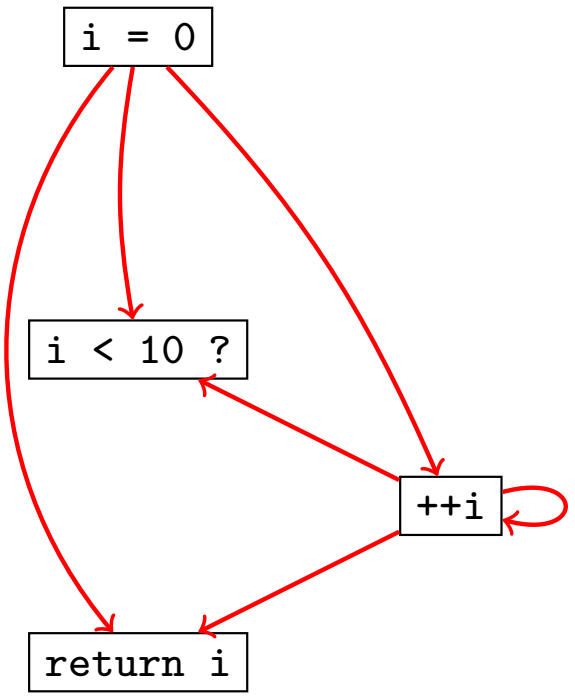
to instrukcja odczytująca x

Ścieżka wolna dla x

to ścieżka w CFG omijająca definicje x

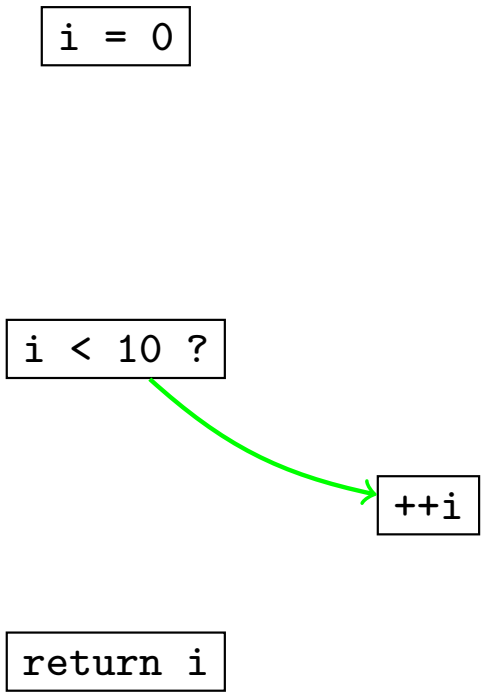
Krawędź $u \to v$ w DFG

- u – definicja x
- v – użycie x
- $\exists u \rightsquigarrow v$ wolna dla x

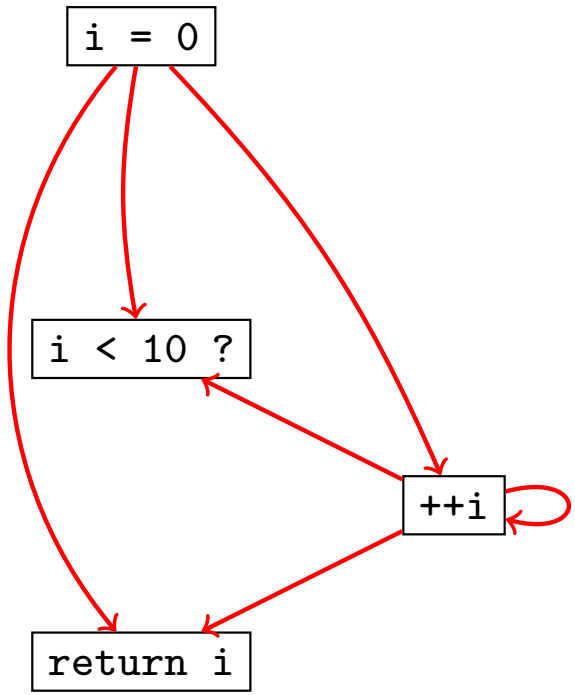


Program Dependence Graph

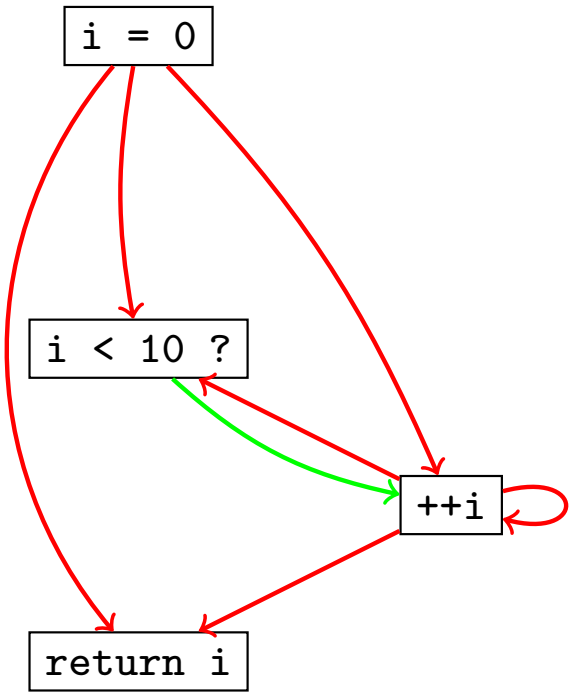
Control Dependence



Data Dependence



Program Dependence Graph



Static Single Assignment form

Definicja zmiennej x

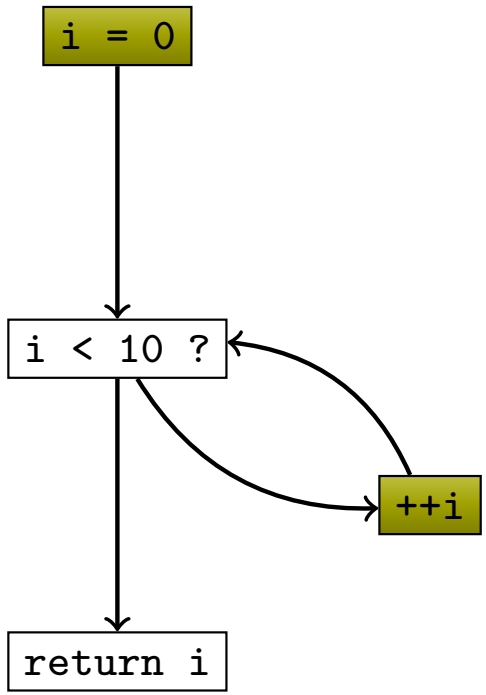
to instrukcja ustawiająca x

Ograniczenie SSA

każda zmienna ma dokładnie 1 definicję

Konstrukcja

- nowa nazwa zmiennej w każdej definicji
- spotkanie 2+ definicji – sztuczna zmienna
- propagacja nowych nazw



Static Single Assignment form

Definicja zmiennej x

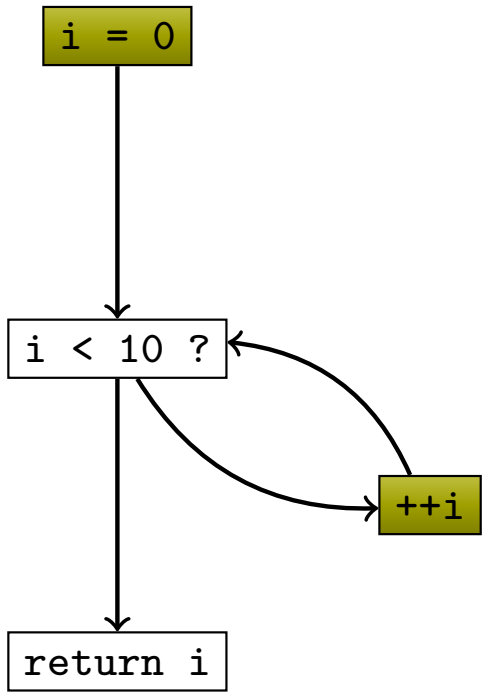
to instrukcja ustawiająca x

Ograniczenie SSA

każda zmienna ma dokładnie 1 definicję

Konstrukcja

- nowa nazwa zmiennej w każdej definicji
- spotkanie 2+ definicji – sztuczna zmienna
- propagacja nowych nazw



