Program Slicing

2. Program Slicing
   - Das tägliche Brot des Wartungsprogrammierers
   - Was ist ein Slice?
   - Slicing-Technik
   - Intraprozedurales Slicing
   - Interprozedurales Slicing
   - Summary-Edges im SDG
   - Slicing-Varianten
   - Anwendungen von Slicing

Slicing

Program Slicing

- Lernziele
  - Verständnis von Slicing-Varianten einschließlich ihrer Berechnung und Anwendbarkeit
- Kontext
  - Erste unmittelbare Anwendung von Compilerbau-Technologie zur Unterstützung des Programmverstehens
  - Program Slicing ist Basis-Technologie für viele weitere Reengineering-Techniken

Das tägliche Brot des Wartungsprogrammierers

```
read (n);
i:= 1;
sum := 0;          -- Was beeinflusst diese Anweisung?
product := 1;
while i <= n loop
          sum := sum + i;
          product := product * i;
          i := i + 1;
end loop;
write (sum);
write (product); -- Wie kommt es zu diesem Wert?
```

Wie kommt es zu diesem Wert?

Backward Slice (write (product)):

```
read (n);
i:= 1;
sum := 0;
product := 1;
while i <= n loop
          sum := sum + i;
          product := product * i;
          i := i + 1;
end loop;
write (sum);
write (product); -- Wie kommt es zu diesem Wert?
```

Wie kommt es zu diesem Wert?

Backward Slice (write (product)):

```
read (n);
i:= 1;
sum := 0;
product := 1;
while i <= n loop
            sum := sum + i;
            product := product * i;
            i := i + 1;
end loop;
write (sum);
write (product); -- Wie kommt es zu diesem Wert?
```

Was wird durch diese Answeisung beeinflust?

Forward Slice (sum := 0):

```
read (n)
i:= 1;
sum := 0;          -- Was beeinflusst diese Anweisung?
product := 1;
while i <= n loop
          sum := sum + i;
          product := product * i;
          i := i + 1;
end loop;
write (sum);
write (product);
```

Was wird durch diese Answeisung beeinflust?

Forward Slice (sum := 0):

```
read (n);
i:= 1;
sum := 0;              -- Was beeinflusst diese Anweisung?
product := 1;
while i <= n loop
                sum := sum + i;
                product := product * i;
                i := i + 1;
end loop;
write (sum);
write (product);
```

Was ist ein Slice?

Slicing (P,S) bezüglich eines Programmpunktes S entfernt jene Teile eines Programms P, die das Verhalten an S nicht beeinflussen.

- Ein **Forward Slice** enthält nur solche Anweisungen von P, die von der Ausführung von S beeinflusst werden.
- Ein **Backward Slice** enthält nur solche Anweisungen von P, die das Programmverhalten an S beeinflussen.

- Ein Punkt S **beeinflusst** einen Punkt S', wenn S' kontroll- oder datenabhängig von S ist.

Slicing-Technik

- Idee und erste Technik des Slicings stammt von Weiser (1984).
- Moderne Slicing-Techniken basieren Abhängigkeitsgraphen
  - Program Dependency Graph (PDG) (Ottenstein und Ottenstein 1984) für intraprozedurales Slicing
  - System Dependency Graph (SDG) für interprozedurales Slicing (Horwitz u. a. 1990)

Slicing-Technik

- Idee und erste Technik des Slicings stammt von Weiser (1984).
- Moderne Slicing-Techniken basieren Abhängigkeitsgraphen
  - Program Dependency Graph (PDG) (Ottenstein und Ottenstein 1984) für intraprozedurales Slicing
  - System Dependency Graph (SDG) für interprozedurales Slicing (Horwitz u. a. 1990)

PDG-basiertes Slicing:

- Kanten $A \rightarrow B$ im PDG: B ist daten- bzw. kontrollabhängig von A
- Forward Slicing: Graph-Traversierung in Richtung der Kontroll- und Datenabhängigkeitskanten
- Backward Slicing: Graph-Traversierung in umgekehrter Richtung der Kontroll- und Datenabhängigkeitskanten

