Kapitel 10
SSA-Form

Prof. Dr. Dr. h.c. Gerhard Goos
ggoos@ipd.info.uni-karlsruhe.de

Institut für Programmstrukturen und Datenorganisation
Universität Karlsruhe

Wintersemester 2007/8

SSA Aufbau

Inhalt

Statische Einmalzuweisung – SSA

Ziel:

- Effizienz prozedurglobaler Optimierungen steigern
- Datenflußanalysen beschleunigen
- Definiert-Benutzt-Beziehungen explizit darstellen

Definition SSA (Static Single Assignment, statische Einmalzuweisung): Ein Programm ist in SSA Form, wenn an jede Variable jeweils an genau einer Programmstelle zugewiesen wird.

- Programm bedeutet hier zunächst Prozedur
- verlangt: Ablauf des Programms ist reduzibel
- Variable bedeutet hier zunächst aliasfreie, lokale Variable
- beachte: SSA bedeutet nicht, daß jeder Wert nur einmal berechnet wird.

SSA intuitiv: Erkenntnisse

- Maschinenbefehle verarbeiten Werte im Speicher/Registern. Es kommt nicht darauf an, wie die zugehörigen Variablen im Quellprogramm heißen. (Viele Werte entstammen der Adreßrechnung und sind namenlos.)
- Wenn zweimal dieselbe Operation auf die gleichen Operanden angewandt wird, kann man eine Operation weglassen (Idee der Wertnumerierung)
  - gleicher Operand heißt hier: gleicher Wert zur Laufzeit
- Ob zwei Operanden gleichen Wert besitzen, kann von der Vorgeschichte, dem Ablaufpfad, der zu der Berechnung führt, abhängen.

SSA intuitiv: schematische Konstruktion

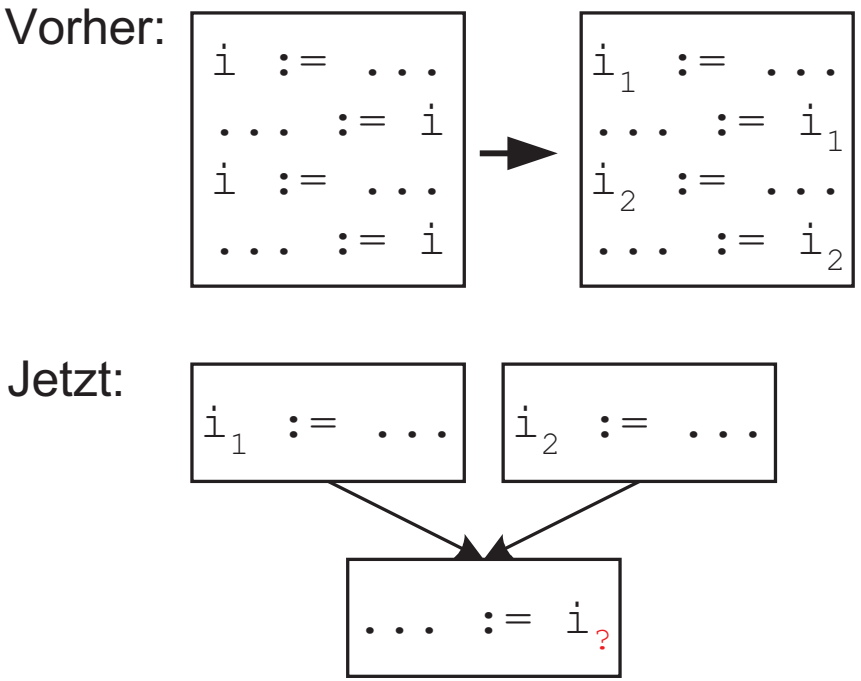
Konstruktionsidee:

- Ersetze alle Zuweisungen durch Vereinbarungen (Definitionen) dynamischer Konstanter, an die kein zweites Mal zugewiesen werden kann (daher SSA: statische Einmalzuweisung)
- Verschiedene Zuweisungen an eine Programmvariable a führen zu Vereinbarungen verschiedener Konstanter $a_1, a_2, \ldots$
- Überall, wo a als Operand benutzt wird, setze die dort gültige Definition a_i ein
- Problem: Was tun, wenn die gültige Definition vom Ablaufpfad abhängt, auf dem die Benutzung als Operand erreicht wird?
- Lösung: Setze zuvor eine Auswahlfunktion (ϕ-Funktion) ein, die die gültige Definition abhängig vom Ablaufpfad selektiert.
- Nebenbei: Diese Idee löst das Datenflußproblem „ordne der Verwendung einer Variablen ihre letzte Definition zu“!

Abstrakte Werte

Problem: Wie verfährt man mit verschiedenen abstrakten Werten bei Ablauf-Vereinigung?

Welches *i* ist hier gültig?

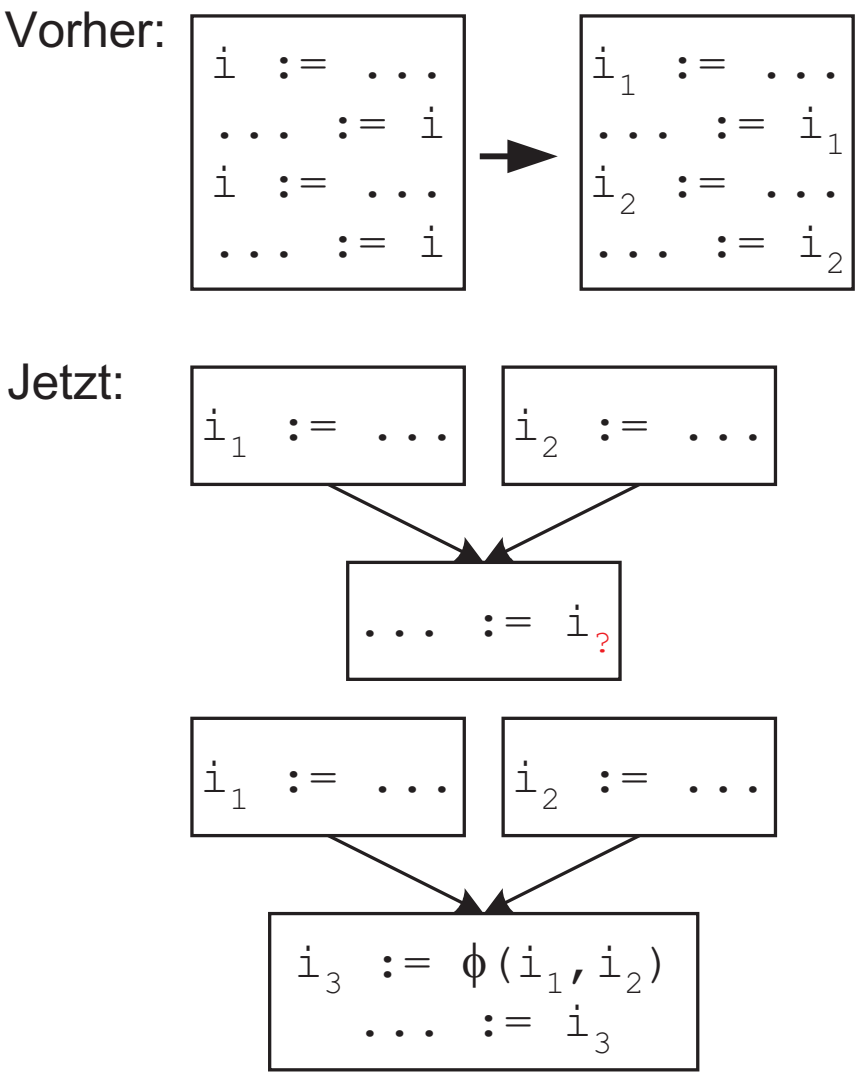


Abstrakte Werte II

Problem: Wie verfährt man mit verschiedenen abstrakten Werten bei Ablauf-Vereinigung?

Welches i ist hier gültig?

Bei Zusammenführungen des Ablaufs den Wert durch eine Pseudooperation $i_3 := \phi(i_1, i_2)$, eine ϕ-Funktion, auswählen.

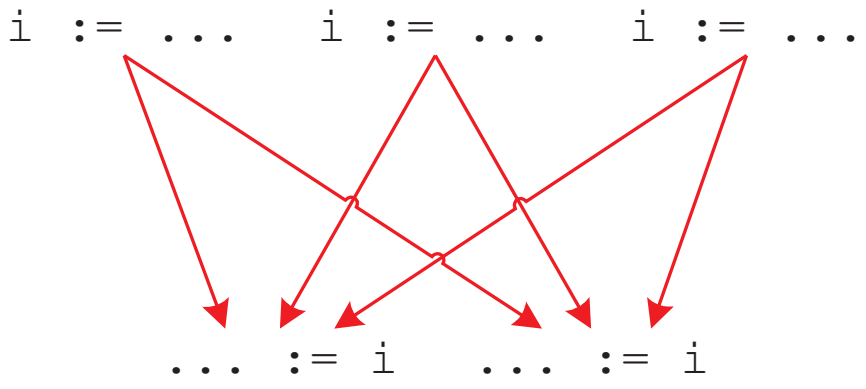


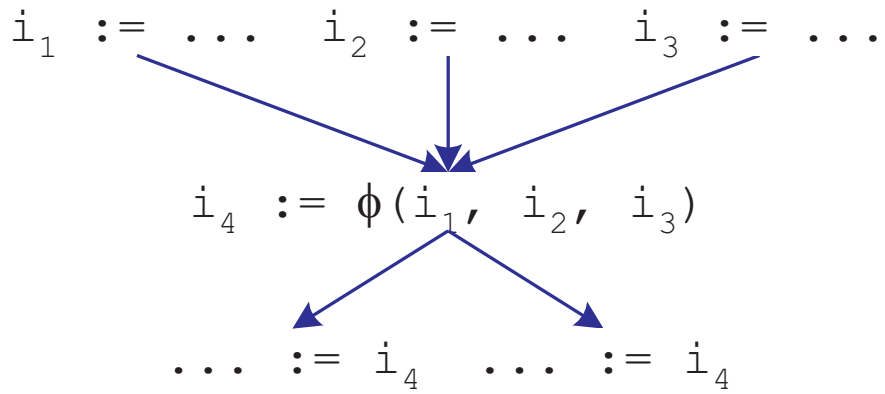
Explizite Definiert-Benutzt-Beziehungen

Die SSA Form verringert den Aufwand zur Darstellung von Definiert-Benutzt-Beziehungen:

vorher: #Defs $\times$ #Bens

jetzt: #Defs $+$ #Bens





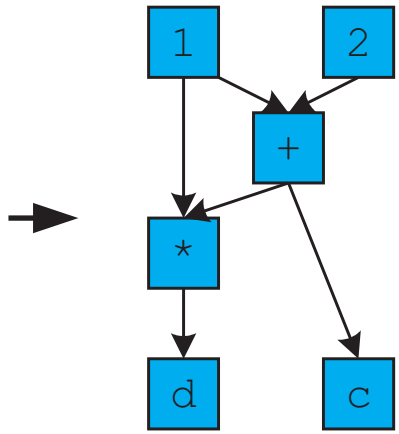
ϕ-Funktionen

- Eine ϕ-Funktion $i_3 := \phi(i_1, i_2)$ wählt abhängig vom Programmablauf einen der Werte i_1, i_2 aus und benutzt ihn als Wert i_3.
- Eine ϕ-Funktion hat genau so viele Operanden, wie der zugehörige Grundblock Vorgänger im Ablaufgraph.
- Das k-te Argument einer ϕ-Funktion ist eineindeutig dem k-ten Vorgänger zugeordnet.
- Das Ergebnis einer ϕ-Funktion ist das Argument, das dem Pfad, auf dem die ϕ-Funktion erreicht wurde, zugeordnet ist.
- ϕ-Funktionen stehen immer am Blockanfang.
- Alle ϕ-Funktionen eines Blocks werden simultan ausgewertet.

Implementierung

- Optimierungen benötigen Definiert-Benutzt-Beziehungen.
- Dafür Operationen in Tripelform als Ecken eines Graphen reinterpretieren:
- Graph-Interpretation:
  - Jeder abstrakte Wert (Wertnummer) ist eine Ecke.
  - Jede Ecke enthält Operation oder Konstante.
  - Def.-Ben. Beziehungen sind Kanten (Datenflußkanten).
  - Umkehrung der Pfeile entspricht Datenabhängigkeiten.
  - Ablaufkante: Welchen Grundblock erreicht ein Sprung? Zu welchem Grundblock gehört eine Operation?
  - Weitere Kantenarten später

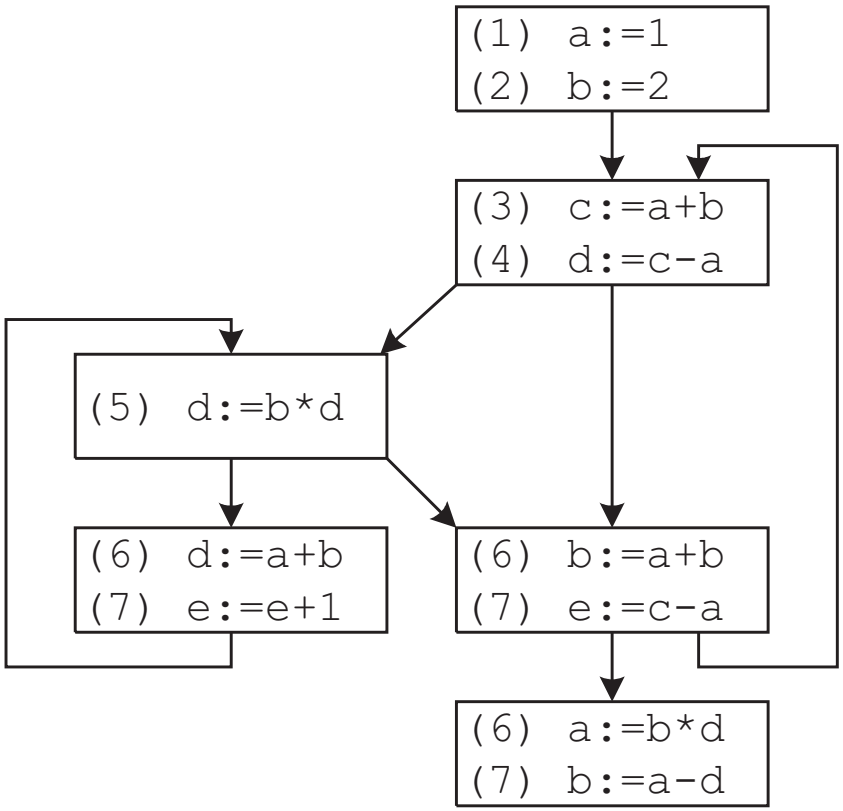
```
a := 1
b := 2
c := a + b
d := c * a
...
```

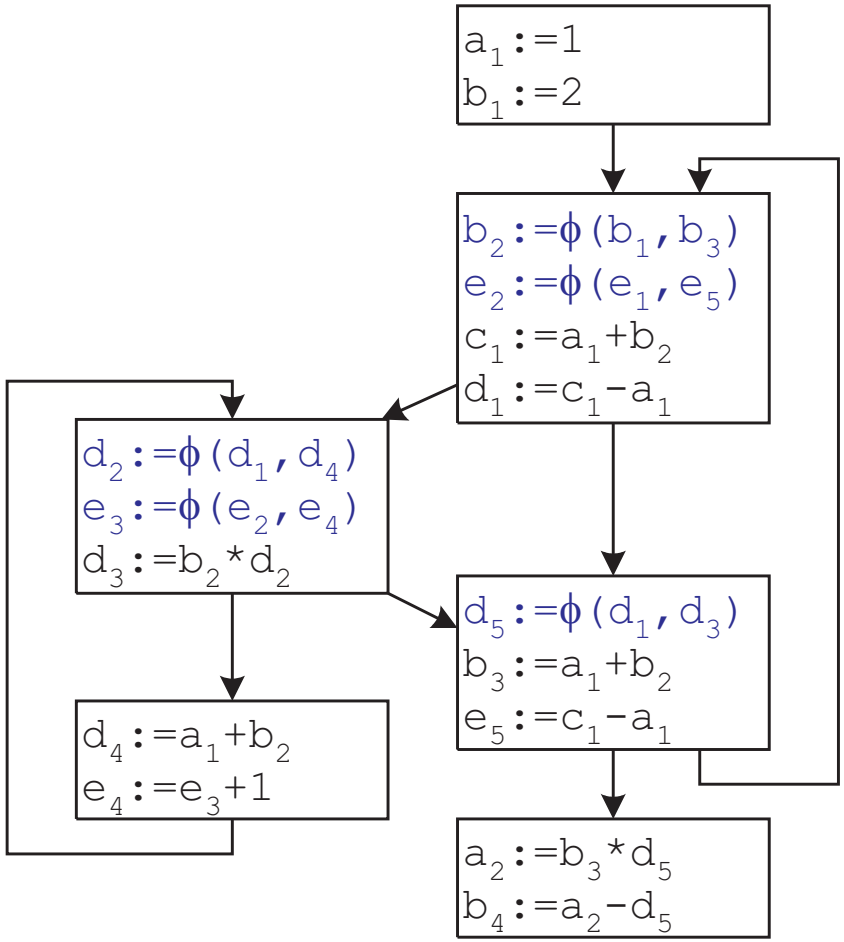


Implementierung: Firm

- Firm ist eine moderne SSA-Zwischendarstellung in Graphform; entstanden aus Diplomarbeiten, Dissertationen, Industriekooperationen des Lehrstuhls 1995-2006.
- Ablauf, Datenfluß und Sequentialisierung von Operationen werden in einem Graphen dargestellt.
- Firm ist ein Abhängigkeitsgraph, die Kantenrichtung ist also umgekehrt zur Datenflußrichtung.
- Zur Unterscheidung der Konzepte sind die Kanten des Graphen typisiert, die Typen heißen Modi.
- Grundblöcke sind auch Ecken, jede andere Ecke hat eine Grundblockecke als Vorgänger (ermöglicht einheitlichen ADT)
- Firm ist durch Angabe der Operationen (Ecken), Kantenmodi und der Signatur der Operationen definiert.

Beispiel: Grundblock- und SSA-Grundblockgraph





Inhalt

Wdh.: Dominanz und Dominatorbäume

- *Dominanz*: $X \geq Y$
  Auf jedem Pfad vom Startblock S im Ablaufgraph kommt X vor Y.
  $\geq$ ist reflexiv: $X \geq X$.
- *Strikte Dominanz*:

$$X > Y \Longrightarrow X \geq Y \wedge X \neq Y.$$

- *Unmittelbare (direkte) Dominanz*: $ddom(X)$

$$X = ddom(Y) \Longrightarrow X > Y \wedge \neg \exists Z : X > Z > Y.$$

- *Nach-Dominanz*: $X \leq Y$ Auf jedem Pfad von Y zum Endblock E im Ablaufgraph kommt Y vor X.
- Übrige Definitionen für Nach-Dominanz analog.

Wdh.: Dominanzgrenze und iterierte DG

- *Dominanzgrenze* $DG(X)$
  Menge von Blöcken die gerade nicht mehr von X dominiert werden.

$$DG(X) := \{Y | \exists P \in Vor(Y) : X \geq P \wedge \neg(X > Y)\}.$$

- Dominanzgrenze einer *Menge* M von Blöcken $DG(M)$

$$DG(M) := \bigcup_{X \in M} DG(X)$$

- *Iterierte Dominanzgrenze* $DG^+(M)$ minimaler Fixpunkt von:

$$\begin{aligned} DG_0 &:= DG(M), \\ DG_{i+1} &:= DG(M \cup DG_i) \end{aligned}$$

Plazierung der ϕ-Funktionen

Wo müssen die ϕ-Funktionen optimal plaziert werden?

- SSA-Eigenschaft muß erfüllt sein (nur eine Definition)
- Programm muß korrekt dargestellt sein
- Minimale Anzahl von ϕ-Funktionen

Satz *Plazierung ϕ-Funktionen*:
Sei P eine Prozedur in SSA Form mit minimaler Anzahl ϕ-Funktionen.
Seien X, Y Grundblöcke in P mit einer Definition von v und Ablaufpfaden $X \rightarrow^+ Z, Y \rightarrow^+ Z$, wobei Z der erste gemeinsame GB dieser Pfade ist.
Dann enthält Z eine ϕ-Funktion für v, *falls noch ein Gebrauch von v folgt*.

Beweisidee

- ϕ-Funktion kann nicht früher eingesetzt werden.
- ϕ-Funktion darf nicht in einen späteren Block Z' eingesetzt werden: Die Wege $Z \rightarrow^+ Z'$ enthalten keine Möglichkeit, die ursprünglichen Definitionen von v zu differenzieren.

Folgerungen

- X bzw. Y müssen alle direkten Vorgänger von Z dominieren, sonst gäbe es einen Gebrauch von v ohne vorherige Definition.
  Daher gilt $Z \in DG(X, Y)$.
  Nebenbei: beim Einsetzen von ϕ-Funktionen werden alle nicht-initialisierten, aber benutzten einfachen Variablen entdeckt!
- Da die ϕ-Funktion in Z eine neue Definition von v ist, werden ϕ-Funktionen in den iterierten Dominanzgrenzen der ursprünglichen Definitionen eingesetzt. Hier werden weitere Definitionen von v, die erst in einem Block $\notin DG(X, Y)$ hinzugenommen werden, vereinigt.

Achtung:
Dominanzgrenzen sind als Konstruktionsvorschrift weniger geeignet!

Inhalt

Wie konstruiert man SSA-Form

- mehrere Konstruktionsverfahren möglich
- im schlimmsten Fall enthalten alle Grundblöcke ϕ-Funktionen für alle Variable:
  - Aufwand $O(n^2)$, $n =$ Anzahl Variable, nicht vermeidbar
  - praktisch ist der Aufwand linear
- Grundidee unseres Verfahrens (Click 1995):
  - während eines Durchlaufs des Strukturbaums (AST) führe erweiterte Wertnumerierung durch:
    - bei nur einem Vorgängerblock Wertnummern übernehmen
    - bei mehreren Vorgängern vorläufige ϕ-Funktionen $\phi'(\ldots)$ einsetzen
    - Argumentliste der ϕ'-Funktionen erweitern, wenn weitere Vorgänger gefunden werden
    - am Ende ϕ'-Funktionen in ϕ-Funktionen umwandeln oder streichen (wenn Wert nicht mehr benötigt)

SSA Aufbau mit Wertnumerierung

- Ausgangspunkt:
  - AST oder Grundblockgraph mit Zuweisungen der Form $x := \tau(y, z)$; x, y, z lokale Variable (also aliasfrei)
  - Anzahl in Prozedur verwendeter lokaler Variablen bekannt (n)
- Vorgehen:
  - Merke in jedem Grundblock aktuellen Wert jeder Variablen, d.h. den definierenden Ausdruck (Reihung der Größe n)
  - Bei Verwendung einer Variablen benutze Wertnummer dieses Ausdrucks
  - Funktionen „hole Wertnummer“ $HW(v)$, „merke Wertnummer“ $MW(v, wn)$
  - $HW(v)$ fügt ϕ-Funktion ein, wenn Variable in Vorgängerblock definiert praktisch: ϕ-Funktionen werden nur generiert, wenn Wert noch lebendig!
  - Berechnen einer Wertnummer für Ausdrücke $t, t = \tau(y, z)$ mit $WN(t)$ wie gehabt

SSA Aufbau mit Wertnumerierung

Für jede Zuweisung $x := \tau(y, z)$:

- hole Wert für $y, z \mapsto HW(y), HW(z)$
- berechne Wertnummer $WN(\tau, y, z)$ für $\tau(y, z)$
- Falls Wertnummer neu, füge Zuweisung $WN(\tau, y, z) := \tau(HW(y), HW(z))$ in Grundblock ein.
- Merke Wert für x: $MW(x, WN(\tau, y, z))$

Bemerkung: Aufruf von WN eliminiert gemeinsame Teilausdrücke!

SSA Aufbau mit Wertnumerierung

Vorgehen von $HW(v)$:

- Wenn in aktuellem Grundblock Wert w für Variable v bereits gemerkt, verwende diesen
- Wenn genau ein Vorgängerblock rufe $HW(v)$ dort auf
- Wenn zwei (mehrere) Vorgängerblöcke:
  - rufe $HW(v)$ in jedem dieser Blöcke auf $\Rightarrow$ liefert Werte w_1, w_2
  - füge Zuweisung $WN(\phi, v, v) := \phi(w_1, w_2, \ldots)$ in aktuellem Grundblock ein.
  - merke neuen Wert für v : $MW(v, WN(\phi, v, v))$
  - gebe neuen Wert als Ergebnis zurück

SSA Aufbau und unbekannte Vorgänger

Beobachtung: Beim Aufbau von Schleifen sind die Wertnummern aus den Vorgängerblöcken u.U. nicht bekannt, $HW(v)$ also (noch) nicht definiert

Abhilfe: Zweistufiges Vorgehen:

- Markiere Blöcke in SSA-Form
- Wenn alle Vorgänger in SSA-Form, berechne ϕ-Funktion wie gehabt.
- Wenn Vorgänger nicht in SSA-Form, füge ϕ' ein und merke Operand des ϕ' zum späteren Fertigstellen (passiert nur an Dominanzgrenzen)
- Wenn Grundblock in SSA-Form gebracht, teste, ob schon vorhandene Nachfolger fertig gestellt werden können, und stelle sie fertig.

SSA Aufbau und unbekannte Vorgänger

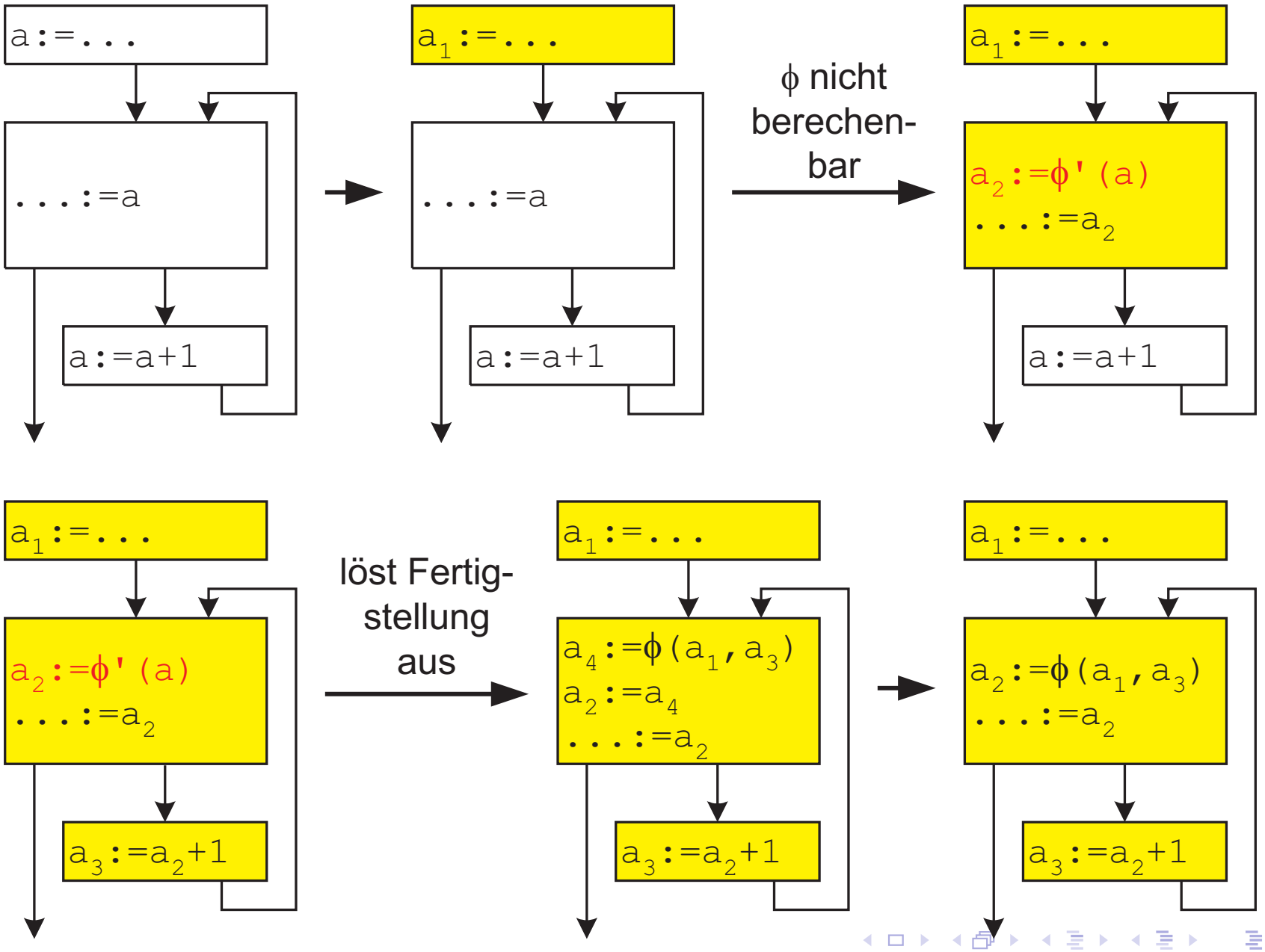
Beobachtungen:

- Geht man beim Aufbau soweit möglich in Ablaufrichtung vor, sind eindeutige Vorgänger immer vor ihren Nachfolgern in SSA-Form: Dominatoren werden immer zuerst aufgebaut.
- Bei Aufbau aus dem AST ist bekannt, wann alle Vorgänger aufgebaut sind (außer bei expliziten Sprüngen mit *goto*).

Folgerung:

- Aufbau bei reduziblem Ablauf effizient!
- Wenn Dominatoren bekannt, ist globale Eliminierung gemeinsamer Teilausdrücke effizient.

unbekannte Vorgänger: Beispiel

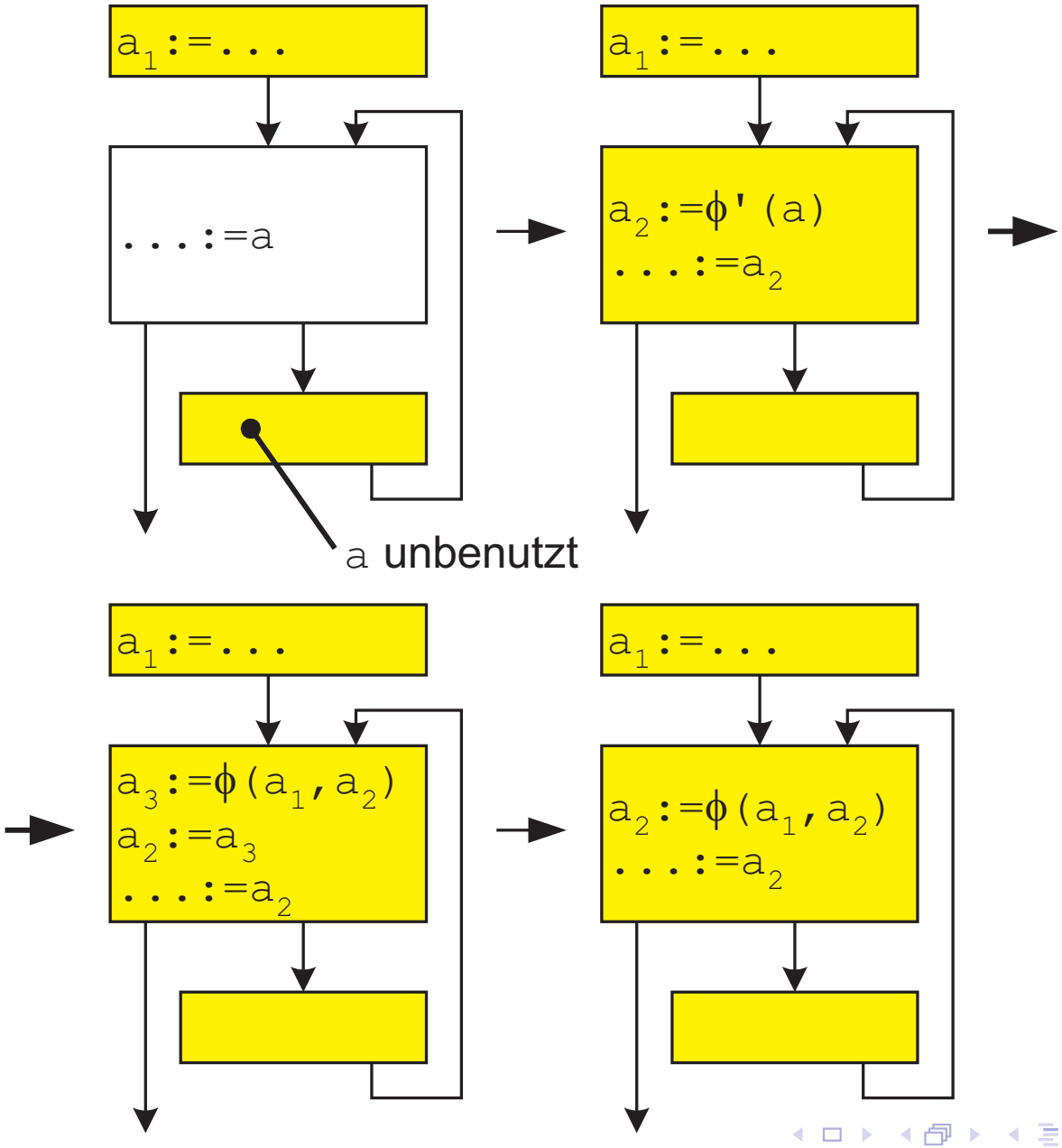


SSA Aufbau - Abbruchkriterien für HW

Beobachtung:
Algorithmus *HW* kann rekursiv über alle Vorgängerblöcke iterieren.

- Erreicht er den Startblock, wird ein undefinierter Wert verwendet (Fehlermeldung: nicht initialisierte Variable)
- Bei zyklischem Ablauf ohne Definitionen wird Endlosrekursion durch ϕ' unterbunden
  (dies ist die zweite wichtige Aufgabe der ϕ'-Funkionen!)

Abbruchkriterien für *HW*: Beispiel

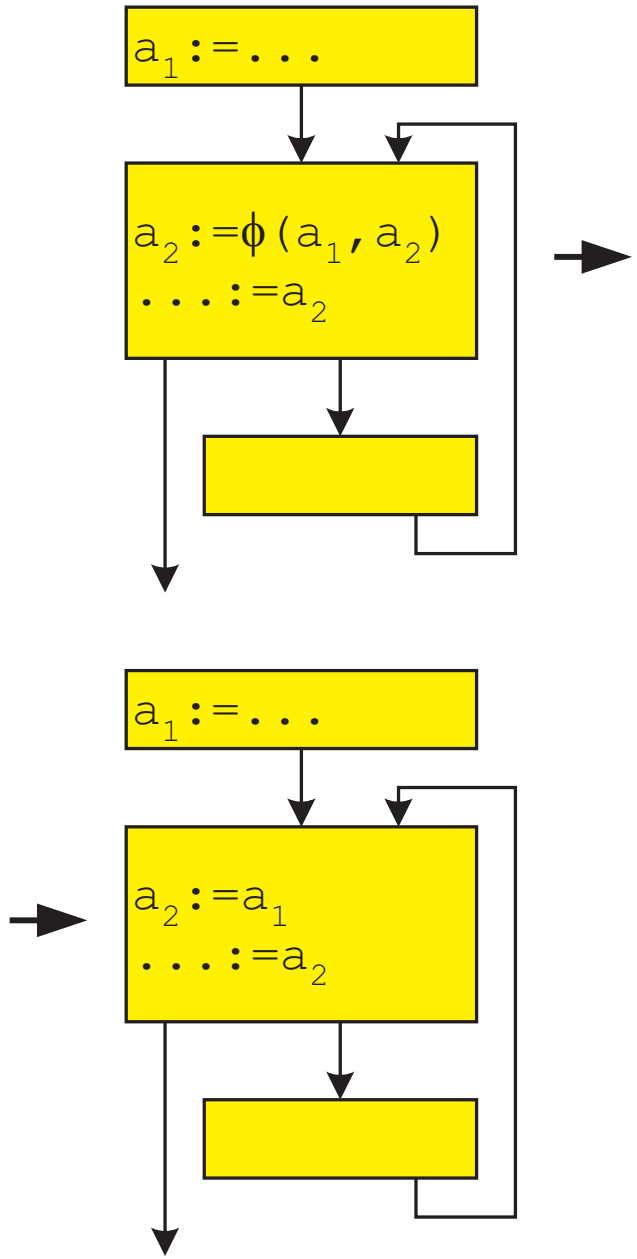


SSA Aufbau und unnötige ϕ-Funktionen

Beobachtung:
Es entstehen unnötige ϕ-Funktionen:

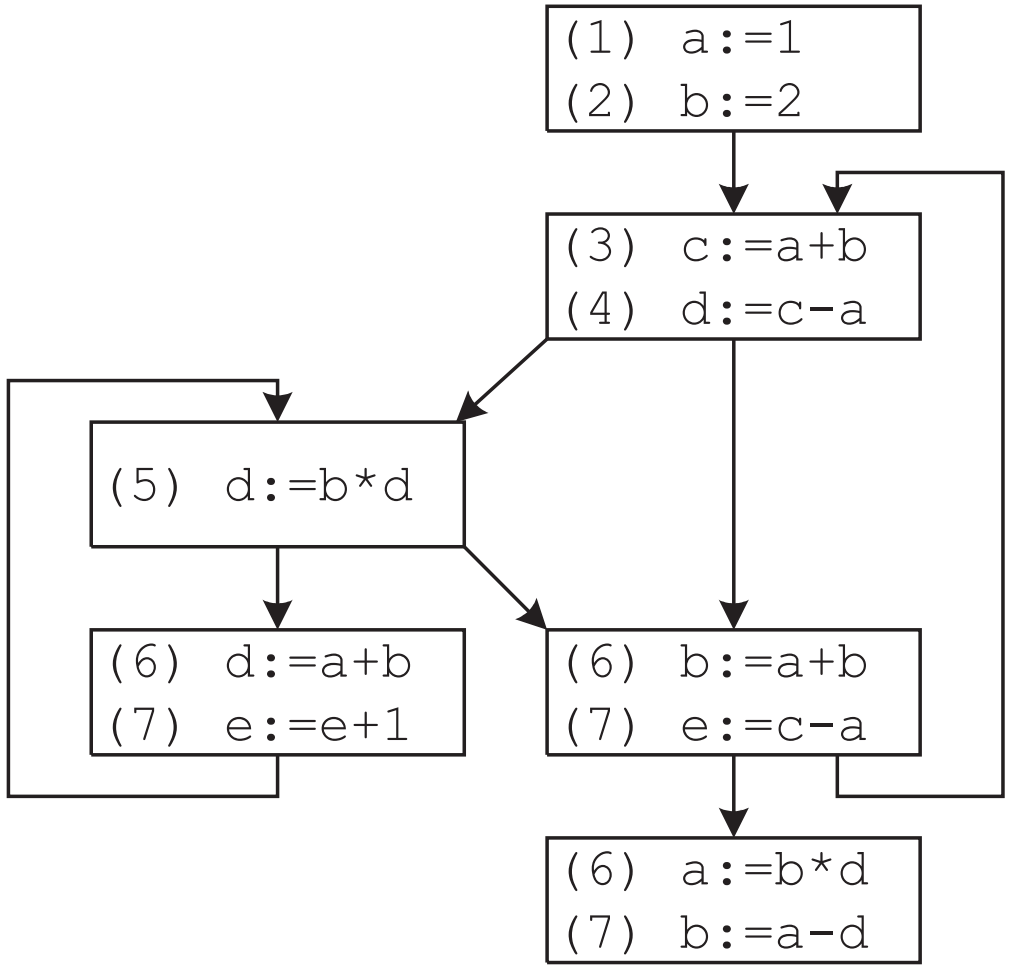
- Die Operanden sind das Ergebnis der ϕ-Funktion
- Da jede Schleife mindestens einen Vorgänger außerhalb der Schleife hat, existiert mindestens ein sinnvoller Operand
- Gibt es nur einen Operanden, so ersetze die ϕ-Funktion durch diesen.

(Bei nicht reduziblem Ablauf gibt es Komplikationen.)

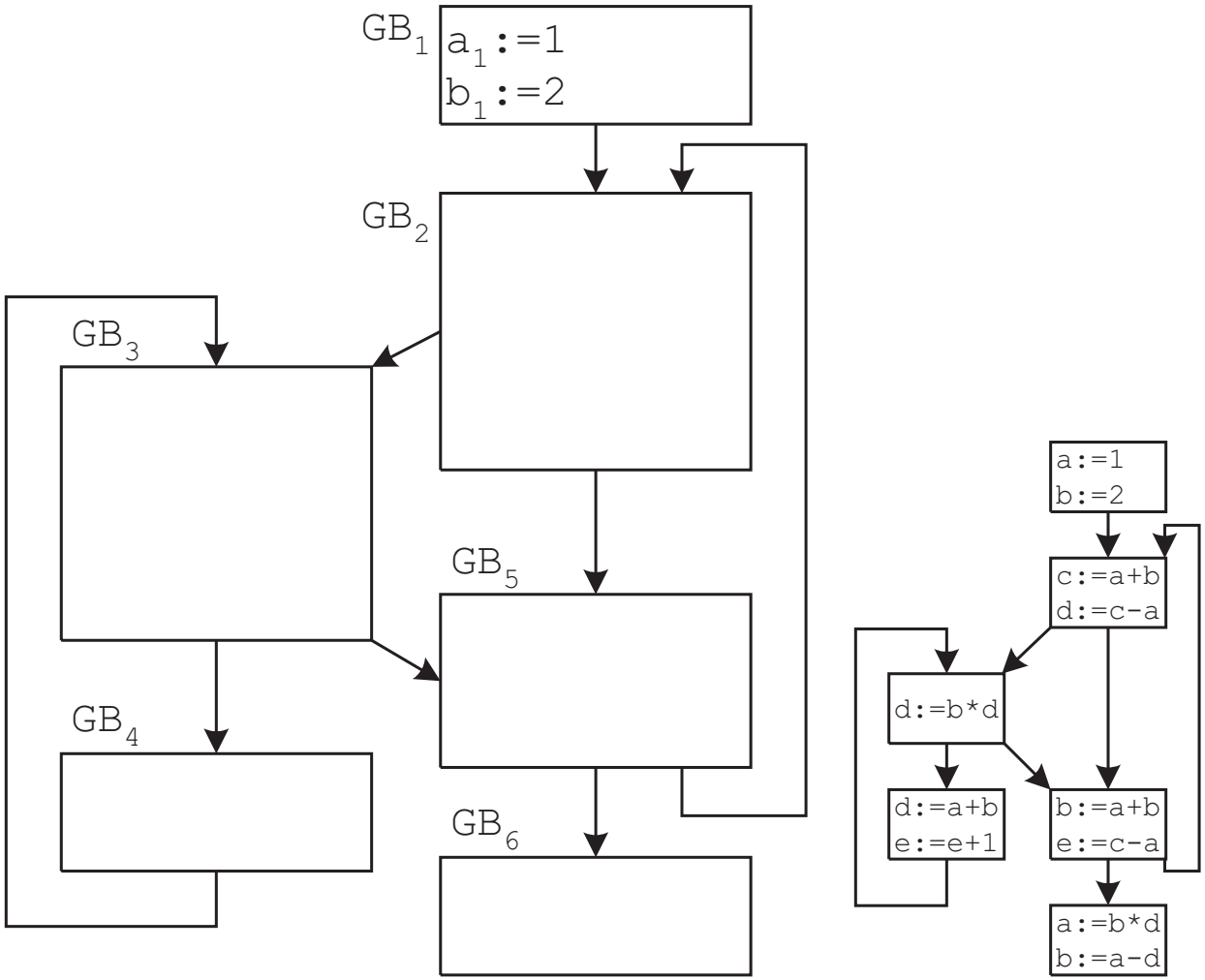


Beispielprogramm und Grundblockgraph

```
 (1) a:=1;
 (2) b:=2;
     while true {
 (3)   c:=a+b;
 (4)   if (d=c-a)
 (5)     while (d=b*d) {
 (6)       d:=a+b;
 (7)       e:=e+1;
       }
 (8)   b:=a+b;
 (9)   if (e=c-a) break;
     }
(10) a:=b*d;
(11) b:=a-d;
```

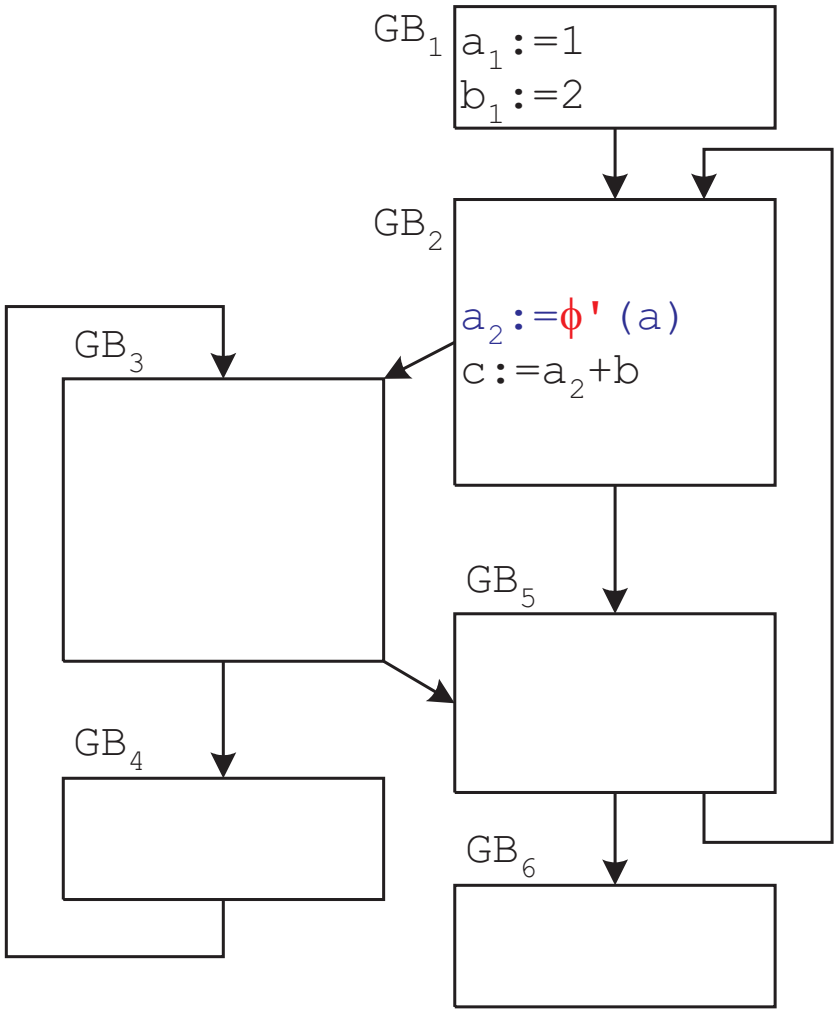


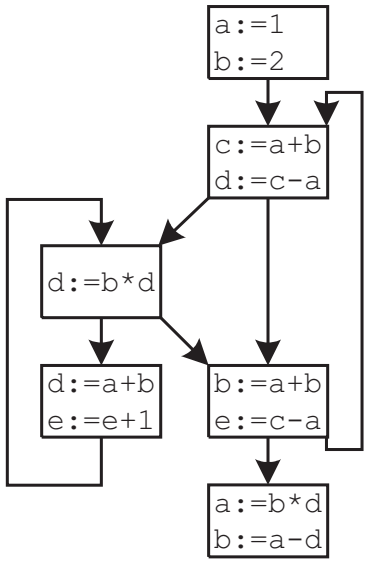
SSA Aufbau Block 1



SSA Aufbau Block 2

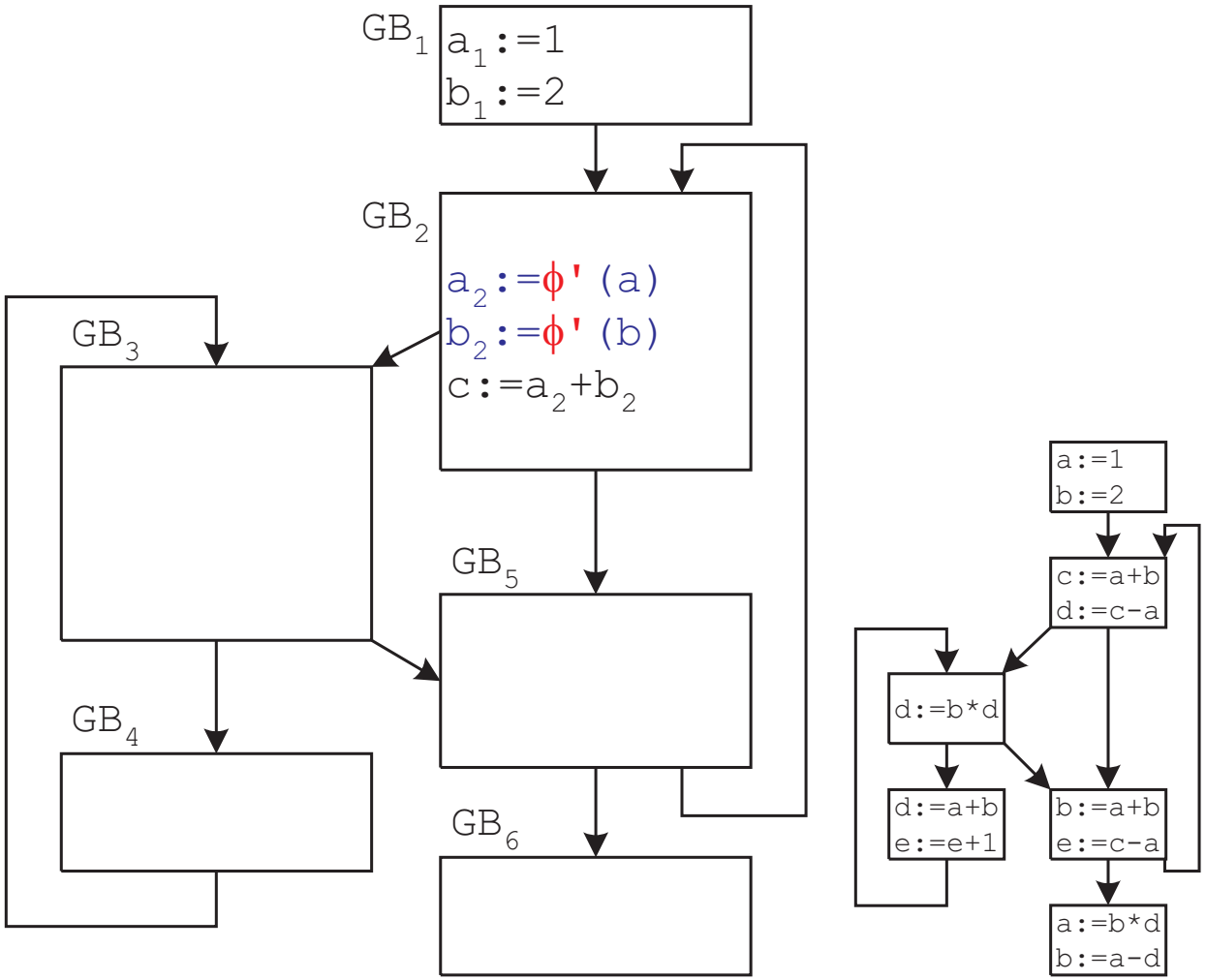
Holen der *wn* für *a* erzeugt zuerst ϕ' für *a* ...





