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

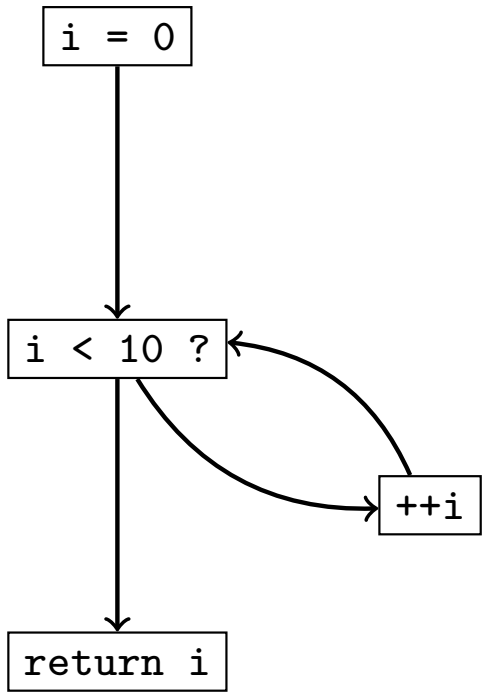
- liniowy ciąg instrukcji
- skoki „na zewnątrz” tylko z ostatniej
- skoki „do wewnątrz” tylko do pierwszej

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

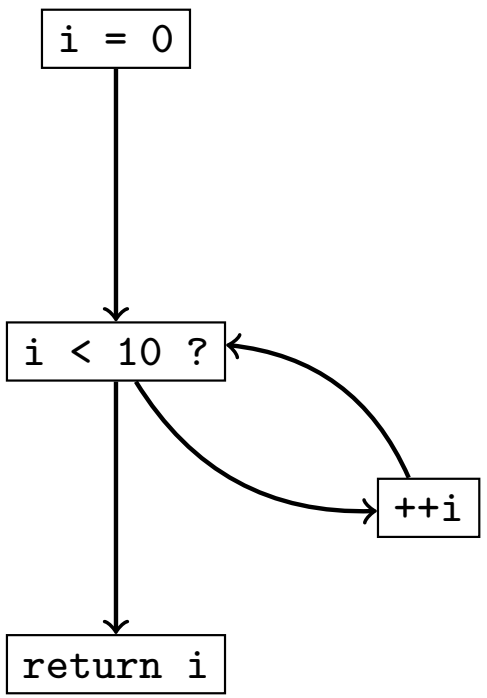
- tworzą drzewo o korzeniu w s
- można wyznaczyć w czasie prawie liniowym

$v \in$ 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

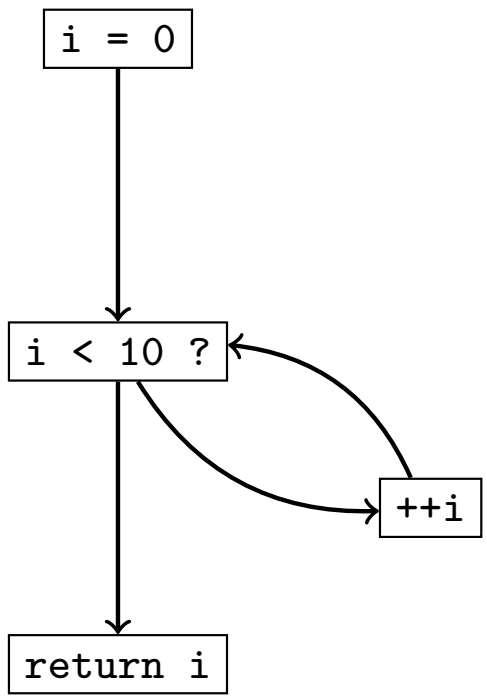
- tworzą drzewo o korzeniu w s
- można wyznaczyć w czasie prawie liniowym

$v \in$ 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

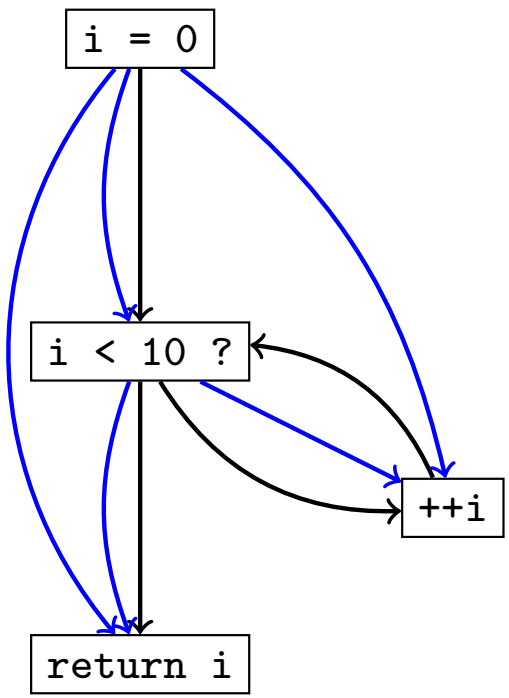
- tworzą drzewo o korzeniu w s
- można wyznaczyć w czasie prawie liniowym

$v \in$ 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

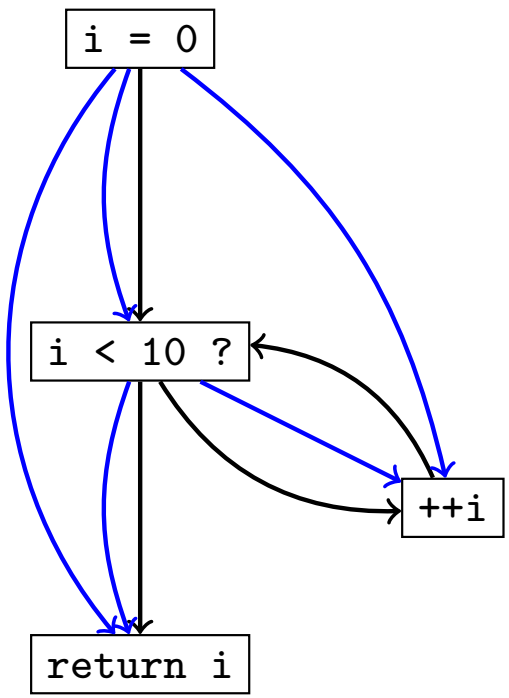
- tworzą drzewo o korzeniu w s
- można wyznaczyć w czasie prawie liniowym