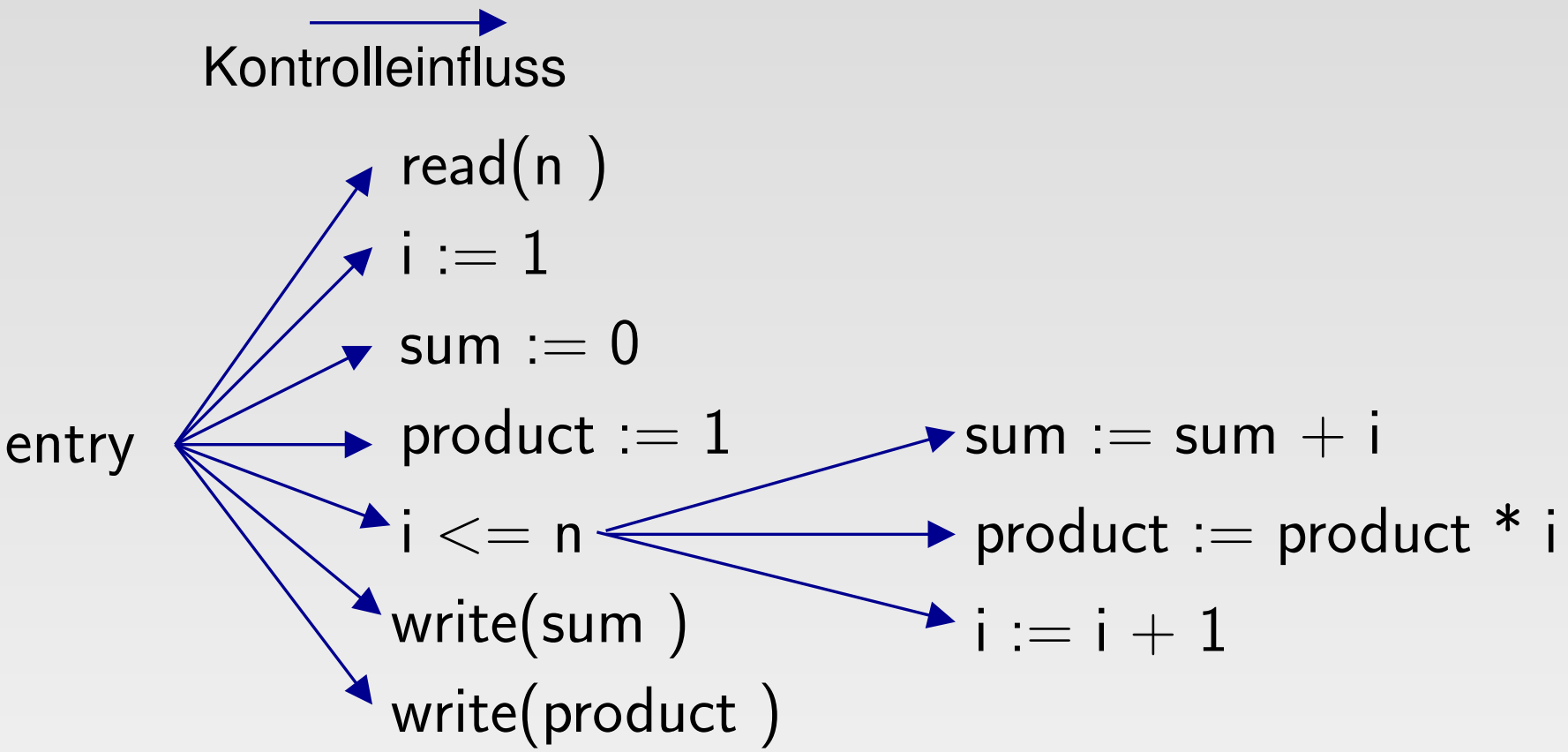
Automatisierung von Slicing für einzelne Funktionen

Program Dependency Graph (PDG) nach Ottenstein und Ottenstein (1984)