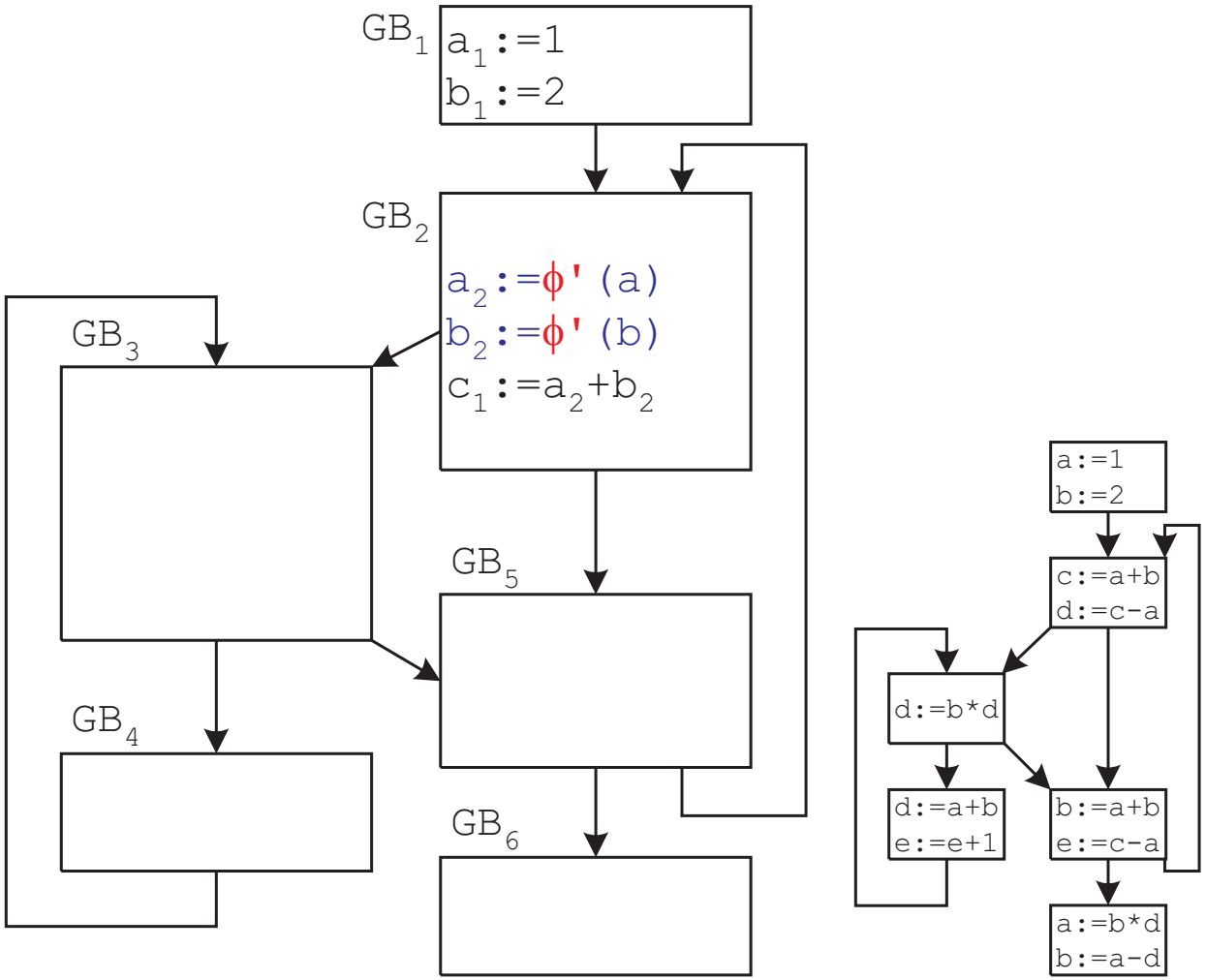
SSA Aufbau Block 2

. . . dann für b . . .



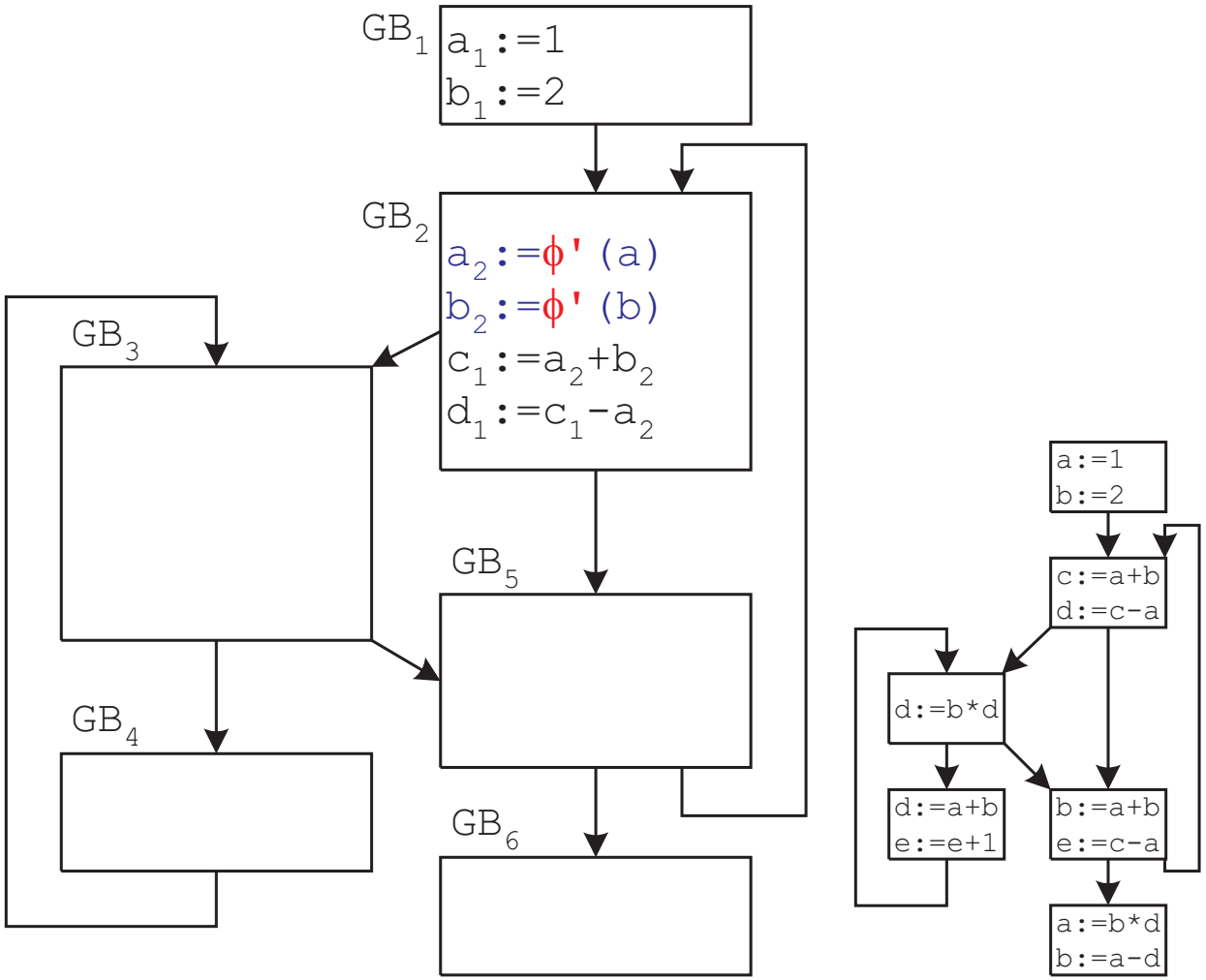
SSA Aufbau Block 2

. . . und schließlich eine *wn* für *c*.

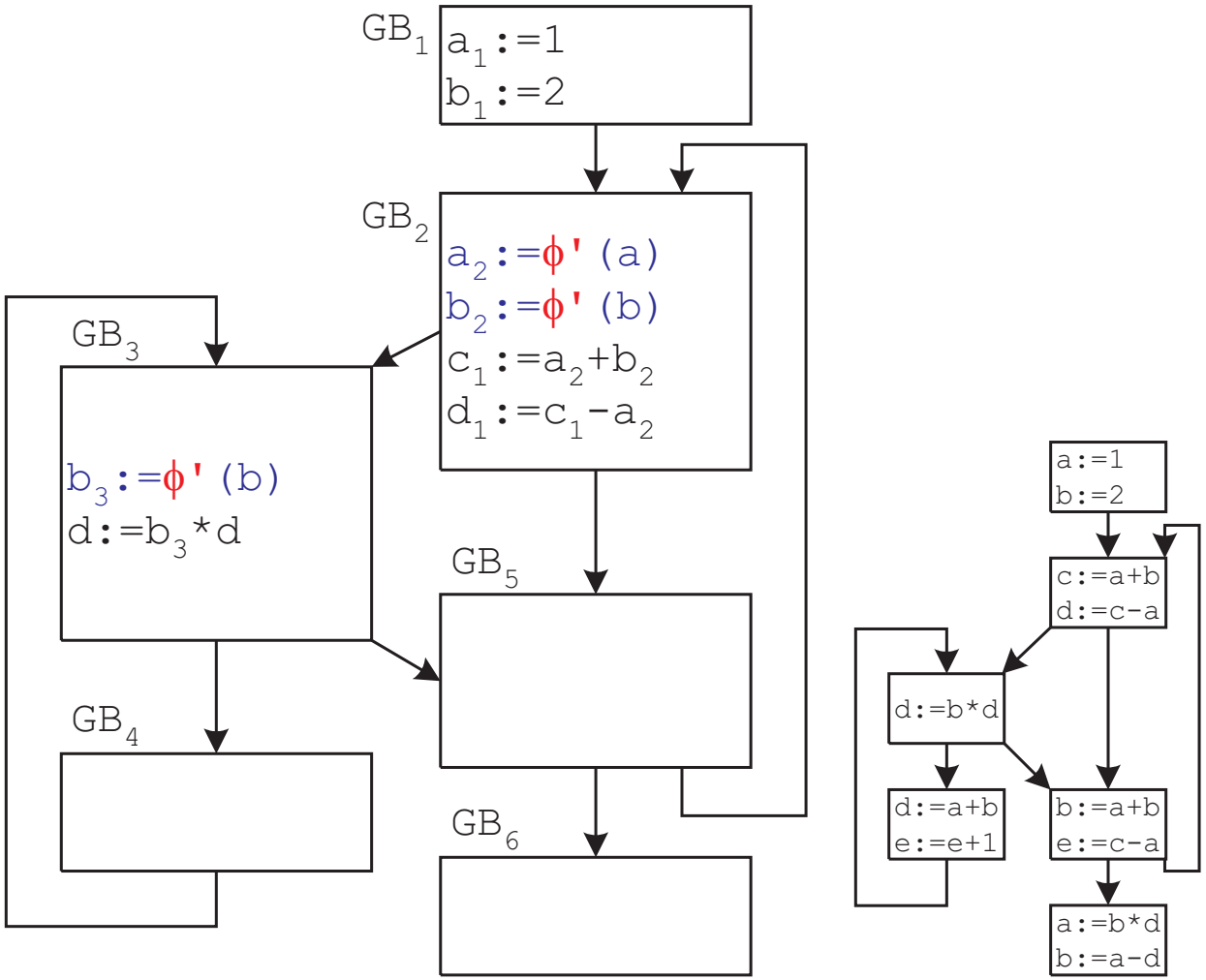


SSA Aufbau Block 2

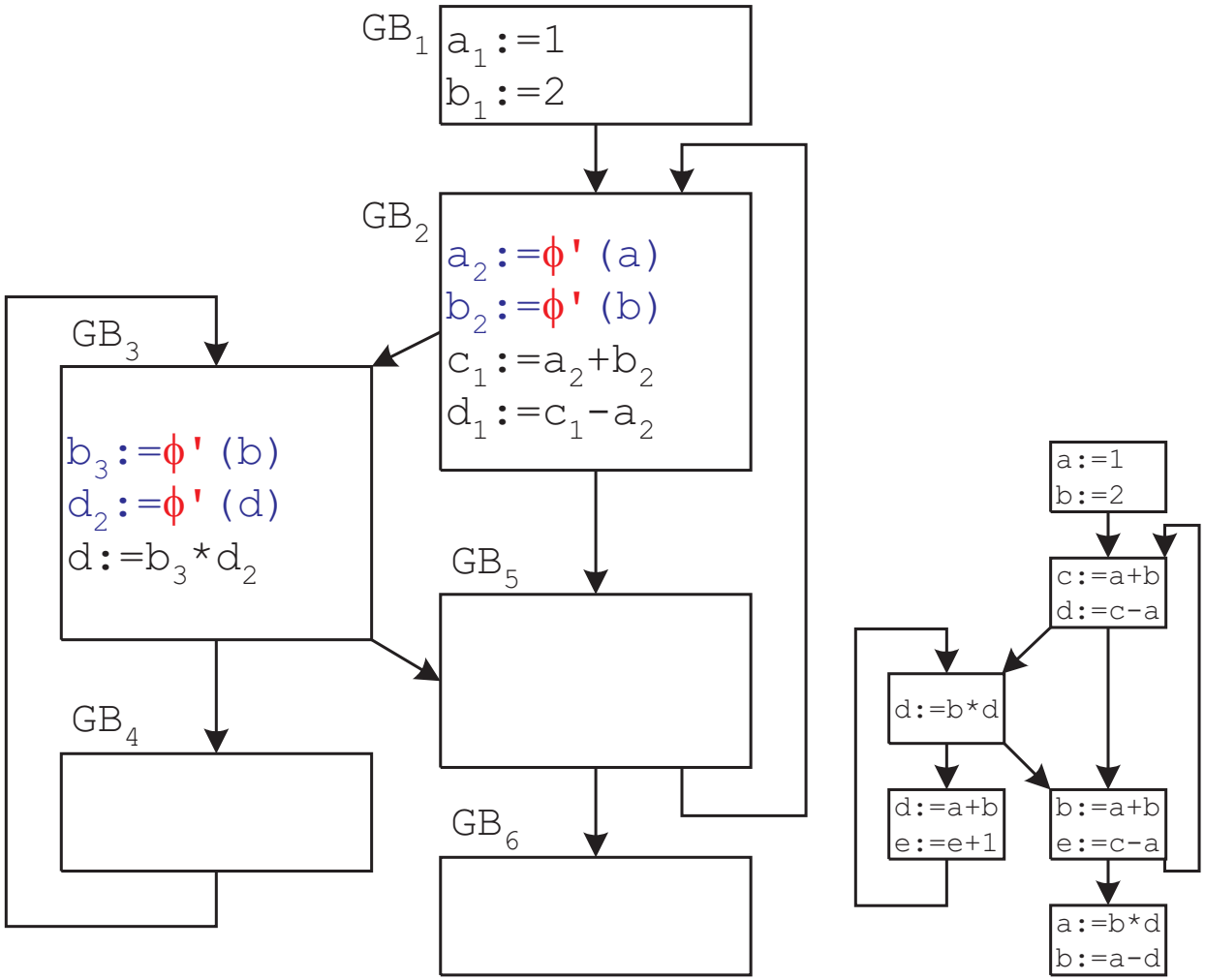
Der Aufbau für $d := c - a$ funktioniert wie normale Wertnumerierung.



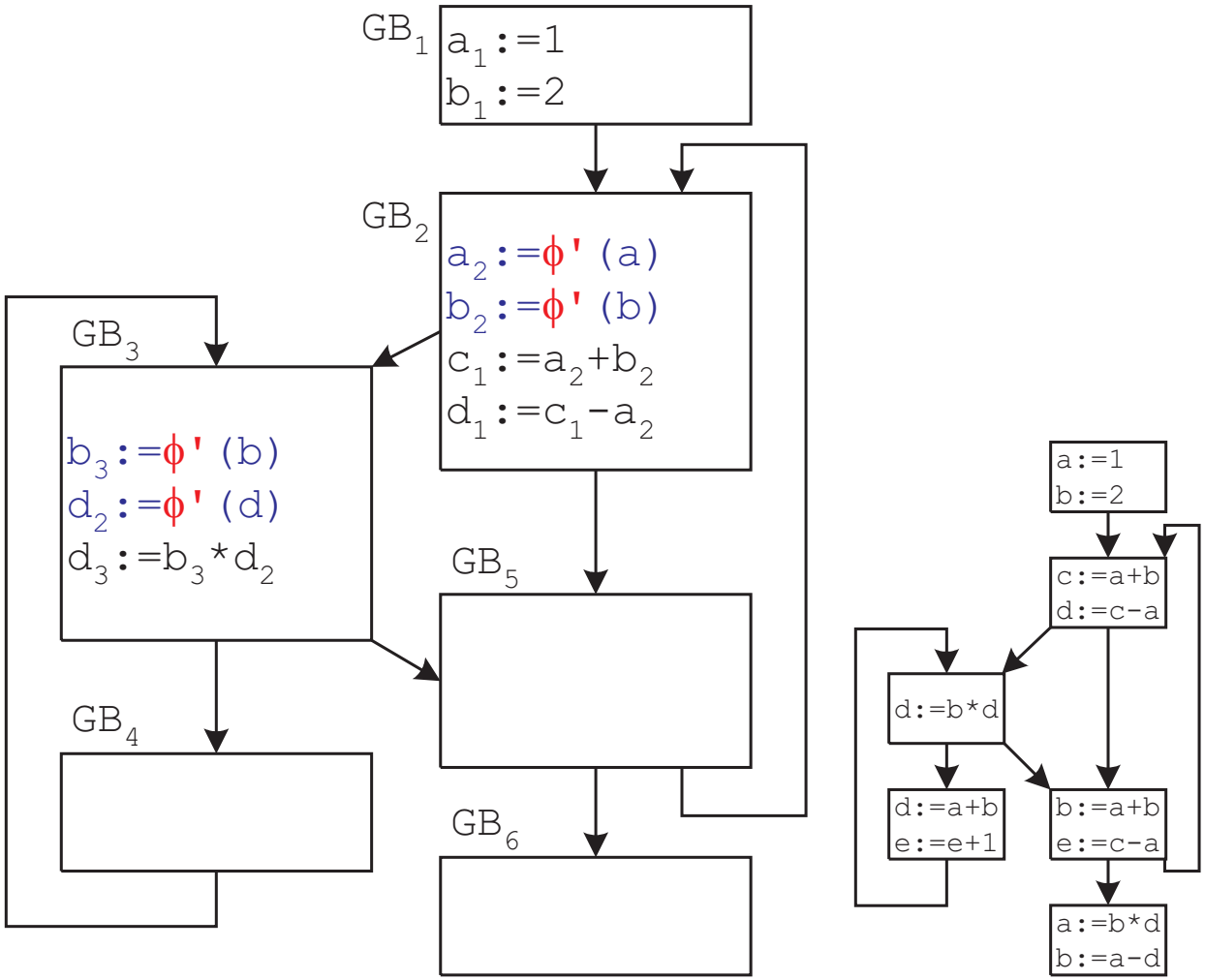
SSA Aufbau Block 3



SSA Aufbau Block 3

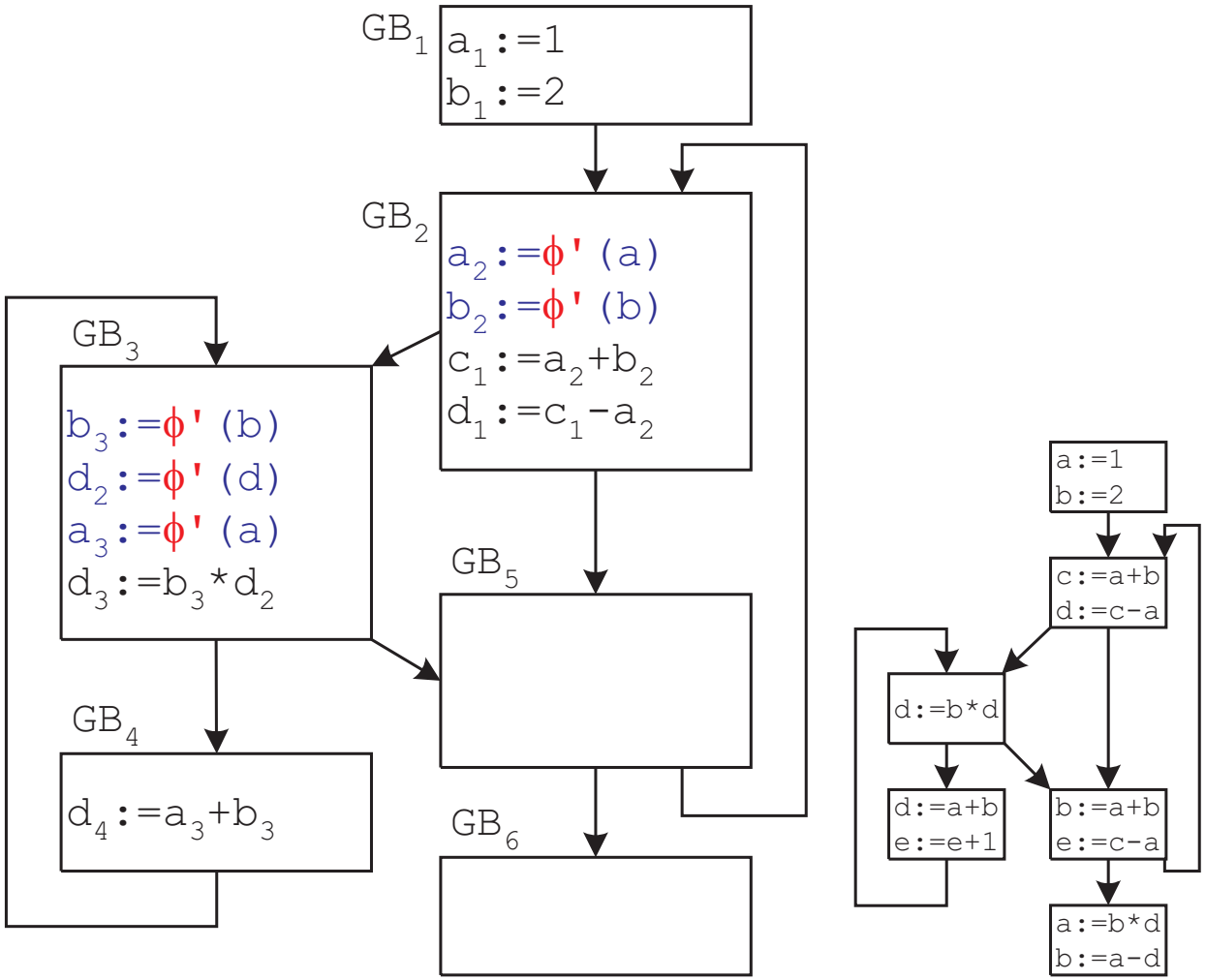


SSA Aufbau Block 3

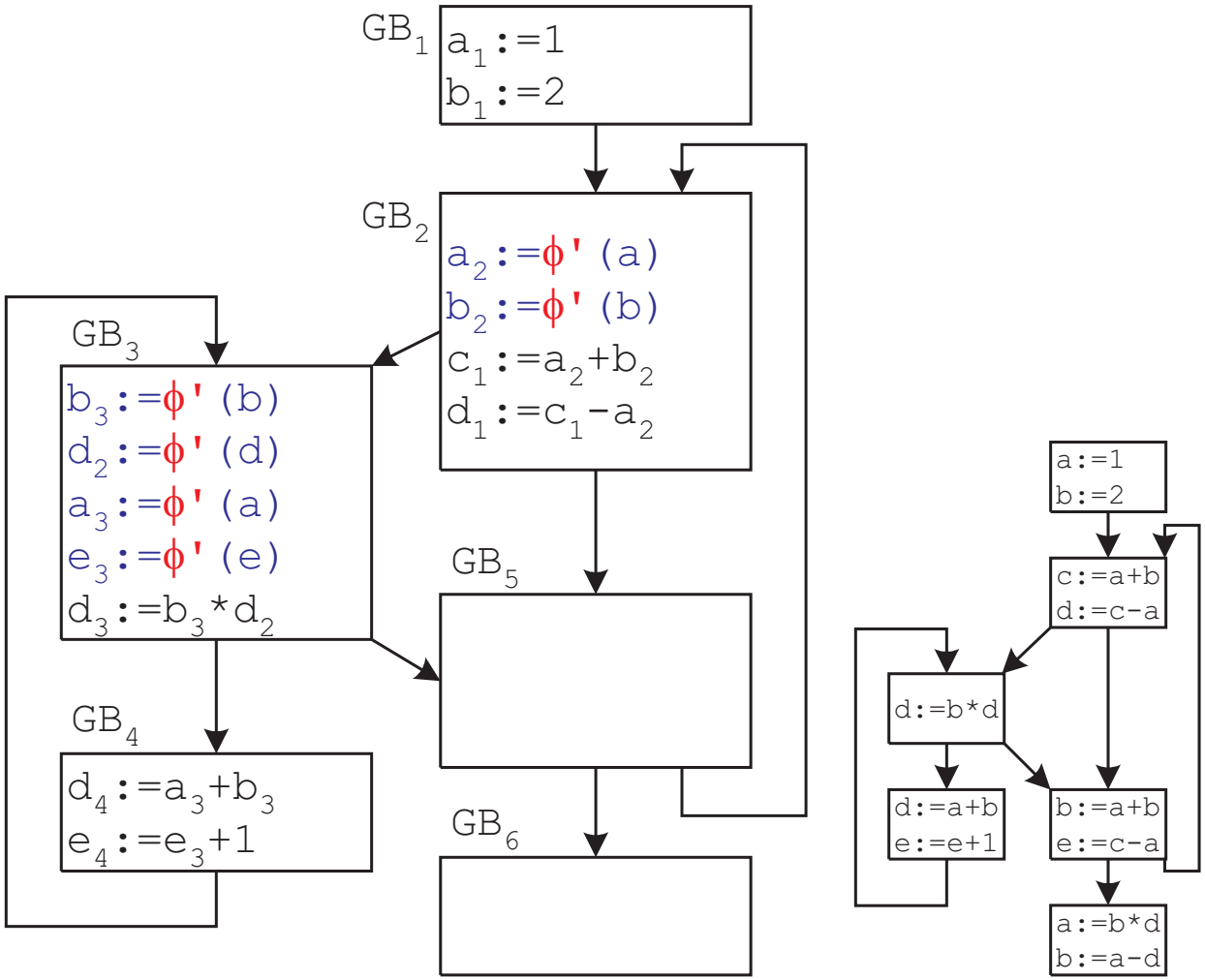


SSA Aufbau Block 4

Der Aufruf $HW(a)$ in 4 führt zu rekursivem Aufruf $HW(a)$ in 3.
Dieser erzeugt in 3 eine neue ϕ'-Funktion für a.



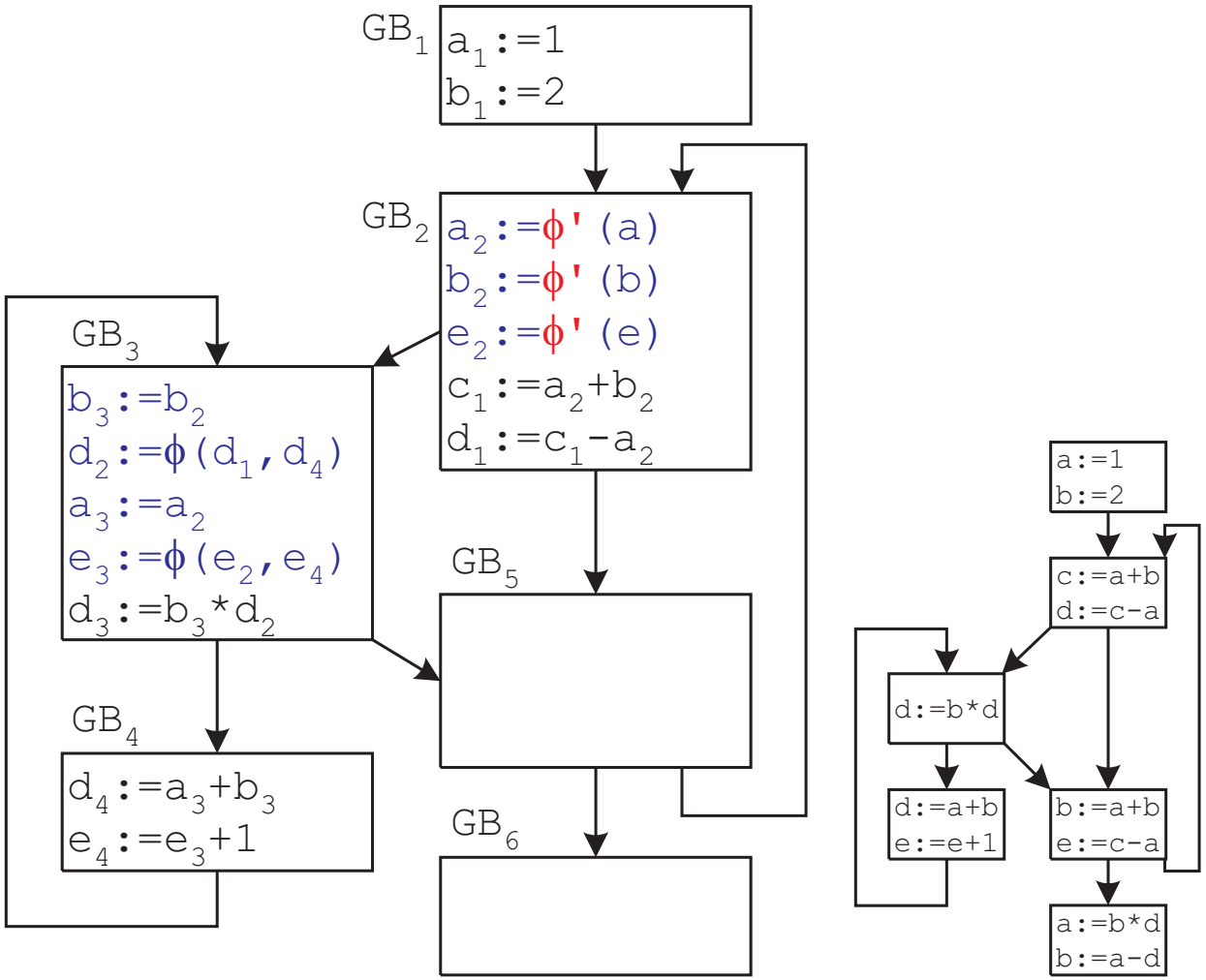
SSA Aufbau Block 4



SSA Aufbau Block 4

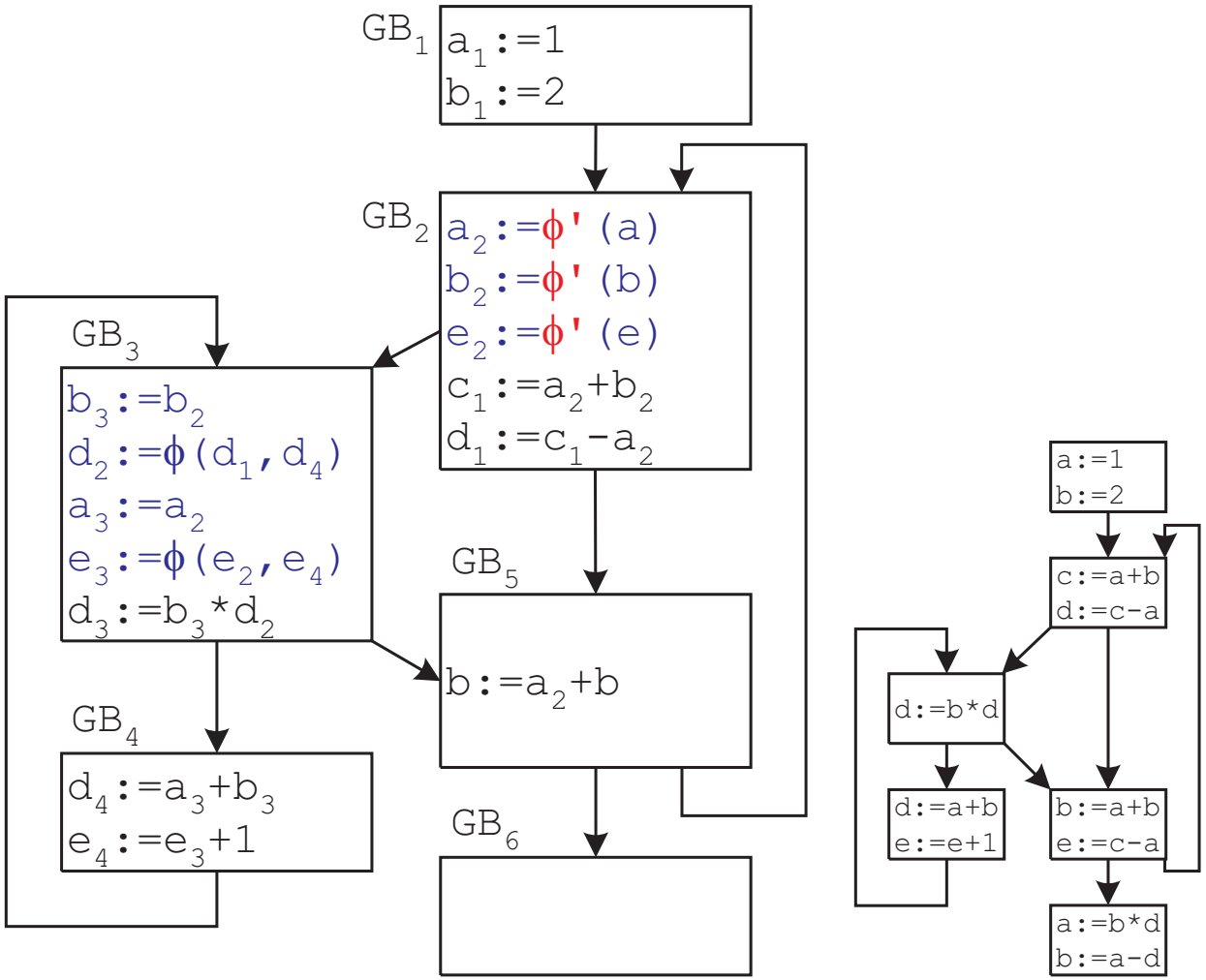
Jetzt alle Vorgänger von Block 3 in SSA Form: ϕ-Funktionen werden berechnet.

Für e wird rekursiv eine ϕ'-Funktion in Block 2 eingesetzt.

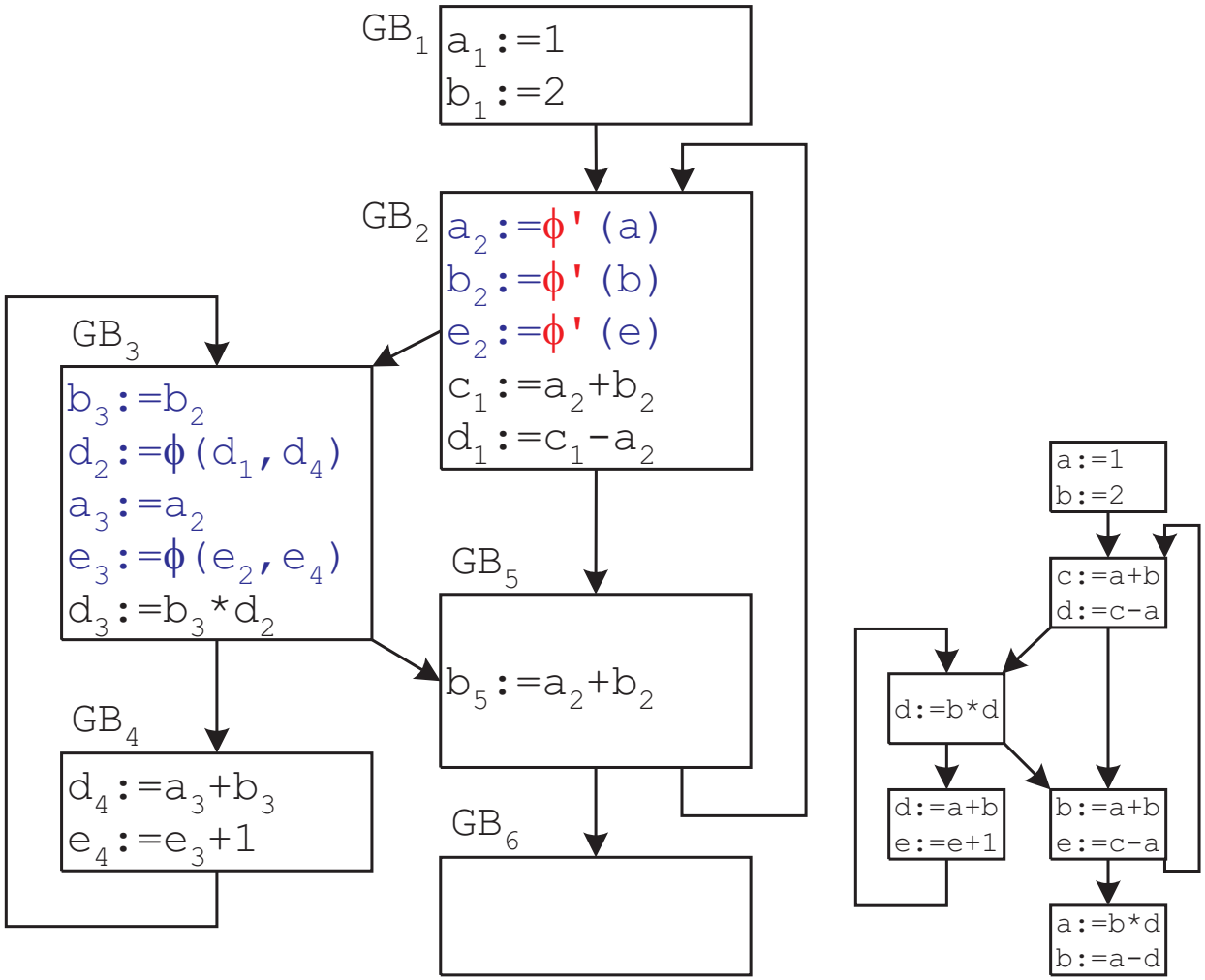


SSA Aufbau Block 5

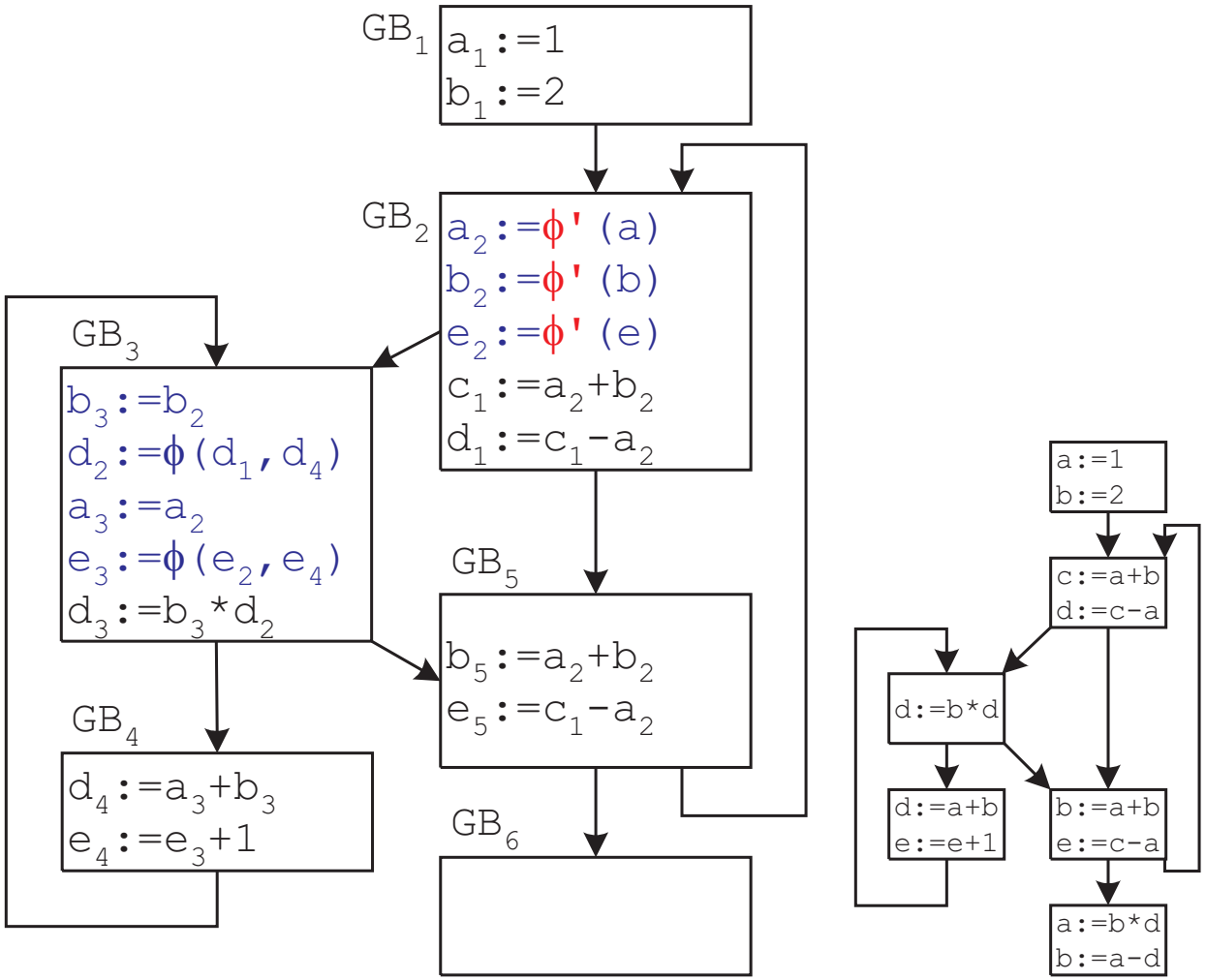
$HW(a)$ in 5 überspringt Kopien, findet eindeutige Definition:
keine ϕ-Funktion nötig.



