



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Optimierende Compiler

6. Einführung in Code-Optimierung

Andreas Koch

FG Eingebettete Systeme und ihre Anwendungen
Informatik, TU Darmstadt

Sommersemester 2007



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Organisatorisches



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

- **Ab jetzt vorgehen nach Cooper & Torczon**
 - Daraus ab jetzt auch die meisten Zeichnungen
- Geplant: Behandle Kapitel 8-10
- Unterstützt durch Papers
 - Werden als PDF auf Web-Seite gelegt



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

- Ab jetzt vorgehen nach Cooper & Torczon
 - Daraus ab jetzt auch die meisten Zeichnungen
- Geplant: Behandle Kapitel 8-10
- Unterstützt durch Papers
 - Werden als PDF auf Web-Seite gelegt



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

- Ab jetzt vorgehen nach Cooper & Torczon
 - Daraus ab jetzt auch die meisten Zeichnungen
- Geplant: Behandle Kapitel 8-10
- Unterstützt durch Papers
 - Werden als PDF auf Web-Seite gelegt



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

- Ab jetzt vorgehen nach Cooper & Torczon
 - Daraus ab jetzt auch die meisten Zeichnungen
- Geplant: Behandle Kapitel 8-10
- Unterstützt durch Papers
 - Werden als PDF auf Web-Seite gelegt



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

- Ab jetzt vorgehen nach Cooper & Torczon
 - Daraus ab jetzt auch die meisten Zeichnungen
- Geplant: Behandle Kapitel 8-10
- Unterstützt durch Papers
 - Werden als PDF auf Web-Seite gelegt



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

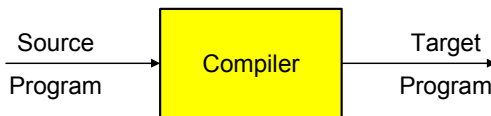
Superlokal

Regional

Global

Einleitung

Auswirkung von Optimierung



Optimierer versucht:

- Overhead von hoher Abstraktionsebene der Eingabesprache zu reduzieren
- Eingabeprogramm effizient auf Zielmaschine abzubilden
 - Vertusche Schwächen der Hardware-Architektur
 - Stärken der Hardware-Architektur auszunutzen
- Sonderfälle auszunutzen

➔ Ziel: Effizienz eines erfahrenen Assembler-Programmierers

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

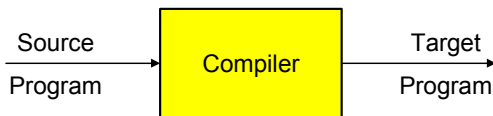
Wertnumerierung

Lokal

Superlokal

Regional

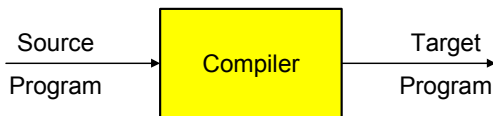
Global



Optimierer versucht:

- Overhead von hoher Abstraktionsebene der Eingabesprache zu reduzieren
- Eingabeprogramm effizient auf Zielmaschine abzubilden
 - Vertusche Schwächen der Hardware-Architektur
 - Stärken der Hardware-Architektur auszunutzen
- Sonderfälle auszunutzen

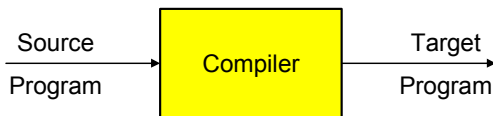
➔ Ziel: Effizienz eines erfahrenen Assembler-Programmierers



Optimierer versucht:

- Overhead von hoher Abstraktionsebene der Eingabesprache zu reduzieren
- Eingabeprogramm effizient auf Zielmaschine abzubilden
 - Vertusche Schwächen der Hardware-Architektur
 - Stärken der Hardware-Architektur auszunutzen
 - Sonderfälle auszunutzen

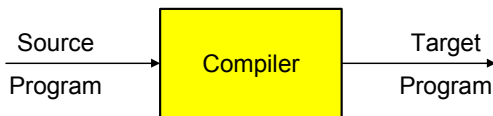
➔ Ziel: Effizienz eines erfahrenen Assembler-Programmierers



Optimierer versucht:

- Overhead von hoher Abstraktionsebene der Eingabesprache zu reduzieren
- Eingabeprogramm effizient auf Zielmaschine abzubilden
 - Vertusche Schwächen der Hardware-Architektur
 - Stärken der Hardware-Architektur auszunutzen
- Sonderfälle auszunutzen

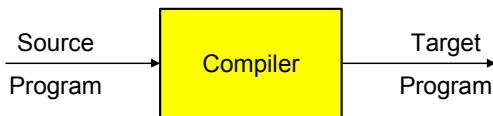
➔ Ziel: Effizienz eines erfahrenen Assembler-Programmierers



Optimierer versucht:

- Overhead von hoher Abstraktionsebene der Eingabesprache zu reduzieren
- Eingabeprogramm effizient auf Zielmaschine abzubilden
 - Vertusche Schwächen der Hardware-Architektur
 - Stärken der Hardware-Architektur auszunutzen
- Sonderfälle auszunutzen

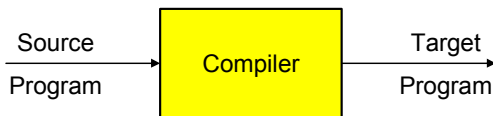
➔ Ziel: Effizienz eines erfahrenen Assembler-Programmierers



Optimierer versucht:

- Overhead von hoher Abstraktionsebene der Eingabesprache zu reduzieren
- Eingabeprogramm effizient auf Zielmaschine abzubilden
 - Vertusche Schwächen der Hardware-Architektur
 - Stärken der Hardware-Architektur auszunutzen
- Sonderfälle auszunutzen

➔ Ziel: Effizienz eines erfahrenen Assembler-Programmierers



Optimierer versucht:

- Overhead von hoher Abstraktionsebene der Eingabesprache zu reduzieren
- Eingabeprogramm effizient auf Zielmaschine abzubilden
 - Vertusche Schwächen der Hardware-Architektur
 - Stärken der Hardware-Architektur auszunutzen
- Sonderfälle auszunutzen

➔ Ziel: Effizienz eines erfahrenen Assembler-Programmierers



Viele Möglichkeiten, entsprechend Vorgabe des Benutzers

- **Schnellster Code**
- Kleinster Code
- Geringste Anzahl an Speicherzugriffen
- Geringste Anzahl von ausgelagerten Speicherseiten
- ...

➔ Optimierung formt Code entsprechend um



Viele Möglichkeiten, entsprechend Vorgabe des Benutzers

- Schnellster Code
- Kleinster Code
- Geringste Anzahl an Speicherzugriffen
- Geringste Anzahl von ausgelagerten Speicherseiten
- ...

➔ Optimierung formt Code entsprechend um



Viele Möglichkeiten, entsprechend Vorgabe des Benutzers

- Schnellster Code
- Kleinster Code
- Geringste Anzahl an Speicherzugriffen
- Geringste Anzahl von ausgelagerten Speicherseiten
- ...

➔ Optimierung formt Code entsprechend um



Viele Möglichkeiten, entsprechend Vorgabe des Benutzers

- Schnellster Code
- Kleinster Code
- Geringste Anzahl an Speicherzugriffen
- Geringste Anzahl von ausgelagerten Speicherseiten
- ...

➔ Optimierung formt Code entsprechend um



Viele Möglichkeiten, entsprechend Vorgabe des Benutzers

- Schnellster Code
- Kleinster Code
- Geringste Anzahl an Speicherzugriffen
- Geringste Anzahl von ausgelagerten Speicherseiten
- ...

➔ Optimierung formt Code entsprechend um



Viele Möglichkeiten, entsprechend Vorgabe des Benutzers

- Schnellster Code
- Kleinster Code
- Geringste Anzahl an Speicherzugriffen
- Geringste Anzahl von ausgelagerten Speicherseiten
- ...

➔ Optimierung formt Code entsprechend um



Viele Möglichkeiten, entsprechend Vorgabe des Benutzers

- Schnellster Code
- Kleinster Code
- Geringste Anzahl an Speicherzugriffen
- Geringste Anzahl von ausgelagerten Speicherseiten
- ...

➔ Optimierung formt Code entsprechend um



Sicherheit Die Bedeutung des Programmes (hier: extern beobachtetes Verhalten) darf **nicht** verändert werden.

Profitabilität Die Optimierung muß in Bezug auf das angestrebte Ziel eine **ausreichende** Verbesserung bringen.

Risiko Welche **negativen** Effekte kann die Optimierung nach sich ziehen?

Anwendbarkeit Gibt es **ausreichend** viele Stellen, an denen die Optimierung angebracht werden kann?



Sicherheit Die Bedeutung des Programmes (hier: extern beobachtetes Verhalten) darf **nicht** verändert werden.

Profitabilität Die Optimierung muß in Bezug auf das angestrebte Ziel eine **ausreichende** Verbesserung bringen.

Risiko Welche **negativen** Effekte kann die Optimierung nach sich ziehen?

Anwendbarkeit Gibt es **ausreichend** viele Stellen, an denen die Optimierung angebracht werden kann?



Sicherheit Die Bedeutung des Programmes (hier: extern beobachtetes Verhalten) darf **nicht** verändert werden.

Profitabilität Die Optimierung muß in Bezug auf das angestrebte Ziel eine **ausreichende** Verbesserung bringen.

Risiko Welche **negativen** Effekte kann die Optimierung nach sich ziehen?

Anwendbarkeit Gibt es **ausreichend** viele Stellen, an denen die Optimierung angebracht werden kann?



Sicherheit Die Bedeutung des Programmes (hier: extern beobachtetes Verhalten) darf **nicht** verändert werden.

Profitabilität Die Optimierung muß in Bezug auf das angestrebte Ziel eine **ausreichende** Verbesserung bringen.

Risiko Welche **negativen** Effekte kann die Optimierung nach sich ziehen?

Anwendbarkeit Gibt es **ausreichend** viele Stellen, an denen die Optimierung angebracht werden kann?



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Redundante Ausdrücke

Redundante Ausdrücke



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Ursprünglicher Code

```
m := 2 * y * z;  
n := 3 * y * z;  
o := 2 * y - z;
```

Umgeschriebener Code

```
t0 := 2 * y;  
m := t0 * z;  
n := 3 * y * z;  
o := t0 - z;
```

Redundanter Ausdruck

Ein Ausdruck $x \text{ op } y$ ist an einer Position L redundant, wenn er in jedem Fall vor L berechnet wurde, und zwischen der Berechnung und L die Operanden x und y nicht verändert wurden.

Redundante Ausdrücke



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Ursprünglicher Code

```
m := 2 * y * z;  
n := 3 * y * z;  
o := 2 * y - z;
```

Umgeschriebener Code

```
t0 := 2 * y;  
m := t0 * z;  
n := 3 * y * z;  
o := t0 - z;
```

Redundanter Ausdruck

Ein Ausdruck $x \text{ op } y$ ist an einer Position L redundant, wenn er in jedem Fall vor L berechnet wurde, und zwischen der Berechnung und L die Operanden x und y nicht verändert wurden.



Ursprünglicher Code

```
m := 2 * y * z;  
n := 3 * y * z;  
o := 2 * y - z;
```

Umgeschriebener Code

```
t0 := 2 * y;  
m := t0 * z;  
n := 3 * y * z;  
o := t0 - z;
```

Redundanter Ausdruck

Ein Ausdruck $x \text{ op } y$ ist an einer Position L redundant, wenn er in jedem Fall vor L berechnet wurde, und zwischen der Berechnung und L die Operanden x und y nicht verändert wurden.

Beschränkung auf Basisblock



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Basisblock (BB)

Längste Folge von Anweisungen ohne Kontrollfluß.

Beispiel:

```
a := b + 42;
if (a > 23) then
  c := a - 46;
  d := b * 15;
else
  c := a + 46;
  d := 0;
  q := false;
endif
```

Basisblöcke:

```
a := b + 42;
```

```
c := a - 46;
d := b * 15;
```

```
c := a + 46;
d := 0;
q := false;
```

Beschränkung auf Basisblock



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Basisblock (BB)

Längste Folge von Anweisungen ohne Kontrollfluß.

Beispiel:

```
a := b + 42;
if (a > 23) then
  c := a - 46;
  d := b * 15;
else
  c := a + 46;
  d := 0;
  q := false;
endif
```

Basisblöcke:

```
a := b + 42;
```

```
c := a - 46;
d := b * 15;
```

```
c := a + 46;
d := 0;
q := false;
```


Beschränkung auf Basisblock



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Basisblock (BB)

Längste Folge von Anweisungen ohne Kontrollfluß.