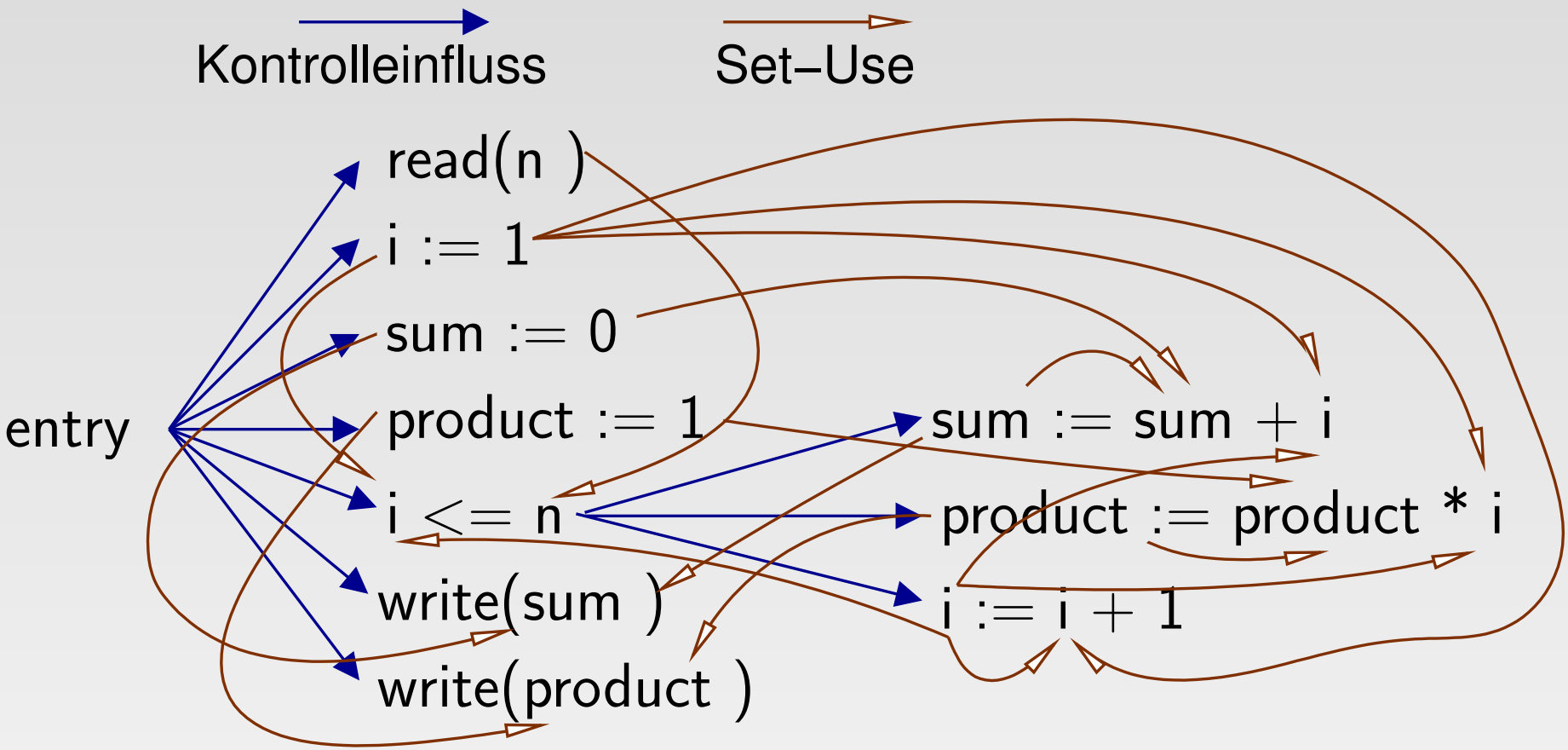
PDG = gerichteter Multigraph für Daten- und Kontrollflussabhängigkeiten.

- Knoten repräsentieren Zuweisungen und Prädikate
  - zusätzlich einen speziellen Entry-Knoten
  - zusätzlich ϕ-Knoten zur Reduktion der Datenabhängigkeitskanten
  - (sowie einen Initial-Definition-Knoten für jede Variable, die benutzt werden kann, bevor sie gesetzt wird)
- Control-Kanten repräsentieren Kontrollabhängigkeiten
  - Startknoten jeder Kontrollabhängigkeitskante ist entweder der Entry-Knoten oder ein Prädikatsknoten
- Flow-Kanten repräsentieren Set-Use-Abhängigkeiten