SSA Aufbau Block 5



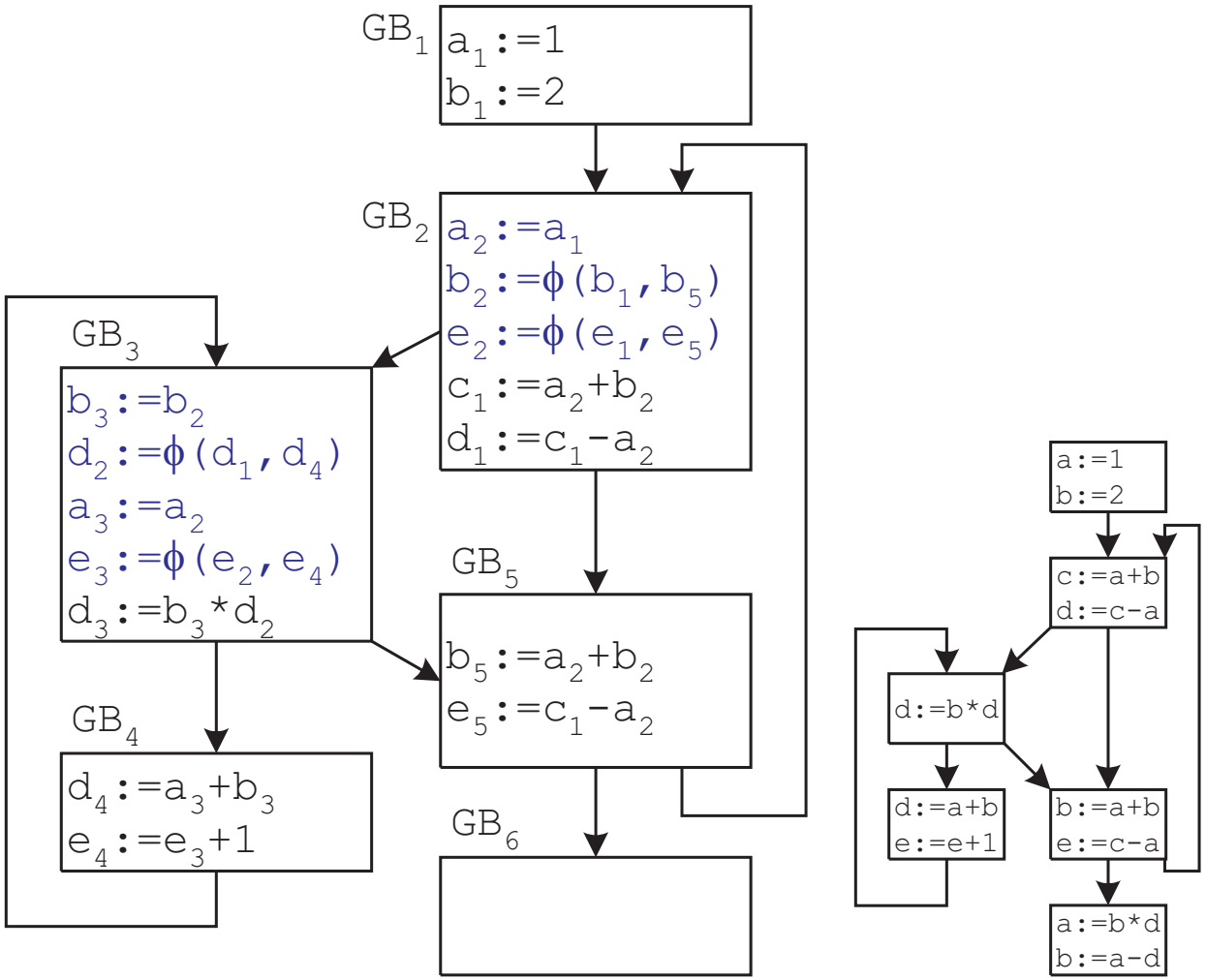
SSA Aufbau Block 5



SSA Aufbau Block 5

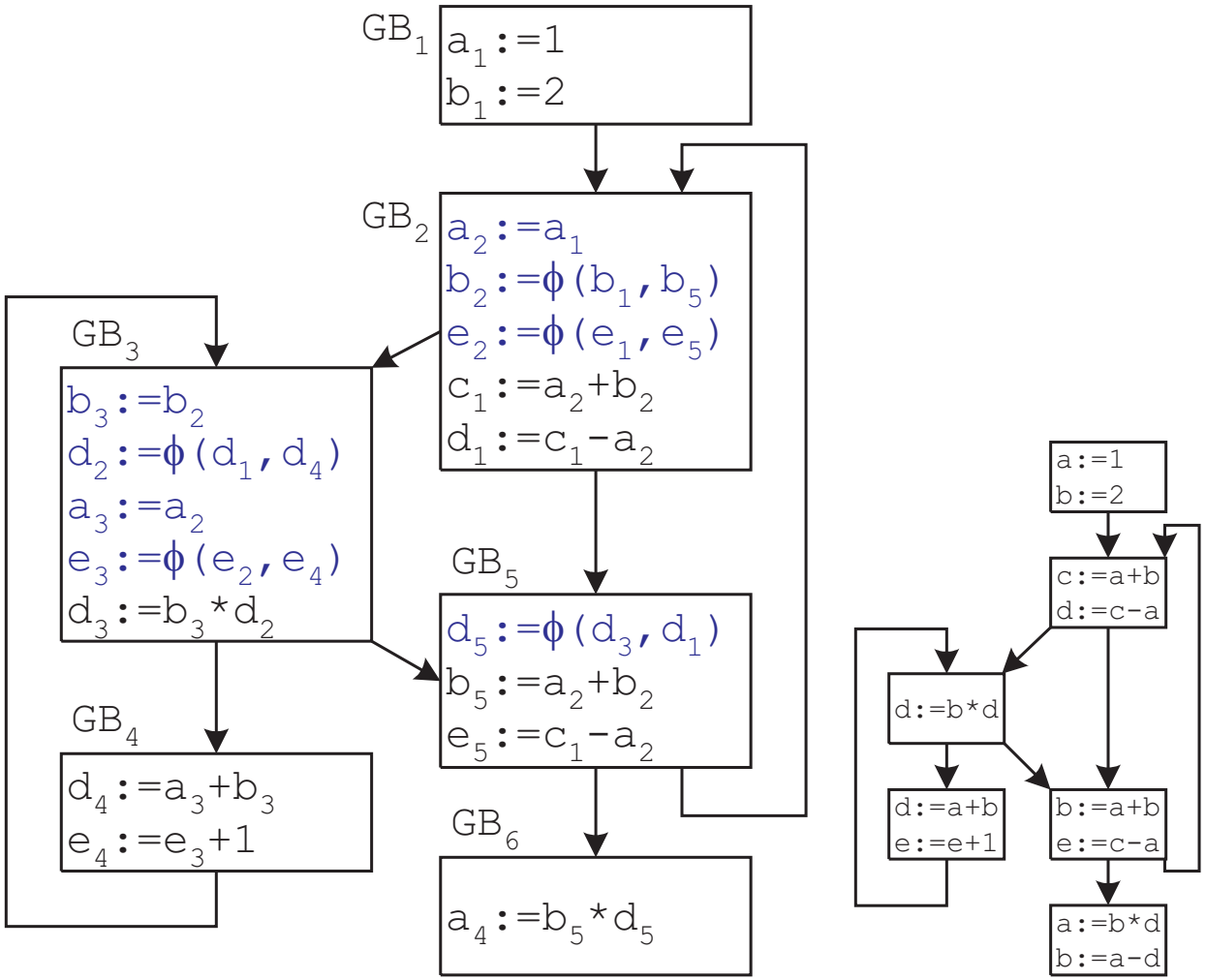
Jetzt alle Vorgänger von Block 2 in SSA Form: ϕ-Funktionen werden berechnet.

Algorithmus bemerkt:
e ist uninitialisiert!
Annahme: Wert e_1

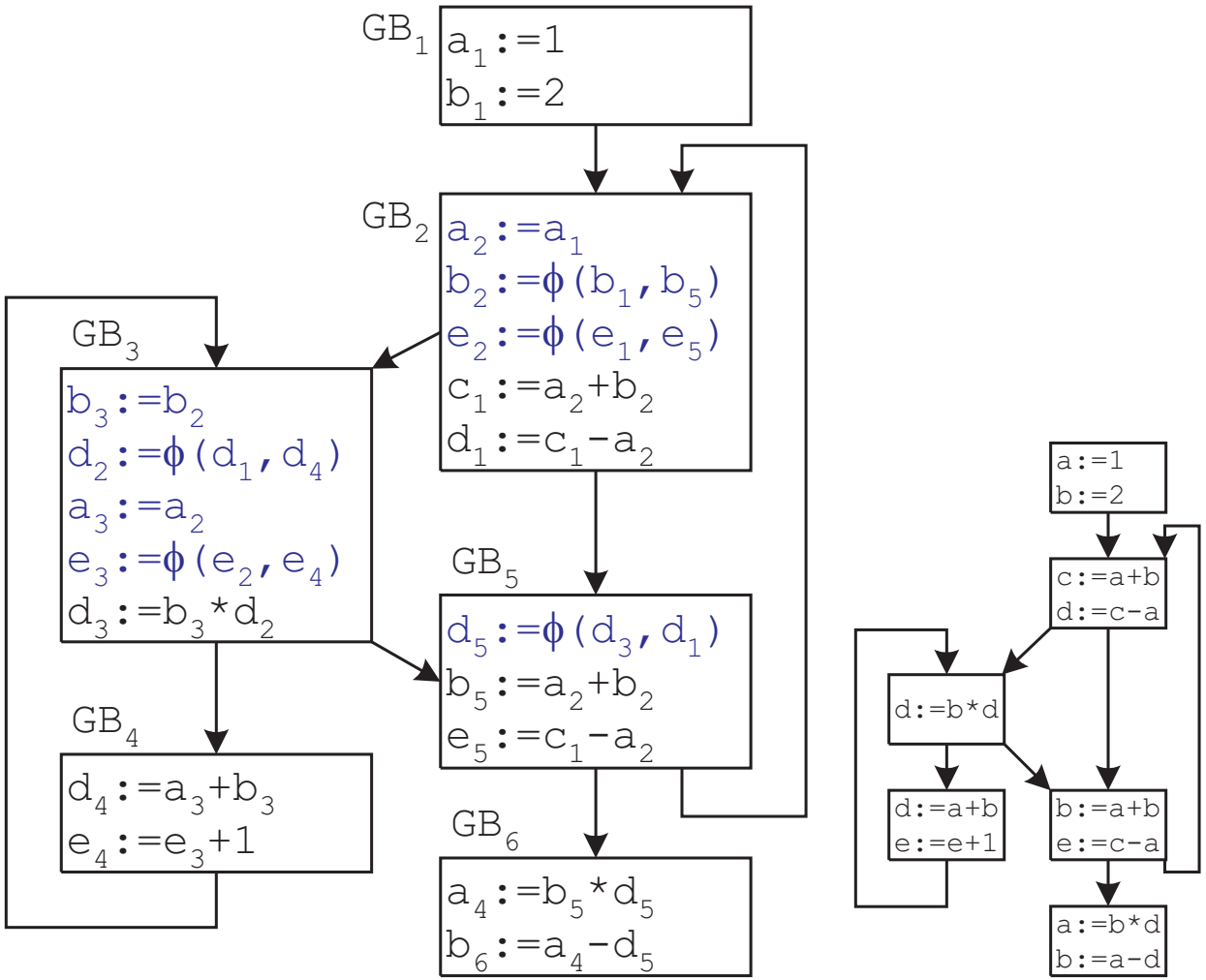


SSA Aufbau Block 6

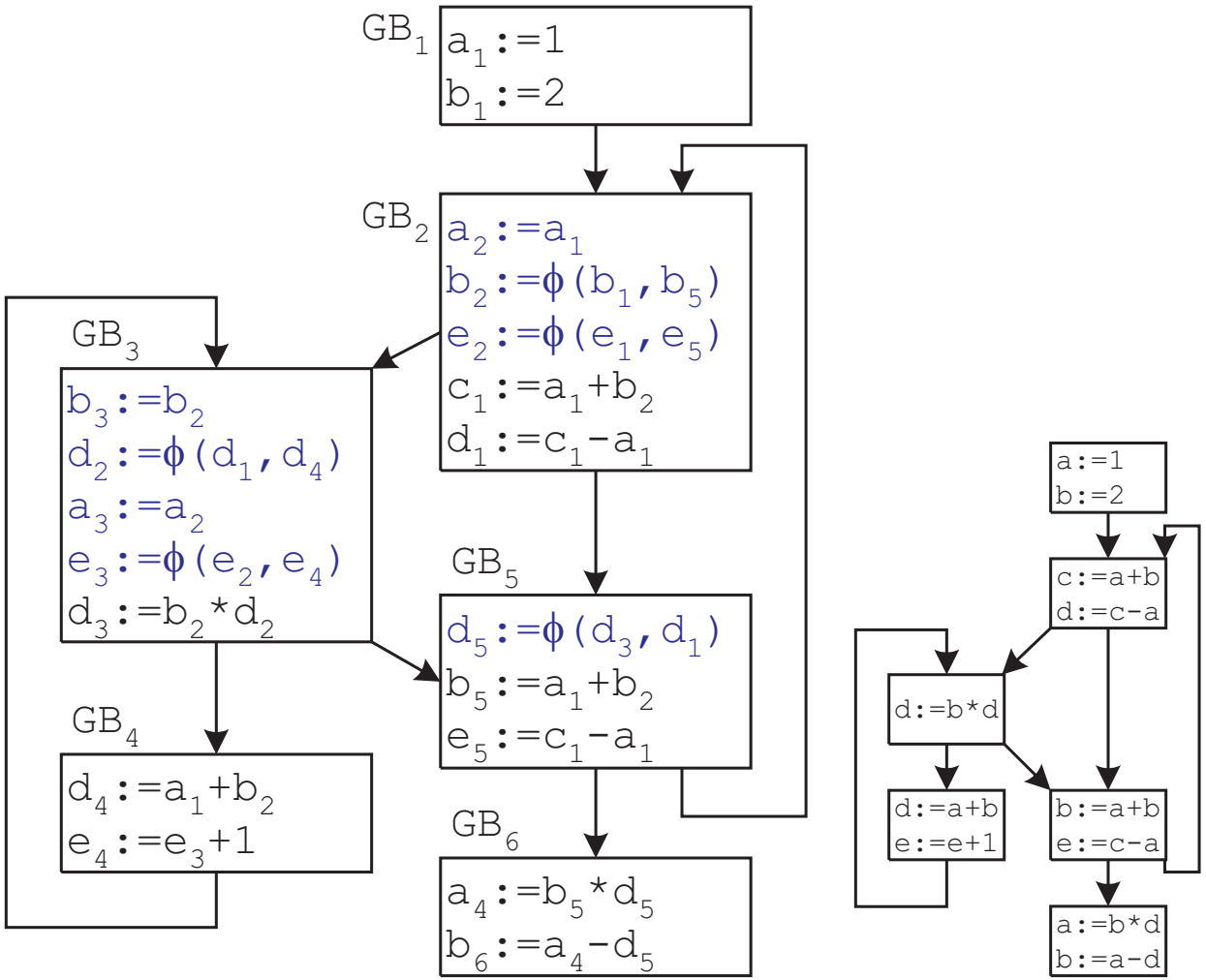
Rekursiver Aufruf von $HW(a)$ in 5 setzt komplette ϕ-Funktion d_5 ein



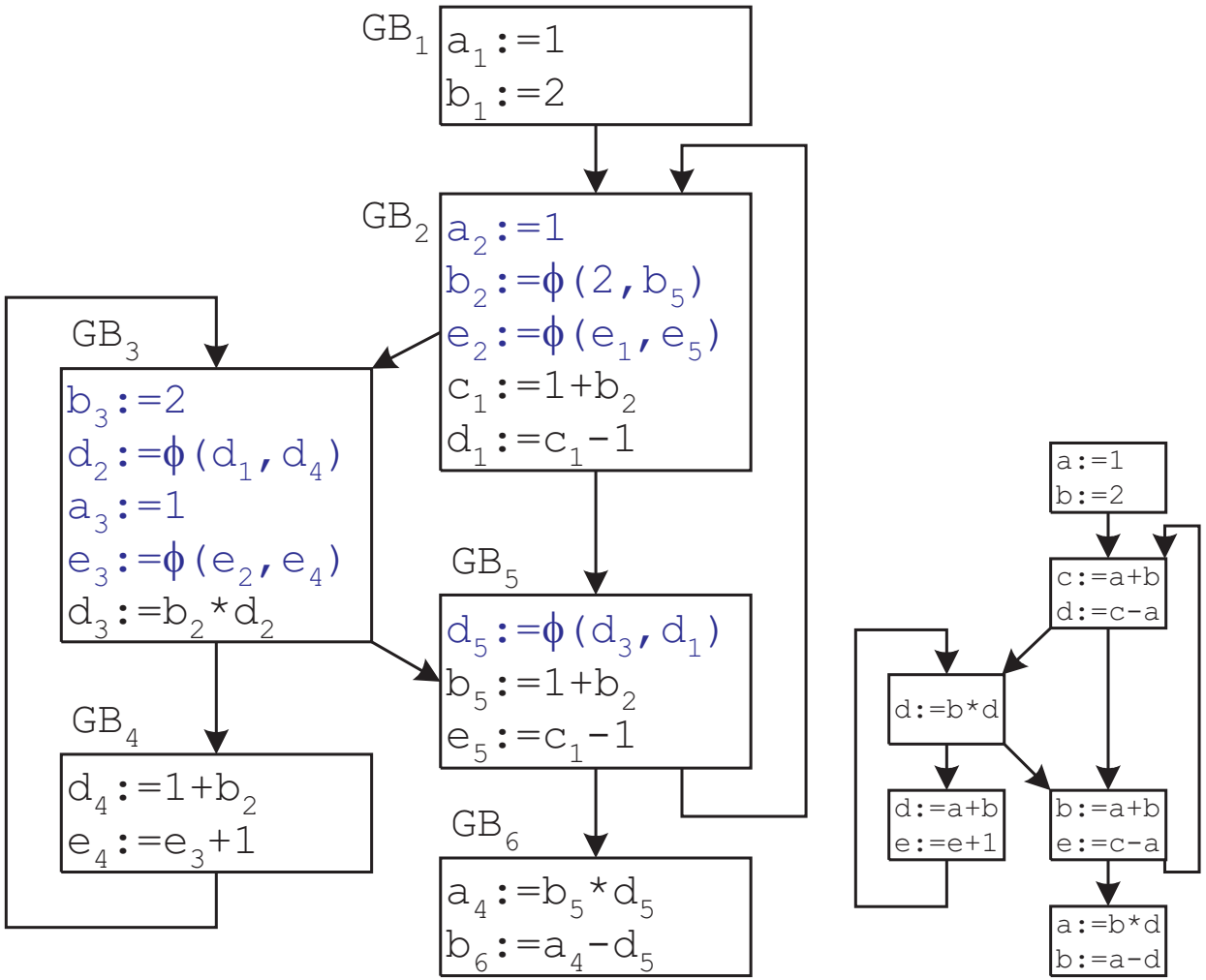
SSA Aufbau Block 6



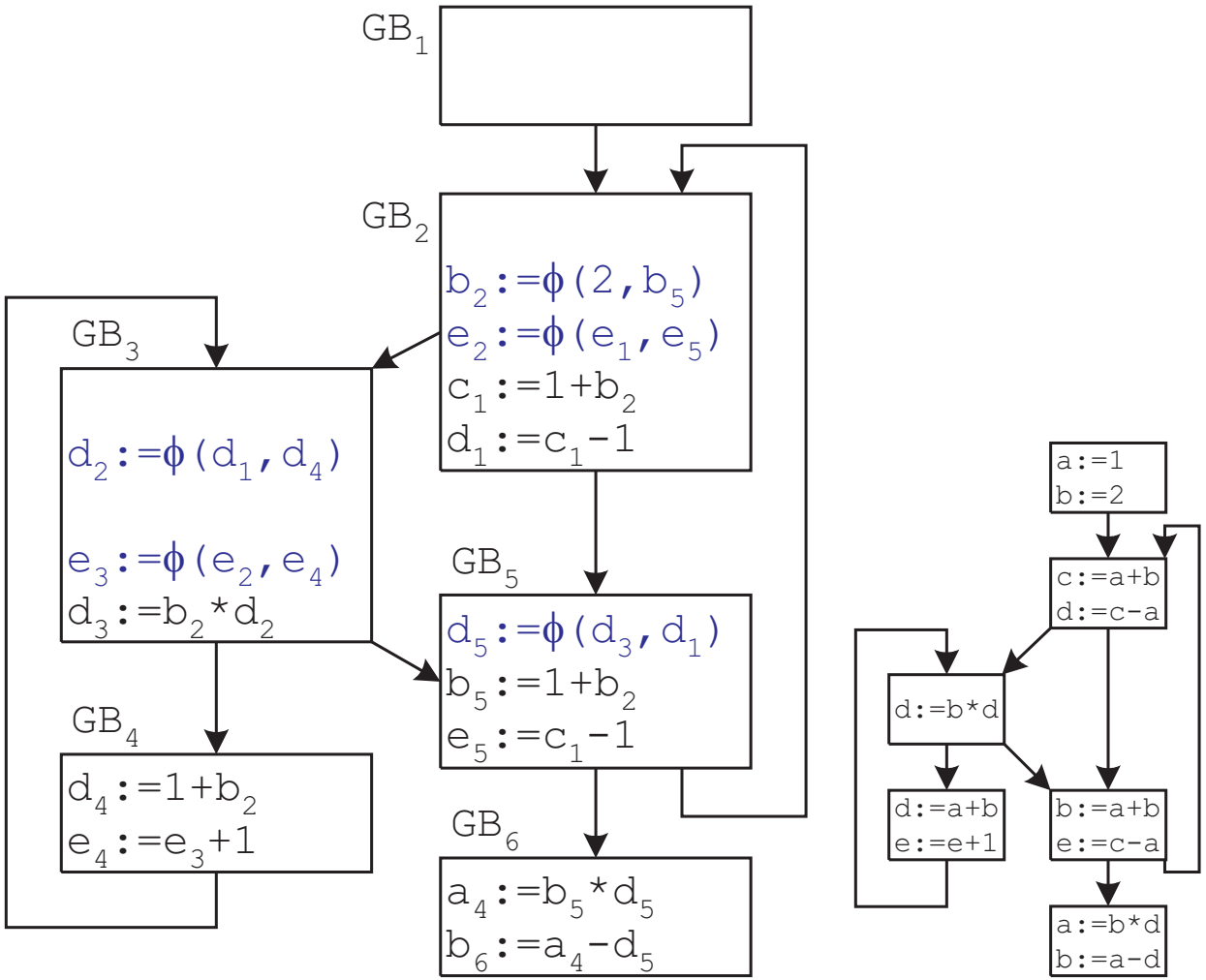
Vereinfachungen: Kopienfortpflanzung



Vereinfachungen: Konstantenfortpflanzung

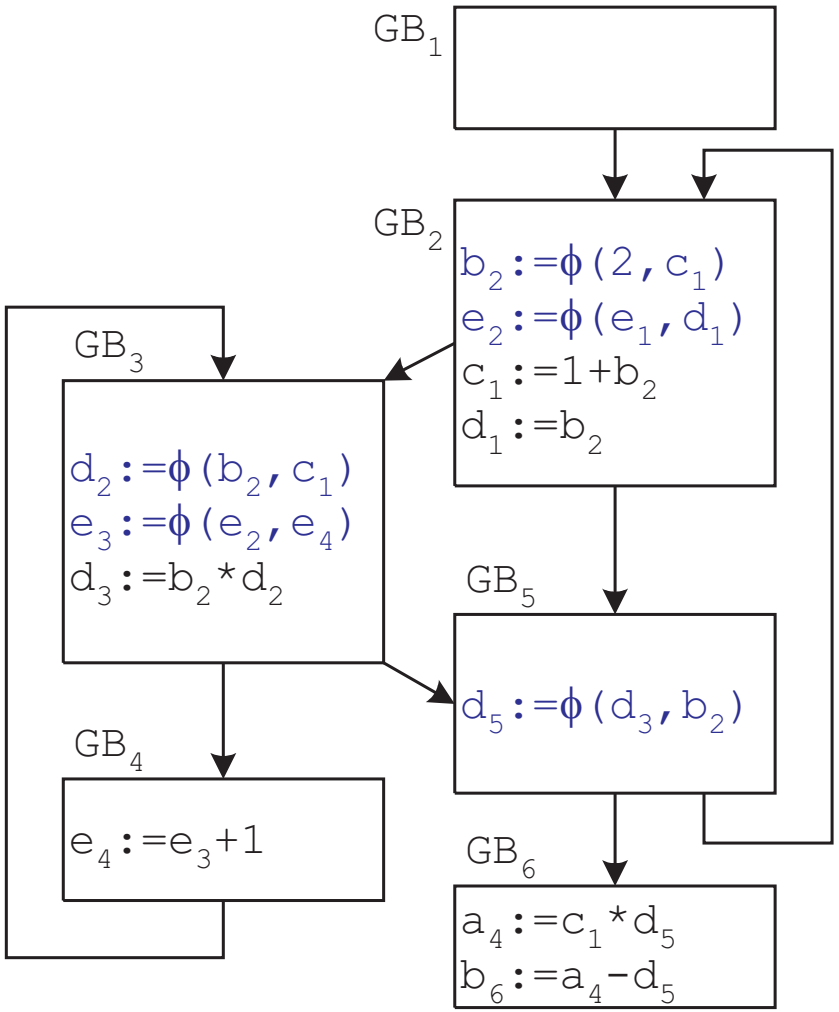


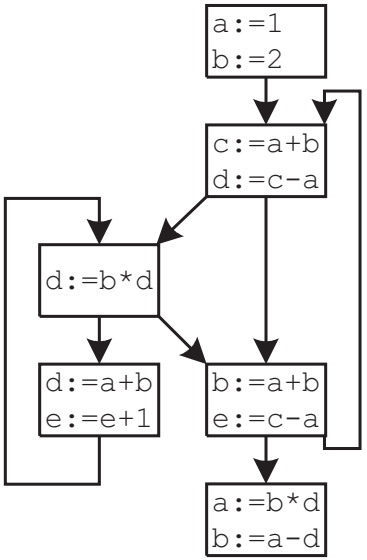
Vereinfachungen: Toten Code eliminieren



Weitere Vereinfachungen

- Gemeinsame Teilausdrücke
- Reassoziation
- konstante Ausdrücke auswerten
- Kopien fortschreiben
- Toten Code eliminieren





SSA-Aufbau aus dem AST

Ein Links-Rechts Baumdurchlauf:

- Halte aktuellen Grundblock in globaler Variablen
- Ausdrücke: Generiere SSA für aktuellen Grundblock, Zwischenergebnisse werden nur einmal verwendet, daher kein Holen/Merken von Wertnummern nötig!

Anweisungen:

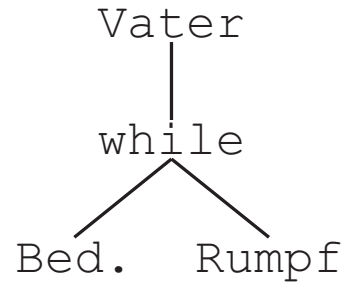
- generiere Grundblöcke
- generiere Code für Grundblöcke
- Füge Ablauf der Grundblöcke zusammen
- Schließe SSA Aufbau für die Grundblöcke ab

Prozeduraufrufe sind in diesem Zusammenhang Ausdrücke.
Bei Sprüngen: Schließe Grundblöcke mit Sprungmarken in einem zweiten Baumdurchlauf ab.

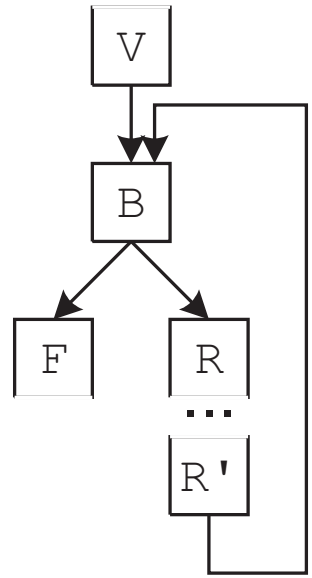
Aufbau aus dem AST, Beispiel while

- Schließe SSA Aufbau für aktuellen Block (V) ab
- Erzeuge neuen Block (B) für Schleifenbedingung
- Füge Ablauf $V \rightarrow B$ ein
- Erzeuge SSA-Code für Bedingung (rekursiver Abstieg)
- Erzeuge neuen aktuellen Block (R) für Schleifenrumpf, merke Block B
- Erzeuge SSA-Code für Rumpf (rekursiver Abstieg)
- Schließe SSA-Aufbau für aktuellen Block (R') ab
- Füge Ablauf $R' \rightarrow B$ ein
- Schließe SSA-Aufbau für Block B ab
- Erzeuge neuen Block (F) für Fortsetzung
- Füge Ablauf $B \rightarrow F$ ein
- Rekursion kehrt zu Vater zurück und generiert weiteren Code in Block F

AST Ausschnitt:



generierter Ablauf:



Zusammenfassung

- SSA bedeutet: statt Variablen dynamische Konstanten
- ϕ-Funktionen nötig bei Ablaufzusammenfluß
- ϕ-Funktionen werden an Dominanzgrenzen plaziert
- Darstellung als Datenflußgraph
- Ermöglicht effiziente Formulierung intraprozeduraler Optimierungen die auf Datenflußanalysen aufbauen
- Aufbau der ϕ-Funktionen: nur, wenn Wert verwendet; rekursiv für Vorgängerblöcke
- Endlosrekursionen vermeiden und Handhabung nicht fertiger Vorgängerblöcke mit ϕ'-Funktionen

Optimierungen auf SSA-Form

Inhalt

Optimierungen (nicht notwendigerweise auf SSA)

- Fortschreiben v. Konstanten u. Kopien, intra- und interprozedural
- Konstanten Falten (Auswerten konstanter Ausdrücke)
- Beseitigen von totem/unerreichbarem Code
- Operatorvereinfachung (Beseitigen von Induktionsvariablen)
- Verschieben von schleifeninvariantem Code
- Eliminieren partieller Redundanzen
- Codeverschiebung, -plazierung
- Spezialisieren und Klonen von Grundblöcken und Prozeduren
- Eliminieren Indexgrenzenprüfung
- Offener Einbau von Prozeduren
- Ablauf-Vereinfachung
- Ausrollen/Verschmelzen/Teilen von Schleifen; Software-Fließband
- Umordnen von Reihungen und -zugriffen (D-Cache-Optimierungen)
- Vorladen von Daten (*prefetching*)
- Optimieren von Blatt-Prozeduren
- Beseitigen von End-Aufrufen und Endrekursionen
- Registerzuteilung

Stand der Forschung

- Offenes Problem:
  - Welche Optimierungen auszuwählen?
  - Optimale Reihenfolge der Optimierungen?
- Es gibt keine aktuellen wissenschaftliche Aussagen zu einer solchen Taxonomie!
- Faustregel:
  - Abgesehen von Cacheoptimierung und Operatorvereinfachung bringt die 1. durchgeführte Optimierung 15% alle weiteren $< 5\%$.
  - Das ist weitgehend unabhängig von der Wahl der Optimierungen und ihrer Reihenfolge.
  - Viele Optimierungen bewirken ähnliches bzw. sind ineinander enthalten.
  - Bei numerischen Programmen bringt Operatorvereinfachung Faktoren > 2; Cacheoptimierung zwischen 2 und 5.

Das Vollbeschäftigungstheorem für Übersetzerbauer

- Satz (Rice, 1953): Zu jedem algorithmisch arbeitenden Übersetzer U gibt es einen Übersetzer U', der für bestimmte Programme kürzeren Code erzeugt.
- Korollar (Vollbeschäftigungstheorem für Übersetzerbauer): Zu jedem optimierenden Übersetzer gibt es einen besseren.
- Beweis des Satzes:
  Annahme: es gibt einen Übersetzer U, der jedes Programm π mit algorithmischen Methoden in das absolut kürzeste Programm $Opt(\pi)$ mit gleichem Ein-/Ausgabeverhalten übersetzt. Sei π ein nicht haltendes Programm ohne E/A. Dann wird π von U übersetzt in:

$$Opt(\pi) \mapsto m : \text{goto } m$$

  Um zu prüfen, ob π nicht hält, muß man nur $Opt(\pi)$ berechnen und das Ergebnis inspizieren. Damit löst man also das Halteproblem. Da das Halteproblem unentscheidbar ist, kann es also ein solches U nicht geben.

Prinzip von Kostenmodellen bei Optimierungen

Kostenmodell: Laufzeit eines Programms, eventuell kombiniert mit Energieverbrauch

- Speicherbedarf kann in Laufzeit umgerechnet werden
  - daher ist oft auch Verkürzung des Codes Laufzeitoptimierung
- Statisch nur konservativ abschätzbar wegen Unkenntnis der
  - Anzahl Wiederholungen von Schleifen
  - Sprungbedingungen
- Selbst bei linearem Code nicht statisch bekannt
  - Befehlsanordnung durch Prozessor
  - Pufferspeicher (Cache): Zugriffe auf Speicher sind datenabhängig
  - Fließbandverarbeitung im Prozessor

Laufzeit gewöhnlich $\neq$ Summe der Laufzeiten der einzelnen Befehle!

Optimierungen auf SSA

Optimierungen

- sind auf SSA mit wenig Analyseaufwand durchführbar
- können oft mit SSA-Aufbau verschränkt werden
- können von weiteren Programmanalysen profitieren (mehr dazu in folgenden Vorlesungen)

Zwei Arten von Transformationen:

- Normalisierende Transformationen
  - bringen SSA-Graph in definierte Form, um unterschiedlich geschriebene Ausdrücke (syntaktisch) vergleichbar zu machen
  - ermöglichen optimierende Transformationen
  - nutzen z.B. algebraische Identitäten (Äquivalenzoperationen)
    - Assoziativgesetz
    - Distributivgesetz
  - sind eingeschränkt durch
    - Auswertungsreihenfolge (Java)
    - Ausnahmen (Java, Eiffel)
    - Gleitpunktarithmetik (nicht assoziativ, nicht distributiv)
- Optimierende Transformationen

Betrachtete Optimierungen

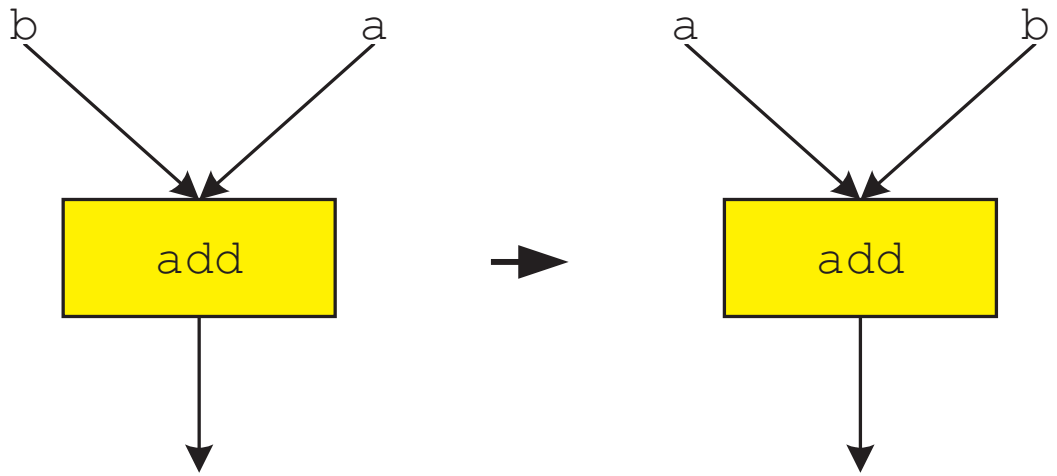
- Operatorvereinfachung (strength reduction)
  - Idee: Ersetze teuere Operationen durch billigere, semantisch äquivalente Operationen
  - Hauptanwendung: Vereinfache Multiplikationen mit Indexvariable in Schleifen zu Additionen (Induktionsanalyse, Lineare Adreßfortschaltung)
- Eliminieren gemeinsamer Teilausdrücke (GTE)
  (common subexpression elimination, CSE)
  - Idee: Eliminiere Mehrfachberechnungen von identischen Werten
  - Hauptanwendung: Adreßrechnung
- Eliminieren partieller Redundanzen (EPR)
  (partial redundancy elimination, PRE)
  schwächere Varianten:Code-Plazierung, Code-Verschiebung
  - Ziel: Vermeide Berechnung von Werten die nicht auf allen Ausführungspfaden gebraucht werden
  - Anwendung: Verschieben von schleifeninvarianten Berechnungen aus Schleifen

Inhalt

Normalisierung - Kommutativgesetz

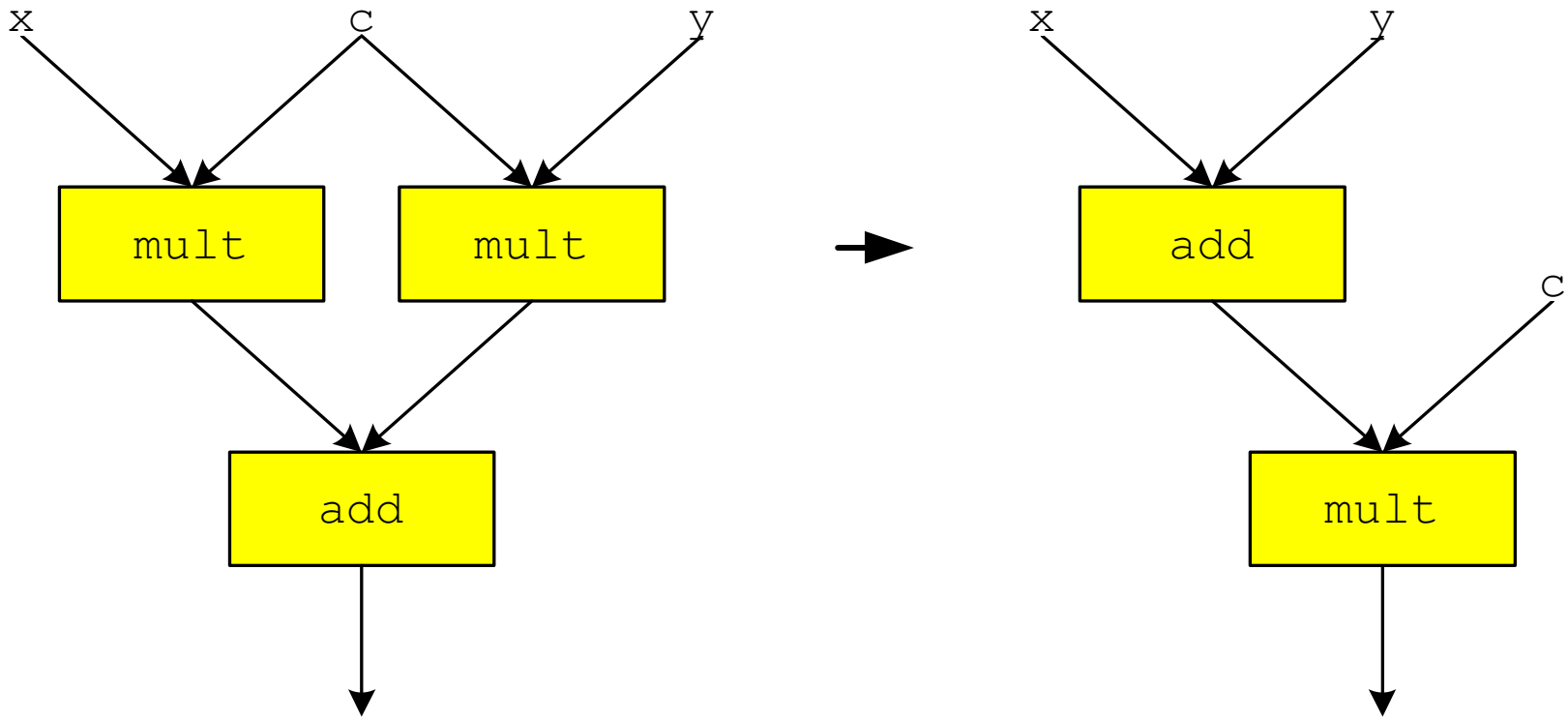
Ziel der Normalisierung: arithmetische Ausdrücke vergleichbar machen, um gemeinsame Teilausdrücke zu finden (Problem: keine kanonische Normalform arithmetischer Ausdrücke)

- Definiere Ordnung B auf SSA-Knoten (z. B. Reihenfolge des Aufbaus)
- Ordne kommutative Operation $\tau(a, b)$ um, so daß $B(a) > B(b)$.



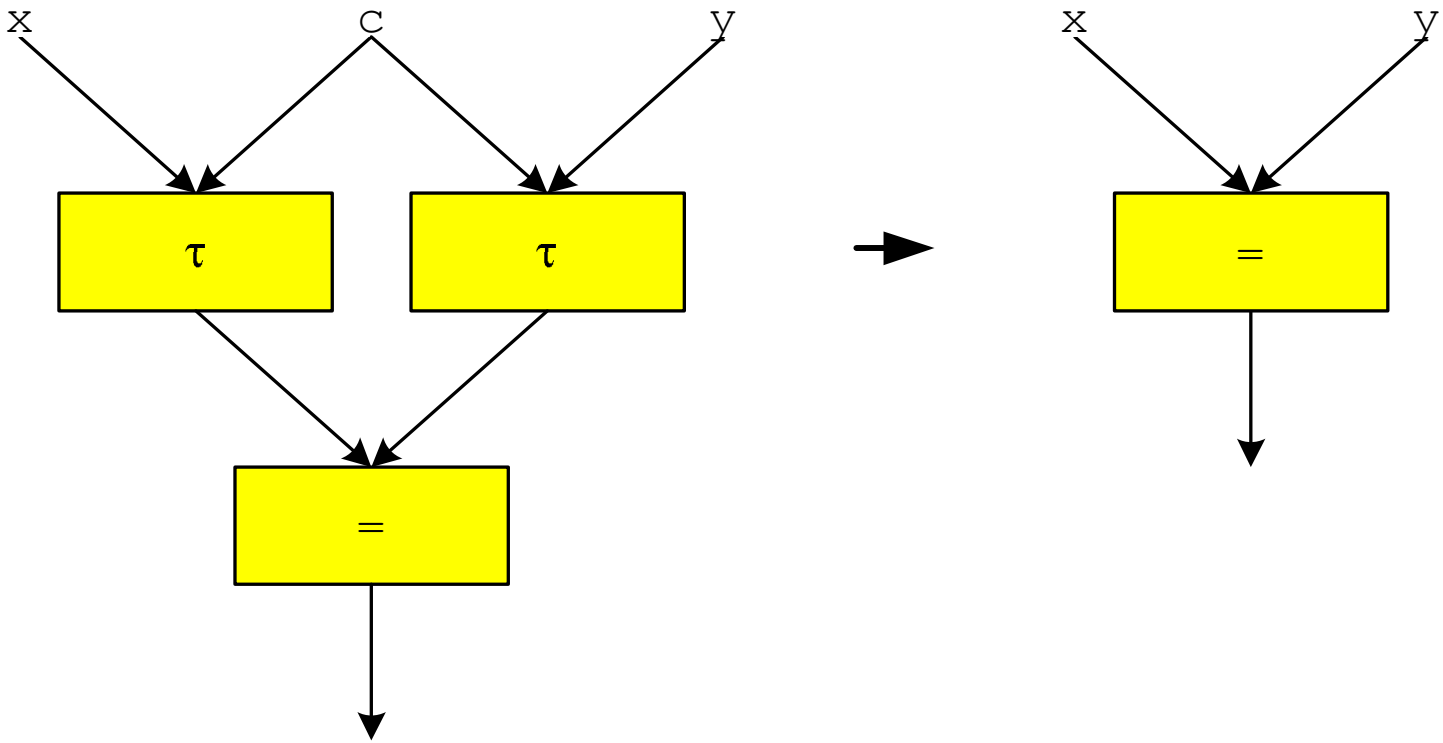
Normalisierung - Distributivgesetz

- Definiere Ordnung B' auf Operationen
- Ordne Operationen $\tau_1 \circ \tau_2$ um, so daß $B'(\tau_1) > B'(\tau_2)$



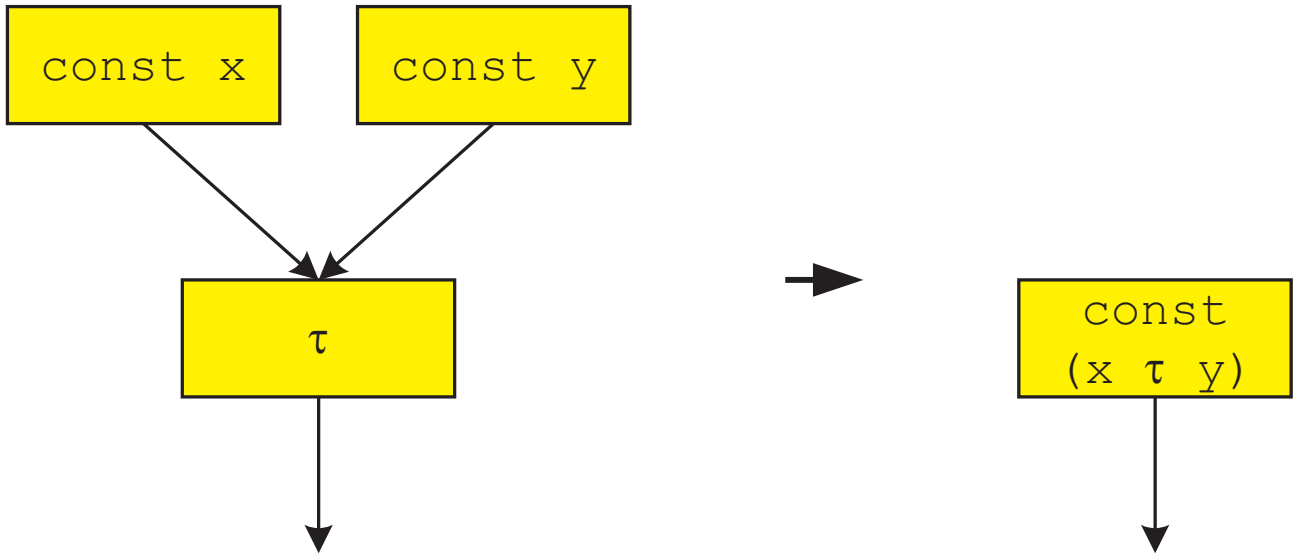
Normalisierung - Vergleiche ändern

- Wenn die Ergebnisse der Operationen $\tau(x, c)$ und $\tau(c, y)$ nur zum Vergleichen benötigt werden, können die Operationen u.U. wegfallen.