Beispiel:

```
a := b + 42;  
if (a > 23) then  
  c := a - 46;  
  d := b * 15;  
else  
  c := a + 46;  
  d := 0  
  q := false;  
endif
```

Basisblöcke:

```
a := b + 42;
```

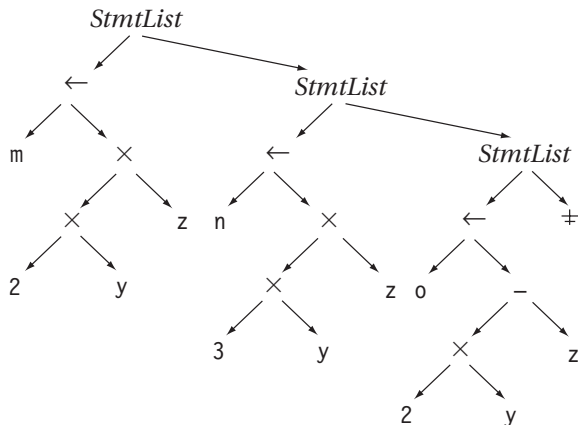
```
c := a - 46;  
d := b * 15;
```

```
c := a + 46;  
d := 0  
q := false;
```

Erster Ansatz: AST-Ebene



Erkennen gleicher Teilbäume, dann Erweiterung auf DAG



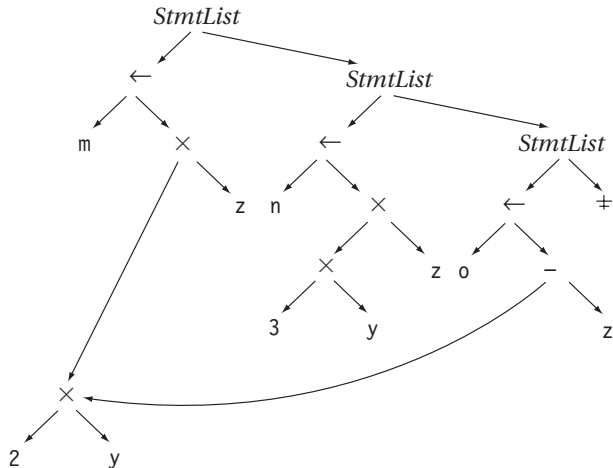
➔ Redundanzelimination in ASG-Konstruktor einbauen



Erster Ansatz: AST-Ebene



Erkennen gleicher Teilbäume, dann Erweiterung auf DAG



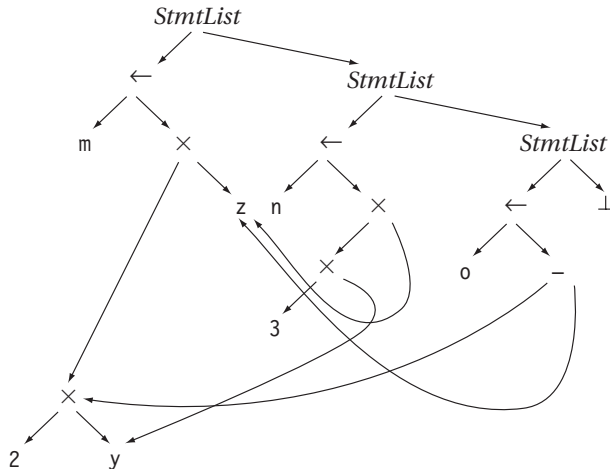
Redundanzelimination in ASG-Konstruktor einbauen



Erster Ansatz: AST-Ebene



Erkennen gleicher Teilbäume, dann Erweiterung auf DAG



➔ Redundanzelimination in ASG-Konstruktor einbauen



- OptComp
- A. Koch
- Orga
- Intro
- Redundanz
- CFGs
- Wertnumerierung
- Lokal
- Superlokal
- Regional
- Global



- Hashing über ganze Unterbäume
- Gleicher Hash \rightarrow vergleiche Struktur genau
- Wenn gleich: Setze bei redundantem Auftreten Zeiger auf Definition um

Problem:

```
m := 2 * y * z;  
y := 3 * y * z;  
o := 2 * y - z;
```

Einfaches Hashing bei $o := \dots$

- findet Definition von $2 * y$ (bei $m := \dots$)
- sieht aber nicht Zuweisung bei $y := \dots$

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Erster Ansatz: Realisierungsidee



- Hashing über ganze Unterbäume
- Gleicher Hash \rightarrow vergleiche Struktur genau
- Wenn gleich: Setze bei redundantem Auftreten Zeiger auf Definition um

Problem:

```
m := 2 * y * z;  
y := 3 * y * z;  
o := 2 * y - z;
```

Einfaches Hashing bei $o := \dots$

- findet Definition von $2 * y$ (bei $m := \dots$)
- sieht aber nicht Zuweisung bei $y := \dots$

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Erster Ansatz: Realisierungsidee



- Hashing über ganze Unterbäume
- Gleicher Hash \rightarrow vergleiche Struktur genau
- Wenn gleich: Setze bei redundantem Auftreten Zeiger auf Definition um

Problem:

```
m := 2 * y * z;  
y := 3 * y * z;  
o := 2 * y - z;
```

Einfaches Hashing bei $o := \dots$

- findet Definition von $2 * y$ (bei $m := \dots$)
- sieht aber nicht Zuweisung bei $y := \dots$

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Erster Ansatz: Realisierungsidee



- Hashing über ganze Unterbäume
- Gleicher Hash \rightarrow vergleiche Struktur genau
- Wenn gleich: Setze bei redundantem Auftreten Zeiger auf Definition um

Problem:

```
m := 2 * y * z;  
y := 3 * y * z;  
o := 2 * y - z;
```

Einfaches Hashing bei $o := \dots$

- findet Definition von $2 * y$ (bei $m := \dots$)
- sieht aber nicht Zuweisung bei $y := \dots$

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Erster Ansatz: Realisierungsidee



- Hashing über ganze Unterbäume
- Gleicher Hash \rightarrow vergleiche Struktur genau
- Wenn gleich: Setze bei redundantem Auftreten Zeiger auf Definition um

Problem:

```
m := 2 * y * z;  
y := 3 * y * z;  
o := 2 * y - z;
```

Einfaches Hashing bei $o := \dots$

- findet Definition von $2 * y$ (bei $m := \dots$)
- sieht aber nicht Zuweisung bei $y := \dots$

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global



- Hashing über ganze Unterbäume
- Gleicher Hash \rightarrow vergleiche Struktur genau
- Wenn gleich: Setze bei redundantem Auftreten Zeiger auf Definition um

Problem:

```
m := 2 * y * z;  
y := 3 * y * z;  
o := 2 * y - z;
```

Einfaches Hashing bei $o := \dots$

- findet Definition von $2*y$ (bei $m := \dots$)
- sieht aber nicht Zuweisung bei $y := \dots$



- Hashing über ganze Unterbäume
- Gleicher Hash \rightarrow vergleiche Struktur genau
- Wenn gleich: Setze bei redundantem Auftreten Zeiger auf Definition um

Problem:

```
m := 2 * y * z;  
y := 3 * y * z;  
o := 2 * y - z;
```

Einfaches Hashing bei $o := \dots$

- findet Definition von $2*y$ (bei $m := \dots$)
- sieht aber nicht Zuweisung bei $y := \dots$

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global



Braucht Mechanismus, um Zuweisungen zu beachten!
Idee

- Verwalte unterschiedliche Versionen von Variablen
- Erhöhe Versionsnummer bei jeder Zuweisung
- Hash nun über Variablennamen und Versionsnummer

```
m0 := 2 * y0 * z0;  
y1 := 3 * y0 * z0;  
o0 := 2 * y1 - z0;
```

- Trägt bei `m0` nun als bekannt den Ausdruck `2*y0` ein
- Unterbindet Wiederverwendung als `2*y1`



Braucht Mechanismus, um Zuweisungen zu beachten!
Idee

- Verwalte unterschiedliche Versionen von Variablen
- Erhöhe Versionsnummer bei jeder Zuweisung
- Hash nun über Variablennamen und Versionsnummer

```
m0 := 2 * y0 * z0;  
y1 := 3 * y0 * z0;  
o0 := 2 * y1 - z0;
```

- Trägt bei `m0` nun als bekannt den Ausdruck `2*y0` ein
- Unterbindet Wiederverwendung als `2*y1`



Braucht Mechanismus, um Zuweisungen zu beachten!
Idee

- Verwalte unterschiedliche Versionen von Variablen
- Erhöhe Versionsnummer bei jeder Zuweisung
- Hash nun über Variablennamen und Versionsnummer

```
m0 := 2 * y0 * z0;  
y1 := 3 * y0 * z0;  
o0 := 2 * y1 - z0;
```

- Trägt bei m_0 nun als bekannt den Ausdruck $2*y_0$ ein
- Unterbindet Wiederverwendung als $2*y_1$



Braucht Mechanismus, um Zuweisungen zu beachten!
Idee

- Verwalte unterschiedliche Versionen von Variablen
- Erhöhe Versionsnummer bei jeder Zuweisung
- Hash nun über Variablennamen und Versionsnummer

```
m0 := 2 * y0 * z0;  
y1 := 3 * y0 * z0;  
o0 := 2 * y1 - z0;
```

- Trägt bei m_0 nun als bekannt den Ausdruck $2*y_0$ ein
- Unterbindet Wiederverwendung als $2*y_1$



Braucht Mechanismus, um Zuweisungen zu beachten!
Idee

- Verwalte unterschiedliche Versionen von Variablen
- Erhöhe Versionsnummer bei jeder Zuweisung
- Hash nun über Variablennamen und Versionsnummer

```
m0 := 2 * y0 * z0;  
y1 := 3 * y0 * z0;  
o0 := 2 * y1 - z0;
```

- Trägt bei `m0` nun als bekannt den Ausdruck `2*y0` ein
- Unterbindet Wiederverwendung als `2*y1`



Braucht Mechanismus, um Zuweisungen zu beachten!
Idee

- Verwalte unterschiedliche Versionen von Variablen
- Erhöhe Versionsnummer bei jeder Zuweisung
- Hash nun über Variablennamen und Versionsnummer

```
m0 := 2 * y0 * z0;  
y1 := 3 * y0 * z0;  
o0 := 2 * y1 - z0;
```

- Trägt bei `m0` nun als bekannt den Ausdruck `2*y0` ein
- Unterbindet Wiederverwendung als `2*y1`



Durchnumerieren von Variablenversionen

- **Sehr** nützlicher Trick!

Wann wird potentiell neue Variablenversion erzeugt?

- Bei jeder Zuweisung (einfacher Fall!)
 - `int i; i = 1;` → `i` steht bei Prozessor (globale, nicht lokale Variable)
 - `int i; i = 1; i = 2;` → `i` (zwei Parameter)
- Was bei Zeigeroperationen?
 - `int i; i = 1; int *p; p = &i;` → `i` (zwei Parameter)
 - Jeder Zeiger kann eine Variable zeigen
- Einzige immer sichere Methode
 - `int i; i = 1; int *p; p = &i; *p = 2;` → `i` (zwei Parameter)
 - Bei Schreiben via Zeiger Variable einer Variable
- Effizienter: **Beweisen**, dass Zeiger nur auf bestimmte Variablen zeigen kann

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global



Durchnumerieren von Variablenversionen

- **Sehr** nützlicher Trick!

Wann wird potentiell neue Variablenversion erzeugt?

- Bei jeder Zuweisung (einfacher Fall!)
 - `int i; i = 1;` → `i` steht bei Prozessor (globale, nicht lokale Variable)
 - `int i; i = 1; i = 2;` → `i` (zwei Parameter)
- Was bei Zeigeroperationen?
 - `int i; i = 1; int *p; p = &i;` → `i` (zwei Parameter)
 - Jeder Zeiger kann eine Variable zeigen
- Einzige immer sichere Methode
 - `int i; i = 1; int *p; p = &i; *p = 2;` → `i` (zwei Parameter)
 - Bei Schreiben via Zeiger Variable einer Variable
- Effizienter: **Beweisen**, dass Zeiger nur auf bestimmte Variablen zeigen kann

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global



Durchnumerieren von Variablenversionen

- **Sehr** nützlicher Trick!

Wann wird potentiell neue Variablenversion erzeugt?

- Bei jeder Zuweisung (einfacher Fall!)
 - Vorsicht bei Prozeduren (globale, nicht-lokale Variablen, `var`-Parameter)
- Was bei Zeigeroperationen?
 - Woanders im Programm: `p = &y; ...; *p = 42;`
 - Jeder Zeiger **könnte** eine Variable verändern
- Einzige immer sichere Methode
 - Bei Schreiben via Zeiger Versionen **aller** Variablen erhöhen
- Effizienter: **Beweisen**, dass Zeiger nur auf bestimmte Variablen zeigen kann
 - Nicht trivial (*pointer disambiguation*).



Durchnumerieren von Variablenversionen

- **Sehr** nützlicher Trick!

Wann wird potentiell neue Variablenversion erzeugt?

- Bei jeder Zuweisung (einfacher Fall!)
 - Vorsicht bei Prozeduren (globale, nicht-lokale Variablen, **var**-Parameter)
- Was bei Zeigeroperationen?
 - Woanders im Programm: `p = &y; ...; *p = 42;`
 - Jeder Zeiger **könnte** eine Variable verändern
- Einzige immer sichere Methode
 - Bei Schreiben via Zeiger Versionen **aller** Variablen erhöhen
- Effizienter: **Beweisen**, dass Zeiger nur auf bestimmte Variablen zeigen kann
 - Nicht trivial (*pointer disambiguation*).



Durchnumerieren von Variablenversionen

- **Sehr** nützlicher Trick!

Wann wird potentiell neue Variablenversion erzeugt?

- Bei jeder Zuweisung (einfacher Fall!)
 - Vorsicht bei Prozeduren (globale, nicht-lokale Variablen, **var**-Parameter)
- Was bei Zeigeroperationen?
 - Woanders im Programm: `p = &y; ...; *p = 42;`
 - Jeder Zeiger **könnte** eine Variable verändern
- Einzige immer sichere Methode
 - Bei Schreiben via Zeiger Versionen **aller** Variablen erhöhen
- Effizienter: **Beweisen**, dass Zeiger nur auf bestimmte Variablen zeigen kann
 - Nicht trivial (*pointer disambiguation*).



Durchnumerieren von Variablenversionen

- **Sehr** nützlicher Trick!

Wann wird potentiell neue Variablenversion erzeugt?

- Bei jeder Zuweisung (einfacher Fall!)
 - Vorsicht bei Prozeduren (globale, nicht-lokale Variablen, **var**-Parameter)
- Was bei Zeigeroperationen?
 - Woanders im Programm: **p = &y; ...; *p = 42;**
 - Jeder Zeiger **könnte** eine Variable verändern
- Einzige immer sichere Methode
 - Bei Schreiben via Zeiger Versionen **aller** Variablen erhöhen
- Effizienter: **Beweisen**, dass Zeiger nur auf bestimmte Variablen zeigen kann
 - Nicht trivial (*pointer disambiguation*).



Durchnumerieren von Variablenversionen

- **Sehr** nützlicher Trick!

Wann wird potentiell neue Variablenversion erzeugt?