Program Slicing mit PDG



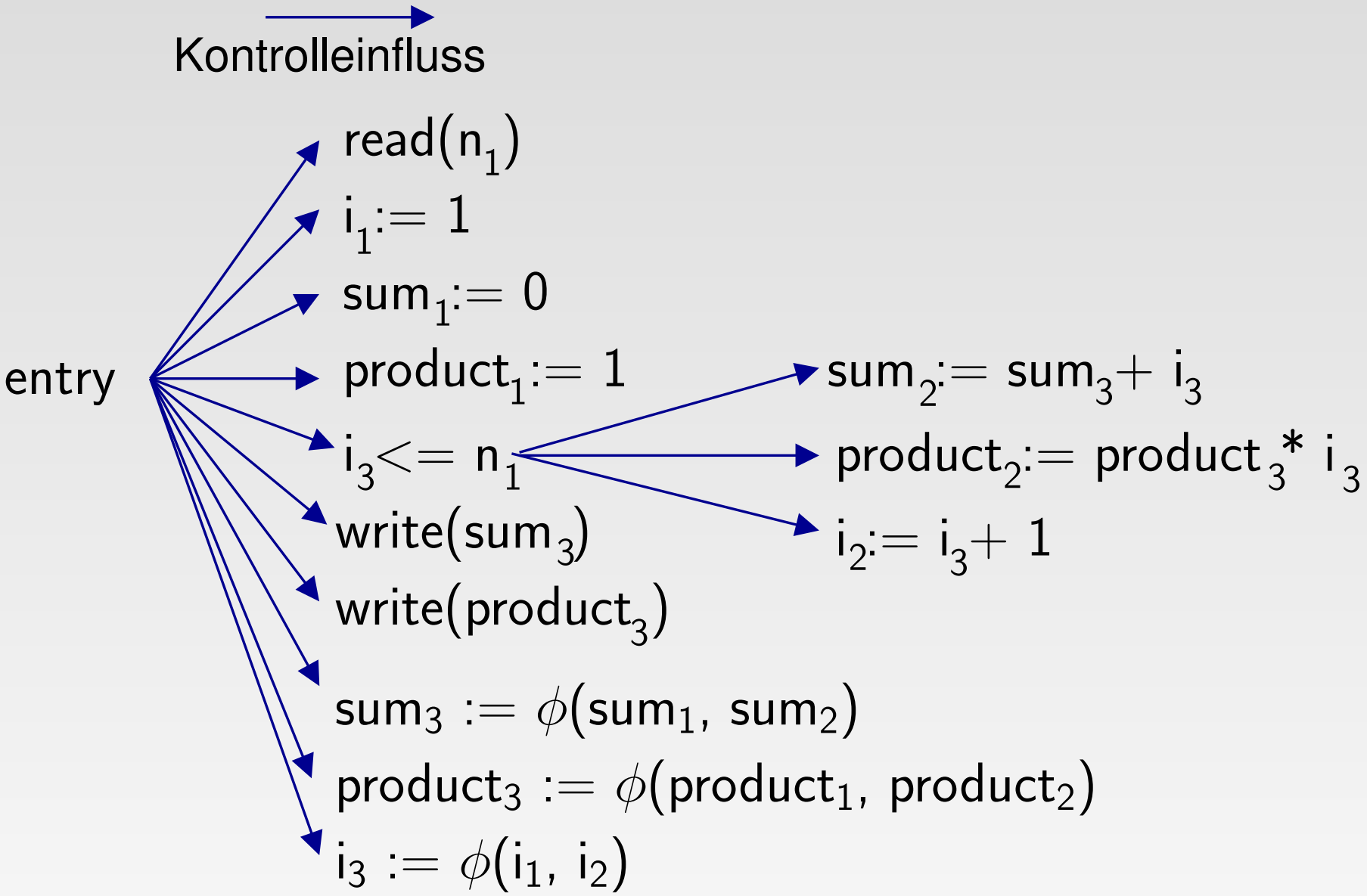
Program Slicing mit PDG



Synthese von PDG und SSA

- PDG wird aufgebaut für Kontrollflussgraph mit SSA-Form
  - ϕ-Knoten erscheinen als Zuweisungsknoten im PDG
  - führt in der Regel zur Reduktion von Flow-Kanten
  - Set-Set-Kanten werden nicht benötigt (ϕ-Knoten führen zur korrekten Serialisierung)

Program Slicing mit PDG/SSA



Automatisierung von Slicing für große Programme

Interprozedurales Slicing

```
procedure Main is
  sum, i : Integer;
begin
 sum := 0;
 i := 1;
 while i < 11 loop
   A (sum, i);
 end loop;
 put (sum);
end Main;

procedure A
(x : in out Integer;
 y : in out Integer) is
begin
 Add (x, y);
 Inc (y);
end A;
```

```
procedure Add
 (a : in out Integer;
  b : in out Integer) is
begin
 a := a + b;
end Add;

procedure Inc
  (z : in out Integer) is
 one : Integer := 1;
begin
 Add (z, one);
end Inc;
```

Annahme: Copy-in/Copy-out-Parameterübergabe

System Dependency Graph (SDG) nach Horwitz u. a. (1990)

- PDG stellt Abhängigkeiten innerhalb einer Funktion dar
- System Dependency Graph (SDG) stellt globale Abhängigkeiten dar: PDGs für verschiedene Unterprogramme werden vernetzt über interprozedurale Kontroll- und Datenflusskanten
  - Aufruf: Call-Knoten
  - aktuelle Parameter: actual-in / actual-out-Knoten (copy-in/copy-out-Parameterübergabe vorausgesetzt)
  - formale Parameter: formal-in / formal-out-Knoten
  - transitive Abhängigkeiten via PDG: Summary-Edges