$v \in$ 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

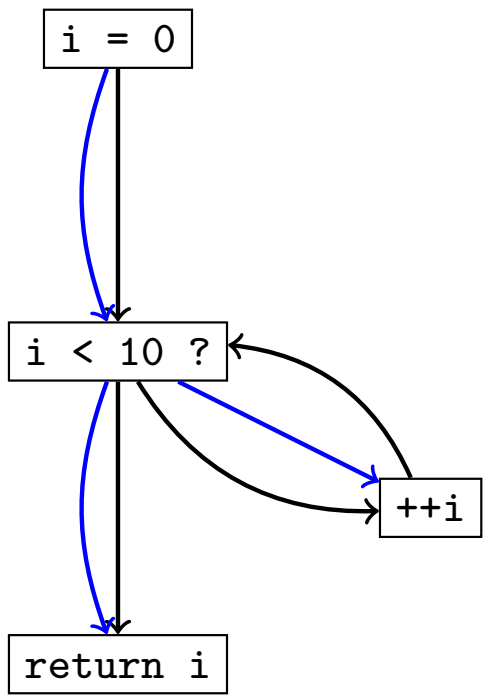
- tworzą drzewo o korzeniu w s
- można wyznaczyć w czasie prawie liniowym

$v \in$ 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

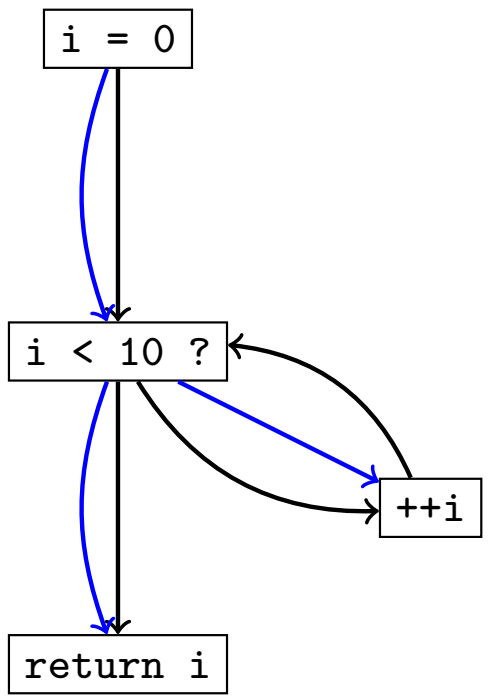
- tworzą drzewo o korzeniu w s
- można wyznaczyć w czasie prawie liniowym

$v \in$ 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

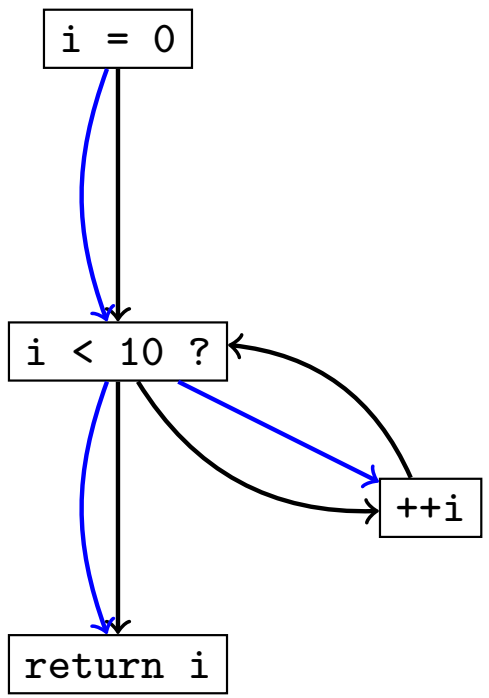
- tworzą drzewo o korzeniu w s
- można wyznaczyć w czasie prawie liniowym

$v \in$ 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

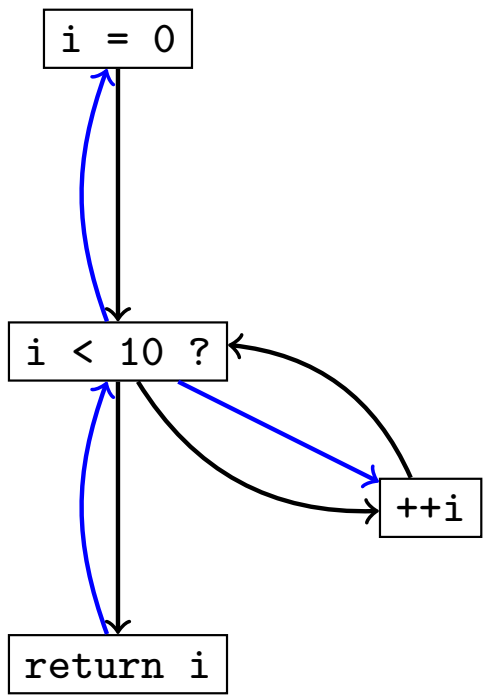
- tworzą drzewo o korzeniu w s
- można wyznaczyć w czasie prawie liniowym

$v \in$ DOMINANCE FRONTIER(u)