- Bei jeder Zuweisung (einfacher Fall!)
 - Vorsicht bei Prozeduren (globale, nicht-lokale Variablen, **var**-Parameter)
- Was bei Zeigeroperationen?
 - Woanders im Programm: **p = &y; ...; *p = 42;**
 - Jeder Zeiger **könnte** eine Variable verändern
- Einzige immer sichere Methode
 - Bei Schreiben via Zeiger Versionen **aller** Variablen erhöhen
- Effizienter: **Beweisen**, dass Zeiger nur auf bestimmte Variablen zeigen kann
 - Nicht trivial (*pointer disambiguation*).



Durchnumerieren von Variablenversionen

- **Sehr** nützlicher Trick!

Wann wird potentiell neue Variablenversion erzeugt?

- Bei jeder Zuweisung (einfacher Fall!)
 - Vorsicht bei Prozeduren (globale, nicht-lokale Variablen, **var**-Parameter)
- Was bei Zeigeroperationen?
 - Woanders im Programm: **p = &y; ...; *p = 42;**
 - Jeder Zeiger **könnte** eine Variable verändern
- Einzige immer sichere Methode
 - Bei Schreiben via Zeiger Versionen **aller** Variablen erhöhen
- Effizienter: **Beweisen**, dass Zeiger nur auf bestimmte Variablen zeigen kann
 - Nicht trivial (*pointer disambiguation*).



Durchnumerieren von Variablenversionen

- **Sehr** nützlicher Trick!

Wann wird potentiell neue Variablenversion erzeugt?

- Bei jeder Zuweisung (einfacher Fall!)
 - Vorsicht bei Prozeduren (globale, nicht-lokale Variablen, **var**-Parameter)
- Was bei Zeigeroperationen?
 - Woanders im Programm: `p = &y; ...; *p = 42;`
 - Jeder Zeiger **könnte** eine Variable verändern
- Einzige immer sichere Methode
 - Bei Schreiben via Zeiger Versionen **aller** Variablen erhöhen
- Effizienter: **Beweisen**, dass Zeiger nur auf bestimmte Variablen zeigen kann

- Nicht trivial (*pointer disambiguation*).



Durchnumerieren von Variablenversionen

- **Sehr** nützlicher Trick!

Wann wird potentiell neue Variablenversion erzeugt?

- Bei jeder Zuweisung (einfacher Fall!)
 - Vorsicht bei Prozeduren (globale, nicht-lokale Variablen, **var**-Parameter)
- Was bei Zeigeroperationen?
 - Woanders im Programm: **p = &y; ...; *p = 42;**
 - Jeder Zeiger **könnte** eine Variable verändern
- Einzige immer sichere Methode
 - Bei Schreiben via Zeiger Versionen **aller** Variablen erhöhen
- Effizienter: **Beweisen**, dass Zeiger nur auf bestimmte Variablen zeigen kann
 - Nicht trivial (*pointer disambiguation*).



Durchnumerieren von Variablenversionen

- **Sehr** nützlicher Trick!

Wann wird potentiell neue Variablenversion erzeugt?

- Bei jeder Zuweisung (einfacher Fall!)
 - Vorsicht bei Prozeduren (globale, nicht-lokale Variablen, **var**-Parameter)
- Was bei Zeigeroperationen?
 - Woanders im Programm: **p = &y; ...; *p = 42;**
 - Jeder Zeiger **könnte** eine Variable verändern
- Einzige immer sichere Methode
 - Bei Schreiben via Zeiger Versionen **aller** Variablen erhöhen
- Effizienter: **Beweisen**, dass Zeiger nur auf bestimmte Variablen zeigen kann
 - Nicht trivial (*pointer disambiguation*).



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Kontrollflußgraphen

Neue Zwischendarstellung: Kontrollflußgraphen



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

- **Kontrollfluß bisher vernachlässigt!**
- Arbeiten auf Basisblöcken
- **Nicht ausreichend** als allgemeine Zwischendarstellung
- Erweiterung auf **Graph** von Basisblöcken
 - Am Ende jedes Basisblockes (bedingter) Sprung zum nächsten Block
 - Kanten symbolisieren **Kontrollfluß**
- Sehr gut für viele Optimierungen brauchbar
- Häufig verwendete Zwischendarstellung im Optimierer

Neue Zwischendarstellung: Kontrollflußgraphen



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

- Kontrollfluß bisher vernachlässigt!
- Arbeiten auf Basisblöcken
- **Nicht ausreichend** als allgemeine Zwischendarstellung
- Erweiterung auf **Graph** von Basisblöcken
 - Am Ende jedes Basisblockes (bedingter) Sprung zum nächsten Block
 - Kanten symbolisieren **Kontrollfluß**
- Sehr gut für viele Optimierungen brauchbar
- Häufig verwendete Zwischendarstellung im Optimierer

Neue Zwischendarstellung: Kontrollflußgraphen



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

- Kontrollfluß bisher vernachlässigt!
- Arbeiten auf Basisblöcken
- **Nicht ausreichend** als allgemeine Zwischendarstellung
- Erweiterung auf **Graph** von Basisblöcken
 - Am Ende jedes Basisblockes (bedingter) Sprung zum nächsten Block
 - Kanten symbolisieren **Kontrollfluß**
- Sehr gut für viele Optimierungen brauchbar
- Häufig verwendete Zwischendarstellung im Optimierer



- Kontrollfluß bisher vernachlässigt!
- Arbeiten auf Basisblöcken
- **Nicht ausreichend** als allgemeine Zwischendarstellung
- Erweiterung auf **Graph** von Basisblöcken
 - Am Ende jedes Basisblockes (bedingter) Sprung zum nächsten Block
 - Kanten symbolisieren **Kontrollfluß**
- Sehr gut für viele Optimierungen brauchbar
- Häufig verwendete Zwischendarstellung im Optimierer



- Kontrollfluß bisher vernachlässigt!
- Arbeiten auf Basisblöcken
- **Nicht ausreichend** als allgemeine Zwischendarstellung
- Erweiterung auf **Graph** von Basisblöcken
 - Am Ende jedes Basisblockes (bedingter) Sprung zum nächsten Block
 - Kanten symbolisieren **Kontrollfluß**
- Sehr gut für viele Optimierungen brauchbar
- Häufig verwendete Zwischendarstellung im Optimierer



- Kontrollfluß bisher vernachlässigt!
- Arbeiten auf Basisblöcken
- **Nicht ausreichend** als allgemeine Zwischendarstellung
- Erweiterung auf **Graph** von Basisblöcken
 - Am Ende jedes Basisblockes (bedingter) Sprung zum nächsten Block
 - Kanten symbolisieren **Kontrollfluß**
- Sehr gut für viele Optimierungen brauchbar
- Häufig verwendete Zwischendarstellung im Optimierer



- Kontrollfluß bisher vernachlässigt!
- Arbeiten auf Basisblöcken
- **Nicht ausreichend** als allgemeine Zwischendarstellung
- Erweiterung auf **Graph** von Basisblöcken
 - Am Ende jedes Basisblockes (bedingter) Sprung zum nächsten Block
 - Kanten symbolisieren **Kontrollfluß**
- Sehr gut für viele Optimierungen brauchbar
- Häufig verwendete Zwischendarstellung im Optimierer



- Kontrollfluß bisher vernachlässigt!
- Arbeiten auf Basisblöcken
- **Nicht ausreichend** als allgemeine Zwischendarstellung
- Erweiterung auf **Graph** von Basisblöcken
 - Am Ende jedes Basisblockes (bedingter) Sprung zum nächsten Block
 - Kanten symbolisieren **Kontrollfluß**
- Sehr gut für viele Optimierungen brauchbar
- Häufig verwendete Zwischendarstellung im Optimierer



Engl. *control flow graph* (CFG)

- **Knoten** sind Basisblöcke
- **Kanten** sind Sprünge zwischen den Blöcken
- Sprünge treten also nur am **Ende** eines Blocks auf!
- Sprungziel ist immer ein **Blockanfang**

→ in diesem Fall: *Control Flow Graph*

→ in diesem Fall: *Control Flow Graph*



Engl. *control flow graph* (CFG)

- **Knoten** sind Basisblöcke
- **Kanten** sind Sprünge zwischen den Blöcken
- Sprünge treten also nur am **Ende** eines Blocks auf!
- Sprungziel ist immer ein **Blockanfang**
 - In Triangle: if/then/else, while/do
 - In LLVM: while/for/switch/branch/indirect branch/return/return with cleanup/return with cleanup and unwind label/return with cleanup and unwind label and unwind to here
 - Allgemeiner Fall deutlich komplizierter



Engl. *control flow graph* (CFG)

- **Knoten** sind Basisblöcke
- **Kanten** sind Sprünge zwischen den Blöcken
- Sprünge treten also nur am **Ende** eines Blocks auf!
- Sprungziel ist immer ein **Blockanfang**
 - In Triangle: if/then/else, while/do
 - In LLVM: Branching Instructions
 - Allgemeiner Fall deutlich komplizierter



Engl. *control flow graph* (CFG)

- **Knoten** sind Basisblöcke
- **Kanten** sind Sprünge zwischen den Blöcken
- Sprünge treten also nur am **Ende** eines Blocks auf!
- Sprungziel ist immer ein **Blockanfang**
 - In Triangle: if/then/else, while/do
 - Strukturierte Programmierung
 - Allgemeiner Fall deutlich komplizierter
 - goto
 - setjmp()/longjmp()



Engl. *control flow graph* (CFG)

- **Knoten** sind Basisblöcke
- **Kanten** sind Sprünge zwischen den Blöcken
- Sprünge treten also nur am **Ende** eines Blocks auf!
- Sprungziel ist immer ein **Blockanfang**
 - In Triangle: if/then/else, while/do
 - Strukturierte Programmierung
 - Allgemeiner Fall deutlich komplizierter
 - goto
 - setjmp()/longjmp()



Engl. *control flow graph* (CFG)

- **Knoten** sind Basisblöcke
- **Kanten** sind Sprünge zwischen den Blöcken
- Sprünge treten also nur am **Ende** eines Blocks auf!
- Sprungziel ist immer ein **Blockanfang**
 - In Triangle: if/then/else, while/do
 - Strukturierte Programmierung
 - Allgemeiner Fall deutlich komplizierter
 - goto
 - setjmp()/longjmp()



Engl. *control flow graph* (CFG)

- **Knoten** sind Basisblöcke
- **Kanten** sind Sprünge zwischen den Blöcken
- Sprünge treten also nur am **Ende** eines Blocks auf!
- Sprungziel ist immer ein **Blockanfang**
 - In Triangle: if/then/else, while/do
 - Strukturierte Programmierung
 - Allgemeiner Fall deutlich komplizierter
 - goto
 - setjmp()/longjmp()



Engl. *control flow graph* (CFG)

- **Knoten** sind Basisblöcke
- **Kanten** sind Sprünge zwischen den Blöcken
- Sprünge treten also nur am **Ende** eines Blocks auf!
- Sprungziel ist immer ein **Blockanfang**
 - In Triangle: if/then/else, while/do
 - Strukturierte Programmierung
 - Allgemeiner Fall deutlich komplizierter
 - goto
 - `setjmp()/longjmp()`



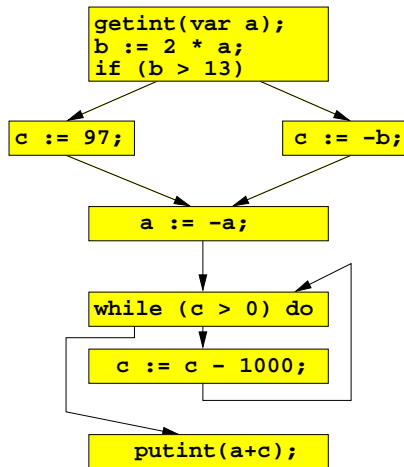
Engl. *control flow graph* (CFG)

- **Knoten** sind Basisblöcke
- **Kanten** sind Sprünge zwischen den Blöcken
- Sprünge treten also nur am **Ende** eines Blocks auf!
- Sprungziel ist immer ein **Blockanfang**
 - In Triangle: if/then/else, while/do
 - Strukturierte Programmierung
 - Allgemeiner Fall deutlich komplizierter
 - goto
 - setjmp()/longjmp()

Beispiel Kontrollflußgraph 1



```
getint(var a);  
b := 2 * a;  
if (b > 13) then  
  c := 97;  
else  
  c := -b;  
a := -a;  
while (c > 0) do  
  c := c - 1000;  
putint(a+c);
```



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Beispiel Kontrollflußgraph 2



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

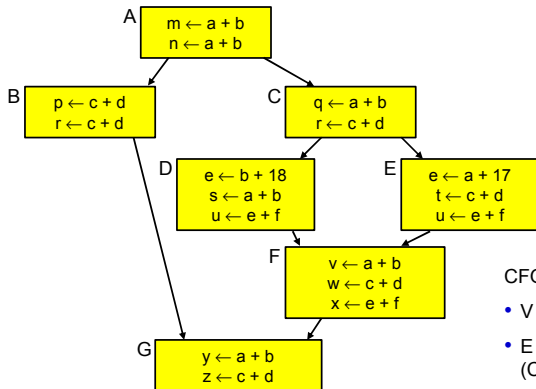
Wertnumerierung

Lokal

Superlokal

Regional

Global



CFG $G = (V, E)$

- $V = \{A, B, C, D, E, F, G\}$
- $E = \{(A, B), (A, C), (B, G), (C, D), (C, E), (D, F), (E, F), (F, E)\}$
- $|V| = 7, |E| = 8$



- Eben beschrieben: Einfachste Variante
- Idee: Idee der **Variablenversionen** direkt im CFG widerspiegeln
- Konzept: Jede Variable wird im CFG **genau einmal** zugewiesen
- **Static Single Assignment**-Form

Ab hier Sprung in 7. Block, später Rückkehr hierher!



- Eben beschrieben: Einfachste Variante
- Idee: Idee der **Variablenversionen** direkt im CFG widerspiegeln
- Konzept: Jede Variable wird im CFG **genau einmal** zugewiesen
- **Static Single Assignment**-Form

Ab hier Sprung in 7. Block, später Rückkehr hierher!