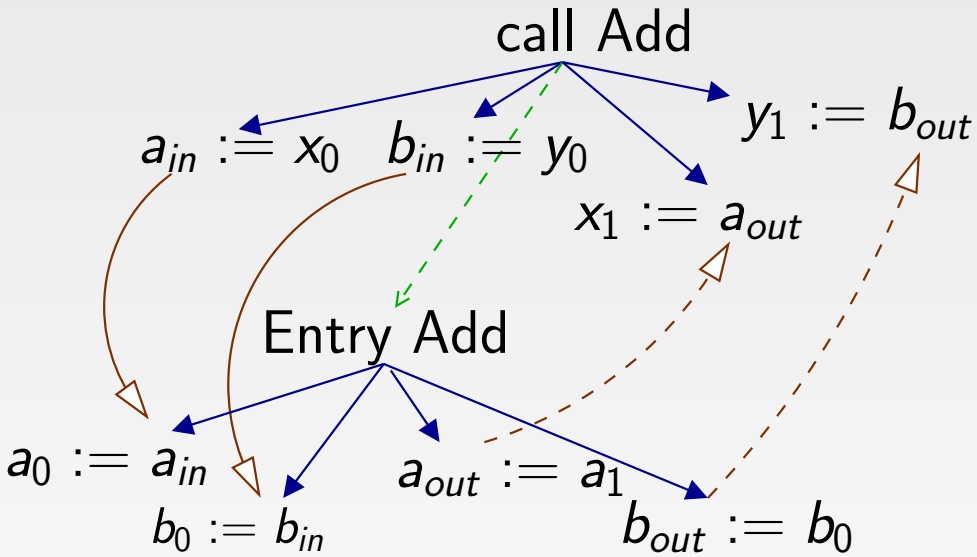
Modellierung von Prozeduraufrufen

Add (x, y) ⇔

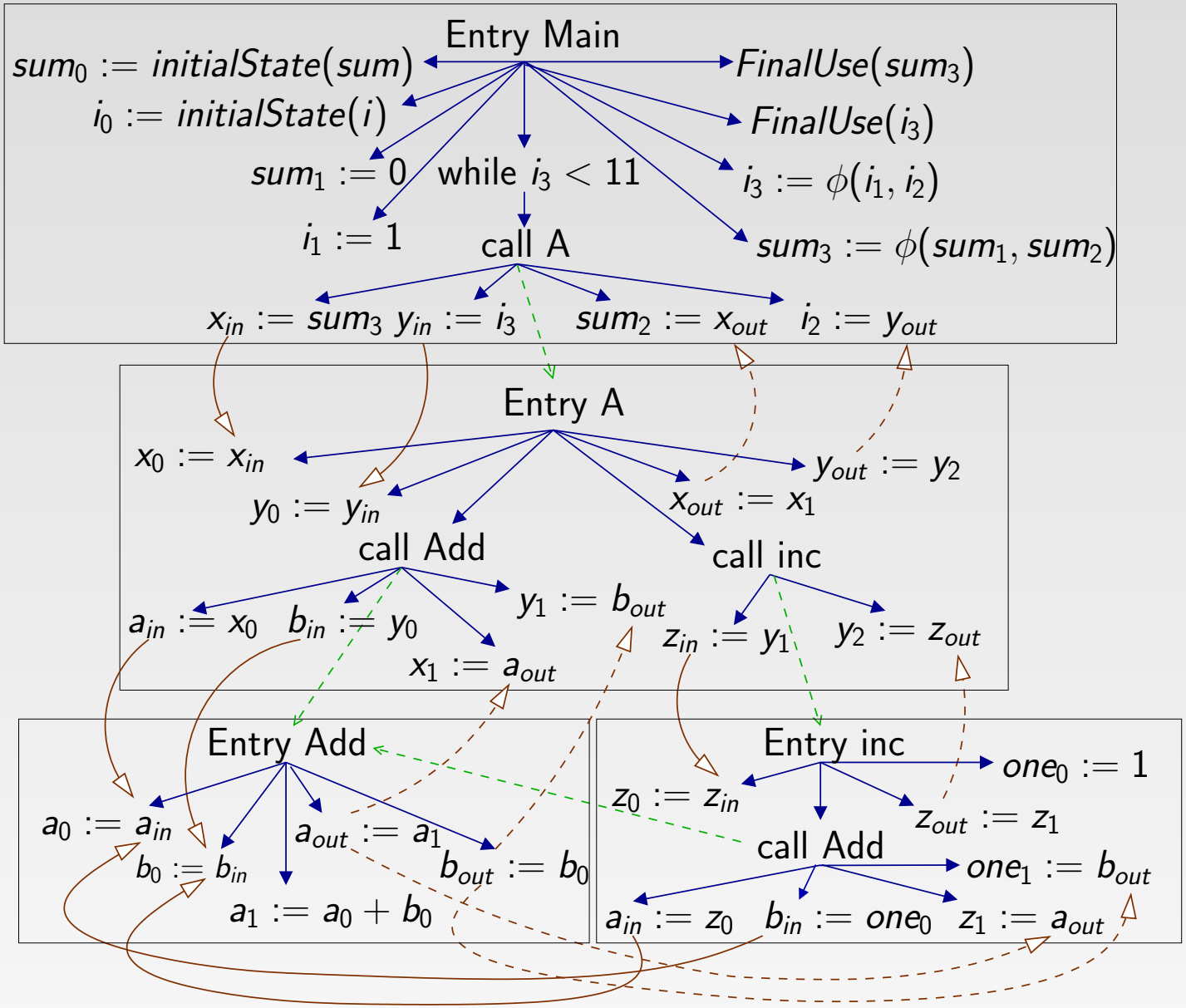
$a_{in} := x_0;$
$b_{in} := y_0;$
Add;
$x_1 := a_{out};$
$y_1 := b_{out};$

```
procedure Add
(x : in out Integer;
 y : in out Integer) is ⇔

procedure Add is
begin
a0 := ain; b0 := bin;
...
aout := a1; bout := b1;
end A;
```