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

POSTDOMINACJA

to dominacja od t po odwróconych krawędziach



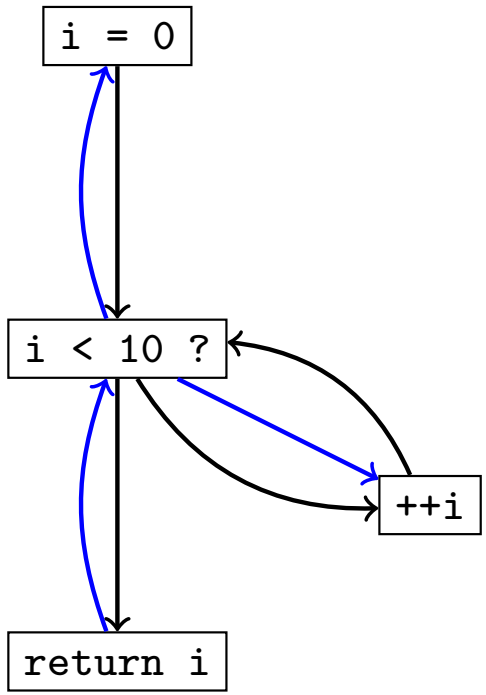
Control Dependence Graph

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



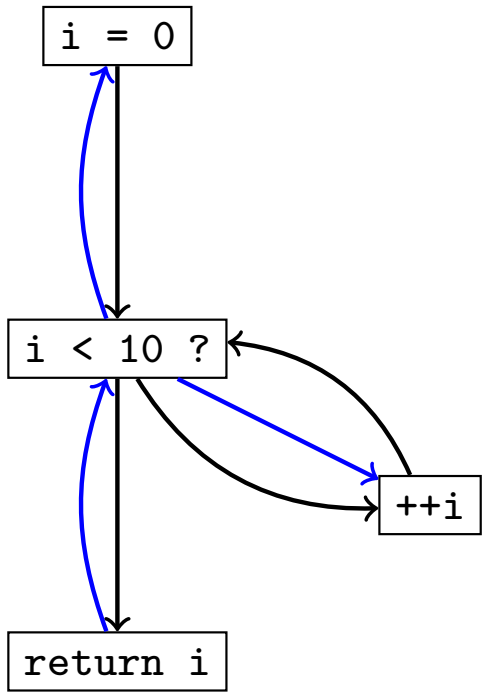
Control Dependence Graph

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



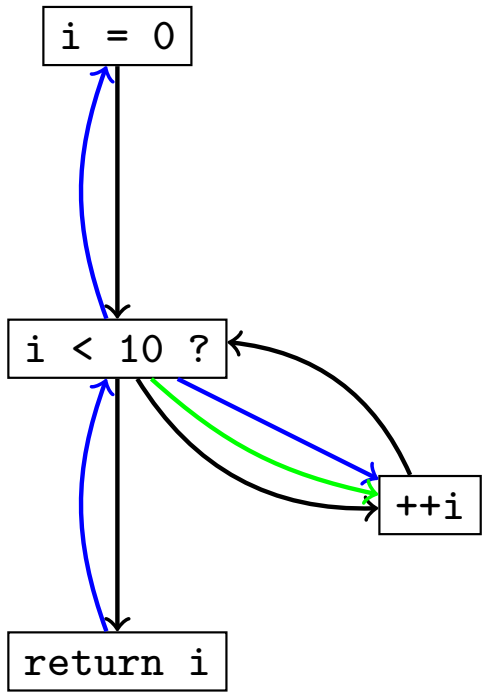
Control Dependence Graph

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

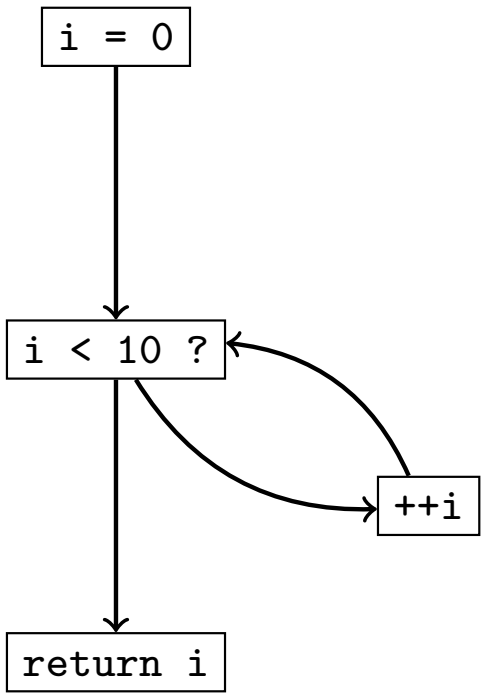
to instrukcja odczytująca x

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

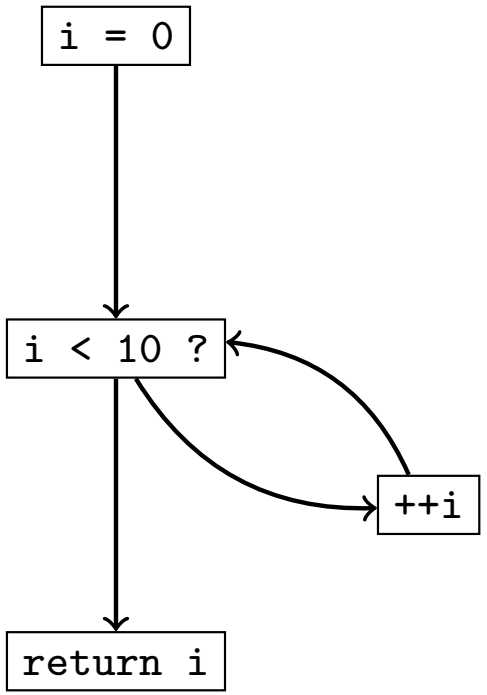
to instrukcja odczytująca x

