Optimierende Compiler

Rückwandlung aus der SSA-Form

Andreas Koch

FG Eingebettete Systeme und ihre Anwendungen
Informatik, TU Darmstadt

Sommersemester 2008

Überblick

- Interaktion zwischen SSA-Rückwandlung und Optimierung
  - Lost-Copy-Problem
  - Swap-Problem
  - Unnötige Kopieranweisungen
  - Probleme bei Platzierung der Kopieranweisungen
- Lösung: Algorithmus nach Briggs, Cooper, Harvey und Simpson
  - Paper liegt auf Web-Seite!
  - Eine Korrektur wird hier in VL besprochen

Überblick

- Interaktion zwischen SSA-Rückwandlung und Optimierung
  - Lost-Copy-Problem
  - Swap-Problem
  - Unnötige Kopieranweisungen
  - Probleme bei Platzierung der Kopieranweisungen
- Lösung: Algorithmus nach Briggs, Cooper, Harvey und Simpson
  - Paper liegt auf Web-Seite!
  - Eine Korrektur wird hier in VL besprochen

Überblick

- Interaktion zwischen SSA-Rückwandlung und Optimierung
  - Lost-Copy-Problem
  - Swap-Problem
  - Unnötige Kopieranweisungen
  - Probleme bei Platzierung der Kopieranweisungen
- Lösung: Algorithmus nach Briggs, Cooper, Harvey und Simpson
  - Paper liegt auf Web-Seite!
  - Eine Korrektur wird hier in VL besprochen

Überblick

- Interaktion zwischen SSA-Rückwandlung und Optimierung
  - Lost-Copy-Problem
  - Swap-Problem
  - Unnötige Kopieranweisungen
  - Probleme bei Platzierung der Kopieranweisungen
- Lösung: Algorithmus nach Briggs, Cooper, Harvey und Simpson
  - Paper liegt auf Web-Seite!
  - Eine Korrektur wird hier in VL besprochen

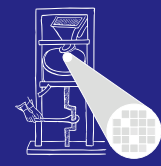
Überblick

- Interaktion zwischen SSA-Rückwandlung und Optimierung
  - Lost-Copy-Problem
  - Swap-Problem
  - Unnötige Kopieranweisungen
  - Probleme bei Platzierung der Kopieranweisungen
- Lösung: Algorithmus nach Briggs, Cooper, Harvey und Simpson
  - Paper liegt auf Web-Seite!
  - Eine Korrektur wird hier in VL besprochen

Überblick

- Interaktion zwischen SSA-Rückwandlung und Optimierung
  - Lost-Copy-Problem
  - Swap-Problem
  - Unnötige Kopieranweisungen
  - Probleme bei Platzierung der Kopieranweisungen
- Lösung: Algorithmus nach Briggs, Cooper, Harvey und Simpson
  - Paper liegt auf Web-Seite!
  - Eine Korrektur wird hier in VL besprochen

Überblick



- Interaktion zwischen SSA-Rückwandlung und Optimierung
  - Lost-Copy-Problem
  - Swap-Problem
  - Unnötige Kopieranweisungen
  - Probleme bei Platzierung der Kopieranweisungen
- Lösung: Algorithmus nach Briggs, Cooper, Harvey und Simpson
  - Paper liegt auf Web-Seite!
  - Eine Korrektur wird hier in VL besprochen

Überblick

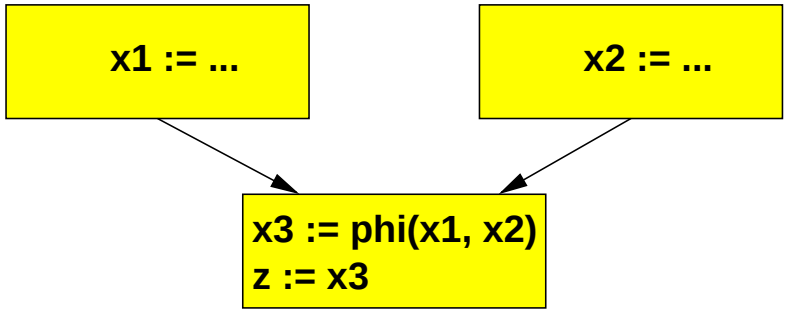
- Interaktion zwischen SSA-Rückwandlung und Optimierung
  - Lost-Copy-Problem
  - Swap-Problem
  - Unnötige Kopieranweisungen
  - Probleme bei Platzierung der Kopieranweisungen
- Lösung: Algorithmus nach Briggs, Cooper, Harvey und Simpson
  - Paper liegt auf Web-Seite!
  - Eine Korrektur wird hier in VL besprochen

Grundlagen

Entfernen von Phi-Knoten
Ersetzen durch Kopieranweisungen

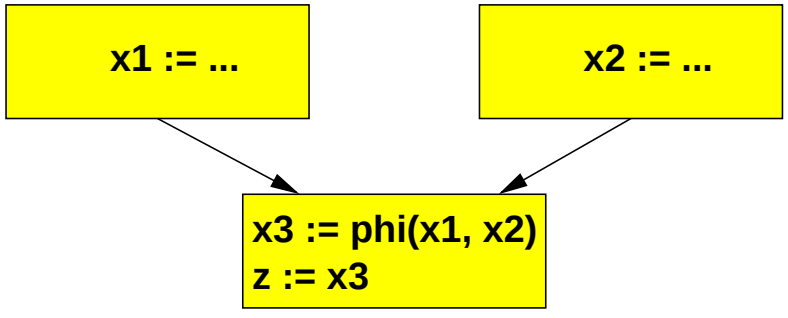
Vorher

Nachher

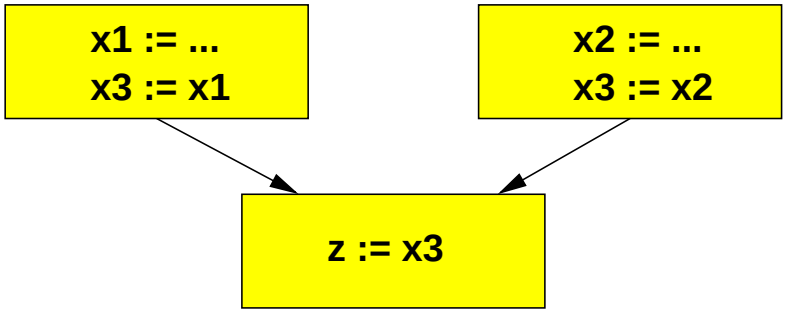


Entfernen von Phi-Knoten
Ersetzen durch Kopieranweisungen

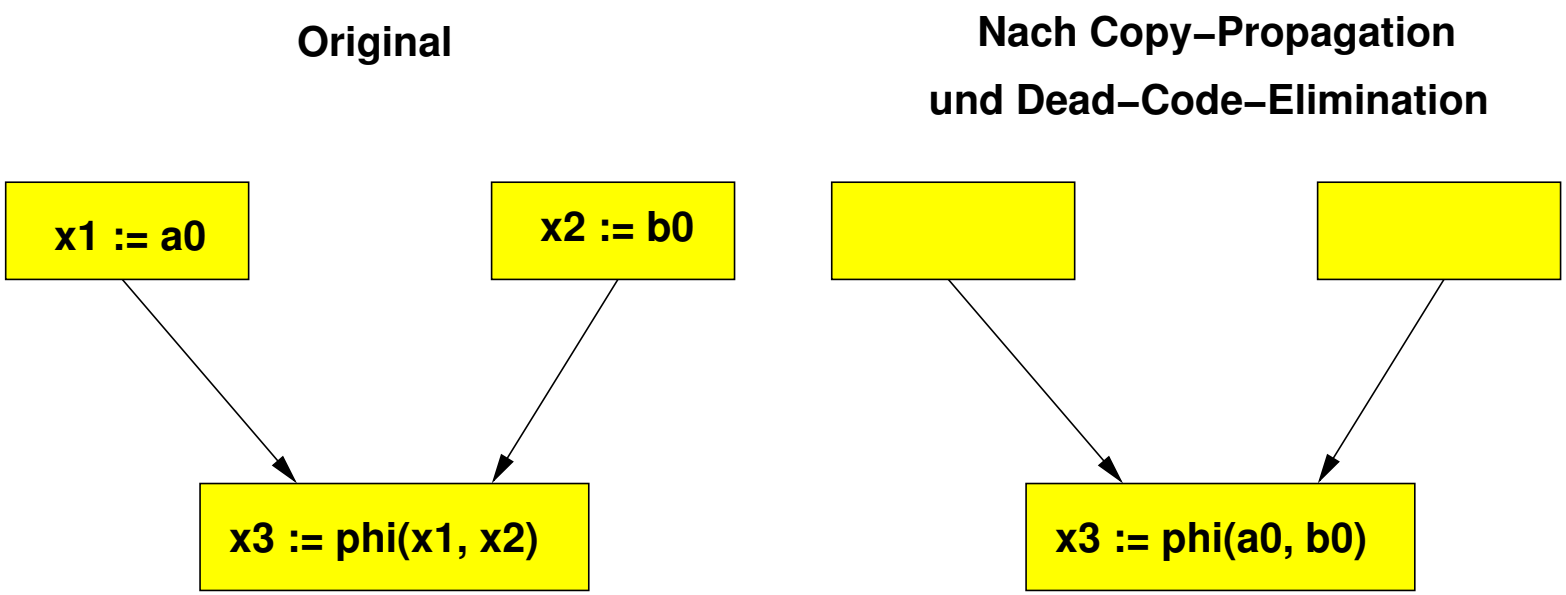
Vorher



Nachher



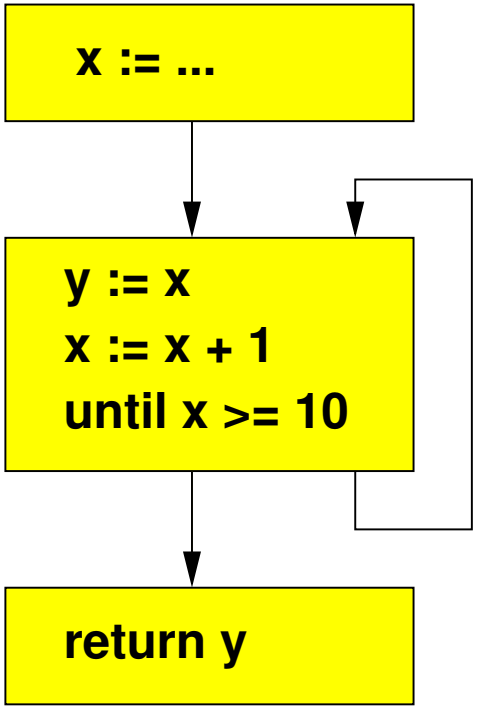
Optimierung durch Copy Propagation



Das “Lost-Copy”-Problem

Einfaches Beispiel: CFG zu SSA

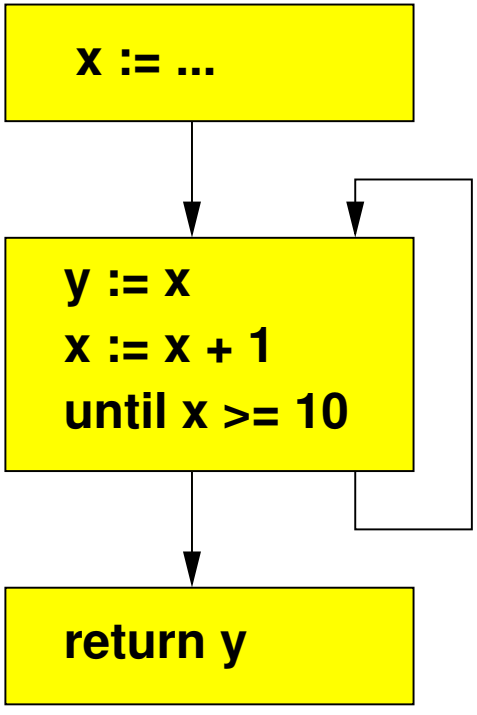
Normale Form



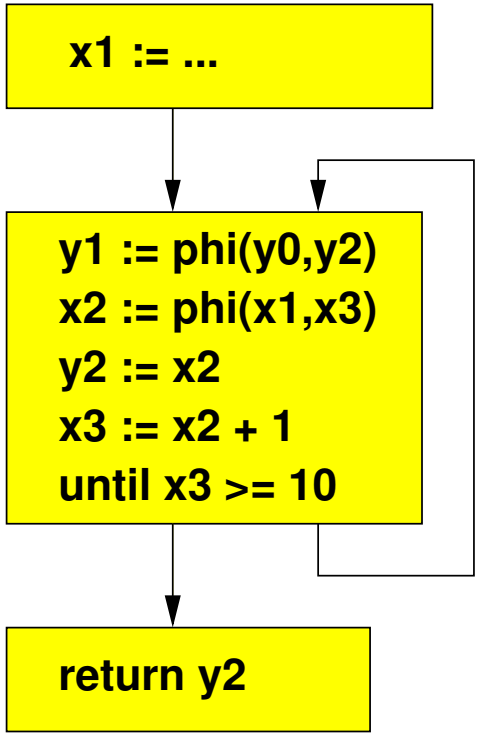
- Hierbei keine Überraschungen

Einfaches Beispiel: CFG zu SSA

Normale Form



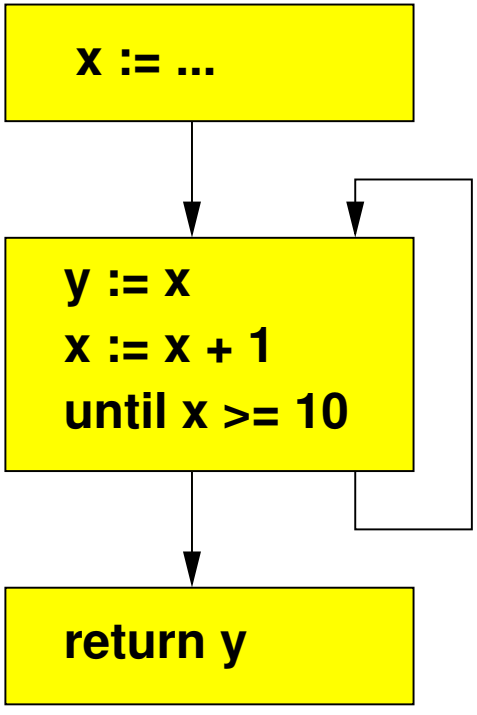
SSA–Form



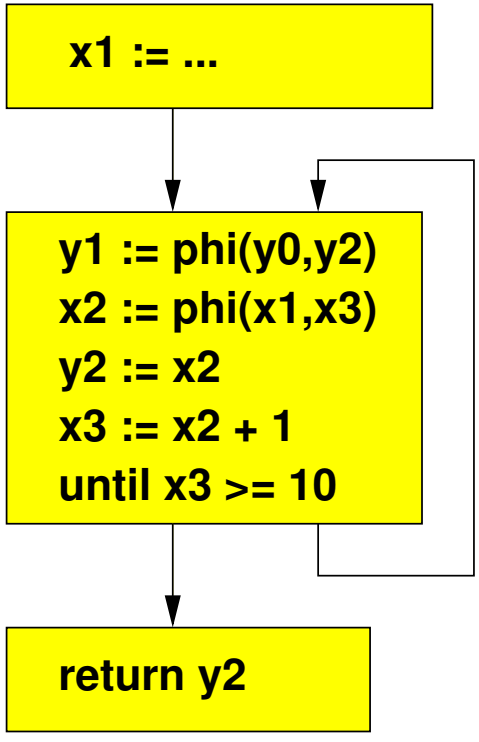
- Hierbei keine Überraschungen

Einfaches Beispiel: CFG zu SSA

Normale Form



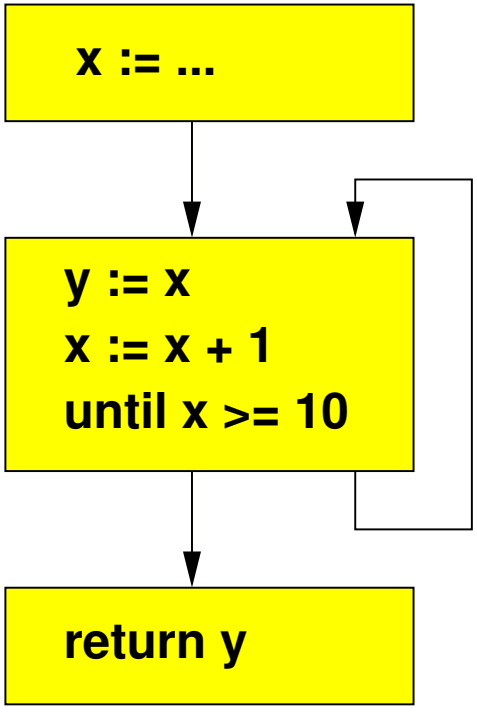
SSA–Form



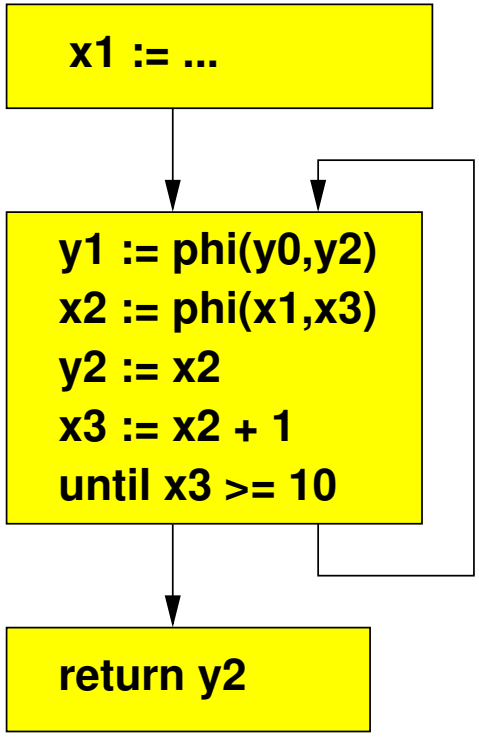
- Hierbei keine Überraschungen

Einfaches Beispiel: CFG zu SSA

Normale Form



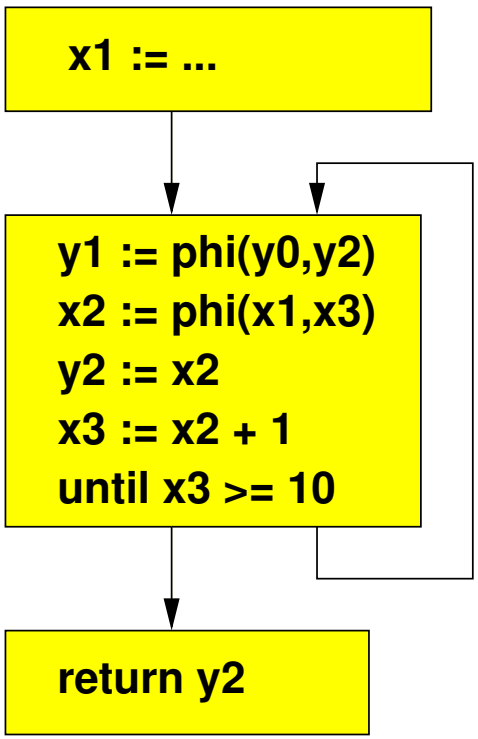
SSA–Form



- Hierbei keine Überraschungen

Einfaches Beispiel: SSA zu CFG

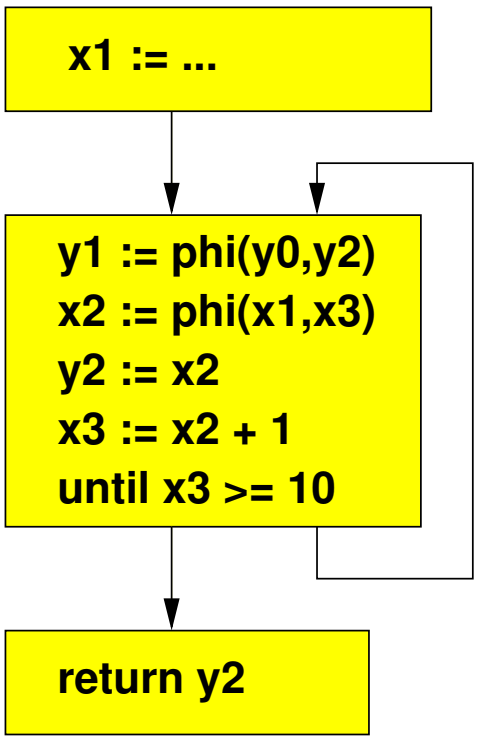
SSA–Form



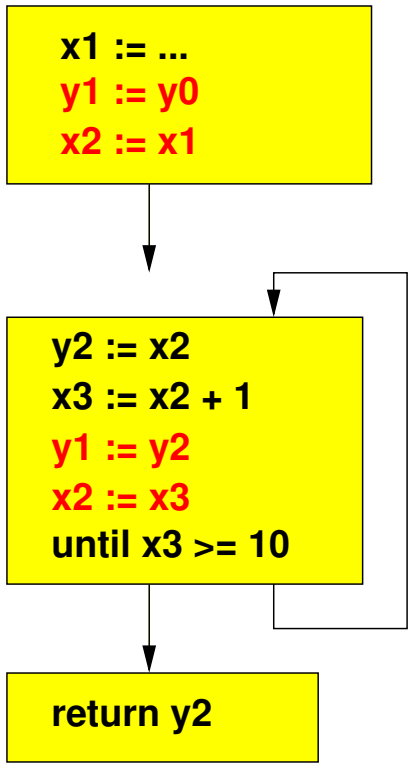
- Immer noch keine Überraschungen

Einfaches Beispiel: SSA zu CFG

SSA–Form



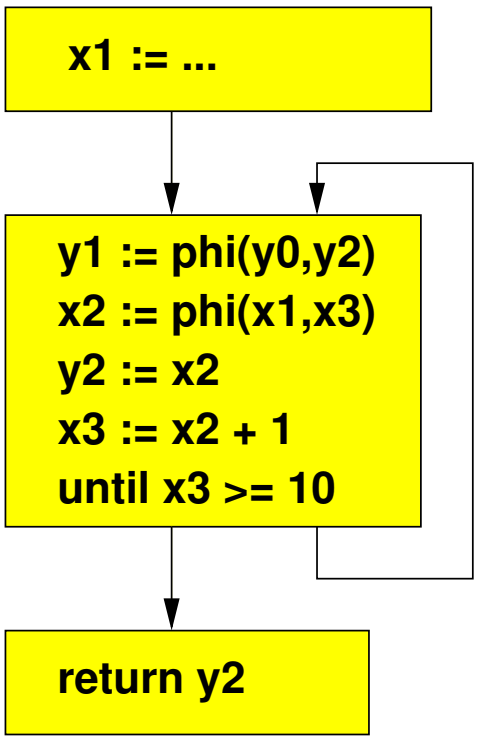
Rücktransformation aus SSA–Form



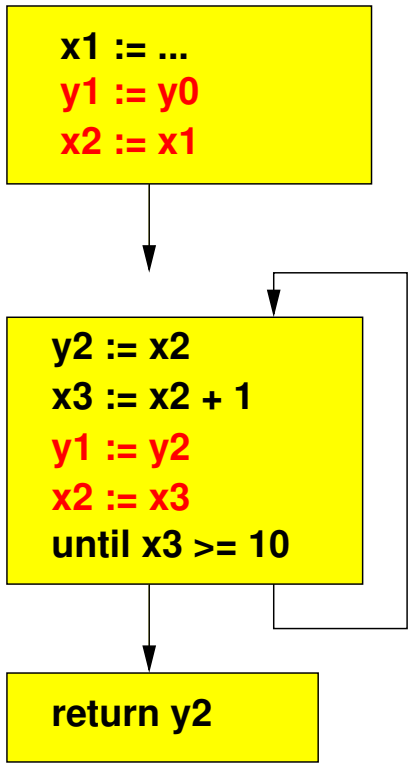
- Immer noch keine Überraschungen

Einfaches Beispiel: SSA zu CFG

SSA–Form



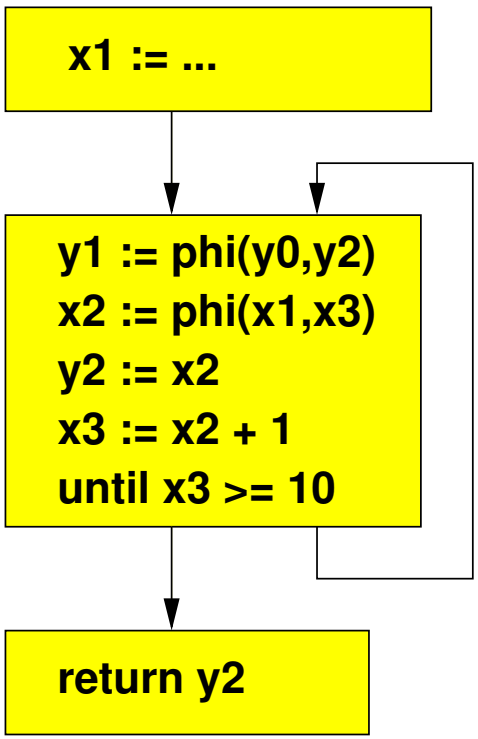
Rücktransformation aus SSA–Form



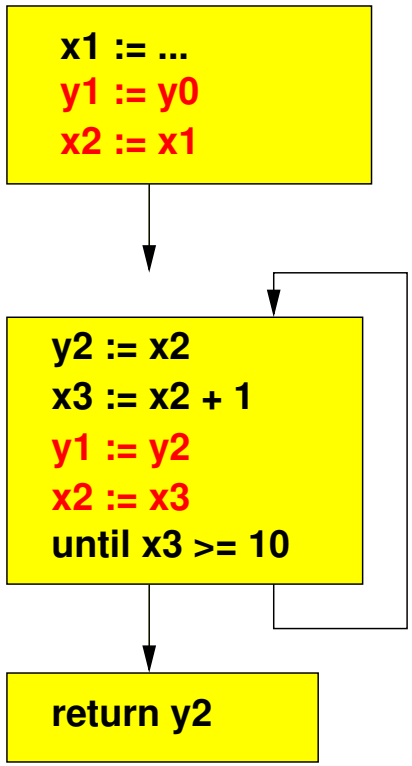
- Immer noch keine Überraschungen

Einfaches Beispiel: SSA zu CFG

SSA–Form



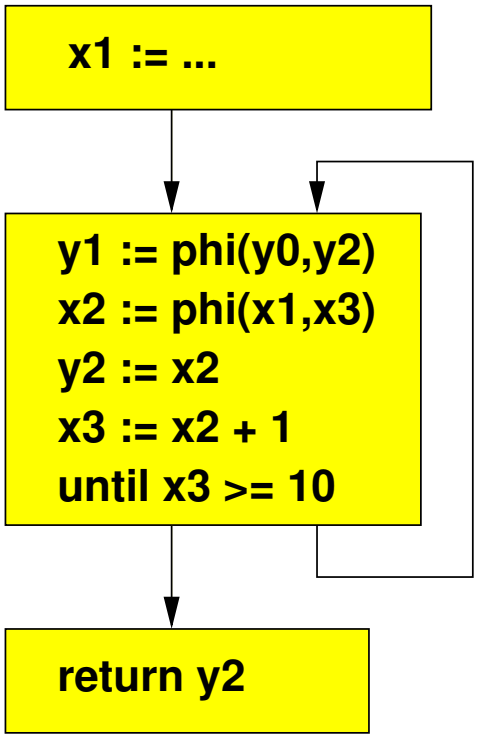
Rücktransformation aus SSA–Form



- Immer noch keine Überraschungen

Optimierung mit Copy Propagation

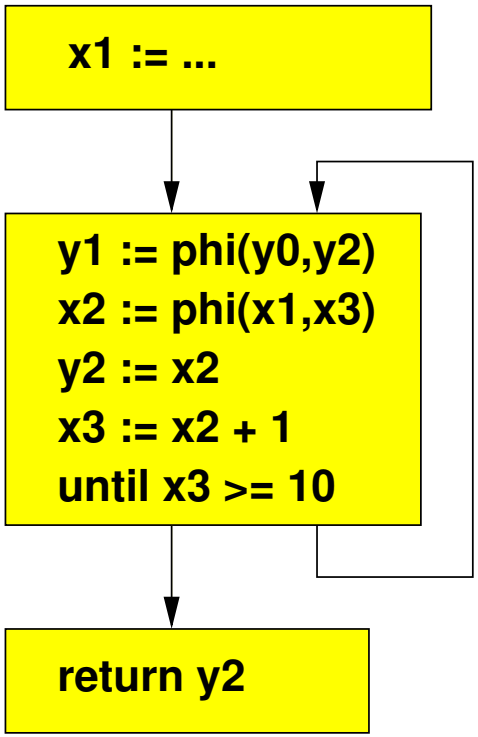
SSA–Form



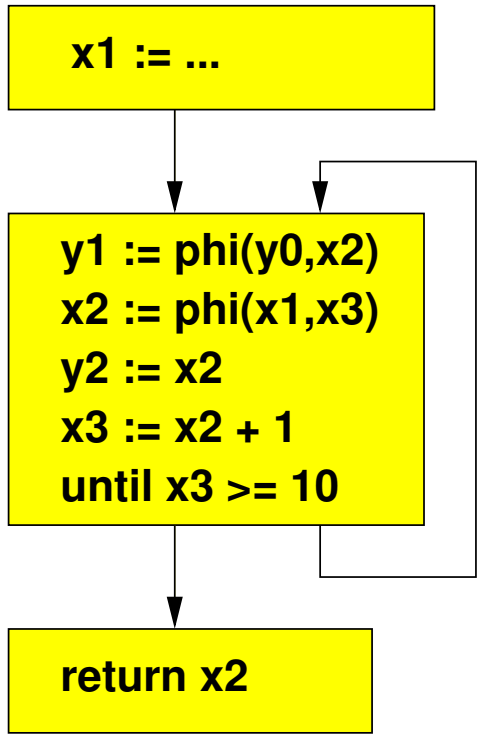
- $y2 := x2$ propagiert

Optimierung mit Copy Propagation

SSA–Form



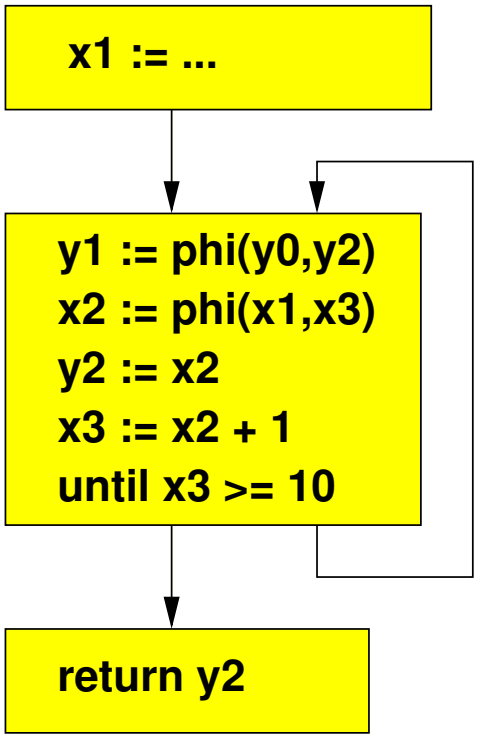
SSA–Form nach Copy–Propagation



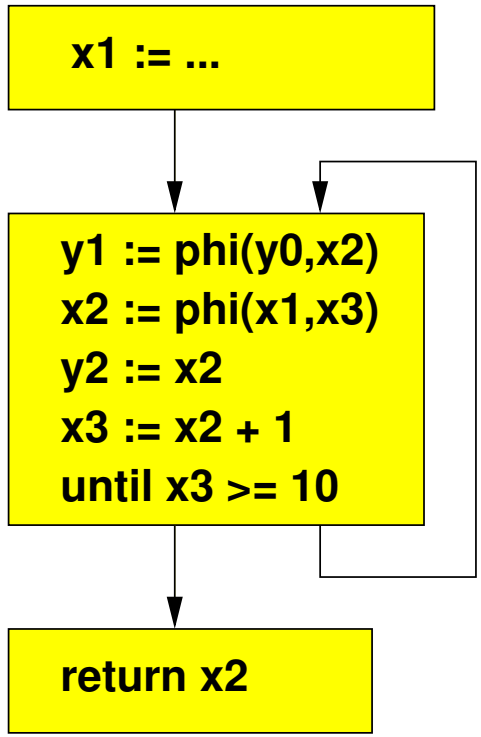
- y2 := x2 propagiert

Optimierung mit Copy Propagation

SSA–Form



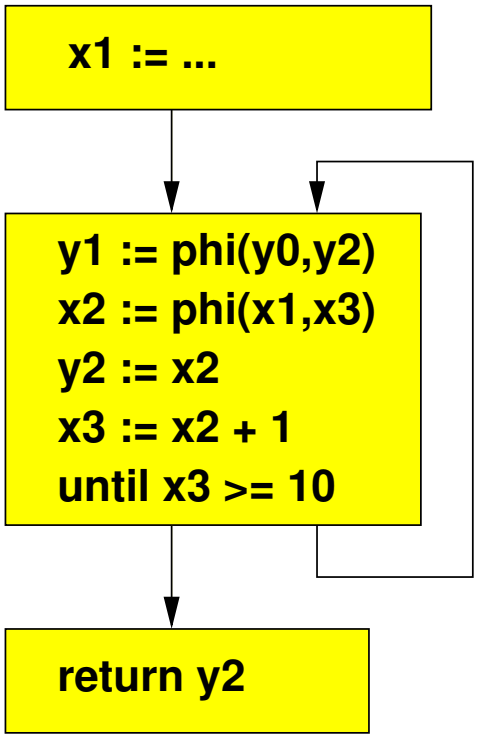
SSA–Form nach Copy–Propagation



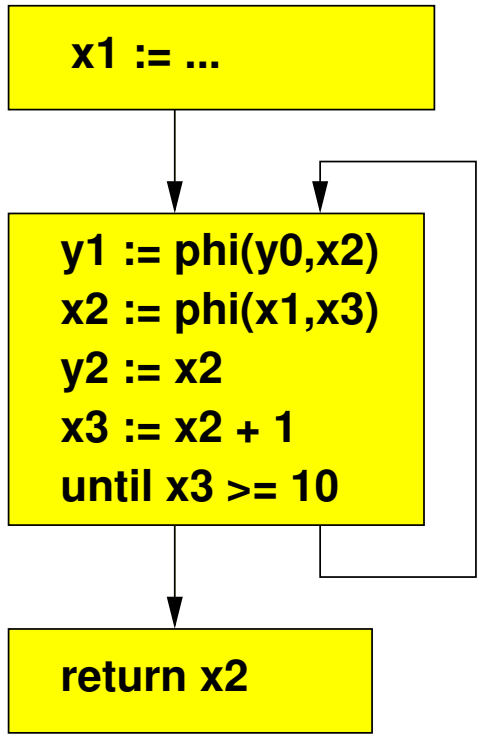
- y2 := x2 propagiert

Optimierung mit Copy Propagation

SSA–Form



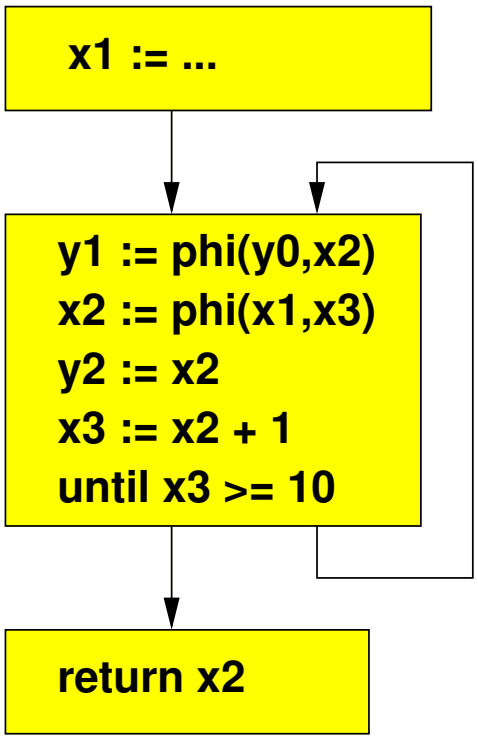
SSA–Form nach Copy–Propagation



- `y2:=x2` propagiert

Weitere Optimierung mit Dead Code Elim.

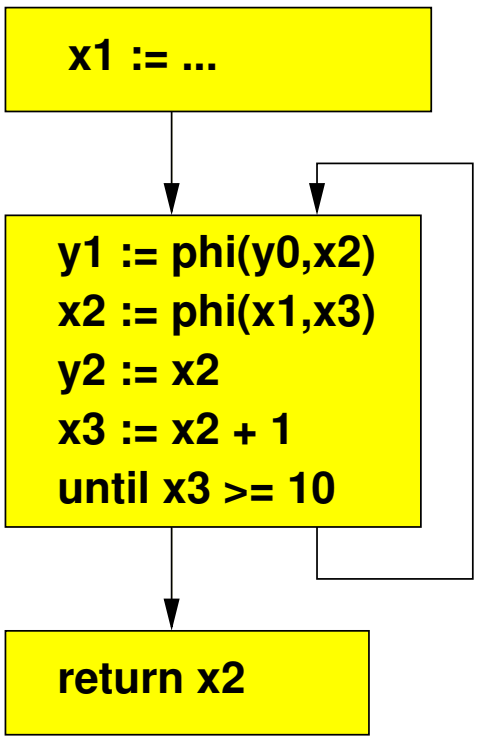
SSA–Form nach Copy–Propagation



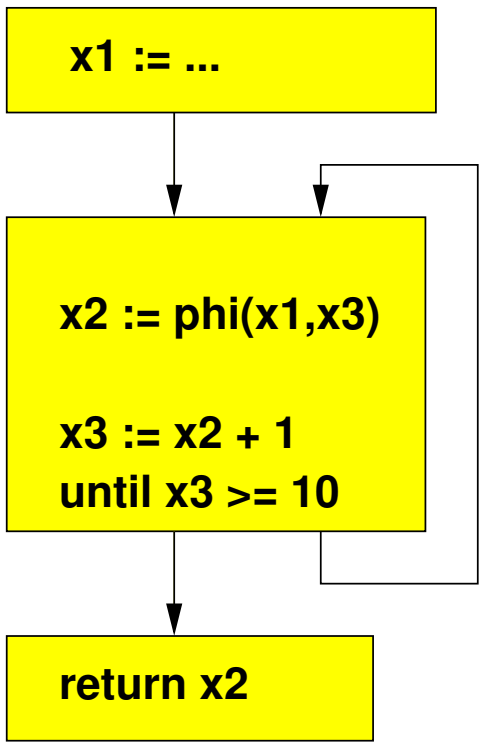
- Unnötige Kopieranweisung und Phi-Funktion für `y1` entfernt

Weitere Optimierung mit Dead Code Elim.