- Eben beschrieben: Einfachste Variante
- Idee: Idee der **Variablenversionen** direkt im CFG widerspiegeln
- Konzept: Jede Variable wird im CFG **genau einmal** zugewiesen
- **Static Single Assignment**-Form

Ab hier Sprung in 7. Block, später Rückkehr hierher!



- Eben beschrieben: Einfachste Variante
- Idee: Idee der **Variablenversionen** direkt im CFG widerspiegeln
- Konzept: Jede Variable wird im CFG **genau einmal** zugewiesen
- **Static Single Assignment**-Form

Ab hier Sprung in 7. Block, später Rückkehr hierher!



- Eben beschrieben: Einfachste Variante
- Idee: Idee der **Variablenversionen** direkt im CFG widerspiegeln
- Konzept: Jede Variable wird im CFG **genau einmal** zugewiesen
- **Static Single Assignment**-Form

Ab hier Sprung in 7. Block, später Rückkehr hierher!



- Eben beschrieben: Einfachste Variante
- Idee: Idee der **Variablenversionen** direkt im CFG widerspiegeln
- Konzept: Jede Variable wird im CFG **genau einmal** zugewiesen
- **Static Single Assignment**-Form

Ab hier Sprung in 7. Block, später Rückkehr hierher!



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

CFG-basierte Redundanzeliminierung mit Wertnumerierung



- **Verwaltet nun Folgen von Anweisungen**
 - Häufige IR, z.B. in SUIF
- Leichter umzustellen



- Verwaltet nun Folgen von Anweisungen
 - Häufige IR, z.B. in SUIF
- Leichter umzustellen



- Verwaltet nun Folgen von Anweisungen
 - Häufige IR, z.B. in SUIF
- Leichter umzustellen



Nun keine Analyse des AST mehr!

Idee

- Verwalte nun nicht Variablennamen in Ausdrücken
 - ... sondern direkt ihre Werte
- Jeder zur Laufzeit verwendete Wert wird durch einen Wert ersetzt
- Der Wert eines Ausdrucks ist ein int (bzw. float)
- Jeder Wert ist ein int (bzw. float)



Nun keine Analyse des AST mehr!

Idee

- Verwalte nun nicht Variablen**namen** in Ausdrücken
- ... sondern direkt ihre **Werte**
 - Jeder zur Laufzeit berechnete Wert wird durchnummeriert
 - Der Wert zweier Ausdrücke e_1 und e_2 bekommt dieselbe Nummer
 - ... wenn e_1 und e_2 **beweisbar immer denselben Wert** liefern



Nun keine Analyse des AST mehr!

Idee

- Verwalte nun nicht Variablen**namen** in Ausdrücken
- ... sondern direkt ihre **Werte**
 - Jeder zur Laufzeit berechnete Wert wird durchnumeriert
 - Der Wert zweier Ausdrücke e_1 und e_2 bekommt dieselbe Nummer
 - ... wenn e_1 und e_2 **beweisbar immer** denselben Wert liefern



Nun keine Analyse des AST mehr!

Idee

- Verwalte nun nicht Variablen**namen** in Ausdrücken
- ... sondern direkt ihre **Werte**
 - Jeder zur Laufzeit berechnete Wert wird durchnummeriert
 - Der Wert zweier Ausdrücke e_1 und e_2 bekommt dieselbe Nummer
 - ... wenn e_1 und e_2 **beweisbar immer** denselben Wert liefern



Nun keine Analyse des AST mehr!

Idee

- Verwalte nun nicht Variablen**namen** in Ausdrücken
- ... sondern direkt ihre **Werte**
 - Jeder zur Laufzeit berechnete Wert wird durchnummeriert
 - Der Wert zweier Ausdrücke e_1 und e_2 bekommt dieselbe Nummer
 - ... wenn e_1 und e_2 **beweisbar immer** denselben Wert liefern



Nun keine Analyse des AST mehr!

Idee

- Verwalte nun nicht Variablen**namen** in Ausdrücken
- ... sondern direkt ihre **Werte**
 - Jeder zur Laufzeit berechnete Wert wird durchnummeriert
 - Der Wert zweier Ausdrücke e_1 und e_2 bekommt dieselbe Nummer
 - ... wenn e_1 und e_2 **beweisbar immer** denselben Wert liefern

Value Numbering (VN)



- e_1 und e_2 **redundant**, wenn
 - sie ihre Operanden mit gleichen Operatoren verknüpfen
 - **und** die Operanden die gleichen Wertnummern haben

Implementierung mit Hash-Table

- Hashe Variablen und Konstanten mit Textdarstellung (`.spelling`)
- Hashe ganze Ausdrücke bestehend aus Operatoren und Wertnummern
- Neue Werte (neue Hash-Keys) bekommen neue Wertnummer zugewiesen

Wichtig: Ausdrücke gleichen Hash-Wertes **müssen** zur Laufzeit gleichen Wert liefern.

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Value Numbering (VN)



- e_1 und e_2 **redundant**, wenn
 - sie ihre Operanden mit gleichen Operatoren verknüpfen
 - **und** die Operanden die gleichen Wertnummern haben

Implementierung mit Hash-Table

- Hashe Variablen und Konstanten mit Textdarstellung (`.spelling`)
- Hashe ganze Ausdrücke bestehend aus Operatoren und Wertnummern
- Neue Werte (neue Hash-Keys) bekommen neue Wertnummer zugewiesen

Wichtig: Ausdrücke gleichen Hash-Wertes **müssen** zur Laufzeit gleichen Wert liefern.

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Value Numbering (VN)



- e_1 und e_2 **redundant**, wenn
 - sie ihre Operanden mit gleichen Operatoren verknüpfen
 - **und** die Operanden die gleichen Wertnummern haben

Implementierung mit Hash-Table

- Hashe Variablen und Konstanten mit Textdarstellung (`.spelling`)
- Hashe ganze Ausdrücke bestehend aus Operatoren und Wertnummern
- Neue Werte (neue Hash-Keys) bekommen neue Wertnummer zugewiesen

Wichtig: Ausdrücke gleichen Hash-Wertes **müssen** zur Laufzeit gleichen Wert liefern.

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Value Numbering (VN)



- e_1 und e_2 **redundant**, wenn
 - sie ihre Operanden mit gleichen Operatoren verknüpfen
 - **und** die Operanden die gleichen Wertnummern haben

Implementierung mit Hash-Table

- Hashe Variablen und Konstanten mit Textdarstellung (`.spelling`)
- Hashe ganze Ausdrücke bestehend aus Operatoren und Wertnummern
- Neue Werte (neue Hash-Keys) bekommen neue Wertnummer zugewiesen

Wichtig: Ausdrücke gleichen Hash-Wertes **müssen** zur Laufzeit gleichen Wert liefern.

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Value Numbering (VN)



- e_1 und e_2 **redundant**, wenn
 - sie ihre Operanden mit gleichen Operatoren verknüpfen
 - **und** die Operanden die gleichen Wertnummern haben

Implementierung mit Hash-Table

- Hashe Variablen und Konstanten mit Textdarstellung (`.spelling`)
- Hashe ganze Ausdrücke bestehend aus Operatoren und Wertnummern
- Neue Werte (neue Hash-Keys) bekommen neue Wertnummer zugewiesen

Wichtig: Ausdrücke gleichen Hash-Wertes **müssen** zur Laufzeit gleichen Wert liefern.

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global



- e_1 und e_2 **redundant**, wenn
 - sie ihre Operanden mit gleichen Operatoren verknüpfen
 - **und** die Operanden die gleichen Wertnummern haben

Implementierung mit Hash-Table

- Hashe Variablen und Konstanten mit Textdarstellung (**.spelling**)
 - Alternativ in Triangle: Adresse der Deklaration im AST
- Hashe ganze Ausdrücke bestehend aus Operatoren und Wertnummern
- Neue Werte (neue Hash-Keys) bekommen neue Wertnummer zugewiesen

Wichtig: Ausdrücke gleichen Hash-Wertes **müssen** zur Laufzeit gleichen Wert liefern.



- e_1 und e_2 **redundant**, wenn
 - sie ihre Operanden mit gleichen Operatoren verknüpfen
 - **und** die Operanden die gleichen Wertnummern haben

Implementierung mit Hash-Table

- Hashe Variablen und Konstanten mit Textdarstellung (**.spelling**)
 - Alternativ in Triangle: Adresse der Deklaration im AST
- Hashe ganze Ausdrücke bestehend aus Operatoren und Wertnummern
- Neue Werte (neue Hash-Keys) bekommen neue Wertnummer zugewiesen

Wichtig: Ausdrücke gleichen Hash-Wertes **müssen** zur Laufzeit gleichen Wert liefern.



- e_1 und e_2 **redundant**, wenn
 - sie ihre Operanden mit gleichen Operatoren verknüpfen
 - **und** die Operanden die gleichen Wertnummern haben

Implementierung mit Hash-Table

- Hashe Variablen und Konstanten mit Textdarstellung (**.spelling**)
 - Alternativ in Triangle: Adresse der Deklaration im AST
- Hashe ganze Ausdrücke bestehend aus Operatoren und Wertnummern
- Neue Werte (neue Hash-Keys) bekommen neue Wertnummer zugewiesen

Wichtig: Ausdrücke gleichen Hash-Wertes **müssen** zur Laufzeit gleichen Wert liefern.



- e_1 und e_2 **redundant**, wenn
 - sie ihre Operanden mit gleichen Operatoren verknüpfen
 - **und** die Operanden die gleichen Wertnummern haben

Implementierung mit Hash-Table

- Hashe Variablen und Konstanten mit Textdarstellung (**.spelling**)
 - Alternativ in Triangle: Adresse der Deklaration im AST
- Hashe ganze Ausdrücke bestehend aus Operatoren und Wertnummern
- Neue Werte (neue Hash-Keys) bekommen neue Wertnummer zugewiesen

Wichtig: Ausdrücke gleichen Hash-Wertes **müssen** zur Laufzeit gleichen Wert liefern.



- e_1 und e_2 **redundant**, wenn
 - sie ihre Operanden mit gleichen Operatoren verknüpfen
 - **und** die Operanden die gleichen Wertnummern haben

Implementierung mit Hash-Table

- Hashe Variablen und Konstanten mit Textdarstellung (**.spelling**)
 - Alternativ in Triangle: Adresse der Deklaration im AST
- Hashe ganze Ausdrücke bestehend aus Operatoren und Wertnummern
- Neue Werte (neue Hash-Keys) bekommen neue Wertnummer zugewiesen

Wichtig: Ausdrücke gleichen Hash-Wertes **müssen** zur Laufzeit gleichen Wert liefern.



- e_1 und e_2 **redundant**, wenn
 - sie ihre Operanden mit gleichen Operatoren verknüpfen
 - **und** die Operanden die gleichen Wertnummern haben

Implementierung mit Hash-Table

- Hashe Variablen und Konstanten mit Textdarstellung (**.spelling**)
 - Alternativ in Triangle: Adresse der Deklaration im AST
- Hashe ganze Ausdrücke bestehend aus Operatoren und Wertnummern
- Neue Werte (neue Hash-Keys) bekommen neue Wertnummer zugewiesen

Wichtig: Ausdrücke gleichen Hash-Wertes **müssen** zur Laufzeit gleichen Wert liefern.



Wie Kollisionen handhaben?

- Bei einfachen Ausdrücken $x \text{ op } y$ leicht vermeidbar
 - 4b für Operator
 - je 14b für Wertnummern der Teilausdrücke
 - insgesamt ein 32b Wort
 - Eindeutig bis max. 16384 Teilausdrücken
- Bei komplizierteren Ausdrücken
 $a \text{ op1 } b \text{ op2 } c \text{ op3 } \dots$
 - Kollisionen nicht mehr einfach vermeidbar
 - Zweistufiges Vorgehen: Hash+Vergleich
 - Alternative: Aufteilen in einfache Ausdrücke
z.B. $(a \text{ op1 } b) \text{ op2 } c$ oder $a \text{ op1 } (b \text{ op2 } c)$
- Hier Beschränkung auf einfache Ausdrücke



Wie Kollisionen handhaben?

- Bei einfachen Ausdrücken $x \text{ op } y$ leicht vermeidbar
 - 4b für Operator
 - je 14b für Wertnummern der Teilausdrücke
 - insgesamt ein 32b Wort
 - Eindeutig bis max. 16384 Teilausdrücken
- Bei komplizierteren Ausdrücken
 $a \text{ op1 } b \text{ op2 } c \text{ op3 } \dots$
 - Kollisionen nicht mehr einfach vermeidbar
 - Zweistufiges Vorgehen: Hash+Vergleich
 - Alternative: Aufteilen in einfache Ausdrücke
- Hier Beschränkung auf einfache Ausdrücke



Wie Kollisionen handhaben?

- Bei einfachen Ausdrücken $x \text{ op } y$ leicht vermeidbar
 - 4b für Operator
 - je 14b für Wertnummern der Teilausdrücke
 - insgesamt ein 32b Wort
 - Eindeutig bis max. 16384 Teilausdrücken
- Bei komplizierteren Ausdrücken
 $a \text{ op1 } b \text{ op2 } c \text{ op3 } \dots$
 - Kollisionen nicht mehr einfach vermeidbar
 - Zweistufiges Vorgehen: Hash+Vergleich
 - Alternative: Aufteilen in einfache Ausdrücke
z.B. $(a \text{ op1 } b) \text{ op2 } c$
- Hier Beschränkung auf einfache Ausdrücke



Wie Kollisionen handhaben?

- Bei einfachen Ausdrücken $x \text{ op } y$ leicht vermeidbar
 - 4b für Operator
 - je 14b für Wertnummern der Teilausdrücke
 - insgesamt ein 32b Wort
 - Eindeutig bis max. 16384 Teilausdrücken
- Bei komplizierteren Ausdrücken
 $a \text{ op1 } b \text{ op2 } c \text{ op3 } \dots$
 - Kollisionen nicht mehr einfach vermeidbar
 - Zweistufiges Vorgehen: Hash+Vergleich
 - Alternative: Aufteilen in einfache Ausdrücke
- Hier Beschränkung auf einfache Ausdrücke



Wie Kollisionen handhaben?