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

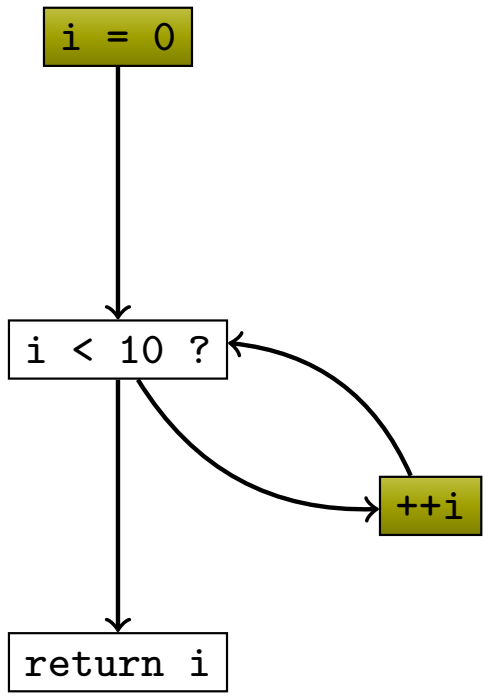
to instrukcja odczytująca x

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

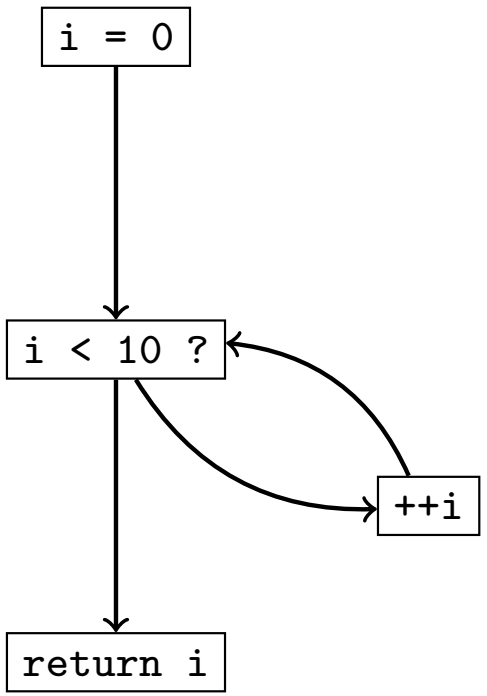
to instrukcja odczytująca x

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

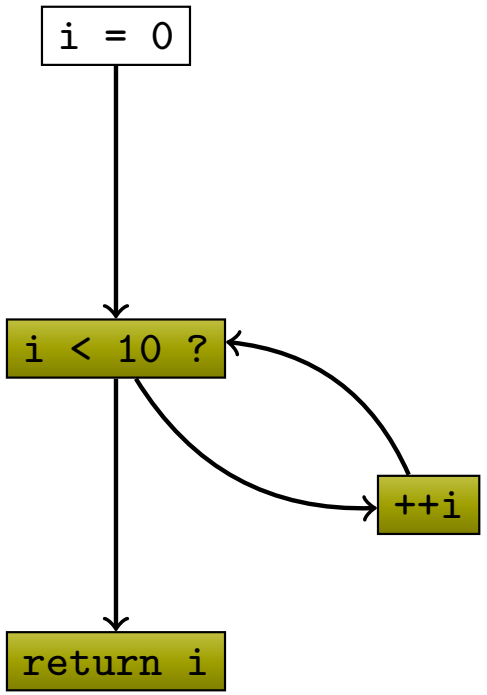
to instrukcja odczytująca x

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

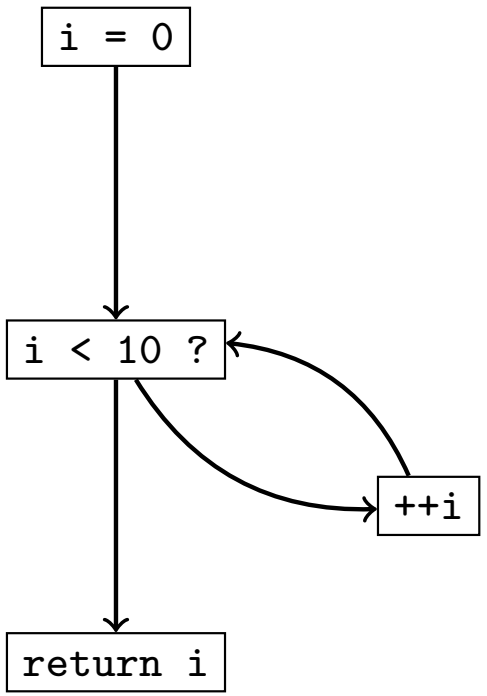
to instrukcja odczytująca x

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

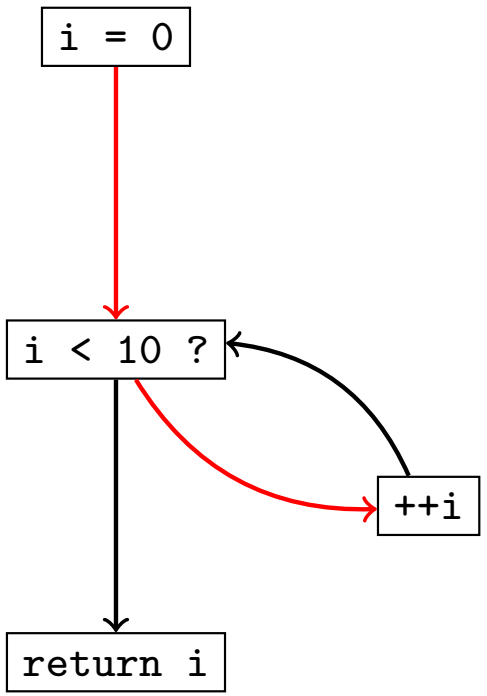
to instrukcja odczytująca x

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

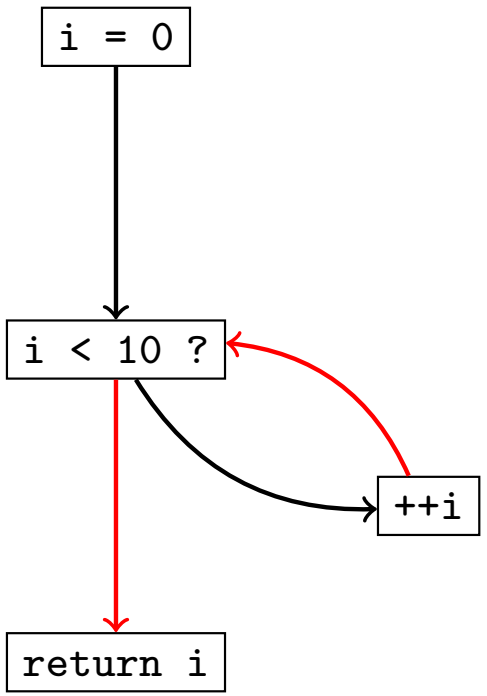