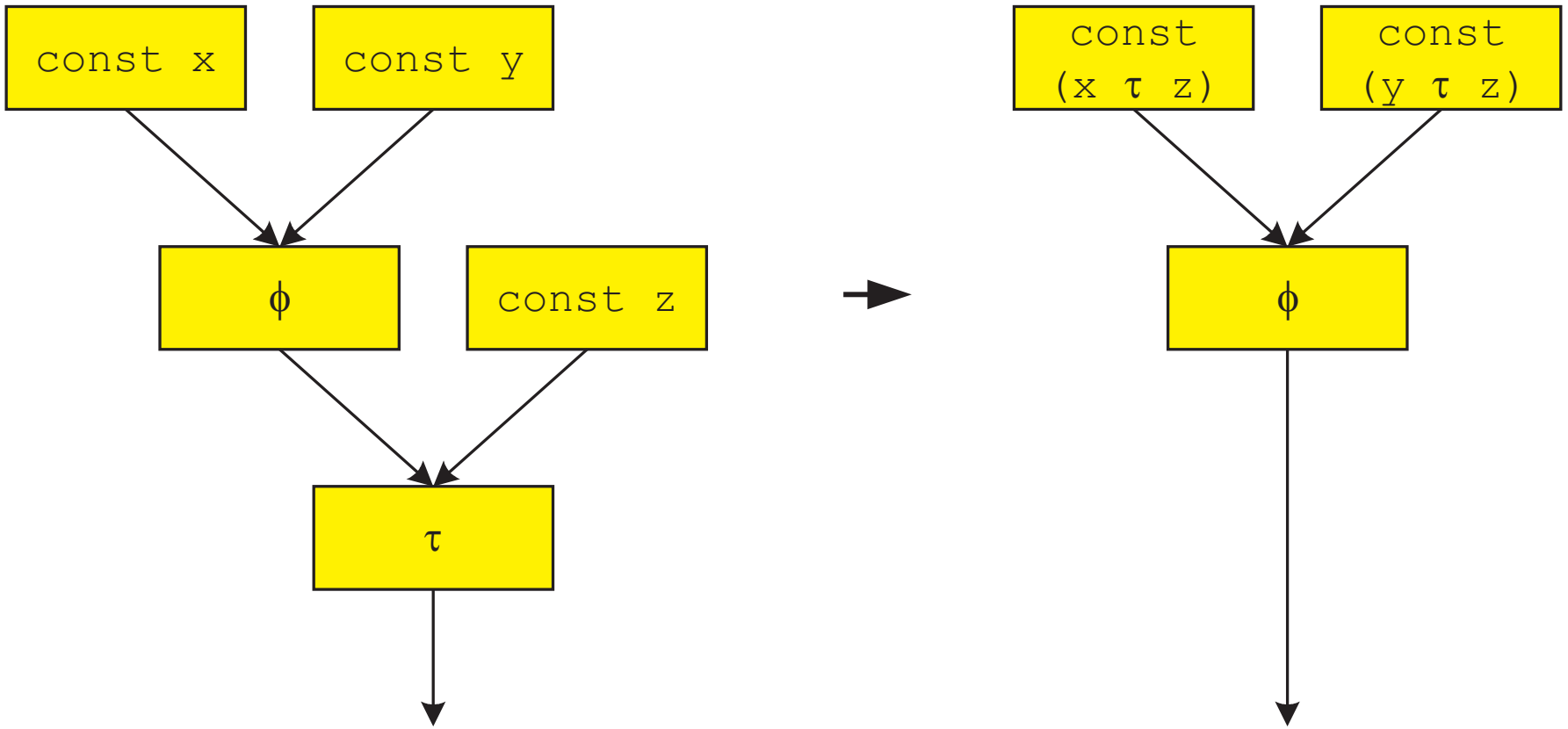
Optimierungen - Konstantenfaltung I

- Berechne (Teil-)Ausdrücke mit konstanten Operanden
- Beachte Arithmetik der Sprache
- Beachte Arithmetik der Zielmaschine
- Zu faltende Konstanten werden zu 80 bis 90% bei der Adreßrechnung erzeugt.



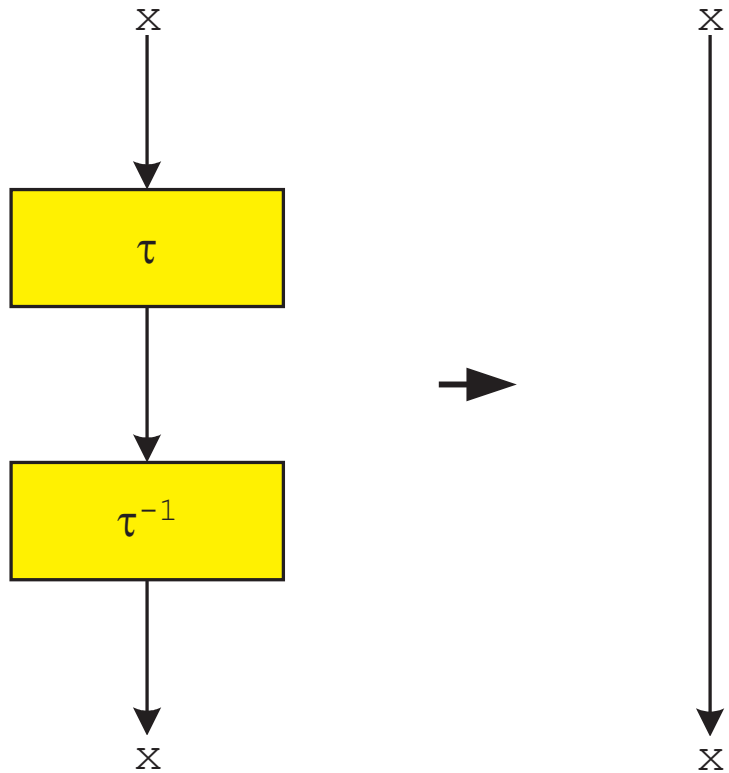
Optimierungen - Konstantenfaltung II

- auch über ϕ-Operationen hinweg möglich!



Optimierung - Inverse Operationen löschen

- oft nach Anwendung anderer Transformationen nötig.



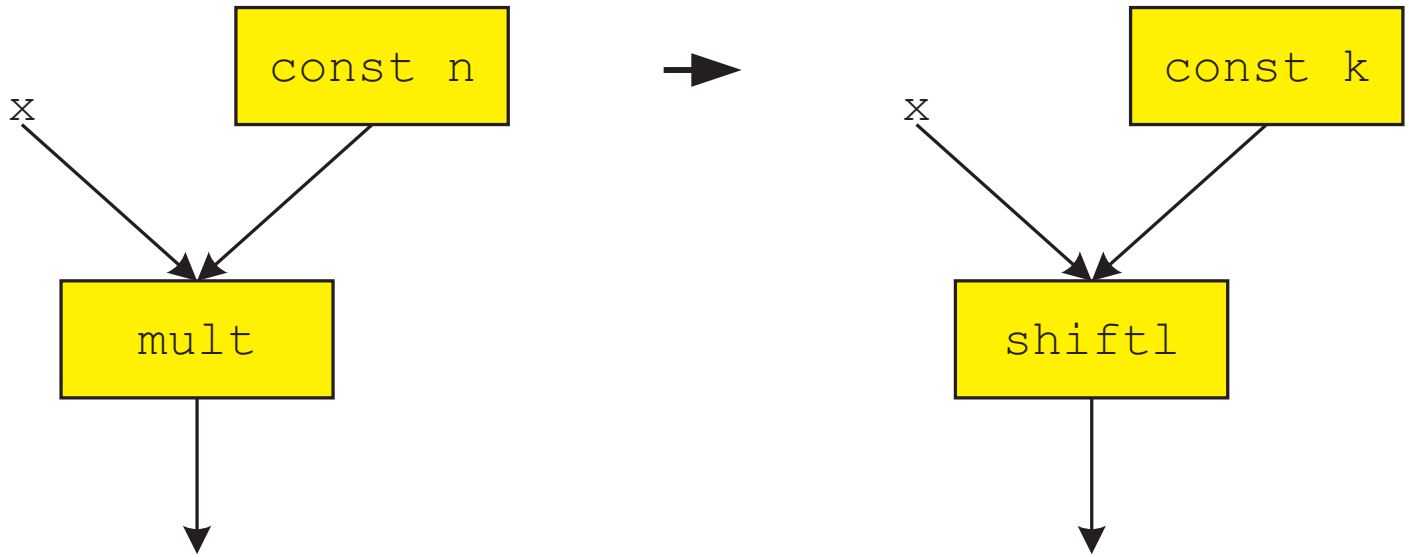
Operatorvereinfachung

Operatorvereinfachung (strength reduction)

- Idee: Ersetze teuere Operationen durch billigere, semantisch äquivalente Operationen
- Hauptanwendung: Vereinfache Multiplikationen mit Indexvariablen in Schleifen zu Additionen (Induktionsanalyse, lineare Adreßfortschaltung)
  - besonders wichtig auf Rechnern mit Multiplikationszeit >> Additionszeit
  - dort bei numerischen Programmen Beschleunigung bis Faktor 3 erzielbar.

Operatorvereinfachung Beispiel I

- elementares Beispiel: Berechnung von $x \times n, n = 2^k$



Operatorvereinfachung Beispiel II

```
for (i=0; i<n; i++)
  for (j=0; j<m; j++)
    ... a[i,j] ...;
```

Adreßberechnung:
⟩a[i,j]⟨=
⟩a[0,0]⟨+ i*m*d + j*d
Kosten pro Iteration:
3 Multiplikationen,
2 Additionen

Ideen eines Maschinensprachprogrammierers:

- initialisiere Adreßvariable adr vor den Schleifen mit ⟩a[0,0]⟨
- in der inneren Schleife:
  `adr = adr + d`
- dann gilt für eine $[0:(n-1),0:(m-1)$ Reihung in zeilenweiser Speicherung jeweils
  ⟨adr⟩ == ⟩a[i,j]⟨
- dabei benutzt:
  ⟩a[i+1,0]⟨==⟩a[i,m-1]⟨+d
- Kosten pro Iteration:
  1 Addition

Operatorvereinfachung Beispiel IIa

```
for (i=0; i<n; i++)
  for (j=0; j<m; j++)
    ... a[i,j] ...;
```

Adreßberechnung:
⟩a[i,j]⟨=
⟩a[0,0]⟨ + i*m*d + j*d
Kosten pro Iteration:
3 Multiplikationen,
2 Additionen

```
adr = ⟩a[0,0]⟨;

for (i=0; i<n; i++) {
  for (j=0; j<m; j++) {

...⟨adr⟩...;

    adr = adr + d;
  }
}
```

Kosten pro Iteration:
1 Addition

Induktionsanalyse (nicht im SSA-Kontext)

Operatorvereinfachung für Induktionsvariable
erfordert Induktionsanalyse

Definition: Induktionsvariable (IV):

- Variable i ist Induktionsvariable, wenn in der Schleife nur Zuweisungen der Form $i := i + c'$ vorkommen
- Variable i' ist Induktionsvariable, wenn i' nur Werte $i' := c * i + c''$ annimmt, wobei i eine Induktionsvariable ist.
- Dabei sind c, c', c'' während Ausführung der Schleife konstant

Transformation:

- Sei i_0 die Vorbesetzung von i vor Schleifenbeginn
  - Neue Variable i_a mit Initialisierung $i_a := c * i_0 + c''$ einführen
  - Zuweisung $i_a := i_a + c * c'$ am Ende der Schleife
  - Ersetze i durch $(i_a - c'')$ **div** c und i' durch i_a

Induktionsanalyse (nicht im SSA-Kontext)

- Einfaches Beispiel:

```
s := 0;
for (i:=0; i<100; i++)
    s := s + a[i];
```

- $> a[i] <$ wird in jedem Schleifendurchlauf berechnet als $> a[0] < + i \times d$
  mit Konstante d: Umfang des Typs von $< a >$

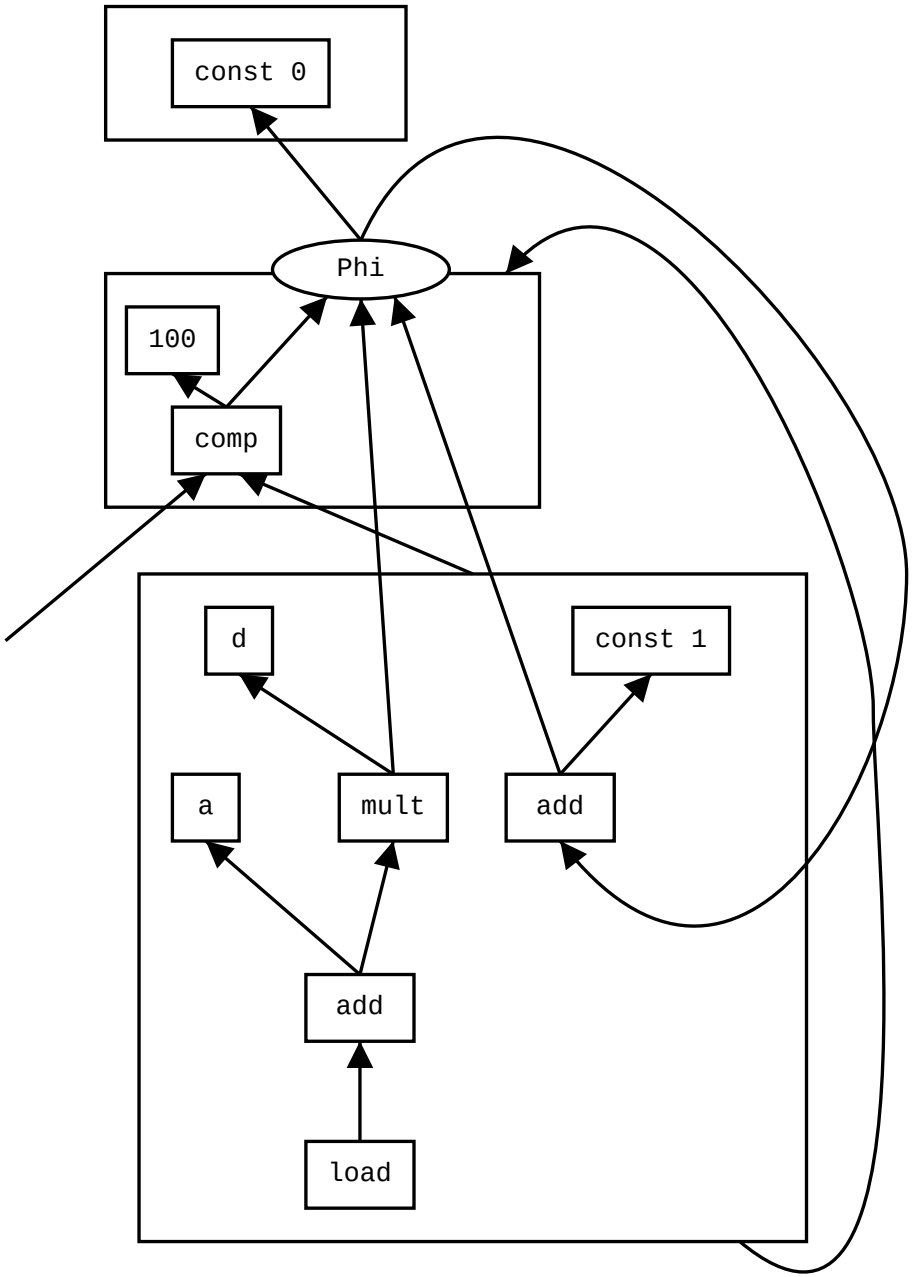
Bestimmung von Induktionsvariablen I (im SSA-Kontext)

- Gesucht: Werte (d.h., SSA-Ecken), die linear mit Anzahl der Schleifendurchläufe wachsen
- Optimistischer Ansatz:
  - Betrachte alle Werte als IV-Kandidaten
  - Bis zum Erreichen des Fixpunkts: Eliminiere Werte t aus der Menge der IV-Kandidaten, falls sie nicht eine der folgenden Formen haben:
    - $t := t' \pm c'$, wobei t' IV-Kandidat ist und c' außerhalb der Schleife berechnet wird (*direkte IV*)
    - $t := c \cdot t' \pm c''$, wobei t' IV-Kandidat ist und c, c'' außerhalb der Schleife berechnet werden (*indirekte IV*)
    - $t := \Phi(t_{i_1}, t_{i_2}, \ldots, t_{i_n})$, wobei alle Operanden des Φ-Operators entweder direkte IV-Kandidaten sind oder außerhalb der Schleife berechnet werden.
  - Beim Erreichen des Fixpunkts bleiben von IV-Kandidaten gerade die Induktionsvariablen übrig.
- In der Praxis auf ganze Zahlen beschränkt

Bestimmung von Induktionsvariablen II (im SSA-Kontext)

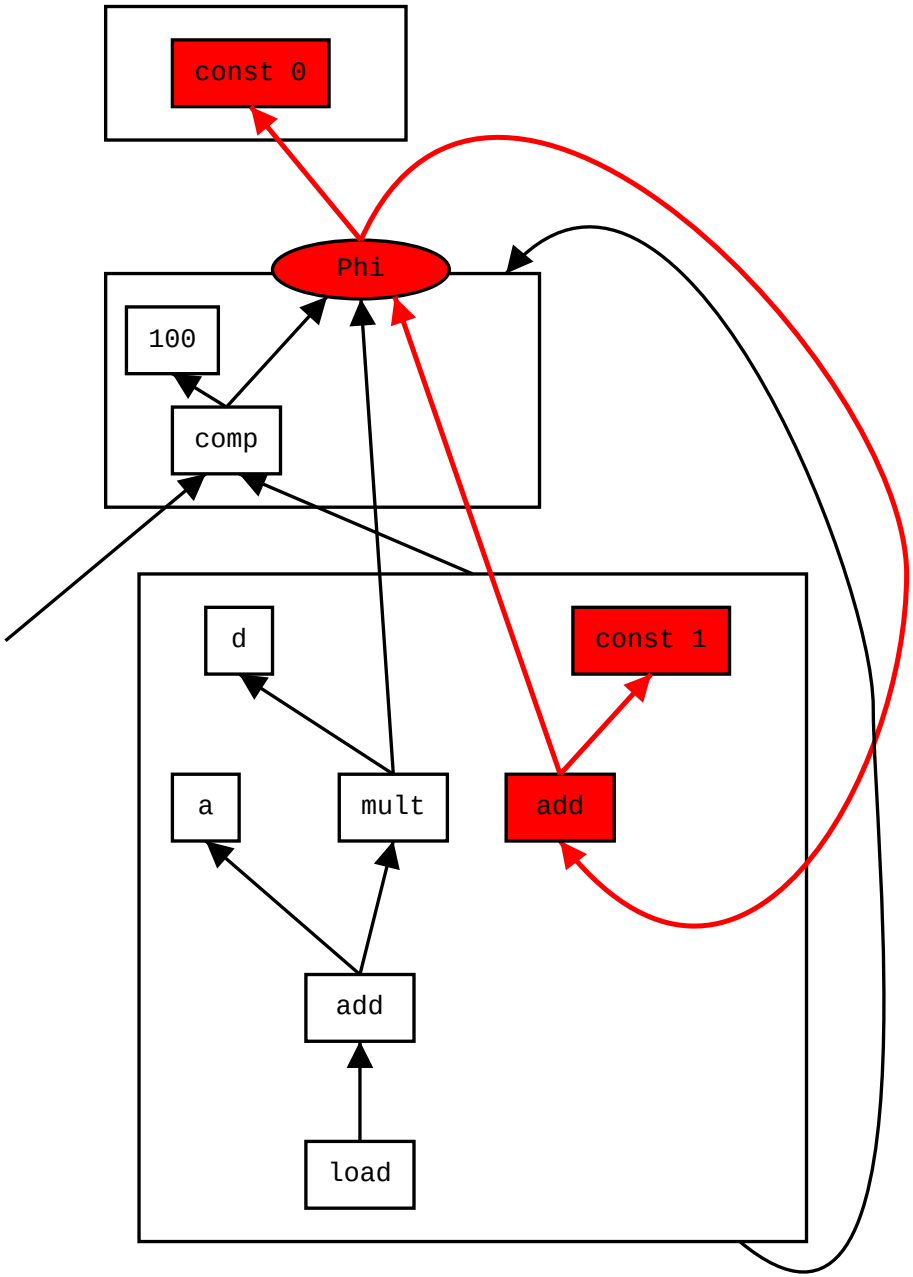
- Definition stark zusammenhängende Komponente (SZK) eines gerichteten Graphen $G = (N, E)$: $N' \subseteq N$ ist SZK von G, wenn für alle $n_i, n_j \in N'$ gilt: Es gibt einen Pfad von n_i nach n_j.
- Beobachtung:
  - Im SSA-Graph wird für Induktionsvariable ein SZK aufgebaut
  - Die SZK ist mit dem Restgraphen über eine Φ-Ecke verbunden
- Vorgehen:
  - Bestimme SZK's des SSA-Graphen (Tarjan-Algorithmus)
  - Überprüfe SZK's auf IV-Kandidaten
  - Für jede Benutzung eines IV-Kandidaten:
    - Vereinfache Berechnung des IV-Kandidaten (Φ-Zyklus)
    - Ändere Benutzung des IV-Kandidaten

Beispiel: Partieller SSA-Graph



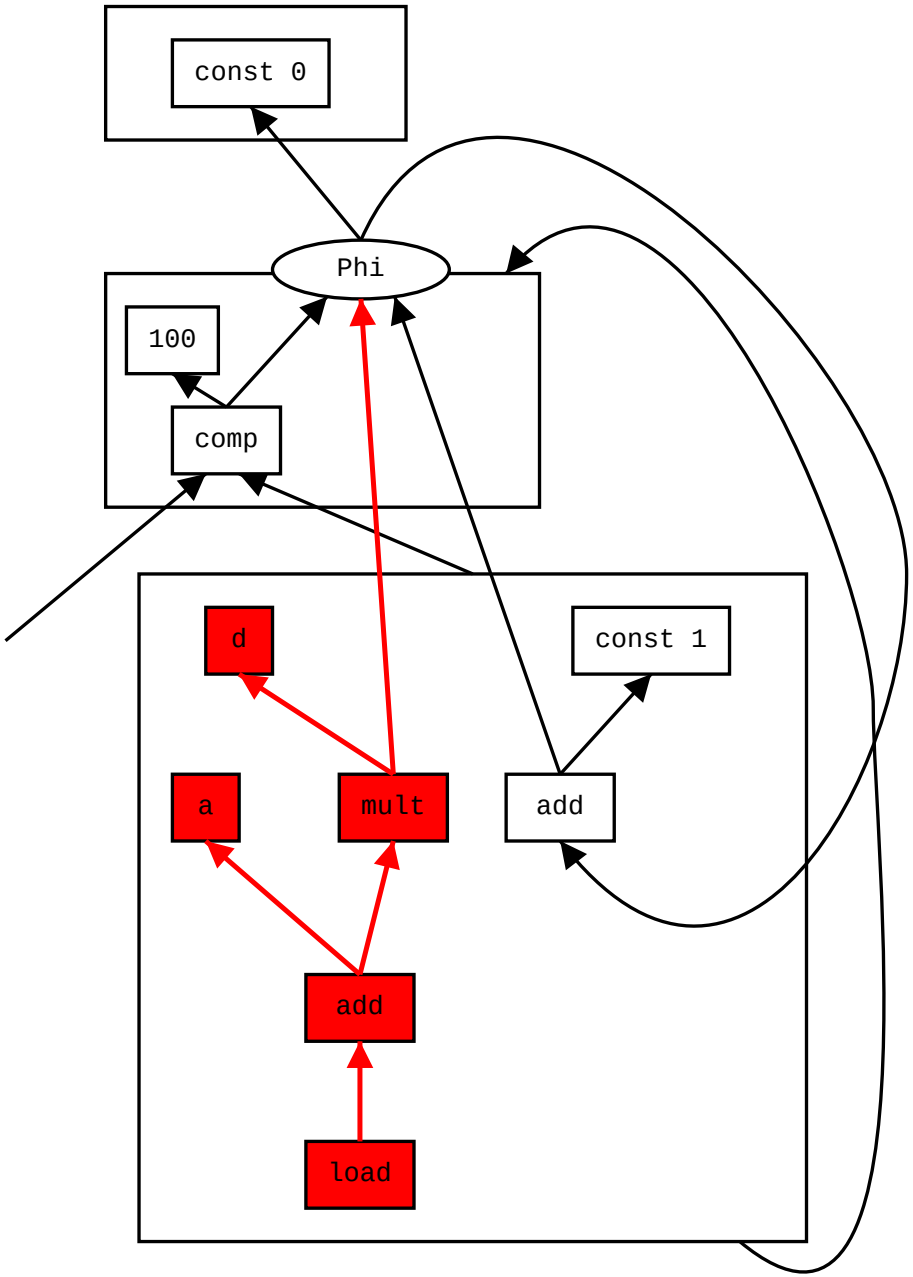
- Gezeigt:
  - Initialisierung und
  - Fortschaltung der Induktionsvariable
  - Berechnung der Lade-Adresse
- Nicht gezeigt:
  - Summierung der geladenen Werte

Beispiel: SZK im SSA-Graph



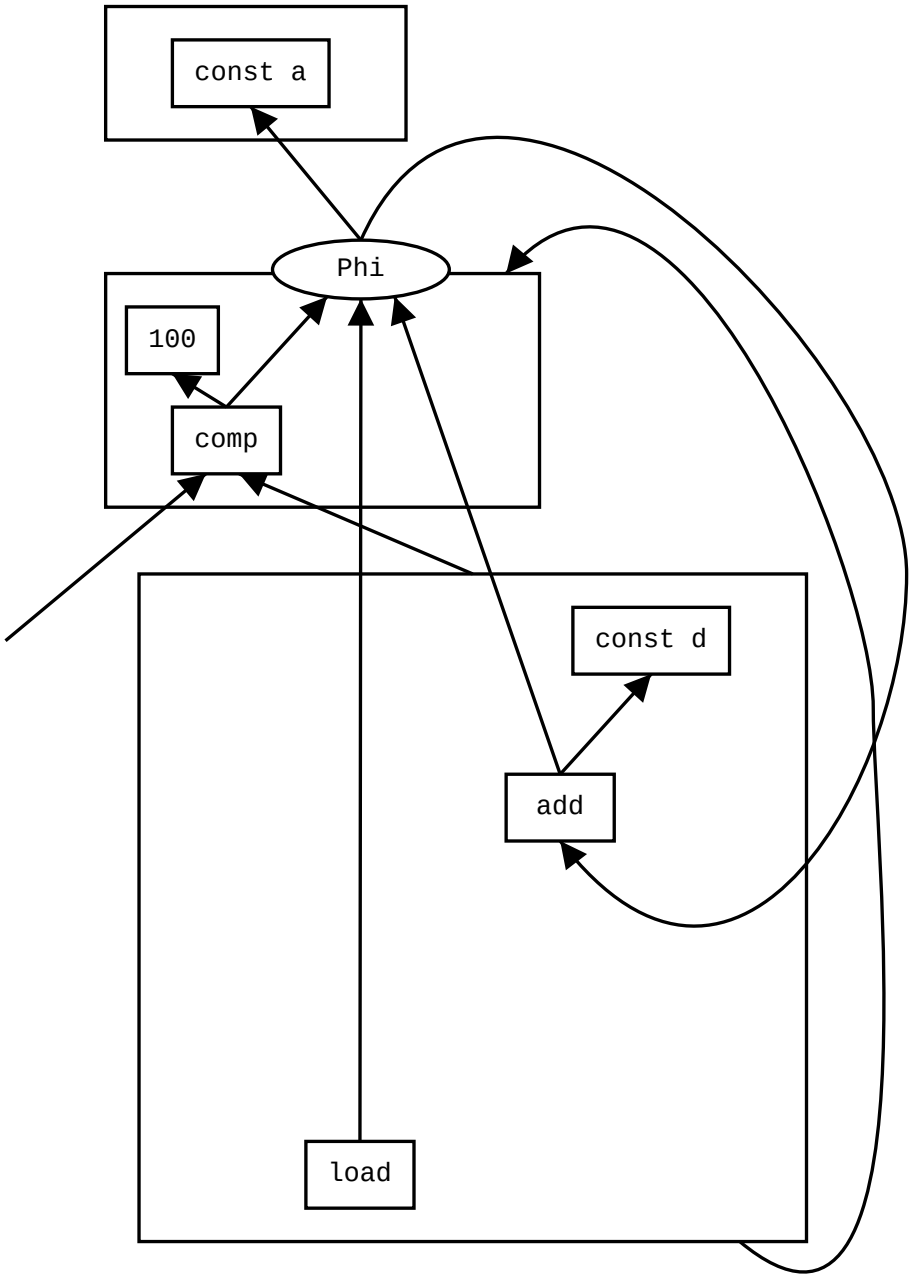
- Initialisierung und
- Fortschaltung der Induktionsvariable
- Φ-Ecke faßt zusammen:
  - Initialisierung
  - Konstante Erhöhung der Induktionsvariable
- bildet *stark zusammenhängende Komponente*
- Folge: Φ-Ecke ist *direkte* Induktionsvariable

Beispiel: Benutzung der Induktionsvariable



- Benutzung der Induktionsvariable (Φ-Ecken) für Adreß-Arithmetik
- Lineare Funktion $a_i := a_0 + d \cdot \Phi$
- Deshalb *indirekte* Induktionsvariable

Beispiel: Umgeformter SSA-Graph



- Bereits geändert:
  - Initialisierung und
  - Fortschaltung der Induktionsvariable
- Noch zu ändern:
  - Vergleich zum Schleifenabbruch

Inhalt

Eliminieren gemeinsamer Teilausdrücke

Ziel:

- Vermeide wiederholte Berechnung von Teilausdrücken

Anwendung:

- Berechnung von Speicheradressen bei Zugriffen auf Felder eines Verbundes (Studie: In PL/1-Programmen 60% aller Berechnungen sind Adreß-Arithmetik)
- Generierter, d. h. nicht von Hand geschriebener Code

Beispiel: Sei `offset_x` die relative Position von Feld `x` in Struktur `A`:

```
A a;
a.x = a.x + 1;

Code:
tmp1 = a + offset_x;
tmp2 = a + offset_x;
>tmp2< = <tmp1> + 1;
```

Eliminieren gemeinsamer Teilausdrücke

Definition Semantische Gleichheit: Wenn zwei Operationen das gleiche Ergebnis liefern, sind sie semantisch gleich. Beobachtung in SSA:

- Sind zwei Ausdrücke k, k' syntaktisch gleich, so sind sie auch semantisch gleich.
  - Syntaktische Gleichheit heißt: Gleiche Operanden, gleiche Operation
  - in Nicht-SSA-Darstellungen gilt dies nicht!

Folgerung:

- Sind k und k' syntaktisch gleich, dann kann k' wegfallen,
  - wenn k k' dominiert

Vorberechnung:

- Normalisierung, Kommutativgesetz anwenden
- Kann mit Eliminierungsalgorithmus verschränkt werden

Eliminieren gemeinsamer Teilausdrücke - Implementierung

- Aufgabe: Gegeben Ecken k mit Operation $\tau(op_1, op_2)$, finde alle Ecken j mit Operation $\tau(op_1, op_2)$.
- Durchführung:
  - Unterhalte Haschtabelle für SSA-Ecken;
  - Berechnung des Hasch-Schlüssels aus τ, op_1 und op_2
  - Für jede Ecke $k = \tau(op_1, op_2)$:

    Suche in Haschtabelle nach k
    Wenn Eintrag j vorhanden und j's Block dominiert k's Block:
     verwende j statt k

    sonst:
     verwende k und trage k in Tabelle ein

- Algorithmus kann in SSA-Aufbau integriert werden
  - SSA-Aufbau liefert alle Dominatoren von b vor dem Block b selbst
  - geringerer Speicherbedarf des SSA-Graphen

Inhalt

Eliminieren partieller Redundanzen

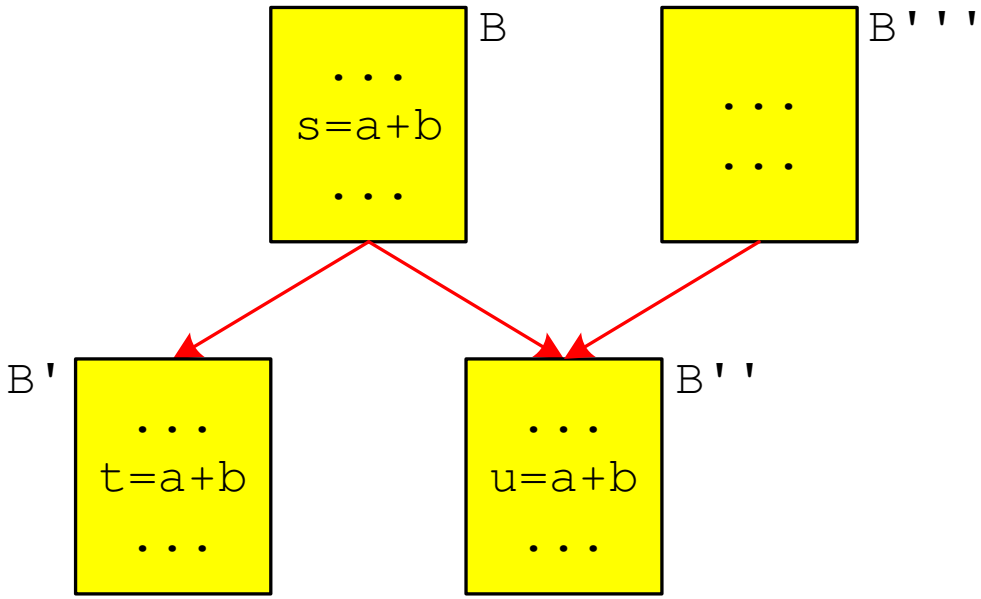
- Problem:
  Operation p (d.h. die Berechnung eines Wertes) wird auf manchen Pfaden mehrfach ausgeführt
- Idee:
  Reduziere die Anzahl der Berechnungen durch Umplazieren der entsprechenden Operationen p

Wiederholung: Sicher verfügbarer Ausdruck

- $a + b$ ist ein sicher/partiell verfügbarer Ausdruck an einer Stelle l, wenn er auf allen Pfaden/einem Pfad vom Funktionsbeginn berechnet wurde, und sich seine Operanden nicht geändert haben
- Bemerkung: Berechenbar mit Datenflußattribut *available expressions*.
  - Partiell verfügbare Ausdrücke ist eine möglich-Analyse
  - Sicher verfügbare Ausdrücke ist eine sicher-Analyse

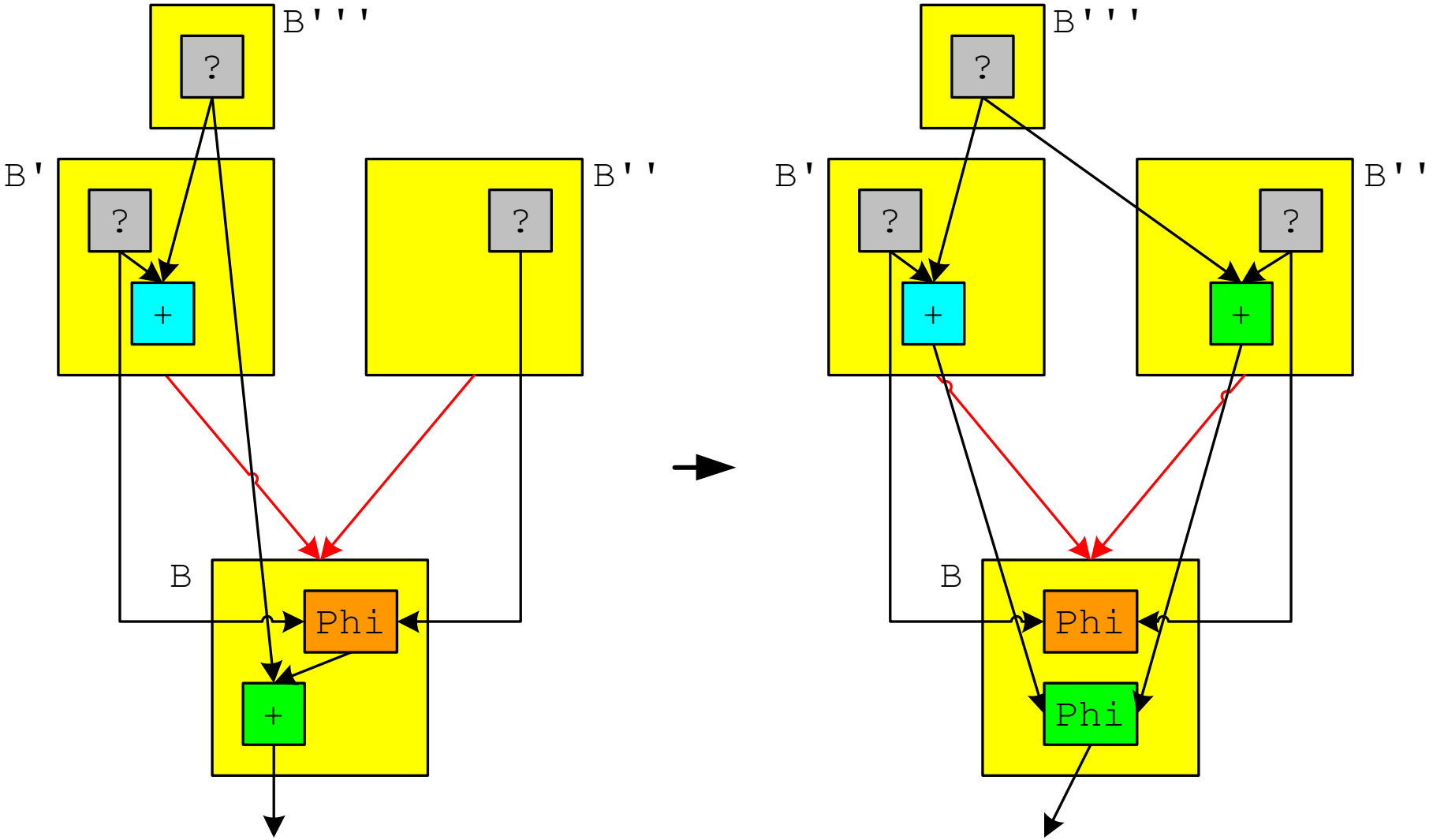
Partielle Redundanz (nicht im SSA-Kontext)

- Eine Berechnung $a + b$ ist sicher/partiell redundant, wenn $a + b$ unmittelbar vor dieser Berechnung sicher/partiell verfügbar ist.
- Beispiel:



Ausdruck $a + b$ ist in B' redundant und in B'' partiell redundant, da es in B''' nicht berechnet wird.

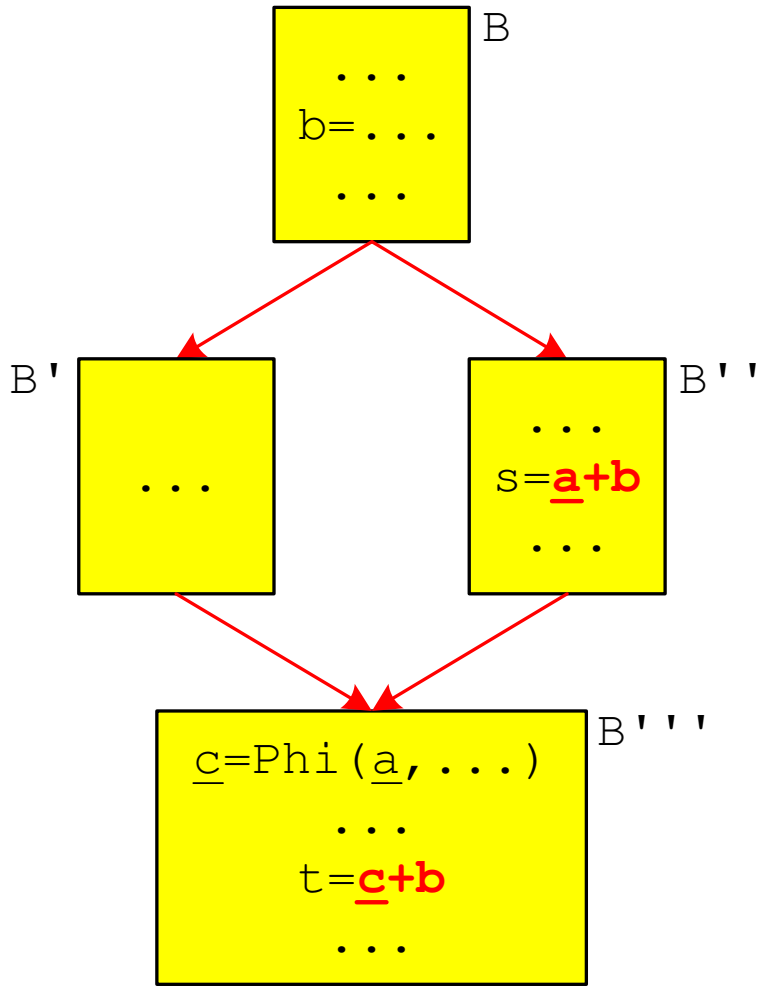
Eliminieren partieller Redundanzen – Beispiel in SSA-Form



Eliminieren partieller Redundanzen

Probleme bei SSA:

- In SSA haben Variable mehrere Versionen $a \rightarrow a_1, a_2, \ldots$.
- Φs vermischen den Wert einer Variable mit dem Ausführungspfad.
- Dadurch können gleiche Berechnungen auch unterschiedliche Versionen haben, z.B. sind $a + b$ sowie $c + b$ die gleichen Berechnungen, wenn der Ausführungspfad B, B'', B''' ist.