SSA–Form nach Copy–Propagation



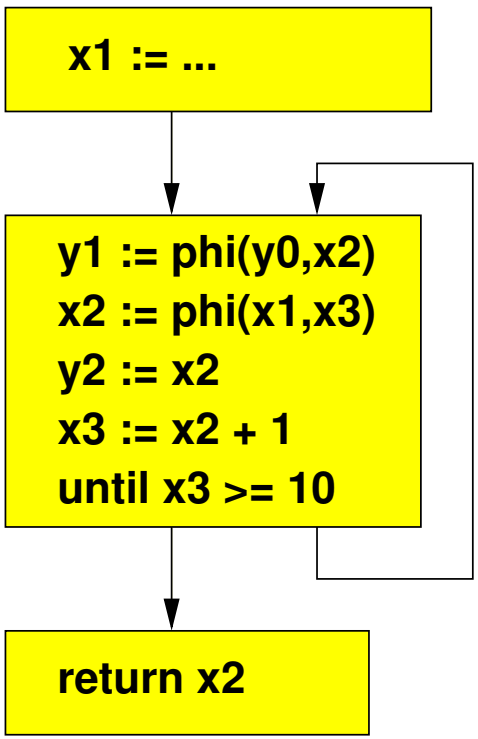
SSA–Form nach Copy–Propagation und Dead–Code Elim.



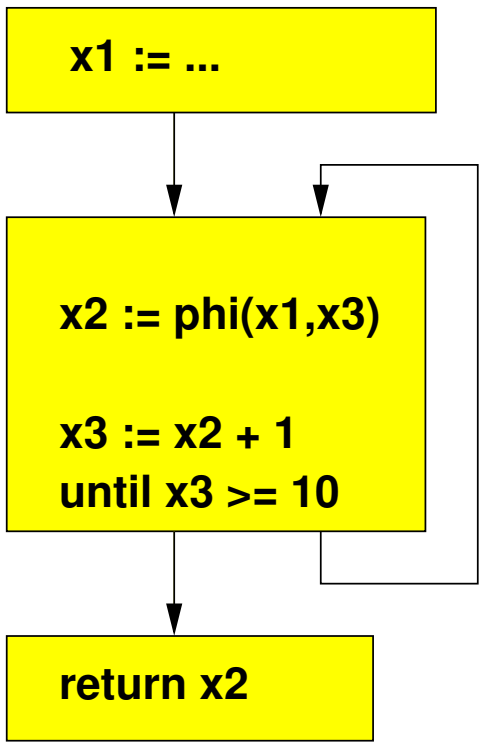
- Unnötige Kopieranweisung und Phi-Funktion für y1 entfernt

Weitere Optimierung mit Dead Code Elim.

SSA–Form nach Copy–Propagation



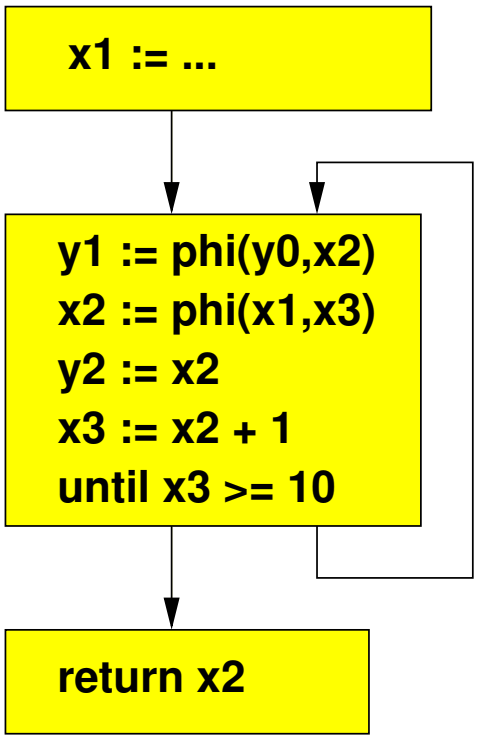
SSA–Form nach Copy–Propagation und Dead–Code Elim.



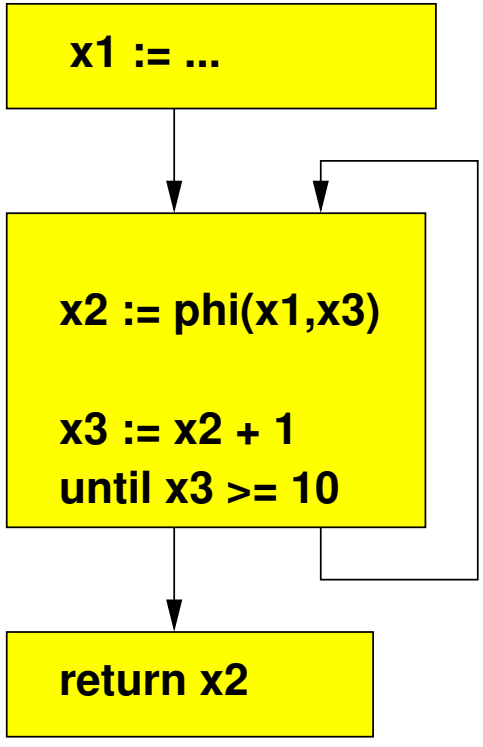
- Unnötige Kopieranweisung und Phi-Funktion für y1 entfernt

Weitere Optimierung mit Dead Code Elim.

SSA–Form nach Copy–Propagation

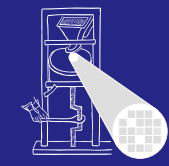


SSA–Form nach Copy–Propagation und Dead–Code Elim.

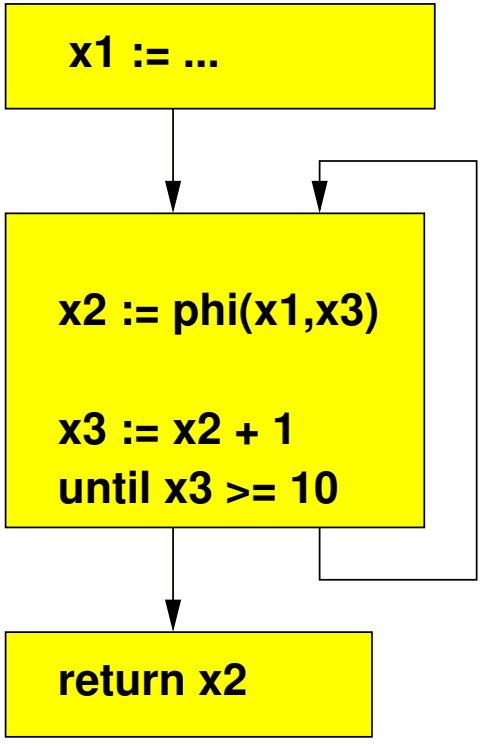


- Unnötige Kopieranweisung und Phi-Funktion für `y1` entfernt

Einfache SSA-Rückwandlung nach CFG



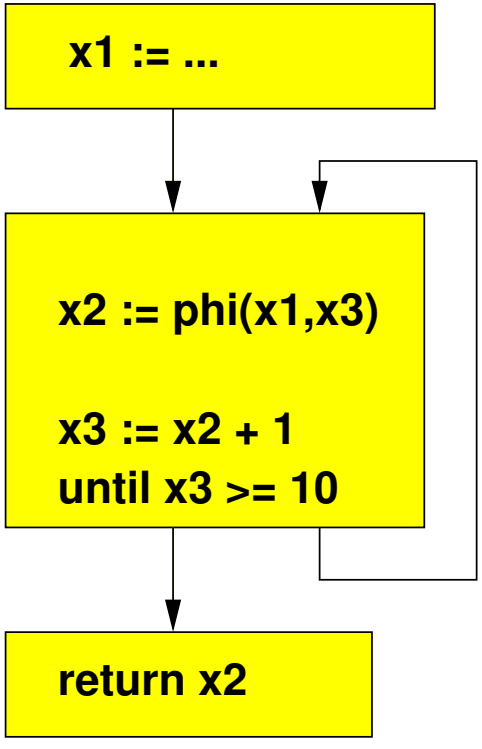
SSA–Form nach Copy–Propagation und Dead–Code Elim.



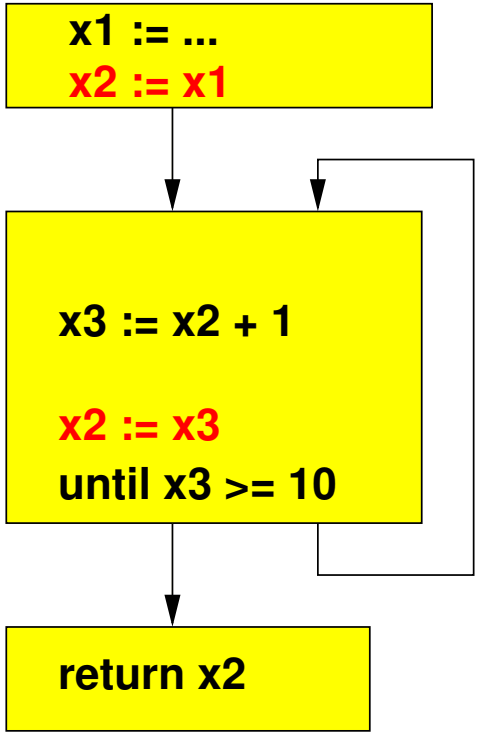
- Fehler: Falscher Wert zurückgegeben!

Einfache SSA-Rückwandlung nach CFG

SSA–Form nach Copy–Propagation und Dead–Code Elim.



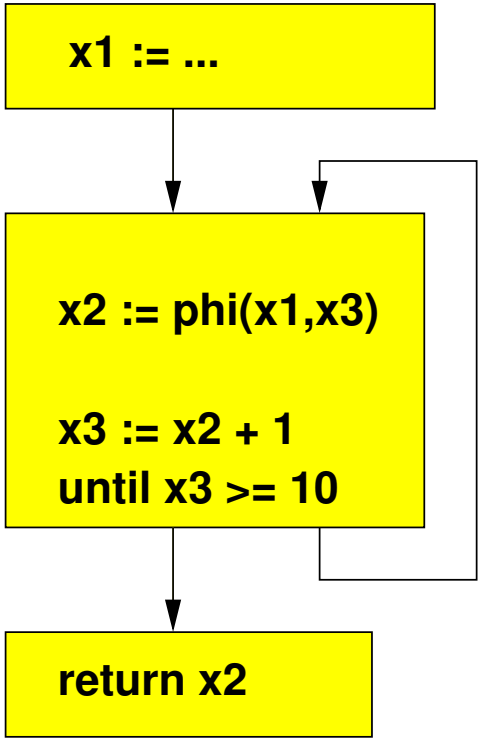
Rücktransformation aus SSA–Form nach Copy–Propagation und Dead–Code Elim.



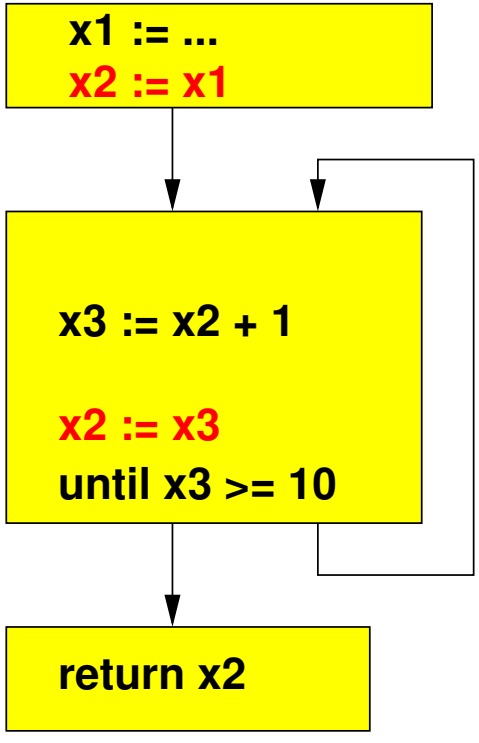
- Fehler: Falscher Wert zurückgegeben!

Einfache SSA-Rückwandlung nach CFG

SSA–Form nach Copy–Propagation und Dead–Code Elim.

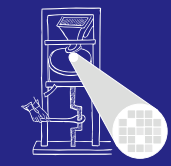


Rücktransformation aus SSA–Form nach Copy–Propagation und Dead–Code Elim.

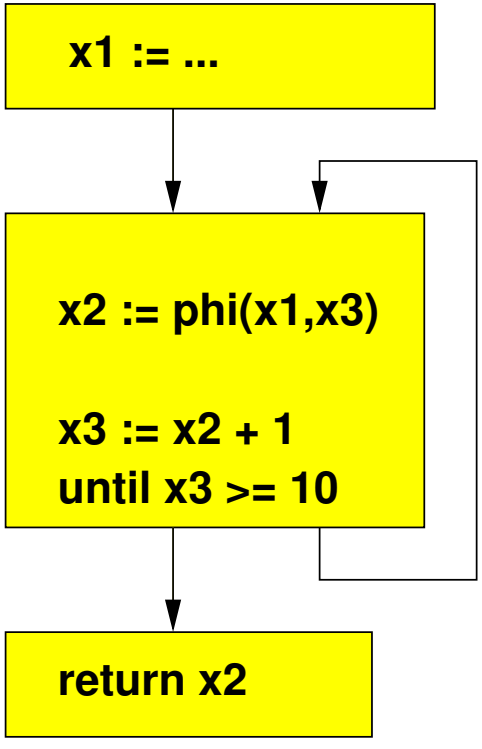


- Fehler: Falscher Wert zurückgegeben!

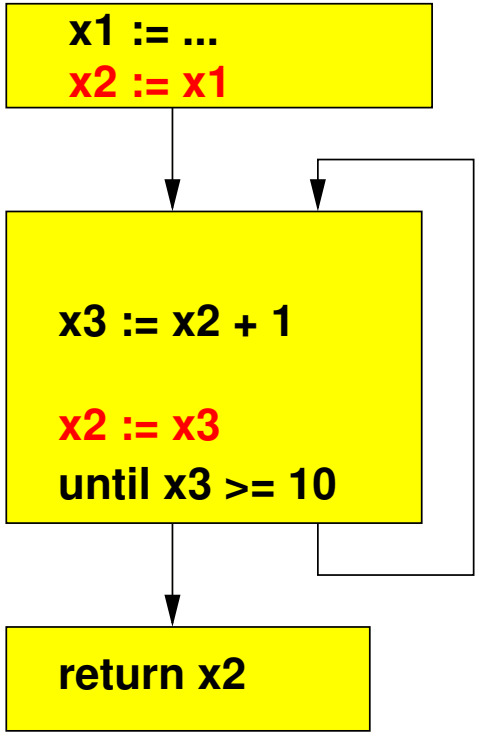
Einfache SSA-Rückwandlung nach CFG



SSA–Form nach Copy–Propagation und Dead–Code Elim.



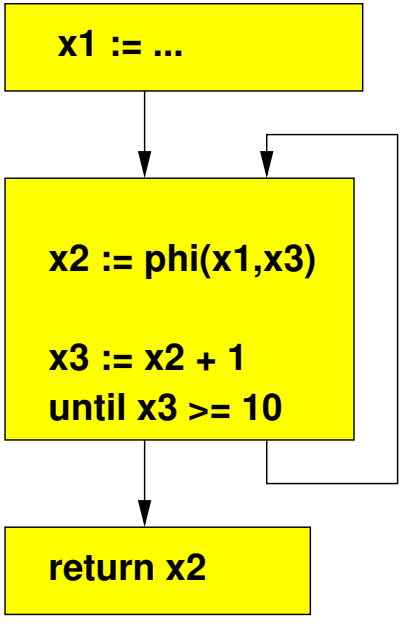
Rücktransformation aus SSA–Form nach Copy–Propagation und Dead–Code Elim.



- Fehler: Falscher Wert zurückgegeben!

Aufspalten Kritischer Kanten

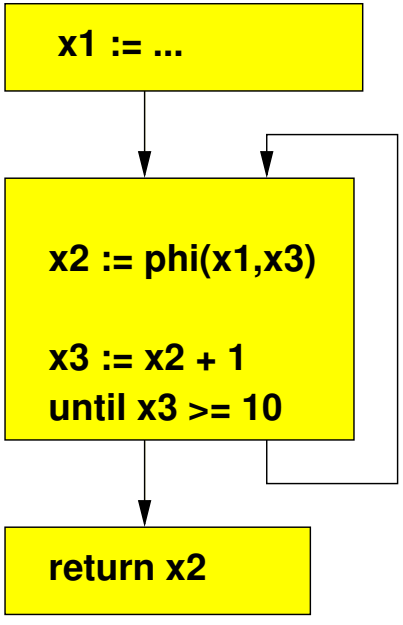
SSA–Form nach Copy–Propagation und Dead–Code Elim.



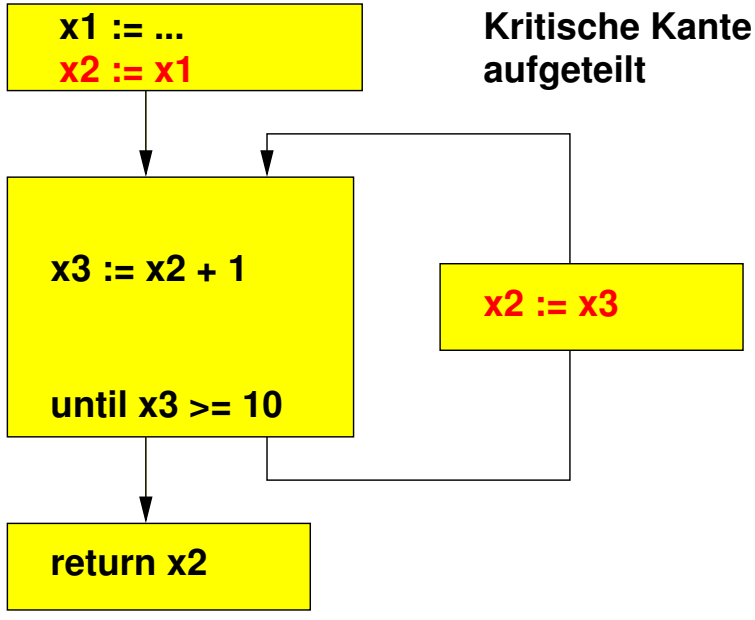
- Jetzt richtig
- Aufspalten kritischer Kanten nicht immer möglich oder wünschenswert
- Andere Lösung?

Aufspalten Kritischer Kanten

SSA–Form nach Copy–Propagation und Dead–Code Elim.



Rücktransformation aus SSA–Form nach Copy–Propagation und Dead–Code Elim.

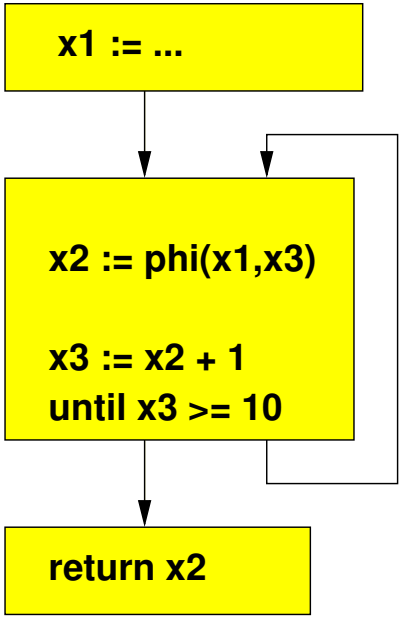


Kritische Kante aufgeteilt

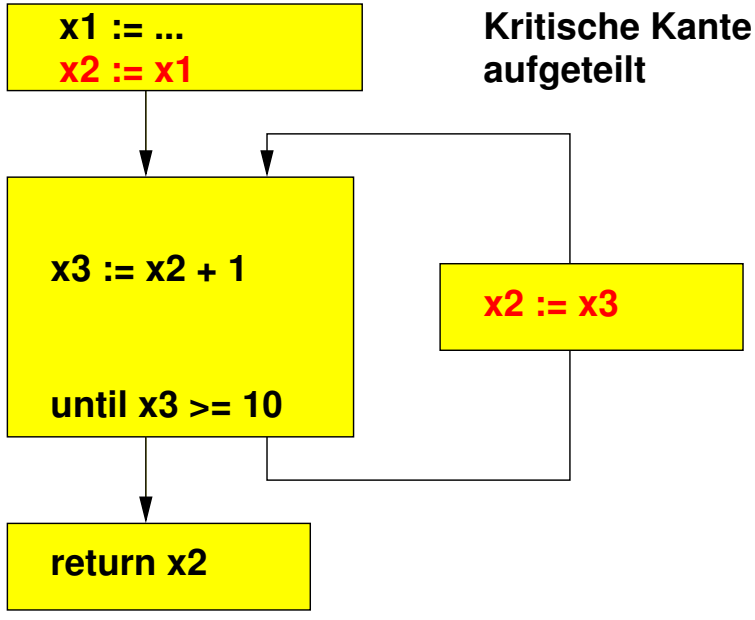
- Jetzt richtig
- Aufspalten kritischer Kanten nicht immer möglich oder wünschenswert
- Andere Lösung?

Aufspalten Kritischer Kanten

SSA–Form nach Copy–Propagation und Dead–Code Elim.



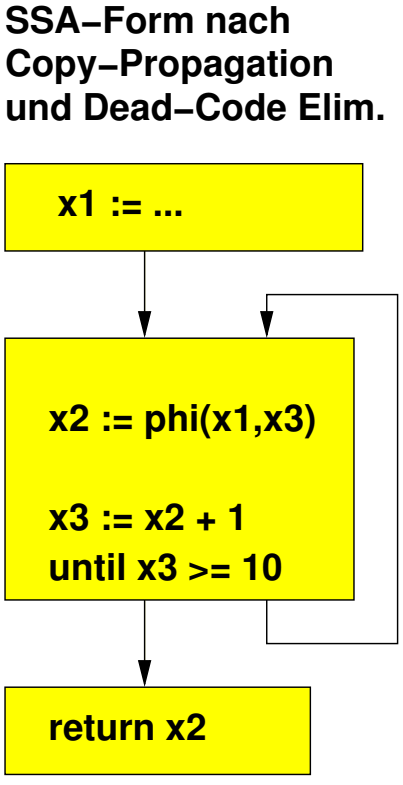
Rücktransformation aus SSA–Form nach Copy–Propagation und Dead–Code Elim.

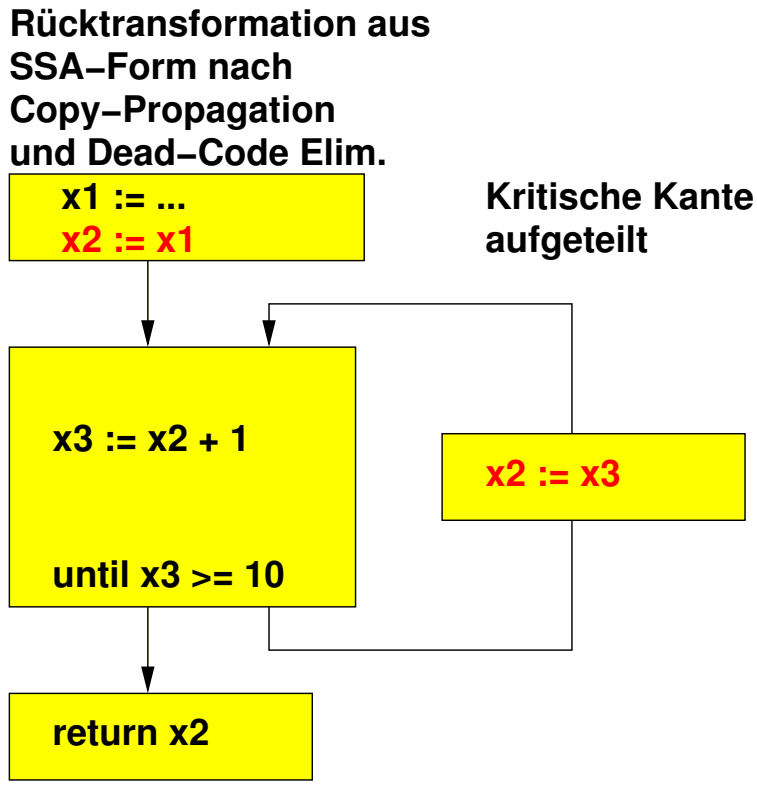


Kritische Kante aufgeteilt

- Jetzt richtig
- Aufspalten kritischer Kanten nicht immer möglich oder wünschenswert
- Andere Lösung?

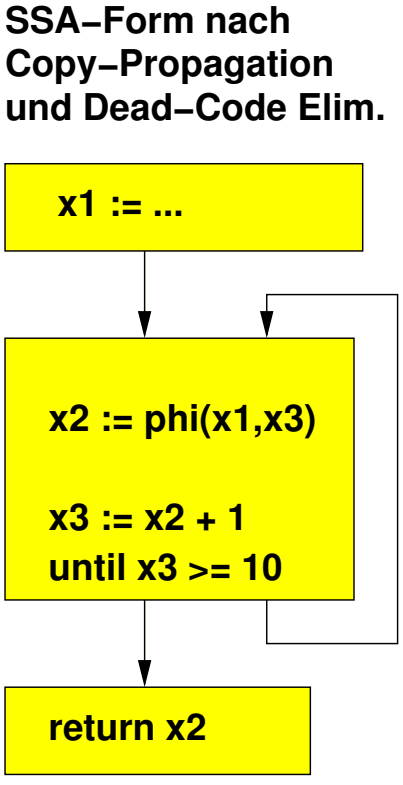
Aufspalten Kritischer Kanten

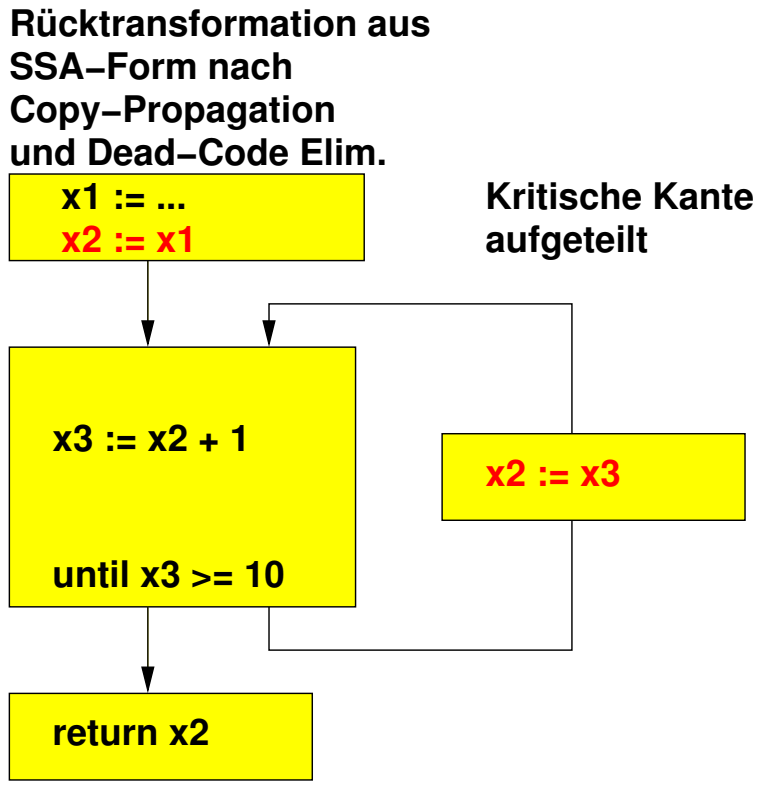




- Jetzt richtig
- Aufspalten kritischer Kanten nicht immer möglich oder wünschenswert
- Andere Lösung?

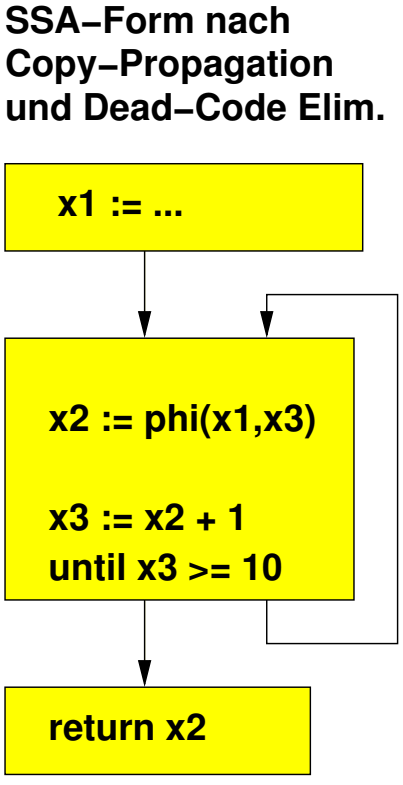
Aufspalten Kritischer Kanten

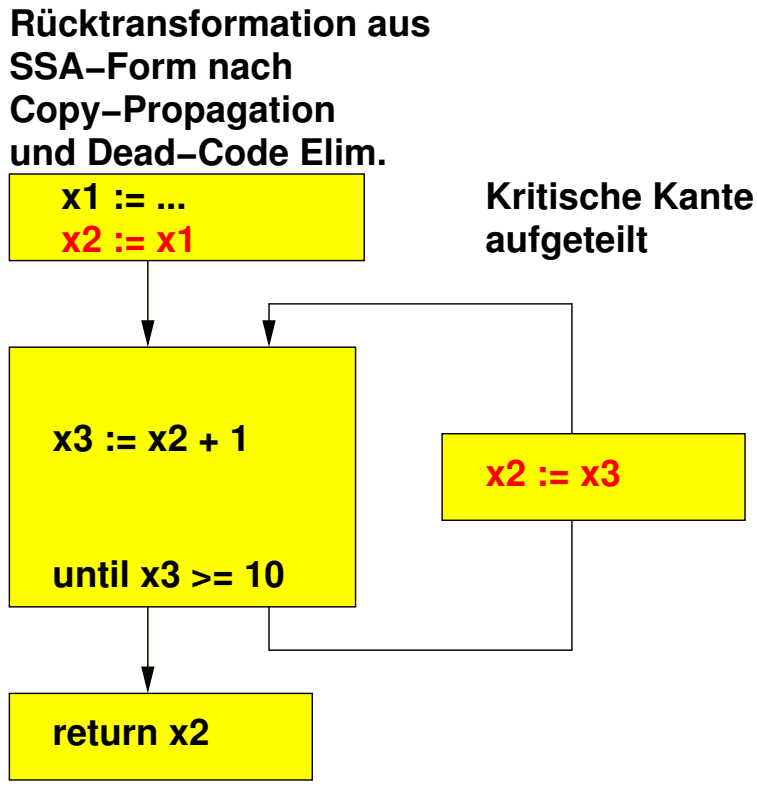




- Jetzt richtig
- Aufspalten kritischer Kanten nicht immer möglich oder wünschenswert
- Andere Lösung?

Aufspalten Kritischer Kanten

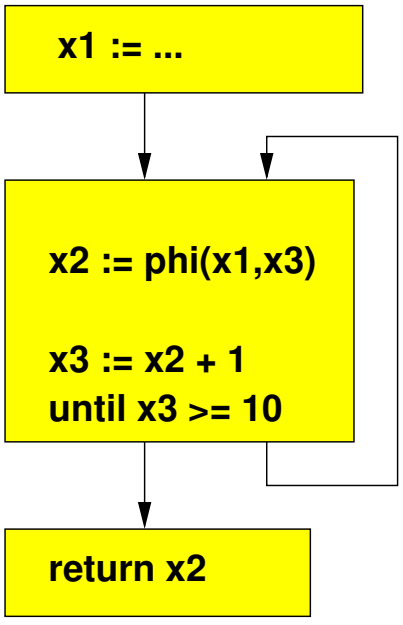




- Jetzt richtig
- Aufspalten kritischer Kanten nicht immer möglich oder wünschenswert
- Andere Lösung?

Lösung: Wert vor Überschreiben sichern

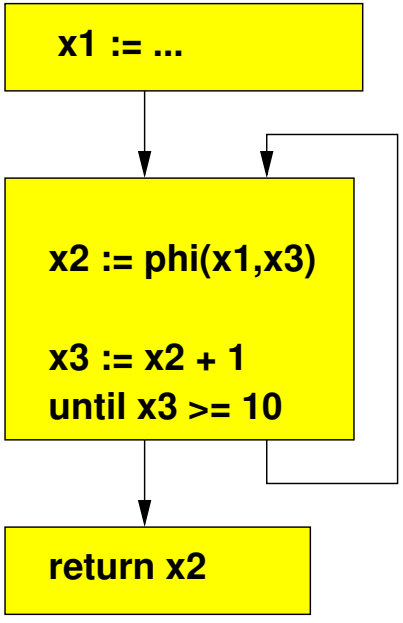
SSA–Form nach Copy–Propagation und Dead–Code Elim.



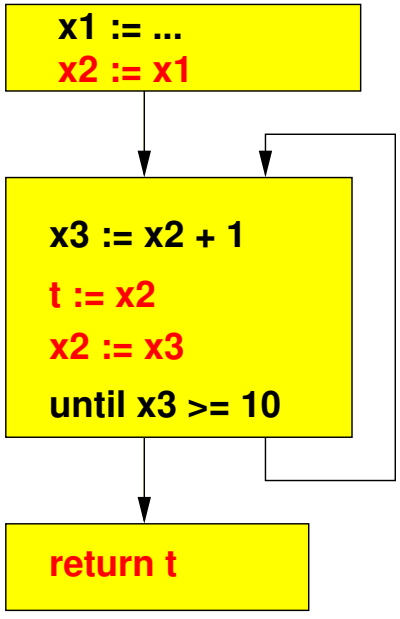
- Erkenne solche Fälle (→ Live Variables)
- Füge Sicherheitskopie ein
- Ersetze spätere Verwendungen durch Sicherheitskopie

Lösung: Wert vor Überschreiben sichern

SSA–Form nach Copy–Propagation und Dead–Code Elim.



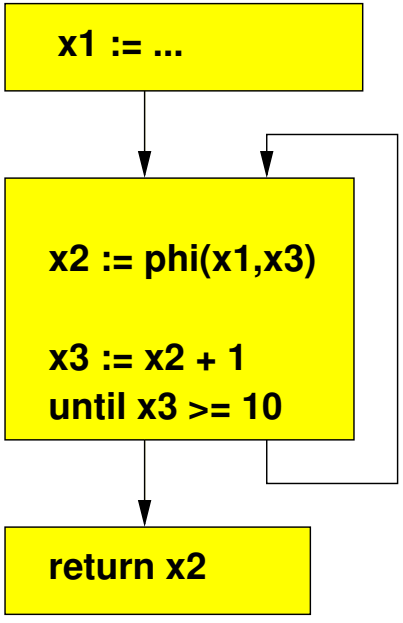
Rücktransformation aus SSA–Form nach Copy–Propagation und Dead–Code Elim.



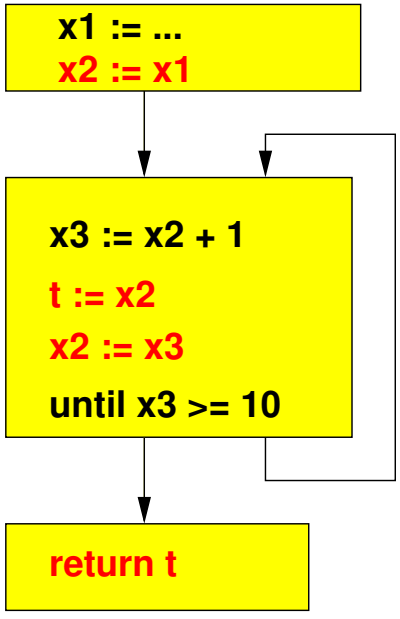
- Erkenne solche Fälle (→ Live Variables)
- Füge Sicherheitskopie ein
- Ersetze spätere Verwendungen durch Sicherheitskopie

Lösung: Wert vor Überschreiben sichern

SSA–Form nach Copy–Propagation und Dead–Code Elim.



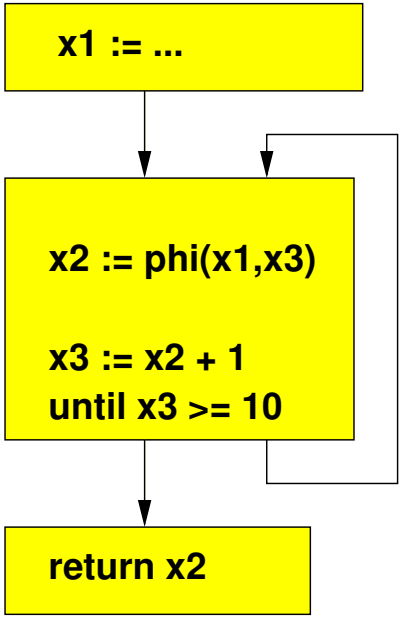
Rücktransformation aus SSA–Form nach Copy–Propagation und Dead–Code Elim.



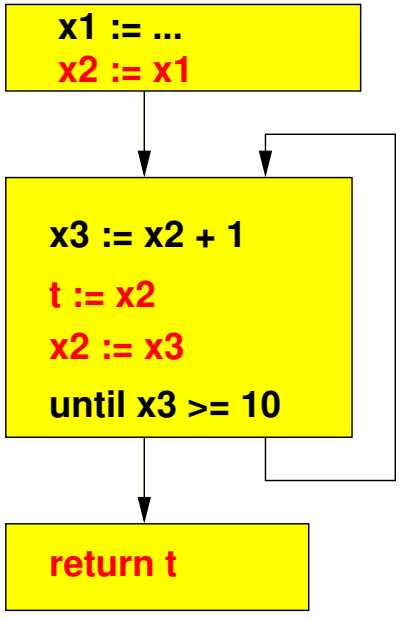
- Erkenne solche Fälle (→ Live Variables)
- Füge Sicherheitskopie ein
- Ersetze spätere Verwendungen durch Sicherheitskopie

Lösung: Wert vor Überschreiben sichern

SSA–Form nach Copy–Propagation und Dead–Code Elim.