to instrukcja odczytująca x

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

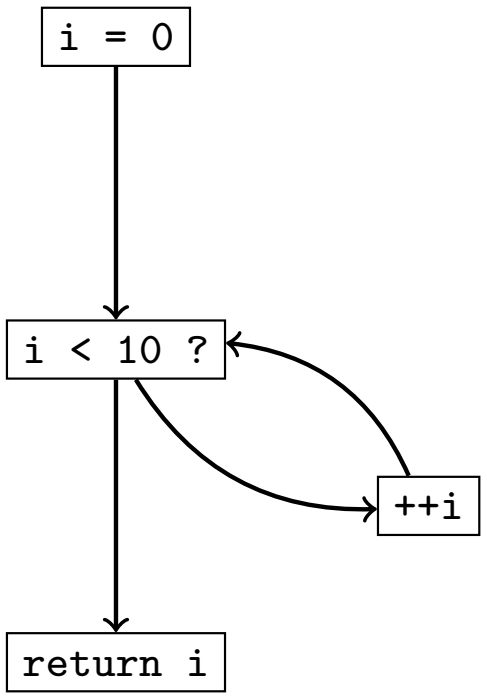
to instrukcja odczytująca x

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

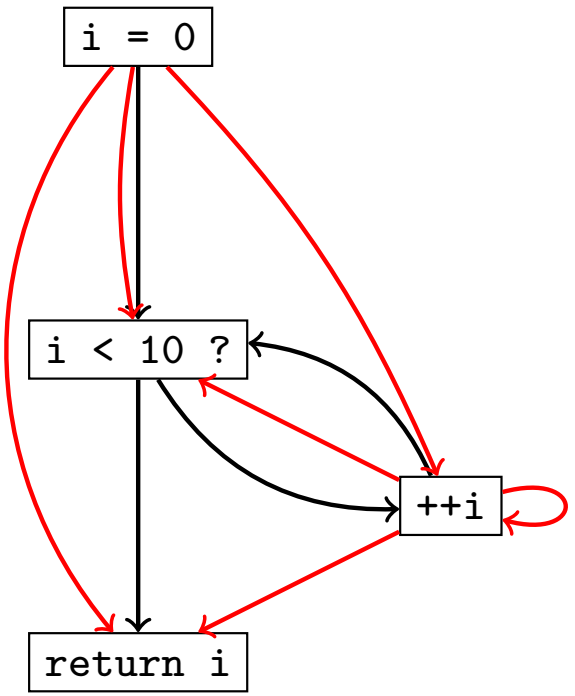
to instrukcja odczytująca x

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

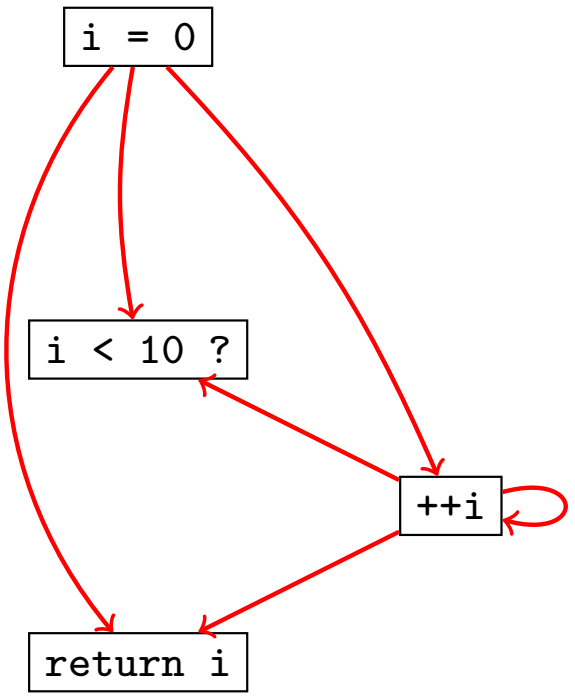
to instrukcja odczytująca x

Ścieżka wolna dla x

to ścieżka w CFG omijająca definicje x

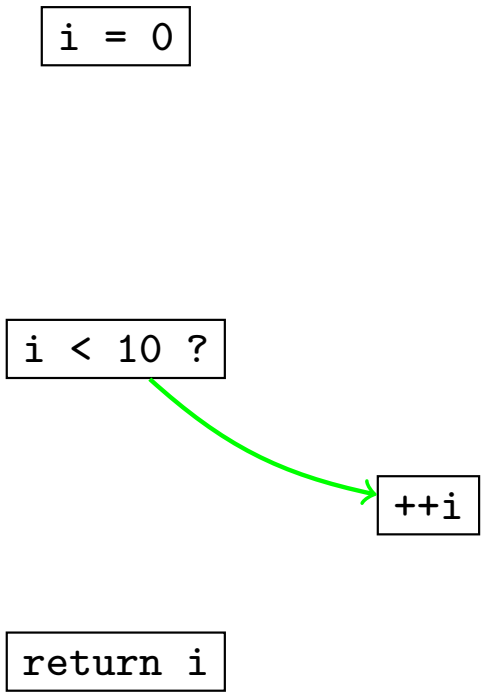
Krawędź $u \to v$ w DFG

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

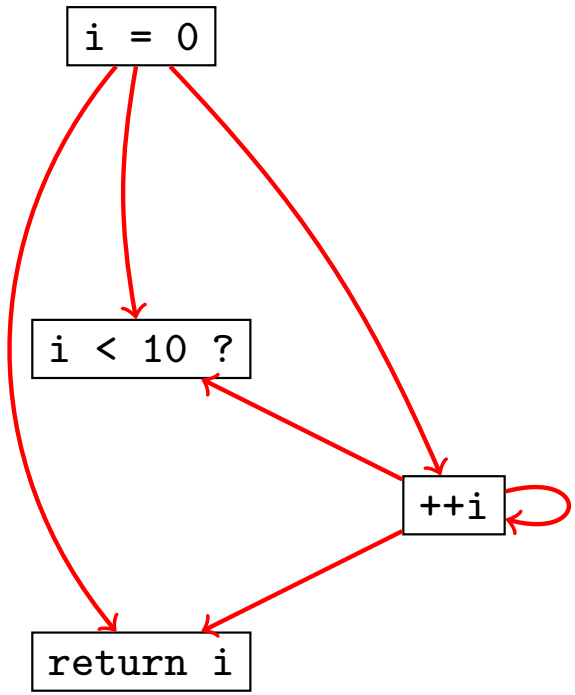


Program Dependence Graph

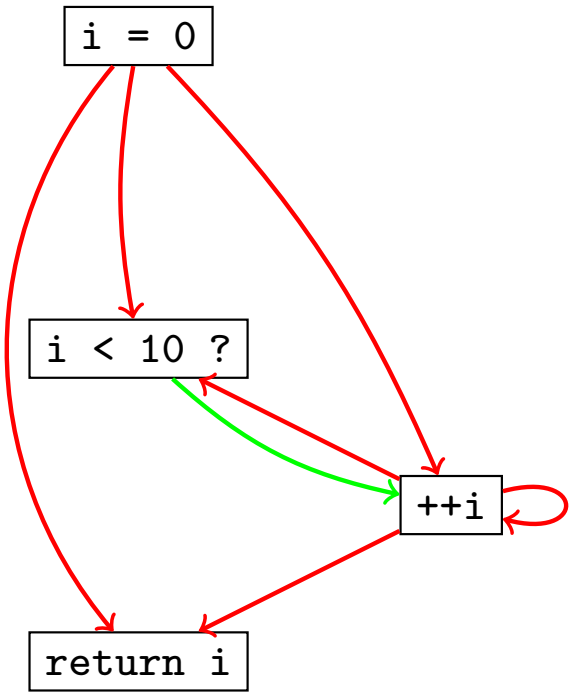