Static Single Assignment form

Definicja zmiennej x

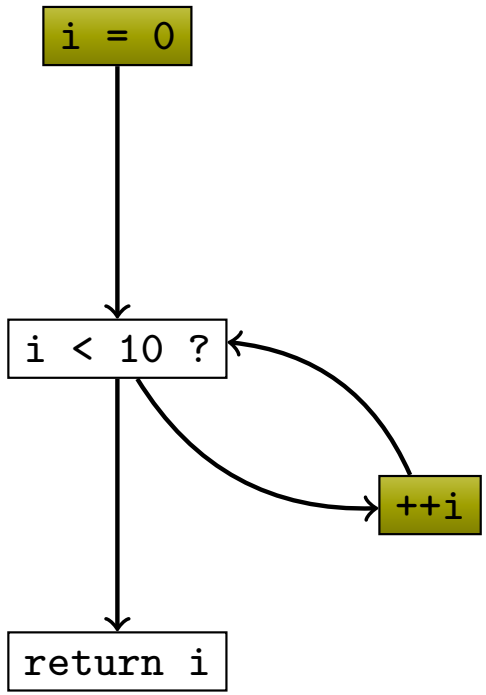
to instrukcja ustawiająca x

Ograniczenie SSA

każda zmienna ma dokładnie 1 definicję

Konstrukcja

- nowa nazwa zmiennej w każdej definicji
- spotkanie 2+ definicji – sztuczna zmienna
- propagacja nowych nazw



Static Single Assignment form

Definicja zmiennej x

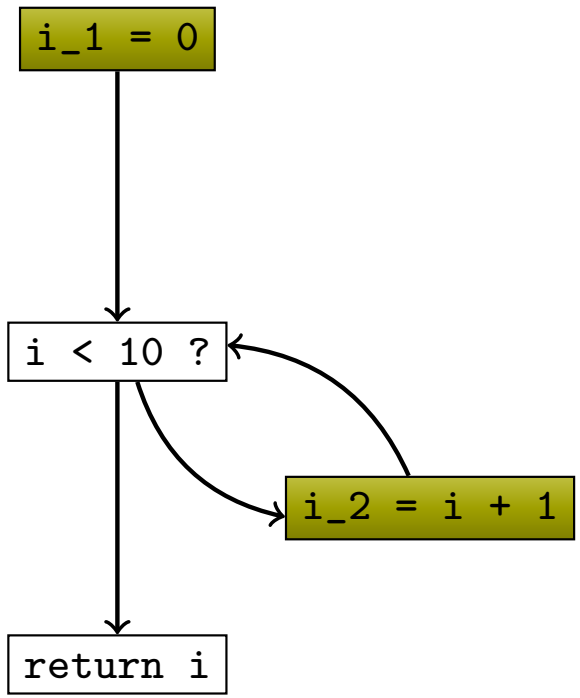
to instrukcja ustawiająca x

Ograniczenie SSA

każda zmienna ma dokładnie 1 definicję

Konstrukcja

- nowa nazwa zmiennej w każdej definicji
- spotkanie 2+ definicji – sztuczna zmienna
- propagacja nowych nazw



Static Single Assignment form

Definicja zmiennej x

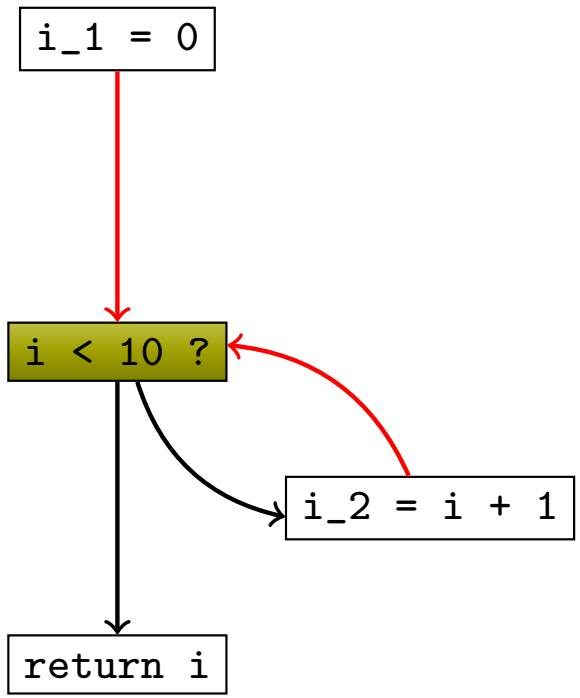
to instrukcja ustawiająca x

Ograniczenie SSA

każda zmienna ma dokładnie 1 definicję

Konstrukcja

- nowa nazwa zmiennej w każdej definicji
- spotkanie 2+ definicji – sztuczna zmienna
- propagacja nowych nazw



Static Single Assignment form

Definicja zmiennej x

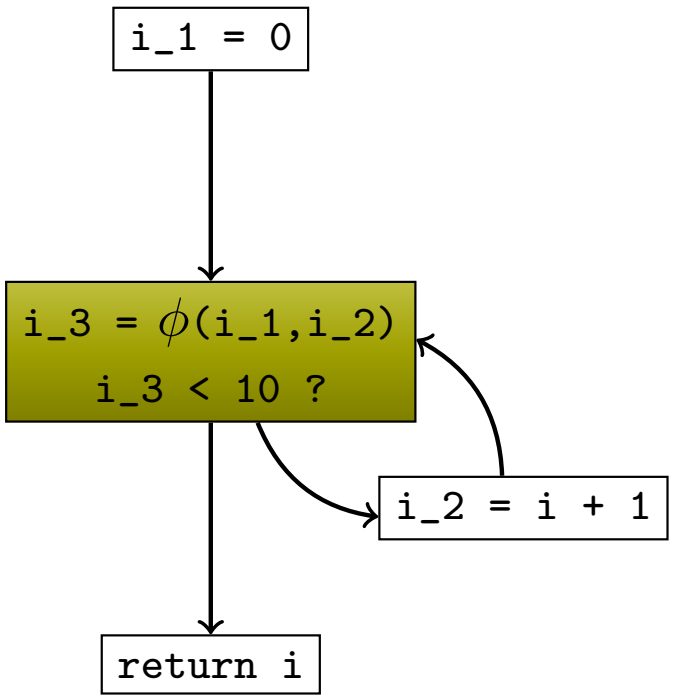
to instrukcja ustawiająca x

Ograniczenie SSA

każda zmienna ma dokładnie 1 definicję

Konstrukcja

- nowa nazwa zmiennej w każdej definicji
- spotkanie 2+ definicji – sztuczna zmienna
- propagacja nowych nazw



Static Single Assignment form

Definicja zmiennej x

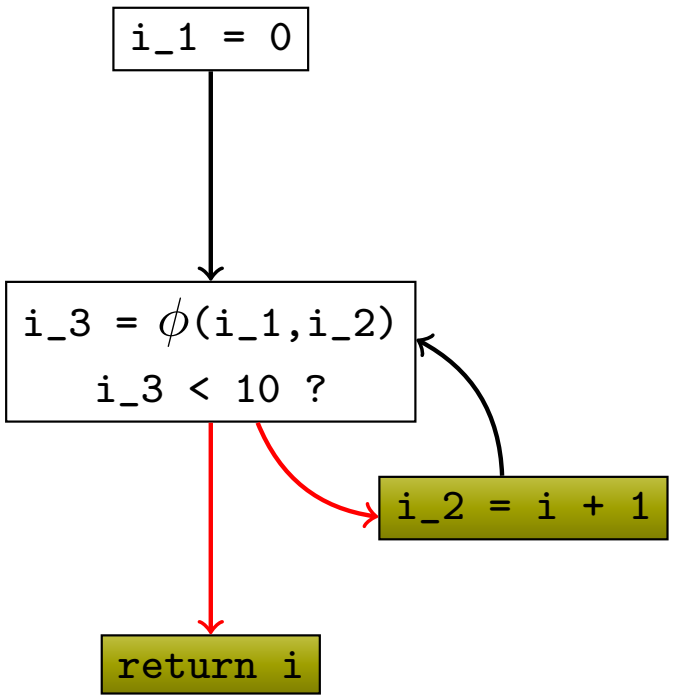
to instrukcja ustawiająca x

Ograniczenie SSA

każda zmienna ma dokładnie 1 definicję

Konstrukcja

- nowa nazwa zmiennej w każdej definicji
- spotkanie 2+ definicji – sztuczna zmienna
- propagacja nowych nazw



Static Single Assignment form

Definicja zmiennej x

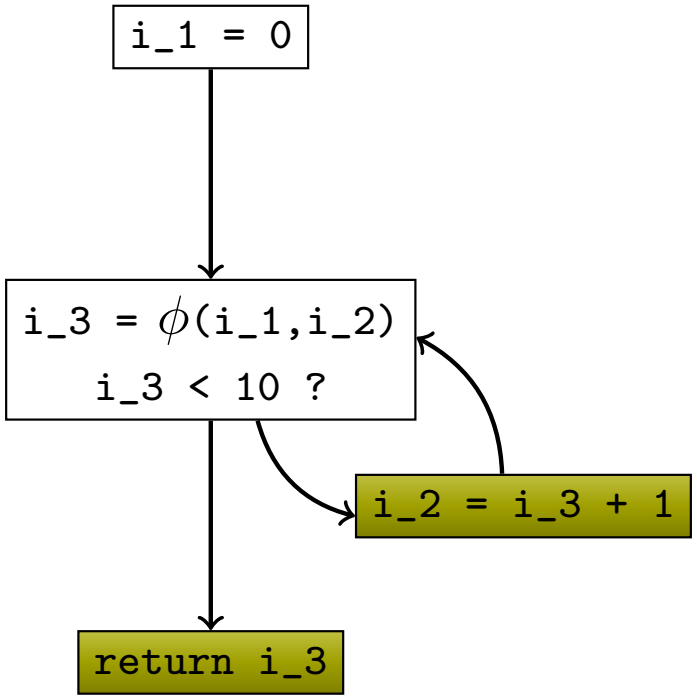
to instrukcja ustawiająca x

Ograniczenie SSA

każda zmienna ma dokładnie 1 definicję

Konstrukcja

- nowa nazwa zmiennej w każdej definicji
- spotkanie 2+ definicji – sztuczna zmienna
- propagacja nowych nazw