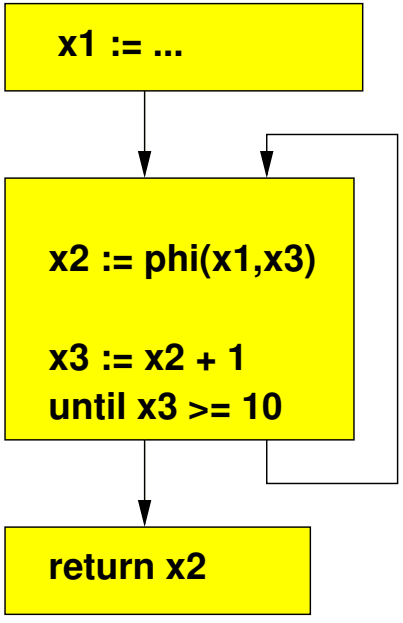
Rücktransformation aus SSA–Form nach Copy–Propagation und Dead–Code Elim.



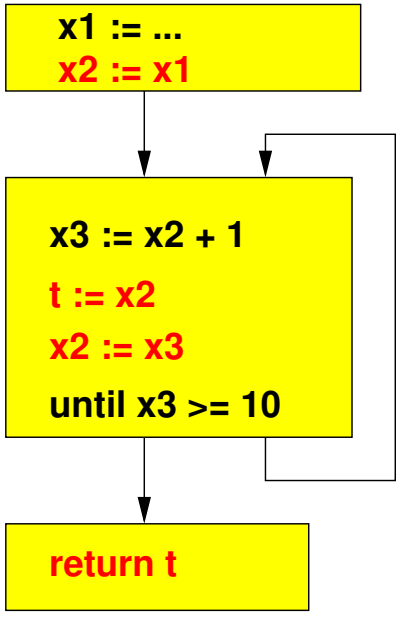
- Erkenne solche Fälle (→ Live Variables)
- Füge Sicherheitskopie ein
- Ersetze spätere Verwendungen durch Sicherheitskopie

Lösung: Wert vor Überschreiben sichern

SSA–Form nach Copy–Propagation und Dead–Code Elim.



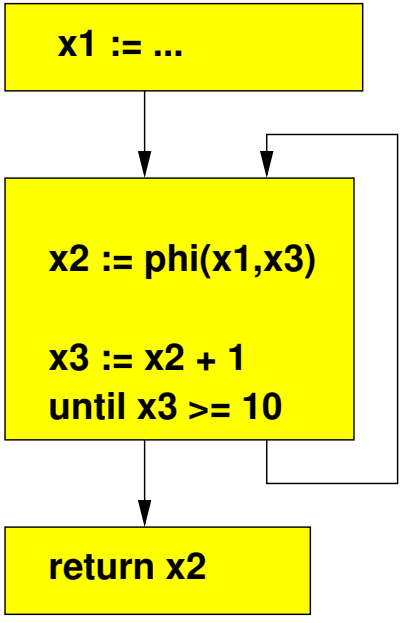
Rücktransformation aus SSA–Form nach Copy–Propagation und Dead–Code Elim.



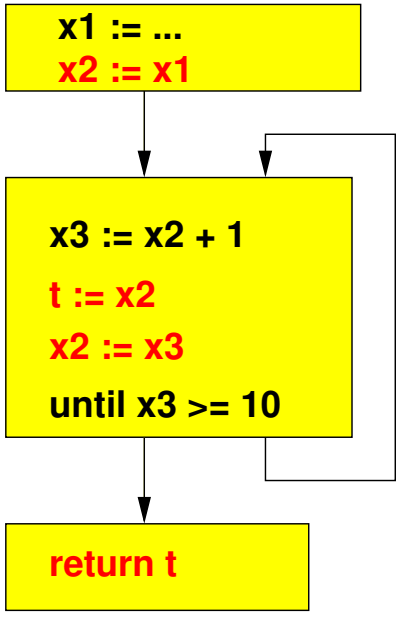
- Erkenne solche Fälle ($\rightarrow$ Live Variables)
- Füge Sicherheitskopie ein
- Ersetze spätere Verwendungen durch Sicherheitskopie

Lösung: Wert vor Überschreiben sichern

SSA–Form nach Copy–Propagation und Dead–Code Elim.

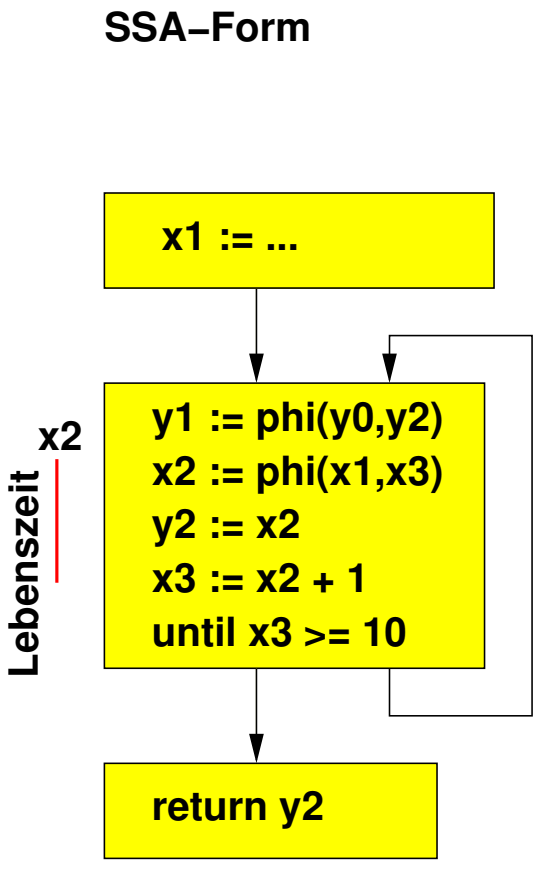


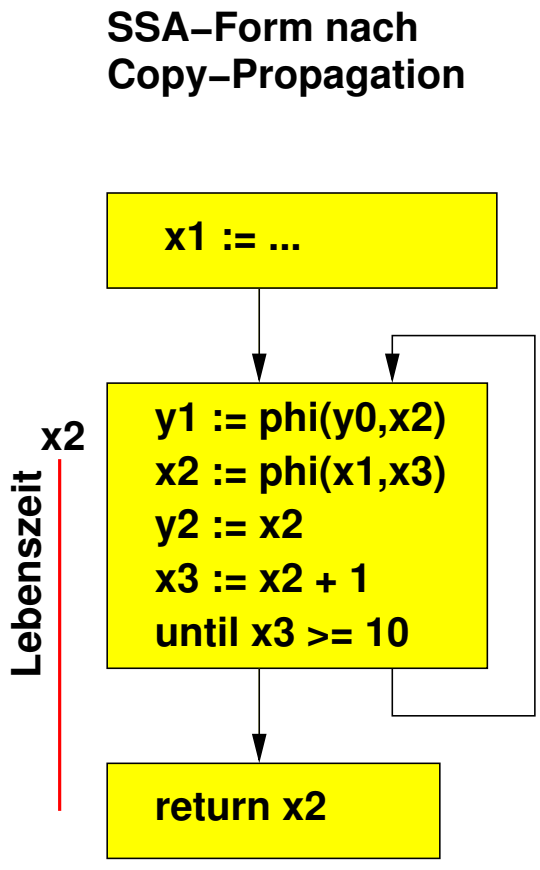
Rücktransformation aus SSA–Form nach Copy–Propagation und Dead–Code Elim.



- Erkenne solche Fälle ($\rightarrow$ Live Variables)
- Füge Sicherheitskopie ein
- Ersetze spätere Verwendungen durch Sicherheitskopie

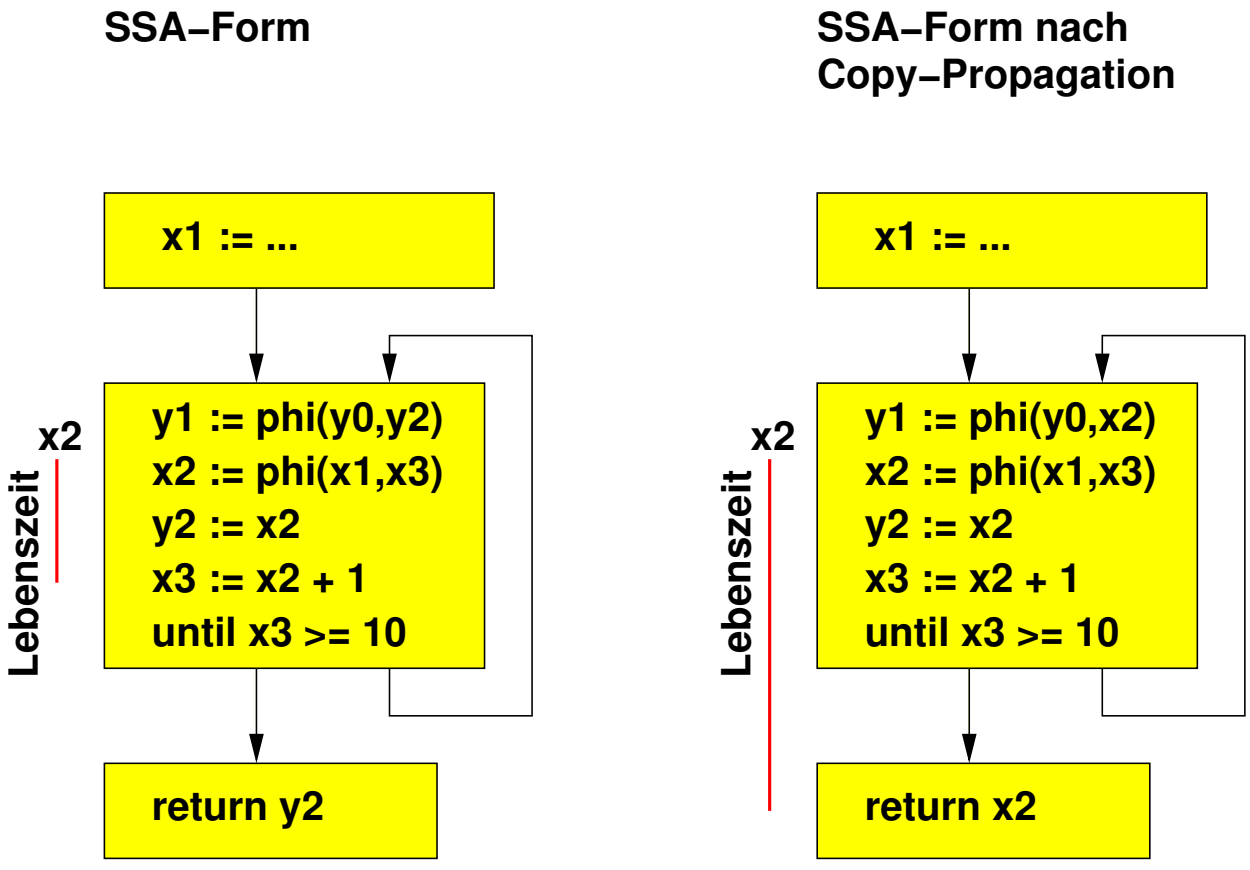
Allgemeines Lost-Copy-Problem 1





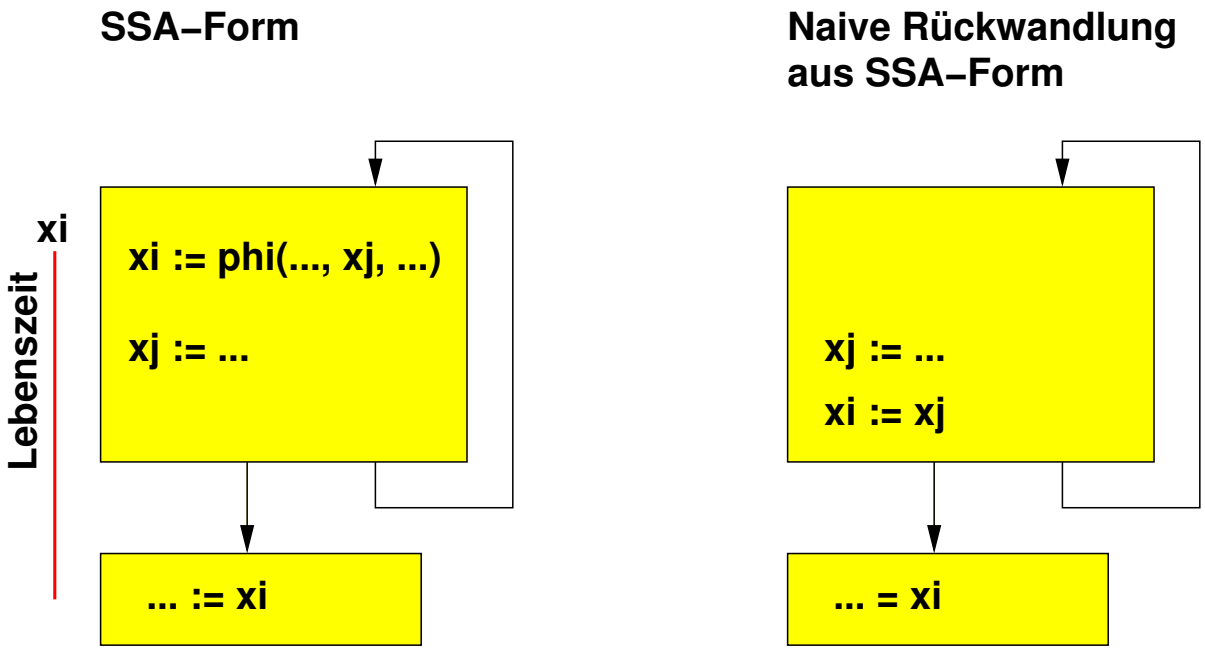
Lebenszeit von Variablen über bestimmende Phi-Funktion hinaus ausgedehnt

Allgemeines Lost-Copy-Problem 1



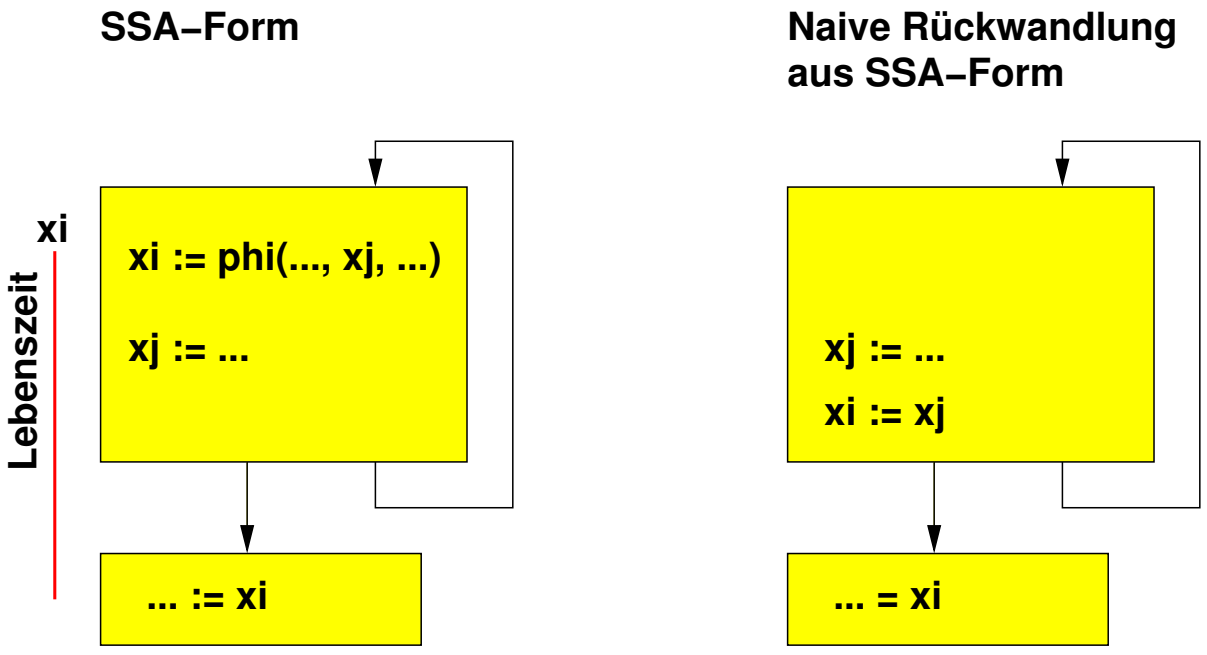
Lebenszeit von Variablen über bestimmende Phi-Funktion hinaus ausgedehnt

Allgemeines Lost Copy-Problem 2



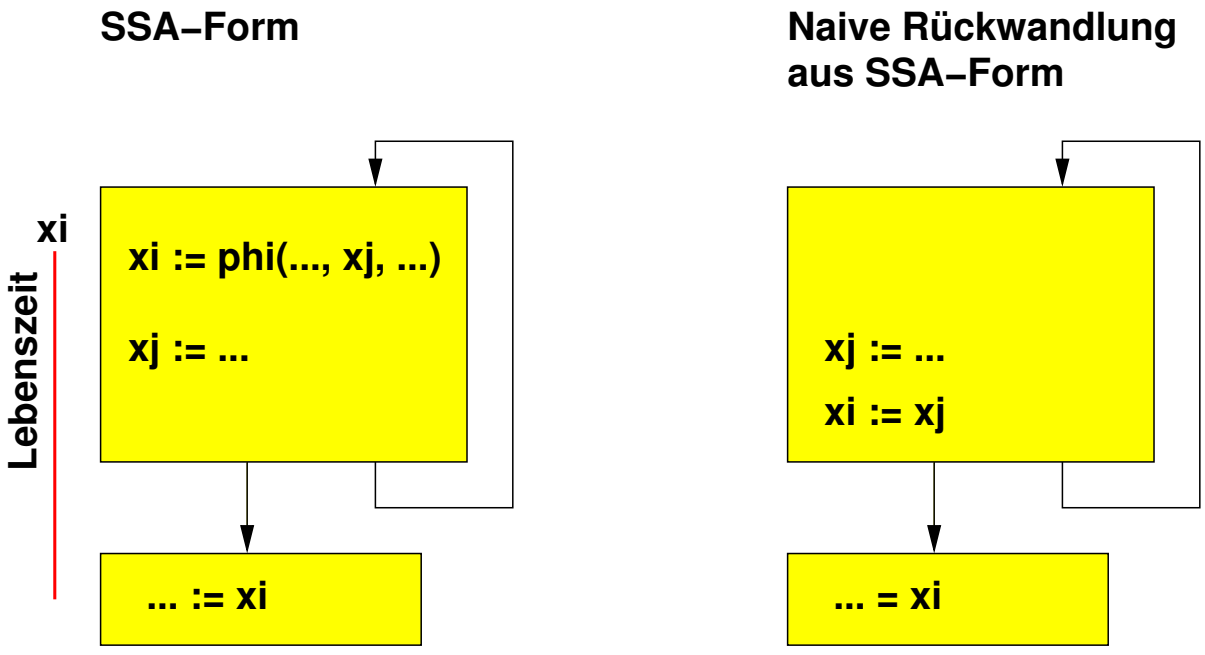
- Naive Umwandlung liefert fehlerhafte Ergebnisse
- Kopieranweisung auf `xi` mitten in dessen Lebenszeit hineingesetzt
- Liefert immer `xj` statt `xi`

Allgemeines Lost Copy-Problem 2



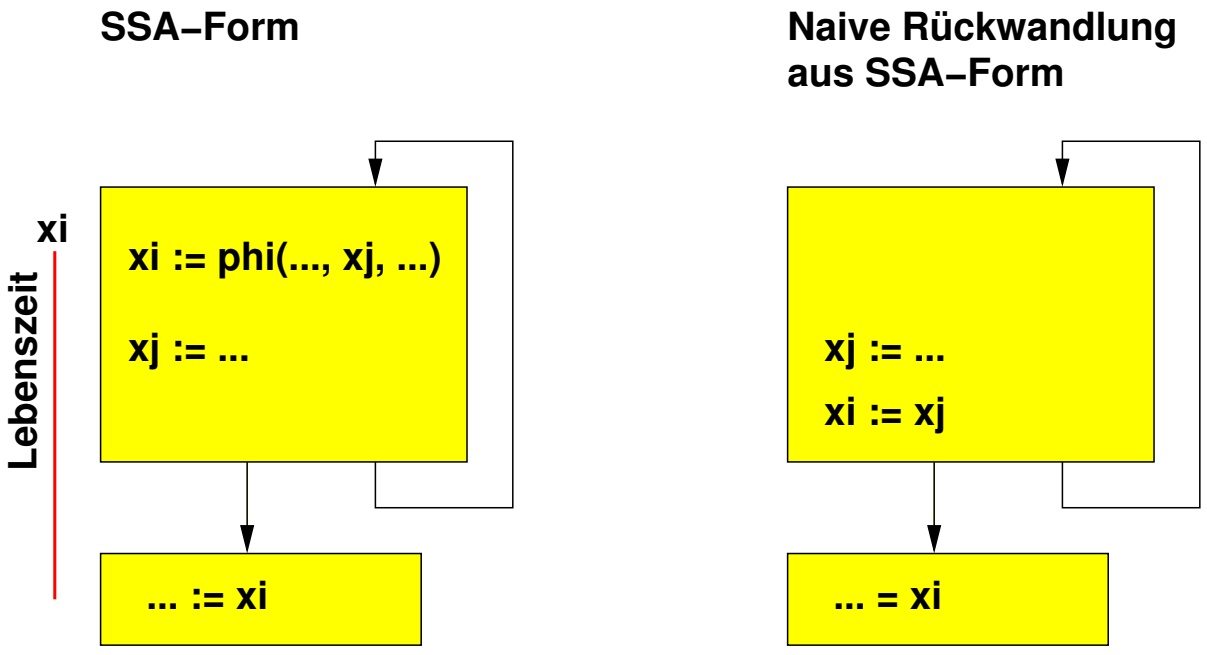
- Naive Umwandlung liefert fehlerhafte Ergebnisse
- Kopieranweisung auf `xi` mitten in dessen Lebenszeit hineingesetzt
- Liefert immer `xj` statt `xi`

Allgemeines Lost Copy-Problem 2



- Naive Umwandlung liefert fehlerhafte Ergebnisse
- Kopieranweisung auf `xi` mitten in dessen Lebenszeit hineingesetzt
- Liefert immer `xj` statt `xi`

Allgemeines Lost Copy-Problem 2



- Naive Umwandlung liefert fehlerhafte Ergebnisse
- Kopieranweisung auf `xi` mitten in dessen Lebenszeit hineingesetzt
- Liefert immer `xj` statt `xi`

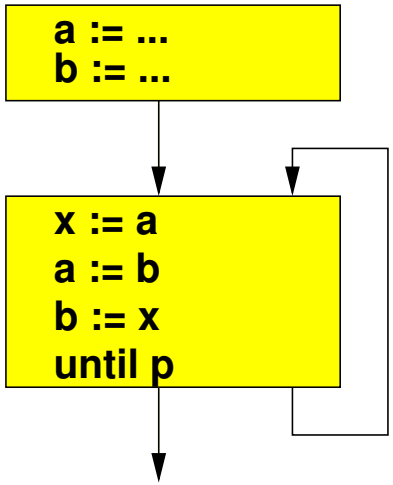
Lösung für “Lost Copy”-Problem

- Für genau welche Variablen müssen Sicherheitskopien erstellt werden?
- Wo müssen die jeweiligen Sicherheitskopien angelegt werden?

Fehlerhafte Vertauschung

Einfaches Beispiel: CFG zu SSA

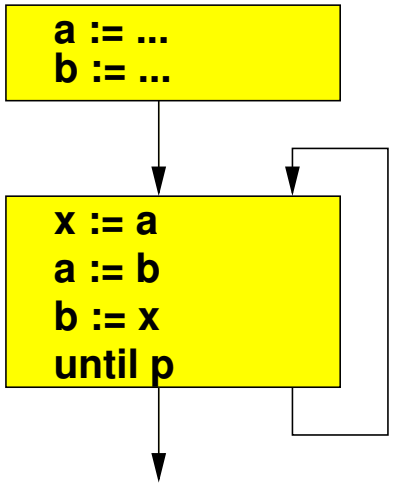
Normale Form



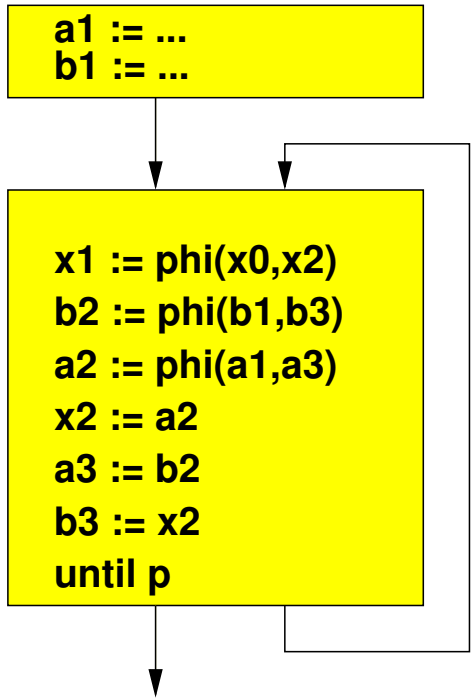
- Hierbei keine Überraschungen

Einfaches Beispiel: CFG zu SSA

Normale Form



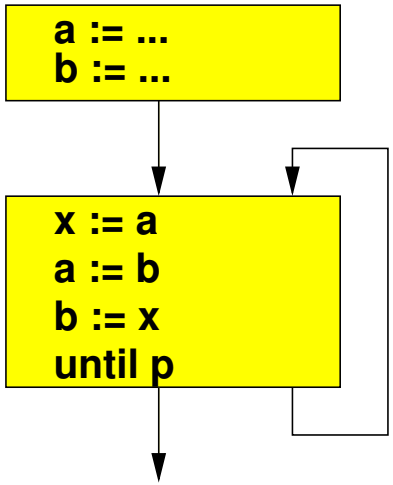
SSA–Form



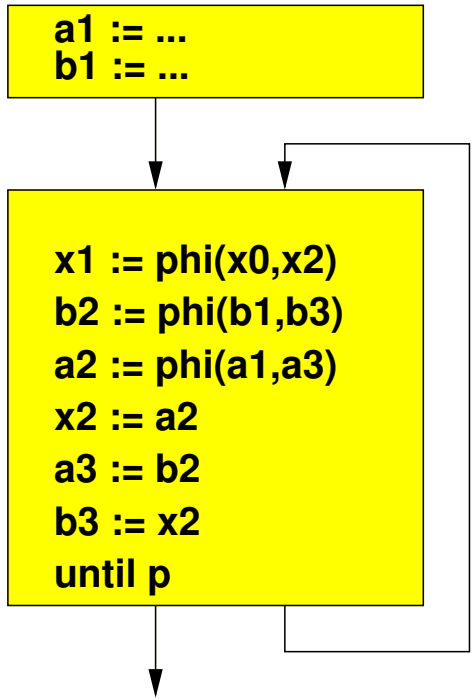
- Hierbei keine Überraschungen

Einfaches Beispiel: CFG zu SSA

Normale Form



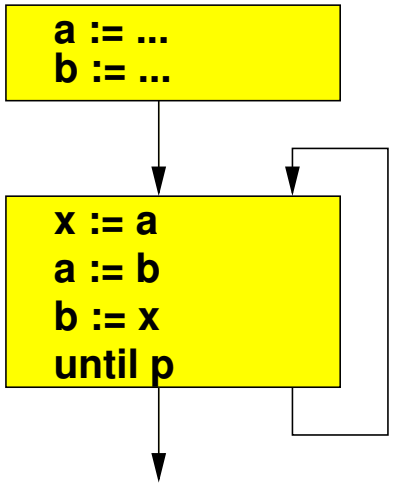
SSA–Form



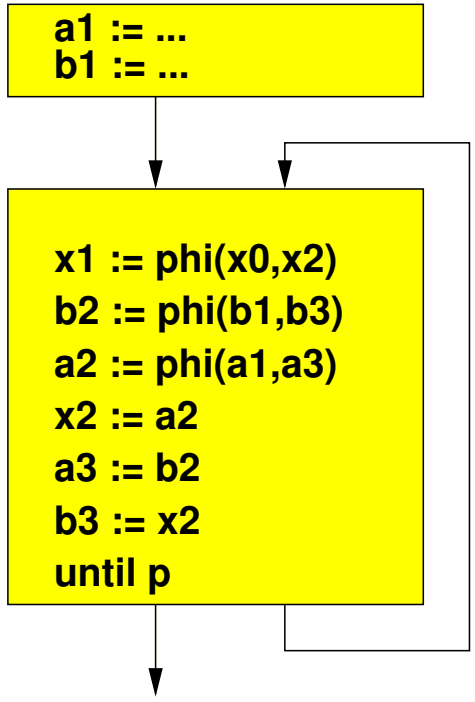
- Hierbei keine Überraschungen

Einfaches Beispiel: CFG zu SSA

Normale Form



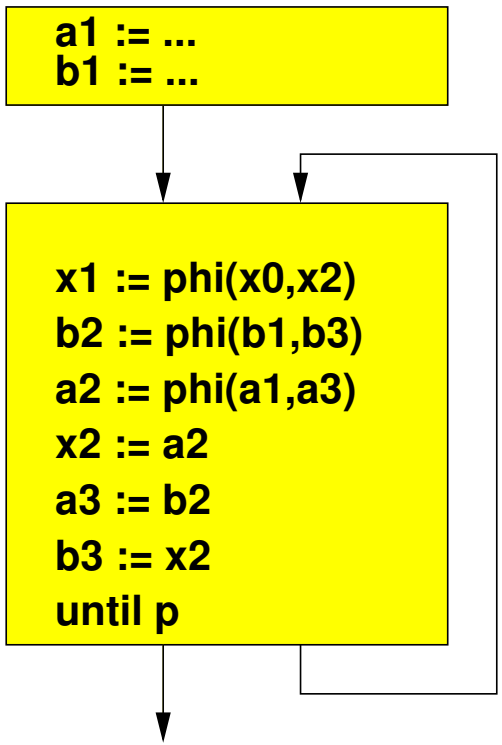
SSA–Form



- Hierbei keine Überraschungen

Einfaches Beispiel: SSA zu CFG

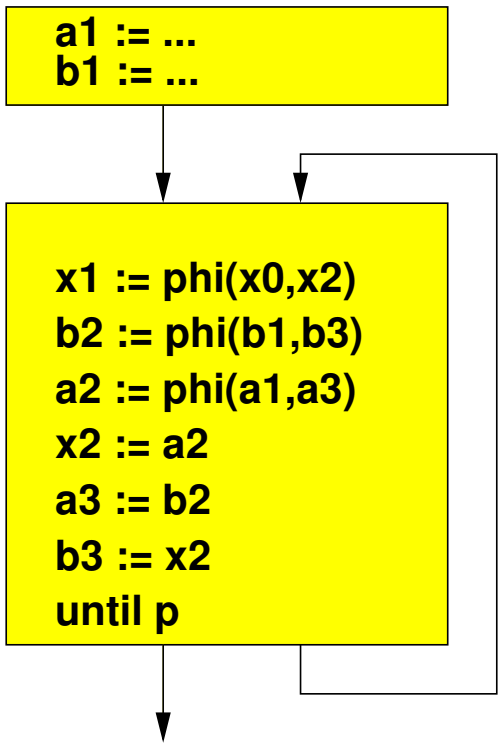
SSA–Form



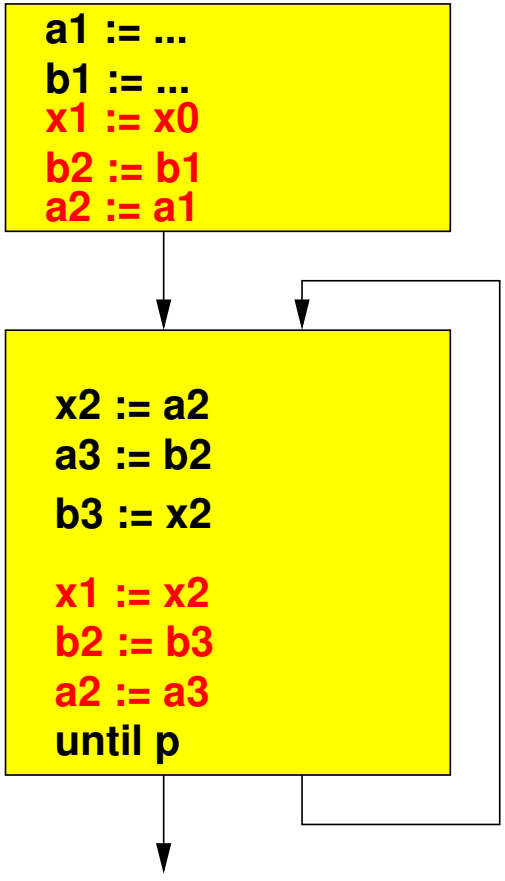
- Immer noch keine Überraschungen

Einfaches Beispiel: SSA zu CFG

SSA–Form



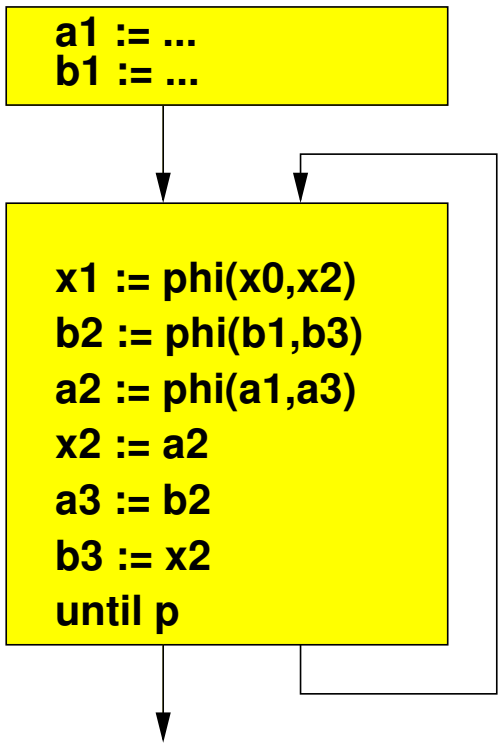
Rückwandlung aus SSA–Form



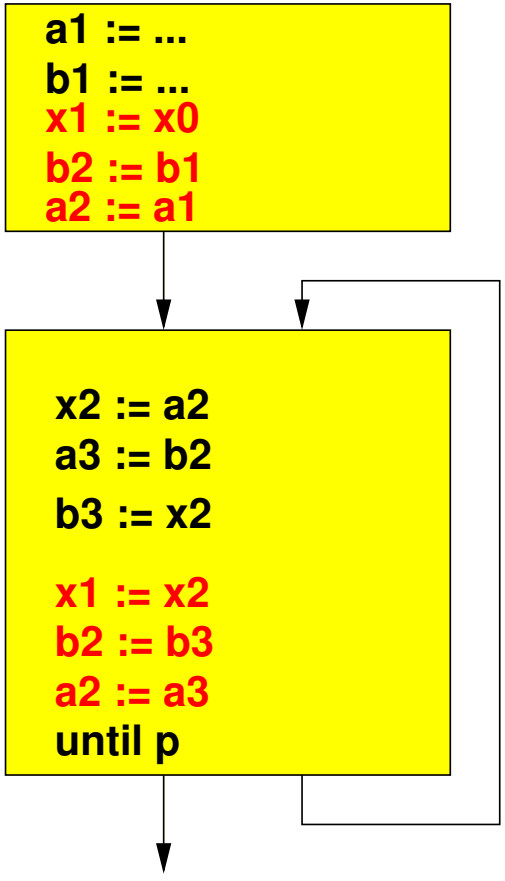
- Immer noch keine Überraschungen

Einfaches Beispiel: SSA zu CFG

SSA–Form



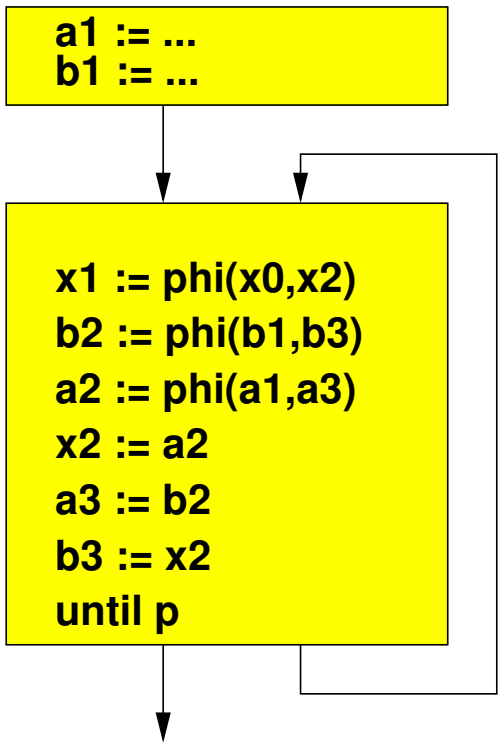
Rückwandlung aus SSA–Form



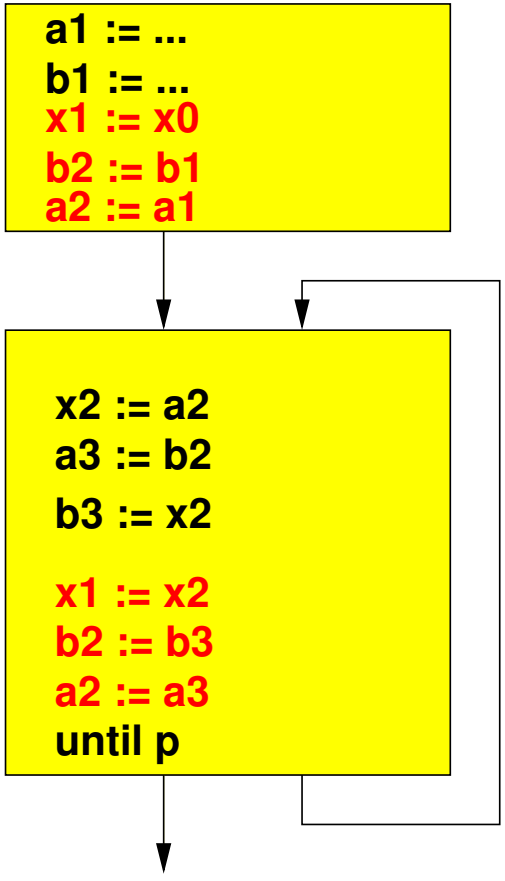
- Immer noch keine Überraschungen

Einfaches Beispiel: SSA zu CFG

SSA–Form



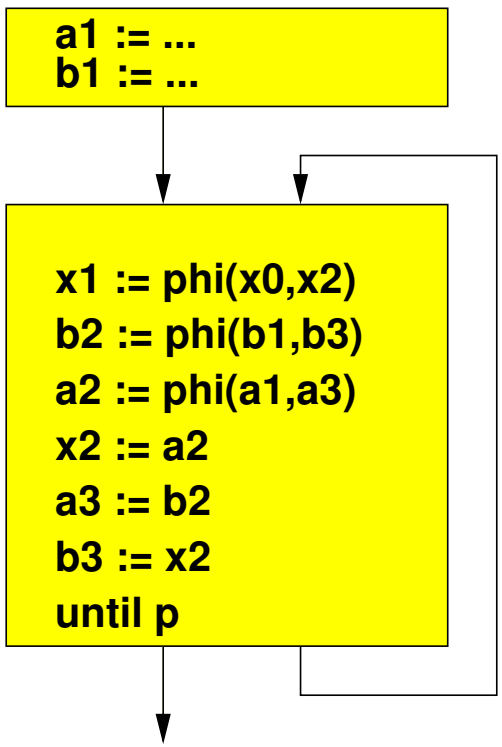
Rückwandlung aus SSA–Form



- Immer noch keine Überraschungen

Optimierung mit Copy Propagation

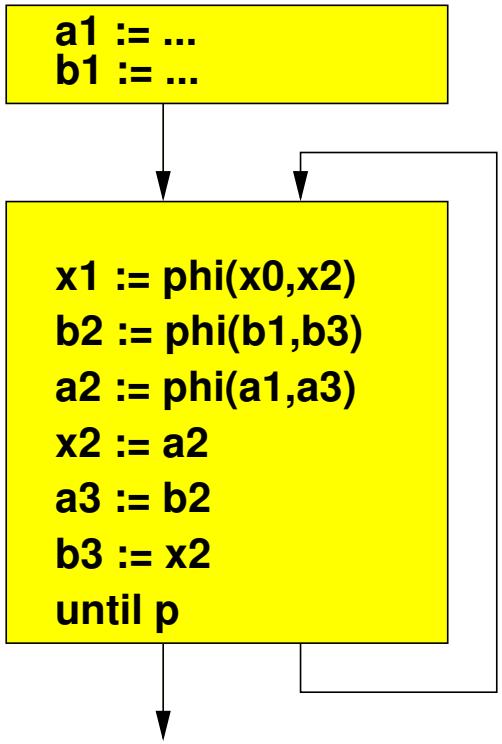
SSA–Form



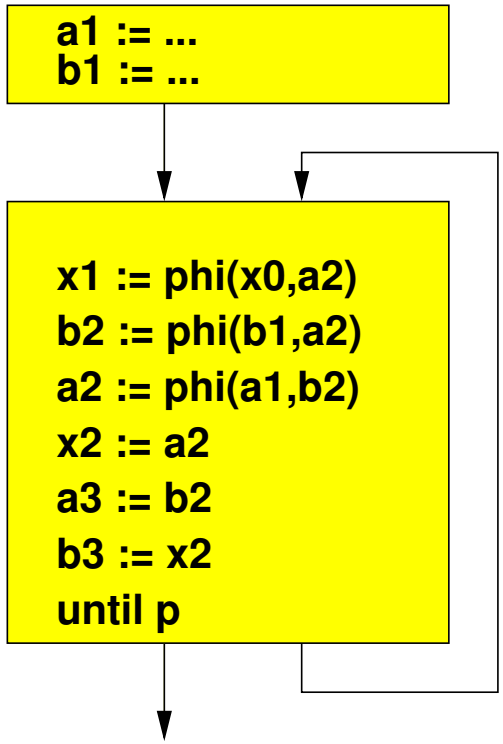
- Propagiert: $x2 := a2$, $a3 := b2$, $b3 := x2$