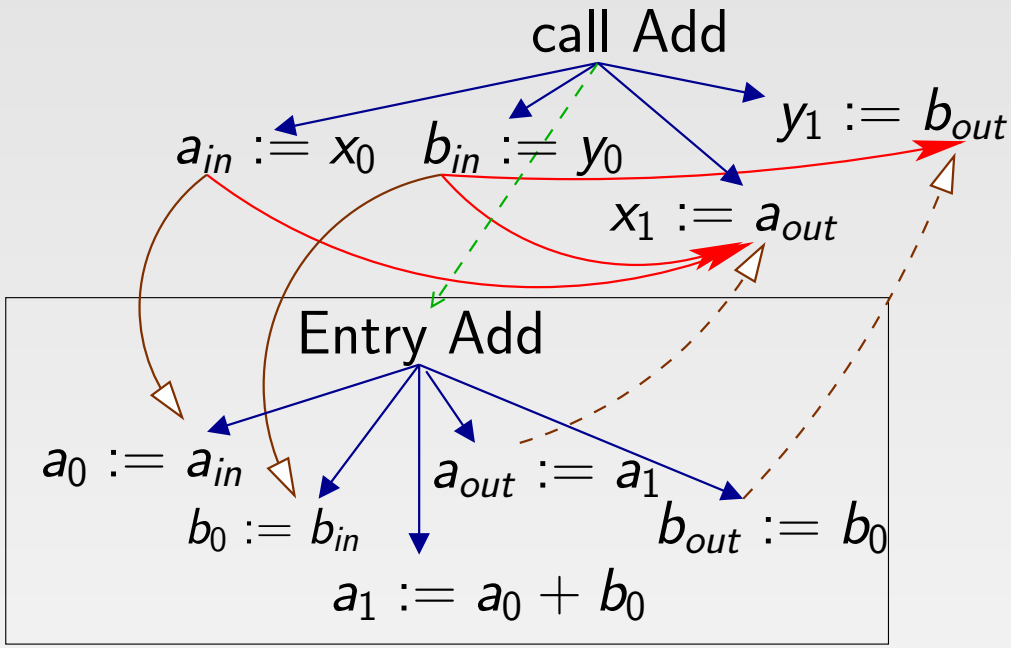


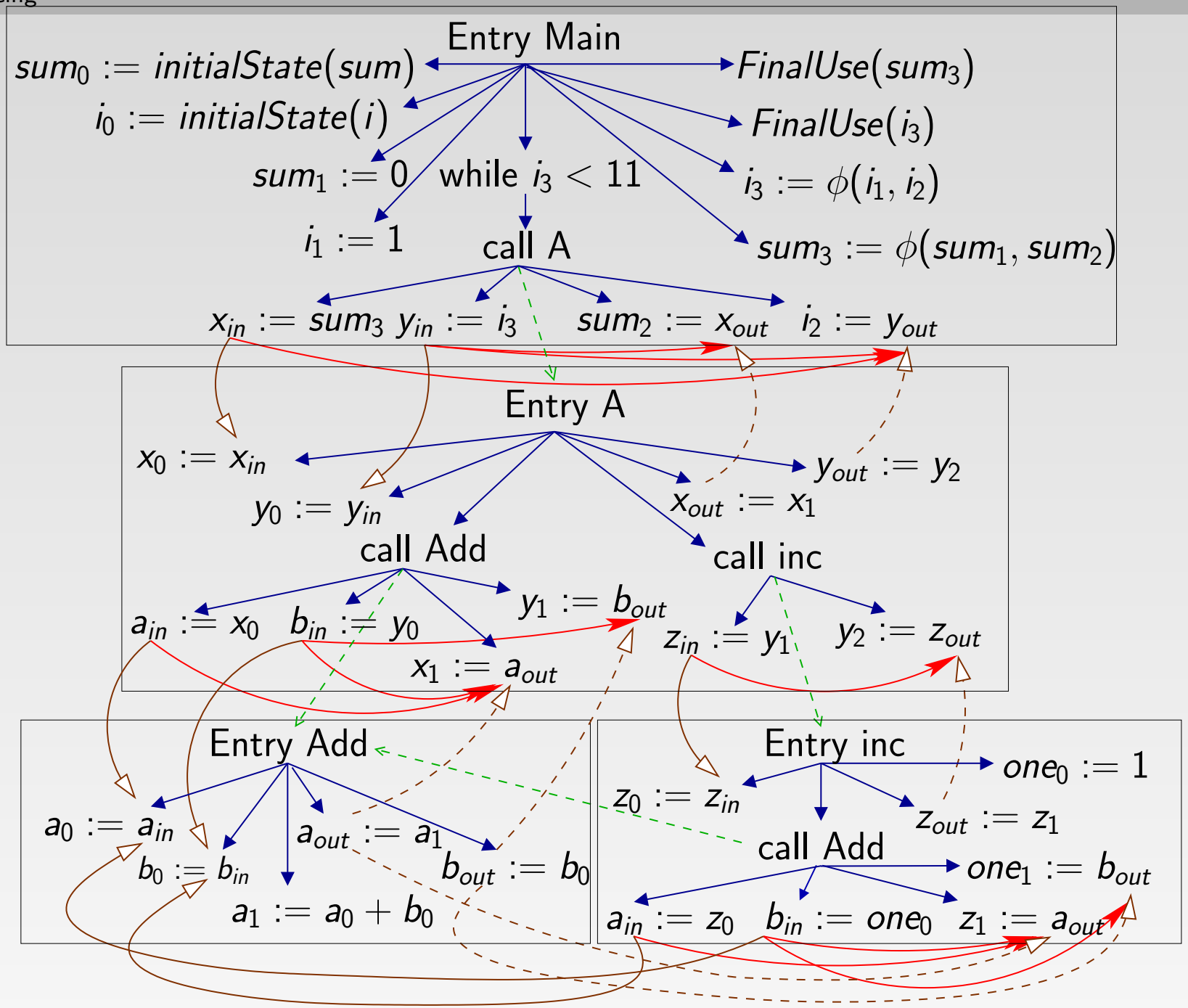
Naives Slicing: Folge allen Flow- und Control-Kanten