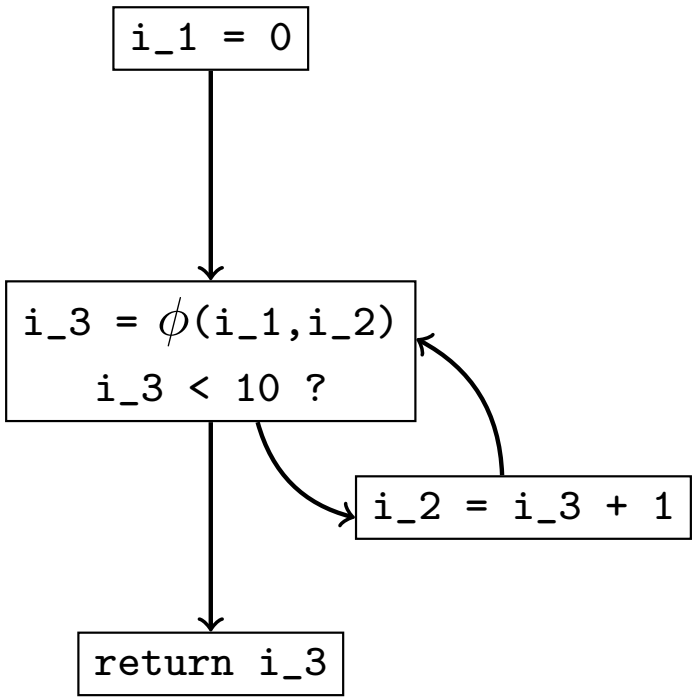
Static Single Information form

Ograniczenia SSA

- każda zmienna ma dokładnie 1 definicję
- definicja zmiennej dominuje każde użycie
- każde użycie zmiennej post-dominuje definicję
- ϕ-użycia są w dominance frontier definicji
- σ-definicje są w postdominance frontier użyć

Konstrukcja

- dla każdej niezależnej gałęzi – sztuczna zmienna
- propagacja nowych nazw



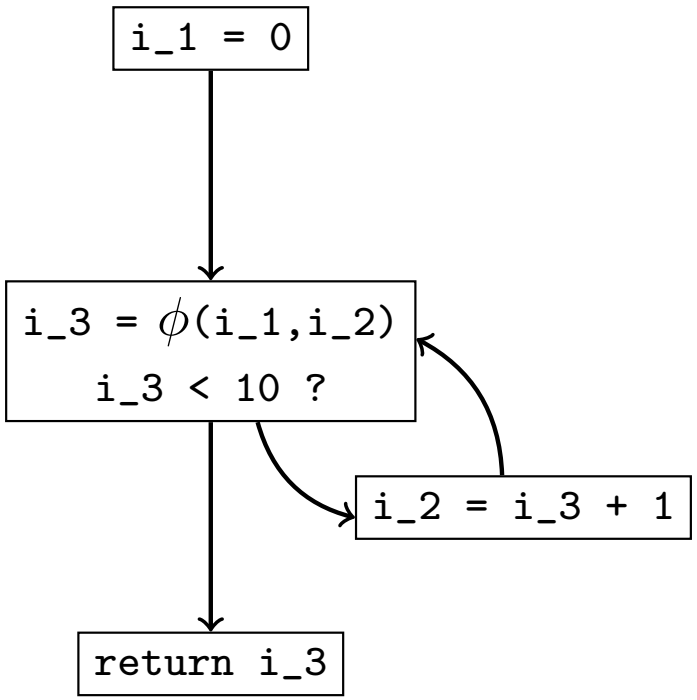
Static Single Information form

Ograniczenia SSI

- każda zmienna ma dokładnie 1 definicję
- definicja zmiennej dominuje każde użycie
- każde użycie zmiennej post-dominuje definicję
- ϕ-użycia są w dominance frontier definicji
- σ-definicje są w postdominance frontier użyć

Konstrukcja

- dla każdej niezależnej gałęzi – sztuczna zmienna
- propagacja nowych nazw



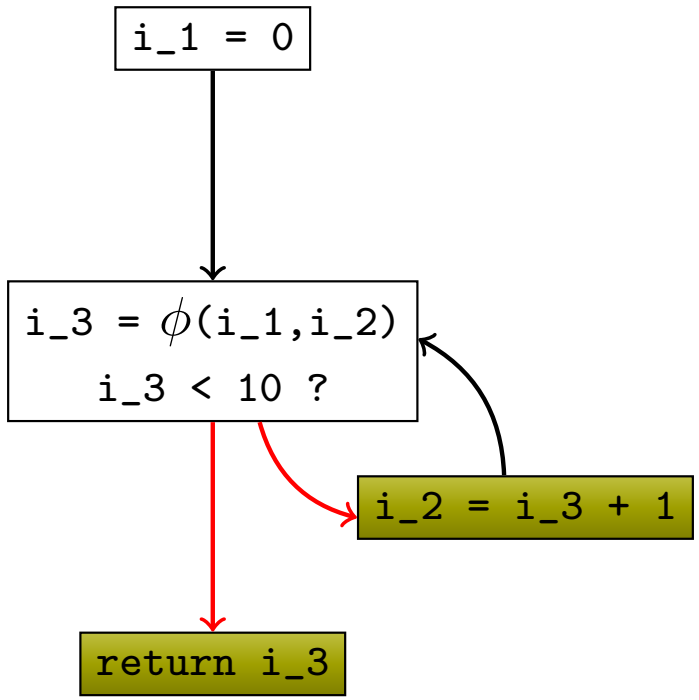
Static Single Information form

Ograniczenia SSI

- każda zmienna ma dokładnie 1 definicję
- definicja zmiennej dominuje każde użycie
- każde użycie zmiennej post-dominuje definicję
- ϕ-użycia są w dominance frontier definicji
- σ-definicje są w postdominance frontier użyć

Konstrukcja

- dla każdej niezależnej gałęzi – sztuczna zmienna
- propagacja nowych nazw



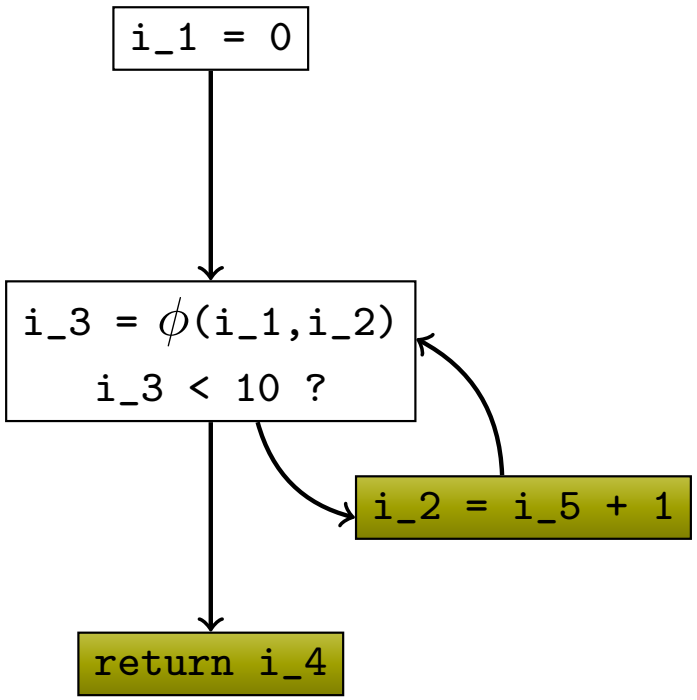
Static Single Information form

Ograniczenia SSI

- każda zmienna ma dokładnie 1 definicję
- definicja zmiennej dominuje każde użycie
- każde użycie zmiennej post-dominuje definicję
- ϕ-użycia są w dominance frontier definicji
- σ-definicje są w postdominance frontier użyć