PRE: Entwicklung im Überblick

- Standard PRE-Algorithmen verwenden bis zu sechs Datenflußgleichungen parallel und arbeiten nicht auf SSA.
- Standard-Verfahren sind:
  - Morel, Renvoise, *Global optimization by suppression of partial redundancies*, 1979
  - Drechsler, Stadel, *A Solution to a Problem with Morel's and Renvoise's Global Optimization*, 1988
  - Knoop, Steffen, Rüthing, *Lazy Code Motion*, 1993
  - Cooper, Briggs, *Effective PRE*, 1994
- Datenflußgleichungen auf SSA bedeutungslos
- SSA wenig geeignet, da Φs gleiche Berechnungen verstecken
- PRE (auf SSA) ist aktuelles Forschungsthema
  - Chow, Kennedy, et al. *Partial Redundancy Elimination in SSA Form*, 1997
  - Bodík, Gupta, Soffa, *Complete Removal of Redundant Expressions*, 1998
  - VanDrunen, *Partial Redundancy Elimination for Global Value Numbering*, 2002

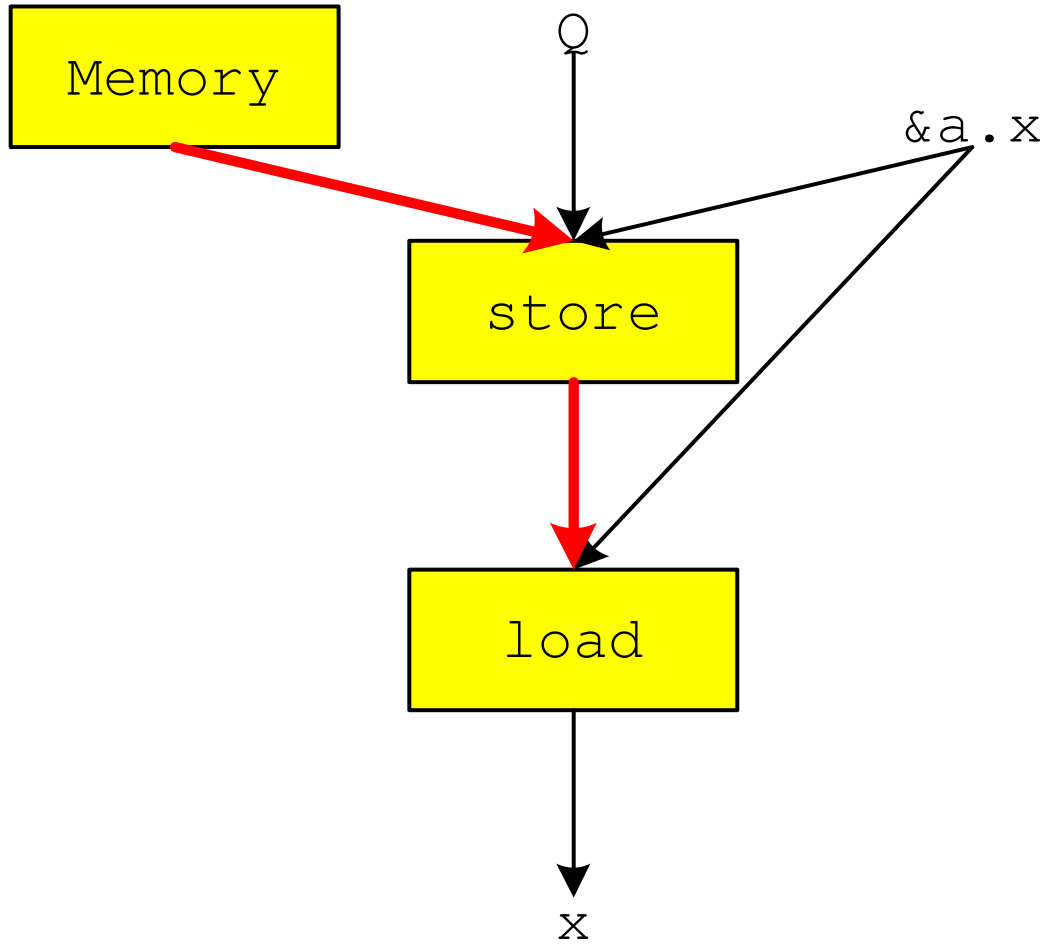
Inhalt

Einfache Speicheroptimierungen – Alias Problematik

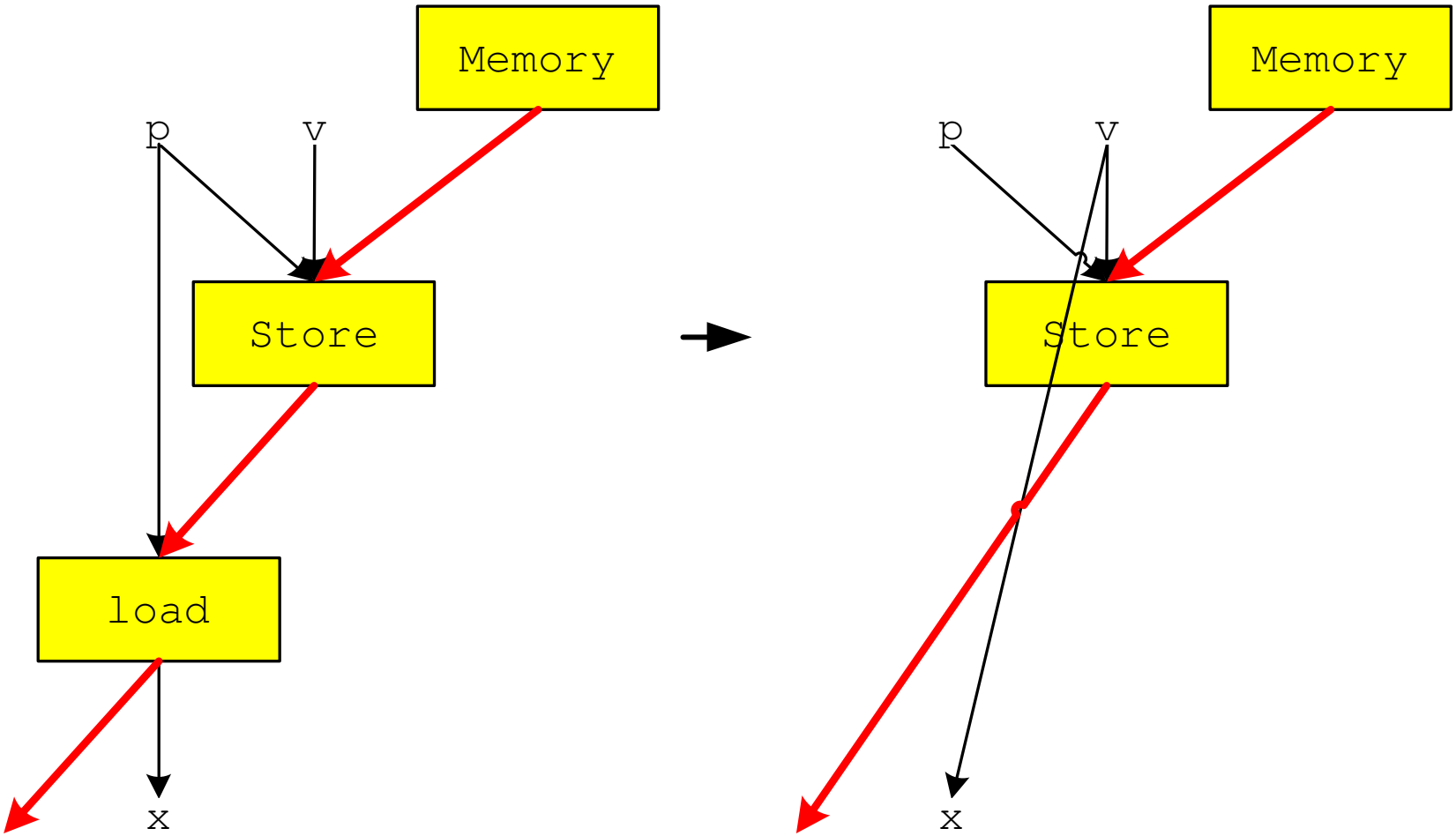
- Problem: Namen bezeichnen Zugriffspfade, keine (absoluten) Adressen
- Definition „Alias“:
  - unterschiedliche (Quelltext-)Namen, aber
  - gleiche (physikalische) Ressource (Adresse)
- Ursache:
  - Reihungen (`a[i]`, `a[j]`)
  - Referenzparameter
  - Zeiger, Referenzen, *address-of*-Operator `&`
  - *Call-by-Name*
- Folge:
  - Zugriffe auf Reihungen/Verbunde sind geordnet; Ordnung *nicht* mit bisherigem SSA-Aufbau darstellbar
  - Pessimistischer Ansatz: Totale Ordnung bei Speicherzugriffen

Beispiel: SSA-Aufbau für Speicherzugriffe

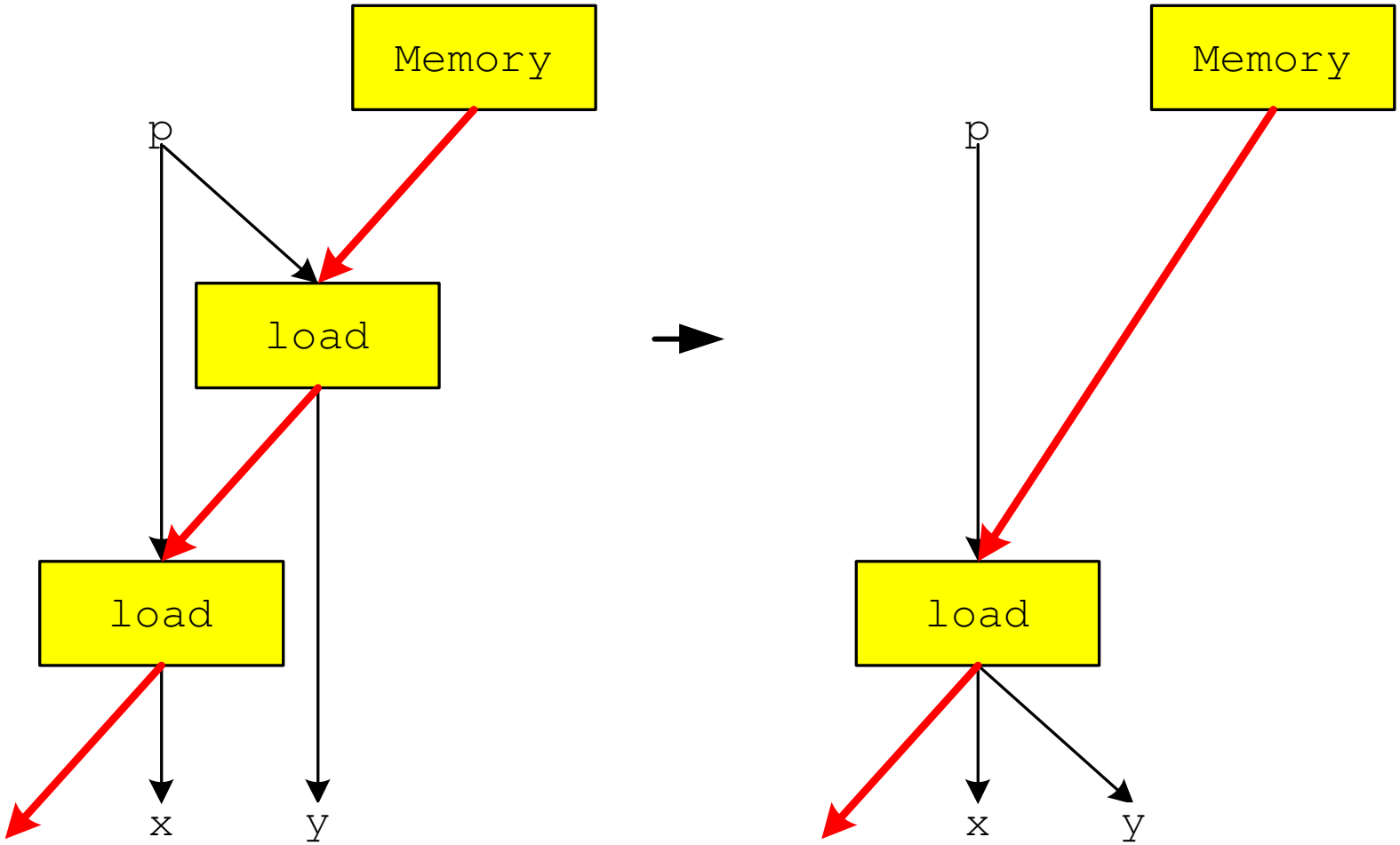
```
foo (A a, A b)
{
  int x;
  a.x = Q;

  x = a.x;
}
```



Optimierungen - Lade-Speichere entfernen



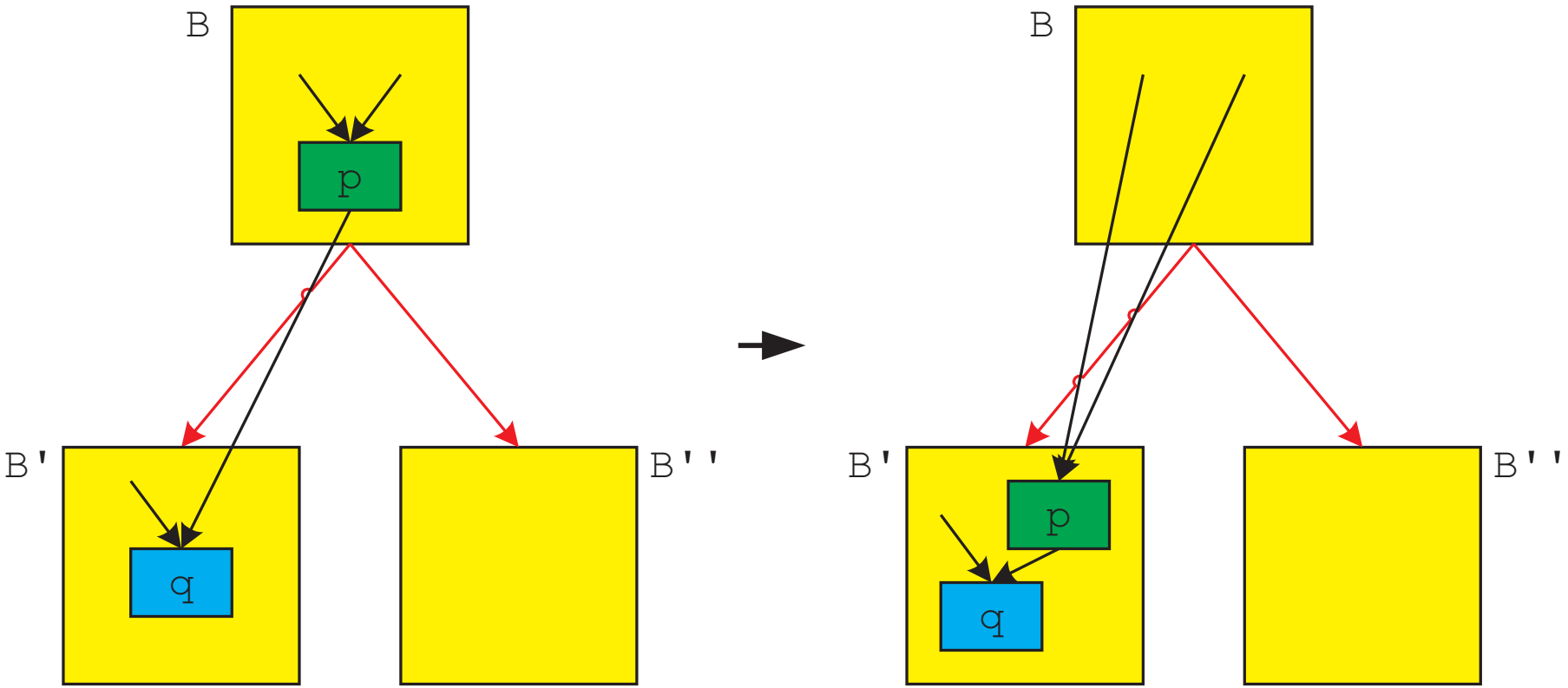
Optimierungen - Lade-Lade entfernen



Eliminieren unnötiger Berechnungen

- Problem:
  - Operation p in Block B
  - Block B dominiert Blöcke B' und B''
    keine Dominanz-Relation zwischen B' und B''
  - Wert von p wird nur in B' benutzt
  - Folge: Wert von p wird auch berechnet, wenn später B'' ausgeführt wird
- Transformation:
  - Verschiebe p, so daß p's Block keine Blöcke B''' dominiert, in denen p nicht benutzt wird
- Hinweis:
  - Eliminieren partieller Redundanzen entfernt Berechnungen die auf manchen Pfaden doppelt oder mehrfach stattfinden. Diese Optimierung hingegen entfernt Berechnungen, die auf manchen Pfaden gar nicht benötigt werden.
  - Es wird nicht beachtet ob Code in/aus Schleifen verschoben wird. Dies bietet weiteres Potential, kompliziert aber die Optimierung.

Eliminieren unnötiger Berechnungen



Eliminieren unnötiger Berechnungen

- Problem: Es gibt einen Ablaufpfad, auf dem p berechnet, aber nicht benutzt wird.
- Wiederholung Nachdominanz: Block B' wird von Block B nachdominiert genau dann, wenn jeder Ablaufpfad, auf dem B' liegt, auch B enthält.
- Beobachtung:
  - Operation p in Block B', Operation in Block B benutzt Ergebnis q von p.
  - Wird Block B' von B nachdominiert, dann wird p immer verwendet!
- Algorithmus zur Erkennung unnötiger Operationen:
  - Markiere alle SSA-Ecken als unnötig.
  - Für jede SSA-Ecke p in Block B, betrachte deren Operanden $q_1, \ldots, q_n$ aus Blöcken $B_1, \ldots, B_n$:
    Wenn für ein $i \in [1 \ldots n]$ Block B_i von Block B nachdominiert wird, markiere q_i als nötig (d. h. *nicht* unnötig).

Eliminieren unnötiger Berechnungen

- Transformation unnötiger Berechnungen:
  - Für jede Benutzung q_i einer unnötigen Operation p, erstelle Kopie p_i von p, die nur von q_i verwendet wird
  - Verschiebe p_i entlang Dominatorbaum „zu q_i hin“
  - Danach gemeinsame Teilausdrücke eliminieren
- Problem: Operationen könnten in Schleifen *hineinverschoben* werden
- Ausweg: Stoppe vor Block C, wenn C Schleifenkopf ist (Verwende stark zusammenhängende Komponenten aus Induktionsanalyse)
- Ergebnis:
  - Operationen sind entweder nötig (also nicht unnötig), oder
  - Operationen sind unnötig, aber stehen nicht in Schleifen, innerhalb derer sie nicht neu berechnet werden
  - Vorsicht: Anwachsen des Codes möglich!

Offener Einbau von Prozeduren - Idee

- In Prozedur p, ersetze Prozeduraufruf an q durch den Rumpf von q unter Ersetzung der formalen Parameter durch die Argumente des Aufrufs.
- Verwaltungsaufwand für Prozeduraufruf:
  - Ablage der Argumente durch den Aufrufer
  - Sichern des Kontextes des Aufrufers
  - Erstellen der Schachtel der aufgerufenen Prozedur
  - Abbau der Schachtel der aufgerufenen Prozedur
  - Wiederherstellen des Kontextes des Aufrufers
  - Rückgabe des Ergebnisses an den Aufrufer
- Wird q an allen Aufrufstellen offen eingebaut, dann ist q redundant
- Optimierung des offen eingebauten Codes im Kontext der Aufrufer

Offener Einbau von Prozeduren - Vorgehen

- Gieriges Einfügen nicht sinnvoll
  - Auch ohne Rekursion exponentielles Anwachsen des Codes
  - Mit Rekusion keine Terminierung
- Profitabel für
  - Prozeduren mit einem oder nur wenigen Aufrufern
  - Prozeduren, die oft mit konstanten Argumenten aufgerufen werden
  - Blattprozeduren
  - Aufrufe in Schleifen
  - (Rekursive Prozeduren mit geringer Rekursionstiefe)
- Entscheidung:
  - Definiere Kostenmaß I auf Prozeduraufrufen (Heuristik)
  - Definiere Maximalwert I_{max}
  - Wenn $I(p, q) < I_{max}$ ist, dann wird q offen in p eingebaut
- Wenn offener Einbau nicht möglich, dann wenigstens *Tail-call*- oder eingeschränkt *Tail-recursion*-Optimierung (in SSA schwierig)

Zusammenfassung

- Viele Optimierungen, jedoch auch mit einer kleinen Auswahl kann viel erreicht werden (SSA-Auf/-Abbbau, Operatorvereinfachung, CSE, PRE).
- Durch SSA-Form werden viel Analysen (Datenfluß) überflüssig bzw. trivial. Optimierungen lassen sich leichter ausführen.
- Bei der Operatorvereinfachung werden teuere (z.B. `mult`) duch billigere (z.B. `add`) Operationen ersetzt.
- Beim gemeinsame Teilausdrücke Eliminieren (CSE) werden die Werte von Berechnungen wiederverwendet.
- Das Eliminieren partieller Redundanzen (PRE) verschiebt Berechungen in Ablaufvorgänger. Dabei sollen die Berechnungen nur genau einmal erfolgen, und ihr Ergebnis auf allen Pfaden gebraucht werden.
- Hauptvorteil der SSA-Form: Nicht nur Ausgangspunkt, sondern auch einheitliches Ziel der Optimierungen

SSA im Backend

Inhalt

Maschinenabhängige Übersetzung

Kernaufgaben

- Befehlsauswahl
- Befehlsanordnung
- Registerzuteilung

Problem: Diese Aufgaben beeinflussen sich gegenseitig.

- Phasenkopplungsproblem

Zusätzlich:

- Spezial-Optimierungen für die jeweilige Zielarchitektur
- Implementierung der Konventionen der Laufzeitumgebung
- Implementierung komplexer Hochsprachen-Elemente (z.B. Vererbung)

Übersetzung in SSA-Form

Erhalte SSA im Backend

- Einheitliche Programmrepräsentation im Übersetzer
- Optimierungen aus der Optimierungsphase wiederverwendbar

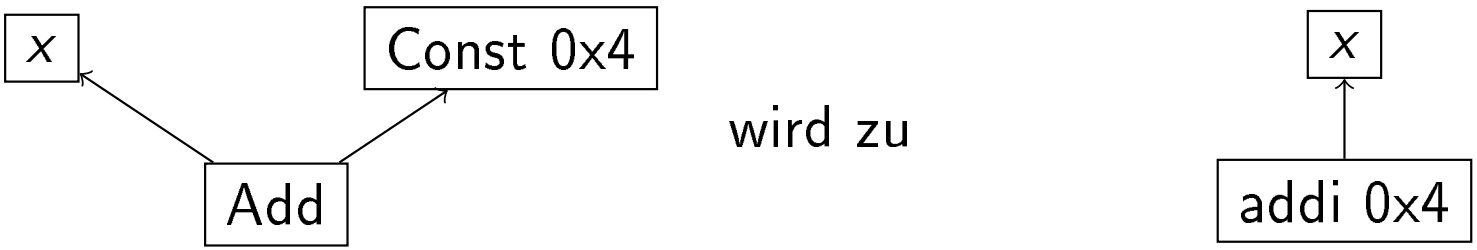
Ablauf der Übersetzung

- Befehlsauswahl
- Anordnung
- Registerzuteilung
- Nach-Anordnung im Rahmen der Registerzuteilung
- SSA-Abbau

Inhalt

Befehlsauswahl – Aufgabe

- Graphtransformation des Zwischensprachengraphs in einen Codegraph
- Muster aus Zwischensprachenecken werden in eine oder mehrere Codeecken überführt



Methoden

Anforderungen

- SSA-Eigenschaft muss erhalten bleiben
- Φ-Funktionen werden nicht angetastet
- Befehle haben *ein* Ergebnis (ggf. Projektionen aus Tupeln nötig)

Problem

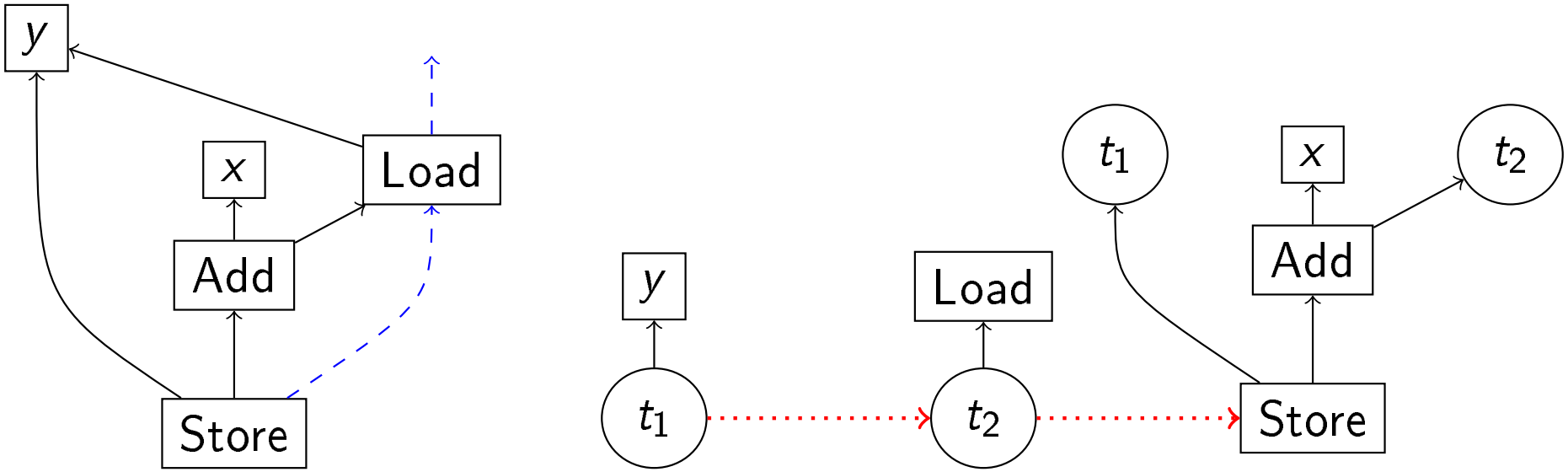
- Zwischendarstellung keine Wälder:
  - In Grundblöcken: DAGs auf Grund von Optimierungen
  - Grundblockübergreifend: DAGs auf Grund von Φ-Funktionen
- Termersetzung nur nach Aufbrechen der DAGs in Bäume anwendbar

Methoden

- Makrosubstitution
- Termersetzungsverfahren
  - CGGG von Boesler, IPD
  - BURS Code-Generator des Java Hotspot Compilers (SUN)
- Einziges Graphersetzungsverfahren:
  PBQP-Verfahren von Eckstein, König und Scholz, TU Wien

Aufbrechen des DAGs in Bäume

Load/Op/Store-Befehle bei ia32



- t_1, t_2 sind neue Hilfsvariablen die genau einmal als Wurzel und beliebig oft als Blätter auftreten können
- Die gepunktete rote Kante wird nicht von TES verwendet sondern gibt die Ausführungsreihenfolge vor
- Die gestrichelte blaue Speicherkante wird nicht mehr benötigt, da die Befehlsanordnung nun z.T. explizit ist
- Das TES kann optimalen Maschinenbefehl nicht finden

Inhalt

Befehlsanordnung

- Ziel: Bessere Nutzung der Parallelarbeit auf Befehlsebene durch geschickte Anordnung der Reihenfolge von Befehlen
- Spezialitäten:
  - Füllen des Verzögerungsschritts nach bedingten Befehlen
  - Vermeiden des Wartens auf den Speicher
  - Kollisionsvermeidung in den Rechenwerken bei mehrstufigen Befehlen
  - Software-Fließband für Schleifen
- Methodik:
  - gierige Algorithmen innerhalb eines Grundblocks
  - Optimierung für Schleifen und Teile des Dominatorbaums
- bei vielen neueren Prozessoren (moderne RISCs, Pentium Pro, Pentium II, III, ..., nicht: IA-64) durch Hardware (out-of-order-execution), aber trotzdem ist Vorarbeit durch den Übersetzer vorteilhaft.

Beispiel Verzögerungsschritt

- Fakt: Auf klassischen (heute veralteten) RISC-Prozessoren dauern bedingte Sprünge 2 Takte. Der 2. Takt kann mit Befehl gefüllt werden, der immer ausgeführt wird, unabhängig von Sprungbedingung. Sprungbedingung darf nicht von diesem Befehl abhängen.
- Beispiel MIPS:

		Takt
`addiu $t1 $t1 1`	$\$t1 := \$t1 + 1$	1
`addiu $t2 $t2 1`	$\$t2 := \$t2 + 1$	2
`beq $t2 $t3 marke`	wenn $\$t2 = \$t3$ dann gehe zu marke	3,4
`nop`	Verzögerungsschritt	4

Bemerkung: Bei heutigen superskalaren Prozessoren mit Sprungvorhersage und Caches ist die Lage nicht mehr so einfach.
Ausnahmen: IA-64 (benötigt explizite Angaben was parallel / spekulativ auszuführen ist).

Beispiel Verzögerungsschritt II

- Original:

		Takt
`addiu $t1 $t1 1`	$\$t1 := \$t1 + 1$	1
`addiu $t2 $t2 1`	$\$t2 := \$t2 + 1$	2
`beq $t2 $t3 marke`	if $\$t2 = \$t3$ then goto marke	3,4
`nop`	Verzögerungsschritt	4

- unzulässig:

		Takt
`addiu $t1 $t1 1`	$\$t1 := \$t1 + 1$	1
`beq $t2 $t3 marke`	if $\$t2 = \$t3$ then goto marke	2,3
`addiu $t2 $t2 1`	$\$t2 := \$t2 + 1$	3

- zulässig:

		Takt
`addiu $t2 $t2 1`	$\$t2 := \$t2 + 1$	1
`beq $t2 $t3 marke`	if $\$t2 = \$t3$ then goto marke	2,3
`addiu $t1 $t1 1`	$\$t1 := \$t1 + 1$	3

Verfahren zur Anordnung

Methode (*list scheduling*): Konstruiere für jeden Grundblock Abhängigkeitsgraph (Info bereits in graphbasierter SSA-Form vorhanden!):

- b hängt von b' ab, wenn
  - In der Quelle b' vor b kommt (nur bei Java o.ä.)
  - mindestens einer der Befehle ein Register oder Speicherzelle beschreibt
  - der andere dasselbe Register (Speicherzelle) liest oder schreibt
- bewerte alle Befehle b mit (# Zyklen für längsten Pfad im Graph beginnend mit b, # abhängiger Befehle)
- $b < b'$ (höchste Priorität), wenn
  - # Zyklen(b) > # Zyklen(b') oder
  - # Zyklen(b) = # Zyklen(b') und # abh. Bef.(b) > # abh. Bef.(b')

Anordnung

- wähle nächsten Befehl b wenn:
  - alle Befehle, von denen b abhängt, bereits angeordnet und mit Beginn des laufenden Takts fertig
  - b hat höchste Priorität
  - ist bedingter Sprung und es bleiben höchstens Befehle, die in den Verzögerungsschritt passen
- gibt es keinen solchen Befehl, so wähle `nop`
- ordne den ausgewählten Befehl als nächsten an und streiche ihn samt Kanten aus dem Abhängigkeitsgraph

Beispiel Anordnung

Befehle:

Befehl	Bedeutung	Dauer (Takte)
`i := j` $+_i$ `k`	integer add	1
`x := y` $+_f$ `z`	float add	2
`i := j * k`	int/float mult	3
`i := j(k)`	lade $<< j > + k >$	2
`j(k) := i`	speichere $>< j > + <$	1
`if i` ρ `j then goto marke`	bedingter Sprung	4

Beispiel II

Programm
(inneres Produkt von a und b):

```
    res := 0;
    i := 1;
wd: h1 := 4*i;
    h2 := h1(a);
    h3 := h1(b);
    h4 := h2*h3;
    res := res+h4;
    i := i+1;
    if i/=n then goto wd
```

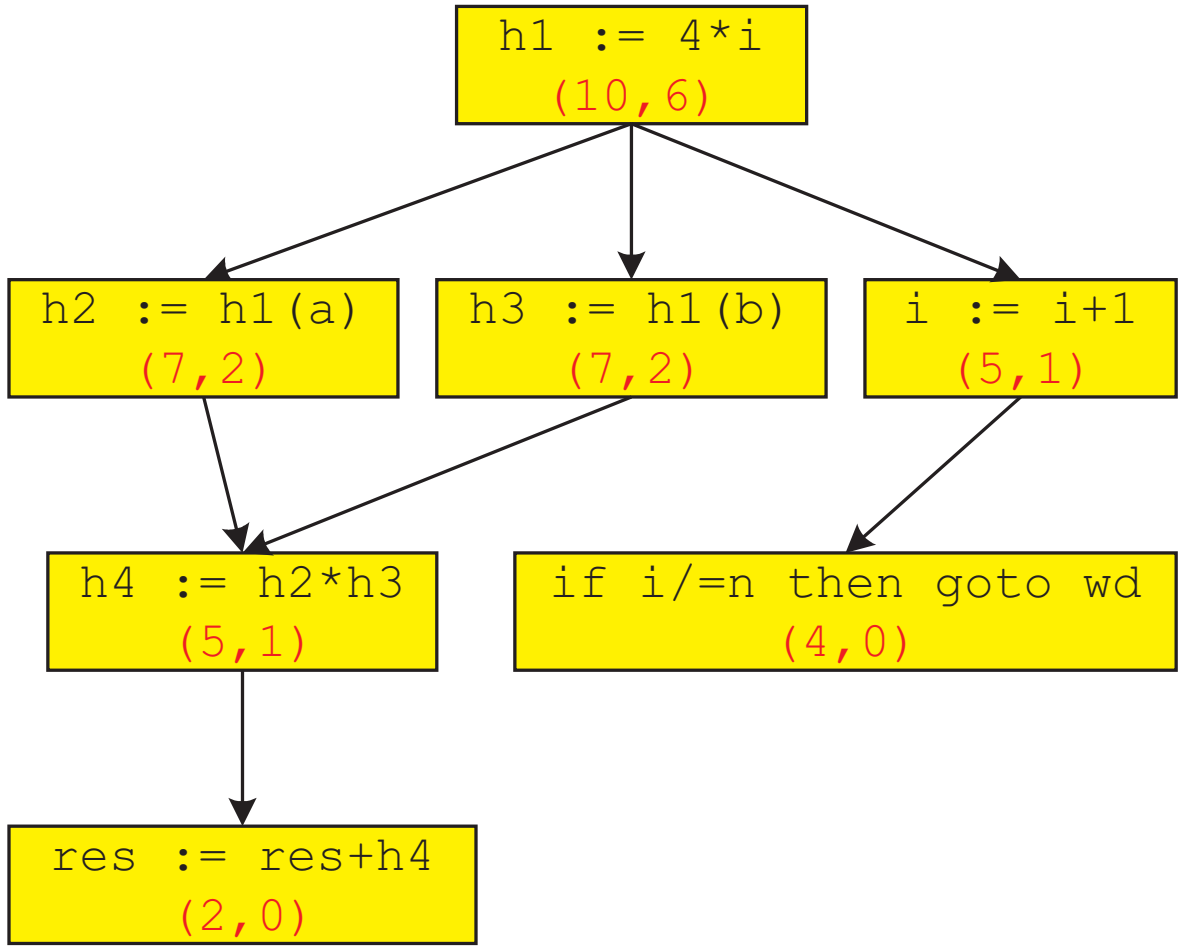
Ausführung Initiale Ordnung:

```
    res := 0;                    Takt: 0
    i := 1;                            1
wd: h1 := 4*i;                         2
    nop;                               3
    nop;                               4
    h2 := h1(a);                       5
    h3 := h1(b);                       6
    nop;                               7
    h4 := h2*h3;                       8
    nop;                               9
    nop;                              10
    res := res+h4;                    11
    i := i+1;                         12
    if i?n then goto wd               13
    nop;                              14
    nop;                              15
    nop;                              16
```

Beispiel III

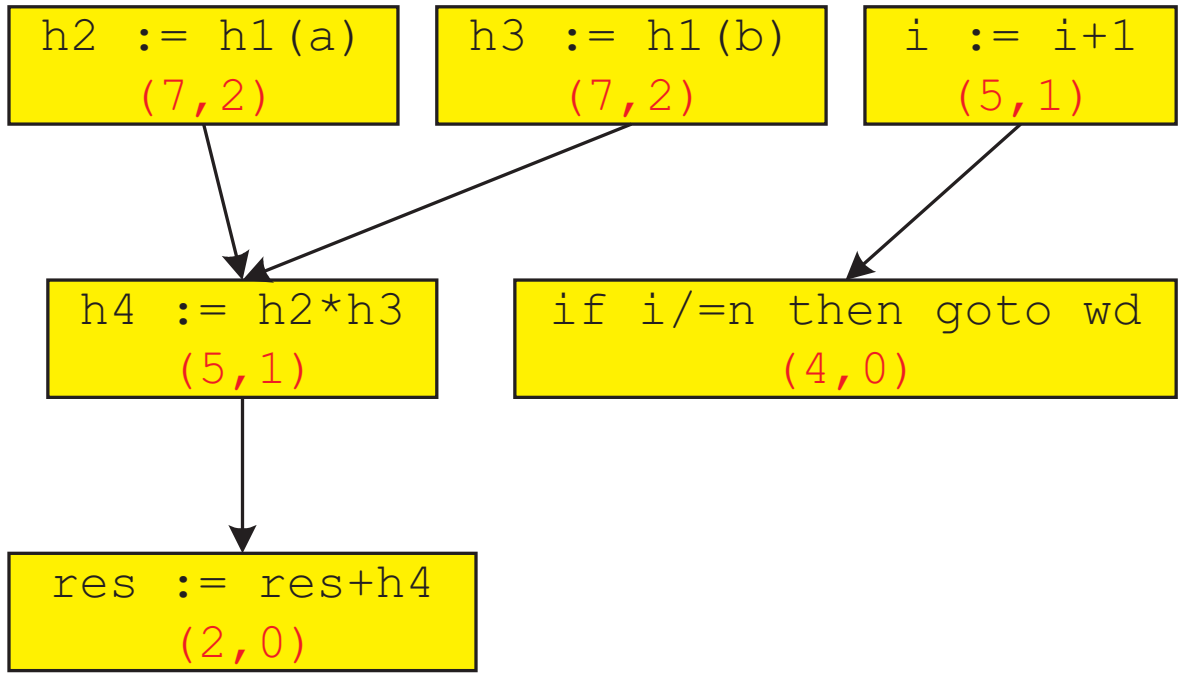
Angeordneter Code und Abhängigkeitsgraph der Schleife:

```
    res := 0;
    i := 1;
wd:
```



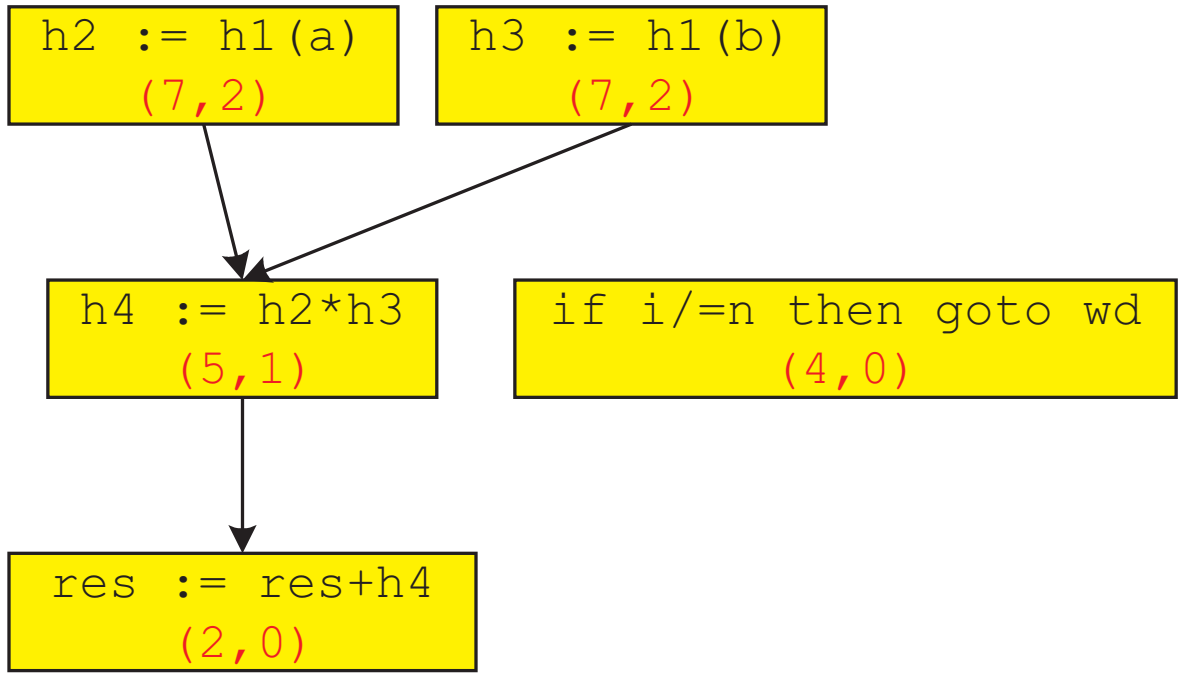
Beispiel IV

```
     res := 0;
     i := 1;
wd:  h1 := 4*i;
```



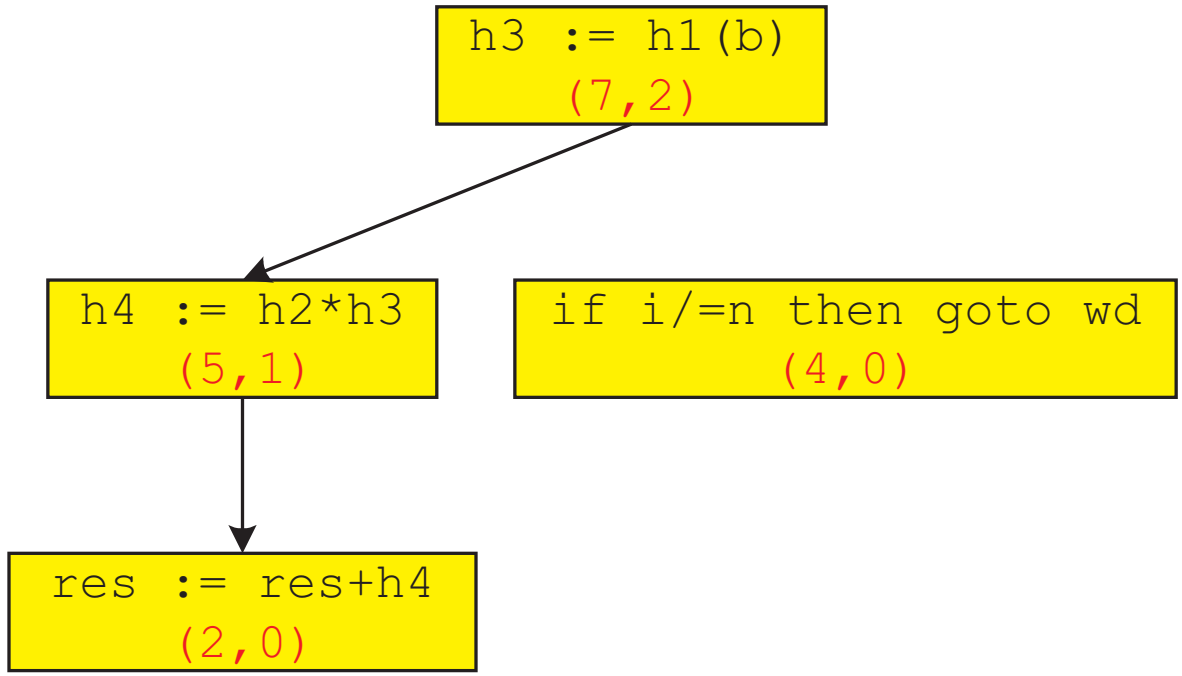
Beispiel V

```
     res := 0;
     i := 1;
wd:  h1 := 4*i;
     i := i+1;
```



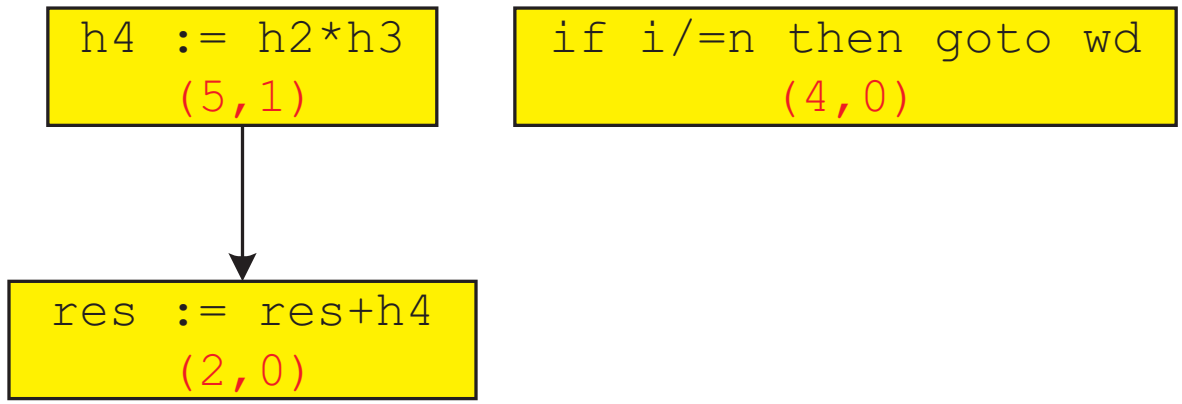
Beispiel VI

```
     res := 0;
     i := 1;
wd:  h1 := 4*i;
     i := i+1;
     h2 := h1(a);
```



Beispiel VII

```
    res := 0;
    i := 1;
wd: h1 := 4*i;
    i := i+1;
    h2 := h1(a);
    h3 := h1(b);
    nop;
```



Beispiel VIII

```
    res := 0;
    i := 1;
wd: h1 := 4*i;
    i := i+1;
    h2 := h1(a);
    h3 := h1(b);
    nop;
    h4 := h2*h3;
```

```
if i/=n then goto wd
      (4,0)
```

```
res := res+h4
   (2,0)
```

Beispiel IX

```
    res := 0;
    i := 1;
wd: h1 := 4*i;
    i := i+1;
    h2 := h1(a);
    h3 := h1(b);
    nop;
    h4 := h2*h3;
    if i/=n then goto wd;
```

res := res+h4
(2,0)

Beispiel X

```
    res := 0;
    i := 1;
wd: h1 := 4*i;
    i := i+1;
    h2 := h1(a);
    h3 := h1(b);
    nop;
    h4 := h2*h3;
    if i/=n then goto wd;
    nop;
```

res := res+h4
(2,0)

Beispiel XI

```
    res := 0;
    i := 1;
wd: h1 := 4*i;
    i := i+1;
    h2 := h1(a);
    h3 := h1(b);
    nop;
    h4 := h2*h3;
    if i/=n then goto wd;
    nop;
    res := res+h4;
```

Gesamtdauer:
11 Takte, Schleife 9 Takte

Verbesserung durch Schleifenausrollen

```
    res := 0;
    i := 1;
wd: h1 := 4*i;
    j := i+1;
    f1 := 4*j;
    h2 := h1(a);
    h3 := h1(b);
    f2 := f1(a);
    f3 := f1(b);
    h4 := h2*h3;
    f4 := f2*f3;
    i := i+2;
    res := res+h4;
    if i<n then goto wd;
    res := res+f4;
    nop;
    nop;
```

Gesamtdauer der Doppelscheife:
15 Takte

(Behandlung ungerades n fehlt noch, ist aber außerhalb der Schleife möglich)

Weitere Verfahren

- Blockanordnung unter Berücksichtigung interner Verarbeitungseinheiten
- Spuranordnung (trace scheduling)
- Software-Fließband

Verarbeitungseinheiten berücksichtigen

- Fakt: Superskalare Prozessoren zerlegen Befehle in Aufgaben für verschiedene Verarbeitungseinheiten (Befehlsentschlüsselung, Speicherzugriff, ganzzahlige, logische Gleitpunkteinheit,...). Außer externen Kollisionen (noch nicht vorhandene Operanden) gibt es interne Kollisionen, weil die benötigte Verarbeitungseinheit noch belegt ist.
- Verfahren (Thomas Müller):
  - Berechne für Befehlsfolgen $b_1, \ldots, b_n$, meist $n = 2$, die minimale Verzögerung $v(b)$ für jeden Befehl b, damit dieser innerhalb der Sequenz kollisionsfrei ausgeführt werden kann
  - Konstruiere endlichen Automaten: Zustand entspricht Belegungszustand der Verarbeitungseinheiten, Eingabe sind Befehle b, Ausgabe $v(b)$.
  - Konstruiere Abhängigkeitsgraph. Wähle Folgebefehl mit $v(b)$ minimal, gleichzeitig Zustandsübergang im Automat

Spuranordnung

- Betrachte häufig ausgeführte Pfade (Spuren) im Ablaufgraph als Einheit und ordne sie gemeinsam an.
  - Spuren aus Ausführungsprofil ermitteln (falls vorhanden)
  - erweiterte Grundblöcke als Spuren
  - allgemeiner: Spuren sind Wege im Dominanzbaum
- Spuranordnung kann Befehle vorzeitig platzieren, bevor klar ist, dass sie wirklich benötigt werden (spekulative Ausführung)
  - Problem: anschließende Korrekturphase nötig, um unerwünschte Wirkung spekulativer Befehle zu beseitigen
  - Korrekturmethode: spekulative Befehle dürfen nur Register, keinen Speicherzustand verändern; Registerinhalt anschließend nicht mehr lebendig
- Problem: Block- und Spuranordnung verändern (meist: erhöhen) die Anzahl der belegten Register: neue Registerzuteilung nötig

Software-Fließband

- Problem: am Ende des Schleifenrumpfs sind alle Verarbeitungseinheiten frei, werden erst im nächsten Schleifendurchgang wieder gefüllt.
  Wie kann fortlaufende Belegung erreicht werden?
- Methoden:
  - Schleifenausrollen, siehe früheres Beispiel
  - vorzeitiger, spekulativer Start des nächsten Schleifendurchlaufs (mit Vorlauf vor Schleifenbeginn und Korrekturphase nach Schleifenende)
  - vorzeitiger Beginn des nächsten Schleifendurchlaufs (Abschlußbefehle des vorigen Schleifendurchlaufs an den Beginn des nächsten) mit entsprechender Korrekturphase
  - Konsequenz: In den meisten Fällen belegen Werte in aufeinanderfolgenden Durchgängen unterschiedliche Register!

  Bemerkung: Die IA-64 unterstützt diese Maßnahme von der Hardwareseite durch „roulierende Register“.