Control Dependence



Data Dependence



Program Dependence Graph



Static Single Assignment form

Definicja zmiennej x

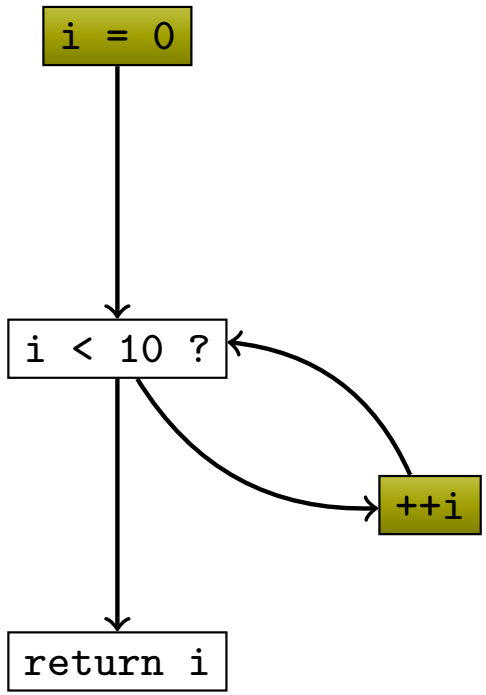
to instrukcja ustawiająca x

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

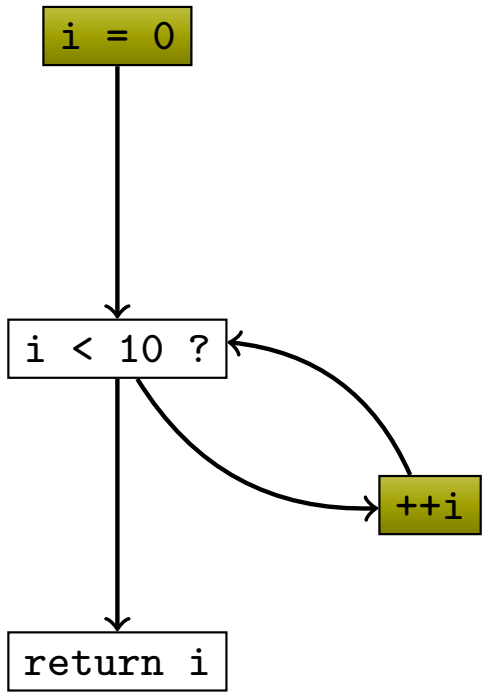
to instrukcja ustawiająca x

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

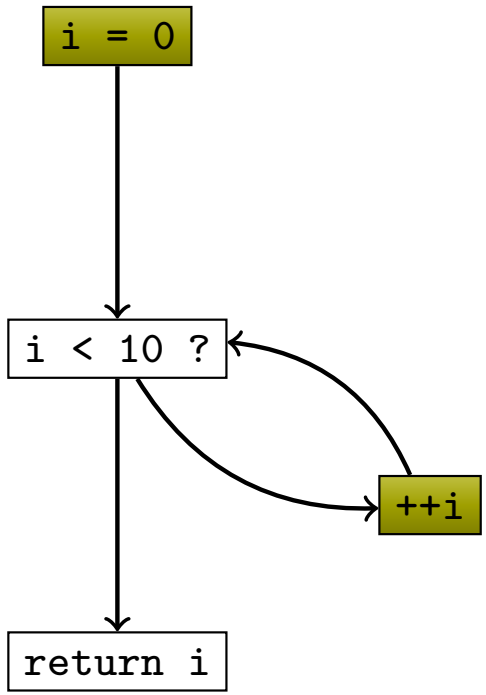
to instrukcja ustawiająca x

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

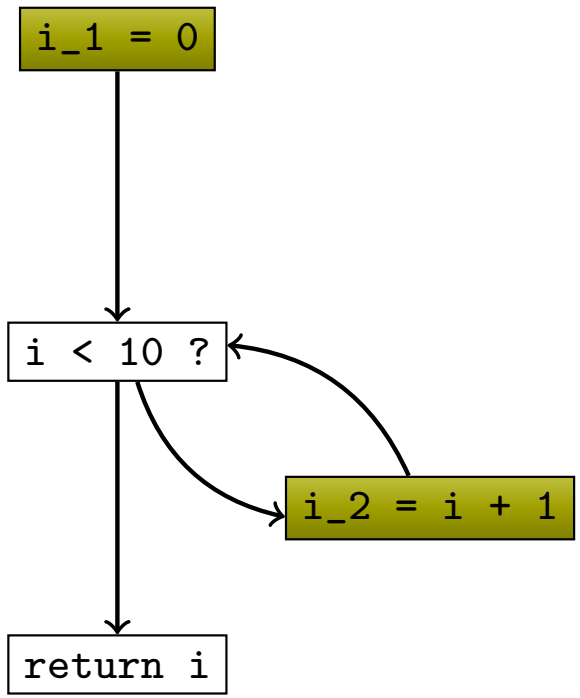
to instrukcja ustawiająca x

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

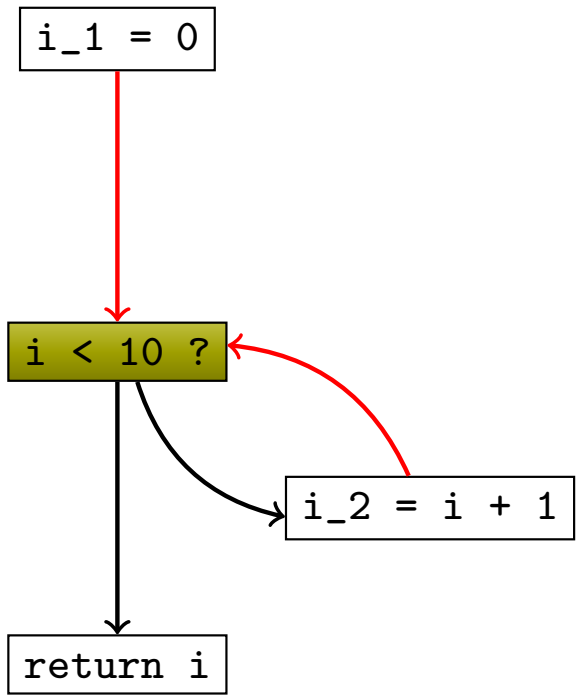
to instrukcja ustawiająca x