- Bei einfachen Ausdrücken $x \text{ op } y$ leicht vermeidbar
 - 4b für Operator
 - je 14b für Wertnummern der Teilausdrücke
 - insgesamt ein 32b Wort
 - Eindeutig bis max. 16384 Teilausdrücken
- Bei komplizierteren Ausdrücken
 $a \text{ op1 } b \text{ op2 } c \text{ op3 } \dots$
 - Kollisionen nicht mehr einfach vermeidbar
 - Zweistufiges Vorgehen: Hash+Vergleich
 - Alternative: Aufteilen in einfache Ausdrücke
- Hier Beschränkung auf einfache Ausdrücke



Wie Kollisionen handhaben?

- Bei einfachen Ausdrücken $x \text{ op } y$ leicht vermeidbar
 - 4b für Operator
 - je 14b für Wertnummern der Teilausdrücke
 - insgesamt ein 32b Wort
 - Eindeutig bis max. 16384 Teilausdrücken
- Bei komplizierteren Ausdrücken
 $a \text{ op1 } b \text{ op2 } c \text{ op3 } \dots$
 - Kollisionen nicht mehr einfach vermeidbar
 - Zweistufiges Vorgehen: Hash+Vergleich
 - Alternative: Aufteilen in einfache Ausdrücke
 - Auch in Triangle (Baum aus Teilausdrücken)
- Hier Beschränkung auf einfache Ausdrücke



Wie Kollisionen handhaben?

- Bei einfachen Ausdrücken $x \text{ op } y$ leicht vermeidbar
 - 4b für Operator
 - je 14b für Wertnummern der Teilausdrücke
 - insgesamt ein 32b Wort
 - Eindeutig bis max. 16384 Teilausdrücken
- Bei komplizierteren Ausdrücken
 $a \text{ op1 } b \text{ op2 } c \text{ op3 } \dots$
 - Kollisionen nicht mehr einfach vermeidbar
 - Zweistufiges Vorgehen: Hash+Vergleich
 - Alternative: Aufteilen in einfache Ausdrücke
 - Auch in Triangle (Baum aus Teilausdrücken)
- Hier Beschränkung auf einfache Ausdrücke



Wie Kollisionen handhaben?

- Bei einfachen Ausdrücken $x \text{ op } y$ leicht vermeidbar
 - 4b für Operator
 - je 14b für Wertnummern der Teilausdrücke
 - insgesamt ein 32b Wort
 - Eindeutig bis max. 16384 Teilausdrücken
- Bei komplizierteren Ausdrücken
 $a \text{ op1 } b \text{ op2 } c \text{ op3 } \dots$
 - Kollisionen nicht mehr einfach vermeidbar
 - Zweistufiges Vorgehen: Hash+Vergleich
 - Alternative: Aufteilen in einfache Ausdrücke
 - Auch in Triangle (Baum aus Teilausdrücken)
- Hier Beschränkung auf einfache Ausdrücke



Wie Kollisionen handhaben?

- Bei einfachen Ausdrücken $x \text{ op } y$ leicht vermeidbar
 - 4b für Operator
 - je 14b für Wertnummern der Teilausdrücke
 - insgesamt ein 32b Wort
 - Eindeutig bis max. 16384 Teilausdrücken
- Bei komplizierteren Ausdrücken
 $a \text{ op1 } b \text{ op2 } c \text{ op3 } \dots$
 - Kollisionen nicht mehr einfach vermeidbar
 - Zweistufiges Vorgehen: Hash+Vergleich
 - Alternative: Aufteilen in einfache Ausdrücke
 - Auch in Triangle (Baum aus Teilausdrücken)
- Hier Beschränkung auf einfache Ausdrücke



Wie Kollisionen handhaben?

- Bei einfachen Ausdrücken $x \text{ op } y$ leicht vermeidbar
 - 4b für Operator
 - je 14b für Wertnummern der Teilausdrücke
 - insgesamt ein 32b Wort
 - Eindeutig bis max. 16384 Teilausdrücken
- Bei komplizierteren Ausdrücken
 $a \text{ op1 } b \text{ op2 } c \text{ op3 } \dots$
 - Kollisionen nicht mehr einfach vermeidbar
 - Zweistufiges Vorgehen: Hash+Vergleich
 - Alternative: Aufteilen in einfache Ausdrücke
 - Auch in Triangle (Baum aus Teilausdrücken)
- Hier Beschränkung auf einfache Ausdrücke



Wie Kollisionen handhaben?

- Bei einfachen Ausdrücken $x \text{ op } y$ leicht vermeidbar
 - 4b für Operator
 - je 14b für Wertnummern der Teilausdrücke
 - insgesamt ein 32b Wort
 - Eindeutig bis max. 16384 Teilausdrücken
- Bei komplizierteren Ausdrücken
 $a \text{ op1 } b \text{ op2 } c \text{ op3 } \dots$
 - Kollisionen nicht mehr einfach vermeidbar
 - Zweistufiges Vorgehen: Hash+Vergleich
 - Alternative: Aufteilen in einfache Ausdrücke
 - Auch in Triangle (Baum aus Teilausdrücken)
- Hier Beschränkung auf einfache Ausdrücke

Beispiel Value Numbering



```
x := a + d;  
y := a;  
z := y + d;
```

Ausdruck	... auf Wertnummern	Wertnummer Ergebnis
a	-	1
d	-	2
a + d	1+2	3
x	3	3
a	1	1
y	1	1
y + d	1+2	3
z	3	3

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Beispiel Value Numbering



```
x := a + d;  
y := a;  
z := y + d;
```

Ausdruck	... auf Wertnummern	Wertnummer Ergebnis
a	-	1
d	-	2
a + d	1+2	3
x	3	3
a	1	1
y	1	1
y + d	1+2	3
z	3	3

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global



Für jeden Ausdruck e der Form $r_e \leftarrow x_e \text{ op}_e y_e$ im Block

- 1 Bestimme Wertnummern von x_e, y_e
- 2 Bestimme Hash-Wert für x_e, y_e, op_e
- 3 Hash-Wert bekannt?
 - Ja: ersetze e durch Kopie von bekanntem Ausdruck, trage dessen Wertnummer für r_e ein.
 - Nein: Trage Hash-Wert neu in Tabelle ein, vergebe neue Wertnummer und trage diese für r_e ein.



Für jeden Ausdruck e der Form $r_e \leftarrow x_e \text{ op}_e y_e$ im Block

- 1 Bestimme Wertnummern von x_e, y_e
- 2 Bestimme Hash-Wert für x_e, y_e, op_e
- 3 Hash-Wert bekannt?
 - Ja: ersetze e durch Kopie von bekanntem Ausdruck, trage dessen Wertnummer für r_e ein.
 - Nein: Trage Hash-Wert neu in Tabelle ein, vergebe neue Wertnummer und trage diese für r_e ein.



Für jeden Ausdruck e der Form $r_e \leftarrow x_e \text{ op}_e y_e$ im Block

- 1 Bestimme Wertnummern von x_e, y_e
- 2 Bestimme Hash-Wert für x_e, y_e, op_e
- 3 Hash-Wert bekannt?
 - Ja: ersetze e durch Kopie von bekanntem Ausdruck, trage dessen Wertnummer für r_e ein.
 - Nein: Trage Hash-Wert neu in Tabelle ein, vergebe neue Wertnummer und trage diese für r_e ein.



Für jeden Ausdruck e der Form $r_e \leftarrow x_e \text{ op}_e y_e$ im Block

- 1 Bestimme Wertnummern von x_e, y_e
- 2 Bestimme Hash-Wert für x_e, y_e, op_e
- 3 Hash-Wert bekannt?
 - Ja: ersetze e durch Kopie von bekanntem Ausdruck, trage dessen Wertnummer für r_e ein.
 - Nein: Trage Hash-Wert neu in Tabelle ein, vergebe neue Wertnummer und trage diese für r_e ein.



Für jeden Ausdruck e der Form $r_e \leftarrow x_e \text{ op}_e y_e$ im Block

- 1 Bestimme Wertnummern von x_e, y_e
- 2 Bestimme Hash-Wert für x_e, y_e, op_e
- 3 Hash-Wert bekannt?
 - Ja: ersetze e durch Kopie von bekanntem Ausdruck, trage dessen Wertnummer für r_e ein.
 - Nein: Trage Hash-Wert neu in Tabelle ein, vergebe neue Wertnummer und trage diese für r_e ein.

Diskussion Value Numbering (VN) 1



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

- **Kommutative Operatoren**
 - Schlage mit beiden Operandenreihenfolgen nach
- Auch Constant Folding während VN möglich
 - Flag in Tabelle für "konstanter Wert"
- Berücksichtige algebraische Eigenschaften beim Hashen
 - Viele Sonderfälle, baue Entscheidungsbaum für Operator
 - Auf Wertnummern, *nicht* auf Variablen

Diskussion Value Numbering (VN) 1



- Kommutative Operatoren
 - Schlage mit beiden Operandenreihenfolgen nach
- Auch Constant Folding während VN möglich
 - Flag in Tabelle für "konstanter Wert"
- Berücksichtige algebraische Eigenschaften beim Hashen
 - Viele Sonderfälle, baue Entscheidungsbaum für Operator
 - Auf Wertnummern, *nicht* auf Variablen

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Diskussion Value Numbering (VN) 1



- Kommutative Operatoren
 - Schlage mit beiden Operandenreihenfolgen nach
- Auch Constant Folding während VN möglich
 - Flag in Tabelle für "konstanter Wert"
- Berücksichtige algebraische Eigenschaften beim Hashen
 - Viele Sonderfälle, baue Entscheidungsbaum für Operator
 - Auf Wertnummern, *nicht* auf Variablen

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Diskussion Value Numbering (VN) 1



- Kommutative Operatoren
 - Schlage mit beiden Operandenreihenfolgen nach
- Auch Constant Folding während VN möglich
 - Flag in Tabelle für “konstanter Wert”
- Berücksichtige algebraische Eigenschaften beim Hashen
 - Viele Sonderfälle, baue Entscheidungsbaum für Operator
 - Auf Wertnummern, *nicht* auf Variablen

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global



- Kommutative Operatoren
 - Schlage mit beiden Operandenreihenfolgen nach
- Auch Constant Folding während VN möglich
 - Flag in Tabelle für “konstanter Wert”
- Berücksichtige algebraische Eigenschaften beim Hashen
 - Viele Sonderfälle, baue Entscheidungsbaum für Operator
 - Auf Wertnummern, *nicht* auf Variablen

Algebraische Eigenschaften

$x \leftarrow y$, $x+0$, $x-0$, $x*1$, $x\div 1$, $x-x$, $x*0$,
 $x\div x$, $x\vee 0$, $x \wedge 0x\text{FF}\dots\text{FF}$,
 $\max(x, \text{MAXINT})$, $\min(x, \text{MININT})$,
 $\max(x, x)$, $\min(y, y)$, and so on ...



- Kommutative Operatoren
 - Schlage mit beiden Operandenreihenfolgen nach
- Auch Constant Folding während VN möglich
 - Flag in Tabelle für “konstanter Wert”
- Berücksichtige algebraische Eigenschaften beim Hashen
 - Viele Sonderfälle, baue Entscheidungsbaum für Operator
 - Auf Wertnummern, *nicht* auf Variablen

Algebraische Eigenschaften

$x \leftarrow y$, $x+0$, $x-0$, $x*1$, $x\div 1$, $x-x$, $x*0$,
 $x\div x$, $x\vee 0$, $x \wedge 0xFF\dots FF$,
 $\max(x, \text{MAXINT})$, $\min(x, \text{MININT})$,
 $\max(x, x)$, $\min(y, y)$, and so on ...



- Kommutative Operatoren
 - Schlage mit beiden Operandenreihenfolgen nach
- Auch Constant Folding während VN möglich
 - Flag in Tabelle für “konstanter Wert”
- Berücksichtige algebraische Eigenschaften beim Hashen
 - Viele Sonderfälle, baue Entscheidungsbaum für Operator
 - Auf Wertnummern, *nicht* auf Variablen

Algebraische Eigenschaften

$x \leftarrow y$, $x+0$, $x-0$, $x*1$, $x\div 1$, $x-x$, $x*0$,
 $x\div x$, $x\vee 0$, $x \wedge 0$ xFF...FF,
 $\max(x, \text{MAXINT})$, $\min(x, \text{MININT})$,
 $\max(x, x)$, $\min(y, y)$, and so on ...