Konstrukcja

- dla każdej niezależnej gałęzi – sztuczna zmienna
- propagacja nowych nazw



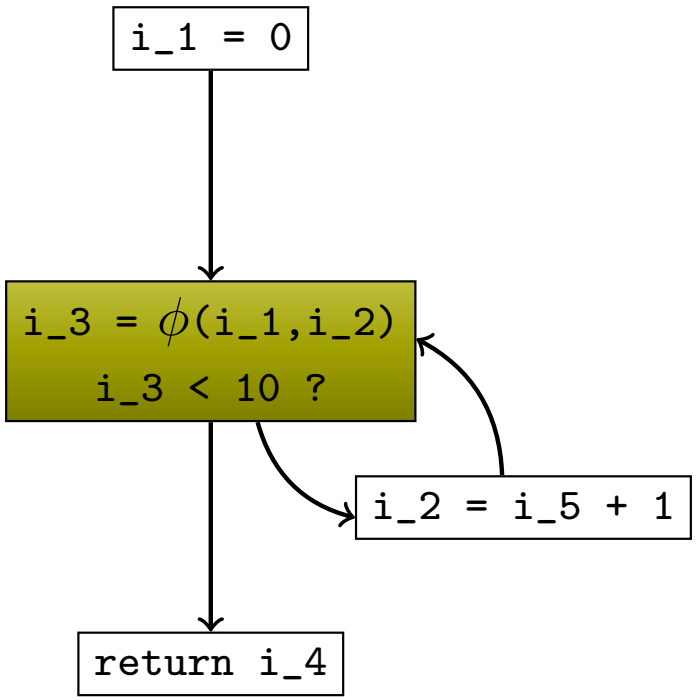
Static Single Information form

Ograniczenia SSI

- każda zmienna ma dokładnie 1 definicję
- definicja zmiennej dominuje każde użycie
- każde użycie zmiennej post-dominuje definicję
- ϕ-użycia są w dominance frontier definicji
- σ-definicje są w postdominance frontier użyć

Konstrukcja

- dla każdej niezależnej gałęzi – sztuczna zmienna
- propagacja nowych nazw



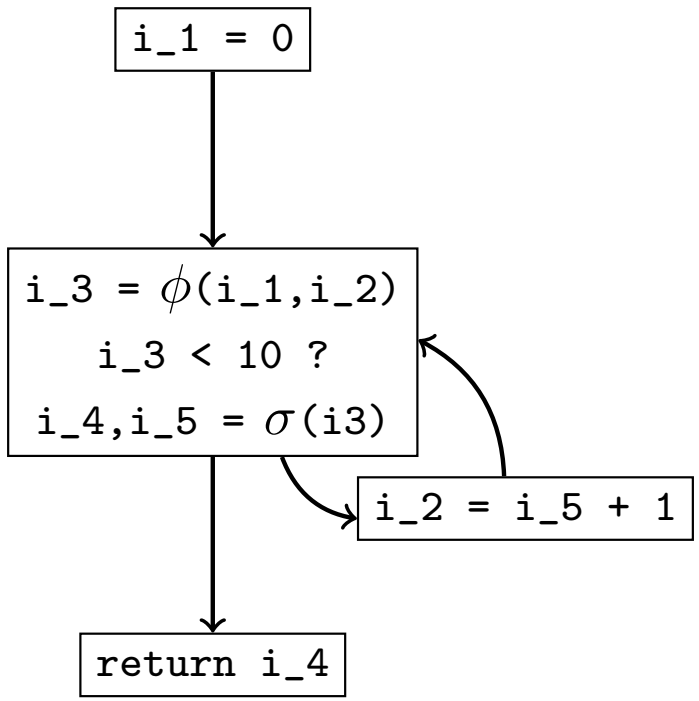
Static Single Information form

Ograniczenia SSI

- każda zmienna ma dokładnie 1 definicję
- definicja zmiennej dominuje każde użycie
- każde użycie zmiennej post-dominuje definicję
- ϕ-użycia są w dominance frontier definicji
- σ-definicje są w postdominance frontier użyć

Konstrukcja

- dla każdej niezależnej gałęzi – sztuczna zmienna
- propagacja nowych nazw



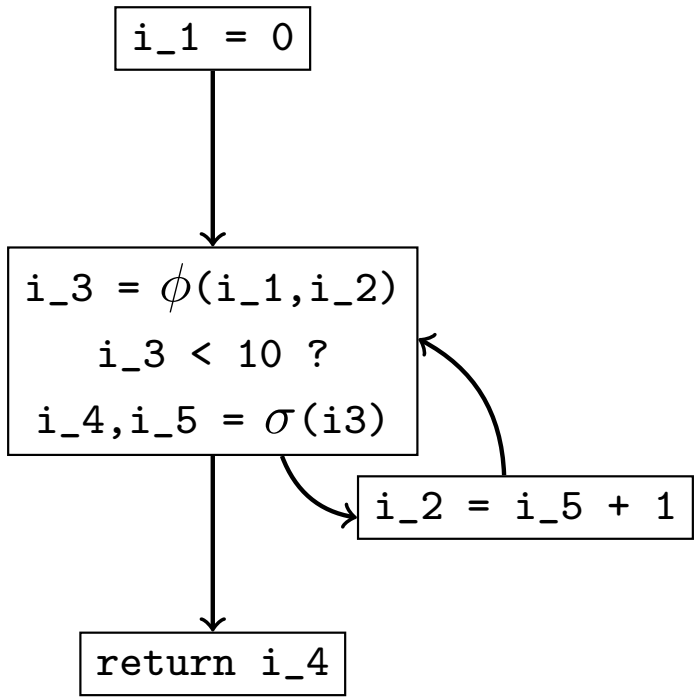
Static Single Information form

Ograniczenia SSI

- każda zmienna ma dokładnie 1 definicję
- definicja zmiennej dominuje każde nie-ϕ-użycie
- każde użycie zmiennej post-dominuje nie-σ-definicję
- ϕ-użycia są w dominance frontier definicji
- σ-definicje są w postdominance frontier użyć

Konstrukcja

- dla każdej niezależnej gałęzi – sztuczna zmienna
- propagacja nowych nazw



Instrumentacja – metody

Czas instrumentacji

- statycznie w procesie kompilacji
- statycznie w momencie uruchamiania programu
- dynamicznie – w czasie działania programu

Język instrumentowany

- język źródłowy (wysokiego poziomu)
- język pośredni / bytecode
- kod binarny (zależny od architektury)

Kod instrumentacji

- oddzielny proces, komunikacja przez IPC
- oddzielny wątek/wątki, komunikacja przez pamięć wspólną
- wbudowany, komunikacja przez wywołania funkcji
- wbudowany, inlining

Instrumentacja – metody

Instrumentacja w czasie kompilacji

- dostęp do informacji z analizy statycznej
- pełny inlining oraz inne optymalizacje
- wymagane wsparcia kompilatora/języka
- brak obsługi generowanego/modyfikowanego kodu

Instrumentacja – metody

Instrumentacja statyczna przy uruchamianiu programu

- niezależność od języka źródłowego
- możliwa analiza statyczna
- możliwe optymalizacje
- brak obsługi generowanego/modyfikowanego kodu
- trudności z odróżnieniem kodu od danych

Instrumentacja – metody

Instrumentacja dynamiczna

- niezależność od języka źródłowego
- łatwa identyfikacja kodu
- obsługa kodu generowanego/modyfikowanego
- dostępny jedynie lokalny widok programu
- lokalne optymalizacje możliwe, lecz trudne

Instrumentacja – metody

Instrumentacja z oddzielnego procesu

- pełna izolacja
- przezroczystość niemal darmowa
- duży koszt komunikacji
- brak dostępu do stanu programu

Instrumentacja – metody

Instrumentacja z dedykowanego wątku

- izolacja kodu
- przezroczystość względnie łatwa do osiągnięcia
- bezpośredni dostęp do stanu programu
- instrumentacja wymaga context-switch
- duży koszt synchronizacji

Instrumentacja – metody

Instrumentacja z „shadow threads”

- izolacja kodu
- przezroczystość względnie łatwa do osiągnięcia
- bezpośredni dostęp do stanu programu
- instrumentacja wymaga context-switch
- podwojone zużycie zasobów dla wątków

Instrumentacja – metody

Instrumentacja wbudowana w kod

- minimalny koszt wydajnościowy
- bezpośredni dostęp do stanu programu
- brak izolacji
- przezroczystość trudna lub niemożliwa do osiągnięcia

Metody modyfikacji kodu

- fault insertion
- jump insertion
- rekompilacja JIT

Rekompilacja dynamiczna

Schemat metody

- basic block – liniowy fragment kodu (oryginalnego)
- oryginalny kod nie jest wykonywany bezpośrednio
- przy pierwszym wejściu do bloku – rekompilacja z dodaniem instrumentacji
- cache skompilowanych fragmentów

Rekompilacja dynamiczna: control flow

Rodzaje skoków (control flow)

- bezpośrednie: adres docelowy stały dla instrukcji
- pośrednie: cel zależny od danych (w tym `return`)
- warunkowość skoku nie wpływa na jego bezpośredniość!