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

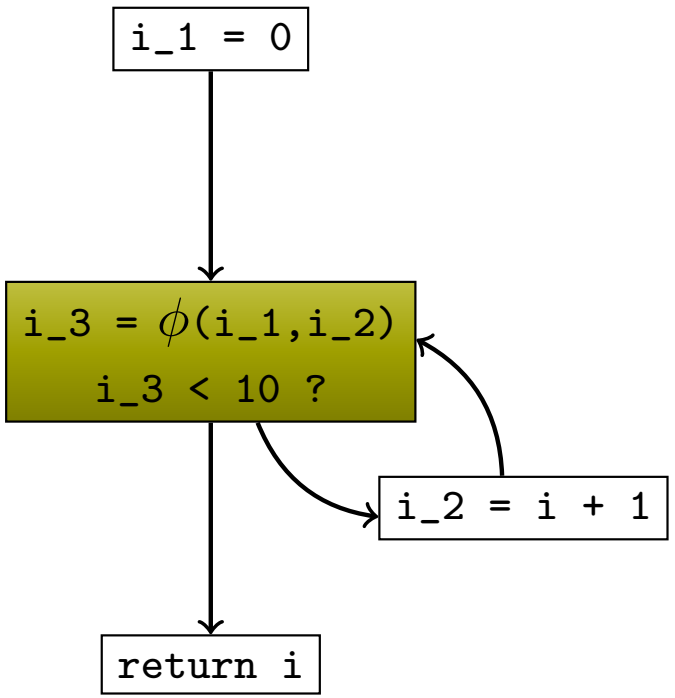
to instrukcja ustawiająca x

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

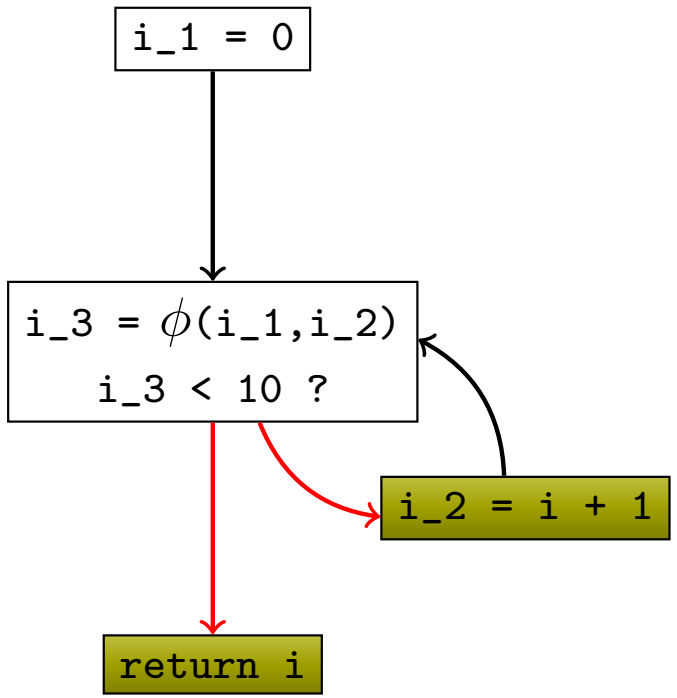
to instrukcja ustawiająca x

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

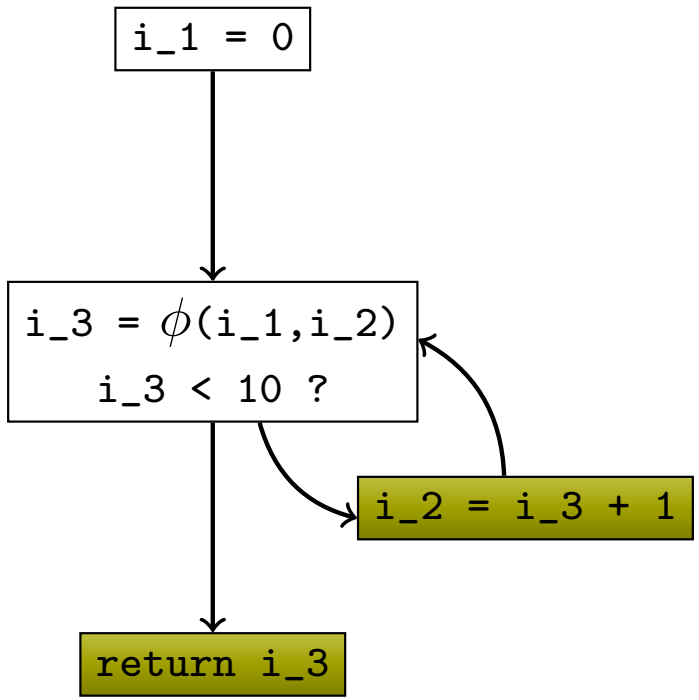
to instrukcja ustawiająca x

Ograniczenie SSA

każda zmienna ma dokładnie 1 definicję

Konstrukcja

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



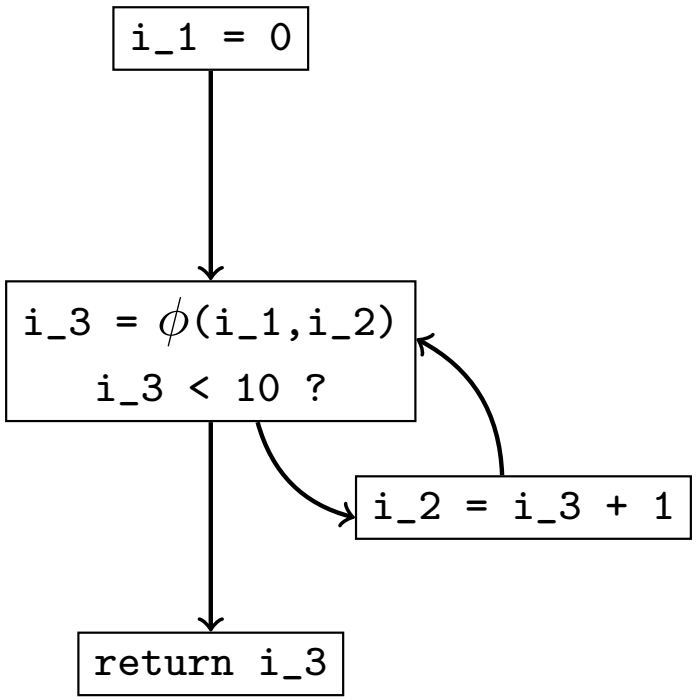
Static Single Information form

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



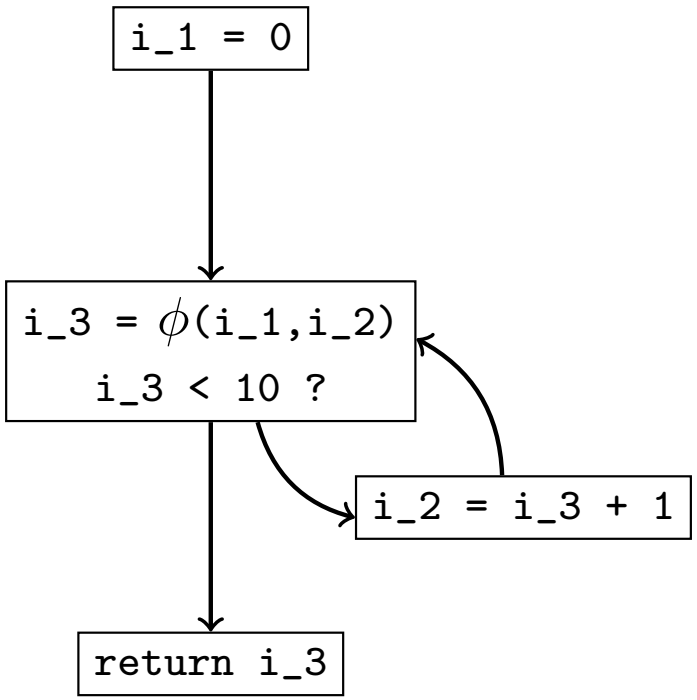
Static Single Information form