Optimierung mit Copy Propagation

SSA–Form



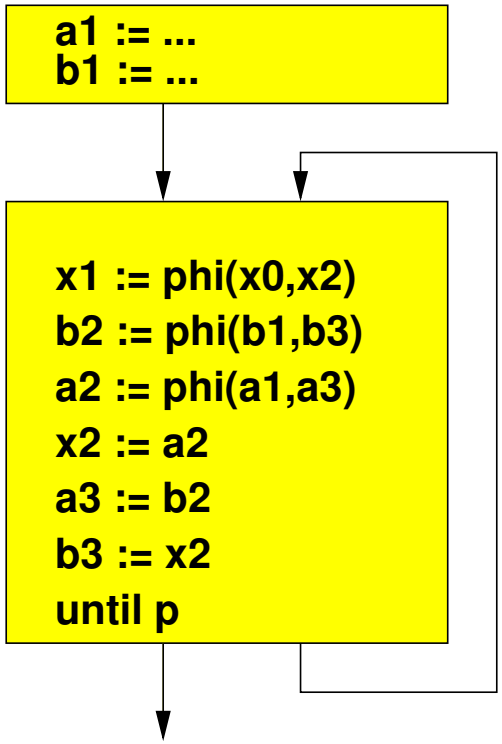
SSA–Form nach Copy Propagation



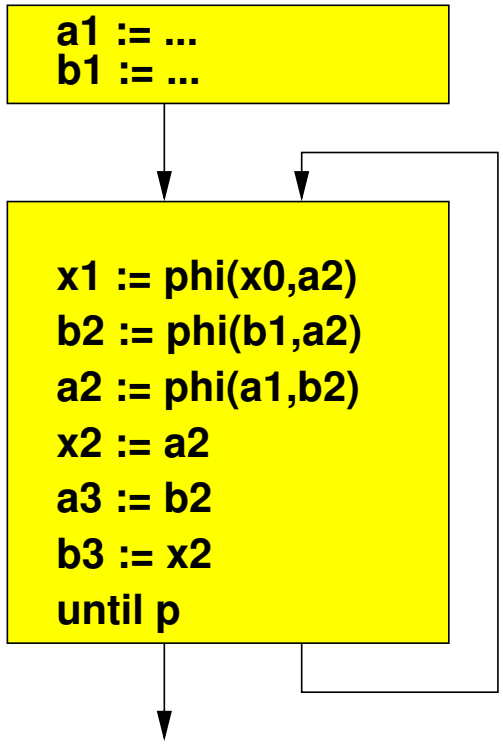
- Propagiert: $x2 := a2$, $a3 := b2$, $b3 := x2$

Optimierung mit Copy Propagation

SSA–Form



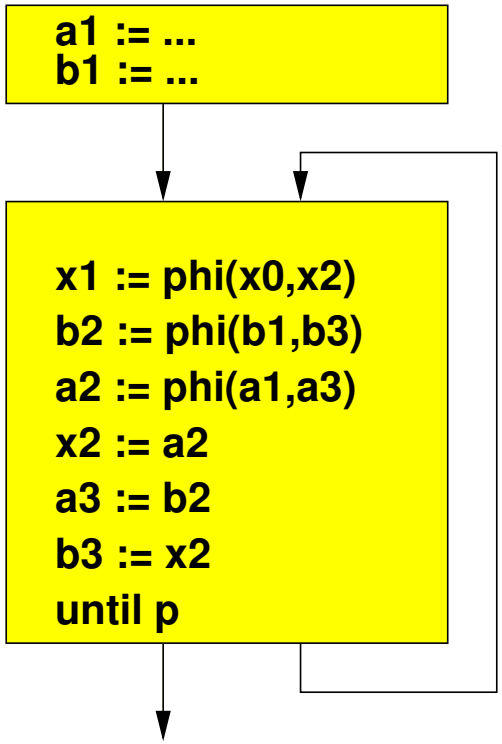
SSA–Form nach Copy Propagation



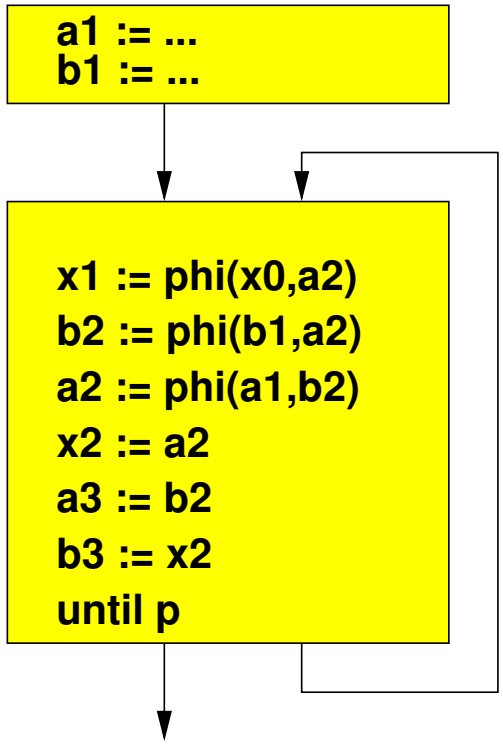
- Propagiert: $x2 := a2$, $a3 := b2$, $b3 := x2$

Optimierung mit Copy Propagation

SSA–Form



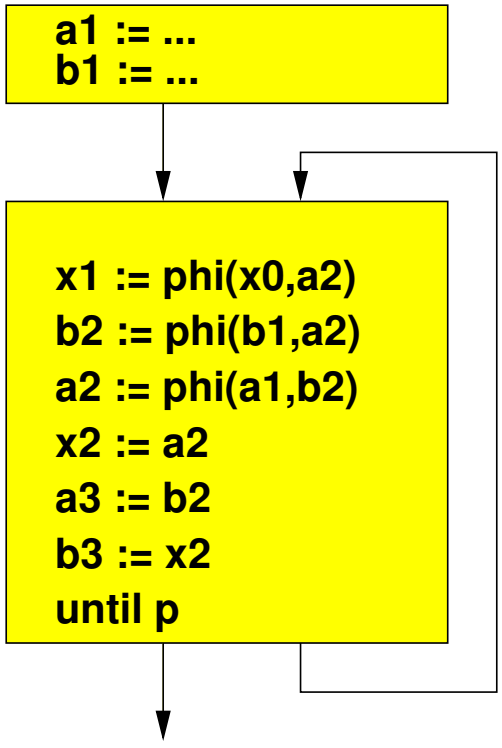
SSA–Form nach Copy Propagation



- Propagiert: `x2:=a2`, `a3:=b2`, `b3:=x2`

Weitere Optimierung mit Dead Code Elim.

SSA–Form nach Copy Propagation

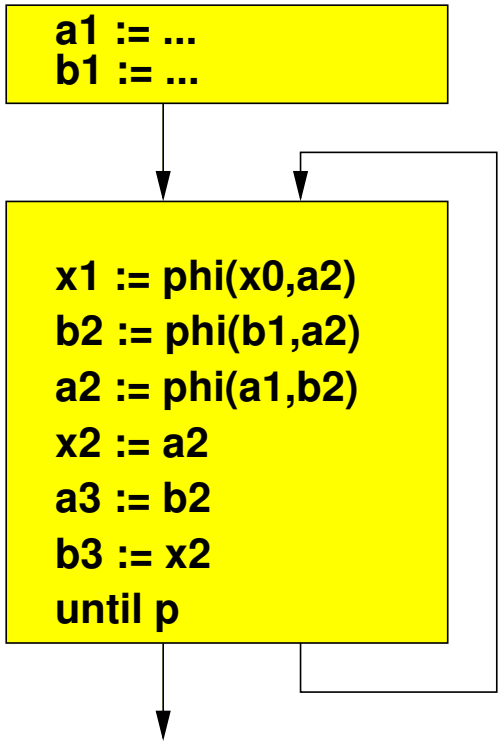


- Unnötige Kopieranweisungen und Phi-Funktion für `x1` entfernt

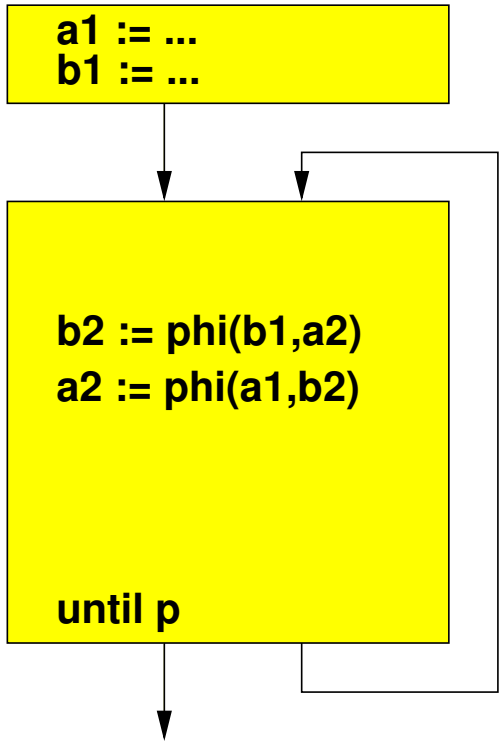
OptComp
A. Koch
Grundlagen
Lost Copy
Vertauschung
Ablaufplanung
Platzierung
Briggs-Algorithmus

Weitere Optimierung mit Dead Code Elim.

SSA–Form nach Copy Propagation



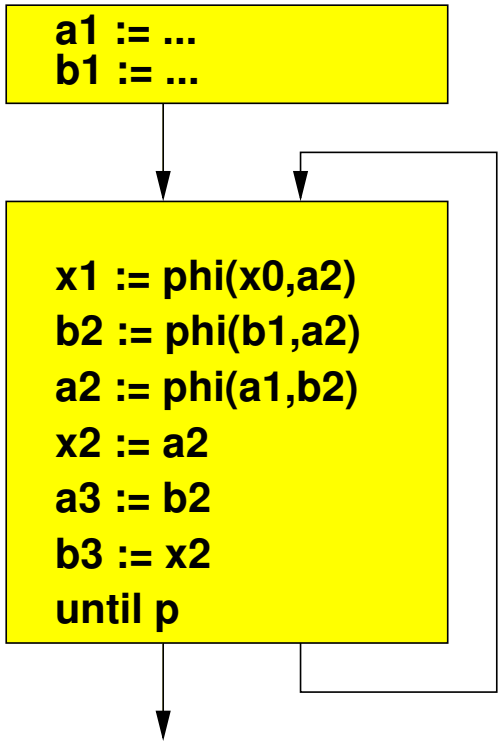
SSA–Form nach Copy Propagation und Dead Code–Elim.



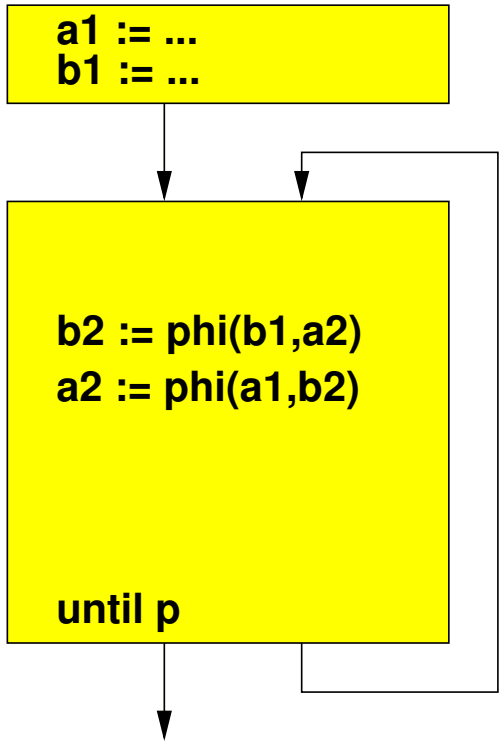
- Unnötige Kopieranweisungen und Phi-Funktion für `x1` entfernt

Weitere Optimierung mit Dead Code Elim.

SSA–Form nach Copy Propagation



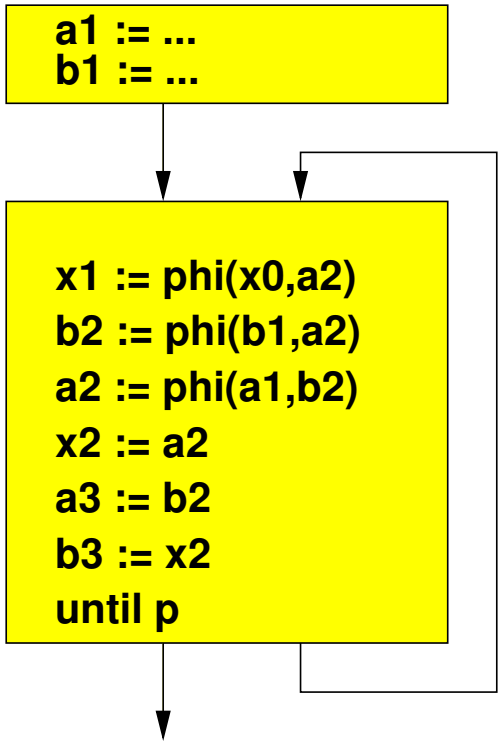
SSA–Form nach Copy Propagation und Dead Code–Elim.



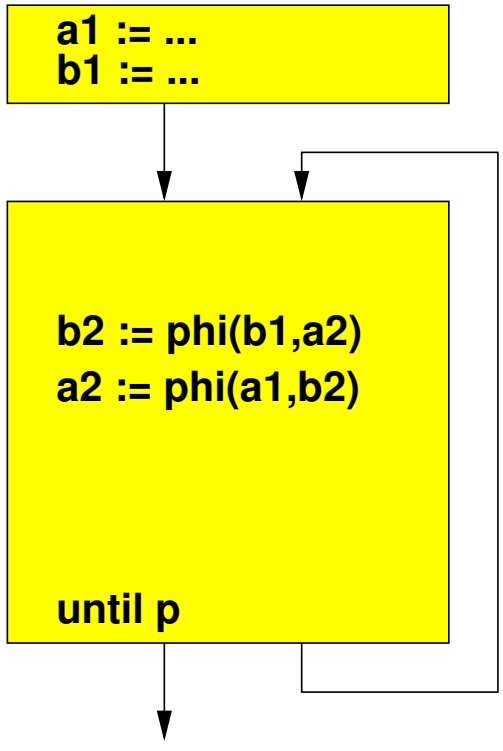
- Unnötige Kopieranweisungen und Phi-Funktion für `x1` entfernt

Weitere Optimierung mit Dead Code Elim.

SSA–Form nach Copy Propagation



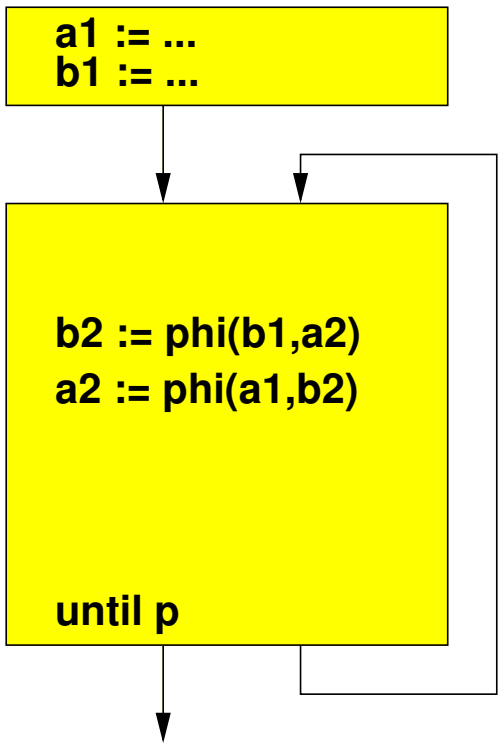
SSA–Form nach Copy Propagation und Dead Code–Elim.



- Unnötige Kopieranweisungen und Phi-Funktion für `x1` entfernt

Einfache SSA-Rückwandlung nach CFG

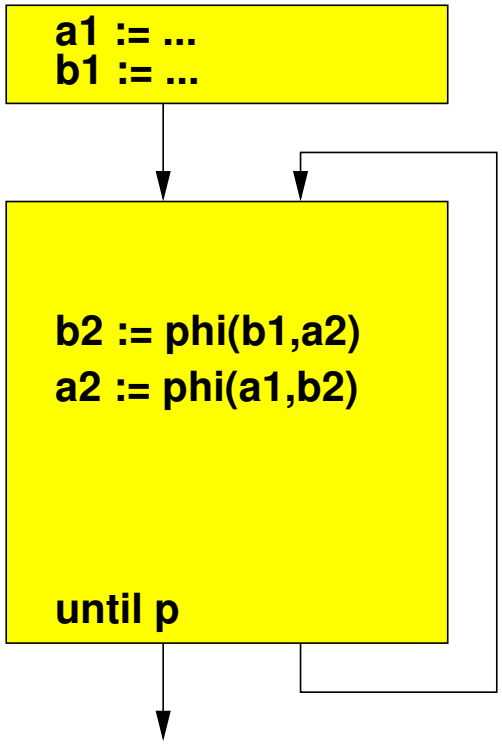
**SSA–Form nach
Copy Propagation
und Dead Code–Elim.**



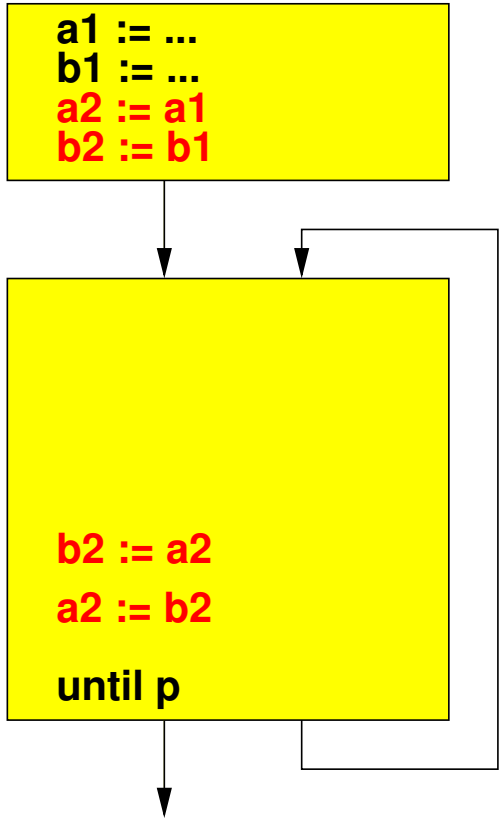
- Fehler: Keine Vertauschung mehr
- Formal werden alle Phi-Funktionen parallel ausgeführt
- Kopieranweisungen aber sequentiell

Einfache SSA-Rückwandlung nach CFG

SSA–Form nach Copy Propagation und Dead Code–Elim.



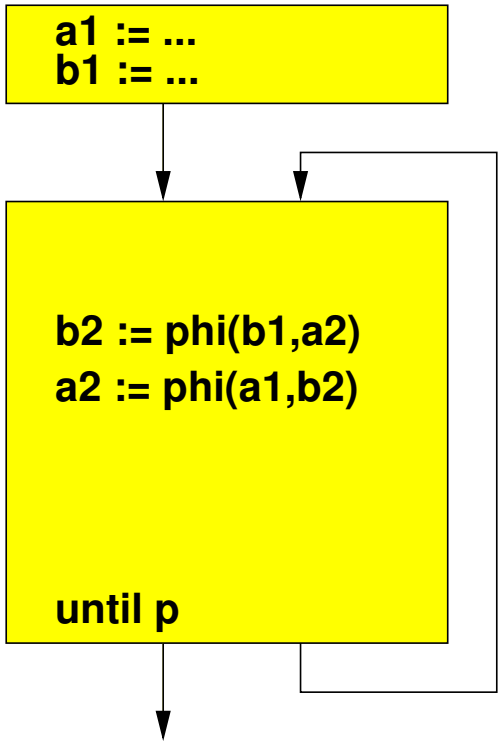
Naive Rückwandlung aus SSA–Form



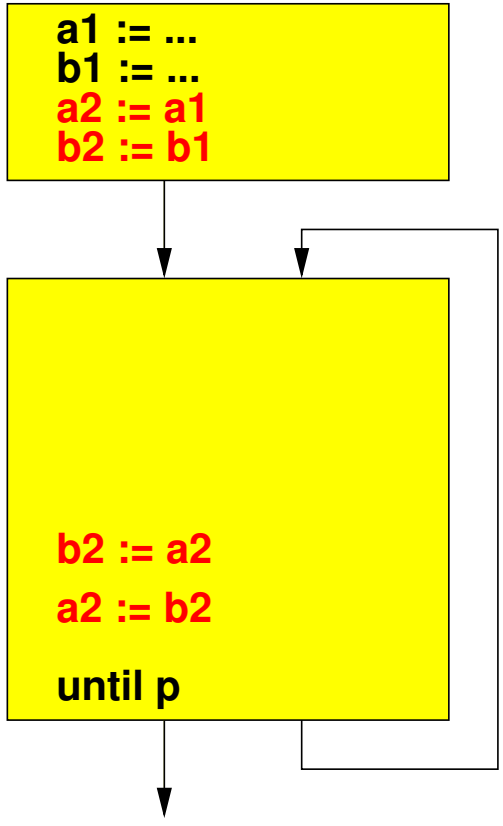
- Fehler: Keine Vertauschung mehr
- Formal werden alle Phi-Funktionen parallel ausgeführt
- Kopieranweisungen aber sequentiell

Einfache SSA-Rückwandlung nach CFG

SSA–Form nach Copy Propagation und Dead Code–Elim.

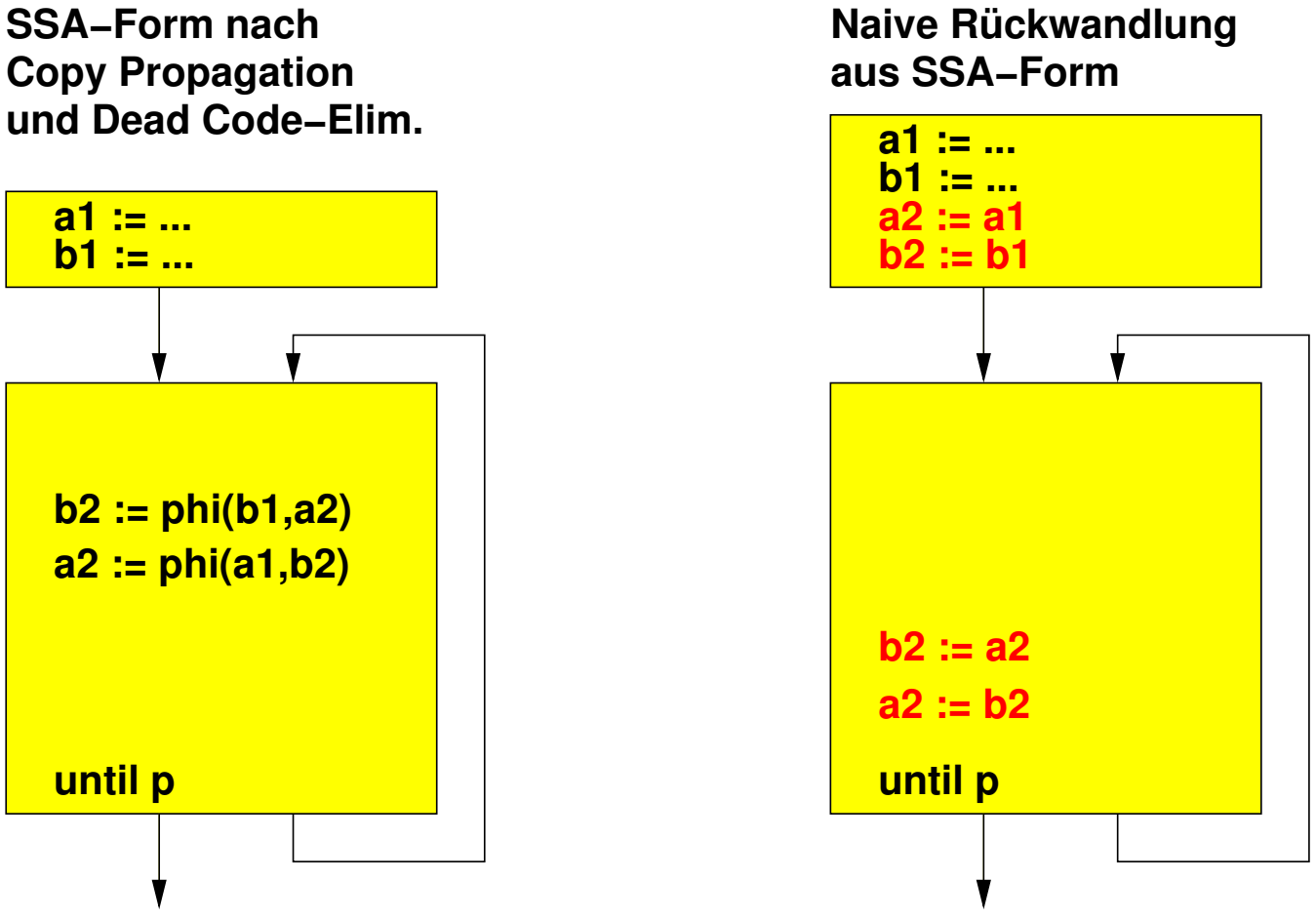


Naive Rückwandlung aus SSA–Form



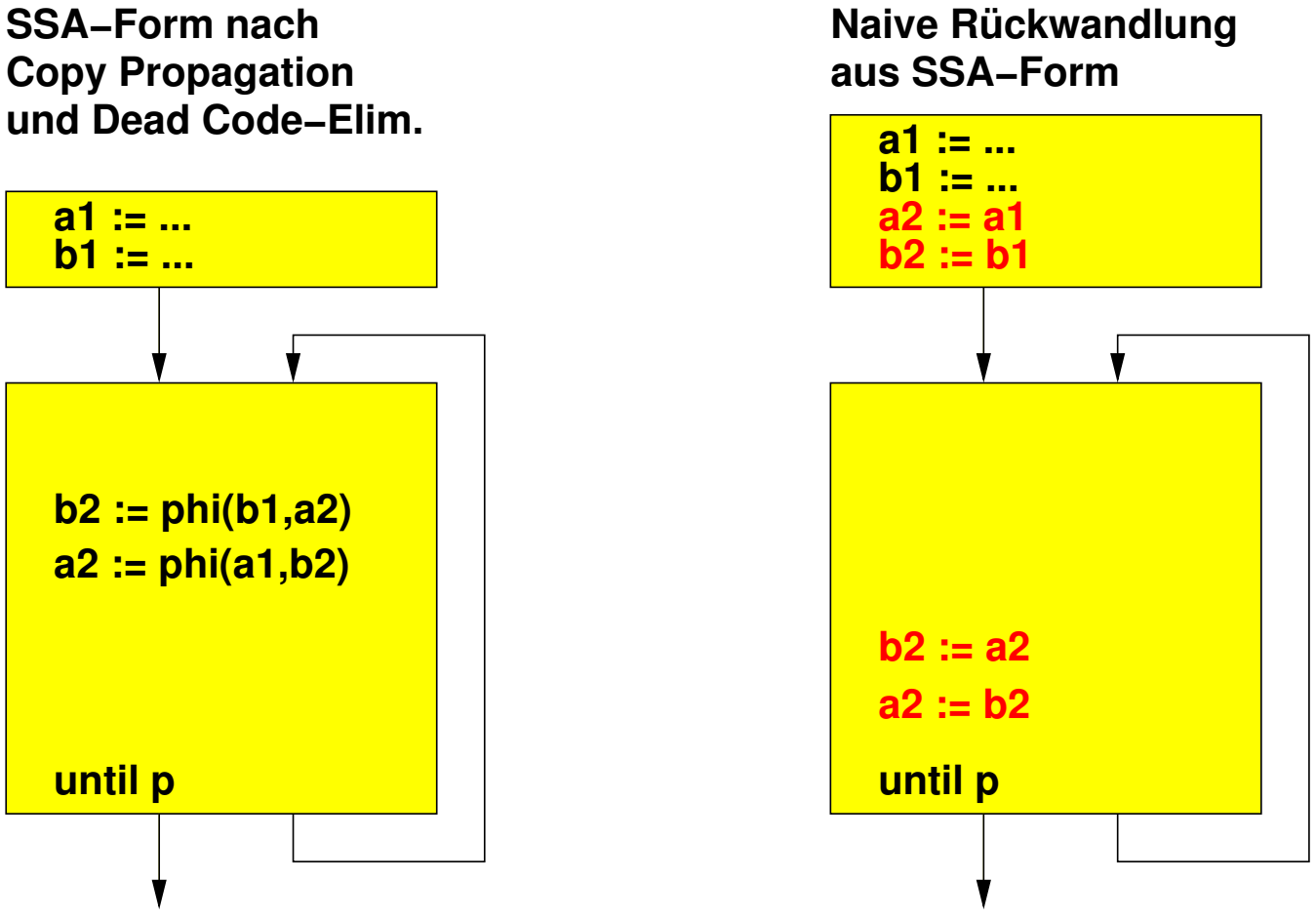
- Fehler: Keine Vertauschung mehr
- Formal werden alle Phi-Funktionen parallel ausgeführt
- Kopieranweisungen aber sequentiell

Einfache SSA-Rückwandlung nach CFG



- Fehler: Keine Vertauschung mehr
- Formal werden alle Phi-Funktionen parallel ausgeführt
- Kopieranweisungen aber sequentiell

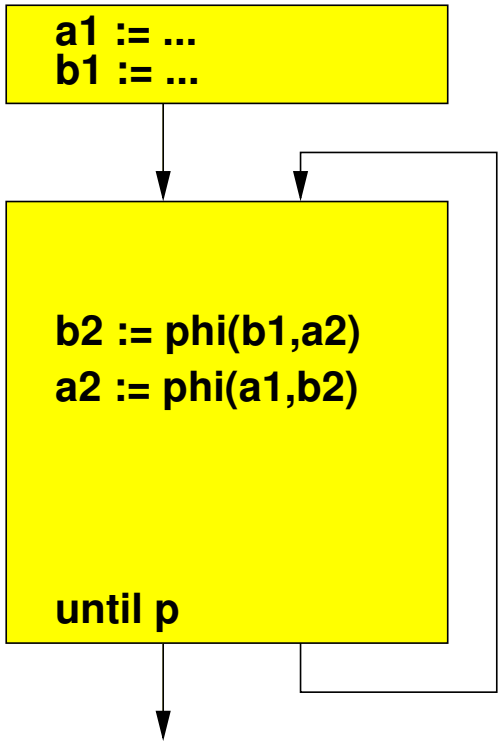
Einfache SSA-Rückwandlung nach CFG



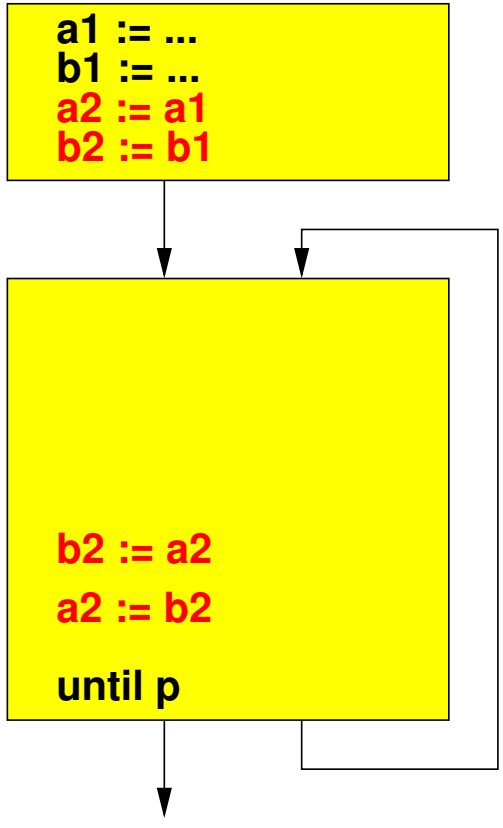
- Fehler: Keine Vertauschung mehr
- Formal werden alle Phi-Funktionen parallel ausgeführt
- Kopieranweisungen aber sequentiell

Einfache SSA-Rückwandlung nach CFG

SSA–Form nach Copy Propagation und Dead Code–Elim.

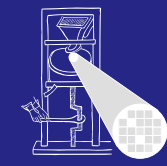


Naive Rückwandlung aus SSA–Form

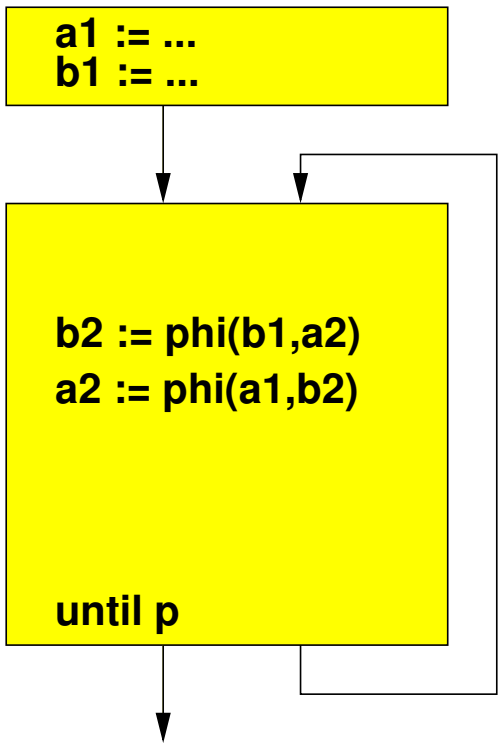


- Fehler: Keine Vertauschung mehr
- Formal werden alle Phi-Funktionen parallel ausgeführt
- Kopieranweisungen aber sequentiell

Diskussion des Fehlers



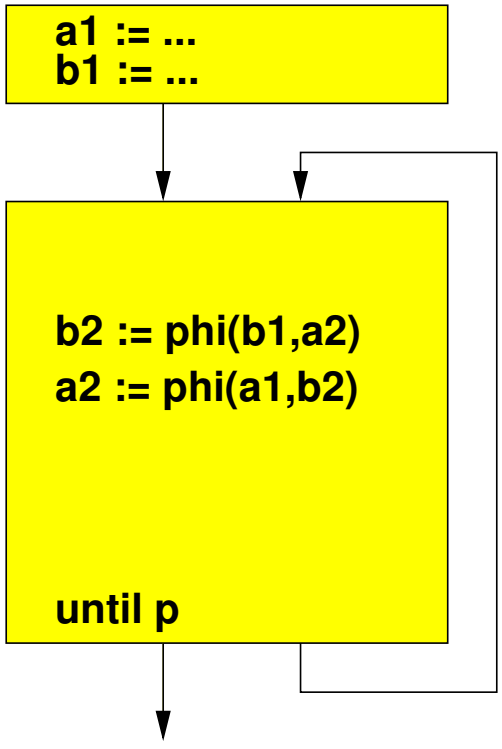
SSA–Form nach Copy Propagation und Dead Code–Elim.