Summary-Edges

Summary-Edges sind spezielle Flow-Kanten und beschreiben die Abhängigkeit der aktuellen out-Parameter von den aktuellen in-Parametern im Aufrufkontext.

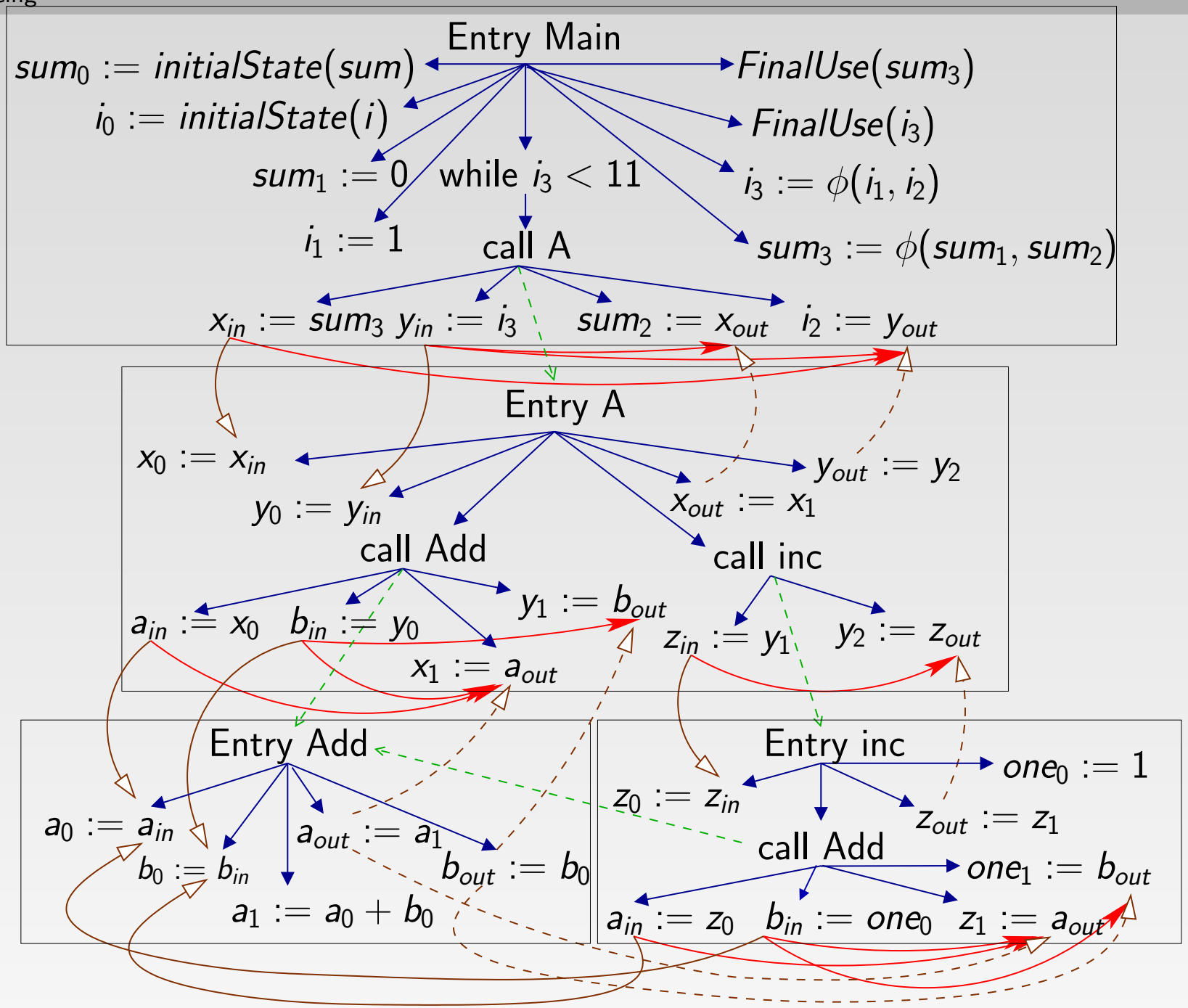



Entry Main
$sum_0 := initialState(sum)$
$i_0 := initialState(i)$
$sum_1 := 0$
while $i_3 < 11$
$i_1 := 1$
call A
$FinalUse(sum_3)$
$FinalUse(i_3)$
$i_3 := \phi(i_1, i_2)$
$sum_3 := \phi(sum_1, sum_2)$
$x_{in} := sum_3$ $y_{in} := i_3$
$sum_2 := x_{out}$
$i_2 := y_{out}$
Entry A
$x_0 := x_{in}$
$y_0 := y_{in}$
call Add
call inc
$y_{out} := y_2$
$x_{out} := x_1$
$a_{in} := x_0$
$b_{in} := y_0$
$y_1 := b_{out}$
$x_1 := a_{out}$
$z_{in} := y_1$
$y_2 := z_{out}$
Entry Add
$a_0 := a_{in}$
$b_0 := b_{in}$
$a_1 := a_0 + b_0$
$a_{out} := a_1$
$b_{out} := b_0$
Entry inc
$z_0 := z_{in}$
$one_0 := 1$
call Add
$z_{out} := z_1$
$a_{in} := z_0$
$b_{in} := one_0$
$one_1 := b_{out}$
$z_1 := a_{out}$

Interprozedurales Slicing (Traversierung)

Backward-Slicing (Prozedur P, Statement S):
Phase 1: Folge rückwärts **Parameter-In-Kanten**, Control- und Flow-Kanten (inklusive Summary-Kanten) ausgehend von S.

- Identifiziert Knoten in Prozeduren, die P (transitiv) aufrufen und von denen S abhängt.
- Da Parameter-Out-Edges ausgenommen sind, werden nur aufrufende Prozeduren besucht.