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



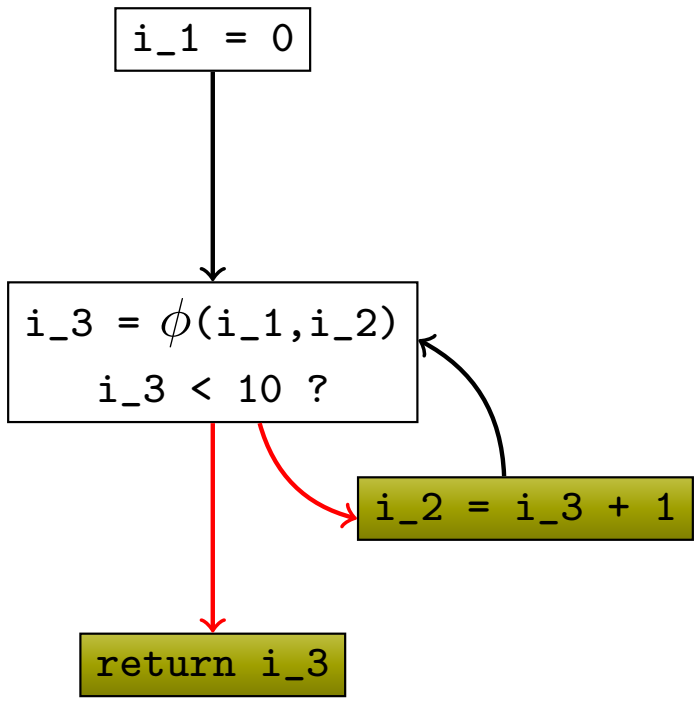
Static Single Information form

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



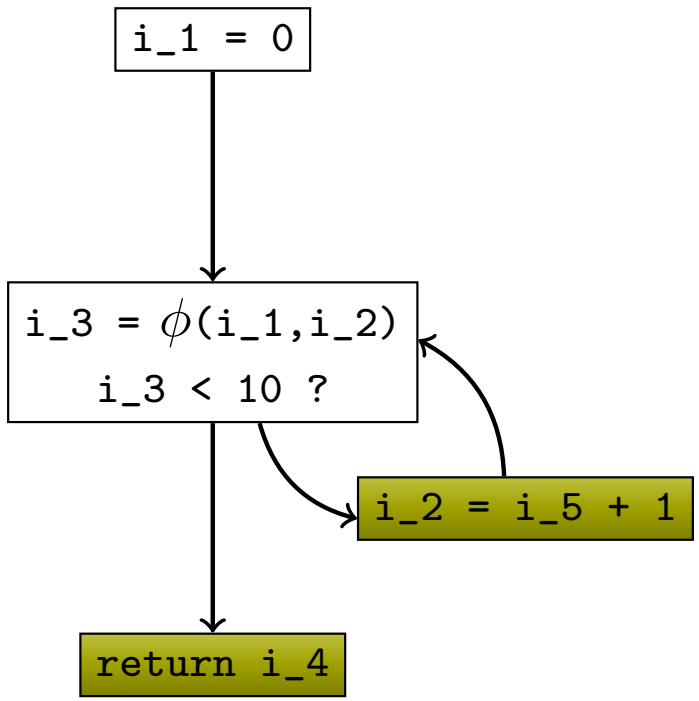
Static Single Information form

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



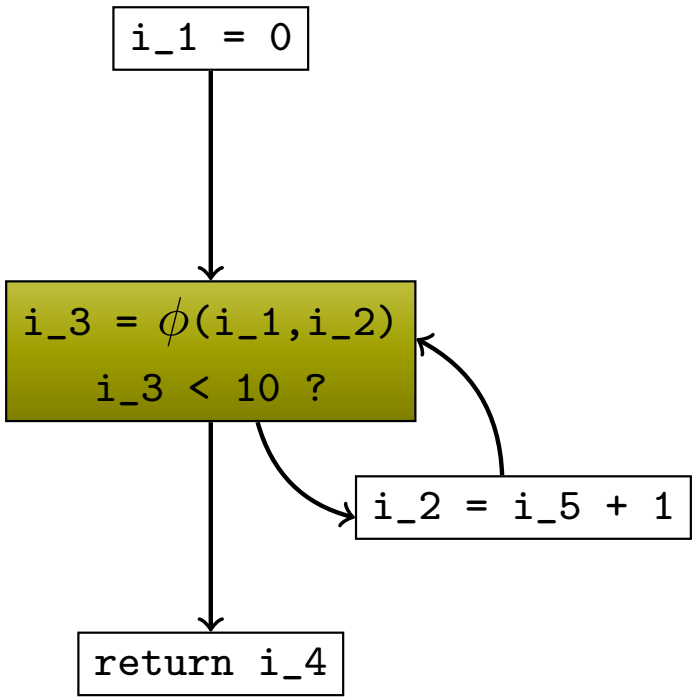
Static Single Information form

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



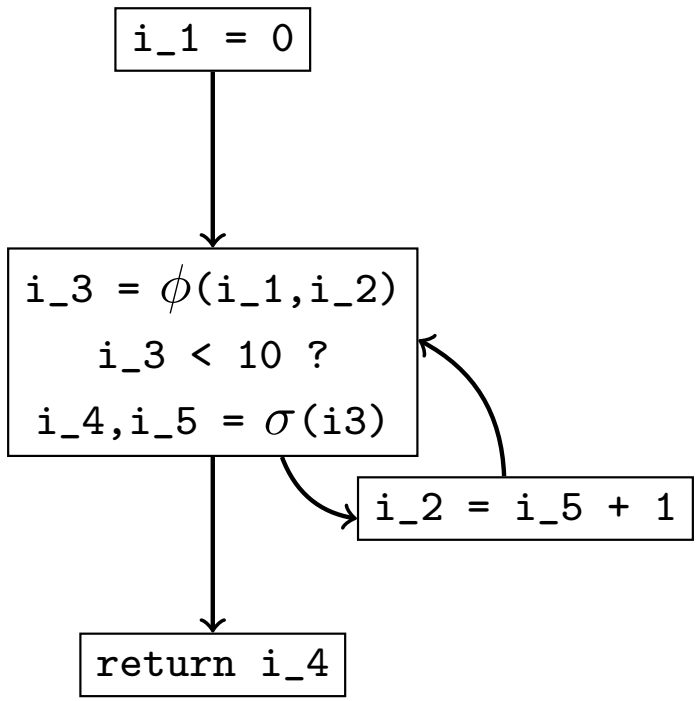
Static Single Information form

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



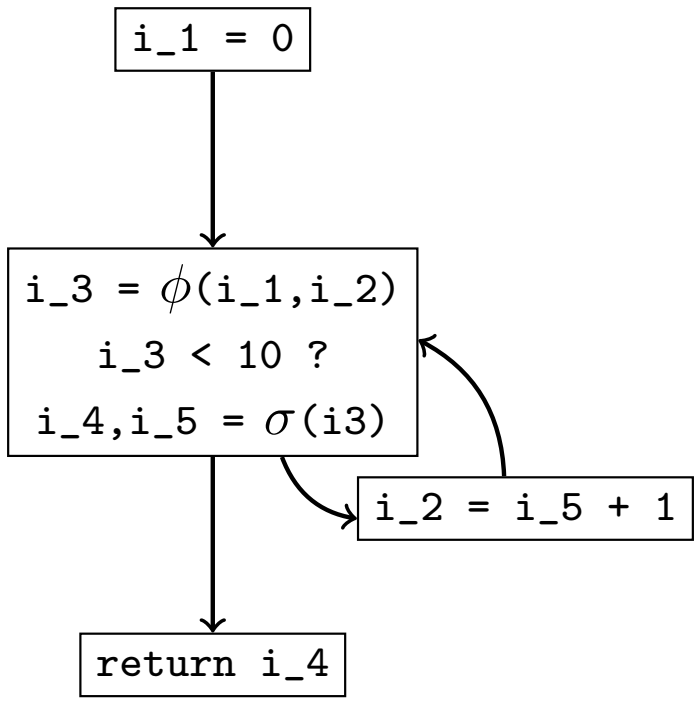
Static Single Information form

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)