Diskussion Value Numbering (VN) 2



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

- Schreiben über Zeiger löscht *alle* Wertnummern
- Schreiben auf Arrayelement mit variablem Index
 - Löscht Wertnummern aller Elemente dieses Arrays
- Schreiben auf ganze Record-Variable
 - Löscht Wertnummern aller Komponenten dieser Record-Variable
- Prozeduraufruf
 - Löscht Wertnummern von `var`-Parametern
 - Löscht Wertnummern von globalen und nicht-lokalen Variablen



- Schreiben über Zeiger löscht *alle* Wertnummern
- Schreiben auf Arrayelement mit variablem Index
 - Löscht Wertnummern aller Elemente dieses Arrays
- Schreiben auf ganze Record-Variable
 - Löscht Wertnummern aller Komponenten dieser Record-Variable
- Prozeduraufruf
 - Löscht Wertnummern von `var`-Parametern
 - Löscht Wertnummern von globalen und nicht-lokalen Variablen

Diskussion Value Numbering (VN) 2



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

- Schreiben über Zeiger löscht *alle* Wertnummern
- Schreiben auf Arrayelement mit variablem Index
 - Löscht Wertnummern aller Elemente dieses Arrays
- Schreiben auf ganze Record-Variable
 - Löscht Wertnummern aller Komponenten dieser Record-Variable
- Prozeduraufruf
 - Löscht Wertnummern von `var`-Parametern
 - Löscht Wertnummern von globalen und nicht-lokalen Variablen

Diskussion Value Numbering (VN) 2



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

- Schreiben über Zeiger löscht *alle* Wertnummern
- Schreiben auf Arrayelement mit variablem Index
 - Löscht Wertnummern aller Elemente dieses Arrays
- Schreiben auf ganze Record-Variable
 - Löscht Wertnummern aller Komponenten dieser Record-Variable
- Prozeduraufruf
 - Löscht Wertnummern von `var`-Parametern
 - Löscht Wertnummern von globalen und nicht-lokalen Variablen



- Schreiben über Zeiger löscht *alle* Wertnummern
- Schreiben auf Arrayelement mit variablem Index
 - Löscht Wertnummern aller Elemente dieses Arrays
- Schreiben auf ganze Record-Variable
 - Löscht Wertnummern aller Komponenten dieser Record-Variable
- Prozeduraufruf
 - Löscht Wertnummern von `var`-Parametern
 - Löscht Wertnummern von globalen und nicht-lokalen Variablen



- Schreiben über Zeiger löscht *alle* Wertnummern
- Schreiben auf Arrayelement mit variablem Index
 - Löscht Wertnummern aller Elemente dieses Arrays
- Schreiben auf ganze Record-Variable
 - Löscht Wertnummern aller Komponenten dieser Record-Variable
- Prozeduraufruf
 - Löscht Wertnummern von `var`-Parametern
 - Löscht Wertnummern von globalen und nicht-lokalen Variablen



- Schreiben über Zeiger löscht *alle* Wertnummern
- Schreiben auf Arrayelement mit variablem Index
 - Löscht Wertnummern aller Elemente dieses Arrays
- Schreiben auf ganze Record-Variable
 - Löscht Wertnummern aller Komponenten dieser Record-Variable
- Prozeduraufruf
 - Löscht Wertnummern von **var**-Parametern
 - Löscht Wertnummern von globalen und nicht-lokalen Variablen



- Schreiben über Zeiger löscht *alle* Wertnummern
- Schreiben auf Arrayelement mit variablem Index
 - Löscht Wertnummern aller Elemente dieses Arrays
- Schreiben auf ganze Record-Variable
 - Löscht Wertnummern aller Komponenten dieser Record-Variable
- Prozeduraufruf
 - Löscht Wertnummern von **var**-Parametern
 - Löscht Wertnummern von globalen und nicht-lokalen Variablen

Beispiel VN mit Umschreiben



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Bisher: Erkennen von Redundanzen, jetzt auch
Umschreiben des Codes

Eingabe-Code

Value Numbering

Umschreiben

Beispiel VN mit Umschreiben



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Bisher: Erkennen von Redundanzen, jetzt auch
Umschreiben des Codes

Eingabe-Code

```
a := b + c;
```

```
b := a - d;
```

```
c := b + c;
```

```
d := a - d;
```

Value Numbering

Umschreiben

Beispiel VN mit Umschreiben



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Bisher: Erkennen von Redundanzen, jetzt auch
Umschreiben des Codes

Eingabe-Code

```
a := b + c;
```

```
b := a - d;
```

```
c := b + c;
```

```
d := a - d;
```

Value Numbering

```
a3 := b1 + c2;
```

```
b5 := a3 - d4;
```

```
c6 := b5 + c2;
```

```
d5 := a3 - d4;
```

Umschreiben

Beispiel VN mit Umschreiben



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Bisher: Erkennen von Redundanzen, jetzt auch
Umschreiben des Codes

Eingabe-Code

```
a := b + c;  
b := a - d;  
c := b + c;  
d := a - d;
```

Value Numbering

```
a3 := b1 + c2;  
b5 := a3 - d4;  
c6 := b5 + c2;  
d5 := a3 - d4;
```

Umschreiben

```
a := b + c;  
b := a - d;  
c := b + c;  
d := b;
```



Bisher: Zugriff auf Werte über Namen (von Variablen)

- Zugriff auf Wert 3 über Variablennamen `a`
- Nicht mehr möglich!
- Möglichkeiten

- `a` → `3` über `Index` (über den Wert haltende Variablen (Array))
- `a` → `3` über `Value` (über den Namen haltende Variablen)



Bisher: Zugriff auf Werte über Namen (von Variablen)

Eingabe-Code

```
a ← x + y
* b ← x + y
  a ← 17
* c ← x + y
```

- Zugriff auf Wert 3 über Variablennamen `a`
- Nicht mehr möglich!
- Möglichkeiten

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global



Bisher: Zugriff auf Werte über Namen (von Variablen)

Eingabe-Code

```
a ← x + y
* b ← x + y
a ← 17
* c ← x + y
```

Value Numbering

```
a3 ← x1 + y2
* b3 ← x1 + y2
a4 ← 17
* c3 ← x1 + y2
```

- Zugriff auf Wert 3 über Variablennamen a
- Nicht mehr möglich!
- Möglichkeiten

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global



Bisher: Zugriff auf Werte über Namen (von Variablen)

Eingabe-Code

```
a ← x + y
* b ← x + y
a ← 17
* c ← x + y
```

Value Numbering

```
a3 ← x1 + y2
* b3 ← x1 + y2
a4 ← 17
* c3 ← x1 + y2
```

Umgeschrieben

```
a3 ← x1 + y2
* b3 ← a3
a4 ← 17
* c3 ← a3 (oops!)
```

- Zugriff auf Wert 3 über Variablennamen a
- Nicht mehr möglich!
- Möglichkeiten



Bisher: Zugriff auf Werte über Namen (von Variablen)

Eingabe-Code

```
a ← x + y
* b ← x + y
a ← 17
* c ← x + y
```

Value Numbering

```
a3 ← x1 + y2
* b3 ← x1 + y2
a4 ← 17
* c3 ← x1 + y2
```

Umgeschrieben

```
a3 ← x1 + y2
* b3 ← a3
a4 ← 17
* c3 ← a3 (oops!)
```

- Zugriff auf Wert 3 über Variablennamen **a**
- Nicht mehr möglich!
- Möglichkeiten
 - Führe Buch über den Wert haltende Variablen (hier **b**)
 - Mache Sicherheitskopien von Variablen (**a³** nach **t³**)
 - Vergebe eindeutige Namen für Zuweisungen



Bisher: Zugriff auf Werte über Namen (von Variablen)

Eingabe-Code

```
a ← x + y
* b ← x + y
a ← 17
* c ← x + y
```

Value Numbering

```
a3 ← x1 + y2
* b3 ← x1 + y2
a4 ← 17
* c3 ← x1 + y2
```

Umgeschrieben

```
a3 ← x1 + y2
* b3 ← a3
a4 ← 17
* c3 ← a3 (oops!)
```

- Zugriff auf Wert 3 über Variablennamen **a**
- Nicht mehr möglich!
- Möglichkeiten
 - Führe Buch über den Wert haltende Variablen (hier **b**)
 - Mache Sicherheitskopien von Variablen (**a³** nach **t³**)
 - Vergebe eindeutige Namen für Zuweisungen



Bisher: Zugriff auf Werte über Namen (von Variablen)

Eingabe-Code

```
a ← x + y
* b ← x + y
a ← 17
* c ← x + y
```

Value Numbering

```
a3 ← x1 + y2
* b3 ← x1 + y2
a4 ← 17
* c3 ← x1 + y2
```

Umgeschrieben

```
a3 ← x1 + y2
* b3 ← a3
a4 ← 17
* c3 ← a3 (oops!)
```

- Zugriff auf Wert 3 über Variablennamen **a**
- Nicht mehr möglich!
- Möglichkeiten
 - Führe Buch über den Wert haltende Variablen (hier **b**)
 - Mache Sicherheitskopien von Variablen (**a³** nach **t³**)
 - **Vergebe eindeutige Namen für Zuweisungen**
 - Kein Überschreiben mehr möglich



Bisher: Zugriff auf Werte über Namen (von Variablen)

Eingabe-Code

```
a ← x + y
* b ← x + y
a ← 17
* c ← x + y
```

Value Numbering

```
a3 ← x1 + y2
* b3 ← x1 + y2
a4 ← 17
* c3 ← x1 + y2
```

Umgeschrieben

```
a3 ← x1 + y2
* b3 ← a3
a4 ← 17
* c3 ← a3 (oops!)
```

- Zugriff auf Wert 3 über Variablennamen **a**
- Nicht mehr möglich!
- Möglichkeiten
 - Führe Buch über den Wert haltende Variablen (hier **b**)
 - Mache Sicherheitskopien von Variablen (**a³** nach **t³**)
 - **Vergebe eindeutige Namen für Zuweisungen**
 - Kein Überschreiben mehr möglich



Bisher: Zugriff auf Werte über Namen (von Variablen)

Eingabe-Code

```
a ← x + y
* b ← x + y
a ← 17
* c ← x + y
```

Value Numbering

```
a3 ← x1 + y2
* b3 ← x1 + y2
a4 ← 17
* c3 ← x1 + y2
```

Umgeschrieben

```
a3 ← x1 + y2
* b3 ← a3
a4 ← 17
* c3 ← a3 (oops!)
```

- Zugriff auf Wert 3 über Variablennamen **a**
- Nicht mehr möglich!
- Möglichkeiten
 - Führe Buch über den Wert haltende Variablen (hier **b**)
 - Mache Sicherheitskopien von Variablen (**a³** nach **t³**)
 - **Vergebe eindeutige Namen für Zuweisungen**

● Kein Überschreiben mehr möglich



Bisher: Zugriff auf Werte über Namen (von Variablen)

Eingabe-Code

```
a ← x + y
* b ← x + y
a ← 17
* c ← x + y
```

Value Numbering

```
a3 ← x1 + y2
* b3 ← x1 + y2
a4 ← 17
* c3 ← x1 + y2
```

Umgeschrieben

```
a3 ← x1 + y2
* b3 ← a3
a4 ← 17
* c3 ← a3 (oops!)
```

- Zugriff auf Wert 3 über Variablennamen **a**
- Nicht mehr möglich!
- Möglichkeiten
 - Führe Buch über den Wert haltende Variablen (hier **b**)
 - Mache Sicherheitskopien von Variablen (**a³** nach **t³**)
 - **Vergebe eindeutige Namen für Zuweisungen**

● Kein Überschreiben mehr möglich



Bisher: Zugriff auf Werte über Namen (von Variablen)

Eingabe-Code

```
a ← x + y
* b ← x + y
a ← 17
* c ← x + y
```

Value Numbering

```
a3 ← x1 + y2
* b3 ← x1 + y2
a4 ← 17
* c3 ← x1 + y2
```

Umgeschrieben

```
a3 ← x1 + y2
* b3 ← a3
a4 ← 17
* c3 ← a3 (oops!)
```

- Zugriff auf Wert 3 über Variablennamen **a**
- Nicht mehr möglich!
- Möglichkeiten
 - Führe Buch über den Wert haltende Variablen (hier **b**)
 - Mache Sicherheitskopien von Variablen (**a³** nach **t³**)
 - **Vergebe eindeutige Namen für Zuweisungen**
 - Kein Überschreiben mehr möglich



Durchnumerieren der LHS-Variablen (→ Variablenversionen)

- Wert 3 verfügbar als a_0^3
 - Hier etwas mehr Verwaltungsaufwand
 - ...echte Probleme kommen noch!
- Überführung von Binärliteratur (→ SSA-Form)

Eindeutig benannte Zuweisungsziele



Durchnumerieren der LHS-Variablen
(→ Variablenversionen)

Eingabe-Code

$$a_0 \leftarrow x_0 + y_0$$

$$* b_0 \leftarrow x_0 + y_0$$

$$a_1 \leftarrow 17$$

$$* c_0 \leftarrow x_0 + y_0$$

- Wert 3 verfügbar als a_0^3
- Hier etwas mehr Verwaltungsaufwand
- ...echte Probleme kommen noch!

Durchnumerieren von Zuweisungszielen (LHS) ist notwendig

Eindeutig benannte Zuweisungsziele



Durchnumerieren der LHS-Variablen
(\rightarrow Variablenversionen)

Eingabe-Code

```
a0 ← x0 + y0  
* b0 ← x0 + y0  
a1 ← 17  
* c0 ← x0 + y0
```

Value Numbering

```
a03 ← x01 + y02  
* b03 ← x01 + y02  
a14 ← 17  
* c03 ← x01 + y02
```

- Wert 3 verfügbar als a_0^3
- Hier etwas mehr Verwaltungsaufwand
- ...echte Probleme kommen noch!

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Eindeutig benannte Zuweisungsziele



Durchnumerieren der LHS-Variablen
(→ Variablenversionen)

Eingabe-Code

```
a0 ← x0 + y0
* b0 ← x0 + y0
a1 ← 17
* c0 ← x0 + y0
```

Value Numbering

```
a03 ← x01 + y02
* b03 ← x01 + y02
a14 ← 17
* c03 ← x01 + y02
```

Umgeschrieben

```
a03 ← x01 + y02
* b03 ← a03
a14 ← 17
* c03 ← a03
```

- Wert 3 verfügbar als a_0^3
- Hier etwas mehr Verwaltungsaufwand
- ...echte Probleme kommen noch!
 - Überschreiten von Basisblockgrenzen (→ SSA-Form)

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Eindeutig benannte Zuweisungsziele



Durchnumerieren der LHS-Variablen
(→ Variablenversionen)

Eingabe-Code

```
a0 ← x0 + y0
* b0 ← x0 + y0
a1 ← 17
* c0 ← x0 + y0
```

Value Numbering

```
a03 ← x01 + y02
* b03 ← x01 + y02
a14 ← 17
* c03 ← x01 + y02
```

Umgeschrieben

```
a03 ← x01 + y02
* b03 ← a03
a14 ← 17
* c03 ← a03
```

- Wert 3 verfügbar als a_0^3
- Hier etwas mehr Verwaltungsaufwand
- ...echte Probleme kommen noch!
 - Überschreiten von Basisblockgrenzen (→ SSA-Form)

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Eindeutig benannte Zuweisungsziele



Durchnumerieren der LHS-Variablen
(→ Variablenversionen)

Eingabe-Code

```
a0 ← x0 + y0
* b0 ← x0 + y0
a1 ← 17
* c0 ← x0 + y0
```

Value Numbering

```
a03 ← x01 + y02
* b03 ← x01 + y02
a14 ← 17
* c03 ← x01 + y02
```

Umgeschrieben

```
a03 ← x01 + y02
* b03 ← a03
a14 ← 17
* c03 ← a03
```

- Wert 3 verfügbar als a_0^3
- Hier etwas mehr Verwaltungsaufwand
- ... echte Probleme kommen noch!
 - Überschreiten von Basisblockgrenzen (→ SSA-Form)

Eindeutig benannte Zuweisungsziele



Durchnumerieren der LHS-Variablen
(→ Variablenversionen)

Eingabe-Code

```
a0 ← x0 + y0
* b0 ← x0 + y0
a1 ← 17
* c0 ← x0 + y0
```

Value Numbering

```
a03 ← x01 + y02
* b03 ← x01 + y02
a14 ← 17
* c03 ← x01 + y02
```

Umgeschrieben

```
a03 ← x01 + y02
* b03 ← a03
a14 ← 17
* c03 ← a03
```

- Wert 3 verfügbar als a_0^3
- Hier etwas mehr Verwaltungsaufwand
- ... echte Probleme kommen noch!
 - Überschreiten von Basisblockgrenzen (→ SSA-Form)

Ist VN sicher?