Skoki bezpośrednie

- zastąpione przez skok do odpowiedniego skompilowanego bloku

Skoki pośrednie

- wyszukanie adresu w skompilowanych blokach

Rekompilacja dynamiczna: control flow

Wywołania procedur

- rzeczywiste adresy zmienione
- bezpośrednie wykorzystanie `call` i `ret` – stos zawiera inne wartości, ważne szczególnie przy PIC
- kodowanie jako `push`/`pop` + `jmp`
- bardziej złożona obsługa błędów

Rekompilacja dynamiczna: control flow

Optymalizacja częstych ścieżek wykonania

- trace – ciąg bloków wykonywanych kolejno
- budowane na „ciepłych” blokach
- tworzone zachłannie aż do istniejącego trace'u, bezpośredniego skoku wstecznego lub limitu długości
- skoki pośrednie zastąpione przez porównanie i skok bezpośredni

Rekompilacja dynamiczna: rejestry

Podejście podstawowe

- wykorzystujemy oryginalne rejestry
- konieczność zapisania/odzyskania stanu przed/po instrumentacji

Optymalizacja

- algorymt realokacji rejestrów
- register renaming, zapisany dla każdego bloku
- kod wyrównujący przy każdym skoku
- analiza żywotności (szczególnie ważna przy `eflags`)

Rekompilacja dynamiczna: stos

Wspólny stos

- program może sięgać poza szczyt
- ryzyko przepełnienia przy instrumentacji

Oddzielny stos

- konieczność przełączania
- utrata sprzętowej predykcji adresu powrotu

Rekompilacja dynamiczna: biblioteki

Obsługa bibliotek

- mapa pamięci dla rozróżnienia kodu programu i bibliotek
- możliwe różne polityki instrumentacji

Problemy/niebezpieczeństwa

- callbacks – adresy kodu przekazywane do biblioteki
- współdzielenie biblioteki przez instrumentację i program

Rekompilacja dynamiczna: zasoby systemowe

Ograniczenia zasobów

- wspólne
- możliwa konieczność ukrywania zużycia

Izolacja zasobów

- ukrywanie
- możliwe niepożądane interakcje (np. `sync`)
- file descriptors – numer to też zasób!

Rekompilacja dynamiczna: wielowątkowość

Dostęp do danych instrumentacji

- adres w rejestrze
- stały adres w pamięci thread-local
- stały adres globalny – skompilowane bloki muszą być thread-local

Synchronizacja

- wspólna synchronizacja programu i instrumentacji – niebezpieczeństwo deadlock'u
- gruboziarnista synchronizacja instrumentacji – ograniczenie wydajności
- w przeciwnym wypadku instrumentacja musi być re-entrant

Rekompilacja dynamiczna: zmiany kodu

Kod generowany

- obsługiwany identycznie jak „zwykły”

Modyfikacje kodu

- odpowiedni blok wyrzucany z cache'u
- konieczność usunięcia wchodzących skoków bezpośrednich
- blok modyfikujący sam siebie (x86) – sprawdzanie spójności po każdej instrukcji
- modyfikacja pomiędzy wątkami (x86) – przy 3+ wątkach wymaga instrukcji synchronizującej, na której można podzielić blok; dla 2 wątków nierozwiązane (?)

Rekompilacja dynamiczna: obsługa błędów

Błędy podczas instrumentacji

- izolowane od aplikacji
- co robić, gdy nie można kontynuować instrumentacji?

Błędy w rekompilowanym kodzie

- wykryte przy kompilacji (np. invalid opcode) – podział bloku przed błędem
- wykryte w czasie działania – konieczność przetłumaczenia kontekstu

Rekompilacja dynamiczna: debugger

„Obsługa" debuggera

- pełna izolacja nieosiągalna
- komplikacje przy tłumaczeniu kontekstu
- wątek debuggera – instrumentowany czy nie?

Analiza wpływu (taint analysis)

Zarys techniki

- podział źródeł danych na „bezpieczne” i „niebezpieczne” (znaczone)
- śledzenie przepływu znaczone (tainted) danych
- analiza interesujących sytuacji

Zastosowania

- wykrywanie ataków typu code-injection, SQL injection, cross-site-scripting
- generowanie filtrów bezpieczeństwa
- analiza przepływu zastrzeżonych informacji
- analiza oprogramowania typu malware
- wykrywanie naruszeń protokołów
- ...

Taint analysis

Miary wpływu

- jakościowa (true/false)
- z uwzględnieniem źródła
- ilościowe: liczba bitów, entropia/channel capacity
- przybliżone: przedziały, zbiory wartości

Typowo wykrywane sytuacje

- wykonanie znaczonego kodu
- skok pod znaczony adres
- odczyt/zapis pod znaczony adres

Taint analysis – źródła błędów

Overtainting (false positives)

- zbyt mała szczegółowość
- propagacja przez obecne w programie warunki bezpieczeństwa

Undertainting (false negatives)

- implicit flow

Taint analysis – wyzwania

Znakowanie wskaźników

- uwzględnione: overtainting, zwłaszcza przy strukturach danych o bogatej strukturze pamięciowej
- pominięte: undertainting (ilościowo niewielki, ale istotny dla wykrywania buffer overruns itp.)

Control flow taint

- jedynie analiza dynamiczna: niemożliwy do wykrycia
- analiza statyczna (preprocessing): overtainting

Control flow taint

Obranie konkretnej ścieżki wykonania zawiera informacje

```
if (x==true) y = true; else y = false;
// x==y
```

Nieobranie konkretnej ścieżki wykonania również

```
y = z = false;
if (x==false) z = true;
if (z==false) y = true;
// x==y
```

Rozwiązanie konserwatywne

wszystkie zmienne przypisywane we wszystkich instrukcjach zależnych (control-flow dependent) od znakowanej zmiennej muszą być znakowane.

Taint analysis – wyzwania

Usuwanie oznaczenia (sanitization)

- funkcje stałe (np. `xor eax, eax`) i jednokierunkowe
- konstrukcje kompilacji (np. `switch`)
- rozwiązanie: adnotacje (nie zawsze możliwe)

Opóźnione wykrycie

- przykład: return address overwrite – wykryty dopiero w momencie skoku
- rozwiązanie: postprocessing logów wykonania (nie zawsze praktyczne)

Programy współbieżne

Determinizm programów współbieżnych

- wewnętrzny – sekwencja operacji (wraz z konkretnymi wartościami) każdego wątku niezależna od przeplotu
- zewnętrzny – obserwowalne efekty całego programu niezależne od przeplotu

Synchronizacja

Każda forma synchronizacji wymaga komunikacji!

- explicite – locks, message passing, ...
- implicite – poprzez pamięć współdzieloną

Programy współbieżne

Niebezpieczeństwa współbieżności

- zagłodzenie, w tym livelock/deadlock
- brak atomowości operacji

Trudności w wykrywaniu błędów

- skomplikowane modele statyczne
- zależność od przeplotu, wystąpienie często mało prawdopodobne
- zależność od konkretnej architektury/wersji CPU

Data Races

Data Race

Nieatomowość operacji obserwowalna jedynie w przypadku „jednoczesnego" dostępu do tych samych danych przez różne wątki.

Ograniczenia „jednoczesności"

- sekcja krytyczna: operacje/bloki mogą być wykonane w dowolnej kolejności, ale rozłącznie
- jednoznaczna kolejność: jedna operacja musi poprzedzać drugą

Data Races – formalnie

Porządek zdarzeń w programie

- Operacje każdego wątku t uporządkowane liniowo:

$$t_1 < t_2 < \ldots < t_{n_t}$$

- Konkretne wykonanie programu (przeplot wątków) – rozszerzenie liniowe „$\prec$” porządku „$<$”
- Komunikacja pomiędzy instrukcją x a y (wymaga $x \prec y$):

$$x \longrightarrow y$$

- Relacja „happens before” ($\lhd$) – domknięcie przechodnie sumy porządków „$<$” i „$\longrightarrow$” (w oczywisty sposób zawsze zgodna z „$\prec$”)

Data Races – formalnie

Porządek zdarzeń w programie

- Operacje każdego wątku t uporządkowane liniowo:

$$t_1 < t_2 < \ldots < t_{n_t}$$

- Konkretne wykonanie programu (przeplot wątków) – rozszerzenie liniowe „$\prec$” porządku „$<$”
- Komunikacja pomiędzy instrukcją x a y (wymaga $x \prec y$):

$$x \rightarrow y$$

- Relacja „happens before” ($\lhd$) – domknięcie przechodnie sumy porządków „$<$” i „$\rightarrow$” (w oczywisty sposób zawsze zgodna z „$\prec$”)

Data Races – formalnie

Porządek zdarzeń w programie

- Operacje każdego wątku t uporządkowane liniowo:

$$t_1 < t_2 < \ldots < t_{n_t}$$

- Konkretne wykonanie programu (przeplot wątków) – rozszerzenie liniowe „$\prec$” porządku „$<$”
- Komunikacja pomiędzy instrukcją x a y (wymaga $x \prec y$):

$$x \rightarrow y$$

- Relacja „happens before” ($\lhd$) – domknięcie przechodnie sumy porządków „$<$” i „$\rightarrow$” (w oczywisty sposób zawsze zgodna z „$\prec$”)