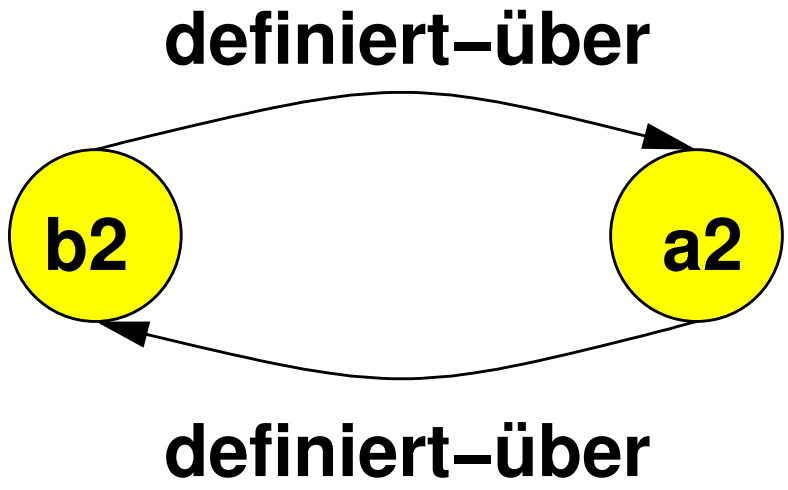


- Korrektes Vorgehen: Zyklus aufbrechen
- Einen der benötigten Werte kopieren, dann Kopie benutzen

Diskussion des Fehlers

SSA–Form nach Copy Propagation und Dead Code–Elim.



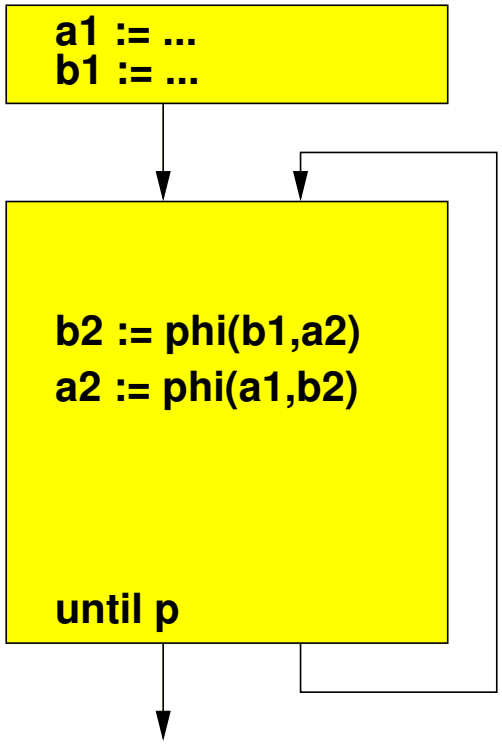


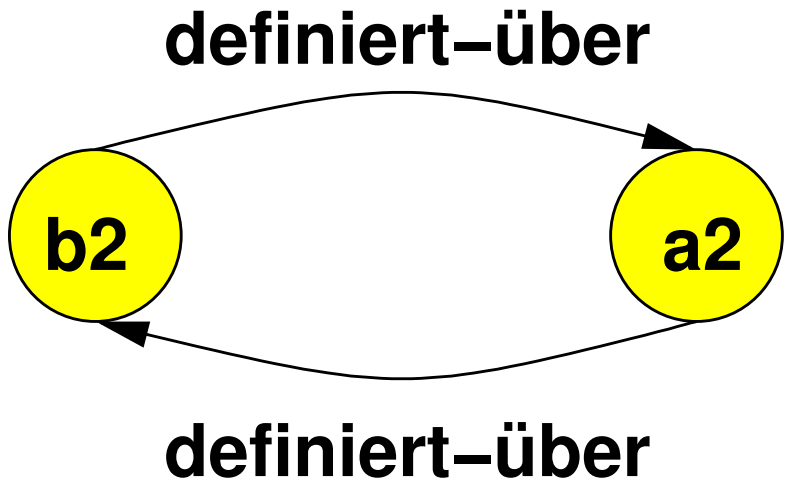
- Kopieranweisungen voneinander abhängig
- Welche zuerst ausführen?
- Korrektes Vorgehen: Zyklus aufbrechen
- Einen der benötigten Werte kopieren, dann Kopie benutzen

OptComp
A. Koch
Grundlagen
Lost Copy
Vertauschung
Ablaufplanung
Platzierung
Briggs-Algorithmus

Diskussion des Fehlers

SSA–Form nach Copy Propagation und Dead Code–Elim.



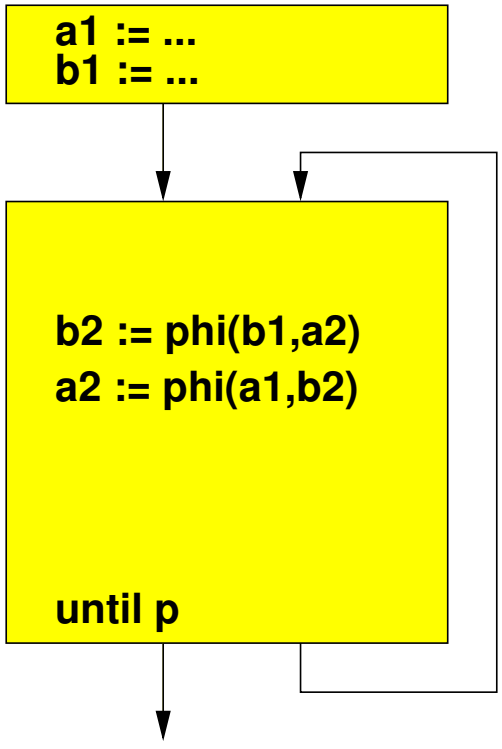


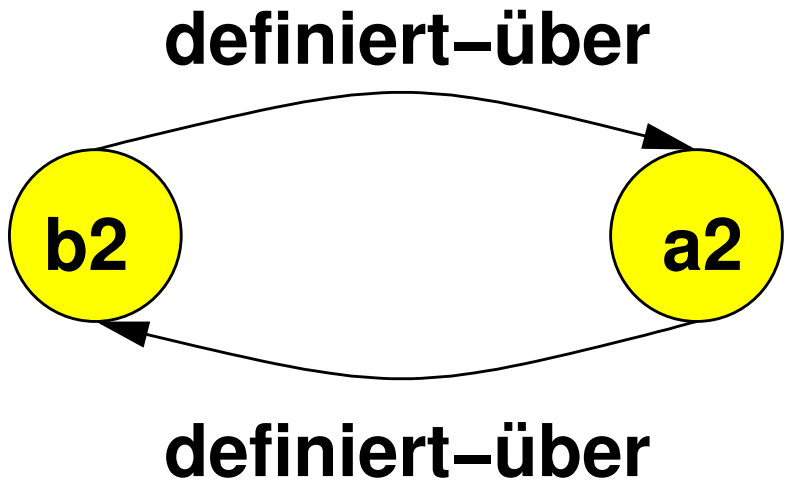
- Kopieranweisungen voneinander abhängig
- Welche zuerst ausführen?
- Korrektes Vorgehen: Zyklus aufbrechen
- Einen der benötigten Werte kopieren, dann Kopie benutzen

OptComp
A. Koch
Grundlagen
Lost Copy
Vertauschung
Ablaufplanung
Platzierung
Briggs-Algorithmus

Diskussion des Fehlers

SSA–Form nach Copy Propagation und Dead Code–Elim.



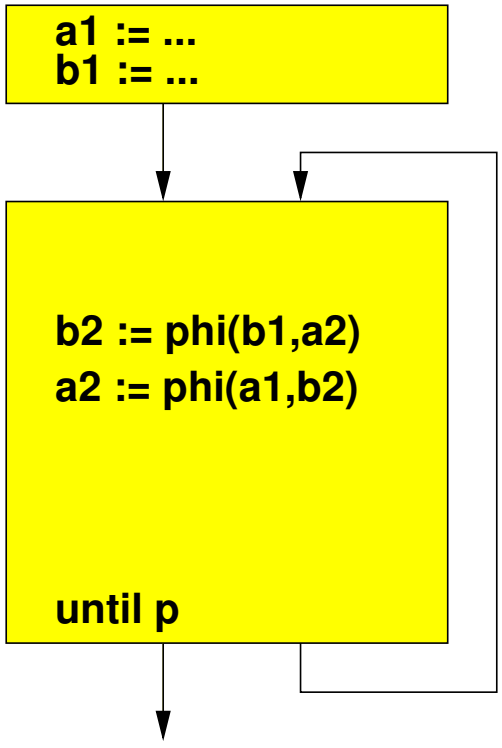


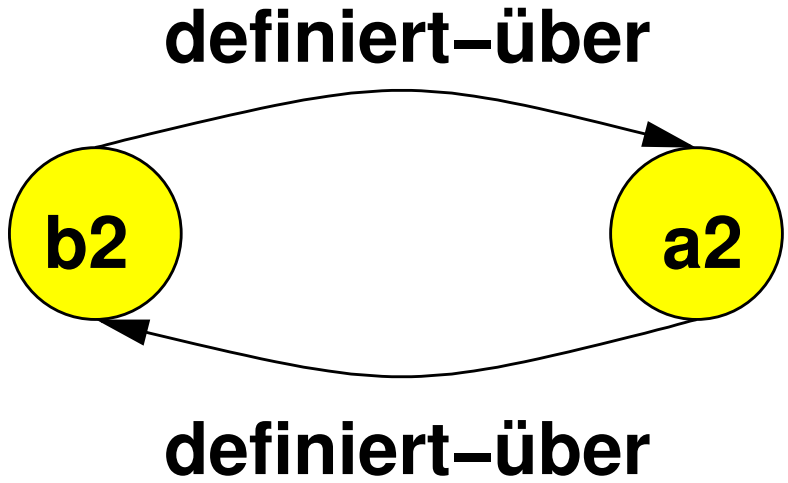
- Kopieranweisungen voneinander abhängig
- Welche zuerst ausführen?

- Korrektes Vorgehen: Zyklus aufbrechen
- Einen der benötigten Werte kopieren, dann Kopie benutzen

Diskussion des Fehlers

SSA–Form nach Copy Propagation und Dead Code–Elim.





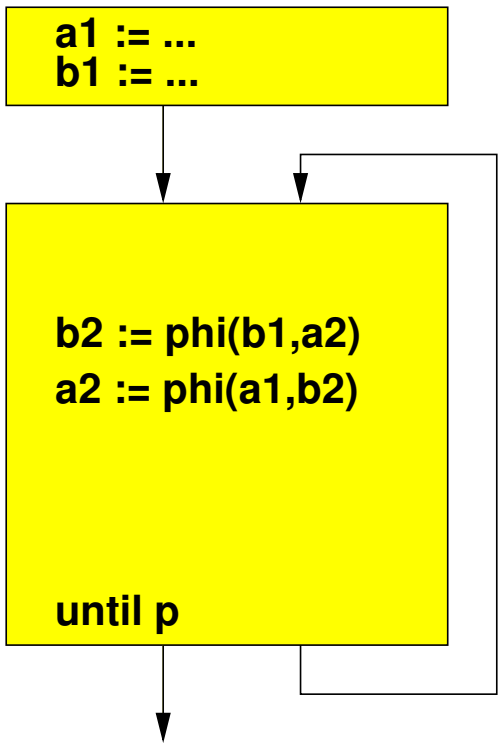
- Kopieranweisungen voneinander abhängig
- Welche zuerst ausführen?

- Korrektes Vorgehen: Zyklus aufbrechen
- Einen der benötigten Werte kopieren, dann Kopie benutzen

OptComp
A. Koch
Grundlagen
Lost Copy
Vertauschung
Ablaufplanung
Platzierung
Briggs-Algorithmus

Korrekte SSA-Rückwandlung nach CFG

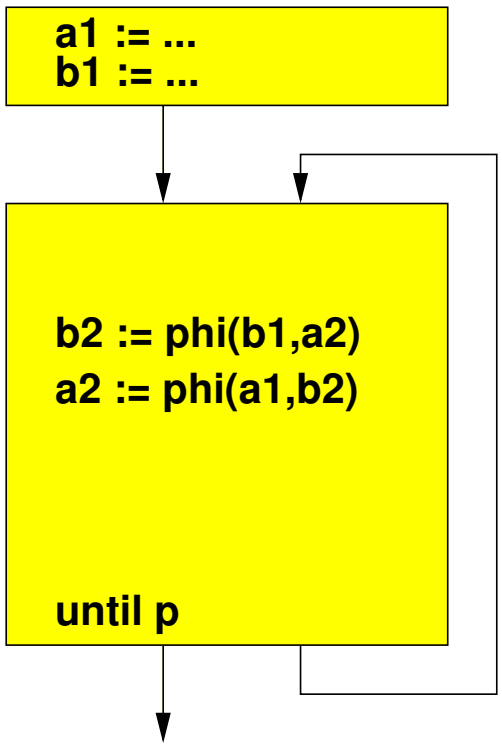
**SSA–Form nach
Copy Propagation
und Dead Code–Elim.**



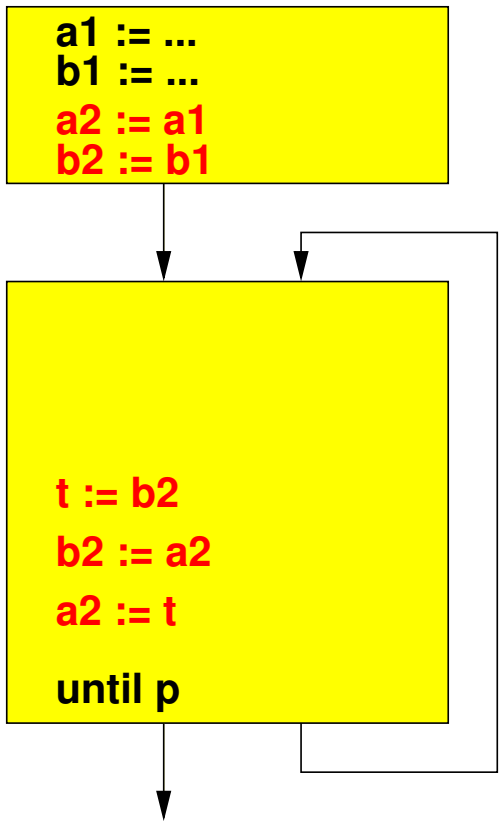
- Rechnet nun korrekt

Korrekte SSA-Rückwandlung nach CFG

SSA–Form nach Copy Propagation und Dead Code–Elim.



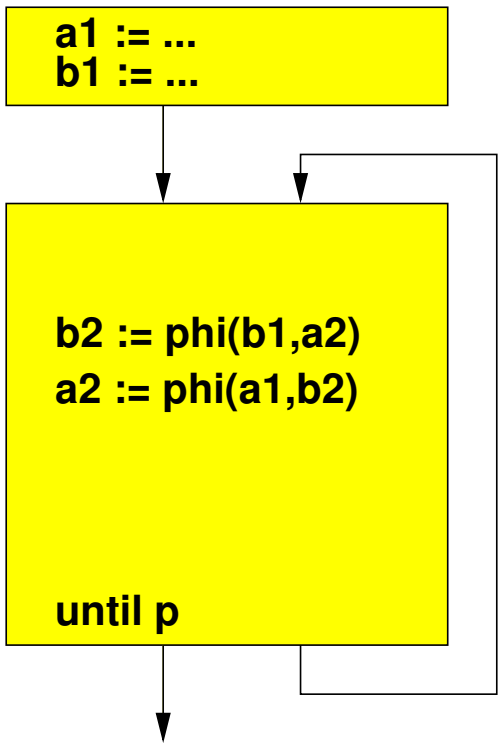
Korrekte Rückwandlung aus SSA–Form



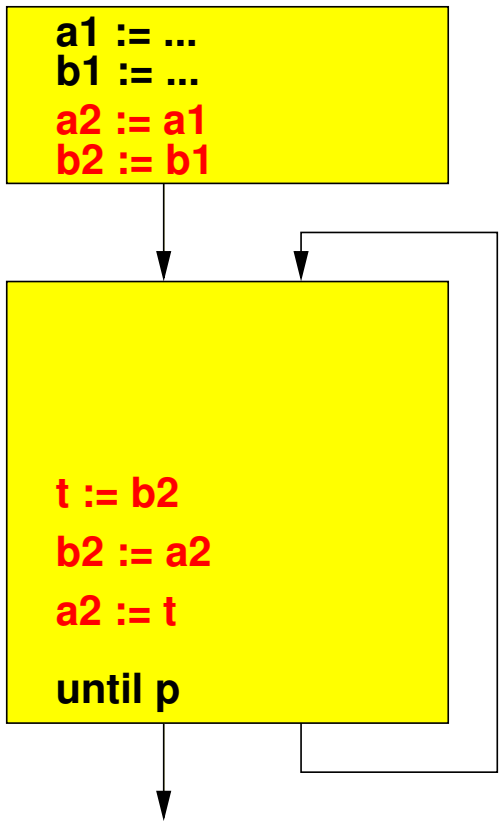
- Rechnet nun korrekt

Korrekte SSA-Rückwandlung nach CFG

SSA–Form nach Copy Propagation und Dead Code–Elim.



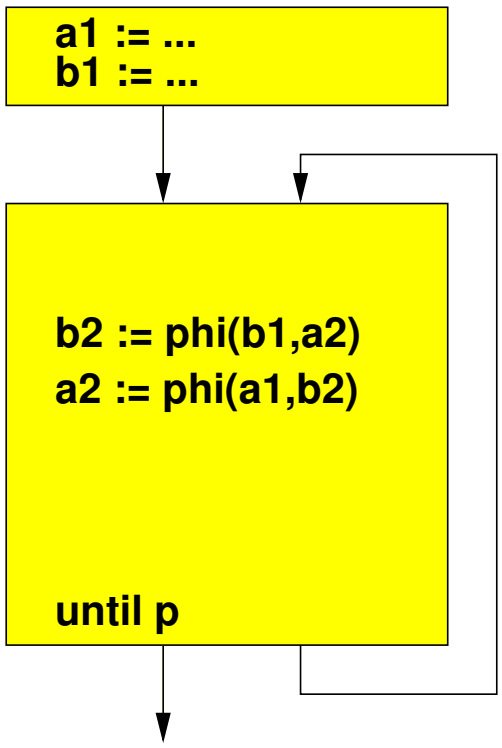
Korrekte Rückwandlung aus SSA–Form



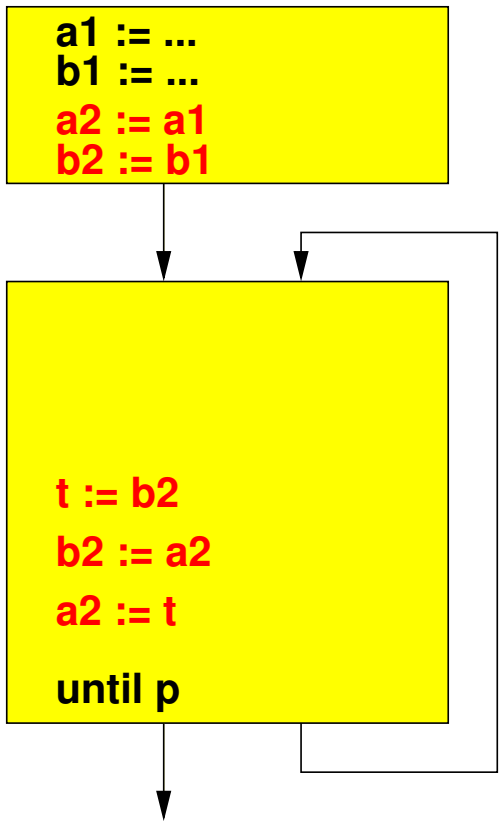
- Rechnet nun korrekt

Korrekte SSA-Rückwandlung nach CFG

SSA–Form nach Copy Propagation und Dead Code–Elim.



Korrekte Rückwandlung aus SSA–Form



- Rechnet nun korrekt

Lösung für Vertauschungsproblem

Abhängigkeitszyklen durch Einführen einer block-lokalen Kopie eines Phi-Funktion-Parameters aufbrechen.

- Unterschied zur Lösung des “Lost Copy”-Problems
- Dort block-übergreifende Sicherheitskopie von Phi-Funktions-Ergebnis

Sind bei allen Arten von Abhängigkeiten zwischen Phi-Funktionen block-lokale Kopien erforderlich?

Lösung für Vertauschungsproblem

Abhängigkeitszyklen durch Einführen einer block-lokalen Kopie eines Phi-Funktion-Parameters aufbrechen.

- Unterschied zur Lösung des “Lost Copy”-Problems
- Dort block-übergreifende Sicherheitskopie von Phi-Funktions-Ergebnis

Sind bei allen Arten von Abhängigkeiten zwischen Phi-Funktionen block-lokale Kopien erforderlich?

Lösung für Vertauschungsproblem

Abhängigkeitszyklen durch Einführen einer block-lokalen Kopie eines Phi-Funktion-Parameters aufbrechen.

- Unterschied zur Lösung des “Lost Copy”-Problems
- Dort block-übergreifende Sicherheitskopie von Phi-Funktions-Ergebnis

Sind bei allen Arten von Abhängigkeiten zwischen Phi-Funktionen block-lokale Kopien erforderlich?

Lösung für Vertauschungsproblem

Abhängigkeitszyklen durch Einführen einer block-lokalen Kopie eines Phi-Funktion-Parameters aufbrechen.

- Unterschied zur Lösung des “Lost Copy”-Problems
- Dort block-übergreifende Sicherheitskopie von Phi-Funktions-Ergebnis

Sind bei allen Arten von Abhängigkeiten zwischen Phi-Funktionen block-lokale Kopien erforderlich?

Lösung für Vertauschungsproblem

Abhängigkeitszyklen durch Einführen einer block-lokalen Kopie eines Phi-Funktion-Parameters aufbrechen.

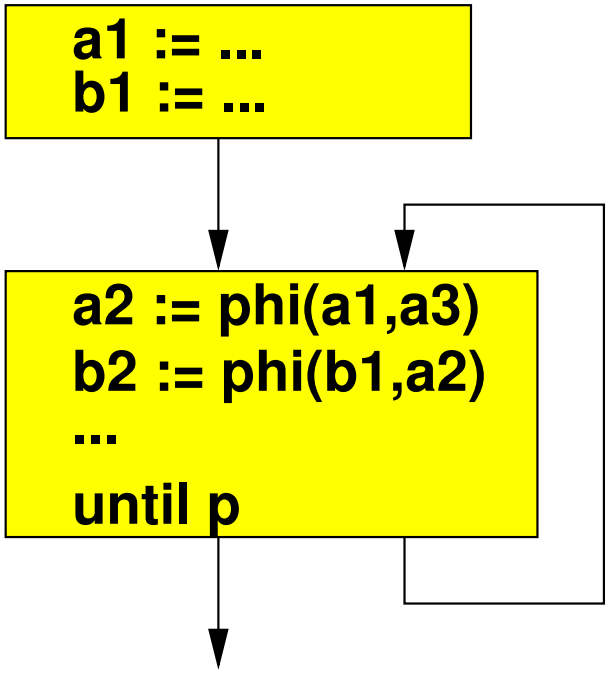
- Unterschied zur Lösung des “Lost Copy”-Problems
- Dort block-übergreifende Sicherheitskopie von Phi-Funktions-Ergebnis

Sind bei allen Arten von Abhängigkeiten zwischen Phi-Funktionen block-lokale Kopien erforderlich?

Ablaufplanung statt Kopieren

Rückwandlung von SSA nach CFG

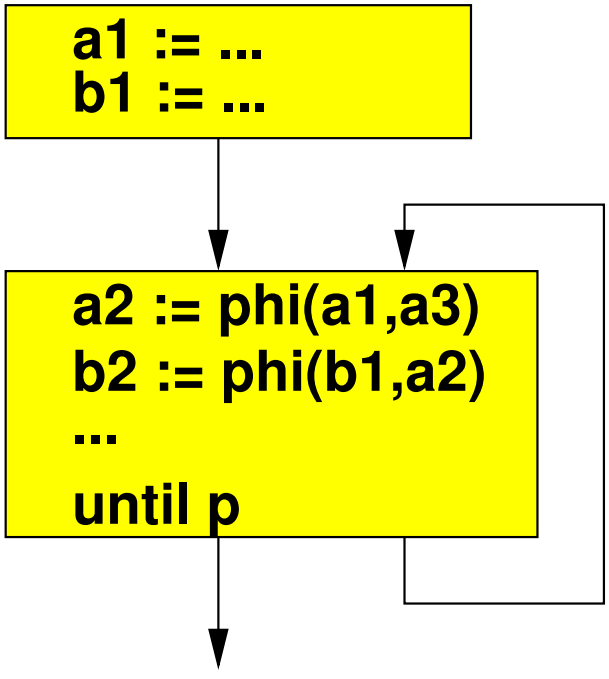
SSA–Form



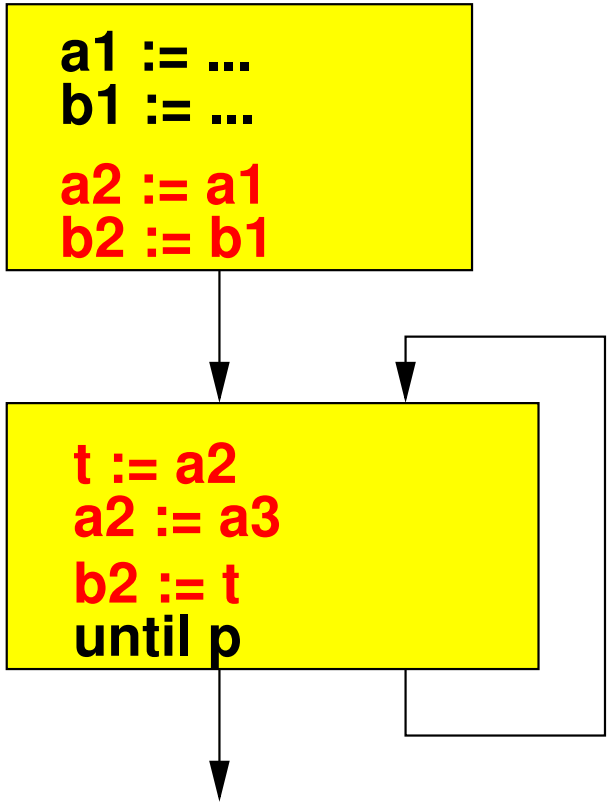
- b2 ist von a2 abhängig ⟶ a2 lokal kopieren
- Rechnet korrekt
- Kopie ist aber unnötiger Aufwand!

Rückwandlung von SSA nach CFG

SSA–Form



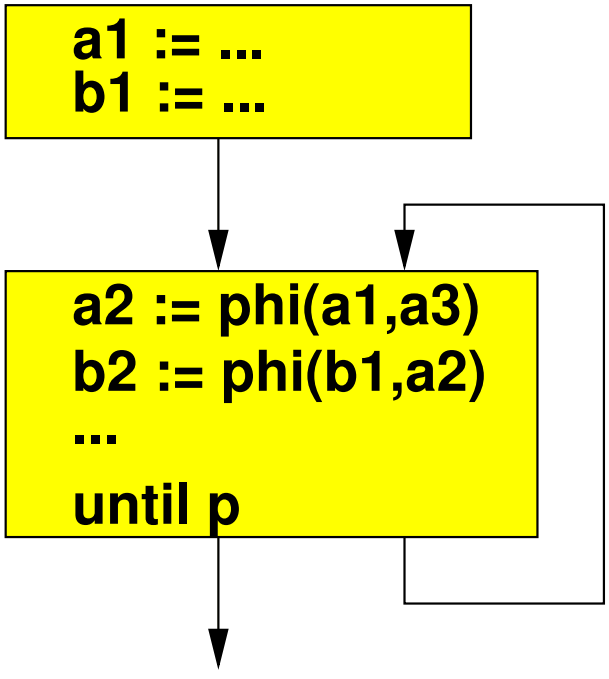
Rücktransformation



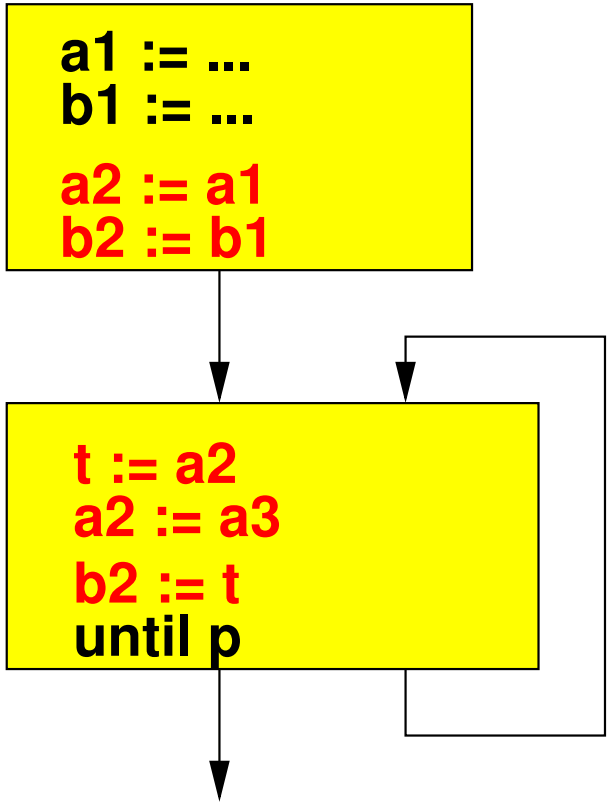
- b2 ist von a2 abhängig ⟶ a2 lokal kopieren
- Rechnet korrekt
- Kopie ist aber unnötiger Aufwand!

Rückwandlung von SSA nach CFG

SSA–Form



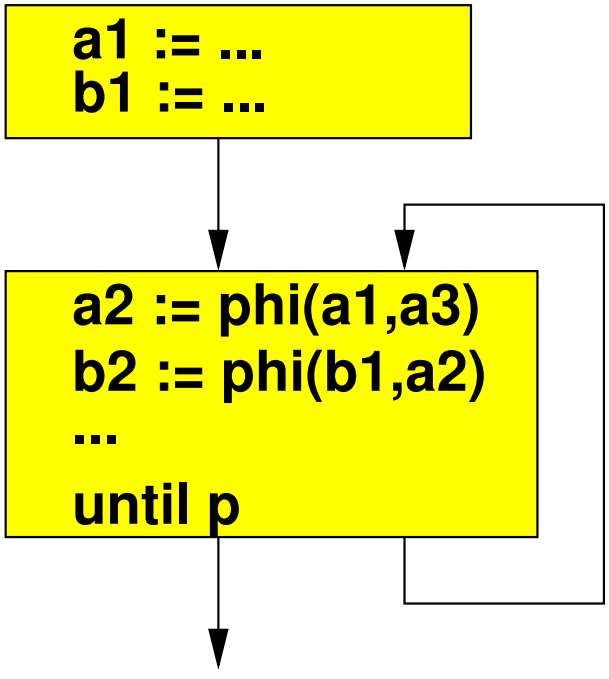
Rücktransformation



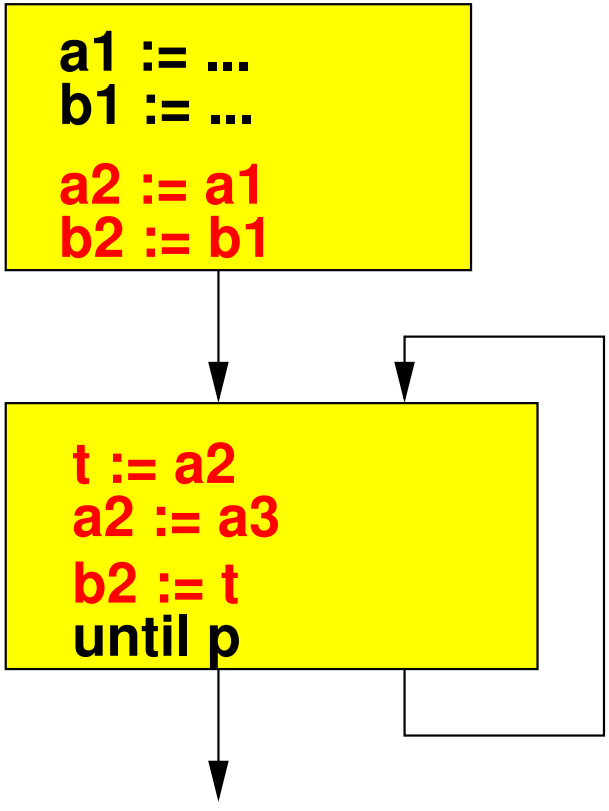
- b2 ist von a2 abhängig ⟶ a2 lokal kopieren
- Rechnet korrekt
- Kopie ist aber unnötiger Aufwand!

Rückwandlung von SSA nach CFG

SSA–Form



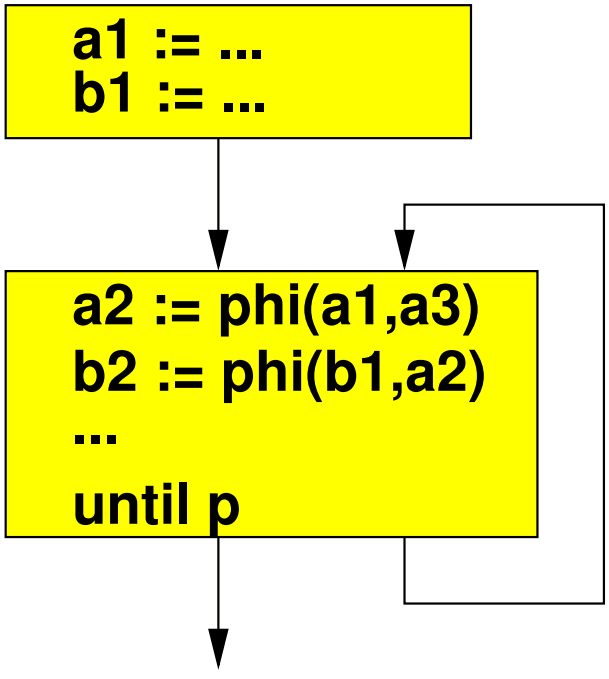
Rücktransformation



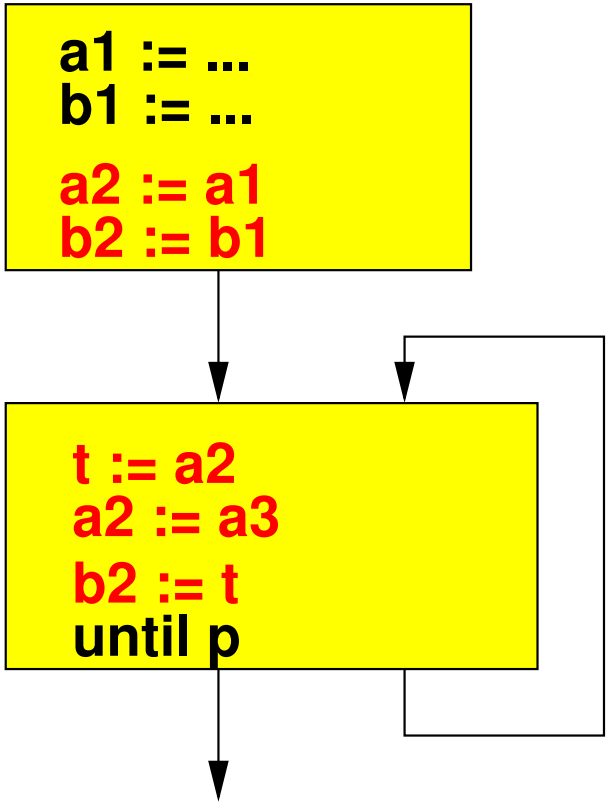
- `b2` ist von `a2` abhängig → `a2` lokal kopieren
- Rechnet korrekt
- Kopie ist aber unnötiger Aufwand!

Rückwandlung von SSA nach CFG

SSA–Form



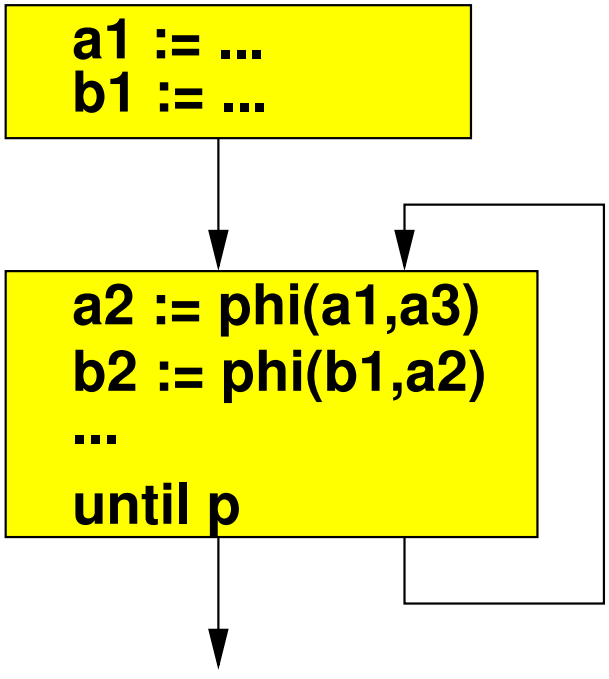
Rücktransformation



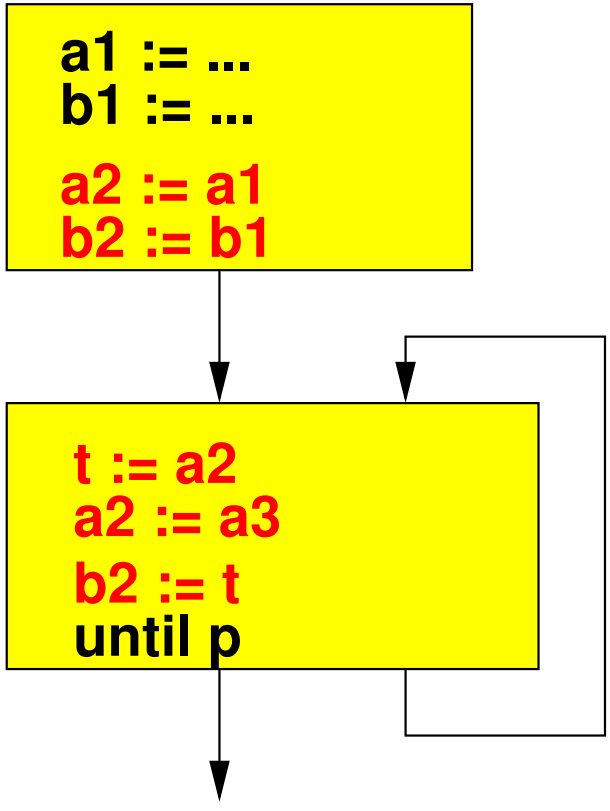
- `b2` ist von `a2` abhängig → `a2` lokal kopieren
- Rechnet korrekt
- Kopie ist aber unnötiger Aufwand!