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

- Hash-Tabelle beginnt leer
 - Ausdrücke werden bei Durchgehen des Blocks eingetragen
 - Falls $(op, VN(x), VN(y))$ in Tabelle vorkommt
 - Ist Ausdruck mindestens einmal bereits in Block vorgekommen
 - x und y sind nicht neubelegt worden
- ⇒ Falls $(op, VN(x), VN(y))$ eine Wertnummer hat, kann er gefahrlos benutzt werden

Algorithmus

```
void Wertnumerierung(Block b, HashTabelle T, int vn) {
    for (int i = 0; i < b.GetSize(); i++)
        T.Insert(b.GetValue(i), vn);
}
```

Ist VN sicher?



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

- Hash-Tabelle beginnt leer
 - Ausdrücke werden bei Durchgehen des Blocks eingetragen
 - Falls $(op, VN(x), VN(y))$ in Tabelle vorkommt
 - Ist Ausdruck mindestens einmal bereits in Block vorgekommen
 - x und y sind nicht neubelegt worden
- ⇒ Falls $(op, VN(x), VN(y))$ eine Wertnummer hat, kann er gefahrlos benutzt werden

Algorithmus



- Hash-Tabelle beginnt leer
 - Ausdrücke werden bei Durchgehen des Blocks eingetragen
 - Falls $(op, VN(x), VN(y))$ in Tabelle vorkommt
 - Ist Ausdruck mindestens einmal bereits in Block vorgekommen
 - x und y sind nicht neubelegt worden
 - Algorithmus verwendet Wertnummern statt Variablen!
- ➔ Falls $(op, VN(x), VN(y))$ eine Wertnummer hat, kann er gefahrlos benutzt werden

Algorithmus

- beweist inkrementell, dass $(op, VN(x), VN(y))$ redundant
- modifiziert Code, aber invalidiert nicht Tabelle

Ist VN sicher?



- Hash-Tabelle beginnt leer
 - Ausdrücke werden bei Durchgehen des Blocks eingetragen
 - Falls $(op, VN(x), VN(y))$ in Tabelle vorkommt
 - Ist Ausdruck mindestens einmal bereits in Block vorgekommen
 - x und y sind nicht neubelegt worden
 - Algorithmus verwendet Wertnummern statt Variablen!
- ➔ Falls $(op, VN(x), VN(y))$ eine Wertnummer hat, kann er gefahrlos benutzt werden

Algorithmus

- beweist inkrementell, dass $(op, VN(x), VN(y))$ redundant
- modifiziert Code, aber invalidiert nicht Tabelle

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Ist VN sicher?



- Hash-Tabelle beginnt leer
 - Ausdrücke werden bei Durchgehen des Blocks eingetragen
 - Falls $(op, VN(x), VN(y))$ in Tabelle vorkommt
 - Ist Ausdruck mindestens einmal bereits in Block vorgekommen
 - x und y sind nicht neubelegt worden
 - Algorithmus verwendet **Wertnummern** statt Variablen!
- ➔ Falls $(op, VN(x), VN(y))$ eine Wertnummer hat, kann er gefahrlos benutzt werden

Algorithmus

- **beweist** inkrementell, dass $(op, VN(x), VN(y))$ redundant
- modifiziert Code, aber invalidiert nicht Tabelle

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Ist VN sicher?



- Hash-Tabelle beginnt leer
 - Ausdrücke werden bei Durchgehen des Blocks eingetragen
 - Falls $(op, VN(x), VN(y))$ in Tabelle vorkommt
 - Ist Ausdruck mindestens einmal bereits in Block vorgekommen
 - x und y sind nicht neubelegt worden
 - Algorithmus verwendet **Wertnummern** statt Variablen!
- ➔ Falls $(op, VN(x), VN(y))$ eine Wertnummer hat, kann er gefahrlos benutzt werden

Algorithmus

- **beweist** inkrementell, dass $(op, VN(x), VN(y))$ redundant
- modifiziert Code, aber invalidiert nicht Tabelle

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Ist VN sicher?



- Hash-Tabelle beginnt leer
 - Ausdrücke werden bei Durchgehen des Blocks eingetragen
 - Falls $(op, VN(x), VN(y))$ in Tabelle vorkommt
 - Ist Ausdruck mindestens einmal bereits in Block vorgekommen
 - x und y sind nicht neubelegt worden
 - Algorithmus verwendet **Wertnummern** statt Variablen!
- ➔ Falls $(op, VN(x), VN(y))$ eine Wertnummer hat, kann er gefahrlos benutzt werden

Algorithmus

- **beweist** inkrementell, dass $(op, VN(x), VN(y))$ redundant
- modifiziert Code, aber invalidiert nicht Tabelle

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Ist VN sicher?



- Hash-Tabelle beginnt leer
 - Ausdrücke werden bei Durchgehen des Blocks eingetragen
 - Falls $(op, VN(x), VN(y))$ in Tabelle vorkommt
 - Ist Ausdruck mindestens einmal bereits in Block vorgekommen
 - x und y sind nicht neubelegt worden
 - Algorithmus verwendet **Wertnummern** statt Variablen!
- ➔ Falls $(op, VN(x), VN(y))$ eine Wertnummer hat, kann er gefahrlos benutzt werden

Algorithmus

- **beweist** inkrementell, dass $(op, VN(x), VN(y))$ redundant
- **modifiziert** Code, aber **invalidiert** nicht Tabelle

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Ist VN sicher?



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

- Hash-Tabelle beginnt leer
 - Ausdrücke werden bei Durchgehen des Blocks eingetragen
 - Falls $(op, VN(x), VN(y))$ in Tabelle vorkommt
 - Ist Ausdruck mindestens einmal bereits in Block vorgekommen
 - x und y sind nicht neubelegt worden
 - Algorithmus verwendet **Wertnummern** statt Variablen!
- ➔ Falls $(op, VN(x), VN(y))$ eine Wertnummer hat, kann er gefahrlos benutzt werden

Algorithmus

- **beweist** inkrementell, dass $(op, VN(x), VN(y))$ redundant
- modifiziert Code, aber invalidiert nicht Tabelle



- Hash-Tabelle beginnt leer
 - Ausdrücke werden bei Durchgehen des Blocks eingetragen
 - Falls $(op, VN(x), VN(y))$ in Tabelle vorkommt
 - Ist Ausdruck mindestens einmal bereits in Block vorgekommen
 - x und y sind nicht neubelegt worden
 - Algorithmus verwendet **Wertnummern** statt Variablen!
- ➔ Falls $(op, VN(x), VN(y))$ eine Wertnummer hat, kann er gefahrlos benutzt werden

Algorithmus

- **beweist** inkrementell, dass $(op, VN(x), VN(y))$ redundant
- modifiziert Code, aber invalidiert nicht Tabelle

Ist VN profitabel?



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

- Wenn Wiederbenutzung billiger ist als Neuberechnung

- Übliche Annahme
- Bei Registermaschinen potentiell problematisch (*register spill*)

- Zusätzliches Constant Folding ist *immer* profitabel

- Neuberechnung braucht immer 1+ zusätzliche Register
 - Load Immediate braucht genau 1 zusätzliches Register
 - Immediate Instruktion braucht 0 zusätzliche Register
- ```
ADD R0, #8
```

- Algebraische Eigenschaften

- Entfernte Operationen sind immer nützlich ( $x + 0$ )
- Vereinfachung hängt von Zielmaschine ab ( $2*x, x+x$ )
- Kann aber leicht berücksichtigt werden

# Ist VN profitabel?



- Wenn Wiederbenutzung billiger ist als Neuberechnung
  - Übliche Annahme
    - Bei Registermaschinen potentiell problematisch (*register spill*)
- Zusätzliches Constant Folding ist *immer* profitabel
  - Neuberechnung braucht immer 1+ zusätzliche Register
  - Load Immediate braucht genau 1 zusätzliches Register
  - Immediate Instruktion braucht 0 zusätzliche Register

```
ADD R0, #8
```
- Algebraische Eigenschaften
  - Entfernte Operationen sind immer nützlich ( $x + 0$ )
  - Vereinfachung hängt von Zielmaschine ab ( $2*x, x+x$ )
  - Kann aber leicht berücksichtigt werden

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

# Ist VN profitabel?



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

- Wenn Wiederbenutzung billiger ist als Neuberechnung
  - Übliche Annahme
  - Bei Registermaschinen potentiell problematisch (*register spill*)
- Zusätzliches Constant Folding ist *immer* profitabel
  - Neuberechnung braucht immer 1+ zusätzliche Register
  - Load Immediate braucht genau 1 zusätzliches Register
  - Immediate Instruktion braucht 0 zusätzliche Register

```
ADD R0, #8
```
- Algebraische Eigenschaften
  - Entfernte Operationen sind immer nützlich ( $x + 0$ )
  - Vereinfachung hängt von Zielmaschine ab ( $2*x, x+x$ )
  - Kann aber leicht berücksichtigt werden



# Ist VN profitabel?



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

- Wenn Wiederbenutzung billiger ist als Neuberechnung
  - Übliche Annahme
  - Bei Registermaschinen potentiell problematisch (*register spill*)
- Zusätzliches Constant Folding ist *immer* profitabel
  - Neuberechnung braucht immer 1+ zusätzliche Register
  - Load Immediate braucht genau 1 zusätzliches Register
  - Immediate Instruktion braucht 0 zusätzliche Register

```
ADD R0, #8
```
- Algebraische Eigenschaften
  - Entfernte Operationen sind immer nützlich ( $x + 0$ )
  - Vereinfachung hängt von Zielmaschine ab ( $2*x, x+x$ )
  - Kann aber leicht berücksichtigt werden

# Ist VN profitabel?



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

- Wenn Wiederbenutzung billiger ist als Neuberechnung
  - Übliche Annahme
  - Bei Registermaschinen potentiell problematisch (*register spill*)
- Zusätzliches Constant Folding ist *immer* profitabel
  - Neuberechnung braucht immer 1+ zusätzliche Register
  - Load Immediate braucht genau 1 zusätzliches Register
  - Immediate Instruktion braucht 0 zusätzliche Register

```
ADD R0, #8
```
- Algebraische Eigenschaften
  - Entfernte Operationen sind immer nützlich ( $x + 0$ )
  - Vereinfachung hängt von Zielmaschine ab ( $2*x, x+x$ )
  - Kann aber leicht berücksichtigt werden

# Ist VN profitabel?



- Wenn Wiederbenutzung billiger ist als Neuberechnung
  - Übliche Annahme
  - Bei Registermaschinen potentiell problematisch (*register spill*)
- Zusätzliches Constant Folding ist *immer* profitabel
  - Neuberechnung braucht immer 1+ zusätzliche Register
  - Load Immediate braucht genau 1 zusätzliches Register
  - Immediate Instruktion braucht 0 zusätzliche Register

```
ADD R0, #8
```
- Algebraische Eigenschaften
  - Entfernte Operationen sind immer nützlich ( $x + 0$ )
  - Vereinfachung hängt von Zielmaschine ab ( $2^*x, x+x$ )
  - Kann aber leicht berücksichtigt werden

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global



- Wenn Wiederbenutzung billiger ist als Neuberechnung
  - Übliche Annahme
  - Bei Registermaschinen potentiell problematisch (*register spill*)
- Zusätzliches Constant Folding ist *immer* profitabel
  - Neuberechnung braucht immer 1+ zusätzliche Register
  - Load Immediate braucht genau 1 zusätzliches Register
  - Immediate Instruktion braucht 0 zusätzliche Register

```
ADD R0, #8
```

- Algebraische Eigenschaften

- Entfernte Operationen sind immer nützlich ( $x + 0$ )
- Vereinfachung hängt von Zielmaschine ab ( $2^*x, x+x$ )
- Kann aber leicht berücksichtigt werden



- Wenn Wiederbenutzung billiger ist als Neuberechnung
  - Übliche Annahme
  - Bei Registermaschinen potentiell problematisch (*register spill*)
- Zusätzliches Constant Folding ist *immer* profitabel
  - Neuberechnung braucht immer 1+ zusätzliche Register
  - Load Immediate braucht genau 1 zusätzliches Register
  - Immediate Instruktion braucht 0 zusätzliche Register

```
ADD R0, #8
```
- Algebraische Eigenschaften
  - Entfernte Operationen sind immer nützlich ( $x + 0$ )
  - Vereinfachung hängt von Zielmaschine ab ( $2*x, x+x$ )
  - Kann aber leicht berücksichtigt werden