- Die Effekte nicht-besuchter Prozeduren werden aber nicht ignoriert, da Summary-Edges traversiert werden.


Entry Main
$sum_0 := initialState(sum)$
$i_0 := initialState(i)$
$sum_1 := 0$
while $i_3 < 11$
$i_1 := 1$
call A
$FinalUse(sum_3)$
$FinalUse(i_3)$
$i_3 := \phi(i_1, i_2)$
$sum_3 := \phi(sum_1, sum_2)$
$x_{in} := sum_3$ $y_{in} := i_3$
$sum_2 := x_{out}$
$i_2 := y_{out}$
Entry A
$x_0 := x_{in}$
$y_0 := y_{in}$
call Add
call inc
$y_{out} := y_2$
$x_{out} := x_1$
$a_{in} := x_0$
$b_{in} := y_0$
$y_1 := b_{out}$
$x_1 := a_{out}$
$z_{in} := y_1$
$y_2 := z_{out}$
Entry Add
$a_0 := a_{in}$
$b_0 := b_{in}$
$a_{out} := a_1$
$b_{out} := b_0$
$a_1 := a_0 + b_0$
Entry inc
$z_0 := z_{in}$
$one_0 := 1$
$z_{out} := z_1$
call Add
$one_1 := b_{out}$
$a_{in} := z_0$
$b_{in} := one_0$
$z_1 := a_{out}$

Interprozedurales Slicing (Traversierung)

Backward-Slicing (Prozedur P, Statement S):
Phase 2: Folge rückwärts **Parameter-Out-Kanten**, Control- und Flow-Kanten (inklusive Summary-Kanten) ausgehend von allen Knoten, die in Phase 1 identifiziert wurden.

- Identifiziert Knoten in Prozeduren, die von P (transitiv) aufgerufen werden und von denen S abhängt.
- Da Parameter-In-Edges ausgenommen sind, werden nur aufgerufene Prozeduren besucht.
- Es gilt wieder: Die Effekte nicht-besuchter Prozeduren werden nicht ignoriert, da Summary-Edges traversiert werden.