Inhalt

Registerzuteilung

- Problem: Annahme während der Optimierungsphase: Anzahl verfügbarer Register ist unbeschränkt. Aufgabe: Reduktion auf die tatsächlich verfügbaren, endlich vielen Register.
- Prinzip: Alle Register nach Möglichkeit gleich behandeln. Unterscheide Register nach:
  - Gebrauch durch Hardware festgelegt (Befehlszähler, Bedingungsanzeige, Gleitpunkt-/allgemeine Register, Adressregister, gerade/ungerade z.B. `mult`, ...)
  - Gebrauch durch Konventionen Laufzeitsystem festgelegt
  - freie Register
- Verfahren zur Zuteilung freier Register:
  - Lokale Registerzuteilung mit Freiliste (*on the fly*)
  - *linear-scan register allocation*
  - Graphfärben

Registerzuteilung – Literatur

- G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, *Register Allocation via Coloring*, Computer Languages, 6(1), January 1981.
- P. Briggs, K. Cooper, and L. Torczon, *Improvements to Graph Coloring Register Allocation*, Transactions on Programming Languages and Systems, 16(3), May 1994.
- F. Chow and J. Hennessy, *The PriorityBased Coloring Approach to Register Allocation*, Transactions on Programming Languages and Systems, 12(4), October 1990.
- Traub, Holloway, and Smith, *Quality and Speed in Linearscan Register Allocation*, ACM SIGPLAN '98 Conference on Programming Language Design and Implementation, June 1998, pp. 142151.
- Sebastian Hack, Daniel Grund, Gerhard Goos, *Register allocation for programs in SSA-form*, Andreas Zeller, Alan Mycroft (Ed.), Compiler Construction 2006, Springer, March 2006.

Registerzuteilung: Wann/Wo

Möglichkeiten „Wann“:

- Nach Codeselektion
  - Kosten in Codeselektion von Registerzuteilung abhängig,
  - Auslagerungscode (*spill-code*) von Registerzuteilung abhängig,
  - Bestimmter Code nur mit bestimmten Registern auswählbar.
- Vor Codeselektion
  - Codeselektion definiert Anzahl der benötigten Register
  - Manche Werte werden nie explizit berechnet, z.B. Werte auf Adressierungspfaden
- Codeselektion – Registerzuteilung – Codeselektion
- Während der Codeselektion (*on the fly*)
- Aber, Lebendigkeit nur definiert nach Anordnung der Befehle

Möglichkeiten „Wo“:

- Ausdrücke
- Grundblöcke (lokal)
- Schleifen
- Prozeduren (global)
- Programme

Registerzuteilung – Aufgaben

- Registerzuteilung hat zur Aufgabe die Programmvariablen auf Prozessorregister abzubilden
- Aufgaben im Detail:

  Zuteilen Finde Abbildung von Programmvariablen auf Prozessorregister

  Auslagern Lagere Variablen in den Hauptspeicher aus, falls nicht genug Register verfügbar sind

  Verschmelzen Eliminiere unnötiges Kopieren von Variablen in dem Programm

Inhalt

Partitionierung des Registersatzes

1. Dringend (in Register) verfügbare Werte:
   Kellerpegel, Haldenpegel, Schachtel, etc.
2. Globale Werte – (Prozedur-)globale Zuteilung
3. Zwischenergebnisse in Ausdrucksbäumen
   - 4-5 Register freihalten
   - *on the fly* zuteilen

Beobachtung: In graphbasierter SSA 2. und 3. nicht unterscheidbar.

Lokale Registerzuteilung mit Freiliste (*on the fly*)

- Benutze für Zuteilung nur Register die nicht immer benötigt werden (z.B. Kellerpegel)
- Falls Register benötigt wird, rufe *allocReg*(), falls es nicht mehr benötigt wird, *freeReg*(*r*).
  - *allocReg*(): Gibt ein freies Register zurück, falls keines mehr frei ist, löse Ausname aus. Entferne Register aus der Freiliste.
  - *freeReg*(*r*): Füge Register *r* in die Freiliste ein.
- Achtung: Falls Register fehlen, kann kein Programm erzeugt werden (die ersten Turbo Pascal Übersetzer funktionierten wirklich so), ggf. muss dieses Verfahren um die Möglichkeit des Auslagerns erweitert werden.

Inhalt

linear-scan register allocation

- Die Laufzeit der Graphfärbung ist (sehr) hoch.
- Linear-scan register allocation ist eine Erweiterung des *on the fly* Ansatzes:
  - beschaffe Lebendigkeitsinformation und Ablaufflussgraph.
  - durchlaufe das Programm in umgekehrter Postfixordnung.
  - verarbeite Kandidaten beim Durchlaufen
  - Ein- und Auslagern heuristisch,
    benutzt lokale Informationen

linear-scan – Algorithmus

1. Berechne lebendige Variable
2. Behandle die Register als Behälter mit einem gültigen Wert pro Zeitpunkt
3. Behandle die Register so, als wären sie ein Loch, wenn sie unbenutzt sind
4. In einen Behälter kommen mehrere lebendige Variable, wenn sie disjunkte lebendige Bereiche im Ablaufflussgraphen haben
5. Lebendige Bereiche sind entweder im Speicher oder in Registern
   Ordne einer Variable t ein Register r zu, wenn
   - t noch nicht zugeordnet ist und r ein Loch hat, das groß genug ist um den lebendige Bereiche von t aufzunehmen
   - wähle das r mit dem kleinsten solchen Loch
   - andernfalls lagere den Kandidaten mit den geringsten Kosten in den Speicher aus

$\Longrightarrow$ 1. Durchgang: Zuordnung; 2. Durchgang: Codeanpassung

kann noch verbessert werden: zweite Chance für Variable

Inhalt

Registerzuteilung mit Graphfärbung (nach Chaitin)

Prinzip (Chaitin 1981):

- Konstruiere für jede Prozedur einen Interferenzgraph
- Ecken sind die Variablen (auch temporäre) des Programms
- Ecken e, e' durch Kante verbinden, wenn sie nicht gleichzeitig dasselbe Register belegen können.
  - Grund von Unverträglichkeit: überlappende Lebensdauer.
  - Information: Definition und Benutzung von Werten ($\rightarrow$ SSA!).
- Graphfärbung mit minimaler Farbanzahl (*chromatische Zahl* $\chi(G)$) liefert die Minimalanzahl benötigter Register und gleichzeitig die Registerzuordnung.

Globale Register-Vergabe

- Interferenzgraph (*register inference graph*):
  - Ungerichtet
  - Ecken: symbolische Register – Werte aus Wertnumerierung
  - Kanten: zwischen Ecken, die gleichzeitig aktiven Werten entsprechen
- Konstruktion des Interferenzgraph:
  - Topologisches Sortieren der halbgeordneten Berechnungen in SSA Regionen
    Achtung: Das ist Aufgabe der Befehlsanordnung, für guten Code reicht topologisches Sortieren nicht
  - Definition-Benutzungs-Information explizit im SSA Graphen
  - Registerinterferenz direkt bestimmbar

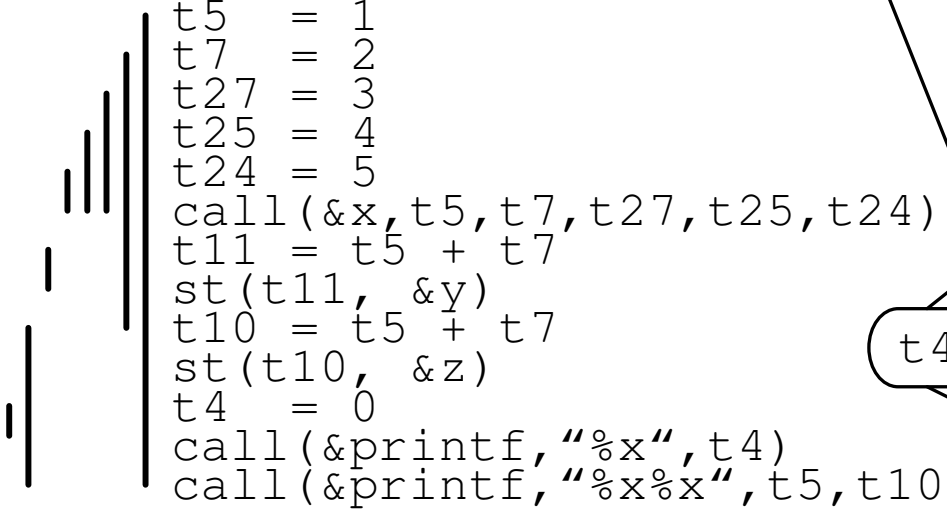
Graphenfärben

- Ist Register-Interferenz-Graph mit k = Anzahl der Register färbbar?
  Aber: Bestimmung der chromatischen Zahl ist *NP-Problem*.
- Hinreichendes Kriterium liefert folgende linear laufende Heuristik:
  1. Wähle Ecke n mit Grad kleiner k aus.
  2. Nicht möglich? Ausgabe: Weiß nicht, ob k-färbbar.
  3. Sonst eliminiere n und seine Kanten.
  4. Gehe zu 1. wenn Graph nicht leer.
     Sonst Ausgabe: k-färbbar.
- Färbe Graphen in umgekehrter Eliminierungsfolge.

Beispiel – Interferenzgraph

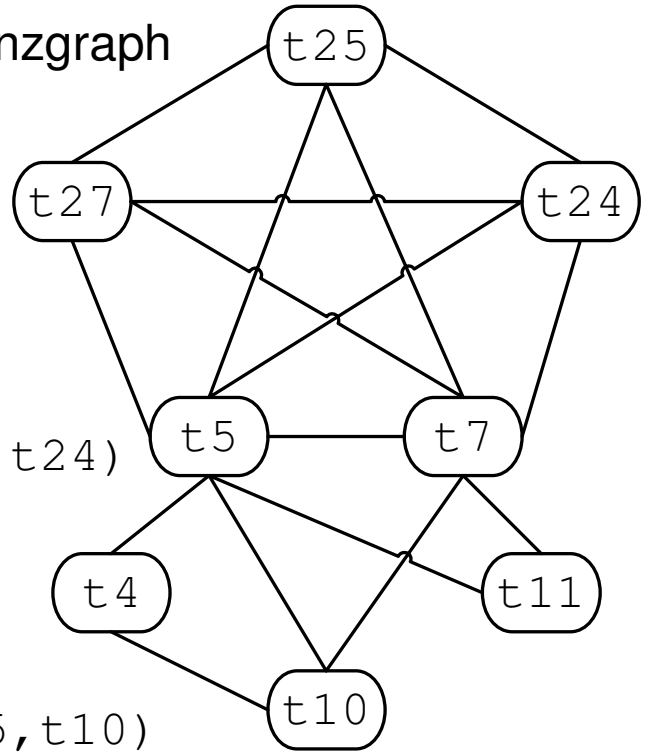
Lebens-
zeiten

Programm



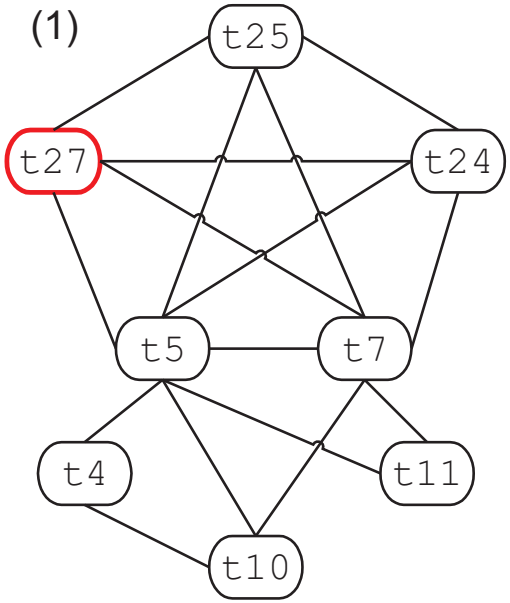
```
t5  = 1
t7  = 2
t27 = 3
t25 = 4
t24 = 5
call(&x,t5,t7,t27,t25,t24)
t11 = t5 + t7
st(t11, &y)
t10 = t5 + t7
st(t10, &z)
t4  = 0
call(&printf,"%x",t4)
call(&printf,"%x%x",t5,t10)
```

Interferenzgraph

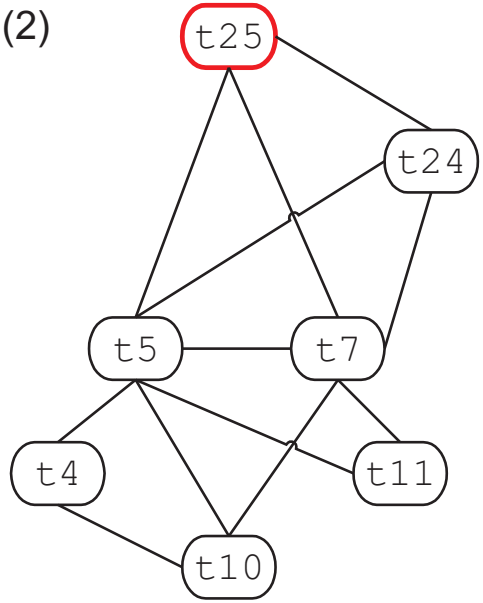


Beispiel – Ecken eliminieren ($k = 5$)

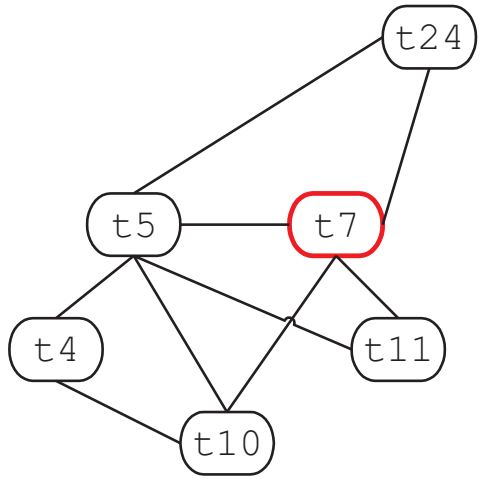
(1)



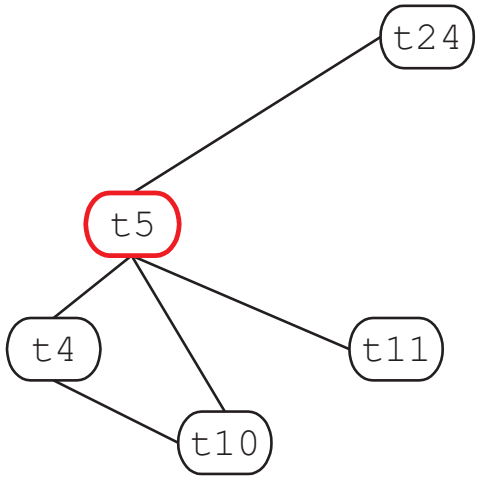
(2)



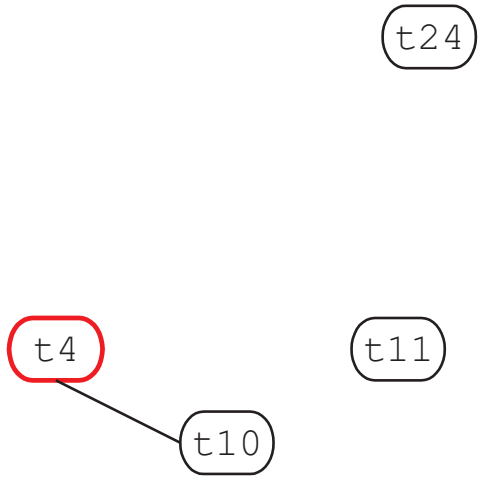
(3)



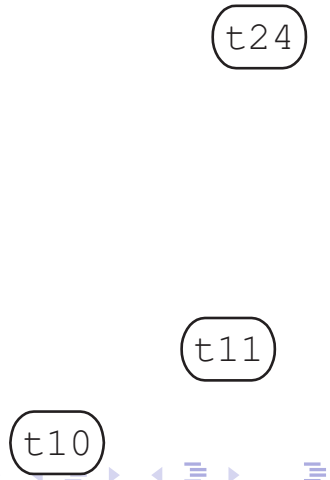
(4)



(5)

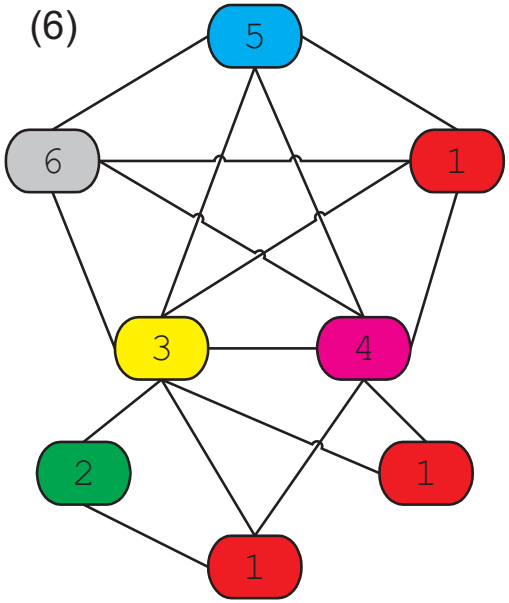


(6)

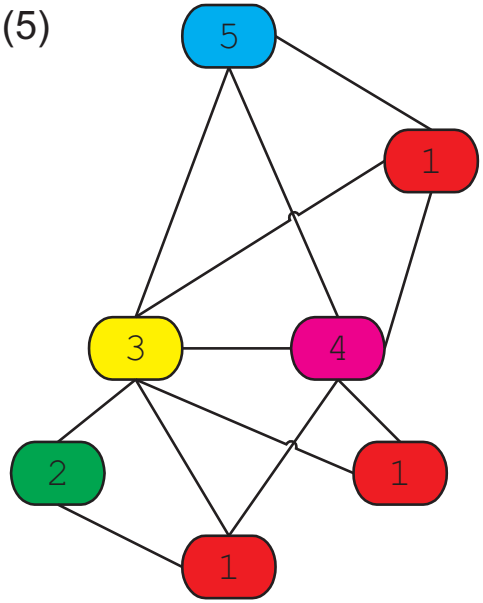


Beispiel – Färbung der Ecken

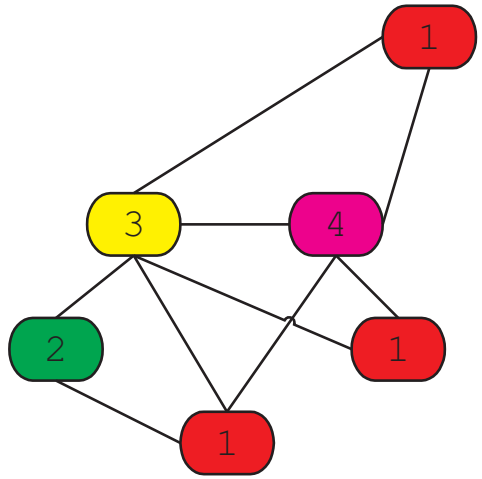
(6)



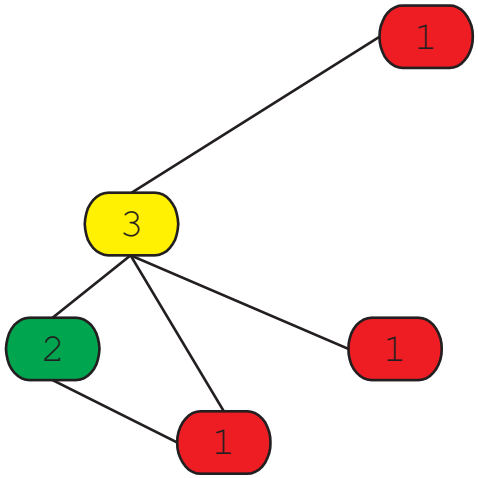
(5)



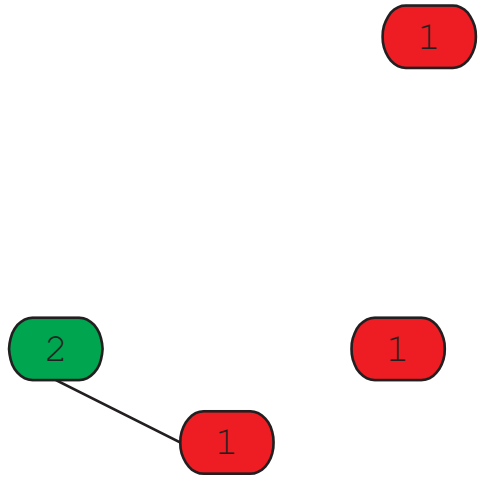
(4)



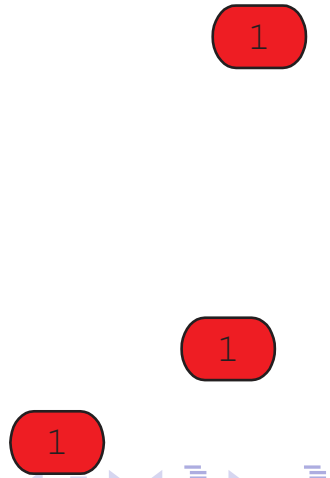
(3)



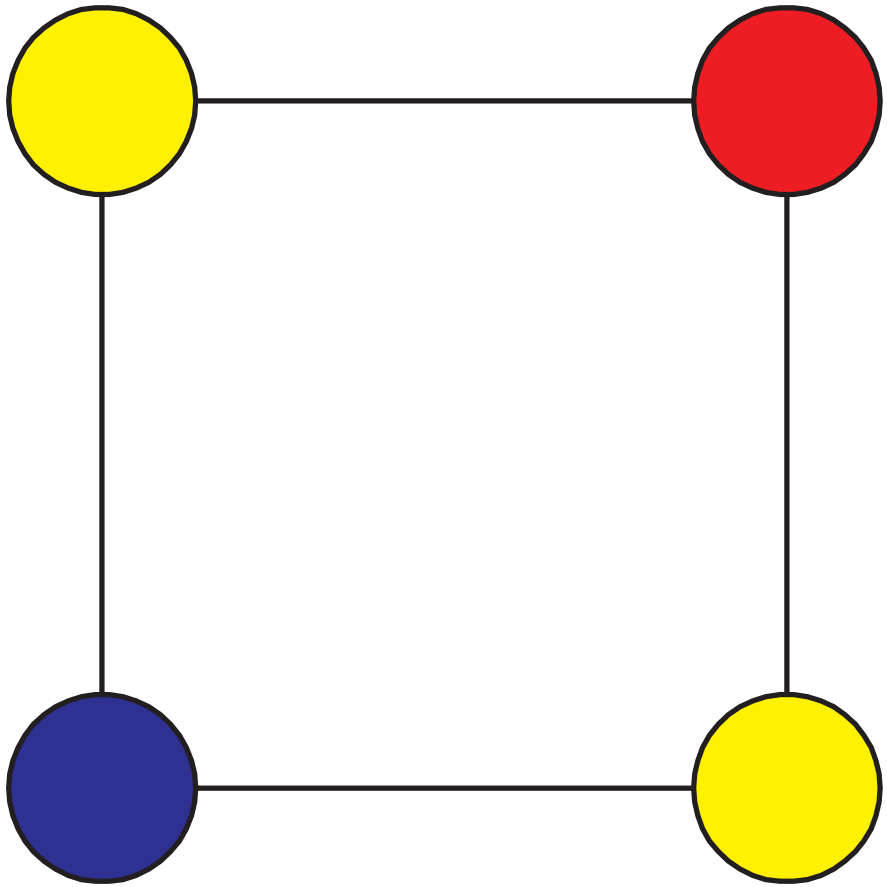
(2)



(1)



Offensichtlich 2-färbbar aber Heuristik scheitert



Heuristik unterstellt, daß alle Nachbarecken unterschiedlich gefärbt sein müssen.

Färbung nicht gefunden

Wenn Färbung nicht gefunden wurde, kann die Heuristik iteriert werden:

- Eliminiere eine Ecke m aus Register-Interferenz-Graph
- Entsprechender Wert kommt nicht in globales Register, sondern wird in den Speicher ausgelagert.
- Neuer Versuch:
  Graphenfärben des Rest-Register-Interferenz-Graphen
- Auswahl von m heuristisch siehe 1. bis 4. der nachfolgenden Rangfolge

Register auslagern

- Genügen die Register nicht („Registerdruck zu hoch“, Graph nicht k-färbbar), so müssen Werte in den Speicher ausgelagert werden
- Auswahl der auszulagernden Werte (Rangfolge):
  1. Wert kann mit einem (oder wenigen) Befehlen aus anderen Registerinhalten wieder berechnet werden
  2. Wert schon im Speicher vorhanden oder mit einem Speicherzugriff wiederberechenbar
  3. Wert wird möglichst lange nicht benötigt
  4. Wert interferiert mit vielen anderen
- Bei 1. und 2. kein Auslagern nötig, 3. nur angenähert beurteilbar, z.B. innerhalb eines Grundblocks
- Probleme:
  - Auslagern kann während der Registerzuteilung, aber auch danach nötig werden, z.B. während Befehlsanordnung
  - Befehlsanordnung kann die Bedingungen verändern

Weitere Verbesserungen

Bevor die Registerzuteilung beginnt (Chow & Hennessy):

- Konstanten aufspalten, d.h. die Konstanten die in SSA-Form zusammengezogen wurden, unmittelbar vor ihrer Verwendung in den Code platzieren.
- Allgemeiner – Rematerialisierung: Werte die sich leicht (wenige Takte $<$ Speicherzugriffzeit) wiederberechnen lassen, nicht in Register belassen

Experimenteller Vergleich

Angegeben: Jeweils Quotient aus *linear-scan* / Graphenfärben

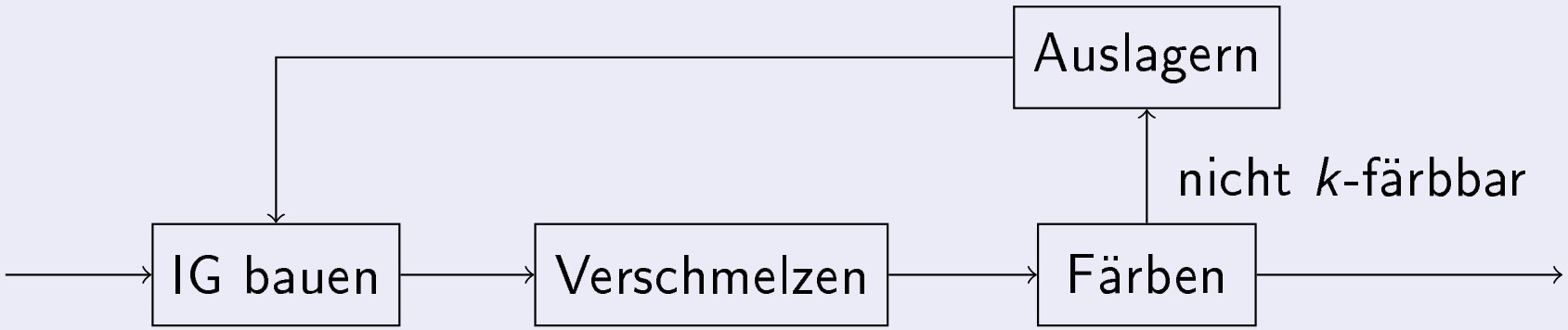
Prozedur	Datenstruktur [Elemente]	Optimiererlaufzeit [s]
cvrin.c	245 / 1061	1.5 / 0.4
twldrv.f	6218 / 51796	3.7 / 8.8
fpppp.f	6697 / 116926	4.5 / 15.8
field()	7611 / 86741	4.9 / 14.9

Benchmark	dyn. Befehlsanzahl	Ausführungszeit
alvinn	1.000	0.995
doduc	1.002	1.018
eqntott	1.000	1.003
espresso	1.013	1.060
fpppp	1.052	1.043
li	1.018	0.966
tomcatv	1.000	0.995
compress	1.002	1.020
m88ksim	1.008	1.024
sort	1.035	1.082
wc	1.000	1.011

Inhalt

Registerzuteilung mit Graphfärbung (nach Chaitin/Briggs)

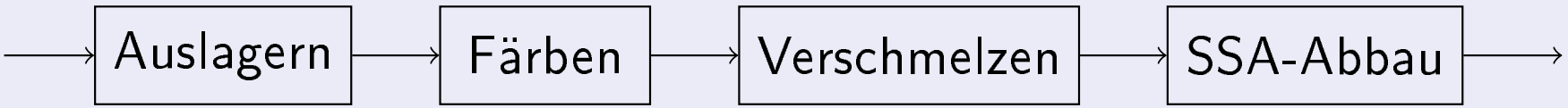
Architektur mit Iteration (non-SSA)



- Jeder ungerichtete Graph kann als Interferenzgraph (IG) auftreten
- Die Bestimmung der chromatischen Zahl ist NP-vollständig
- Färben ist eine Heuristik $\Rightarrow$ Iteration notwendig
  Selbst wenn Heuristik $n + k$ Farben schätzt, muss nach Auslagen von n Registern keine Lösung mit k Registern zu finden sein
- Auslagern ist auf IG fokussiert

Registerzuteilung mit Graphfärbung (nach Hack/Goos)

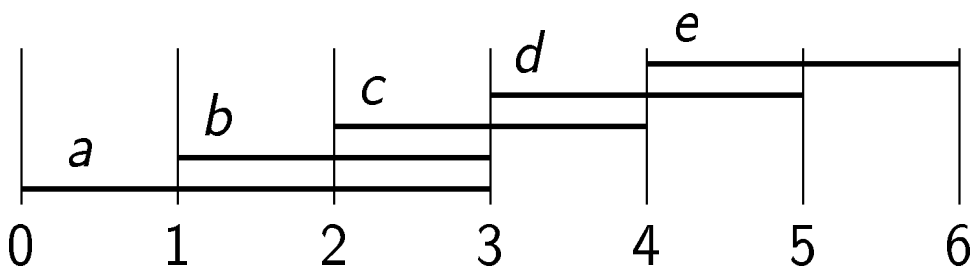
Architektur ohne Iteration (SSA)



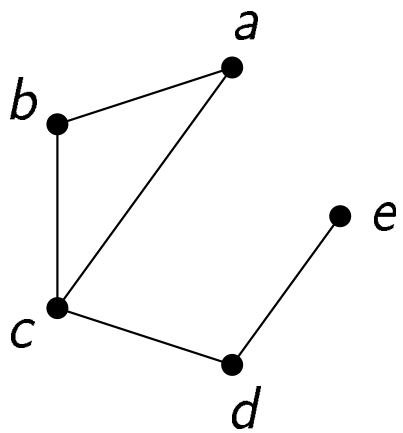
- Wegen Chordalität[1] der SSA IGs:
  - Trennen von Auslagern und Verschmelzen möglich
  - Färben in polynomieller Zeit
- Registerdruck ist ein präzises Maß für Zahl benötigter Register
- Wir wissen vor dem Färben welche Werte ausgelagert werden müssen
  - ▶ Alle Marken bei denen der Registerdruck größer als k ist
- Auslagern kann vor dem Färben geschehen und
- Das Färben scheitert anschließend nie!
- Verschmelzen neu formuliert als Optimierungsproblem mit einer Kostenfunktion für Färbungen ▶ Komplexität ist jetzt hier

[1] Definition folgt noch

Warum sind SSA-IGs chordal? — intuitiv

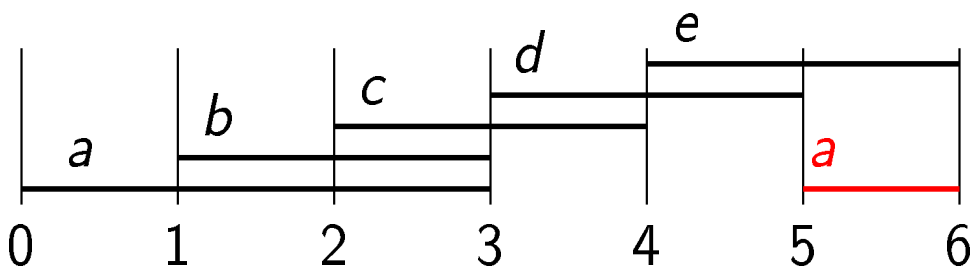


- Jedes Intervall entspricht der Lebenszeit einer Variablen
  - einer Ecke im IG

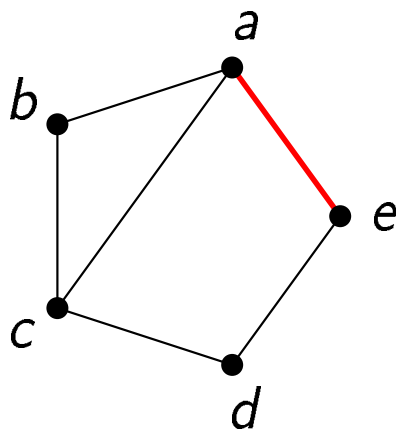


- Kann durch Einfügen einer Kante (a, e) ein Zyklus gebildet werden?

Warum sind SSA-IGs chordal? — intuitiv

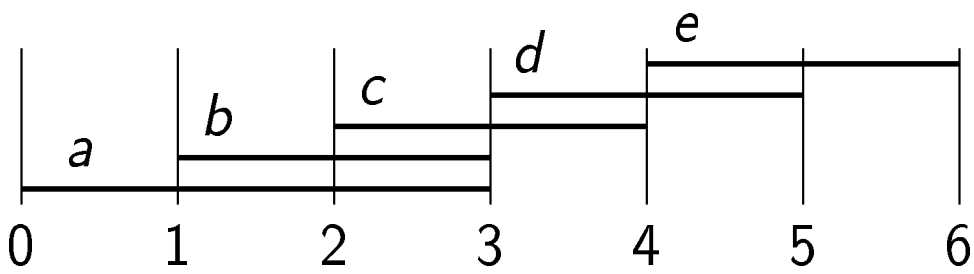


- Jedes Intervall entspricht der Lebenszeit einer Variablen
  - einer Ecke im IG

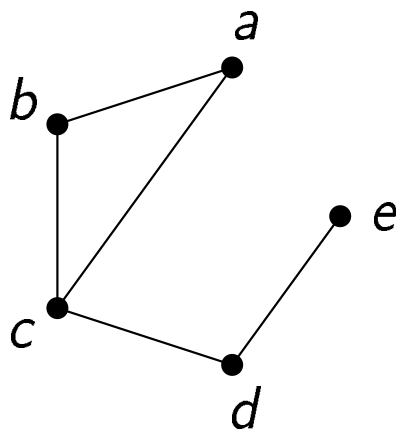


- Kann durch Einfügen einer Kante (a, e) ein Zyklus gebildet werden?
- Das geht nur wenn a erneut bei 5 beginnt

Warum sind SSA-IGs chordal? — intuitiv



- Jedes Intervall entspricht der Lebenszeit einer Variablen
  - einer Ecke im IG

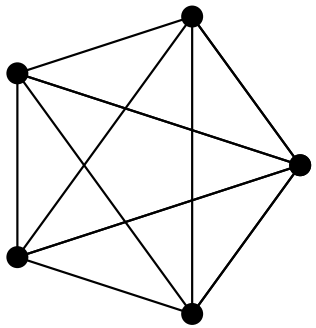


- Kann durch Einfügen einer Kante (a, e) ein Zyklus gebildet werden?
- Das geht nur wenn a erneut bei 5 beginnt
- Das verletzt jedoch die SSA-Eigenschaft, da a 2 Definitionen hätte!

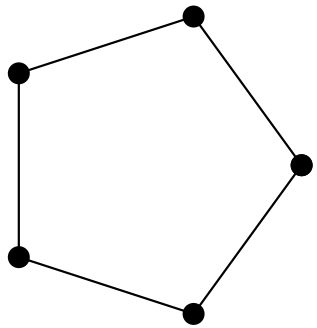
Inhalt

Vollständige Graphen and Zyklen

Grundlagen



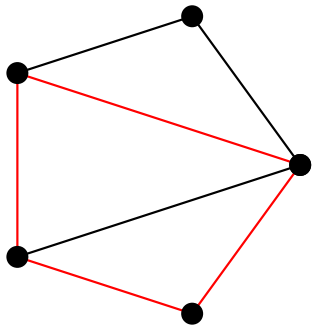
Vollständiger Graph K^5



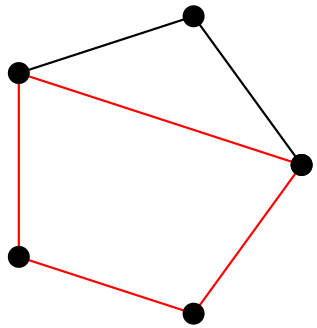
Zyklus C^5

Teilgraphen / Untergraphen

Grundlagen



Graph mit einem
C^4-Teilgraphen

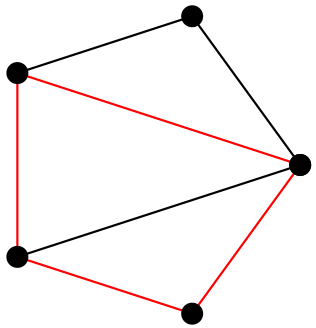


Graph mit einem
C^4-Untergraphen

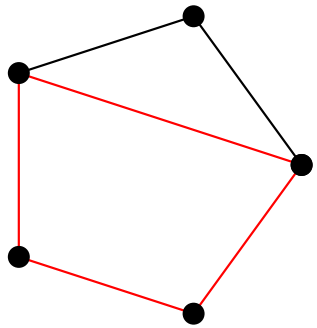
Untergraphen werden auch induzierte Teilgraphen genannt

Teilgraphen / Untergraphen

Grundlagen



Graph mit einem
C^4-Teilgraphen



Graph mit einem
C^4-Untergraphen

Untergraphen werden auch induzierte Teilgraphen genannt

Definition

Komplette Untergraphen heißen Cliquen

Cliquenzahl und chromatische Zahl

Grundlagen

$\omega(G)$ Eckenzahl der größten Clique in G

$\chi(G)$ Anzahl der Farben in einer minimalen Färbung von G

Cliquenzahl und chromatische Zahl

Grundlagen

$\omega(G)$ Eckenzahl der größten Clique in G

$\chi(G)$ Anzahl der Farben in einer minimalen Färbung von G

Korollar

$\omega(G) \leq \chi(G)$ gilt für jeden Graphen G

Cliquenzahl und chromatische Zahl

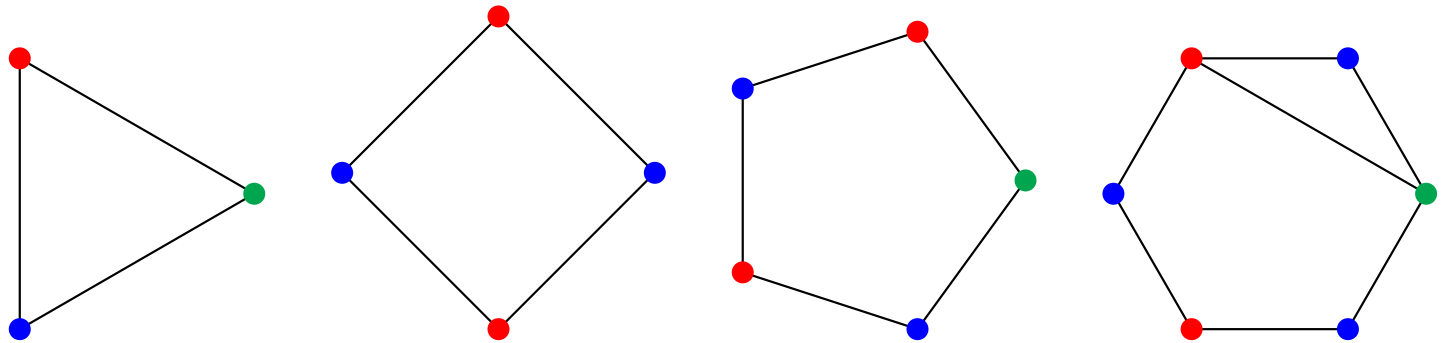
Grundlagen

$\omega(G)$ Eckenzahl der größten Clique in G

$\chi(G)$ Anzahl der Farben in einer minimalen Färbung von G

Korollar

$\omega(G) \leq \chi(G)$ gilt für jeden Graphen G



$\omega(G)$	3	2	2	3
$\chi(G)$	3	2	3	3

Perfekte Graphen

Grundlagen

Definition

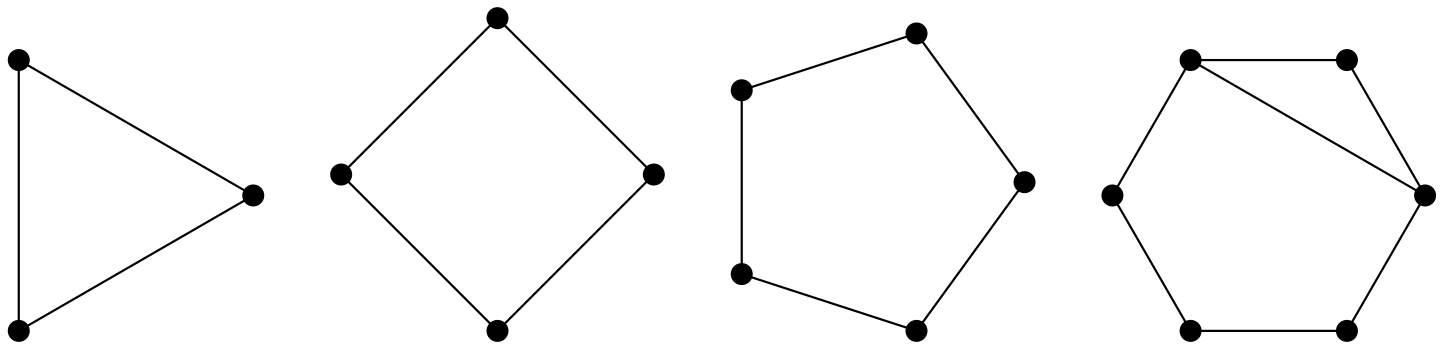
G ist perfekt $\iff$ $\chi(H) = \omega(H)$ gilt für alle Untergraphen H von G

Perfekte Graphen

Grundlagen

Definition

G ist perfekt $\iff$ $\chi(H) = \omega(H)$ gilt für alle Untergraphen H von G



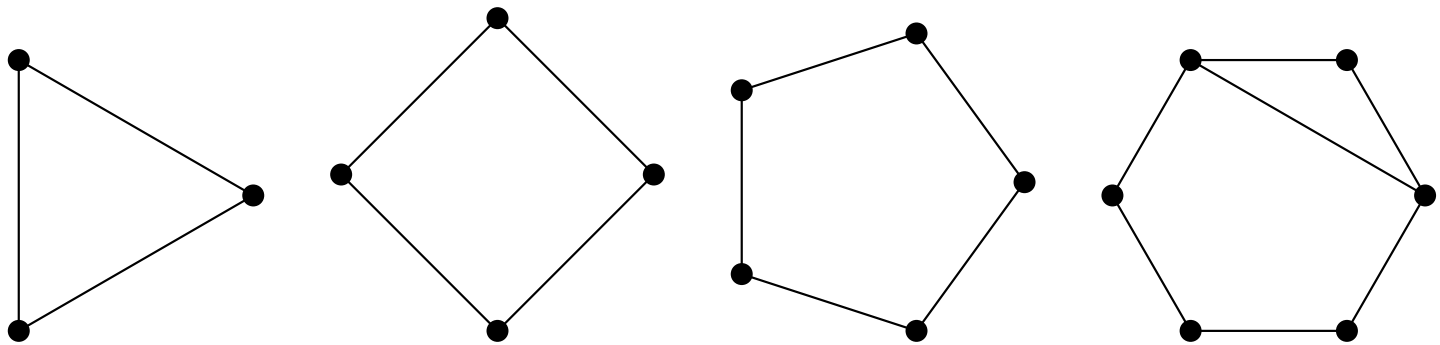
perfekt?

Perfekte Graphen

Grundlagen

Definition

G ist perfekt $\iff$ $\chi(H) = \omega(H)$ gilt für alle Untergraphen H von G



perfekt?	✓	✓		

Chordale Graphen

Grundlagen

Definition

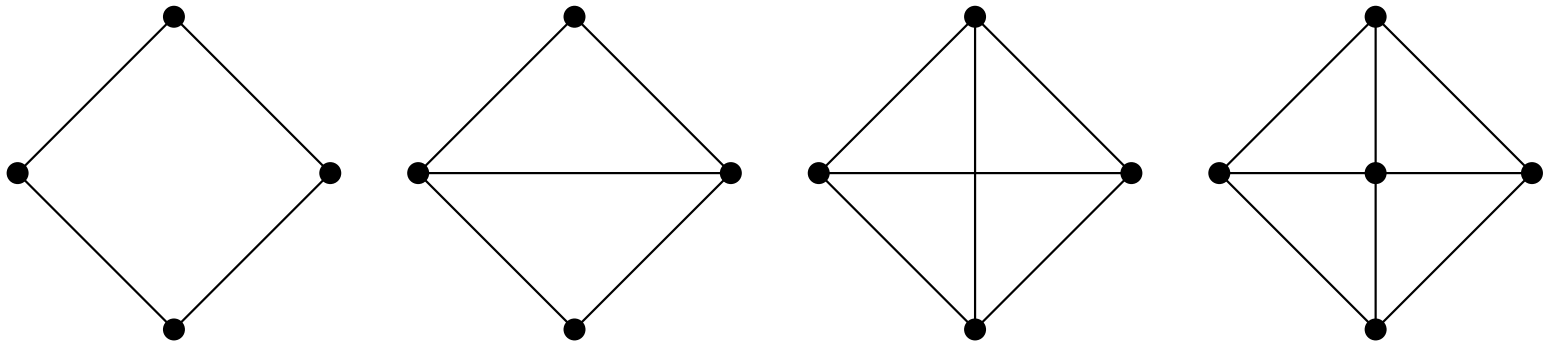
G ist chordal $\iff$ G enthält keine Untergraphen mit Zyklen länger 3

Chordale Graphen

Grundlagen

Definition

G ist chordal $\iff$ G enthält keine Untergraphen mit Zyklen länger 3



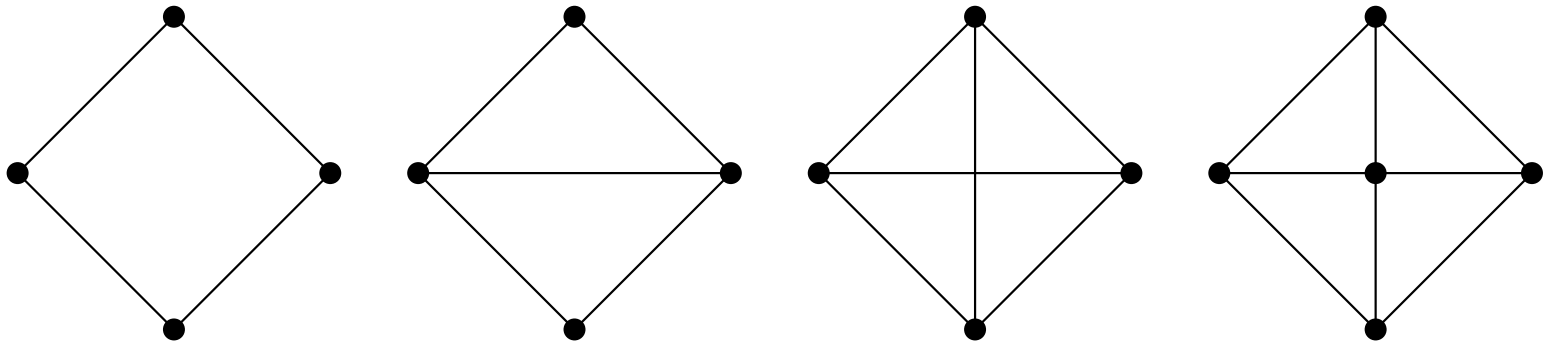
chordal?