Data Races – formalnie

Porządek zdarzeń w programie

- Operacje każdego wątku t uporządkowane liniowo:

$$t_1 < t_2 < \ldots < t_{n_t}$$

- Konkretne wykonanie programu (przeplot wątków) – rozszerzenie liniowe „$\prec$” porządku „$<$”
- Komunikacja pomiędzy instrukcją x a y (wymaga $x \prec y$):

$$x \rightarrow y$$

- Relacja „happens before” ($\lhd$) – domknięcie przechodnie sumy porządków „$<$” i „$\rightarrow$” (w oczywisty sposób zawsze zgodna z „$\prec$”)

Data Races – formalnie

Między operacjami x i y na pamięci wspólnej występuje data race, gdy

- x i y dotykają wspólnej komórki pamięci
- przynajmniej jedna z x i y to zapis
- nie zachodzi $x \lhd y$ ani $y \lhd x$

Definicja zależy od przeplotu „$\prec$”

```
x := 0;
acquire(l);
y++;
release(l);
```

```
acquire(l);
y++;
release(l);
x := 1;
```

Data Races – wykrywanie

Dynamiczne wykrywanie data races

- pesymistyczne (zakładające brak synchronizacji implicite) – liczne false positives
- optymistyczne (zakładające maksymalną synchronizację implicite) – false negatives
- tak czy inaczej wymaga oglądania różnych przeplotów (w ogólności NP-trudne)

Techniki

- lock erasure
- vector clocks
- lock sets
- hybrydy i heurystyki

Lock Erasure

Dla każdego wątku t

- zbiór aktywnych locków L_t, na początku $\{t\}$

Dla każdej zmiennej x

- zbiór locków C_x, które były aktywne przy każdym dotychczasowym dostępie do x, na początku universum

Lock Erasure

ACQUIRE(L) *//by thread t*

$$L_t := L_t \cup \{l\}$$

RELEASE(L) *//by thread t*

$$L_t := L_t - \{l\}$$

READ/WRITE(X) *//by thread t*

$$C_x := C_x \cap L_t$$

$$C_x = \emptyset \Rightarrow \text{ report race}$$

Lock Erasure

Zalety

- łatwa implementacja
- obsługa zmiennych prywatnych wątków
- niska średnia złożoność (zbiory L_t zazwyczaj niewielkie)

Wady

- wymuszona polityka synchronizacji
- brak obsługi transferu własności obiektu

Vector Clocks

Dla każdego wątku t

- zbiór aktywnych locków L_t, na początku $\emptyset$
- monotoniczny „zegar” (licznik instrukcji synchronizujących)
- najnowszy widoczny (w sensie $\triangleleft$) czas każdego innego wątku u, $B_t(u)$ ($B_t(t)$ to „czas lokalny” t)

Dla każdej zmiennej x

- zbiór locków C_x
- ostatni czas dostępu z każdego wątku t, $S_x(t)$, z zachowaniem jedynie operacji maksymalnych w sensie $\triangleleft$

Vector Clocks

acquire/release(l) *//by thread t*

jak w technice LockErasure

fork(u) *//by thread t*

$$L_u := \emptyset;$$
$$B_u := B_t;$$
$$B_u(u) := 1;$$
$$B_t(t) := B_t(t) + 1$$

join(u) *//by thread t*

$$B_t := B_t \oplus B_u$$

read/write(x) *//by thread t*

$$S_x := S_x \oplus \{(t, B_t(t))\}$$
$$|S_x| > 1 \Rightarrow C_x := C_x \cap L_t$$
$$\text{else} \Rightarrow C_x := L_t$$
$$|S_x| > 1 \wedge C_x = \emptyset \Rightarrow \text{ report race}$$

Vector Clocks

Zalety

- precyzja (brak false positives względem $\lhd$)
- obsługa zmiennych thread-local, transferu własności

Wady

- złożoność operacji $\Theta(n)$

Vector Clocks – optymalizacje

Założenie

- wykrywamy tylko pierwszy race dla każdej zmiennej

Operacje zapisu

- zapisy do x są liniowo uporządkowane przez $\lhd$
- pamiętamy tylko ostatni

Operacje odczytu

- łatwo wykryć, czy zmienna jest thread-local ($|S_x| = 1$) albo chroniona przez lock ($|C_x| > 0$)
- odczyty takich zmiennych są liniowo uporządkowane przez $\lhd$
- pamiętamy tylko ostatni, zmieniając reprezentację gdy trzeba

Lock Sets (a.k.a. Goldilocks)

Synchronizacja przez

- porządek instrukcji w obrębie wątku
- locks
- „zmienne synchronizujące” (np. `volatile`)

Algorytm oblicza (implicite) relację $\lhd$ jako domknięcie przechodnie sumy tych mechanizmów synchronizacji.

Dla każdej zmiennej x

- lock-set L_x, mogący zawierać identyfikatory wątków, locków i „zmiennych synchronizujących”, na początku $\emptyset$

Lock Sets

READ/WRITE(X) *//by thread t*

$$L_x \neq \emptyset \wedge t \notin L_x \Rightarrow \text{ report race}$$
$$L_x := \{t\}$$

READ(V) *//by thread t, volatile*

$$\forall x : v \in L_x \Rightarrow L_x := L_x \cup \{t\}$$

WRITE(V) *//by thread t, volatile*

$$\forall x : t \in L_x \Rightarrow L_x := L_x \cup \{v\}$$

Lock Sets

ACQUIRE(L) *//by thread t*

$$\forall x : l \in L_x \Rightarrow L_x := L_x \cup \{t\}$$

RELEASE(L) *//by thread t*

$$\forall x : t \in L_x \Rightarrow L_x := L_x \cup \{l\}$$

FORK(U) *//by thread t*

$$\forall x : t \in L_x \Rightarrow L_x := L_x \cup \{u\}$$

JOIN(U) *//by thread t*

$$\forall x : u \in L_x \Rightarrow L_x := L_x \cup \{t\}$$

Lock Sets

Implementacja naiwna

całkowicie niepraktyczna :)

Implementacja leniwa

- update'y do L_x wirtualne – zapisywane na liście
- dla każdej komórki – wskaźnik do ostatniego update'u dotyczącego dostępu do niej
- jedyne zapytanie o L_x to czy $L_x = \emptyset \vee t \in L_x$ – realizowane przez przeglądanie listy update'ów
- update'y niewidoczne (zasłonięte przez późniejsze przypisanie $L_x := \{t\}$) usuwane z listy

Mutation testing

Competent Programmer Hypothesis

- programiści zazwyczaj piszą programy bliskie poprawnej wersji
- $\Rightarrow$ zakładamy, że ewentualne błędy mogą być poprawione przez drobne zmiany w kodzie źródłowym

Coupling Effect

- złożone błędy są powiązane (coupled) z prostymi tak silnie, że wykrywając te drugie automatycznie wykryjemy wysoki odsetek pierwszych
- $\Rightarrow$ testy odrzucające błędne programy bliskie poprawnym są bardzo skuteczne również bez założenia CPH

Mutation testing

Competent Programmer Hypothesis

- programiści zazwyczaj piszą programy bliskie poprawnej wersji
- $\Rightarrow$ zakładamy, że ewentualne błędy mogą być poprawione przez drobne zmiany w kodzie źródłowym

Coupling Effect

- złożone błędy są powiązane (coupled) z prostymi tak silnie, że wykrywając te drugie automatycznie wykryjemy wysoki odsetek pierwszych
- $\Rightarrow$ testy odrzucające błędne programy bliskie poprawnym są bardzo skuteczne również bez założenia CPH

Mutation testing

Competent Programmer Hypothesis

- programiści zazwyczaj piszą programy bliskie poprawnej wersji
- $\Rightarrow$ zakładamy, że ewentualne błędy mogą być poprawione przez drobne zmiany w kodzie źródłowym

Coupling Effect

- złożone błędy są powiązane (coupled) z prostymi tak silnie, że wykrywając te drugie automatycznie wykryjemy wysoki odsetek pierwszych
- $\Rightarrow$ testy odrzucające błędne programy bliskie poprawnym są bardzo skuteczne również bez założenia CPH

Mutation testing

Wejście

- program p
- zbiór testów T

Proces

- ewaluacja p na T, poprawki
- modyfikacja p poprzez składniowe „operatory mutacyjne”
- $\rightarrow$ zbiór mutantów $M = \{m_1, m_2, \ldots, m_k\}$
- ewaluacja każdego m_i na T ($\rightarrow$ „żywe” i „martwe”)
- usunięcie mutantów semantycznie równoważnych p
- wzbogacenie T o testy zabijające pozostałe mutanty