Rückwandlung von SSA nach CFG

SSA–Form

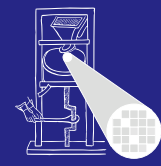


Rücktransformation

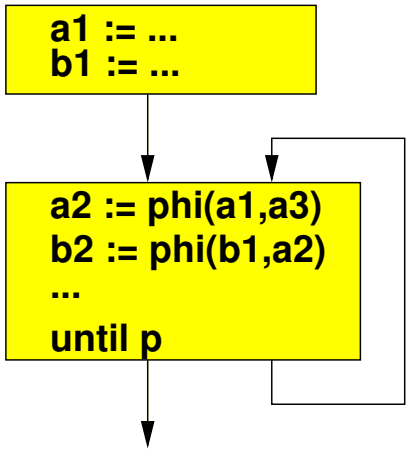


- `b2` ist von `a2` abhängig → `a2` lokal kopieren
- Rechnet korrekt
- Kopie ist aber unnötiger Aufwand!

Elegantere Lösung



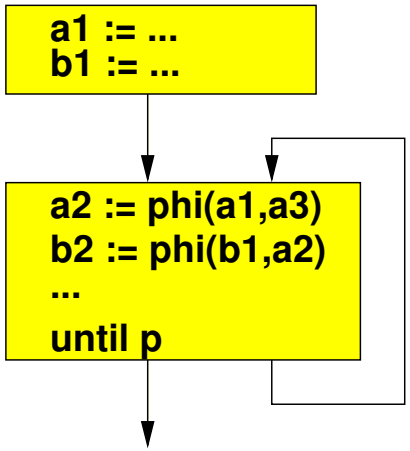
SSA–Form



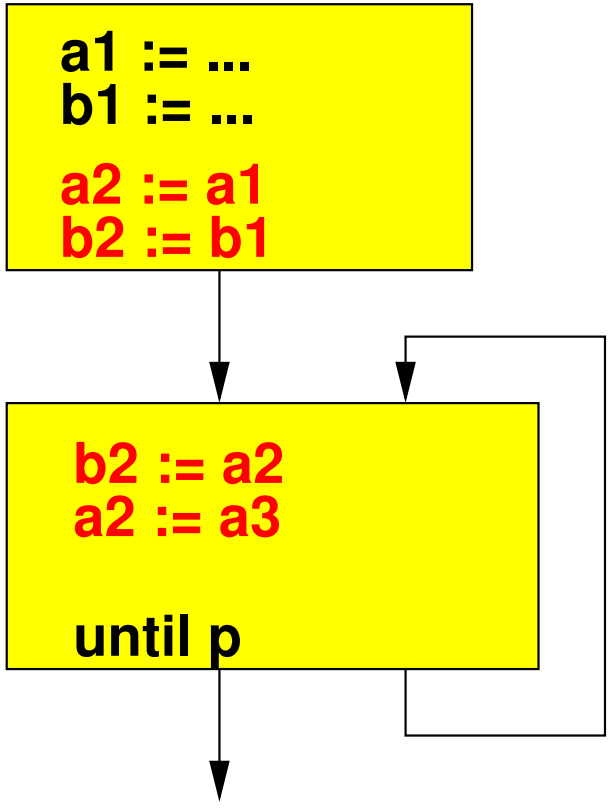
- Keine Kopie nötig, da keine zirkuläre Abhängigkeit
- Geschickte Ablaufplanung der Kopien
- Topologische Reihenfolge im Graphen

Elegantere Lösung

SSA–Form



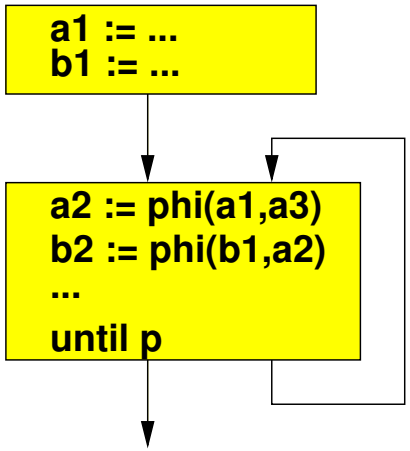
Rücktransformation



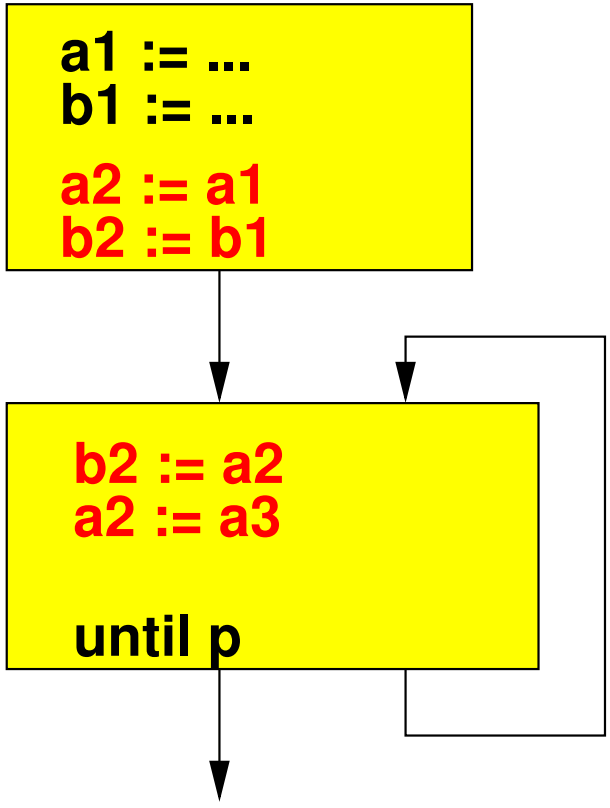
- Keine Kopie nötig, da keine zirkuläre Abhängigkeit
- Geschickte Ablaufplanung der Kopien
- Topologische Reihenfolge im Graphen

Elegantere Lösung

SSA–Form

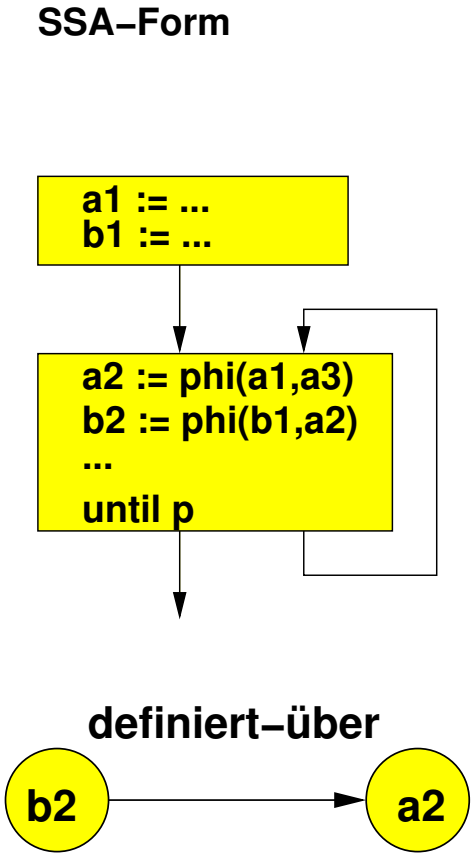


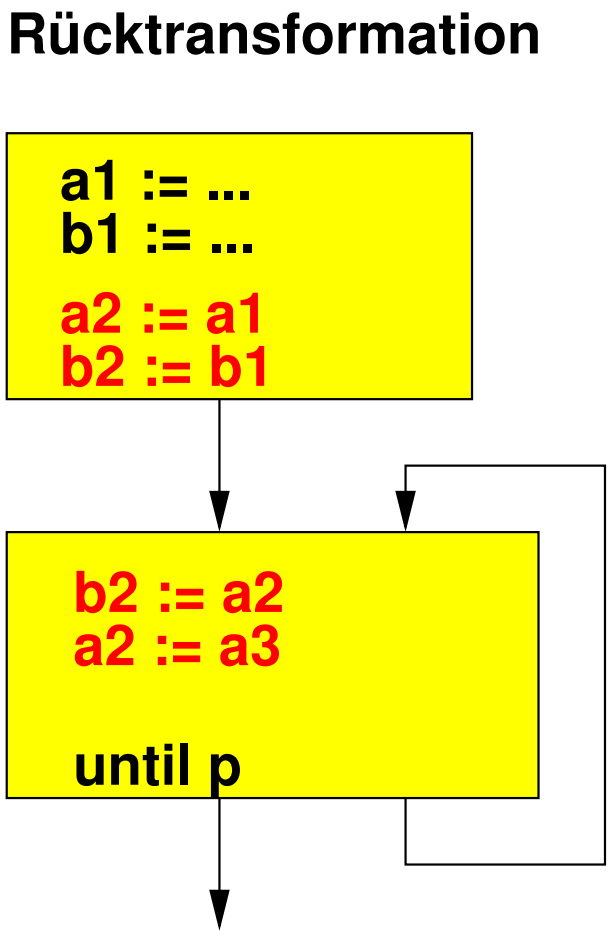
Rücktransformation



- Keine Kopie nötig, da keine zirkuläre Abhängigkeit
- Geschickte Ablaufplanung der Kopien
- Topologische Reihenfolge im Graphen

Elegantere Lösung

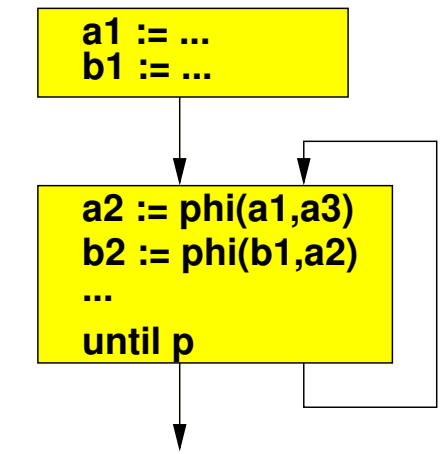




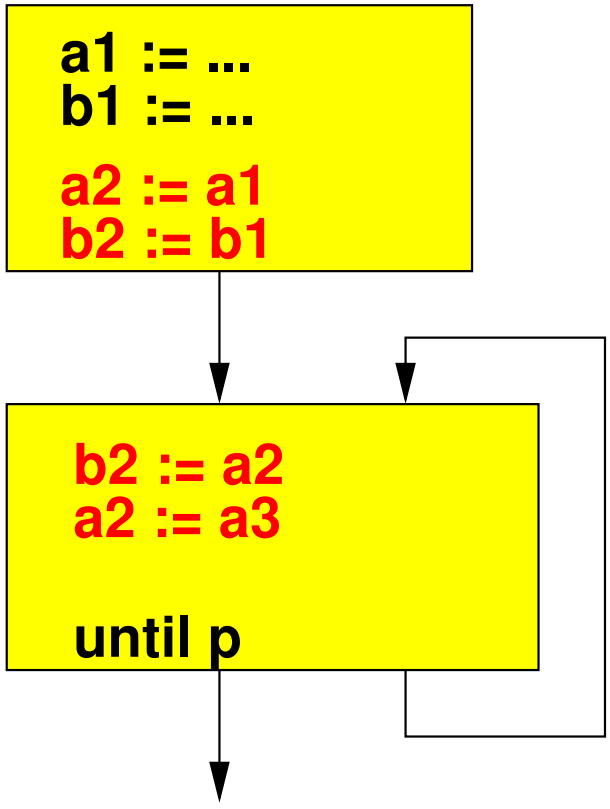
- Keine Kopie nötig, da keine zirkuläre Abhängigkeit
- Geschickte Ablaufplanung der Kopien
- Topologische Reihenfolge im Graphen

Elegantere Lösung

SSA–Form

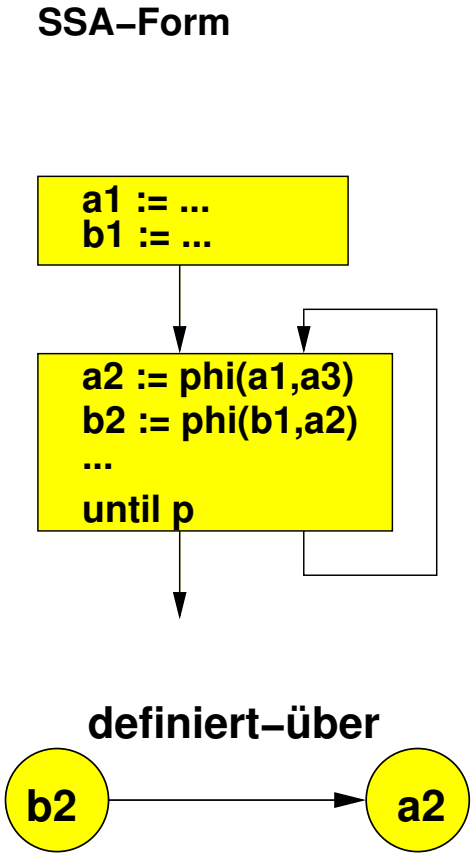


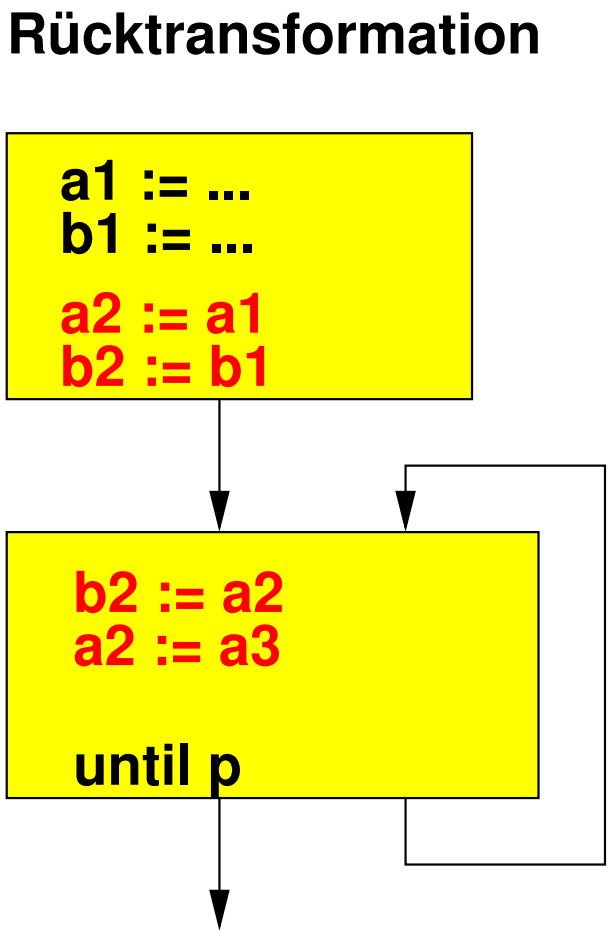
Rücktransformation



- Keine Kopie nötig, da keine zirkuläre Abhängigkeit
- Geschickte Ablaufplanung der Kopien
- Topologische Reihenfolge im Graphen

Elegantere Lösung



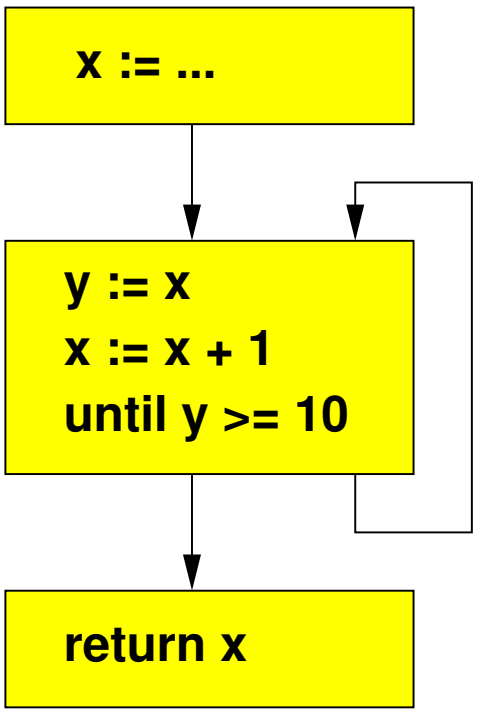


- Keine Kopie nötig, da keine zirkuläre Abhängigkeit
- Geschickte Ablaufplanung der Kopien
- Topologische Reihenfolge im Graphen

Platzierung der Phi-Funktionen

Einfaches Beispiel: CFG nach SSA

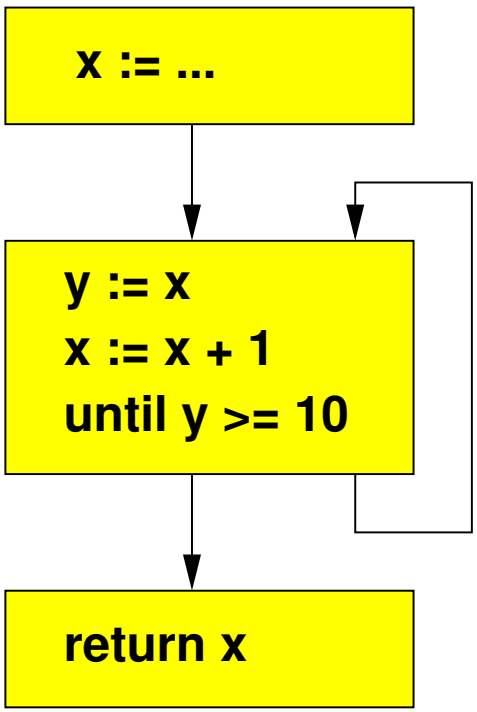
Normale Form



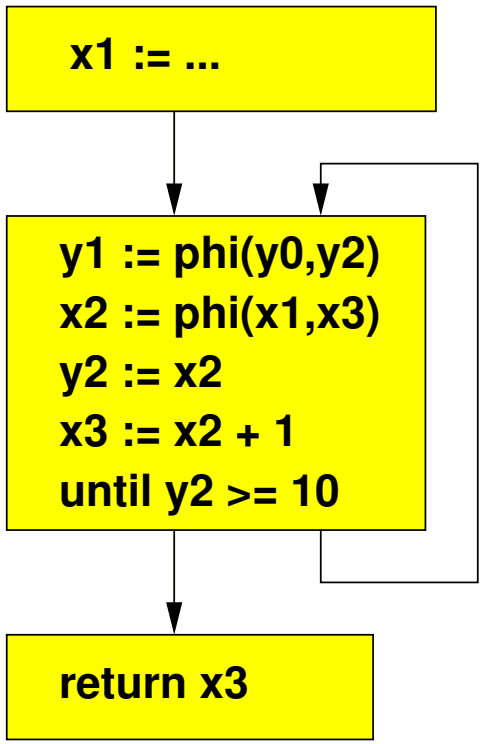
- Keine Überraschungen

Einfaches Beispiel: CFG nach SSA

Normale Form



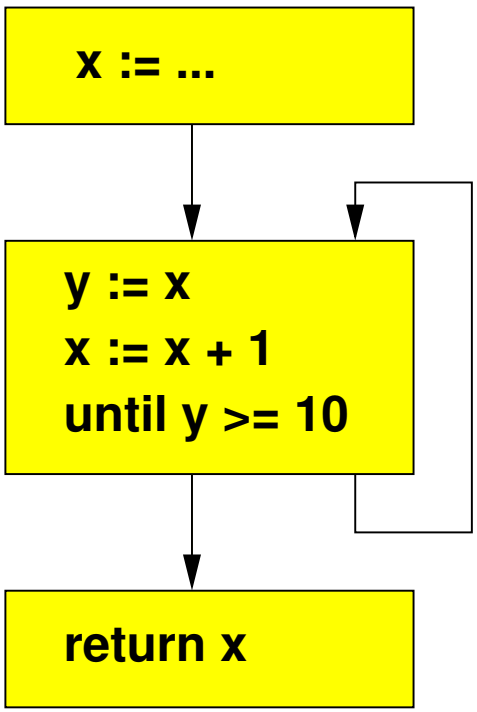
SSA–Form



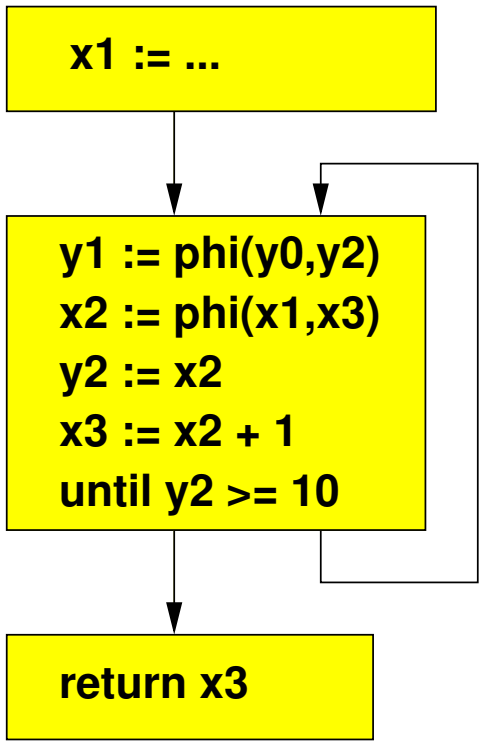
- Keine Überraschungen

Einfaches Beispiel: CFG nach SSA

Normale Form



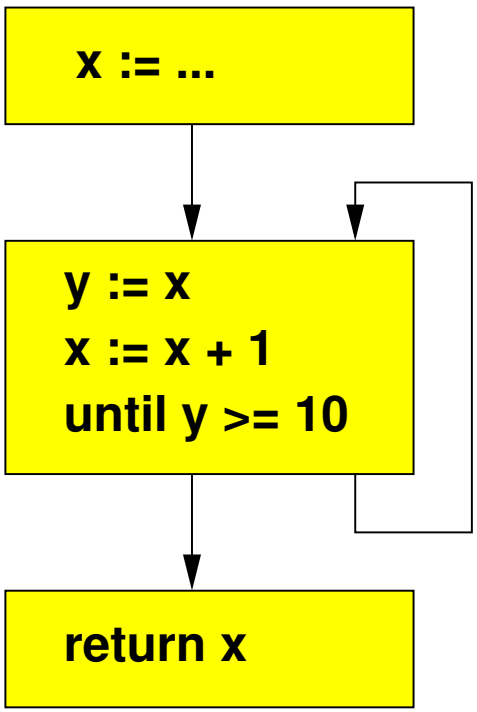
SSA–Form



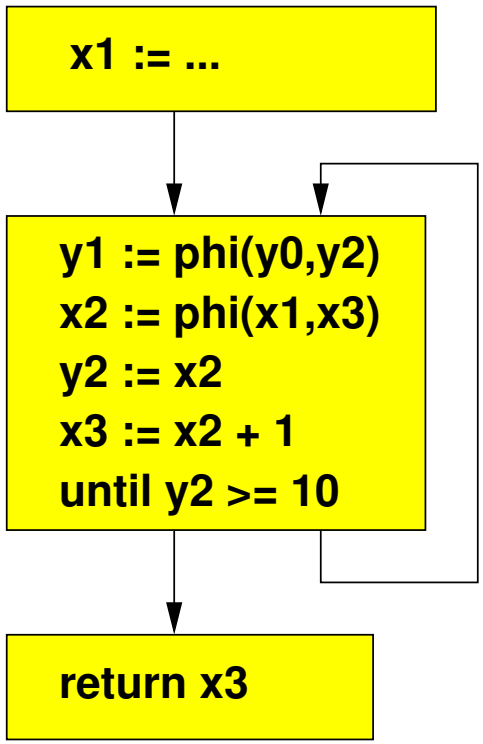
- Keine Überraschungen

Einfaches Beispiel: CFG nach SSA

Normale Form



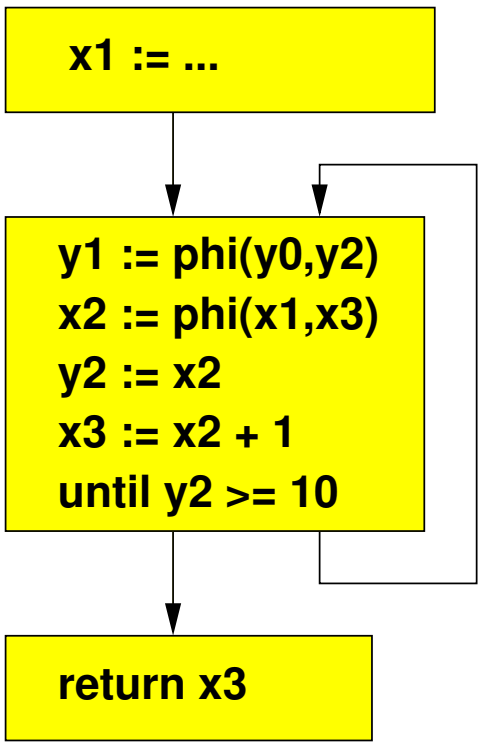
SSA–Form



- Keine Überraschungen

Copy-Propagation und Dead-Code-Elim.

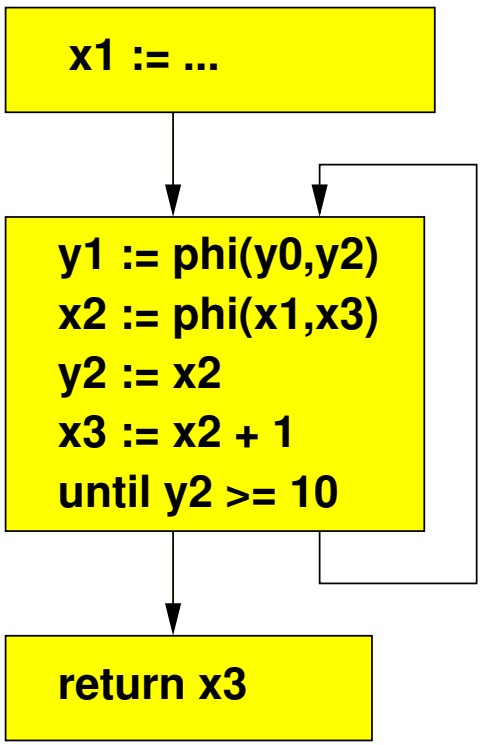
SSA–Form



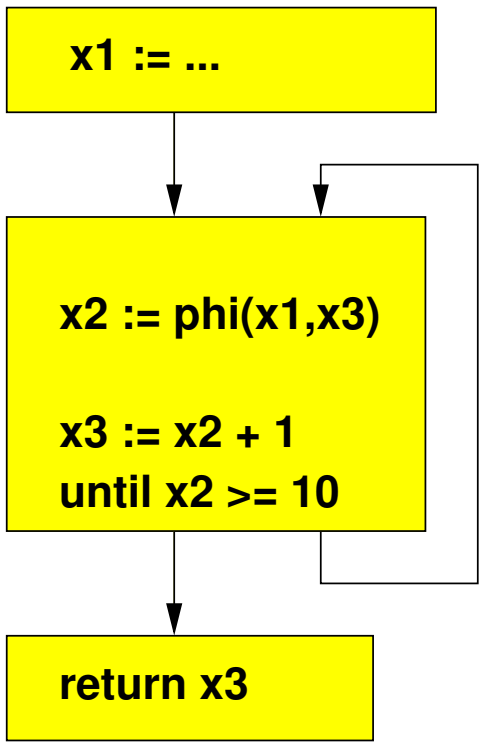
- Keine Überraschungen

Copy-Propagation und Dead-Code-Elim.

SSA–Form



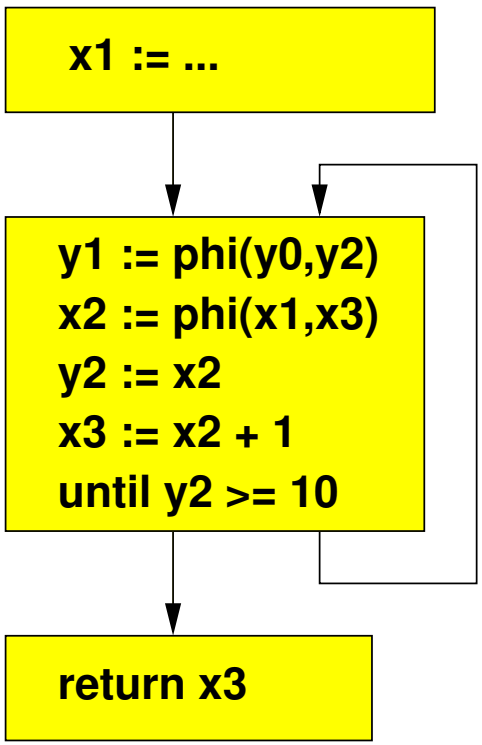
SSA–Form nach Copy–Propagation und Dead–Code Elim.



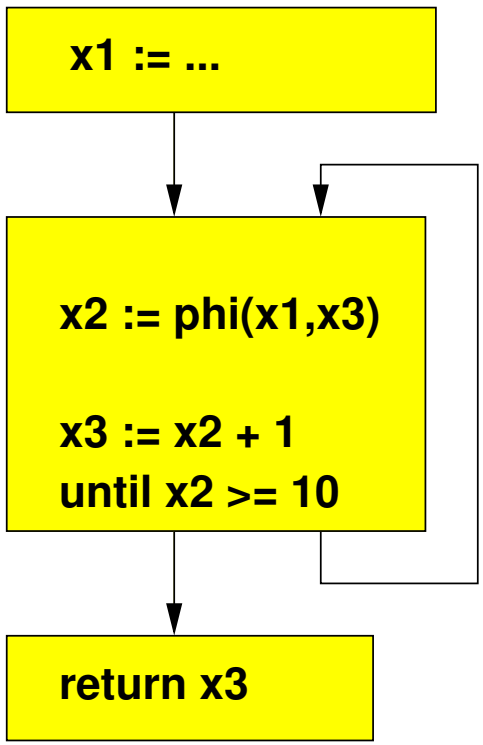
- Keine Überraschungen

Copy-Propagation und Dead-Code-Elim.

SSA–Form



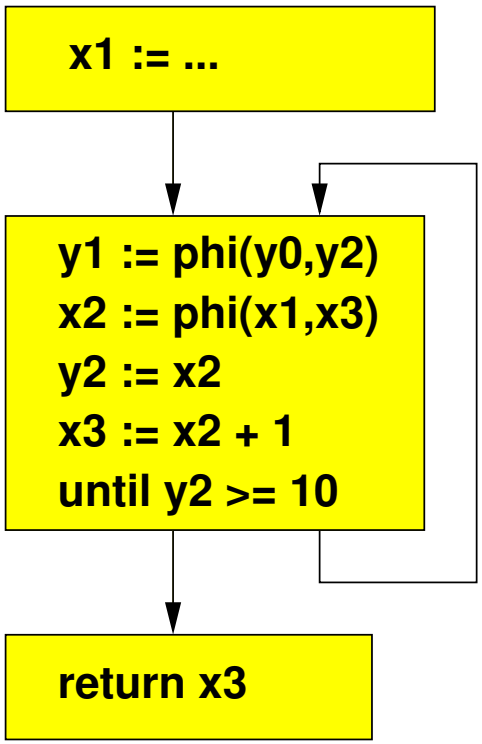
SSA–Form nach Copy–Propagation und Dead–Code Elim.



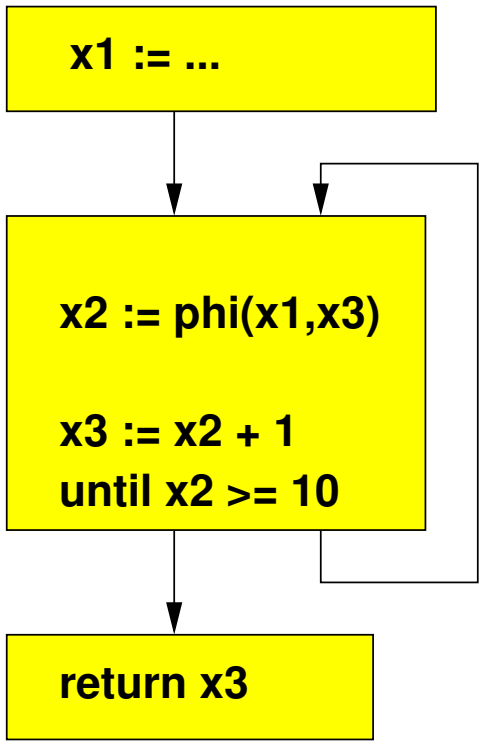
- Keine Überraschungen

Copy-Propagation und Dead-Code-Elim.

SSA–Form

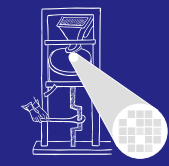


SSA–Form nach Copy–Propagation und Dead–Code Elim.

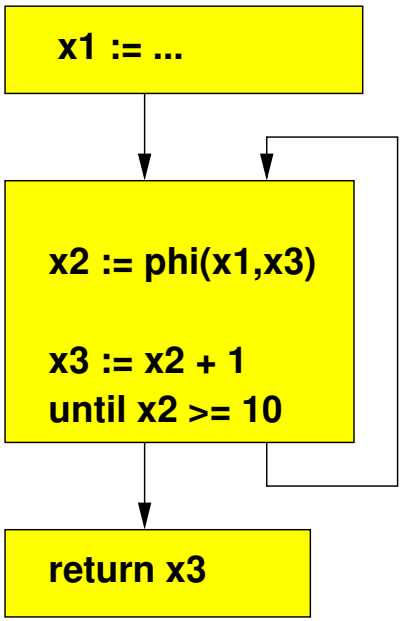


- Keine Überraschungen

Rückwandlung aus SSA-Form

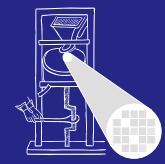


SSA–Form nach Copy–Propagation und Dead–Code Elim.

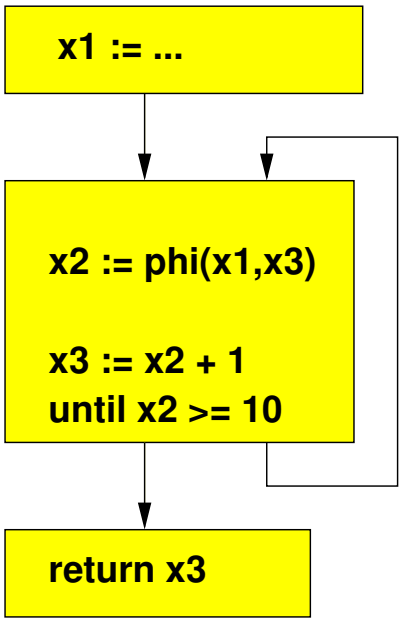


- Fehler: Wert für nächste Iteration
- ... wird schon am Ende der aktuellen Iteration verwendet
- Phi-Kopieranweisung in Lebenszeit von x2 eingefügt!

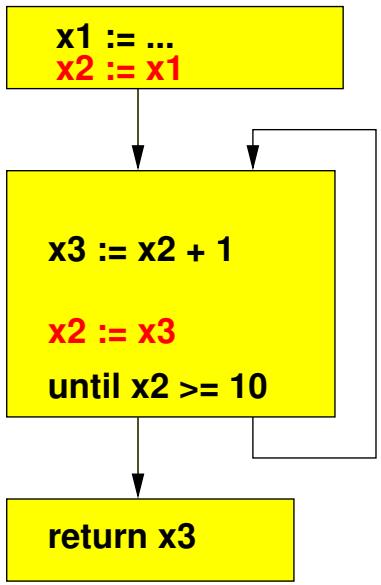
Rückwandlung aus SSA-Form



SSA–Form nach Copy–Propagation und Dead–Code Elim.

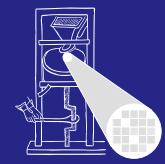


Rücktransformation aus SSA–Form nach Copy–Propagation und Dead–Code Elim.

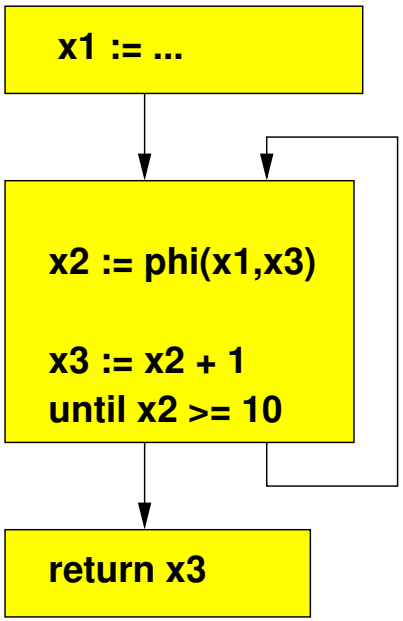


- Fehler: Wert für nächste Iteration
- ... wird schon am Ende der aktuellen Iteration verwendet
- Phi-Kopieranweisung in Lebenszeit von x2 eingefügt!

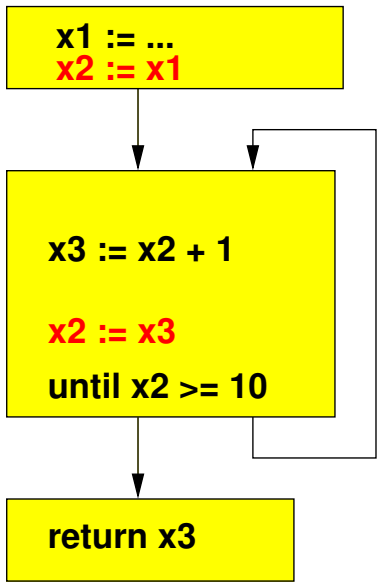
Rückwandlung aus SSA-Form



SSA–Form nach Copy–Propagation und Dead–Code Elim.



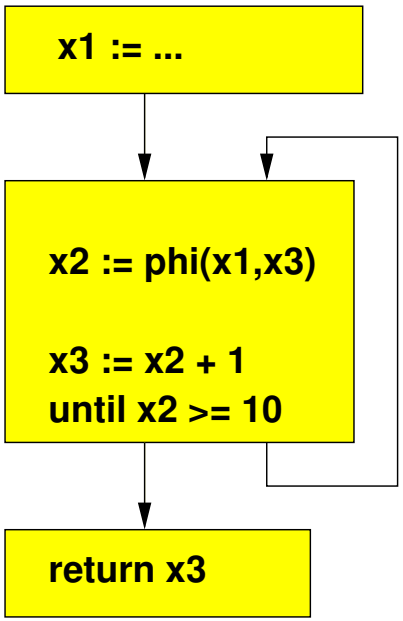
Rücktransformation aus SSA–Form nach Copy–Propagation und Dead–Code Elim.