Chordale Graphen

Grundlagen

Definition

G ist chordal $\iff$ *G* enthält keine Untergraphen mit Zyklen länger 3



chordal?		✓	✓

Theorem

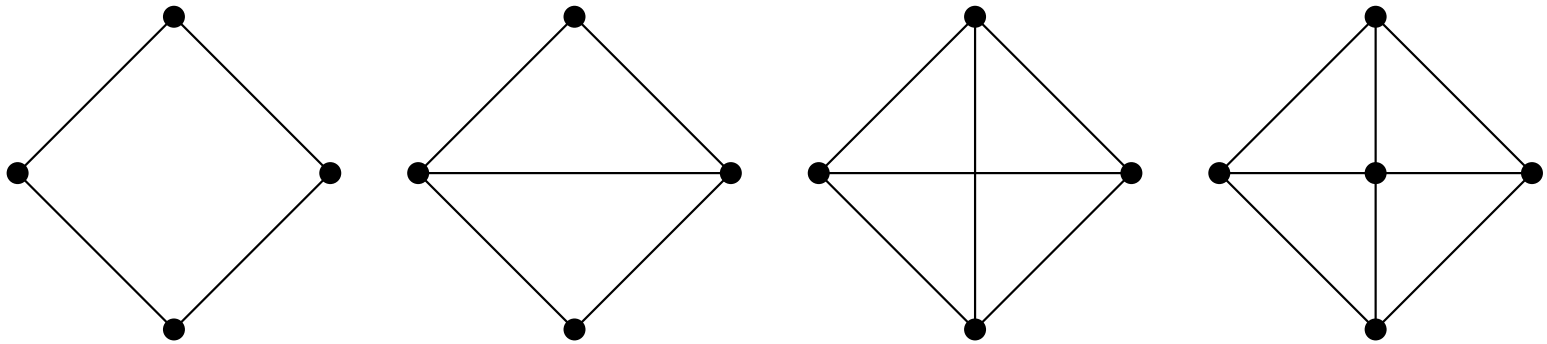
Chordale Graphen sind perfekt

Chordale Graphen

Grundlagen

Definition

G ist chordal $\iff$ G enthält keine Untergraphen mit Zyklen länger 3



chordal?		✓	✓

Theorem

Chordale Graphen sind perfekt

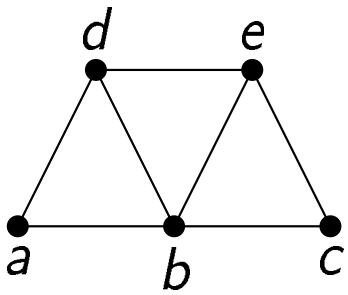
Theorem

Chordale Graphen sind in $O(|V| \cdot \omega(G))$ optimal färbbar

Inhalt

Färbung

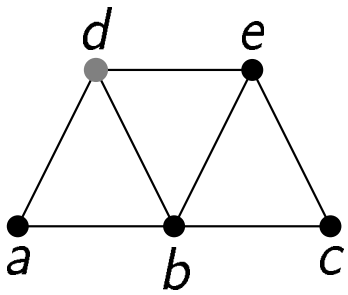
- Entferne schrittweise Ecken aus dem Graphen



Eliminationsschema

Färbung

- Entferne schrittweise Ecken aus dem Graphen

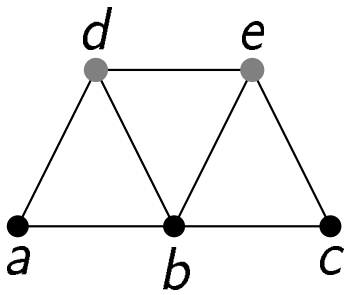


Eliminationsschema

d,

Färbung

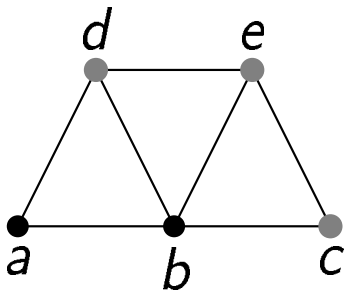
- Entferne schrittweise Ecken aus dem Graphen



Eliminationsschema
d, e,

Färbung

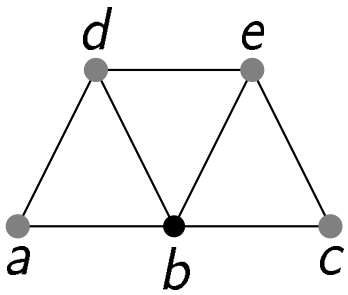
- Entferne schrittweise Ecken aus dem Graphen



Eliminationsschema
d, e, c,

Färbung

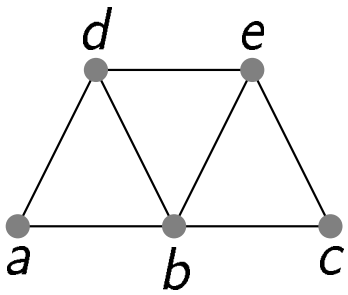
- Entferne schrittweise Ecken aus dem Graphen



Eliminationsschema
d, e, c, a,

Färbung

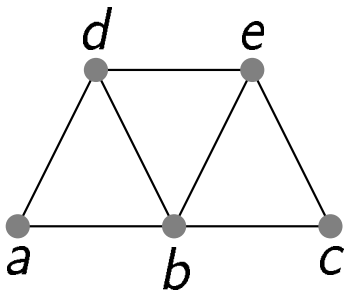
- Entferne schrittweise Ecken aus dem Graphen



Eliminationsschema
d, e, c, a, b

Färbung

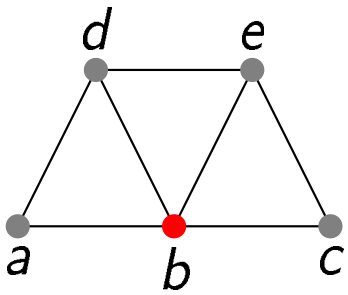
- Entferne schrittweise Ecken aus dem Graphen
- Füge die Ecken in umgekehrter Reihenfolge wieder ein
- Weise jeder Ecke die nächste verfügbare Farbe zu



Eliminationsschema
d, e, c, a, b

Färbung

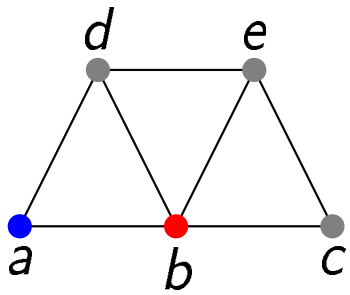
- Entferne schrittweise Ecken aus dem Graphen
- Füge die Ecken in umgekehrter Reihenfolge wieder ein
- Weise jeder Ecke die nächste verfügbare Farbe zu



Eliminationsschema
d, e, c, a,

Färbung

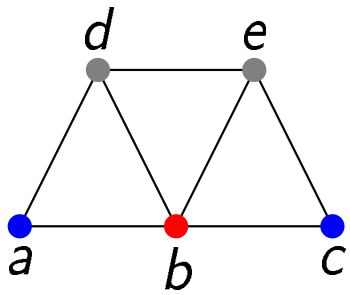
- Entferne schrittweise Ecken aus dem Graphen
- Füge die Ecken in umgekehrter Reihenfolge wieder ein
- Weise jeder Ecke die nächste verfügbare Farbe zu



Eliminationsschema
d, e, c,

Färbung

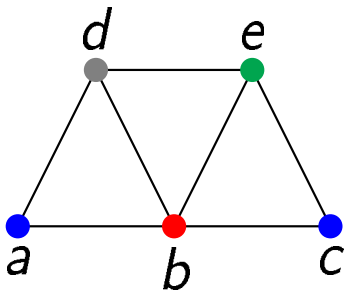
- Entferne schrittweise Ecken aus dem Graphen
- Füge die Ecken in umgekehrter Reihenfolge wieder ein
- Weise jeder Ecke die nächste verfügbare Farbe zu



Eliminationsschema
d, e,

Färbung

- Entferne schrittweise Ecken aus dem Graphen
- Füge die Ecken in umgekehrter Reihenfolge wieder ein
- Weise jeder Ecke die nächste verfügbare Farbe zu

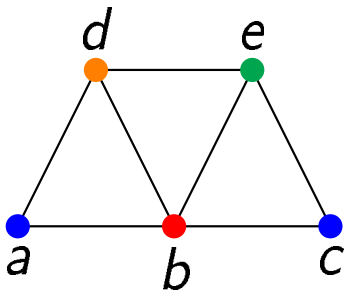


Eliminationsschema

d,

Färbung

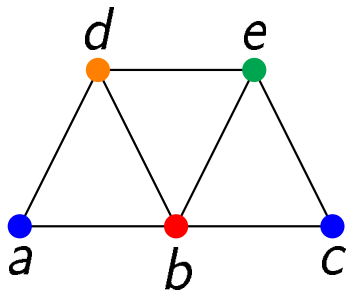
- Entferne schrittweise Ecken aus dem Graphen
- Füge die Ecken in umgekehrter Reihenfolge wieder ein
- Weise jeder Ecke die nächste verfügbare Farbe zu



Eliminationsschema

Färbung

- Entferne schrittweise Ecken aus dem Graphen
- Füge die Ecken in umgekehrter Reihenfolge wieder ein
- Weise jeder Ecke die nächste verfügbare Farbe zu



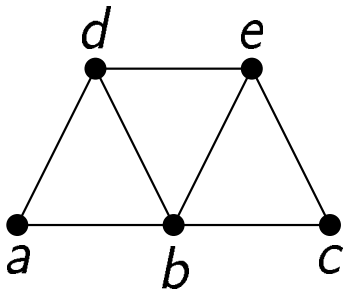
Eliminationsschema

Theorem (Graphentheorie)

Für jeden Graph existiert ein Eliminationsschema, dass zu einer optimalen Färbung führt.

Perfekte Eliminationsschemata

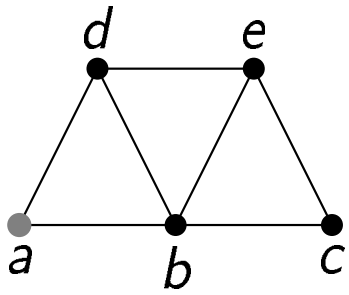
- Annahme: alle (noch nicht eliminierten) Nachbarn einer Ecke n bilden eine Clique



Eliminationschema

Perfekte Eliminationsschemata

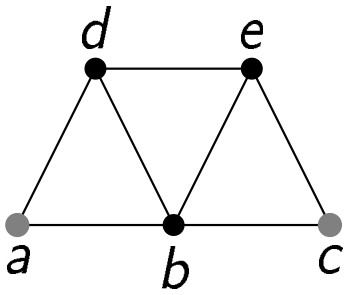
- Annahme: alle (noch nicht eliminierten) Nachbarn einer Ecke n bilden eine Clique



Eliminationschema
a,

Perfekte Eliminationsschemata

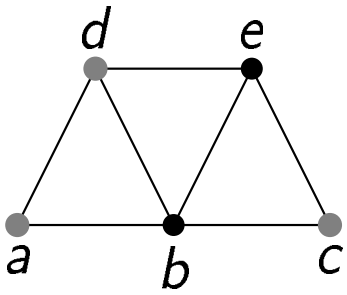
- Annahme: alle (noch nicht eliminierten) Nachbarn einer Ecke n bilden eine Clique



Eliminationschema
a, c,

Perfekte Eliminationsschemata

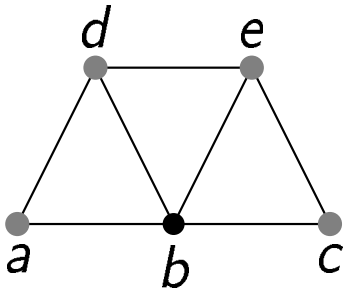
- Annahme: alle (noch nicht eliminierten) Nachbarn einer Ecke n bilden eine Clique



Eliminationschema
a, c, d,

Perfekte Eliminationsschemata

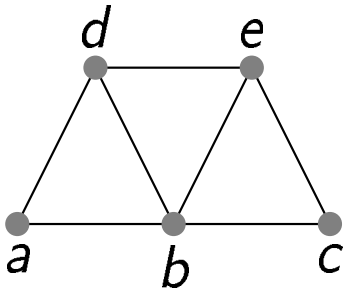
- Annahme: alle (noch nicht eliminierten) Nachbarn einer Ecke n bilden eine Clique



Eliminationschema
a, c, d, e,

Perfekte Eliminationsschemata

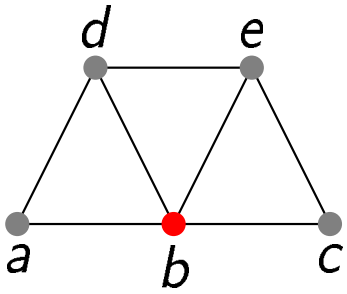
- Annahme: alle (noch nicht eliminierten) Nachbarn einer Ecke n bilden eine Clique



Eliminationschema
a, c, d, e, b

Perfekte Eliminationsschemata

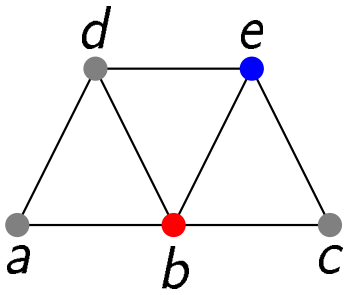
- Annahme: alle (noch nicht eliminierten) Nachbarn einer Ecke n bilden eine Clique



Eliminationschema
a, c, d, e,

Perfekte Eliminationsschemata

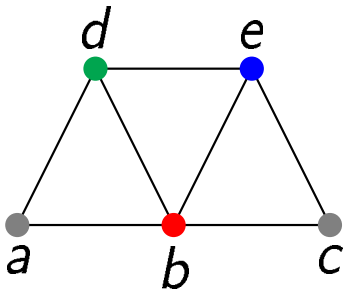
- Annahme: alle (noch nicht eliminierten) Nachbarn einer Ecke n bilden eine Clique



Eliminationschema
a, c, d,

Perfekte Eliminationsschemata

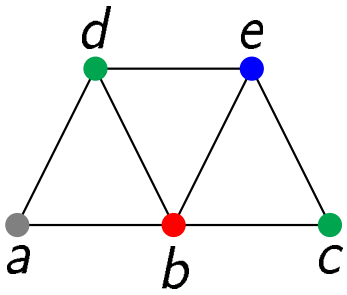
- Annahme: alle (noch nicht eliminierten) Nachbarn einer Ecke n bilden eine Clique



Eliminationschema
a, c,

Perfekte Eliminationsschemata

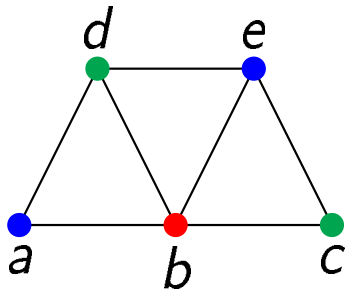
- Annahme: alle (noch nicht eliminierten) Nachbarn einer Ecke n bilden eine Clique



Eliminationschema
a,

Perfekte Eliminationsschemata

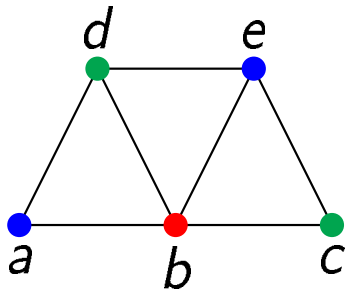
- Annahme: alle (noch nicht eliminierten) Nachbarn einer Ecke n bilden eine Clique



Eliminationschema

Perfekte Eliminationsschemata

- Annahme: alle (noch nicht eliminierten) Nachbarn einer Ecke n bilden eine Clique



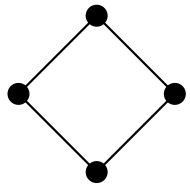
Eliminationschema

Theorem (Graphentheorie)

- *Ein PES erlaubt eine optimale Färbung in polynomieller Zeit*
- *Die Anzahl der benötigten Farben ist durch die größte Clique beschränkt*

Perfekte Eliminationsschemata

- Graphen mit Untergraphen, die Zyklen größer 3 sind, haben keine PES, z.B.

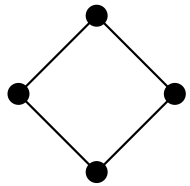


- Graphen, die ein PES besitzen, sind *chordal*
- Chordale Graphen sind *perfekt*, d.h.

$$\chi(H) = \omega(H) \text{ für jedes } H \subseteq G$$

Perfekte Eliminationsschemata

- Graphen mit Untergraphen, die Zyklen größer 3 sind, haben keine PES, z.B.



- Graphen, die ein PES besitzen, sind *chordal*
- Chordale Graphen sind *perfekt*, d.h.

$$\chi(H) = \omega(H) \text{ für jedes } H \subseteq G$$

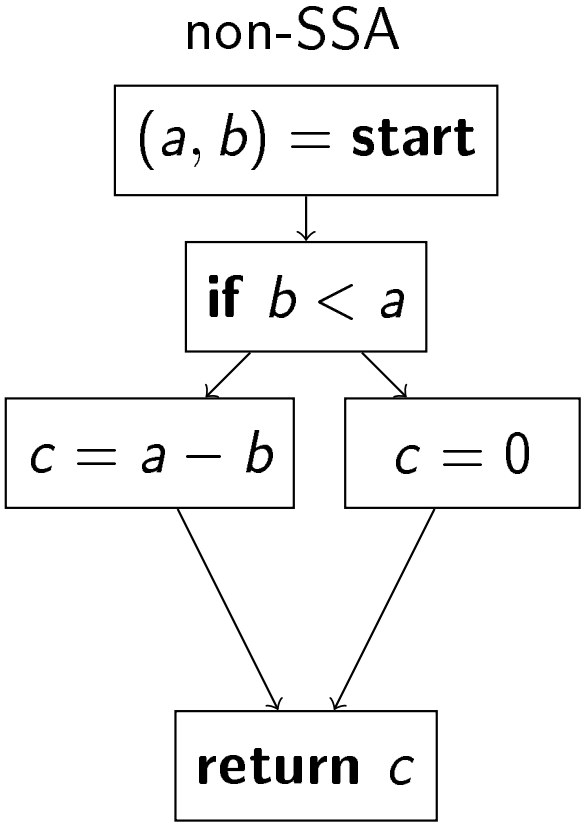
Theorem

- Die Dominanzrelation von SSA-Programmen induziert ein PES im IG.
- SSA IGs sind also chordal

SSA-Form

Beweis – SSA IGs sind chordal

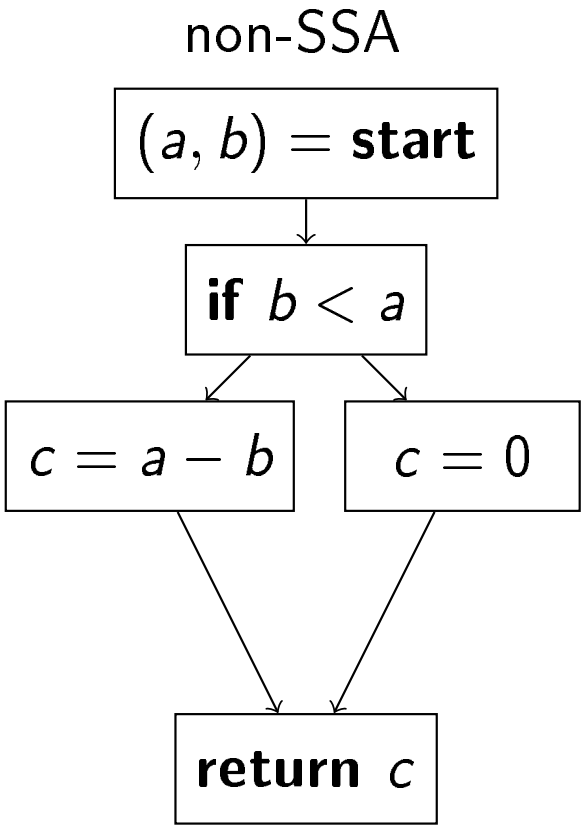
- Jede Variable hat genau eine Definition
  - Variablen sind dynamische Konstanten (Werte)

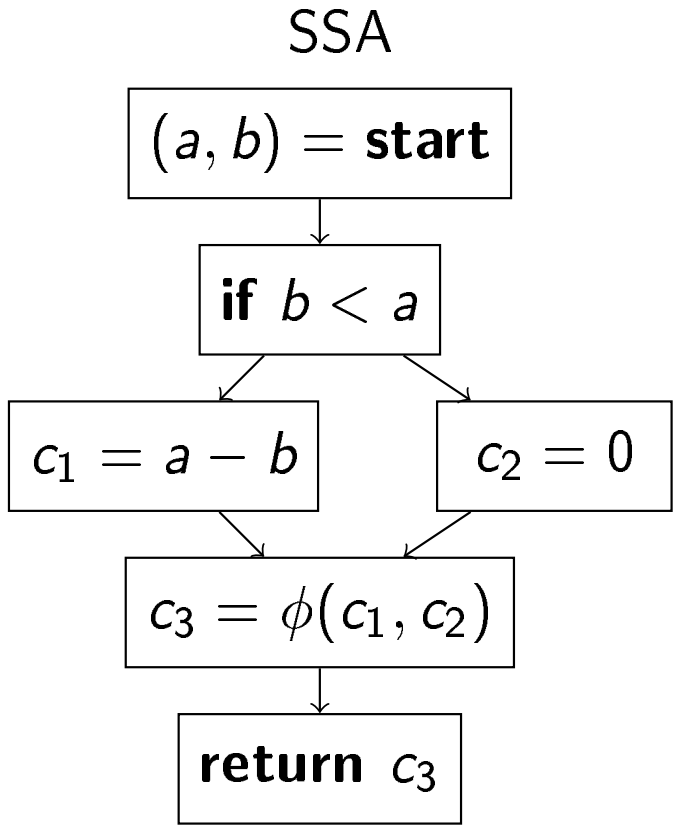


SSA-Form

Beweis – SSA IGs sind chordal

- Jede Variable hat genau eine Definition
  - Variablen sind dynamische Konstanten (Werte)
- ϕ-Funktionen wählen Werte abhängig vom Steuerfluss aus





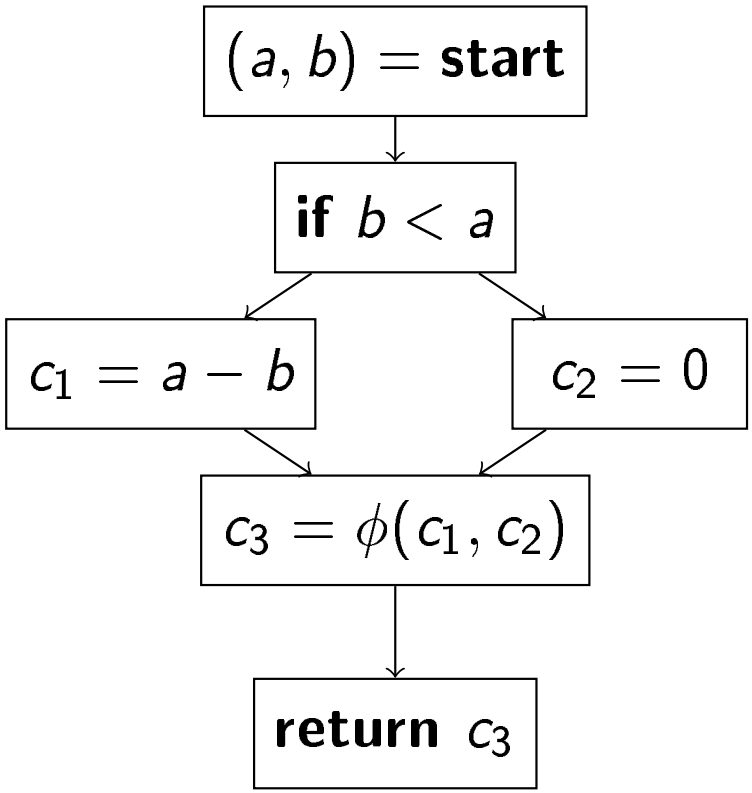
Dominanzrelation

Beweis – SSA IGs sind chordal

Entscheidend für SSA-Programme ist die Dominanzrelation:

Definition

ℓ_1 dominatiert ℓ_2 wenn jeder Pfad von **start** nach ℓ_2 über ℓ_1 führt



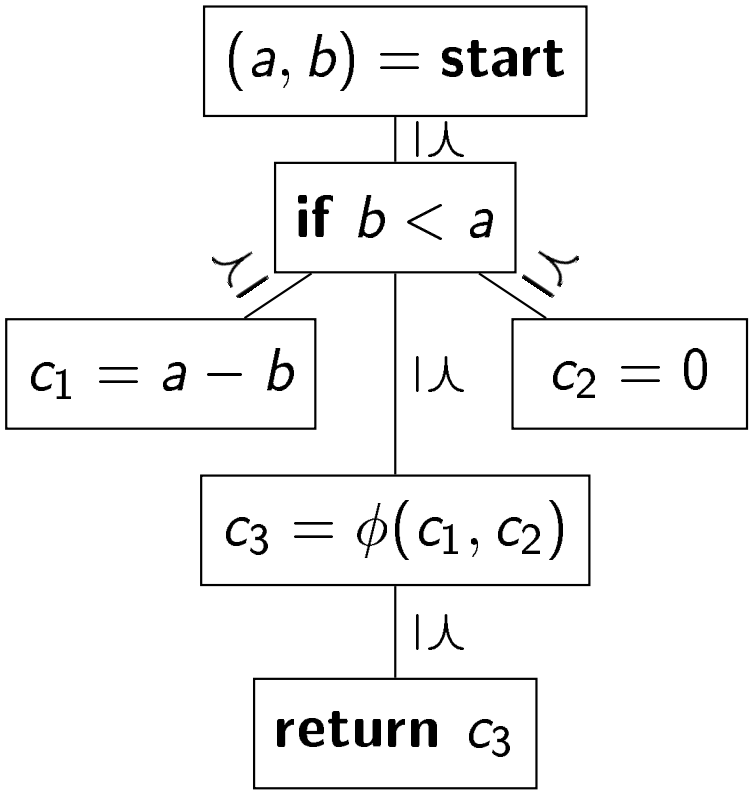
Dominanzrelation

Beweis – SSA IGs sind chordal

Entscheidend für SSA-Programme ist die Dominanzrelation:

Definition

ℓ_1 dominatiert ℓ_2 wenn jeder Pfad von **start** nach ℓ_2 über ℓ_1 führt

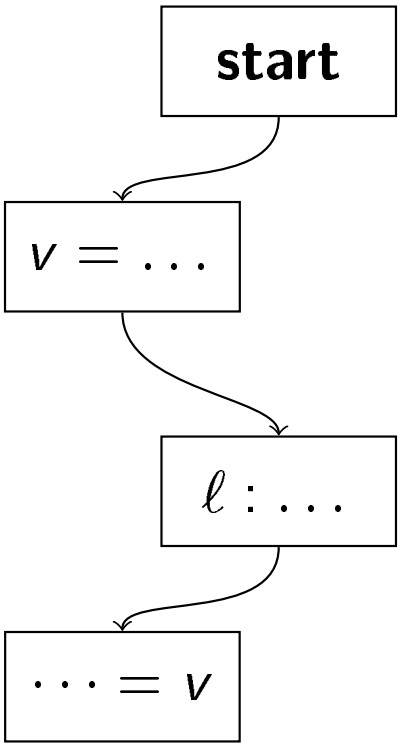


- Jede Ecke hat einen eindeutigen *direkten Dominator*
- Die Dominanzrelation induziert also einen Baum im Steuerflussgraphen
- Die Dominanzrelation ist somit eine Halbordnung

Lebendigkeit und Dominanz

Budimlić, PLDI '02

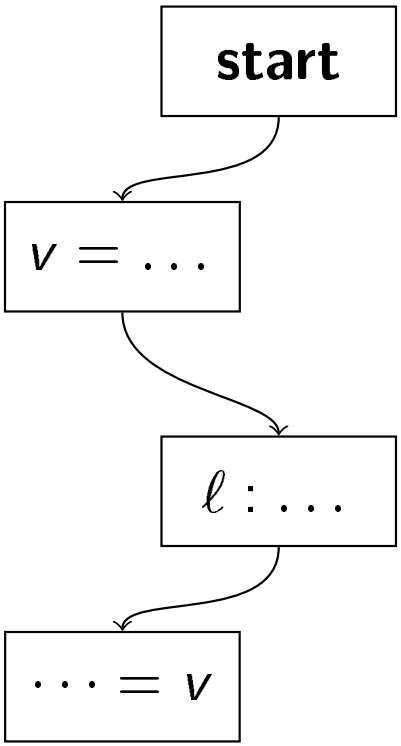
- Jede Programmstelle ℓ, an der ein Wert v lebendig ist, wird durch die Definition von v dominiert.
- Wir schreiben: $\mathcal{D}_v \preceq \ell$



Lebendigkeit und Dominanz

Budimlić, PLDI '02

- Jede Programmstelle ℓ, an der ein Wert v lebendig ist, wird durch die Definition von v dominiert.
- Wir schreiben: $\mathcal{D}_v \preceq \ell$



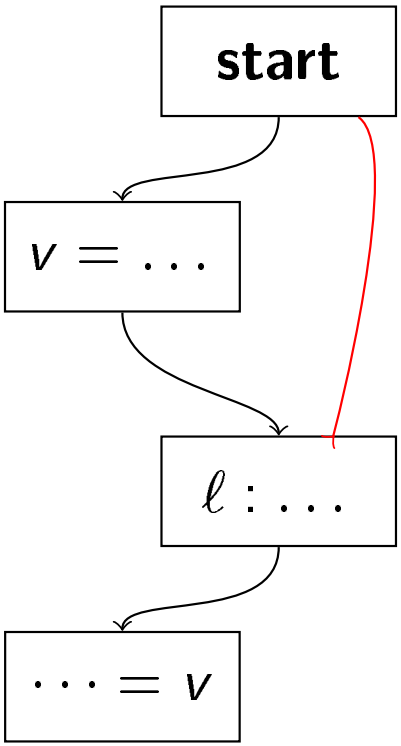
Widerspruchsbeweis

- Sei ℓ nicht von $\mathcal{D}_v$ dominiert.

Lebendigkeit und Dominanz

Budimlić, PLDI '02

- Jede Programmstelle ℓ, an der ein Wert v lebendig ist, wird durch die Definition von v dominiert.
- Wir schreiben: $\mathcal{D}_v \preceq \ell$



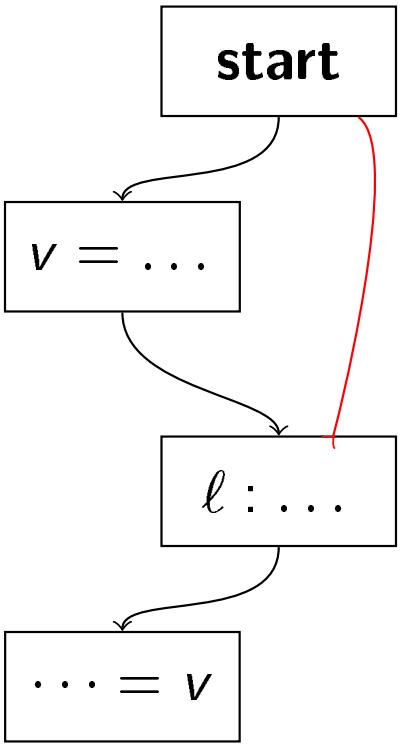
Widerspruchsbeweis

- Sei ℓ nicht von $\mathcal{D}_v$ dominiert.
- Dann existiert ein Pfad von **start** zu einer Benutzung von v, der die Definition von v nicht enthält.

Lebendigkeit und Dominanz

Budimlić, PLDI '02

- Jede Programmstelle ℓ, an der ein Wert v lebendig ist, wird durch die Definition von v dominiert.
- Wir schreiben: $\mathcal{D}_v \preceq \ell$



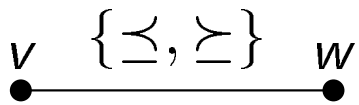
Widerspruchsbeweis

- Sei ℓ nicht von $\mathcal{D}_v$ dominiert.
- Dann existiert ein Pfad von **start** zu einer Benutzung von v, der die Definition von v nicht enthält.
- Widerspruch: Jeder Wert muss vor seiner Benutzung definiert werden.

Interferenz und Dominanz I

Beweis – SSA IGs sind chordal

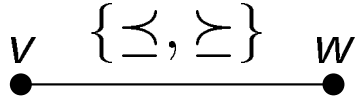
- Annahme: v, w interferieren, d.h. sie sind beide an einer Programmstelle ℓ lebendig
- Dann gilt: $\mathcal{D}_v \preceq \ell$ und $\mathcal{D}_w \preceq \ell$
- Da die Dominanzrelation einen Baum bildet, gilt entweder $\mathcal{D}_v \preceq \mathcal{D}_w$ oder $\mathcal{D}_w \preceq \mathcal{D}_v$



Interferenz und Dominanz I

Beweis – SSA IGs sind chordal

- Annahme: v, w interferieren, d.h. sie sind beide an einer Programmstelle ℓ lebendig
- Dann gilt: $\mathcal{D}_v \preceq \ell$ und $\mathcal{D}_w \preceq \ell$
- Da die Dominanzrelation einen Baum bildet, gilt entweder $\mathcal{D}_v \preceq \mathcal{D}_w$ oder $\mathcal{D}_w \preceq \mathcal{D}_v$

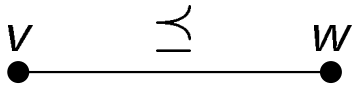


Folgerungen

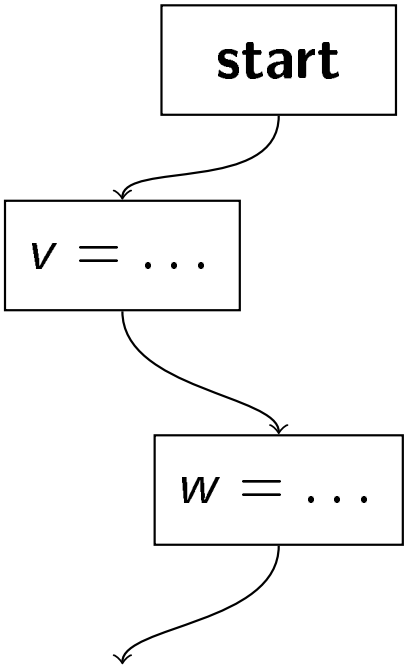
- Jede Kante im IG ist in Bezug auf die Dominanz gerichtet
- Der IG ist ein “Extrakt” der Dominanzrelation

Interferenz and Dominanz II

Budimlić, PLDI '02

- Sei 

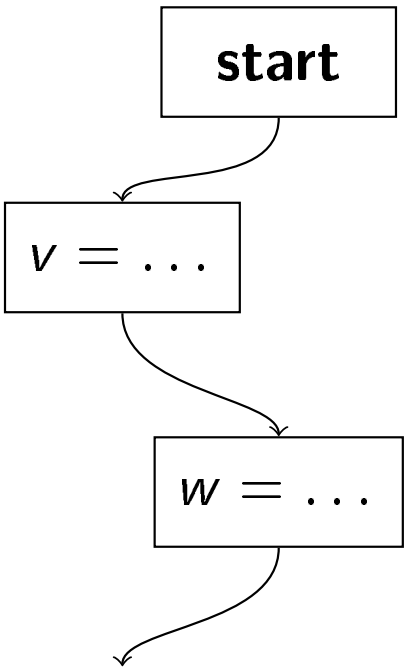
- Dann ist v lebendig an $\mathcal{D}_w$



Interferenz and Dominanz II

Budimlić, PLDI '02

- Sei $v \preceq w$
- Dann ist v lebendig an $\mathcal{D}_w$



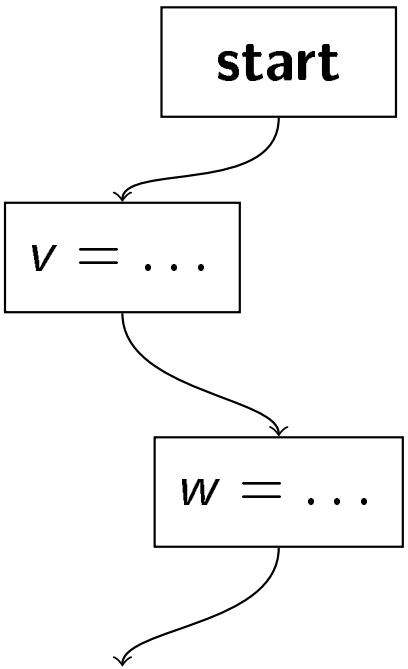
Widerspruchsbeweis

- Sei v nicht lebendig an $\mathcal{D}_w$

Interferenz and Dominanz II

Budimlić, PLDI '02

- Sei $v \preceq w$
- Dann ist v lebendig an $\mathcal{D}_w$



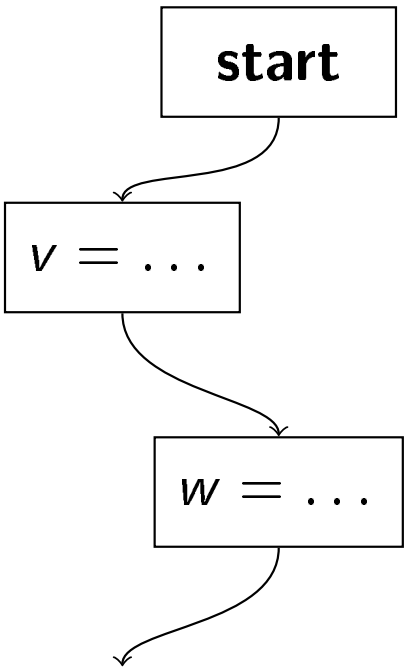
Widerspruchsbeweis

- Sei v nicht lebendig an $\mathcal{D}_w$
- Dann existiert kein Pfad von $\mathcal{D}_w$ zu einer beliebigen Benutzung von v

Interferenz and Dominanz II

Budimlić, PLDI '02

- Sei $v \preceq w$
- Dann ist v lebendig an $\mathcal{D}_w$



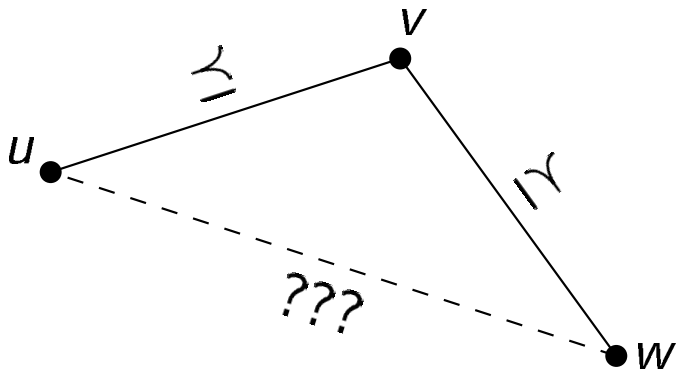
Widerspruchsbeweis

- Sei v nicht lebendig an $\mathcal{D}_w$
- Dann existiert kein Pfad von $\mathcal{D}_w$ zu einer beliebigen Benutzung von v
- Also interferieren v und w nicht $\lightning$

Interferenz and Dominanz III

Beweis – SSA IGs sind chordal

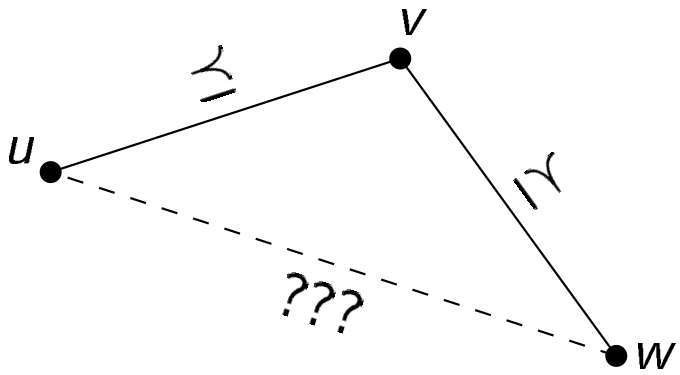
- Betrachten wir drei Ecken u, v, w im IG:



Interferenz and Dominanz III

Beweis – SSA IGs sind chordal

- Betrachten wir drei Ecken u, v, w im IG:

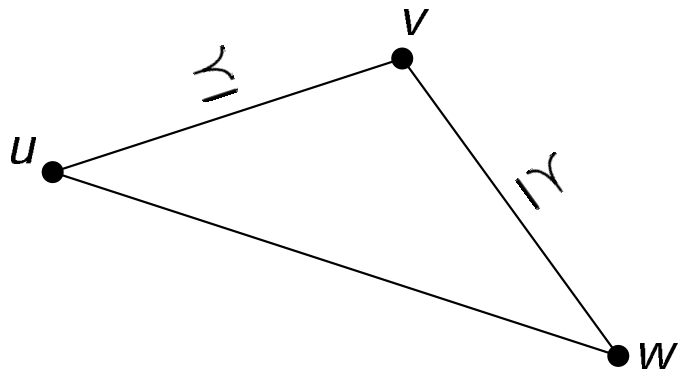


- u, w sind lebendig an $\mathcal{D}_v$

Interferenz and Dominanz III

Beweis – SSA IGs sind chordal

- Betrachten wir drei Ecken u, v, w im IG:



- u, w sind lebendig an $\mathcal{D}_v$
- Somit interferieren sie

Schlussfolgerung

Alle Werte

- Interferieren mit v
- Ihre Definitionen dominieren die Definition von v

sind Mitglieder derselben Clique

Dominanz and PESs

- Bevor ein Wert v zu einem PES hinzugefügt wird, füge alle Werte, deren Definitionen von $\mathcal{D}_v$ dominiert werden, ein
- Somit führt eine Post-Order Besuchsreihenfolge des Dominanzbaums zu einem PES
- IGs von SSA-Programmen können in $O(\chi(G) \cdot |V|)$ gefärbt werden
- Dazu muss der IG selbst nicht einmal berechnet werden

Inhalt

Auslagern

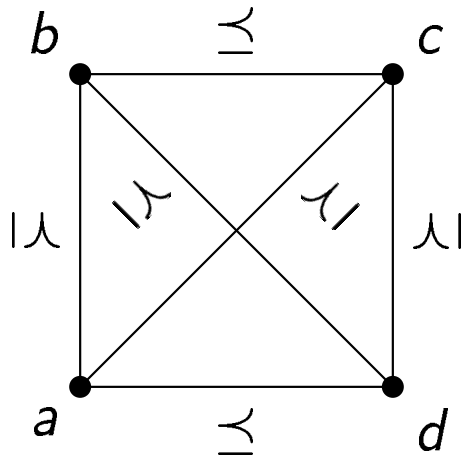
Theorem

Für jede Clique in einem IG existiert eine Programmstelle, an der alle Ecken der Clique lebendig sind.

Auslagern

Theorem

Für jede Clique in einem IG existiert eine Programmstelle, an der alle Ecken der Clique lebendig sind.



- Die Dominanz induziert also eine *totale Ordnung* innerhalb der Clique
  $\Rightarrow$ Es existiert ein "größter" Wert d
- Alle anderen sind lebendig an der Definitionsstelle von d
- Existieren nur drei Register, muss a, b oder c an $\mathcal{D}_d$ ausgelagert sein

Auslagern

Konsequenzen

- Die chromatische Zahl des IG ist exakt durch die Anzahl der lebendigen Werte an den Programmstellen festgelegt
- Reduktion der Anzahl der lebendigen Werte an jeder Programmstelle auf k mach den IG k-färbbar
- Wir kennen a-priori die Programmstellen, an denen Werte ausgelagert werden müssen
  - An allen Programmstellen mit einem Registerdruck größer k
- Auslagern kann also vor dem Färben durchgeführt werden und
- Färben mit k Farben gelingt danach stets

Auslagern

Konsequenzen

- Die chromatische Zahl des IG ist exakt durch die Anzahl der lebendigen Werte an den Programmstellen festgelegt
- Reduktion der Anzahl der lebendigen Werte an jeder Programmstelle auf k mach den IG k-färbbar
- Wir kennen a-priori die Programmstellen, an denen Werte ausgelagert werden müssen
  - An allen Programmstellen mit einem Registerdruck größer k
- Auslagern kann also vor dem Färben durchgeführt werden und
- Färben mit k Farben gelingt danach stets

Schlussfolgerung

- Im Gegensatz zu Chaitin/Briggs-Zuteilern keine Iteration notwendig
- Der IG muss höchstens einmal aufgebaut werden (falls überhaupt)

Auslagern

Allgemein zu lösende Probleme beim Auslagern

- Wenn mehrere Werte ausgelagert werden könnten, welchen wählen?
- Wo genau platziert man die Auslagerungs-/Einlagerungsinstruktionen?
- Ist die Wiederberechnung (Rematerialisierung) manchmal günstiger als ein Auslagern?
  - Konstanten
  - Prozedurargumente auf dem Keller
  - Einfache Berechnungen ...
- Auslagerungsplätze zusammenfassen (um Platz zu sparen)?

Auslagern

Allgemein zu lösende Probleme beim Auslagern

- Wenn mehrere Werte ausgelagert werden könnten, welchen wählen?
- Wo genau platziert man die Auslagerungs-/Einlagerungsinstruktionen?
- Ist die Wiederberechnung (Rematerialisierung) manchmal günstiger als ein Auslagern?
  - Konstanten
  - Prozedurargumente auf dem Keller
  - Einfache Berechnungen ...
- Auslagerungsplätze zusammenfassen (um Platz zu sparen)?

Farach und Liberatore

- Optimales Auslagern innerhalb eines Grundblocks ist NP-vollständig.