Mutation testing

Wejście

- program p
- zbiór testów T

Proces

- ewaluacja p na T, poprawki
- modyfikacja p poprzez składniowe „operatory mutacyjne”
- $\rightarrow$ zbiór mutantów $M = \{m_1, m_2, \ldots, m_k\}$
- ewaluacja każdego m_i na T ($\rightarrow$ „żywe” i „martwe”)
- usunięcie mutantów semantycznie równoważnych p
- wzbogacenie T o testy zabijające pozostałe mutanty

Mutation testing

Wejście

- program p
- zbiór testów T

Proces

- ewaluacja p na T, poprawki
- modyfikacja p poprzez składniowe „operatory mutacyjne”
- $\rightarrow$ zbiór mutantów $M = \{m_1, m_2, \ldots, m_k\}$
- ewaluacja każdego m_i na T ($\rightarrow$ „żywe” i „martwe”)
- usunięcie mutantów semantycznie równoważnych p
- wzbogacenie T o testy zabijające pozostałe mutanty

Mutation testing – score

$$\mu(T, p, M) = \frac{\text{number of mutants killed by } T}{\text{number of mutants non-equivalent to } p}$$

Mutation testing – operatory

Operatory mutacyjne (Fortran)

- AAR: array ref. for array ref.
- ABS: absolute value insertion
- ACR: array ref. for constant
- AOR: arithmetic operator repl.
- ASR: array ref. for scalar
- CAR: constant for array ref.
- CNR: comparable array name repl.
- CRP: constant repl.
- CSR: constant for scalar
- DER: DO statement alterations
- DSA: DATA statement alterations
- GLR: GOTO label repl.
- LCR: logical connector repl.
- ROR: relational operator repl.
- RSR: RETURN statement repl.
- SAN: statement analysis
- SAR: scalar for array ref.
- SCR: scalar for constant
- SDL: statement deletion
- SRC: source constant repl.
- SVR: scalar value repl.
- UOI: unary operator insertion

Mutation testing – operatory

Operatory mutacyjne (OOP)

- zamiana typu na nadklasę lub interfejs
- zmiana typu przy tworzeniu obiektu
- zmiana kolejności argumentów w deklaracji metody
- zmiana kolejności argumentów wywołania
- usunięcie wariantu (overload) metody
- usunięcie argumentu/ów wywołania
- dodanie/usunięcie metody/pola przesłaniającego
- zmiana dostępności
- zmiana pól klasy na pola obiektu i odwrotnie
- usunięcie obsługi wyjątku

Mutation testing – koszt

Liczba mutantów

- wysoki koszt testowania
- „human oracle problem”

Wykrywanie mutantów równoważnych

- nierozstrzygalne
- heurystyki kosztowne obliczeniowo
- trudne nawet dla człowieka ($\sim$ 10% błędów)

Mutation testing – redukcja kosztu

Redukcja liczby mutantów

- próbkowanie (sampling)
- klasteryzacja
- mutacje selektywne
- mutacje wyższego rzędu

Redukcja kosztów obliczeniowych

- silne/słabe mutacje
- integracja z kompilacją
- schematy mutacji
- mutacje postkompilacyjne
- zrównoleglenie

Redukcja liczby mutantów

Próbkowanie – losowy podzbiór zbioru mutantów

Rozkład jednorodny

- 10% mutantów $\Rightarrow$ skuteczność $\sim$ 85%

Bayesian sequential probability ratio test (SPRT)

- przerywa losowanie, gdy nowe mutacje mało zmieniają
- lepsze dopasowanie do konkretnych testów niż wariant jednorodny
- można ustalić żądany (prawdopodobny) próg skuteczności

Redukcja liczby mutantów

Próbkowanie – losowy podzbiór zbioru mutantów

Rozkład jednorodny

- 10% mutantów $\Rightarrow$ skuteczność $\sim$ 85%

Bayesian sequential probability ratio test (SPRT)

- przerywa losowanie, gdy nowe mutacje mało zmieniają
- lepsze dopasowanie do konkretnych testów niż wariant jednorodny
- można ustalić żądany (prawdopodobny) próg skuteczności

Redukcja liczby mutantów

Próbkowanie – losowy podzbiór zbioru mutantów

Rozkład jednorodny

- 10% mutantów $\Rightarrow$ skuteczność $\sim$ 85%

Bayesian sequential probability ratio test (SPRT)

- przerywa losowanie, gdy nowe mutacje mało zmieniają
- lepsze dopasowanie do konkretnych testów niż wariant jednorodny
- można ustalić żądany (prawdopodobny) próg skuteczności

Redukcja liczby mutantów

Klasteryzacja

- miara podobieństwa mutantów
- podział na klasy podobieństwa
- testowanie reprezentantów klas

Miara podobieństwa

- podzbiór „zabójczych” testów
- analiza zakresów zmiennych

Algorytmy

- k median (k-means)
- aglomeracyjny
- ...

Redukcja liczby mutantów

Klasteryzacja

- miara podobieństwa mutantów
- podział na klasy podobieństwa
- testowanie reprezentantów klas

Miara podobieństwa

- podzbiór „zabójczych” testów
- analiza zakresów zmiennych

Algorytmy

- k median (k-means)
- aglomeracyjny
- ...

Redukcja liczby mutantów

Klasteryzacja

- miara podobieństwa mutantów
- podział na klasy podobieństwa
- testowanie reprezentantów klas

Miara podobieństwa

- podzbiór „zabójczych” testów
- analiza zakresów zmiennych

Algorytmy

- k median (k-means)
- aglomeracyjny
- ...

Algorytm k median

Inicjalizacja

- zbiór wartości V
- ustalony parametr k
- arbitralny (losowy) podział V na

$$V = V_1 \cup V_2 \cup \ldots V_k$$

(V_i rozłączne i mniej więcej równoliczne)

Krok algorytmu

- oblicz centrum (medianę) każdego V_i: m_i
- oblicz odległości $d_i(v)$ każdego $v \in V$ od każdego m_i
- $V_i' := \{v : d_i(v) < d_{i'}(v) \text{ dla } i' \neq i\}$
- zakończ, jeśli $\{V_i'\}_i = \{V_i\}_i$

Algorytm aglomeracyjny

Inicjalizacja

- zbiór wartości V
- każde $v \in V$ tworzy osobny klaster $C_v = \{v\}$

Budowa drzewa

- oblicz odległości wszystkich klastrów (minimalne, średnie, maksymalne, odległości centrów, ...)
- połącz dwa najbliższe klastry

Podział drzewa

- na k fragmentów: algorytm dynamiczny bottom-up
- z odległością progową: algorytm zachłanny

Redukcja liczby mutantów

Mutacje selektywne

- AAR: array ref. for array ref.
- ABS: absolute value insertion
- ACR: array ref. for constant
- AOR: arithmetic operator repl.
- ASR: array ref. for scalar
- CAR: constant for array ref.
- CNR: comparable array name repl.
- CRP: constant repl.
- CSR: constant for scalar
- DER: DO statement alterations
- DSA: DATA statement alterations
- GLR: GOTO label repl.
- LCR: logical connector repl.
- ROR: relational operator repl.
- RSR: RETURN statement repl.
- SAN: statement analysis
- SAR: scalar for array ref.
- SCR: scalar for constant
- SDL: statement deletion
- SRC: source constant repl.
- SVR: scalar value repl.
- UOI: unary operator insertion

liczba mutantów: $< 40\%$; skuteczność $\sim 99.5\%$

Redukcja liczby mutantów

Mutacje selektywne

- AAR: array ref. for array ref.
- ABS: absolute value insertion
- ACR: array ref. for constant
- AOR: arithmetic operator repl.
- ASR: array ref. for scalar
- CAR: constant for array ref.
- CNR: comparable array name repl.
- CRP: constant repl.
- CSR: constant for scalar
- DER: DO statement alterations
- DSA: DATA statement alterations
- GLR: GOTO label repl.
- LCR: logical connector repl.
- ROR: relational operator repl.
- RSR: RETURN statement repl.
- SAN: statement analysis
- SAR: scalar for array ref.
- SCR: scalar for constant
- SDL: statement deletion
- SRC: source constant repl.
- SVR: scalar value repl.
- UOI: unary operator insertion

liczba mutantów: $< 40\%$; skuteczność $\sim 99.5\%$

Redukcja kosztów obliczeniowych

Silna mutacja

- test zabija mutanta jeśli ostateczny wynik testowanego programu jest różny od oczekiwanego
- pasuje do modelu black-box
- wymaga ewaluacji całego programu/mutanta

Słaba mutacja

- program podzielony na komponenty $C_1, \ldots, C_r$
- mutacja (pierwszego rzędu) w komponencie C_i
- test zabija mutanta jeśli którykolwiek wynik C_i jest różny od „oczekiwanego” (oryginalnego)
- umożliwia szybsze wykrycie zabitych mutantów oraz równoczesne sprawdzanie mutacji w różnych komponentach
- wymaga odczytywania stanu programu

Redukcja kosztów obliczeniowych

Drobne komponenty (fine-grained)

- odczyt/zapis komórki pamięci, wartość wyrażenia arytmetycznego/logicznego
- najszybsza ewaluacja
- najsłabsze odwzorowanie silnych mutacji ($\sim 70\%$)

Średnie komponenty (coarse-grained)

- pierwsze obliczenie wyrażenia/instrukcji/LCS; k-te wykonanie LCS
- szybka ewaluacja, łatwiejsza implementacja (np. przez instrumentację)
- odwzorowanie $\sim 85 - 95\%$ silnych mutacji

Komponenty „naturalne”

- zgodne z rzeczywistym podziałem testowanego systemu
- bardzo dobre odwzorowanie silnych mutacji
- powinny być przetestowane „poziom niżej”!