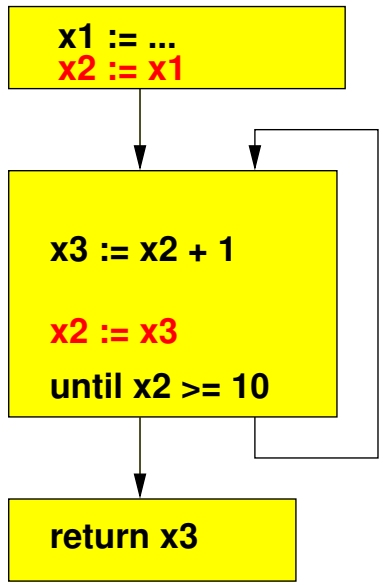
- Fehler: Wert für nächste Iteration
- ... wird schon am Ende der aktuellen Iteration verwendet
- Phi-Kopieranweisung in Lebenszeit von x2 eingefügt!

Rückwandlung aus SSA-Form

SSA–Form nach Copy–Propagation und Dead–Code Elim.



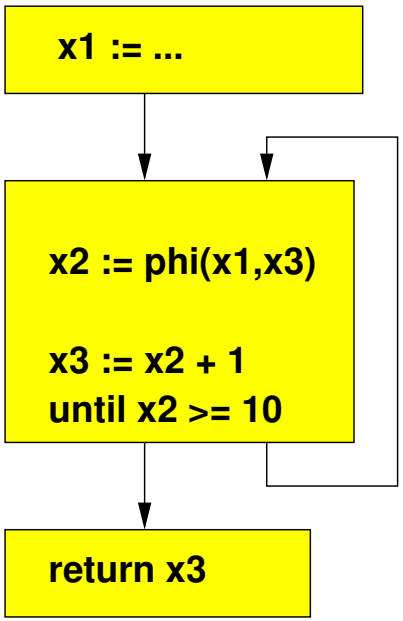
Rücktransformation aus SSA–Form nach Copy–Propagation und Dead–Code Elim.



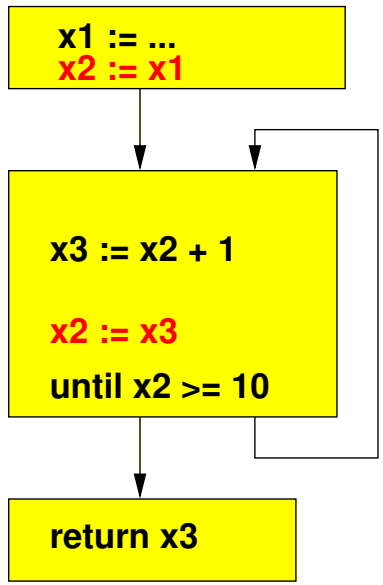
- Fehler: Wert für nächste Iteration
  - ... aus aufgelöster Phi-Funktion
- ... wird schon am Ende der aktuellen Iteration verwendet
  - ... in bedingtem Sprung am Block-Ende
- Phi-Kopieranweisung in Lebenszeit von x2 eingefügt!

Rückwandlung aus SSA-Form

SSA–Form nach Copy–Propagation und Dead–Code Elim.

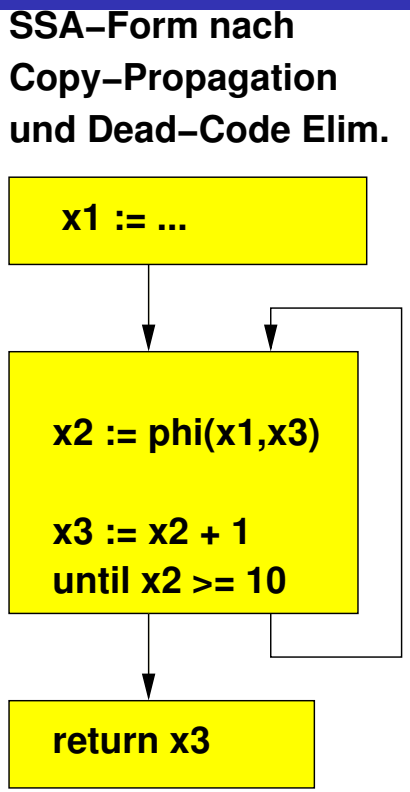


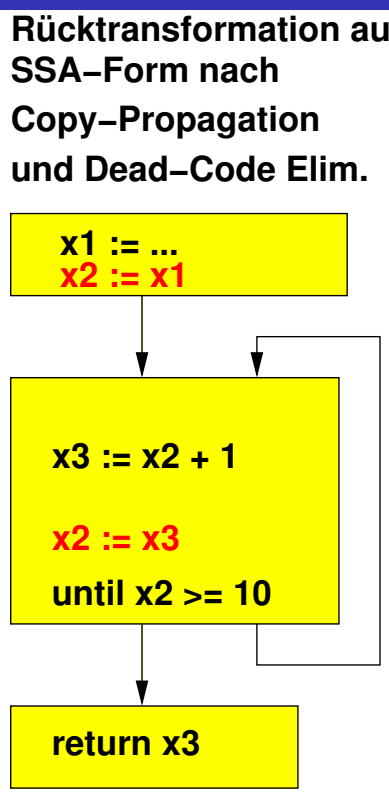
Rücktransformation aus SSA–Form nach Copy–Propagation und Dead–Code Elim.



- Fehler: Wert für nächste Iteration
  - ... aus aufgelöster Phi-Funktion
- ... wird schon am Ende der aktuellen Iteration verwendet
  - ... in bedingtem Sprung am Block-Ende
- Phi-Kopieranweisung in Lebenszeit von x2 eingefügt!

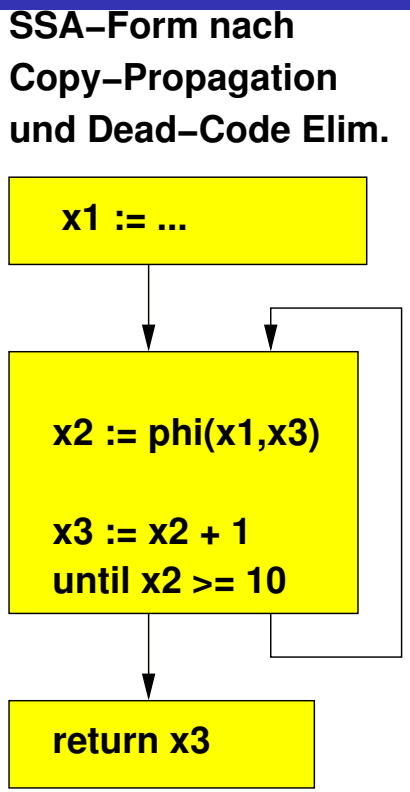
Rückwandlung aus SSA-Form

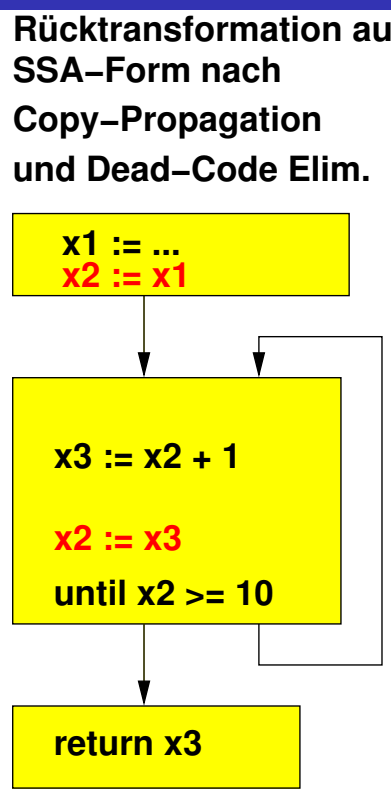




- Fehler: Wert für nächste Iteration
  - ... aus aufgelöster Phi-Funktion
- ... wird schon am Ende der aktuellen Iteration verwendet
  - ... in bedingtem Sprung am Block-Ende
- Phi-Kopieranweisung in Lebenszeit von x2 eingefügt!

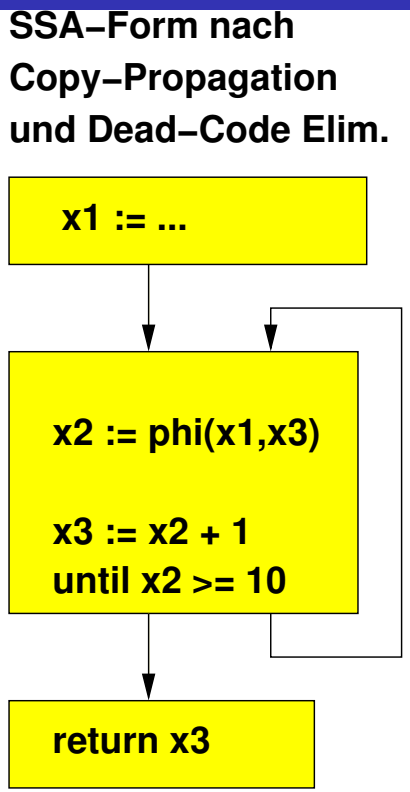
Rückwandlung aus SSA-Form

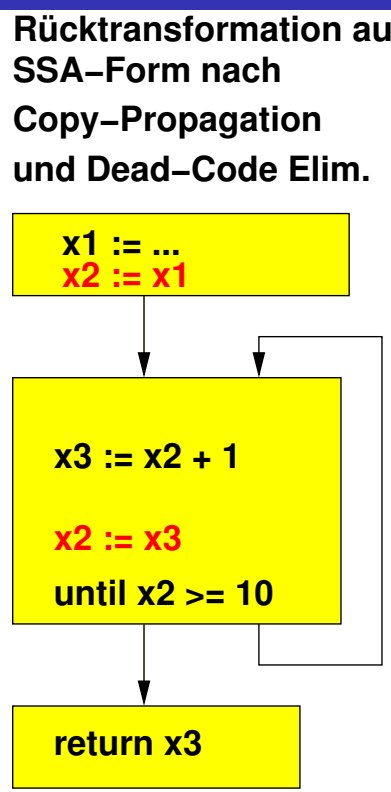




- Fehler: Wert für nächste Iteration
  - ... aus aufgelöster Phi-Funktion
- ... wird schon am Ende der aktuellen Iteration verwendet
  - ... in bedingtem Sprung am Block-Ende
- Phi-Kopieranweisung in Lebenszeit von x2 eingefügt!

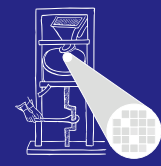
Rückwandlung aus SSA-Form



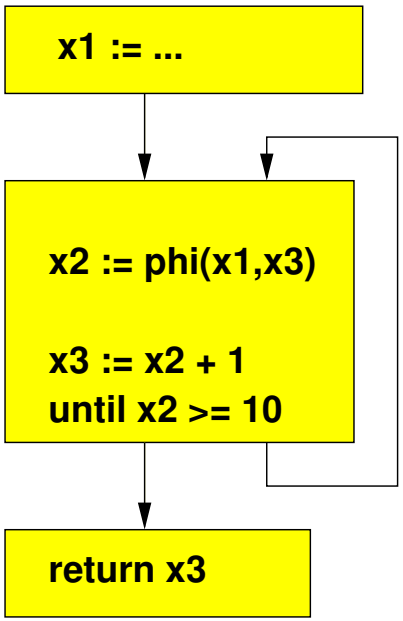


- Fehler: Wert für nächste Iteration
  - ... aus aufgelöster Phi-Funktion
- ... wird schon am Ende der aktuellen Iteration verwendet
  - ... in bedingtem Sprung am Block-Ende
- Phi-Kopieranweisung in Lebenszeit von `x2` eingefügt!

Korrekte Rückwandlung aus SSA-Form



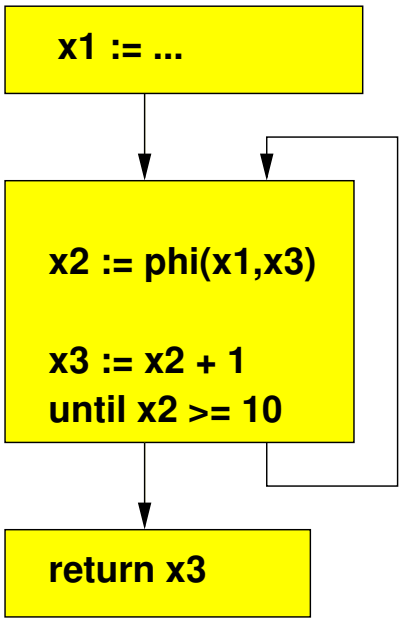
SSA–Form nach Copy–Propagation und Dead–Code Elim.



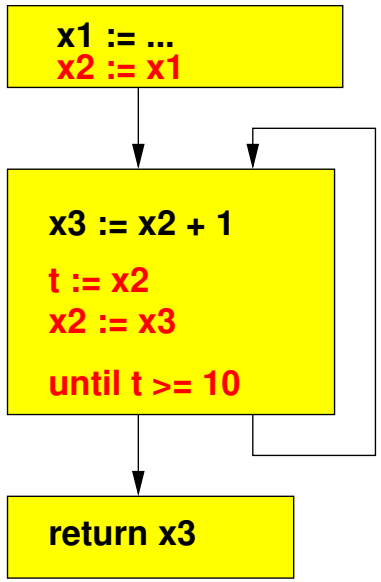
- Gleiches Vorgehen wie bei Lost-Copy
- Dafür auch bereits vorher angelegte Kopie benutzbar

Korrekte Rückwandlung aus SSA-Form

SSA–Form nach Copy–Propagation und Dead–Code Elim.



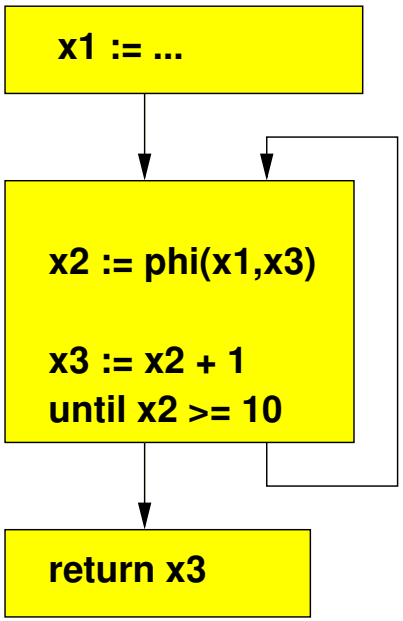
Rücktransformation aus SSA–Form nach Copy–Propagation und Dead–Code Elim.



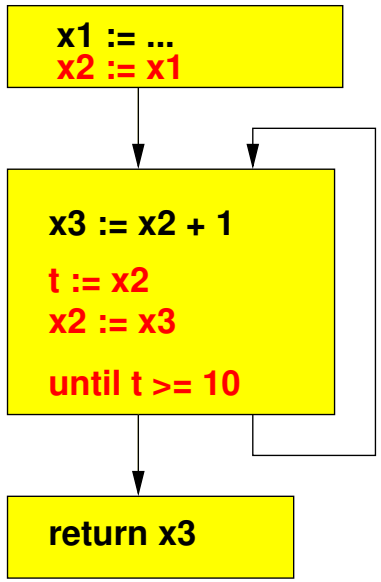
- Gleiches Vorgehen wie bei Lost-Copy
- Dafür auch bereits vorher angelegte Kopie benutzbar

Korrekte Rückwandlung aus SSA-Form

SSA–Form nach Copy–Propagation und Dead–Code Elim.

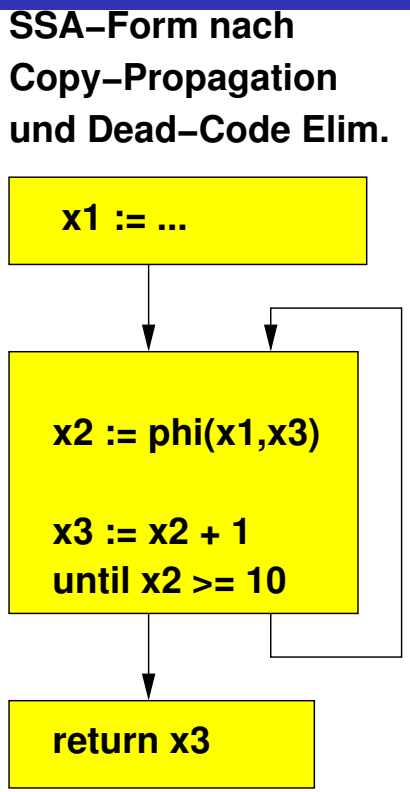


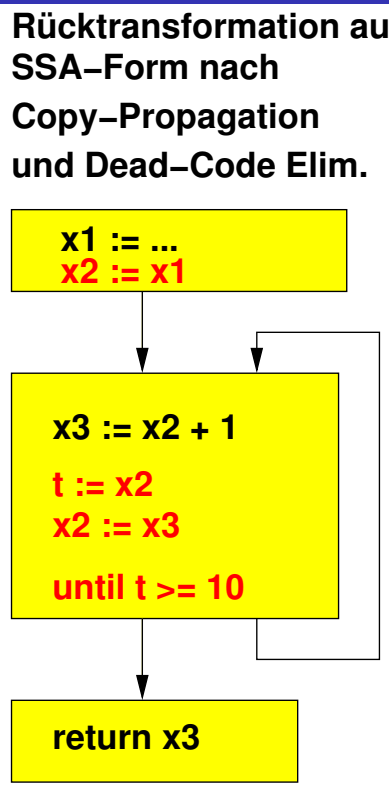
Rücktransformation aus SSA–Form nach Copy–Propagation und Dead–Code Elim.



- Gleiches Vorgehen wie bei Lost-Copy
- Dafür auch bereits vorher angelegte Kopie benutzbar

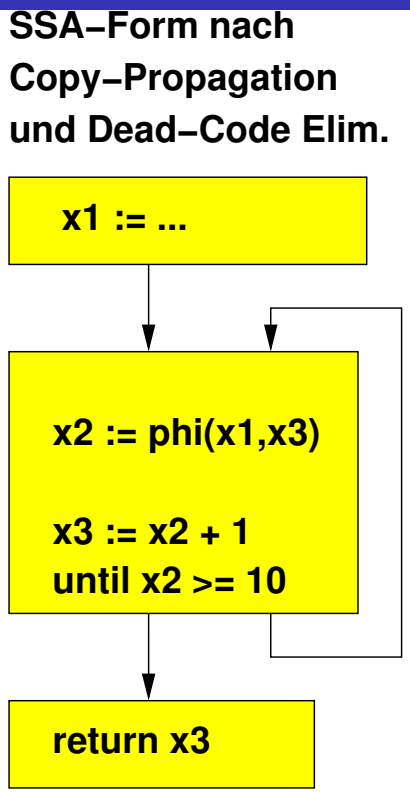
Korrekte Rückwandlung aus SSA-Form

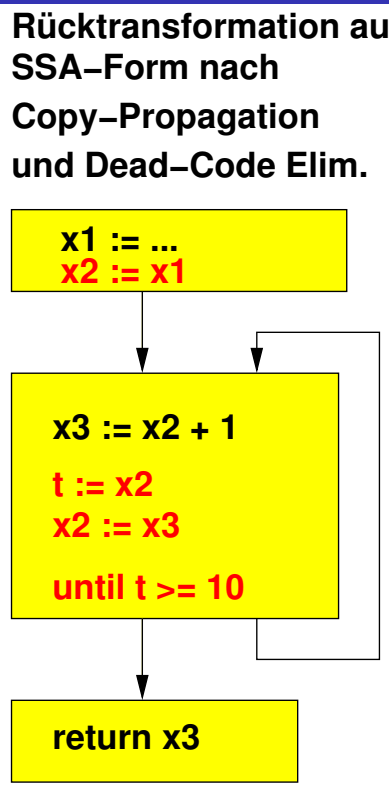




- Gleiches Vorgehen wie bei Lost-Copy
  - Sicherheitskopie von aktuellem Wert vor Überschreiben anlegen
  - Dann spätere Verwendungen durch Kopie austauschen
- Dafür auch bereits vorher angelegte Kopie benutzbar
  - Eventuell bereits wegen Inter-Block Liveness angelegt

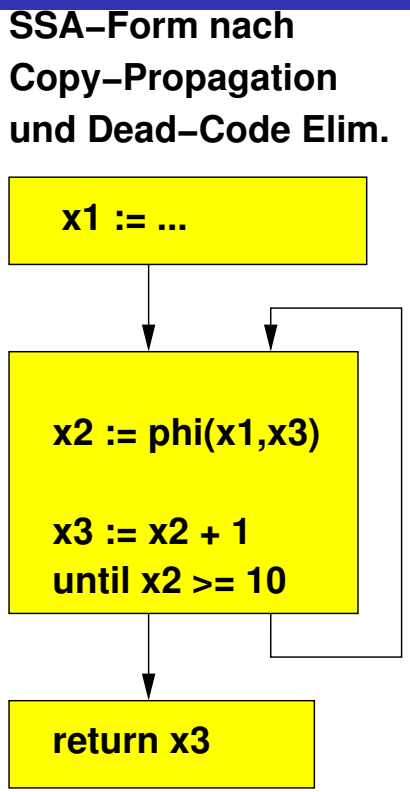
Korrekte Rückwandlung aus SSA-Form

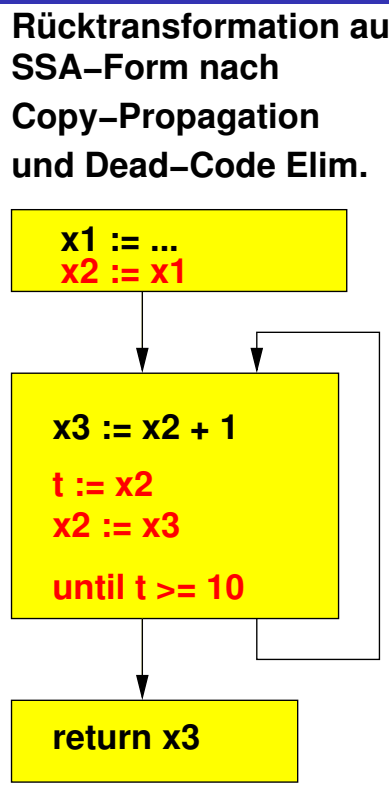




- Gleiches Vorgehen wie bei Lost-Copy
  - Sicherheitskopie von aktuellem Wert vor Überschreiben anlegen
  - Dann spätere Verwendungen durch Kopie austauschen
- Dafür auch bereits vorher angelegte Kopie benutzbar
  - Eventuell bereits wegen Inter-Block Liveness angelegt

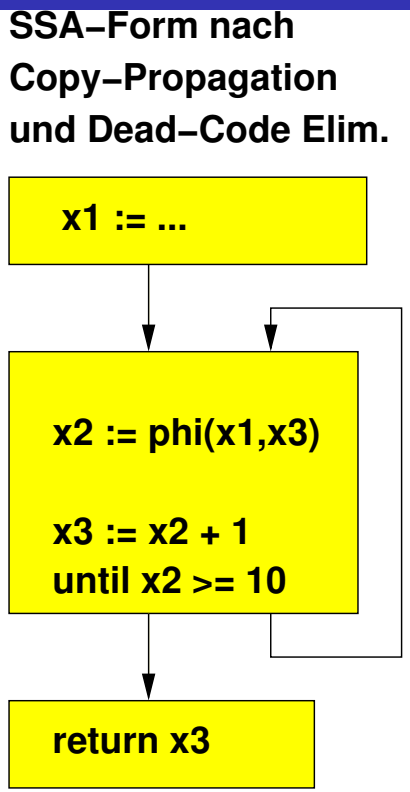
Korrekte Rückwandlung aus SSA-Form

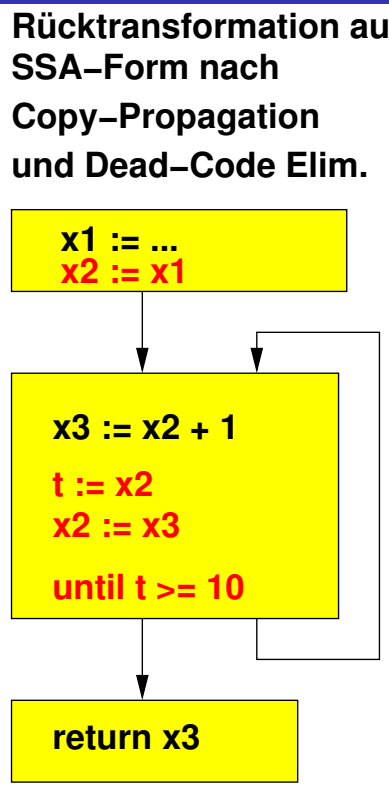




- Gleiches Vorgehen wie bei Lost-Copy
  - Sicherheitskopie von aktuellem Wert vor Überschreiben anlegen
  - Dann spätere Verwendungen durch Kopie austauschen
- Dafür auch bereits vorher angelegte Kopie benutzbar
  - Eventuell bereits wegen Inter-Block Liveness angelegt

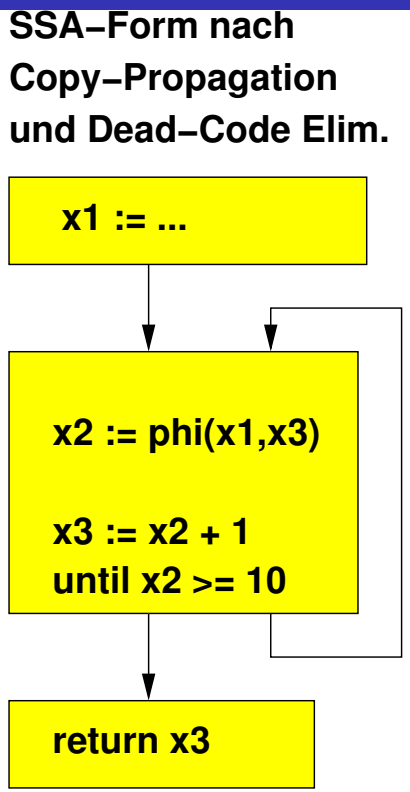
Korrekte Rückwandlung aus SSA-Form

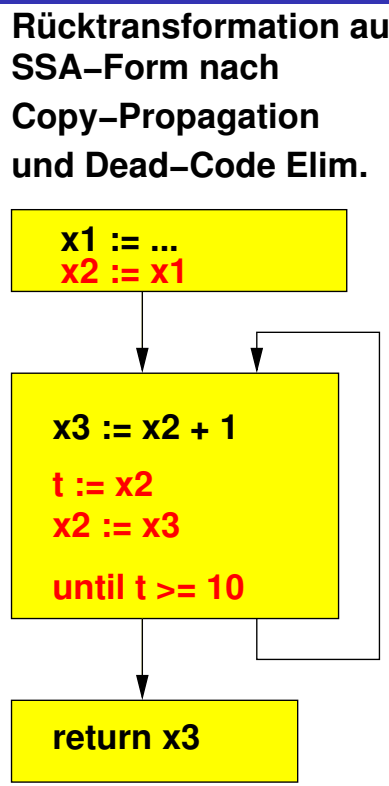




- Gleiches Vorgehen wie bei Lost-Copy
  - Sicherheitskopie von aktuellem Wert vor Überschreiben anlegen
  - Dann spätere Verwendungen durch Kopie austauschen
- Dafür auch bereits vorher angelegte Kopie benutzbar
  - Eventuell bereits wegen Inter-Block Liveness angelegt

Korrekte Rückwandlung aus SSA-Form





- Gleiches Vorgehen wie bei Lost-Copy
  - Sicherheitskopie von aktuellem Wert vor Überschreiben anlegen
  - Dann spätere Verwendungen durch Kopie austauschen
- Dafür auch bereits vorher angelegte Kopie benutzbar
  - Eventuell bereits wegen Inter-Block Liveness angelegt

Gesammelte Anforderungen an Algorithmus

- Füge Inter-Block Kopien ein, wenn Kopieranweisungen von aufgelösten Phi-Funktionen in Lebenszeit von Phi-Ergebnis liegen
- Füge lokale Kopien von Phi-Parametern ein, um zyklische Abhängigkeiten aufzubrechen
- Ordne sonstige Kopierfunktionen in richtiger Reihenfolge an
  - Sequentielle Abarbeitung muss gleiches Ergebnis wie parallele Phi-Funktionen ergeben
- Falls Ziel einer Phi-Funktion in bedingten Sprüngen am Block-Ende benutzt wird
  - Verwende dort eine eventuell bereits angelegte Inter-Block-Kopie des Ergebnisses
  - Oder lege neue lokale Kopie an und verwende diese

Gesammelte Anforderungen an Algorithmus

- Füge Inter-Block Kopien ein, wenn Kopieranweisungen von aufgelösten Phi-Funktionen in Lebenszeit von Phi-Ergebnis liegen
- Füge lokale Kopien von Phi-Parametern ein, um zyklische Abhängigkeiten aufzubrechen
- Ordne sonstige Kopierfunktionen in richtiger Reihenfolge an
  - Sequentielle Abarbeitung muss gleiches Ergebnis wie parallele Phi-Funktionen ergeben
- Falls Ziel einer Phi-Funktion in bedingten Sprüngen am Block-Ende benutzt wird
  - Verwende dort eine eventuell bereits angelegte Inter-Block-Kopie des Ergebnisses
  - Oder lege neue lokale Kopie an und verwende diese

Gesammelte Anforderungen an Algorithmus

- Füge Inter-Block Kopien ein, wenn Kopieranweisungen von aufgelösten Phi-Funktionen in Lebenszeit von Phi-Ergebnis liegen
- Füge lokale Kopien von Phi-Parametern ein, um zyklische Abhängigkeiten aufzubrechen
- Ordne sonstige Kopierfunktionen in richtiger Reihenfolge an
  - Sequentielle Abarbeitung muss gleiches Ergebnis wie parallele Phi-Funktionen ergeben
- Falls Ziel einer Phi-Funktion in bedingten Sprüngen am Block-Ende benutzt wird
  - Verwende dort eine eventuell bereits angelegte Inter-Block-Kopie des Ergebnisses
  - Oder lege neue lokale Kopie an und verwende diese

Gesammelte Anforderungen an Algorithmus

- Füge Inter-Block Kopien ein, wenn Kopieranweisungen von aufgelösten Phi-Funktionen in Lebenszeit von Phi-Ergebnis liegen
- Füge lokale Kopien von Phi-Parametern ein, um zyklische Abhängigkeiten aufzubrechen
- Ordne sonstige Kopierfunktionen in richtiger Reihenfolge an
  - Sequentielle Abarbeitung muss gleiches Ergebnis wie parallele Phi-Funktionen ergeben
- Falls Ziel einer Phi-Funktion in bedingten Sprüngen am Block-Ende benutzt wird
  - Verwende dort eine eventuell bereits angelegte Inter-Block-Kopie des Ergebnisses
  - Oder lege neue lokale Kopie an und verwende diese

Gesammelte Anforderungen an Algorithmus

- Füge Inter-Block Kopien ein, wenn Kopieranweisungen von aufgelösten Phi-Funktionen in Lebenszeit von Phi-Ergebnis liegen
- Füge lokale Kopien von Phi-Parametern ein, um zyklische Abhängigkeiten aufzubrechen
- Ordne sonstige Kopierfunktionen in richtiger Reihenfolge an
  - Sequentielle Abarbeitung muss gleiches Ergebnis wie parallele Phi-Funktionen ergeben
- Falls Ziel einer Phi-Funktion in bedingten Sprüngen am Block-Ende benutzt wird
  - Verwende dort eine eventuell bereits angelegte Inter-Block-Kopie des Ergebnisses
  - Oder lege neue lokale Kopie an und verwende diese

Gesammelte Anforderungen an Algorithmus

- Füge Inter-Block Kopien ein, wenn Kopieranweisungen von aufgelösten Phi-Funktionen in Lebenszeit von Phi-Ergebnis liegen
- Füge lokale Kopien von Phi-Parametern ein, um zyklische Abhängigkeiten aufzubrechen
- Ordne sonstige Kopierfunktionen in richtiger Reihenfolge an
  - Sequentielle Abarbeitung muss gleiches Ergebnis wie parallele Phi-Funktionen ergeben
- Falls Ziel einer Phi-Funktion in bedingten Sprüngen am Block-Ende benutzt wird
  - Verwende dort eine eventuell bereits angelegte Inter-Block-Kopie des Ergebnisses
  - Oder lege neue lokale Kopie an und verwende diese

Gesammelte Anforderungen an Algorithmus

- Füge Inter-Block Kopien ein, wenn Kopieranweisungen von aufgelösten Phi-Funktionen in Lebenszeit von Phi-Ergebnis liegen
- Füge lokale Kopien von Phi-Parametern ein, um zyklische Abhängigkeiten aufzubrechen
- Ordne sonstige Kopierfunktionen in richtiger Reihenfolge an
  - Sequentielle Abarbeitung muss gleiches Ergebnis wie parallele Phi-Funktionen ergeben
- Falls Ziel einer Phi-Funktion in bedingten Sprüngen am Block-Ende benutzt wird
  - Verwende dort eine eventuell bereits angelegte Inter-Block-Kopie des Ergebnisses
  - Oder lege neue lokale Kopie an und verwende diese

Algorithmus nach Briggs, Cooper, Harvey und Simpson

Grundlagen des Algorithmus

- Verwaltete Daten
- Durch welche Inter-Block Kopie soll eine Variable ersetzt werden?
- Durch welche lokale (Intra-Block) Kopie soll eine Variable ersetzt werden?

Grundlagen des Algorithmus

- Verwaltete Daten
- Durch welche Inter-Block Kopie soll eine Variable ersetzt werden?
- Durch welche lokale (Intra-Block) Kopie soll eine Variable ersetzt werden?

Grundlagen des Algorithmus

- Verwaltete Daten
- Durch welche Inter-Block Kopie soll eine Variable ersetzt werden?
- Durch welche lokale (Intra-Block) Kopie soll eine Variable ersetzt werden?