Auslagern

Spezielle SSA-Probleme

- ϕ-Funktionen werden gleichzeitig und am Anfang eines Grundblocks ausgeführt, deshalb kann weder vor noch zwischen ihnen aus-/eingelagert werden
- Deshalb müssen Werte vor Betreten des Grundblocks ausgelagert werden
  - Argumente von ϕ-Funktionen sind ausgelagert
  - ϕ-Funktionen bei denen alle Argumente ausgelagert sind benötigen kein Register
  - Vorsicht: Beim SSA-Abbau entstehen hier Speicherkopien statt Registerkopien

Auslagern

Implementierung

- Heuristik nach Belady:
  - Bevorzuge diejenigen Werte bei der Auslagerung, deren nächste Benutzung am weitesten in der Zukunft liegt
    - Was ist die "Entfernung" bei Verzweigungen?
  - Beachte: Benutzungen außerhalb von Schleifen sind stets "weiter" in der Zukunft als innerhalb!
  - Einlagerungsinstruktionen möglich vor einer Schleife platzieren
- Auch als ILP formulierbar

Inhalt

SSA-Abbau

- Gegeben eine k-Färbung eines SSA-IGs
- Können wir daraus eine gültige Registerzuteilung mit k Registern für das zugehörige non-SSA Programm erzeugen?

SSA-Abbau

- Gegeben eine k-Färbung eines SSA-IGs
- Können wir daraus eine gültige Registerzuteilung mit k Registern für das zugehörige non-SSA Programm erzeugen?

Zentrale Frage

Wie behandeln wir ϕ-Funktionen?

ϕ-Funktionen

- Alle ϕ-Funktionen in einem Grundblock

$$\begin{aligned} y_1 &\leftarrow \phi(x_{11}, \ldots, x_{n1}) \\ &\vdots \\ y_m &\leftarrow \phi(x_{1m}, \ldots, x_{nm}) \end{aligned}$$

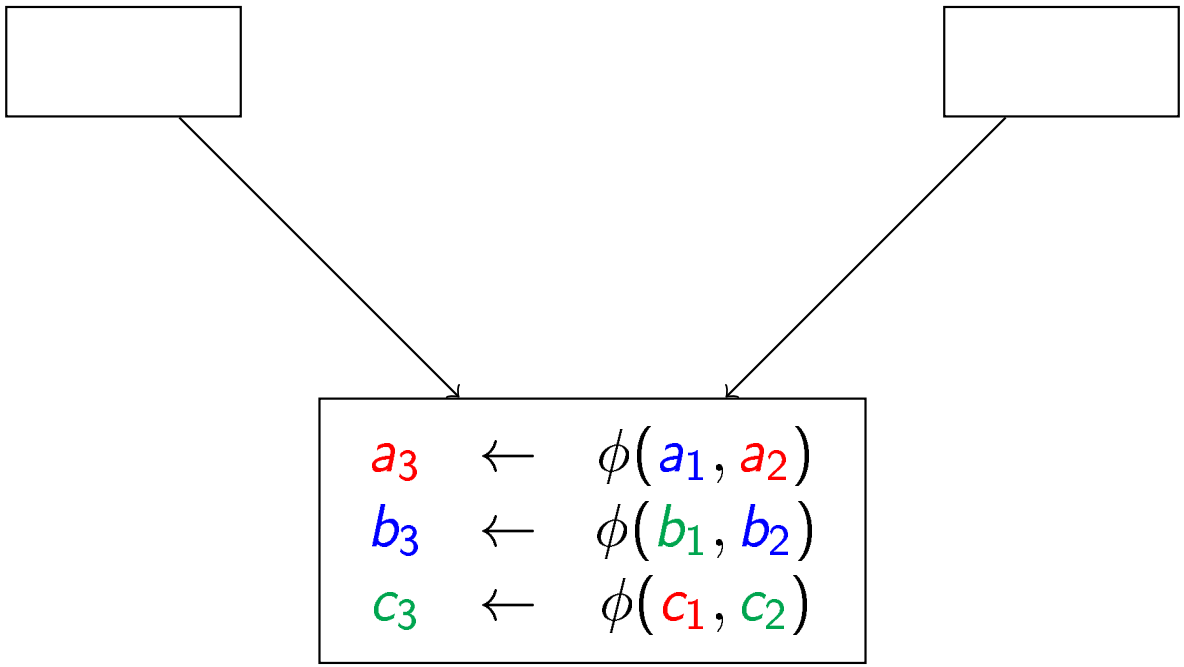
  werden gleichzeitig und vor allen anderen Instruktionen in diesem Grundblock ausgeführt.

- Betreten wir diesen Block über die i-te Kante, wirken die ϕ-Funktionen wie eine parallele Kopierinstruktion

$$(y_1, \ldots, y_m) \leftarrow (x_{i1}, \ldots, x_{im})$$

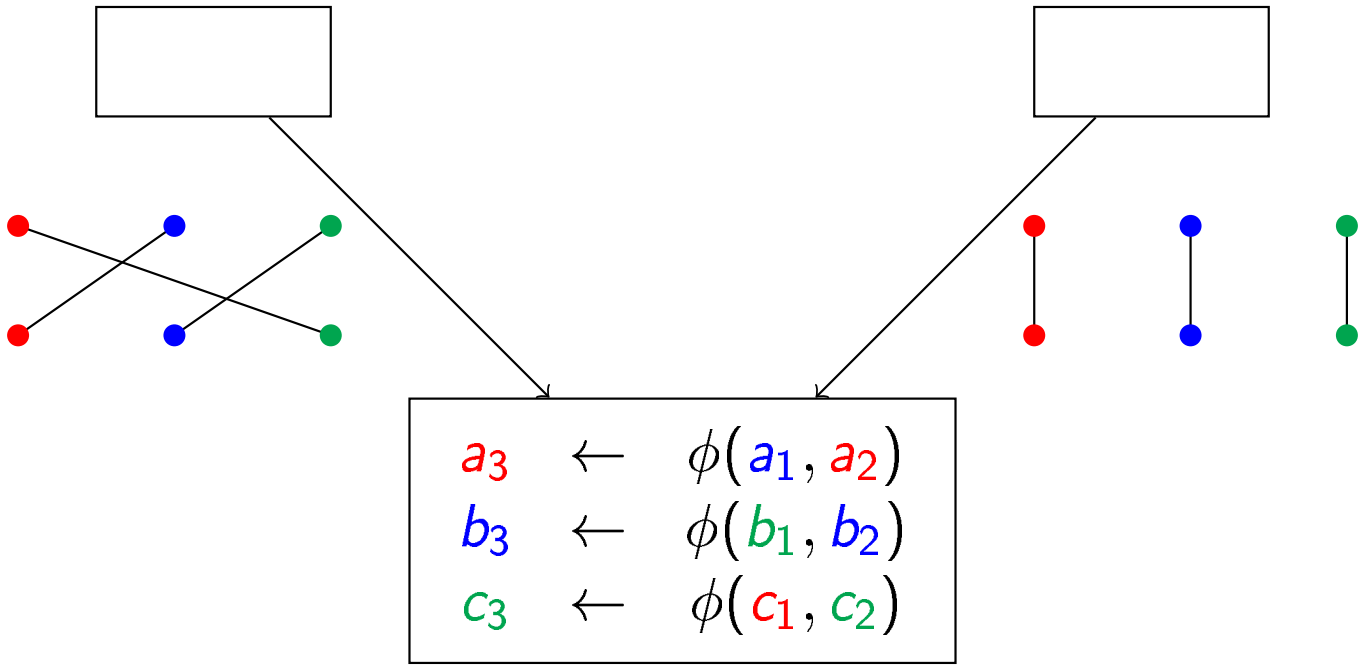
ϕ-Funktionen

- Beispiel



ϕ-Funktionen

- Beispiel



- Die ϕs stellen Registerpermutationen auf den Steuerflusskanten dar

ϕ-Funktionen

- Beispiel

$$
\begin{aligned}
a_3 &\leftarrow \phi(a_1, a_2) \\
b_3 &\leftarrow \phi(b_1, b_2) \\
c_3 &\leftarrow \phi(c_1, c_2)
\end{aligned}
$$

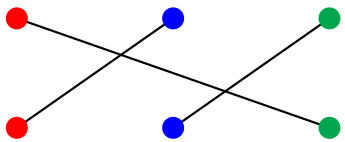
ϕ-Funktionen

- Beispiel

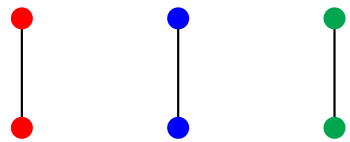
$$
\begin{aligned}
a_3 &\leftarrow \phi(a_1, a_2) \\
b_3 &\leftarrow \phi(b_1, b_2) \\
c_3 &\leftarrow \phi(c_1, c_2)
\end{aligned}
$$

- Die ϕs stellen Registerpermutationen auf den Steuerflusskanten dar

Auf Kante 1

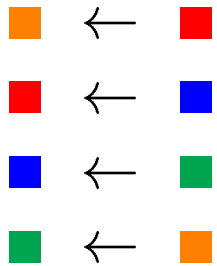


Auf Kante 2



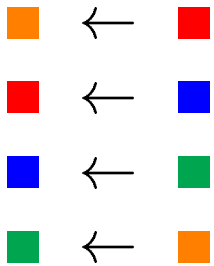
Permutationen

- Eine Permutation kann mit Hilfe von Kopien implementiert werden, wenn ein Hilfsregister ■ verfügbar ist

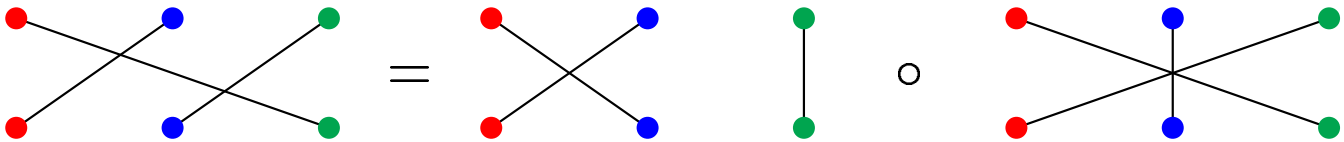


Permutationen

- Eine Permutation kann mit Hilfe von Kopien implementiert werden, wenn ein Hilfsregister ■ verfügbar ist

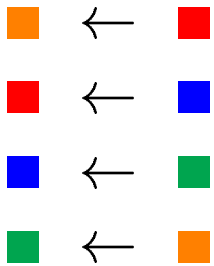


- Permutationen können als Folge von Transpositionen (also Vertauschungen) implementiert werden

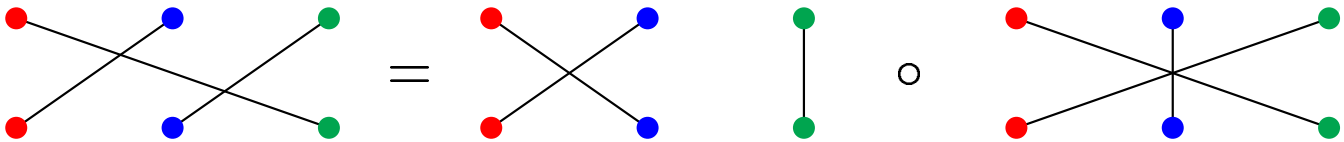


Permutationen

- Eine Permutation kann mit Hilfe von Kopien implementiert werden, wenn ein Hilfsregister ■ verfügbar ist



- Permutationen können als Folge von Transpositionen (also Vertauschungen) implementiert werden



- Eine Transposition kann als Folge von drei `xor`-Instruktionen ohne Verwendung eines zusätzlichen Registers implementiert werden

Unterschiede zum klassischen SSA-Abbau

Parallele Kopien

$$(a', b', c', d') \leftarrow (a, b, c, d)$$

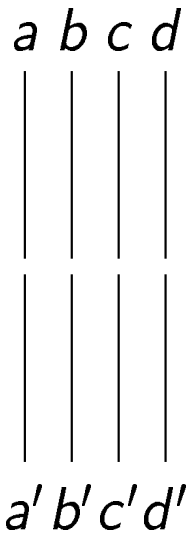
Sequentielle Kopien

$$\begin{aligned} d' &\leftarrow d \\ c' &\leftarrow c \\ b' &\leftarrow b \\ a' &\leftarrow a \end{aligned}$$

Unterschiede zum klassischen SSA-Abbau

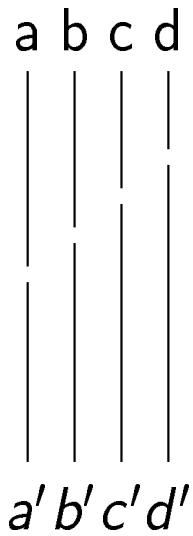
Parallele Kopien

$$(a', b', c', d') \leftarrow (a, b, c, d)$$



Sequentielle Kopien

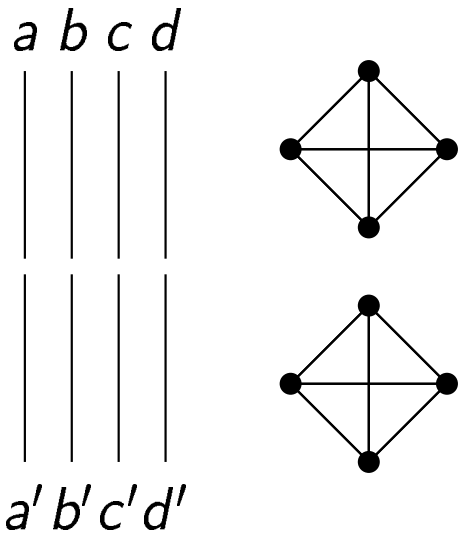
$$\begin{aligned} d' &\leftarrow d \\ c' &\leftarrow c \\ b' &\leftarrow b \\ a' &\leftarrow a \end{aligned}$$



Unterschiede zum klassischen SSA-Abbau

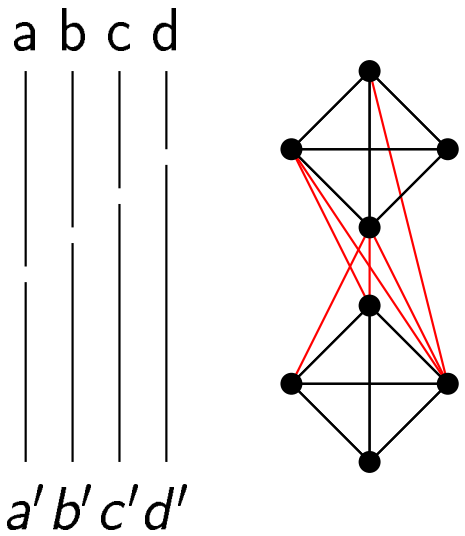
Parallele Kopien

$$(a', b', c', d') \leftarrow (a, b, c, d)$$



Sequentielle Kopien

$$\begin{aligned} d' &\leftarrow d \\ c' &\leftarrow c \\ b' &\leftarrow b \\ a' &\leftarrow a \end{aligned}$$



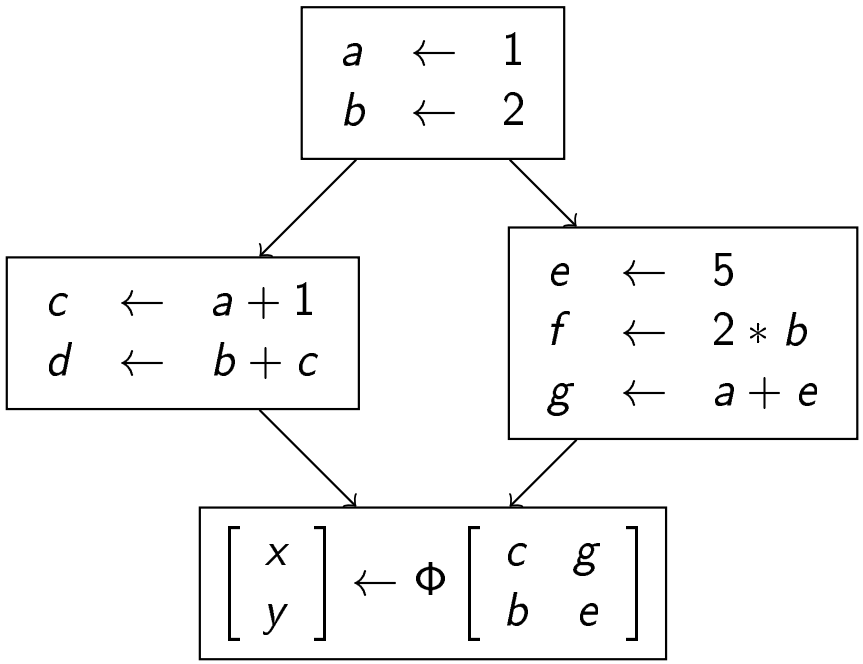
Verschmelzung

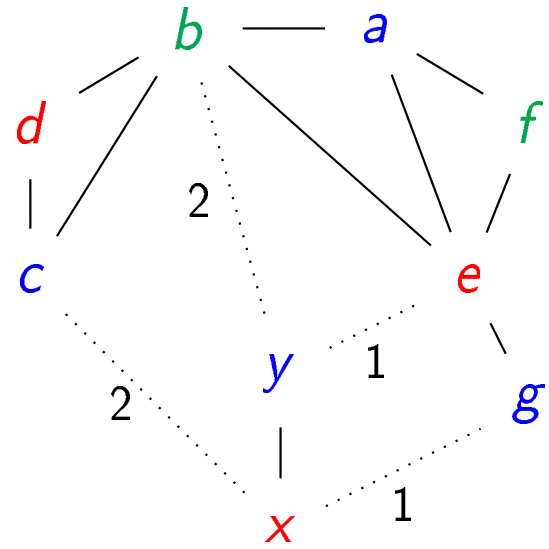
- Minimiere die Anzahl der dynamisch ausgeführten (statisch eingefügten) Kopierinstruktionen
- Üblich: Verschmelzung von Ecken im IG, um Kopien zu vermeiden
  - Führt zu nicht-chordalen Graphen
    - Man verliert die sichere Kenntnis der chromatischen Zahl
  - Werden aggressiv Ecken verschmolzen, können Spills entstehen, um Kopien zu vermeiden

Verschmelzung

Problemmodell

- Gegeben: Eine minimale Färbung des IG

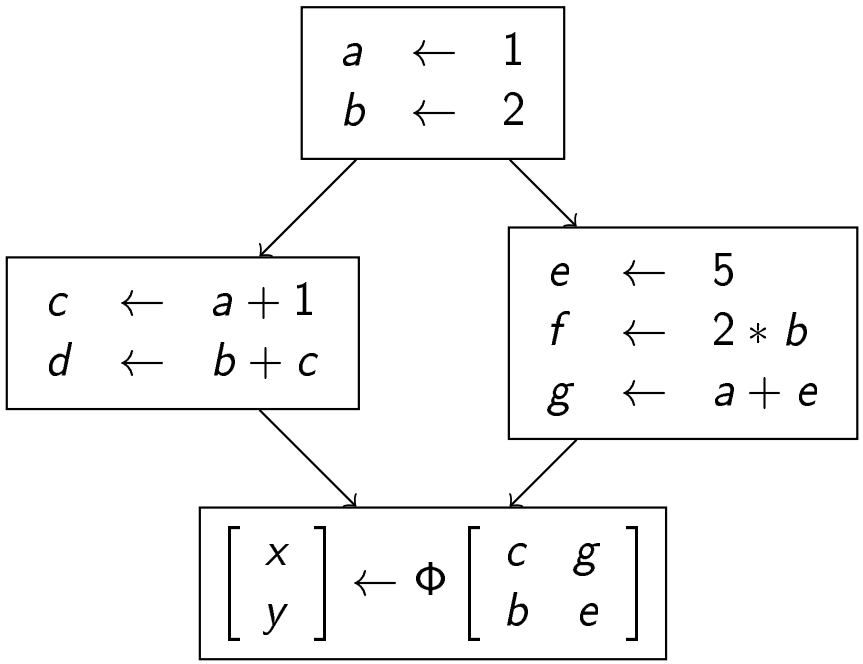


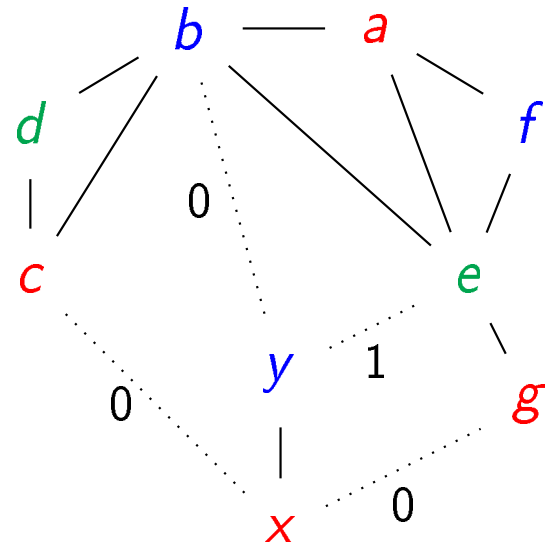


Verschmelzung

Problemmodell

- Gegeben: Eine minimale Färbung des IG
- Gesucht: Eine erlaubte ($< k$) Färbung mit minimalen Kosten
  - Kosten sind die gewichtete Summe der Gleichfärbekanten
  - Unbenutzte Farben dürfen verwendet werden
  - Weder IG noch Programm dürfen geändert werden





Verschmelzung

Formal

- Finde eine k-Färbung $\mathcal{C}$ des IG, die so vielen ϕ-Operanden und Ergebnissen wie möglich die gleiche Farbe zuweist.

$$\min_{\mathcal{C}} \sum_{\phi} costs(\mathcal{C}, \phi)$$

mit

$$costs\left(\mathcal{C}, y \leftarrow \phi(x_1, \ldots, x_n)\right) = \sum_{i=1}^{n} \begin{cases} 0 & \text{falls } \mathcal{C}(y) = \mathcal{C}(x_i) \\ w_{yx_i} & \text{sonst} \end{cases}$$

Verschmelzung

Lösungsstrategien

Komplexität

- Problem ist NP-vollständig in der Anzahl der ϕs

Algorithmen

- Eine Greedy-Heuristik
- Eine optimale Methode, die ILP[2] benutzt

[2] Integer Linear Programming

Verschmelzung

Heuristik

- Idee: Ändere die Farben um besser zusammenpassende Paare zu erhalten
- Problem: Ob der Farbwechsel möglich ist, ist nicht lokal entscheidbar
- Darum
  - Betrachte jedes ϕ separat
  - Versuche den ϕ-Operanden und dem Ergebnis dieselbe Farbe zu geben
  - Löse Farbunverträglichkeiten rekursiv durch den IG
  - Falls dies fehlschlägt markiere den Konflikt lokal und setze fort

Verschmelzung

Formalisierung als ILP

- Entscheidungsvariablen stellen Zustände / Färbungen dar
- Färbung: $x_{ic} = 1 \Leftrightarrow$ Ecke i hat Farbe c
- Zulässigkeit: $y_{ij} = 1 \Leftrightarrow$ Ecke i und j haben verschiedene Farben

$$
\begin{aligned}
\min f = \quad & \sum_{e \in Q} w_e \cdot y_{ij} & \\
\textit{mit} \quad & \sum_{c} x_{ic} = 1 & v_i \in V \\
& x_{ic} + x_{jc} \leq 1 & [v_i, v_j] \in E \\
& y_{ij} \geq x_{ic} - x_{jc} & [v_i, v_j] \in Q \\
& & \\
& y_{ij}, x_{ic} \in \{0, 1\} &
\end{aligned}
$$

Verschmelzung

ILP Beispiel mit 3 Ecken und 3 Farben

Verschmelzung

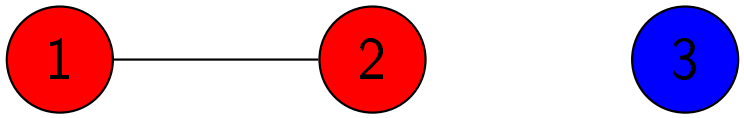
ILP Beispiel mit 3 Ecken und 3 Farben

$$
\begin{array}{lll}
\text{min} & ??? & \\
\textit{mit} & x_{11} + x_{12} + x_{13} & = 1 \\
 & x_{21} + x_{22} + x_{23} & = 1 \\
 & x_{31} + x_{32} + x_{33} & = 1
\end{array}
$$

Färbung

Verschmelzung

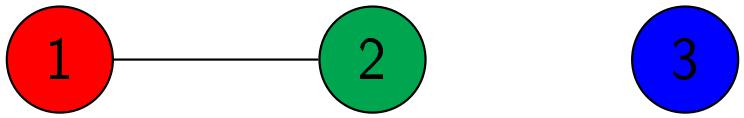
ILP Beispiel mit 3 Ecken und 3 Farben



$$
\begin{array}{lll}
\text{min} & ??? & \\
\textit{mit} & x_{11} + x_{12} + x_{13} = 1 & \\
 & x_{21} + x_{22} + x_{23} = 1 & \text{Färbung} \\
 & x_{31} + x_{32} + x_{33} = 1 & \\
 & & \\
\lightning & x_{11} + x_{21} \leq 1 & \\
 & x_{12} + x_{22} \leq 1 & \text{Interferenz} \\
 & x_{13} + x_{23} \leq 1 &
\end{array}
$$

Verschmelzung

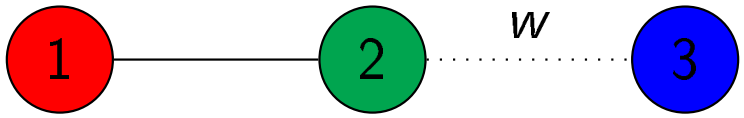
ILP Beispiel mit 3 Ecken und 3 Farben



$$
\begin{array}{lll}
\text{min} & ??? & \\
\textit{mit} & x_{11} + x_{12} + x_{13} = 1 & \\
 & x_{21} + x_{22} + x_{23} = 1 & \text{Färbung} \\
 & x_{31} + x_{32} + x_{33} = 1 & \\
 & & \\
 & x_{11} + x_{21} \leq 1 & \\
 & x_{12} + x_{22} \leq 1 & \text{Interferenz} \\
 & x_{13} + x_{23} \leq 1 &
\end{array}
$$

Verschmelzung

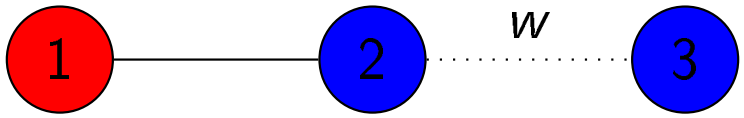
ILP Beispiel mit 3 Ecken und 3 Farben



$$
\begin{array}{lll}
\text{min} & w \cdot y_{23} & \\
\textit{mit} & x_{11} + x_{12} + x_{13} = 1 & \\
 & x_{21} + x_{22} + x_{23} = 1 & \text{Färbung} \\
 & x_{31} + x_{32} + x_{33} = 1 & \\
 & & \\
 & x_{11} + x_{21} \leq 1 & \\
 & x_{12} + x_{22} \leq 1 & \text{Interferenz} \\
 & x_{13} + x_{23} \leq 1 & \\
 & & \\
 & y_{23} \geq x_{21} - x_{31} & \\
 & y_{23} \geq x_{22} - x_{32} & \text{Affinität} \\
 & y_{23} \geq x_{23} - x_{33} &
\end{array}
$$

Verschmelzung

ILP Beispiel mit 3 Ecken und 3 Farben

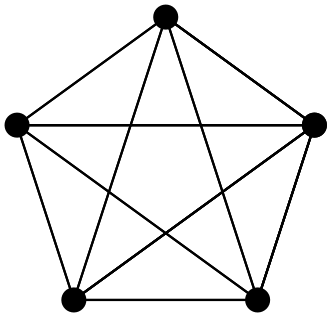


$$
\begin{array}{lll}
\text{min} & w \cdot y_{23} & \\
\textit{mit} & x_{11} + x_{12} + x_{13} = 1 & \\
 & x_{21} + x_{22} + x_{23} = 1 & \text{Färbung} \\
 & x_{31} + x_{32} + x_{33} = 1 & \\
 & & \\
 & x_{11} + x_{21} \leq 1 & \\
 & x_{12} + x_{22} \leq 1 & \text{Interferenz} \\
 & x_{13} + x_{23} \leq 1 & \\
 & & \\
 & y_{23} \geq x_{21} - x_{31} & \\
 & y_{23} \geq x_{22} - x_{32} & \text{Affinität} \\
 & y_{23} \geq x_{23} - x_{33} &
\end{array}
$$

Verschmelzung

Verbesserung der ILP Lösungszeit

- Cliquen Ungleichungen



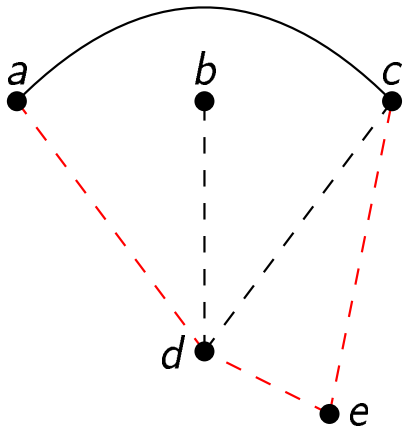
Ersetze $O(n^2)$ Ungleichungen $x_{ic} + x_{jc} \leq 1$

durch eine: $\sum_{i=1}^{n} x_{ic} \leq 1$

Verschmelzung

Verbesserung der ILP Lösungszeit

- Cliquen Ungleichungen
- Pfad Ungleichungen



Verwende gegenseitigen Ausschluss von Interferenz- und Gleichfärbekanten

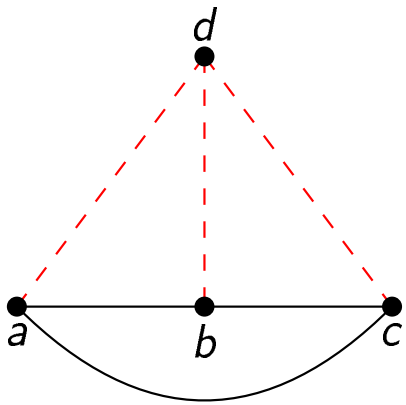
$y_{ad} + y_{cd} \geq 1$

$y_{ad} + y_{de} + y_{ec} \geq 1$

Verschmelzung

Verbesserung der ILP Lösungszeit

- Cliquen Ungleichungen
- Pfad Ungleichungen
- Cliquen-Pfad Ungleichungen



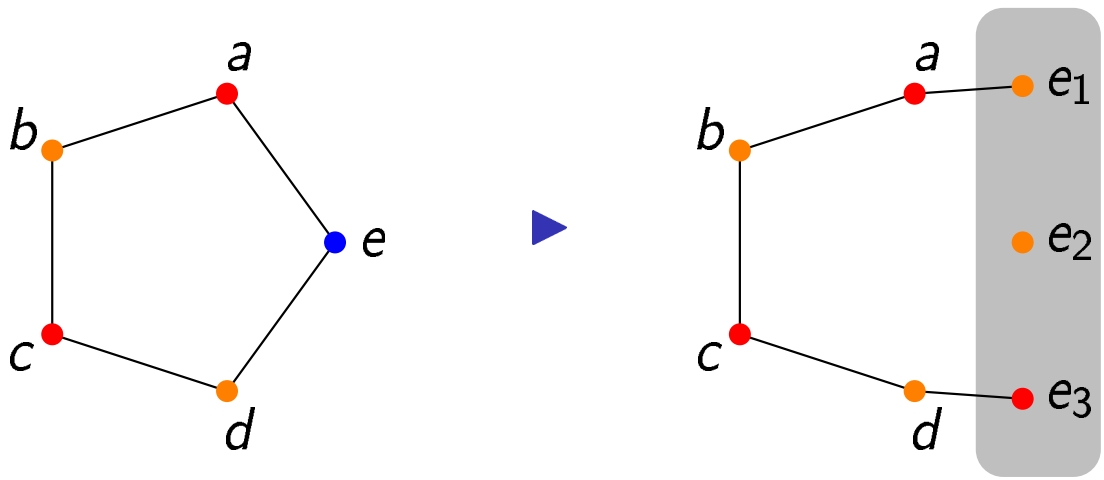
Mehrfache Verwendung desselben Arguments, z.B. $\begin{pmatrix} a \\ b \\ c \end{pmatrix} = \Phi \begin{pmatrix} d & e \\ d & f \\ d & g \end{pmatrix}$

führt zu $y_{ad} + y_{bd} + y_{cd} \geq 2$

Schlussfolgerung

SSA-Aufbau

- SSA-Aufbau erzeugt Kopien
  (ϕ-Funktionen sind Kopien auf Steuerflusskanten)
- Diese Kopien "zerreißen" den IG
- Ecken werden durch stabile Mengen (von Ecken) ersetzt

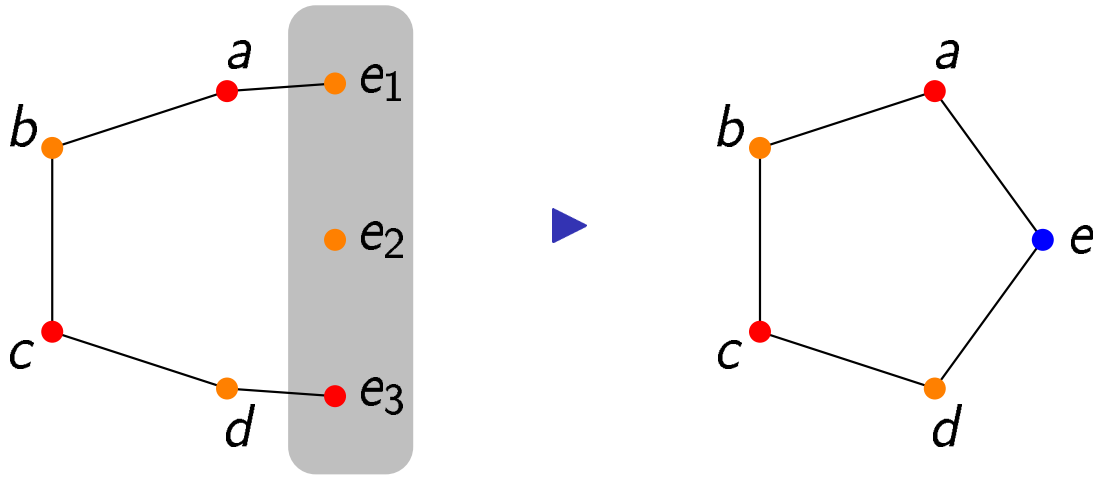


- Zyklen im IG werden aufgebrochen
- Der IG wird chordal

Schlussfolgerung

Klassischer SSA-Abbau

- SSA-Abbau verschmilzt aggressiv Kopien
  ohne die Anzahl der noch verfügbaren Register zu beachten
- Stabile Mengen werden zu Ecken



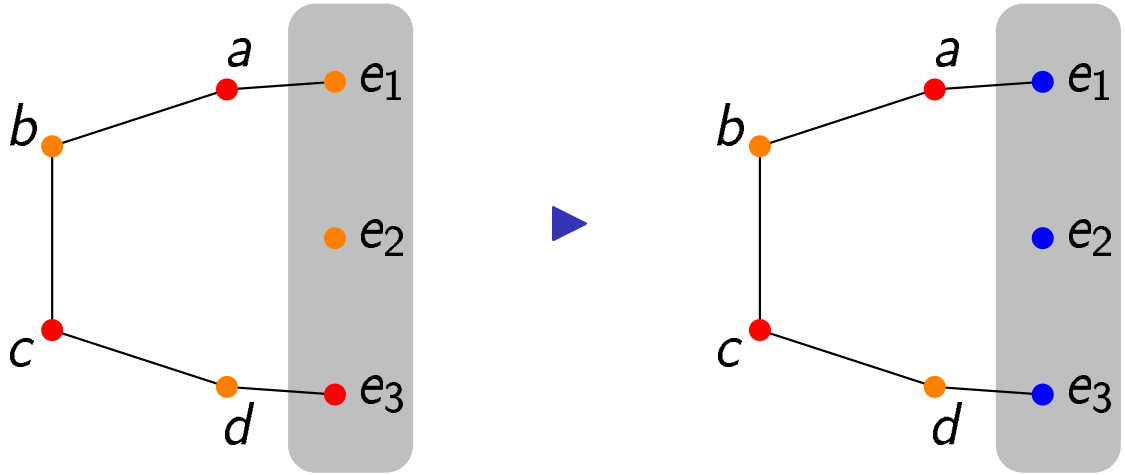
- Möglicherweise werden Zyklen erzeugt
- Dadurch kann sich die chromatische Zahl erhöhen

Schlussfolgerung

- Die vom SSA-Aufbau eingeführten Kopien sollten (möglichst) entfernt werden
- Sei $\chi(G) = 2$ und $k = 3$ Farben (Register) verfügbar
- Dann kann die freie Farbe für das Verschmelzen verwendet werden

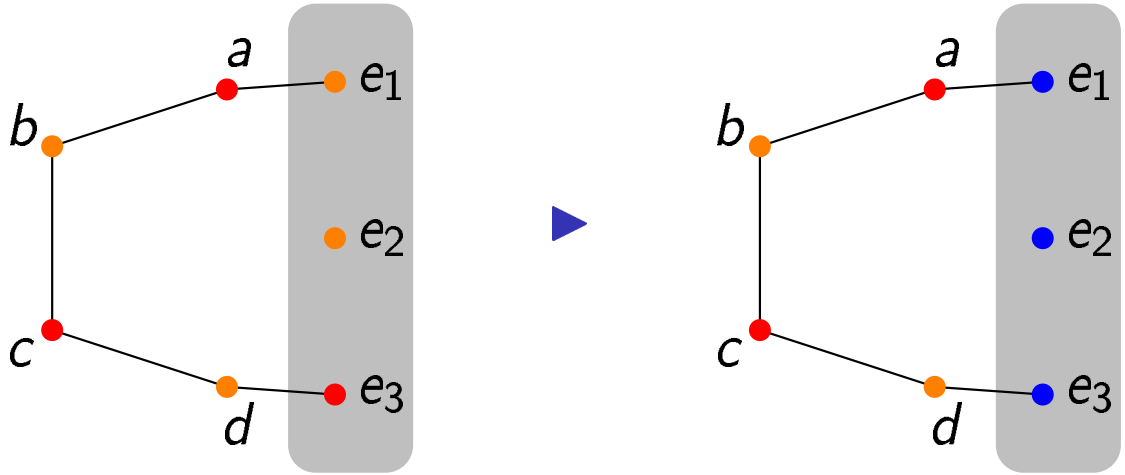
Schlussfolgerung

- Die vom SSA-Aufbau eingeführten Kopien sollten (möglichst) entfernt werden
- Sei $\chi(G) = 2$ und $k = 3$ Farben (Register) verfügbar
- Dann kann die freie Farbe für das Verschmelzen verwendet werden



Schlussfolgerung

- Die vom SSA-Aufbau eingeführten Kopien sollten (möglichst) entfernt werden
- Sei $\chi(G) = 2$ und $k = 3$ Farben (Register) verfügbar
- Dann kann die freie Farbe für das Verschmelzen verwendet werden



- Falls nur zwei Farben existieren: Kopie von e_2 nach e_3 stehen lassen
- Erkenntnis: Es ist besser, Verschmelzen nicht als das Zusammenlegen von Ecken zu verstehen

Registerzuteilung – Zusammenfassung

- Registerzuteilung ist NP-vollständig
- Hack/Goos liefert ein polynomielles Verfahren zur Graphfärbung, das die Registerzuteilung in 3 sequentielle Einzelschritte zerlegt
- Verschmelzung und Auslagerung bleibt aber NP-vollständig
- Selbst bei Verwendung heuristischer Verfahren kann die Registerzuteilung den Großteil der Übersetzungszeit brauchen
- Für große Graphen liefert *linear-scan* in vielen Anwendungsszenarien schneller eine brauchbare Lösung
- Ein weiterer Ansatz: Formulierung als ganzzahliges lineares Programm (ILP) (langsam)
- Das Zusammenspiel der zielmaschinenabhängigen Optimierungen ist offen

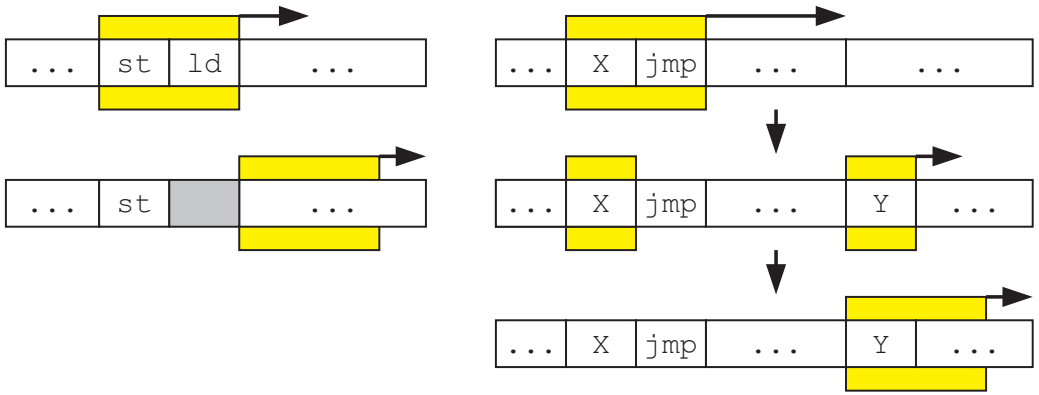
Inhalt

Nachoptimierung

- Grundidee (McKeeman, peephole optimization, 1965):
  schiebe ein Fenster, das n Befehle umfasst, über den erzeugten Code und ersetze die Befehlsfolge im Fenster durch eine kostengünstigere
- meist $n = 2$
- bei bedingten Sprüngen folgt das Fenster beiden Ausführungspfaden
- Grundlage der Vereinfachung: Maschinensimulation (wie bei einfacher Codeerzeugung)
- besonders wirksam bei CISC-Architekturen:
  - an Fugen zwischen dem Code unabhängiger Ausdrucksbäume
  - um komplizierte Adressierungsmodi zu nutzen:
    `a[i]; i++` $\rightarrow$ `a[i++]`
  - zur Komposition oder Zerlegung von Komplexbefehlen
- Hinweis: Eigentlich vor Befehlsanordnung (da Befehle geändert werden $\rightarrow$Pipeline), aber nicht ohne Befehls-Anordnung bzw. -Auswahl machbar.

Nachoptimierung nach Davidson/Fraser

- Methode:
  - betrachte Befehle als (endliche) Zustandstransduktoren, die den endlichen Prozessor- und Speicherzustand verändern, z.B. Befehle auf RTL-Ebene betrachten
  - berechne vorab die Wirkung von Paaren ($n = 2$) oder Tripeln ($n = 3$) solcher Befehle und überdecke sie durch andere, billigere Befehlsfolgen
  - konstruiere Tabelle ersetzbarer Befehlskombinationen
  - schiebe das Fenster über die Befehlsfolgen und ersetze entsprechend Tabelle unter Fortschaltung Maschinenzustand



Typische Beispiele

`store R,a; load a,R`	$\rightarrow$	`store R,a`	überflüssiges Laden
`imul 2,R; xxx`	$\rightarrow$	`ashl 2,R; xxx`	Konstantenmult.
`iadd x,R; cmp 0,R`	$\rightarrow$	`iadd x,R`	überflüssige Vergleiche
`if B then a:=x end`	$\rightarrow$	`t:=B; a:=(t) x`	Grundblock verlängern

Weitere Nachoptimierungen I

- Zusammenziehen von Sprüngen:
  - `goto M; ... M: goto L` → `goto L; ... M: goto L`
  - `if B then goto M; ... M: if B' then goto L` →
    `if B then goto L; ... M: if B' then goto L`
    falls `B'` aus `B` folgt
- Zusammenziehen von Grundblöcken, wenn der erste mit unbedingtem Sprung endigt und der zweite nur diesen Vorgänger hat
- Bedingungsumkehr und statische Sprungvorhersage:
  - `if B then goto M; L: ...` →
    `if ¬B then goto L; M: ...`
    falls dies zur schnelleren Ausführung des häufigsten Pfades führt
    häufigster Pfad: Schleifen werden wiederholt und plausible Annahmen, z.B. ganze Zahlen sind nicht negativ
  - `if B then goto M; goto L: M: ...` →
    `if ¬B then goto L; M: ...`

Weitere Nachoptimierungen II

- Enden verschmelzen
- leere Schleifen streichen: `while i<n loop i := i+1 end` $\rightarrow \emptyset$
  wenn anschließend `i` nicht mehr benötigt
  Vorsicht: diese Transformation ist partiell korrekt, könnte also nicht terminierende Programme zum Terminieren bringen!
- Endrekursion beseitigen (wenn nicht vorher geschehen):
  `p(x) is M: ... p(y); return; ... end` $\rightarrow$
  `p(x) is M: ... x := y; goto M; return; ... end`
- Ausdrücke, die Fehler verursachen durch Ausnahmeaufruf ersetzen:
  `print 1/0` $\rightarrow$ `raise ZERO_DIVIDE`
- Explizite Operationen auf Adressierungspfad setzen:
  `R := R + const; load (R),R'` $\rightarrow$ `load const(R),R'`
  wenn anschließend `<R>` nicht mehr benötigt

Nachoptimierungen – Zusammenfassung

- Automatenmodell
- viele Einzelaufgaben, wenig Systematik, oft stark prozessorabhängig
- eigentlich als Optimierungsmaßnahmen auf dem Zielcode entwickelt, z.T. aber auch vor der Codeerzeugung auf der Zwischensprache durchführbar
- Nachoptimierung beeinflusst Registervergabe, Befehlsanordnung, Konstantenfaltung, Lebendigkeitsanalyse
- Laufzeitgewinn von weit über 10% erzielbar
- manche Übersetzer (z.b. Urform lcc) benutzen nur Nachoptimierung

Zusammenfassung

- Registerzuteilung, Befehlsanordnung sind für die Funktion nötig, Nachoptimierung ist wegen Performance unerlässlich.
- Hier ist besonders zwischen Codequalität und Übersetzergeschwindigkeit abzuwägen. (Extremfälle: JIT-Übersetzer, eingebettete Systeme)
- Eine Kombination / Integration der maschinenabhängigen Optimierungen ist bis heute nur teilweise gelungen.

  Kombination von Befehlsanordnung und Registerzuteilung: Proebsting & Fischer, *Optimal Code Scheduling for DelayedLoad Architectures*, ACM Transactions on Programming Language Design and Implementation, Toronto, Canada, June 1991, pp. 256267.
- Unbehandeltes Problem: Berücksichtigung und Nutzung bedingter Befehle (predicated instructions) der IA-64

Übersetzerbau: Zusammenfassung

- Übersetzer für höhere Programmiersprachen sind
  - mittelgroße Programme (5-300k Zeilen)
  - mit stabiler Softwarearchitektur
  - mit hohen Qualitätsanforderungen
  - Einzelaufgaben mit z.T. sehr komplexen Algorithmen
    - oft theoretisch gut fundiert
    - automatisch aus Spezifikationen erzeugbar
  - viele NP Probleme in Optimierung und Codeerzeugung
  - hoher Forschungsbedarf in Optimierung und Codeerzeugung
  - Interaktion mit Prozessorentwurf!
- oft irreführende Gerüchte, was geht und was nicht geht