Redukcja kosztów obliczeniowych

Drobne komponenty (fine-grained)

- odczyt/zapis komórki pamięci, wartość wyrażenia arytmetycznego/logicznego
- najszybsza ewaluacja
- najsłabsze odwzorowanie silnych mutacji ($\sim 70\%$)

Średnie komponenty (coarse-grained)

- pierwsze obliczenie wyrażenia/instrukcji/LCS; k-te wykonanie LCS
- szybka ewaluacja, łatwiejsza implementacja (np. przez instrumentację)
- odwzorowanie $\sim 85 - 95\%$ silnych mutacji

Komponenty „naturalne”

- zgodne z rzeczywistym podziałem testowanego systemu
- bardzo dobre odwzorowanie silnych mutacji
- powinny być przetestowane „poziom niżej”!

Redukcja kosztów obliczeniowych

Integracja z kompilatorem

- kompilator generuje wszystkie mutanty „na raz”
- wykorzystuje podobieństwo mutantów i oryginału
- wykorzystuje informacje z procesu kompilacji i optymalizacji (control flow, data flow, . . .)

Zalety

- znacząco niższy koszt kompilacji
- eliminacja części mutantów równoważnych

Wady

- wymaga zmiany kompilatora
- zaawansowane techniki (klasteryzacja itp.) na razie niedostępne

Redukcja kosztów obliczeniowych

Integracja z kompilatorem

- kompilator generuje wszystkie mutanty „na raz”
- wykorzystuje podobieństwo mutantów i oryginału
- wykorzystuje informacje z procesu kompilacji i optymalizacji (control flow, data flow, . . .)

Zalety

- znacząco niższy koszt kompilacji
- eliminacja części mutantów równoważnych

Wady

- wymaga zmiany kompilatora
- zaawansowane techniki (klasteryzacja itp.) na razie niedostępne

Redukcja kosztów obliczeniowych

Schematy/szablony mutacji

- każde potencjalne miejsce mutacji zastąpione wywołaniem biblioteki mutującej
- wybór mutacji dokonywany w czasie wykonania

Zalety

- niski koszt kompilacji
- standardowy kompilator
- mutacje nieosiągalne nigdy nie sprawdzane

Wady

- niektóre typy mutacji niedostępne
- zaawansowane techniki niedostępne

Redukcja kosztów obliczeniowych

Schematy/szablony mutacji

- każde potencjalne miejsce mutacji zastąpione wywołaniem biblioteki mutującej
- wybór mutacji dokonywany w czasie wykonania

Zalety

- niski koszt kompilacji
- standardowy kompilator
- mutacje nieosiągalne nigdy nie sprawdzane

Wady

- niektóre typy mutacji niedostępne
- zaawansowane techniki niedostępne

Redukcja kosztów obliczeniowych

Mutacje postkompilacyjne

- zmieniany kod pośredni (bytecode) zamiast źródłowego
- odczyt stanu wbudowywany (injected) jako instrumentacja

Zalety

- szybkie tworzenie mutantów
- standardowy kompilator
- łatwa implementacja
- dostępne zaawansowane techniki redukcji liczby mutantów

Wady

- ograniczone przez postać pośrednią
- niektóre typy mutacji niedostępne

Redukcja kosztów obliczeniowych

Mutacje postkompilacyjne

- zmieniany kod pośredni (bytecode) zamiast źródłowego
- odczyt stanu wbudowywany (injected) jako instrumentacja

Zalety

- szybkie tworzenie mutantów
- standardowy kompilator
- łatwa implementacja
- dostępne zaawansowane techniki redukcji liczby mutantów

Wady

- ograniczone przez postać pośrednią
- niektóre typy mutacji niedostępne

Redukcja kosztów obliczeniowych

Metody równoległe

- wiele mutacji testowanych jednocześnie
- modele SIMD, MIMD lub rozproszony

Mutacje równoważne

Typowe mutanty równoważne

- mutacja nieosiągalna (dead code)
- mutacja nieosiągalna (niespełnialne warunki)
- mutacja nieobserwowalna w sensie przepływu danych
- mutacja wpływa na wydajność, ale nie na wynik

Wykrywanie mutantów równoważnych

- ręczne
- (de-)optymalizacja kodu
- spełnialność więzów
- przekroje (program slicing)
- algorytmy współ-ewolucyjne

Mutacje równoważne

Typowe mutanty równoważne

- mutacja nieosiągalna (dead code)
- mutacja nieosiągalna (niespełnialne warunki)
- mutacja nieobserwowalna w sensie przepływu danych
- mutacja wpływa na wydajność, ale nie na wynik

Wykrywanie mutantów równoważnych

- ręczne
- (de-)optymalizacja kodu
- spełnialność więzów
- przekroje (program slicing)
- algorytmy współ-ewolucyjne

Wykrywanie mutantów równoważnych

Wykrywanie przez (de-)optymalizację kodu

- optymalizacja jako normalizacja kodu
- skuteczność zależna od opcji kompilacji
- przy integracji generowania mutacji z kompilacją możliwa „równoległa optymalizacja”

Wykrywanie mutantów równoważnych

Problem spełnialności więzów

- warunek wystarczający: różny stan końcowy
- warunek konieczny: różny stan bezpośrednio po mutacji
- warunek konieczny: mutacja jest osiągalna

Typowe metody analizy

- domain splitting
- redukcja stałych
- analiza zakresów

Wykrywanie mutantów równoważnych

Wykrywanie przez przekroje programu

- stosowane do słabych mutacji
- tworzymy sztuczną zmienną logiczną $z := \text{true}$
- modyfikacja z po mutacji (sprawdzamy identyczność stanu)
- punkt przekroju dla zbioru $\{z\}$, bezpośrednio po tej modyfikacji

Przykłady

Mutacja modyfikująca wyrażenie e na e':

$$z := z \wedge (e = e')$$

Mutacja modyfikująca przypisanie z $x := e$ na $x' := e$:

$$z := z \wedge (x = e \wedge x' = e)$$

Wykrywanie mutantów równoważnych

Wykrywanie przez przekroje programu

- stosowane do słabych mutacji
- tworzymy sztuczną zmienną logiczną $z := \text{true}$
- modyfikacja z po mutacji (sprawdzamy identyczność stanu)
- punkt przekroju dla zbioru $\{z\}$, bezpośrednio po tej modyfikacji

Przykłady

Mutacja modyfikująca wyrażenie e na e':

$$z := z \wedge (e = e')$$

Mutacja modyfikująca przypisanie z $x := e$ na $x' := e$:

$$z := z \wedge (x = e \wedge x' = e)$$

Wykrywanie mutantów równoważnych

Wykrywanie przez przekroje programu

- stosowane do słabych mutacji
- tworzymy sztuczną zmienną logiczną $z := \text{true}$
- modyfikacja z po mutacji (sprawdzamy identyczność stanu)
- punkt przekroju dla zbioru $\{z\}$, bezpośrednio po tej modyfikacji

Przykłady

Mutacja modyfikująca wyrażenie e na e':

$$z := z \wedge (e = e')$$

Mutacja modyfikująca przypisanie z $x := e$ na $x' := e$:

$$z := z \wedge (x = e \wedge x' = e)$$

Wykrywanie mutantów równoważnych

Optymalizacja obliczania przekrojów

- operator mutacji zachowuje odwołania (reference preserving), gdy dotyczy wyrażenia i nie zmienia zbioru występujących w nim zmiennych
- przy stosowanych algorytmach obliczania przekroju takie operatory komutują z operacją przekroju
- $\Rightarrow$ można obliczyć przekroje dla wielu mutacji na raz

Wymuszenie komutowania dla innych operatorów

- mutacja zmienia s na s' w punkcie p
- chcemy mierzyć przekrój (X, q)
- tworzymy nową zmienną y i nowy węzeł r, umieszczony bezpośrednio po p
- w r przypisujemy $y := (y_1, y_2, \ldots, y_k)$, gdzie $\{y_i\}$ to zbiór zmiennych istotnych dla dokładnie jednej z instrukcji s i s'
- zmieniamy przekrój (X, q) na $(X \cup \{y\}, q)$