Inter-Block-Daten

- Algorithmus bearbeitet Blöcke in Pre-Order-Reihenfolge im Dominatorbaum
  - Bei Abstieg: Inter-block Daten aus Vorgänger übernehmen
  - Bei Aufstieg: Inter-block Daten zurücksetzen

➥Analog zu Geltungsbereichen von Symboltabellen

- Globale Hash-Map Variable → Stack: `Stacks[v]`
- Bei Anlegen einer neuen Kopie
- Bei Verlassen eines Geltungsbereichs

Inter-Block-Daten

- Algorithmus bearbeitet Blöcke in Pre-Order-Reihenfolge im Dominatorbaum
  - Bei Abstieg: Inter-block Daten aus Vorgänger übernehmen
  - Bei Aufstieg: Inter-block Daten zurücksetzen

➥Analog zu Geltungsbereichen von Symboltabellen

- Globale Hash-Map Variable → Stack: `Stacks[v]`
- Bei Anlegen einer neuen Kopie
- Bei Verlassen eines Geltungsbereichs

Inter-Block-Daten

- Algorithmus bearbeitet Blöcke in Pre-Order-Reihenfolge im Dominatorbaum
  - Bei Abstieg: Inter-block Daten aus Vorgänger übernehmen
  - Bei Aufstieg: Inter-block Daten zurücksetzen

➥Analog zu Geltungsbereichen von Symboltabellen

- Globale Hash-Map Variable → Stack: `Stacks[v]`
- Bei Anlegen einer neuen Kopie
- Bei Verlassen eines Geltungsbereichs

Inter-Block-Daten

- Algorithmus bearbeitet Blöcke in Pre-Order-Reihenfolge im Dominatorbaum
  - Bei Abstieg: Inter-block Daten aus Vorgänger übernehmen
  - Bei Aufstieg: Inter-block Daten zurücksetzen

➥Analog zu Geltungsbereichen von Symboltabellen

- Globale Hash-Map Variable → Stack: `Stacks[v]`
- Bei Anlegen einer neuen Kopie
- Bei Verlassen eines Geltungsbereichs

Inter-Block-Daten

- Algorithmus bearbeitet Blöcke in Pre-Order-Reihenfolge im Dominatorbaum
  - Bei Abstieg: Inter-block Daten aus Vorgänger übernehmen
  - Bei Aufstieg: Inter-block Daten zurücksetzen

➥Analog zu Geltungsbereichen von Symboltabellen

- Globale Hash-Map Variable → Stack: `Stacks[v]`
- Bei Anlegen einer neuen Kopie
  - Push des Ziels auf Stack von Ursprungsvariable
  - Spätere Verwendungen von Variable durch letztes Ziel ersetzen
- Bei Verlassen eines Geltungsbereichs
  - Alle in diesem Block gemachten Einträge entfernen
  - Ursprungsvariablen lokal je Block in `pushed` merken

Inter-Block-Daten

- Algorithmus bearbeitet Blöcke in Pre-Order-Reihenfolge im Dominatorbaum
  - Bei Abstieg: Inter-block Daten aus Vorgänger übernehmen
  - Bei Aufstieg: Inter-block Daten zurücksetzen

➥Analog zu Geltungsbereichen von Symboltabellen

- Globale Hash-Map Variable → Stack: `Stacks[v]`
- Bei Anlegen einer neuen Kopie
  - Push des Ziels auf Stack von Ursprungsvariable
  - Spätere Verwendungen von Variable durch letztes Ziel ersetzen
- Bei Verlassen eines Geltungsbereichs
  - Alle in diesem Block gemachten Einträge entfernen
  - Ursprungsvariablen lokal je Block in `pushed` merken

Inter-Block-Daten

- Algorithmus bearbeitet Blöcke in Pre-Order-Reihenfolge im Dominatorbaum
  - Bei Abstieg: Inter-block Daten aus Vorgänger übernehmen
  - Bei Aufstieg: Inter-block Daten zurücksetzen

➥Analog zu Geltungsbereichen von Symboltabellen

- Globale Hash-Map Variable → Stack: `Stacks[v]`
- Bei Anlegen einer neuen Kopie
  - Push des Ziels auf Stack von Ursprungsvariable
  - Spätere Verwendungen von Variable durch letztes Ziel ersetzen
- Bei Verlassen eines Geltungsbereichs
  - Alle in diesem Block gemachten Einträge entfernen
  - Ursprungsvariablen lokal je Block in `pushed` merken

Inter-Block-Daten

- Algorithmus bearbeitet Blöcke in Pre-Order-Reihenfolge im Dominatorbaum
  - Bei Abstieg: Inter-block Daten aus Vorgänger übernehmen
  - Bei Aufstieg: Inter-block Daten zurücksetzen

➥Analog zu Geltungsbereichen von Symboltabellen

- Globale Hash-Map Variable → Stack: `Stacks[v]`
- Bei Anlegen einer neuen Kopie
  - Push des Ziels auf Stack von Ursprungsvariable
  - Spätere Verwendungen von Variable durch letztes Ziel ersetzen
- Bei Verlassen eines Geltungsbereichs
  - Alle in diesem Block gemachten Einträge entfernen
  - Ursprungsvariablen lokal je Block in `pushed` merken

Inter-Block-Daten

- Algorithmus bearbeitet Blöcke in Pre-Order-Reihenfolge im Dominatorbaum
  - Bei Abstieg: Inter-block Daten aus Vorgänger übernehmen
  - Bei Aufstieg: Inter-block Daten zurücksetzen

➥Analog zu Geltungsbereichen von Symboltabellen

- Globale Hash-Map Variable → Stack: `Stacks[v]`
- Bei Anlegen einer neuen Kopie
  - Push des Ziels auf Stack von Ursprungsvariable
  - Spätere Verwendungen von Variable durch letztes Ziel ersetzen
- Bei Verlassen eines Geltungsbereichs
  - Alle in diesem Block gemachten Einträge entfernen
  - Ursprungsvariablen lokal je Block in `pushed` merken

Inter-Block-Daten

- Algorithmus bearbeitet Blöcke in Pre-Order-Reihenfolge im Dominatorbaum
  - Bei Abstieg: Inter-block Daten aus Vorgänger übernehmen
  - Bei Aufstieg: Inter-block Daten zurücksetzen

➥Analog zu Geltungsbereichen von Symboltabellen

- Globale Hash-Map Variable → Stack: `Stacks[v]`
- Bei Anlegen einer neuen Kopie
  - Push des Ziels auf Stack von Ursprungsvariable
  - Spätere Verwendungen von Variable durch letztes Ziel ersetzen
- Bei Verlassen eines Geltungsbereichs
  - Alle in diesem Block gemachten Einträge entfernen
  - Ursprungsvariablen lokal je Block in `pushed` merken

Inter-Block-Daten

- Algorithmus bearbeitet Blöcke in Pre-Order-Reihenfolge im Dominatorbaum
  - Bei Abstieg: Inter-block Daten aus Vorgänger übernehmen
  - Bei Aufstieg: Inter-block Daten zurücksetzen

➥Analog zu Geltungsbereichen von Symboltabellen

- Globale Hash-Map Variable → Stack: `Stacks[v]`
- Bei Anlegen einer neuen Kopie
  - Push des Ziels auf Stack von Ursprungsvariable
  - Spätere Verwendungen von Variable durch letztes Ziel ersetzen
- Bei Verlassen eines Geltungsbereichs
  - Alle in diesem Block gemachten Einträge entfernen
  - Ursprungsvariablen lokal je Block in `pushed` merken

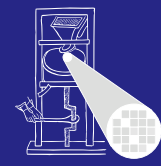
Lokale Intra-Block Daten

- Einfachere Struktur: Je Block
- Lokale Hash-Map von Ursprungsvariable auf Zielvariable: `Map[v]`
- Spätere Verwendungen von Variable durch aktuelle Kopie ersetzen
- Flag, ob Variable als Phi-Parameter verwendet wird: `IsPhiParam[v]`
  - Dann geschickte Ablaufplanung erforderlich
  - Oder sogar Aufbrechen von Zyklus

Lokale Intra-Block Daten

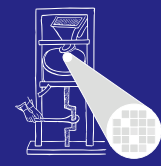
- Einfachere Struktur: Je Block
- Lokale Hash-Map von Ursprungsvariable auf Zielvariable: `Map[v]`
- Spätere Verwendungen von Variable durch aktuelle Kopie ersetzen
- Flag, ob Variable als Phi-Parameter verwendet wird: `IsPhiParam[v]`
  - Dann geschickte Ablaufplanung erforderlich
  - Oder sogar Aufbrechen von Zyklus

Lokale Intra-Block Daten



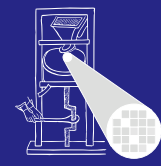
- Einfachere Struktur: Je Block
- Lokale Hash-Map von Ursprungsvariable auf Zielvariable: `Map[v]`
- Spätere Verwendungen von Variable durch aktuelle Kopie ersetzen
- Flag, ob Variable als Phi-Parameter verwendet wird: `IsPhiParam[v]`
  - Dann geschickte Ablaufplanung erforderlich
  - Oder sogar Aufbrechen von Zyklus

Lokale Intra-Block Daten



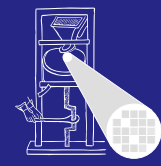
- Einfachere Struktur: Je Block
- Lokale Hash-Map von Ursprungsvariable auf Zielvariable: `Map[v]`
- Spätere Verwendungen von Variable durch aktuelle Kopie ersetzen
- Flag, ob Variable als Phi-Parameter verwendet wird: `IsPhiParam[v]`
  - Dann geschickte Ablaufplanung erforderlich
  - Oder sogar Aufbrechen von Zyklus

Lokale Intra-Block Daten



- Einfachere Struktur: Je Block
- Lokale Hash-Map von Ursprungsvariable auf Zielvariable: `Map[v]`
- Spätere Verwendungen von Variable durch aktuelle Kopie ersetzen
- Flag, ob Variable als Phi-Parameter verwendet wird: `IsPhiParam[v]`
  - Dann geschickte Ablaufplanung erforderlich
  - Oder sogar Aufbrechen von Zyklus

Lokale Intra-Block Daten



- Einfachere Struktur: Je Block
- Lokale Hash-Map von Ursprungsvariable auf Zielvariable: `Map[v]`
- Spätere Verwendungen von Variable durch aktuelle Kopie ersetzen
- Flag, ob Variable als Phi-Parameter verwendet wird: `IsPhiParam[v]`
  - Dann geschickte Ablaufplanung erforderlich
  - Oder sogar Aufbrechen von Zyklus

Hauptprozedur: `replace_phi_nodes(cfg)`

Bearbeitet gesamten CFG.

Bestimme Live-Variablen durch Datenflussanalyse
for alle Variablen v im CFG **do**
    `Stacks`$[v] \leftarrow \emptyset$
end for
`insert_copies(cfg.entryBlock())`
alle Phi-Funktionen "$x \leftarrow \Phi(\ldots)$" im ganzen CFG löschen

insert_copies(block)

Bearbeitet rekursiv einen Block und die von ihm dominierten Blöcke in Pre-Order des Dominatorbaums.

```
// Hier noch keine lokalen Inter-Block Sicherheitskopien angelegt
Pushed ← ∅
// Umbenennen auf bisherige Inter-Block Sicherheitskopien
for alle Anweisungen i in block do
    for alle Variablen v in i do
        ersetze alle v durch Stacks[v].top, wenn dies ≠ nil
    end for
end for
// einzelnen Block bearbeiten, hier findet die Hauptarbeit statt
schedule_copies(block)
// Kinder rekursiv bearbeiten
for k ist Kind vom block im Dominator-Baum do
    insert_copies(k)
end for
// in diesem Block angelegte Inter-Block Kopien verwerfen
for Variable v in Pushed do
    pop(Stacks[v])
end for
```

schedule_copies(block)

1. Pass: Initialisierung

```
// Sammele in diesem Block benötigte Kopieranweisungen, anfangs leer
Copyset ← ∅
// Meine Kopieranweisungen lösen die Phi-Funktionen meiner Nachfolger auf
for alle Nachfolger s von block do
    // bestimme welcher Vorgänger j der block für s ist
    j ← whichPred(s, block)
    // bearbeite alle Phi-Funktionen im Nachfolger
    for alle Phi-Funktionen "dest := Φ(...)" in s do
        src ← j-ter Operand der Phi-Funktion
        // diese Kopieranweisung werde ich brauchen
        Copyset ← Copyset ∪ {(src, dest)}
        // bisher noch keine lokalen Sicherheitskopien, Identitätsabbildung
        Map[src] ← src
        Map[dest] ← dest
        // src wurde als Parameter einer Phi-Funktion genutzt, später Vorsicht!
        IsPhiParam[src] ← true
    end for
end for
```

schedule_copies(block)

2. Pass: Abarbeitungsreihenfolge bestimmen

- Bestimme Worklist so, dass zunächst nur konfliktfreie Kopien erzeugt werden
- Also nur solche, deren Ziel nicht als Parameter einer anderen Phi-Funktion verwendet wird.
  - Dann ist die Reihenfolge der Kopieranweisungen egal
- Sonst besteht Gefahr von Vertauschungsproblem!

```
// gehe alle in diesem Block zu erzeugenden Kopien durch
for alle Kopien (src, dest) ∈ Copyset do
    if ¬IsPhiParam[dest] then
        Worklist ← Worklist ∪ (src, dest)
        Copyset ← Copyset − {(src, dest)}
    end if
end for
// Worklist enthält jetzt nur die einfachen Fälle
```

schedule_copies(block)

2. Pass: Abarbeitungsreihenfolge bestimmen

- Bestimme Worklist so, dass zunächst nur konfliktfreie Kopien erzeugt werden
- Also nur solche, deren Ziel nicht als Parameter einer anderen Phi-Funktion verwendet wird.
  - Dann ist die Reihenfolge der Kopieranweisungen egal
- Sonst besteht Gefahr von Vertauschungsproblem!

```
// gehe alle in diesem Block zu erzeugenden Kopien durch
for alle Kopien (src, dest) ∈ Copyset do
    if ¬IsPhiParam[dest] then
        Worklist ← Worklist ∪ (src, dest)
        Copyset ← Copyset − {(src, dest)}
    end if
end for
// Worklist enthält jetzt nur die einfachen Fälle
```

schedule_copies(block)

2. Pass: Abarbeitungsreihenfolge bestimmen

- Bestimme Worklist so, dass zunächst nur konfliktfreie Kopien erzeugt werden
- Also nur solche, deren Ziel nicht als Parameter einer anderen Phi-Funktion verwendet wird.
  - Dann ist die Reihenfolge der Kopieranweisungen egal
- Sonst besteht Gefahr von Vertauschungsproblem!

```
// gehe alle in diesem Block zu erzeugenden Kopien durch
for alle Kopien (src, dest) ∈ Copyset do
    if ¬IsPhiParam[dest] then
        Worklist ← Worklist ∪ (src, dest)
        Copyset ← Copyset − {(src, dest)}
    end if
end for
// Worklist enthält jetzt nur die einfachen Fälle
```

schedule_copies(block)

2. Pass: Abarbeitungsreihenfolge bestimmen

- Bestimme Worklist so, dass zunächst nur konfliktfreie Kopien erzeugt werden
- Also nur solche, deren Ziel nicht als Parameter einer anderen Phi-Funktion verwendet wird.
  - Dann ist die Reihenfolge der Kopieranweisungen egal
- Sonst besteht Gefahr von Vertauschungsproblem!

```
// gehe alle in diesem Block zu erzeugenden Kopien durch
for alle Kopien (src, dest) ∈ Copyset do
    if ¬IsPhiParam[dest] then
        Worklist ← Worklist ∪ (src, dest)
        Copyset ← Copyset − {(src, dest)}
    end if
end for
// Worklist enthält jetzt nur die einfachen Fälle
```

schedule_copies(block)

2. Pass: Abarbeitungsreihenfolge bestimmen

- Bestimme Worklist so, dass zunächst nur konfliktfreie Kopien erzeugt werden
- Also nur solche, deren Ziel nicht als Parameter einer anderen Phi-Funktion verwendet wird.
  - Dann ist die Reihenfolge der Kopieranweisungen egal
- Sonst besteht Gefahr von Vertauschungsproblem!

```
// gehe alle in diesem Block zu erzeugenden Kopien durch
for alle Kopien (src, dest) ∈ Copyset do
    if ¬IsPhiParam[dest] then
        Worklist ← Worklist ∪ (src, dest)
        Copyset ← Copyset − {(src, dest)}
    end if
end for
// Worklist enthält jetzt nur die einfachen Fälle
```

schedule_copies(block)

2. Pass: Abarbeitungsreihenfolge bestimmen

- Bestimme Worklist so, dass zunächst nur konfliktfreie Kopien erzeugt werden
- Also nur solche, deren Ziel nicht als Parameter einer anderen Phi-Funktion verwendet wird.
  - Dann ist die Reihenfolge der Kopieranweisungen egal
- Sonst besteht Gefahr von Vertauschungsproblem!

```
// gehe alle in diesem Block zu erzeugenden Kopien durch
for alle Kopien (src, dest) ∈ Copyset do
    if ¬IsPhiParam[dest] then
        Worklist ← Worklist ∪ (src, dest)
        Copyset ← Copyset − {(src, dest)}
    end if
end for
// Worklist enthält jetzt nur die einfachen Fälle
```

schedule_copies(block)

3. Pass: Benötigte Kopien erzeugen

- Es müssen alle Kopien abgearbeitet werden
- Erstmal die “einfachen” aus der Worklist
- Später auch die komplizierteren, die noch in *Copyset* stehen

```
while Worklist ≠ ∅ ∨ Copyset ≠ ∅ do
    // Bearbeite erst einfache Fälle, kann auch komplizierte vereinfachen
    while Worklist ≠ ∅ do
        ➥einfacher Fall, Pass 3a, Teil 1+2
    end while
    // Nun die komplizierten Fälle, erzeugt wiederum neue einfache Fälle
    if Copyset ≠ ∅ then
        // zirkuläre Abhängigkeit, muss durch Kopieren aufgebrochen werden
        ➥komplizierter Fall, Pass 3b
    end if
end while
```

schedule_copies(block)

3. Pass: Benötigte Kopien erzeugen

- Es müssen alle Kopien abgearbeitet werden
- Erstmal die “einfachen” aus der Worklist
- Später auch die komplizierteren, die noch in *Copyset* stehen

```
while Worklist ≠ ∅ ∨ Copyset ≠ ∅ do
    // Bearbeite erst einfache Fälle, kann auch komplizierte vereinfachen
    while Worklist ≠ ∅ do
        ➥einfacher Fall, Pass 3a, Teil 1+2
    end while
    // Nun die komplizierten Fälle, erzeugt wiederum neue einfache Fälle
    if Copyset ≠ ∅ then
        // zirkuläre Abhängigkeit, muss durch Kopieren aufgebrochen werden
        ➥komplizierter Fall, Pass 3b
    end if
end while
```

schedule_copies(block)

3. Pass: Benötigte Kopien erzeugen

- Es müssen alle Kopien abgearbeitet werden
- Erstmal die “einfachen” aus der Worklist
- Später auch die komplizierteren, die noch in *Copyset* stehen

```
while Worklist ≠ ∅ ∨ Copyset ≠ ∅ do
    // Bearbeite erst einfache Fälle, kann auch komplizierte vereinfachen
    while Worklist ≠ ∅ do
        ➥einfacher Fall, Pass 3a, Teil 1+2
    end while
    // Nun die komplizierten Fälle, erzeugt wiederum neue einfache Fälle
    if Copyset ≠ ∅ then
        // zirkuläre Abhängigkeit, muss durch Kopieren aufgebrochen werden
        ➥komplizierter Fall, Pass 3b
    end if
end while
```

schedule_copies(block)

3. Pass: Benötigte Kopien erzeugen

- Es müssen alle Kopien abgearbeitet werden
- Erstmal die “einfachen” aus der Worklist
- Später auch die komplizierteren, die noch in *Copyset* stehen

```
while Worklist ≠ ∅ ∨ Copyset ≠ ∅ do
    // Bearbeite erst einfache Fälle, kann auch komplizierte vereinfachen
    while Worklist ≠ ∅ do
        ➥einfacher Fall, Pass 3a, Teil 1+2
    end while
    // Nun die komplizierten Fälle, erzeugt wiederum neue einfache Fälle
    if Copyset ≠ ∅ then
        // zirkuläre Abhängigkeit, muss durch Kopieren aufgebrochen werden
        ➥komplizierter Fall, Pass 3b
    end if
end while
```

schedule_copies(block)

3. Pass: Benötigte Kopien erzeugen

- Es müssen alle Kopien abgearbeitet werden
- Erstmal die “einfachen” aus der Worklist
- Später auch die komplizierteren, die noch in *Copyset* stehen

while *Worklist* $\neq \emptyset \vee$ *Copyset* $\neq \emptyset$ **do**
    // Bearbeite erst einfache Fälle, kann auch komplizierte vereinfachen
    while *Worklist* $\neq \emptyset$ **do**
        ➥einfacher Fall, Pass 3a, Teil 1+2
    end while
    // Nun die komplizierten Fälle, erzeugt wiederum neue einfache Fälle
    if *Copyset* $\neq \emptyset$ **then**
        // zirkuläre Abhängigkeit, muss durch Kopieren aufgebrochen werden
        ➥komplizierter Fall, Pass 3b
    end if
end while

schedule_copies(block)

3a. Pass: Erzeuge Kopien für einfache Fälle aus *Worklist*, Teil 1

- Es müssen alle Kopien abgearbeitet werden
- Erstmal die “einfachen” aus der Worklist

```
wähle eine beliebige anstehende Kopie (src, dest) ∈ Worklist
Worklist ← Worklist − {(src, dest)}
// Falls dest an Einfügestelle seiner Kopieranweisung live ist
if dest ∈ block.liveOut() then
    // vermeide Lost-Copy-Problem: Erstelle Inter-Block Sicherheitskopie
    erzeuge neue temporäre Variable “tempN”
    erzeuge Kopieranweisung “tempN := dest” vor Blockende
    // Merken für Inter-Block-Umbenennung von dest nach “tempN”
    Stacks[dest].push(``tempN'')
    Pushed ← Pushed ∪ {dest}
    if dest wird verwendet in bedingtem Sprung am Blockende then
        // diesmal einfach, wir haben ja schon eine Kopie in tempN
        ersetze Auftreten von dest durch “tempN” in allen Sprungbedingungen
    end if
else
    // dest ist zwar nicht Inter-Block live, aber vielleicht am Block-Ende
    if dest wird verwendet in bedingtem Sprung am Blockende then
        // falls ja: extra eine lokale Sicherheitskopie anlegen
        erzeuge neue temporäre Variable “tempM”
        erzeuge Kopieranweisung “tempM := dest” vor Blockende
        ersetze Auftreten von dest durch “tempM” in allen Sprungbedingungen
    end if
end if
// füge jetzt Kopieranweisung zum Auflösen der eigentlichen Phi-Funktion ein
➥Pass 3a, Teil 2.
```

`schedule_copies(block)`

3a. Pass: Erzeuge Kopien für einfache Fälle aus *Worklist*, Teil 1

- Es müssen alle Kopien abgearbeitet werden
- Erstmal die “einfachen” aus der Worklist

```
wähle eine beliebige anstehende Kopie (src, dest) ∈ Worklist
Worklist ← Worklist − {(src, dest)}
// Falls dest an Einfügestelle seiner Kopieranweisung live ist
if dest ∈ block.liveOut() then
    // vermeide Lost-Copy-Problem: Erstelle Inter-Block Sicherheitskopie
    erzeuge neue temporäre Variable “tempN”
    erzeuge Kopieranweisung “tempN := dest” vor Blockende
    // Merken für Inter-Block-Umbenennung von dest nach “tempN”
    Stacks[dest].push(``tempN'')
    Pushed ← Pushed ∪ {dest}
    if dest wird verwendet in bedingtem Sprung am Blockende then
        // diesmal einfach, wir haben ja schon eine Kopie in tempN
        ersetze Auftreten von dest durch “tempN” in allen Sprungbedingungen
    end if
else
    // dest ist zwar nicht Inter-Block live, aber vielleicht am Block-Ende
    if dest wird verwendet in bedingtem Sprung am Blockende then
        // falls ja: extra eine lokale Sicherheitskopie anlegen
        erzeuge neue temporäre Variable “tempM”
        erzeuge Kopieranweisung “tempM := dest” vor Blockende
        ersetze Auftreten von dest durch “tempM” in allen Sprungbedingungen
    end if
end if
// füge jetzt Kopieranweisung zum Auflösen der eigentlichen Phi-Funktion ein
➥Pass 3a, Teil 2.
```

`schedule_copies(block)`

3a. Pass: Erzeuge Kopien für einfache Fälle aus *Worklist*, Teil 1

- Es müssen alle Kopien abgearbeitet werden
- Erstmal die “einfachen” aus der Worklist

```
wähle eine beliebige anstehende Kopie (src, dest) ∈ Worklist
Worklist ← Worklist − {(src, dest)}
// Falls dest an Einfügestelle seiner Kopieranweisung live ist
if dest ∈ block.liveOut() then
    // vermeide Lost-Copy-Problem: Erstelle Inter-Block Sicherheitskopie
    erzeuge neue temporäre Variable “tempN”
    erzeuge Kopieranweisung “tempN := dest” vor Blockende
    // Merken für Inter-Block-Umbenennung von dest nach “tempN”
    Stacks[dest].push(``tempN'')
    Pushed ← Pushed ∪ {dest}
    if dest wird verwendet in bedingtem Sprung am Blockende then
        // diesmal einfach, wir haben ja schon eine Kopie in tempN
        ersetze Auftreten von dest durch “tempN” in allen Sprungbedingungen
    end if
else
    // dest ist zwar nicht Inter-Block live, aber vielleicht am Block-Ende
    if dest wird verwendet in bedingtem Sprung am Blockende then
        // falls ja: extra eine lokale Sicherheitskopie anlegen
        erzeuge neue temporäre Variable “tempM”
        erzeuge Kopieranweisung “tempM := dest” vor Blockende
        ersetze Auftreten von dest durch “tempM” in allen Sprungbedingungen
    end if
end if
// füge jetzt Kopieranweisung zum Auflösen der eigentlichen Phi-Funktion ein
➥Pass 3a, Teil 2.
```

`schedule_copies(block)`
3a. Pass: Erzeuge Kopien für einfache Fälle aus *Worklist*, Teil 1

- Es müssen alle Kopien abgearbeitet werden
- Erstmal die “einfachen” aus der Worklist

wähle eine beliebige anstehende Kopie $(src, dest) \in Worklist$
$Worklist \leftarrow Worklist - \{(src, dest)\}$
// Falls *dest* an Einfügestelle seiner Kopieranweisung live ist
if *dest* ∈ `block.liveOut()` **then**
    // vermeide Lost-Copy-Problem: Erstelle Inter-Block Sicherheitskopie
    erzeuge neue temporäre Variable “`tempN`”
    erzeuge Kopieranweisung “`tempN := dest`” vor Blockende
    // Merken für Inter-Block-Umbenennung von *dest* nach “`tempN`”
    `Stacks[dest].push(``tempN'')`
    `Pushed` ← `Pushed` ∪ {*dest*}
    if *dest* wird verwendet in bedingtem Sprung am Blockende **then**
        // diesmal einfach, wir haben ja schon eine Kopie in `tempN`
        ersetze Auftreten von *dest* durch “`tempN`” in allen Sprungbedingungen
    end if
else
    // *dest* ist zwar nicht Inter-Block live, aber vielleicht am Block-Ende
    if *dest* wird verwendet in bedingtem Sprung am Blockende **then**
        // falls ja: extra eine lokale Sicherheitskopie anlegen
        erzeuge neue temporäre Variable “`tempM`”
        erzeuge Kopieranweisung “`tempM := dest`” vor Blockende
        ersetze Auftreten von *dest* durch “`tempM`” in allen Sprungbedingungen
    end if
end if
// füge jetzt Kopieranweisung zum Auflösen der eigentlichen Phi-Funktion ein
➥Pass 3a, Teil 2.

`schedule_copies(block)`

3a. Pass: Erzeuge Kopien für einfache Fälle aus *Worklist*, Teil 2

- Sonderfälle vorher erledigt
  - Variable war Live-Out aus Block
  - Variable in Sprungbedingung am Blockende
- Jetzt Kopieranweisungen für eigentliche Phi-Funktionen erzeugen

```
// Auflösung dieses Teils der Phi-Funktion
erzeuge Kopieranweisung "dest := Map[src]" vor Block-Ende
// merken, wo Wert von src jetzt verfügbar ist
Map[src] ← dest
// wurde so ein Konflikt aus Copyset aufgelöst?
for alle Kopien (s,d) ∈ Copyset mit d = src do
    // es gibt also eine vorher zurückgestellte Kopie mit src als Ziel
    // da oben der Wert von src in dest kopiert wurde,
    // kann src selbst nun überschrieben werden: geschickte Ablaufplanung
    Worklist ← Worklist ∪ {(s,d)}
    Copyset ← Copyset − {(s,d)}
end for
```

`schedule_copies(block)`

3a. Pass: Erzeuge Kopien für einfache Fälle aus *Worklist*, Teil 2

- Sonderfälle vorher erledigt
  - Variable war Live-Out aus Block
  - Variable in Sprungbedingung am Blockende
- Jetzt Kopieranweisungen für eigentliche Phi-Funktionen erzeugen

```
// Auflösung dieses Teils der Phi-Funktion
erzeuge Kopieranweisung "dest := Map[src]" vor Block-Ende
// merken, wo Wert von src jetzt verfügbar ist
Map[src] ← dest
// wurde so ein Konflikt aus Copyset aufgelöst?
for alle Kopien (s,d) ∈ Copyset mit d = src do
    // es gibt also eine vorher zurückgestellte Kopie mit src als Ziel
    // da oben der Wert von src in dest kopiert wurde,
    // kann src selbst nun überschrieben werden: geschickte Ablaufplanung
    Worklist ← Worklist ∪ {(s,d)}
    Copyset ← Copyset − {(s,d)}
end for
```

`schedule_copies(block)`

3a. Pass: Erzeuge Kopien für einfache Fälle aus *Worklist*, Teil 2

- Sonderfälle vorher erledigt
  - Variable war Live-Out aus Block
  - Variable in Sprungbedingung am Blockende
- Jetzt Kopieranweisungen für eigentliche Phi-Funktionen erzeugen

```
// Auflösung dieses Teils der Phi-Funktion
erzeuge Kopieranweisung "dest := Map[src]" vor Block-Ende
// merken, wo Wert von src jetzt verfügbar ist
Map[src] ← dest
// wurde so ein Konflikt aus Copyset aufgelöst?
for alle Kopien (s,d) ∈ Copyset mit d = src do
    // es gibt also eine vorher zurückgestellte Kopie mit src als Ziel
    // da oben der Wert von src in dest kopiert wurde,
    // kann src selbst nun überschrieben werden: geschickte Ablaufplanung
    Worklist ← Worklist ∪ {(s,d)}
    Copyset ← Copyset − {(s,d)}
end for
```

`schedule_copies(block)`

3a. Pass: Erzeuge Kopien für einfache Fälle aus *Worklist*, Teil 2

- Sonderfälle vorher erledigt
  - Variable war Live-Out aus Block
  - Variable in Sprungbedingung am Blockende
- Jetzt Kopieranweisungen für eigentliche Phi-Funktionen erzeugen

```
// Auflösung dieses Teils der Phi-Funktion
erzeuge Kopieranweisung "dest := Map[src]" vor Block-Ende
// merken, wo Wert von src jetzt verfügbar ist
Map[src] ← dest
// wurde so ein Konflikt aus Copyset aufgelöst?
for alle Kopien (s,d) ∈ Copyset mit d = src do
    // es gibt also eine vorher zurückgestellte Kopie mit src als Ziel
    // da oben der Wert von src in dest kopiert wurde,
    // kann src selbst nun überschrieben werden: geschickte Ablaufplanung
    Worklist ← Worklist ∪ {(s,d)}
    Copyset ← Copyset − {(s,d)}
end for
```

`schedule_copies(block)`

3a. Pass: Erzeuge Kopien für einfache Fälle aus *Worklist*, Teil 2

- Sonderfälle vorher erledigt
  - Variable war Live-Out aus Block
  - Variable in Sprungbedingung am Blockende
- Jetzt Kopieranweisungen für eigentliche Phi-Funktionen erzeugen

```
// Auflösung dieses Teils der Phi-Funktion
erzeuge Kopieranweisung "dest := Map[src]" vor Block-Ende
// merken, wo Wert von src jetzt verfügbar ist
Map[src] ← dest
// wurde so ein Konflikt aus Copyset aufgelöst?
for alle Kopien (s,d) ∈ Copyset mit d = src do
    // es gibt also eine vorher zurückgestellte Kopie mit src als Ziel
    // da oben der Wert von src in dest kopiert wurde,
    // kann src selbst nun überschrieben werden: geschickte Ablaufplanung
    Worklist ← Worklist ∪ {(s,d)}
    Copyset ← Copyset − {(s,d)}
end for
```

`schedule_copies(block)`

3a. Pass: Erzeuge Kopien für einfache Fälle aus *Worklist*, Teil 2

- Sonderfälle vorher erledigt
  - Variable war Live-Out aus Block
  - Variable in Sprungbedingung am Blockende
- Jetzt Kopieranweisungen für eigentliche Phi-Funktionen erzeugen

```
// Auflösung dieses Teils der Phi-Funktion
erzeuge Kopieranweisung "dest := Map[src]" vor Block-Ende
// merken, wo Wert von src jetzt verfügbar ist
Map[src] ← dest
// wurde so ein Konflikt aus Copyset aufgelöst?
for alle Kopien (s,d) ∈ Copyset mit d = src do
    // es gibt also eine vorher zurückgestellte Kopie mit src als Ziel
    // da oben der Wert von src in dest kopiert wurde,
    // kann src selbst nun überschrieben werden: geschickte Ablaufplanung
    Worklist ← Worklist ∪ {(s,d)}
    Copyset ← Copyset − {(s,d)}
end for
```

schedule_copies(block)

3b. Pass: Vereinfache komplizierte Fälle aus *Copyset*

- Nun keine einfachen Fälle mehr übrig
  - Geschickte Ablaufplanung hat nicht gereicht
  - Es existieren zirkuläre Abhängigkeiten ($\longrightarrow$ Vertauschungsproblem)
- Aufbrechen durch Kopieroperationen

```
// picke ein Element aus dem Zyklus
wähle ein beliebiges (src, dest) ∈ Copyset
erzeuge neue temporäre Variable "tempO"
erzeuge Kopieranweisung "tempO := dest" vor Block-Ende
// der Wert von dest steht jetzt in tempO
// dest kann also überschrieben werden, der Zyklus ist gebrochen
Map[dest] ← "tempO"
// Nun haben wir wieder einen einfachen Fall für Worklist
Copyset ← Copyset − {(src, dest)}
Worklist ← Worklist ∪ {(src, dest)}
```

`schedule_copies(block)`

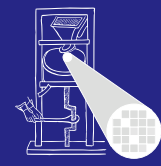
3b. Pass: Vereinfache komplizierte Fälle aus *Copyset*

- Nun keine einfachen Fälle mehr übrig
  - Geschickte Ablaufplanung hat nicht gereicht
  - Es existieren zirkuläre Abhängigkeiten ($\longrightarrow$ Vertauschungsproblem)
- Aufbrechen durch Kopieroperationen

```
// picke ein Element aus dem Zyklus
wähle ein beliebiges (src, dest) ∈ Copyset
erzeuge neue temporäre Variable "tmp0"
erzeuge Kopieranweisung "tmp0 := dest" vor Block-Ende
// der Wert von dest steht jetzt in tmp0
// dest kann also überschrieben werden, der Zyklus ist gebrochen
Map[dest] ← "tmp0"
// Nun haben wir wieder einen einfachen Fall für Worklist
Copyset ← Copyset − {(src, dest)}
Worklist ← Worklist ∪ {(src, dest)}
```

`schedule_copies(block)`

3b. Pass: Vereinfache komplizierte Fälle aus *Copyset*

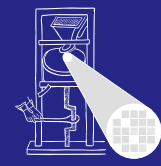


- Nun keine einfachen Fälle mehr übrig
  - Geschickte Ablaufplanung hat nicht gereicht
  - Es existieren zirkuläre Abhängigkeiten ($\rightarrow$ Vertauschungsproblem)
- Aufbrechen durch Kopieroperationen

```
// picke ein Element aus dem Zyklus
wähle ein beliebiges (src, dest) ∈ Copyset
erzeuge neue temporäre Variable "tmp0"
erzeuge Kopieranweisung "tmp0 := dest" vor Block-Ende
// der Wert von dest steht jetzt in tmp0
// dest kann also überschrieben werden, der Zyklus ist gebrochen
Map[dest] ← "tmp0"
// Nun haben wir wieder einen einfachen Fall für Worklist
Copyset ← Copyset − {(src, dest)}
Worklist ← Worklist ∪ {(src, dest)}
```

`schedule_copies(block)`

3b. Pass: Vereinfache komplizierte Fälle aus *Copyset*

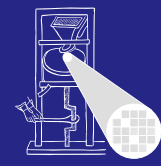


- Nun keine einfachen Fälle mehr übrig
  - Geschickte Ablaufplanung hat nicht gereicht
  - Es existieren zirkuläre Abhängigkeiten ($\rightarrow$ Vertauschungsproblem)
- Aufbrechen durch Kopieroperationen

```
// picke ein Element aus dem Zyklus
wähle ein beliebiges (src, dest) ∈ Copyset
erzeuge neue temporäre Variable "tmp0"
erzeuge Kopieranweisung "tmp0 := dest" vor Block-Ende
// der Wert von dest steht jetzt in tmp0
// dest kann also überschrieben werden, der Zyklus ist gebrochen
Map[dest] ← "tmp0"
// Nun haben wir wieder einen einfachen Fall für Worklist
Copyset ← Copyset − {(src, dest)}
Worklist ← Worklist ∪ {(src, dest)}
```

`schedule_copies(block)`

3b. Pass: Vereinfache komplizierte Fälle aus *Copyset*



- Nun keine einfachen Fälle mehr übrig
  - Geschickte Ablaufplanung hat nicht gereicht
  - Es existieren zirkuläre Abhängigkeiten ($\rightarrow$ Vertauschungsproblem)
- Aufbrechen durch Kopieroperationen

```
// picke ein Element aus dem Zyklus
wähle ein beliebiges (src, dest) ∈ Copyset
erzeuge neue temporäre Variable "tmp0"
erzeuge Kopieranweisung "tmp0 := dest" vor Block-Ende
// der Wert von dest steht jetzt in tmp0
// dest kann also überschrieben werden, der Zyklus ist gebrochen
Map[dest] ← "tmp0"
// Nun haben wir wieder einen einfachen Fall für Worklist
Copyset ← Copyset − {(src, dest)}
Worklist ← Worklist ∪ {(src, dest)}
```

`schedule_copies(block)`

3b. Pass: Vereinfache komplizierte Fälle aus *Copyset*

- Nun keine einfachen Fälle mehr übrig
  - Geschickte Ablaufplanung hat nicht gereicht
  - Es existieren zirkuläre Abhängigkeiten ($\rightarrow$ Vertauschungsproblem)
- Aufbrechen durch Kopieroperationen

// picke ein Element aus dem Zyklus
wähle ein beliebiges $(src, dest) \in Copyset$
erzeuge neue temporäre Variable "`tmp0`"
erzeuge Kopieranweisung "`tmp0 := dest`" vor Block-Ende
// der Wert von *dest* steht jetzt in `tmp0`
// *dest* kann also überschrieben werden, der Zyklus ist gebrochen
`Map`$[dest] \leftarrow$ "`tmp0`"
// Nun haben wir wieder einen einfachen Fall für *Worklist*
$Copyset \leftarrow Copyset - \{(src, dest)\}$
$Worklist \leftarrow Worklist \cup \{(src, dest)\}$

Weiteres Vorgehen

- Beispiele aus Folien so mit Papier und Bleistift durchrechnen
- Erläuterung des Algorithmus in Briggs auf p. 29. . . 33
- Fehler bei Briggs
  - “Variable kommt in Sprungbedingung vor” nicht behandelt
  - Hier in Pass 3a, Teil 1 erledigt
- Implementieren erst nach dem Verstehen!
- Falls doch Vorarbeiten gemacht werden sollen
  - Datenflussanalyse für Live-Variables

Weiteres Vorgehen

- Beispiele aus Folien so mit Papier und Bleistift durchrechnen
- Erläuterung des Algorithmus in Briggs auf p. 29. . . 33
- Fehler bei Briggs
  - “Variable kommt in Sprungbedingung vor” nicht behandelt
  - Hier in Pass 3a, Teil 1 erledigt
- Implementieren erst nach dem Verstehen!
- Falls doch Vorarbeiten gemacht werden sollen
  - Datenflussanalyse für Live-Variables

Weiteres Vorgehen

- Beispiele aus Folien so mit Papier und Bleistift durchrechnen
- Erläuterung des Algorithmus in Briggs auf p. 29. . . 33
- Fehler bei Briggs
  - “Variable kommt in Sprungbedingung vor” nicht behandelt
  - Hier in Pass 3a, Teil 1 erledigt
- Implementieren erst nach dem Verstehen!
- Falls doch Vorarbeiten gemacht werden sollen
  - Datenflussanalyse für Live-Variables

Weiteres Vorgehen

- Beispiele aus Folien so mit Papier und Bleistift durchrechnen
- Erläuterung des Algorithmus in Briggs auf p. 29. . . 33
- Fehler bei Briggs
  - “Variable kommt in Sprungbedingung vor” nicht behandelt
  - Hier in Pass 3a, Teil 1 erledigt
- Implementieren erst nach dem Verstehen!
- Falls doch Vorarbeiten gemacht werden sollen
  - Datenflussanalyse für Live-Variables

Weiteres Vorgehen

- Beispiele aus Folien so mit Papier und Bleistift durchrechnen
- Erläuterung des Algorithmus in Briggs auf p. 29. . . 33
- Fehler bei Briggs
  - “Variable kommt in Sprungbedingung vor” nicht behandelt
  - Hier in Pass 3a, Teil 1 erledigt
- Implementieren erst nach dem Verstehen!
- Falls doch Vorarbeiten gemacht werden sollen
  - Datenflussanalyse für Live-Variables

Weiteres Vorgehen

- Beispiele aus Folien so mit Papier und Bleistift durchrechnen
- Erläuterung des Algorithmus in Briggs auf p. 29. . . 33
- Fehler bei Briggs
  - “Variable kommt in Sprungbedingung vor” nicht behandelt
  - Hier in Pass 3a, Teil 1 erledigt
- Implementieren erst nach dem Verstehen!
- Falls doch Vorarbeiten gemacht werden sollen
  - Datenflussanalyse für Live-Variables

Weiteres Vorgehen

- Beispiele aus Folien so mit Papier und Bleistift durchrechnen
- Erläuterung des Algorithmus in Briggs auf p. 29. . . 33
- Fehler bei Briggs
  - “Variable kommt in Sprungbedingung vor” nicht behandelt
  - Hier in Pass 3a, Teil 1 erledigt
- Implementieren erst nach dem Verstehen!
- Falls doch Vorarbeiten gemacht werden sollen
  - Datenflussanalyse für Live-Variables

Weiteres Vorgehen

- Beispiele aus Folien so mit Papier und Bleistift durchrechnen
- Erläuterung des Algorithmus in Briggs auf p. 29. . . 33
- Fehler bei Briggs
  - “Variable kommt in Sprungbedingung vor” nicht behandelt
  - Hier in Pass 3a, Teil 1 erledigt
- Implementieren erst nach dem Verstehen!
- Falls doch Vorarbeiten gemacht werden sollen
  - Datenflussanalyse für Live-Variables