# Ist VN profitabel?



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

- Wenn Wiederbenutzung billiger ist als Neuberechnung
  - Übliche Annahme
  - Bei Registermaschinen potentiell problematisch (*register spill*)
- Zusätzliches Constant Folding ist *immer* profitabel
  - Neuberechnung braucht immer 1+ zusätzliche Register
  - Load Immediate braucht genau 1 zusätzliches Register
  - Immediate Instruktion braucht 0 zusätzliche Register

```
ADD R0, #8
```
- Algebraische Eigenschaften
  - Entfernte Operationen sind immer nützlich ( $x + 0$ )
  - Vereinfachung hängt von Zielmaschine ab ( $2*x, x+x$ )
  - Kann aber leicht berücksichtigt werden



- Wenn Wiederbenutzung billiger ist als Neuberechnung
  - Übliche Annahme
  - Bei Registermaschinen potentiell problematisch (*register spill*)
- Zusätzliches Constant Folding ist *immer* profitabel
  - Neuberechnung braucht immer 1+ zusätzliche Register
  - Load Immediate braucht genau 1 zusätzliches Register
  - Immediate Instruktion braucht 0 zusätzliche Register

```
ADD R0, #8
```
- Algebraische Eigenschaften
  - Entfernte Operationen sind immer nützlich ( $x + 0$ )
  - Vereinfachung hängt von Zielmaschine ab ( $2*x, x+x$ )
  - Kann aber leicht berücksichtigt werden



- Wenn Wiederbenutzung billiger ist als Neuberechnung
  - Übliche Annahme
  - Bei Registermaschinen potentiell problematisch (*register spill*)
- Zusätzliches Constant Folding ist *immer* profitabel
  - Neuberechnung braucht immer 1+ zusätzliche Register
  - Load Immediate braucht genau 1 zusätzliches Register
  - Immediate Instruktion braucht 0 zusätzliche Register

```
ADD R0, #8
```
- Algebraische Eigenschaften
  - Entfernte Operationen sind immer nützlich ( $x + 0$ )
  - Vereinfachung hängt von Zielmaschine ab ( $2*x, x+x$ )
  - Kann aber leicht berücksichtigt werden



# Wo und wie ist VN anwendbar?



- **Potentiell anwendbar auf alle Ausdrücke eines Blocks**
- Wie passende Stellen finden?
- Linearer Durchlauf über Block
- Konstruiert Modell des dynamischen Programmzustands
- Bei jeder Operation verschiedene Möglichkeiten prüfen

## Zusammenfassung

- VN führt erschöpfende Suche durch
- Folge: Nur begrenzter Rechenaufwand je Operation akzeptabel

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

# Wo und wie ist VN anwendbar?



- Potentiell anwendbar auf alle Ausdrücke eines Blocks
- Wie passende Stellen finden?
- Linearer Durchlauf über Block
- Konstruiert Modell des dynamischen Programmzustands
- Bei jeder Operation verschiedene Möglichkeiten prüfen

## Zusammenfassung

- VN führt erschöpfende Suche durch
- Folge: Nur begrenzter Rechenaufwand je Operation akzeptabel

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

# Wo und wie ist VN anwendbar?



- Potentiell anwendbar auf alle Ausdrücke eines Blocks
- Wie passende Stellen finden?
- Linearer Durchlauf über Block
- Konstruiert Modell des dynamischen Programmzustands
- Bei jeder Operation verschiedene Möglichkeiten prüfen

## Zusammenfassung

- VN führt erschöpfende Suche durch
- Folge: Nur begrenzter Rechenaufwand je Operation akzeptabel

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

# Wo und wie ist VN anwendbar?



- Potentiell anwendbar auf alle Ausdrücke eines Blocks
- Wie passende Stellen finden?
- Linearer Durchlauf über Block
- Konstruiert Modell des dynamischen Programmzustands
- Bei jeder Operation verschiedene Möglichkeiten prüfen

## Zusammenfassung

- VN führt erschöpfende Suche durch
- Folge: Nur begrenzter Rechenaufwand je Operation akzeptabel

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

# Wo und wie ist VN anwendbar?



- Potentiell anwendbar auf alle Ausdrücke eines Blocks
- Wie passende Stellen finden?
- Linearer Durchlauf über Block
- Konstruiert Modell des dynamischen Programmzustands
- Bei jeder Operation verschiedene Möglichkeiten prüfen

## Zusammenfassung

- VN führt erschöpfende Suche durch
- Folge: Nur begrenzter Rechenaufwand je Operation akzeptabel

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

# Wo und wie ist VN anwendbar?



- Potentiell anwendbar auf alle Ausdrücke eines Blocks
- Wie passende Stellen finden?
- Linearer Durchlauf über Block
- Konstruiert Modell des dynamischen Programmzustands
- Bei jeder Operation verschiedene Möglichkeiten prüfen

## Zusammenfassung

- VN führt erschöpfende Suche durch
- Folge: Nur begrenzter Rechenaufwand je Operation akzeptabel

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

# Wo und wie ist VN anwendbar?



- Potentiell anwendbar auf alle Ausdrücke eines Blocks
- Wie passende Stellen finden?
- Linearer Durchlauf über Block
- Konstruiert Modell des dynamischen Programmzustands
- Bei jeder Operation verschiedene Möglichkeiten prüfen

## Zusammenfassung

- VN führt erschöpfende Suche durch
- Folge: Nur begrenzter Rechenaufwand je Operation akzeptabel

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

# Wo und wie ist VN anwendbar?



- Potentiell anwendbar auf alle Ausdrücke eines Blocks
- Wie passende Stellen finden?
- Linearer Durchlauf über Block
- Konstruiert Modell des dynamischen Programmzustands
- Bei jeder Operation verschiedene Möglichkeiten prüfen

## Zusammenfassung

- VN führt erschöpfende Suche durch
- Folge: Nur begrenzter Rechenaufwand je Operation akzeptabel

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global





OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

**Lokal**

Superlokal

Regional

Global

# Lokale Methoden



- **Arbeiten auf Basisblöcken (BB)**
- Für jeden Basisblock gilt
  - Alle Anweisungen werden sequentiell abgearbeitet
  - Falls eine Anweisung ausgeführt wird, werden alle Anweisungen ausgeführt.
- Können sehr genaue Analysen durchführen
- Beweisen dabei i.d.R. stärkere Aussagen als auf größeren Bereichen möglich



- **Arbeiten auf Basisblöcken (BB)**
- **Für jeden Basisblock gilt**
  - Alle Anweisungen werden sequentiell abgearbeitet
  - Falls eine Anweisung ausgeführt wird, werden alle Anweisungen ausgeführt.
- Können sehr genaue Analysen durchführen
- Beweisen dabei i.d.R. stärkere Aussagen als auf größeren Bereichen möglich



- Arbeiten auf Basisblöcken (BB)
- Für jeden Basisblock gilt
  - Alle Anweisungen werden sequentiell abgearbeitet
  - Falls eine Anweisung ausgeführt wird, werden alle Anweisungen ausgeführt.
- Können sehr genaue Analysen durchführen
- Beweisen dabei i.d.R. stärkere Aussagen als auf größeren Bereichen möglich



- Arbeiten auf Basisblöcken (BB)
- Für jeden Basisblock gilt
  - Alle Anweisungen werden sequentiell abgearbeitet
  - Falls eine Anweisung ausgeführt wird, werden alle Anweisungen ausgeführt.
- Können sehr genaue Analysen durchführen
- Beweisen dabei i.d.R. stärkere Aussagen als auf größeren Bereichen möglich



- Arbeiten auf Basisblöcken (BB)
- Für jeden Basisblock gilt
  - Alle Anweisungen werden sequentiell abgearbeitet
  - Falls eine Anweisung ausgeführt wird, werden alle Anweisungen ausgeführt.
- Können sehr genaue Analysen durchführen
- Beweisen dabei i.d.R. stärkere Aussagen als auf größeren Bereichen möglich



- Arbeiten auf Basisblöcken (BB)
- Für jeden Basisblock gilt
  - Alle Anweisungen werden sequentiell abgearbeitet
  - Falls eine Anweisung ausgeführt wird, werden alle Anweisungen ausgeführt.
- Können sehr genaue Analysen durchführen
- Beweisen dabei i.d.R. stärkere Aussagen als auf größeren Bereichen möglich



## Local Value Numbering (LVN)

- Jeweils ein Basisblock betrachtet
- Gute lokale Resultate
- Aber keine Arbeit über Blockgrenzen

## Erweitern des Redundanzbegriffes

### Redundanz über Blockgrenzen

Ein Ausdruck  $x \text{ op } y$  ist an einer Stelle  $L$  genau dann redundant, wenn er auf **jedem** Pfad vom Startknoten des CFGs zur Stelle  $L$  evaluiert worden ist und die Werte seiner Teilausdrücke  $x$  und  $y$  nicht verändert wurden.





## Local Value Numbering (LVN)

- Jeweils ein Basisblock betrachtet
- Gute lokale Resultate
- Aber keine Arbeit über Blockgrenzen

## Erweitern des Redundanzbegriffes

### Redundanz über Blockgrenzen

Ein Ausdruck  $x \text{ op } y$  ist an einer Stelle  $L$  genau dann redundant, wenn er auf **jedem** Pfad vom Startknoten des CFGs zur Stelle  $L$  evaluiert worden ist und die Werte seiner Teilausdrücke  $x$  und  $y$  nicht verändert wurden.



## Local Value Numbering (LVN)

- Jeweils ein Basisblock betrachtet
- Gute lokale Resultate
- Aber keine Arbeit über Blockgrenzen

## Erweitern des Redundanzbegriffes

### Redundanz über Blockgrenzen

Ein Ausdruck  $x \text{ op } y$  ist an einer Stelle  $L$  genau dann redundant, wenn er auf **jedem** Pfad vom Startknoten des CFGs zur Stelle  $L$  evaluiert worden ist und die Werte seiner Teilausdrücke  $x$  und  $y$  nicht verändert wurden.



## Local Value Numbering (LVN)

- Jeweils ein Basisblock betrachtet
- Gute lokale Resultate
- Aber keine Arbeit über Blockgrenzen

Erweitern des Redundanzbegriffes

### Redundanz über Blockgrenzen

Ein Ausdruck  $x \text{ op } y$  ist an einer Stelle  $L$  genau dann redundant, wenn er auf **jedem** Pfad vom Startknoten des CFGs zur Stelle  $L$  evaluiert worden ist und die Werte seiner Teilausdrücke  $x$  und  $y$  nicht verändert wurden.



## Local Value Numbering (LVN)

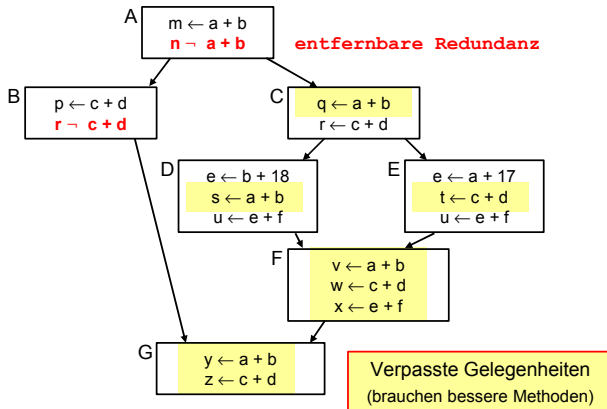
- Jeweils ein Basisblock betrachtet
- Gute lokale Resultate
- Aber keine Arbeit über Blockgrenzen

## Erweitern des Redundanzbegriffes

### Redundanz über Blockgrenzen

Ein Ausdruck  $x \text{ op } y$  ist an einer Stelle  $L$  genau dann redundant, wenn er auf **jedem** Pfad vom Startknoten des CFGs zur Stelle  $L$  evaluiert worden ist und die Werte seiner Teilausdrücke  $x$  und  $y$  nicht verändert wurden.

# Beispiel LVN



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

**Superlokal**

Regional

Global

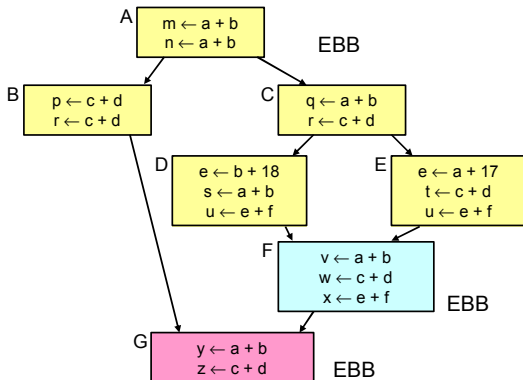
# Superlokale Methoden

# Superlokale Methoden 1

## Arbeiten auf Extended Basic Blocks (EBBs)



- EBB  $B = \{b_1, b_2, \dots, b_n\}$ , mit BBs  $b_i$
- Nur  $b_1$  darf im CFG mehrere oder keine Vorgänger haben
- Alle anderen  $b_i$  haben genau einen Vorgänger
- EBB ist Baum aus BBs mit  $b_1$  als Wurzel

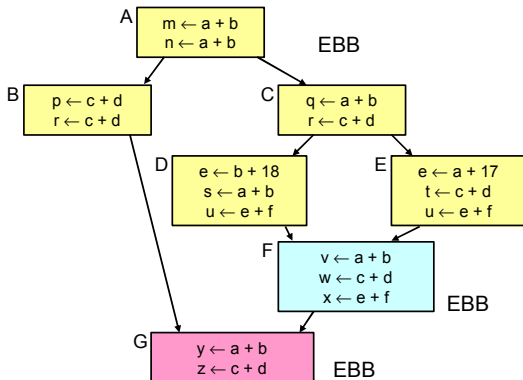


# Superlokale Methoden 1

## Arbeiten auf Extended Basic Blocks (EBBs)



- EBB  $B = \{b_1, b_2, \dots, b_n\}$ , mit BBs  $b_i$
- Nur  $b_1$  darf im CFG mehrere oder keine Vorgänger haben
- Alle anderen  $b_i$  haben genau einen Vorgänger
- EBB ist Baum aus BBs mit  $b_1$  als Wurzel



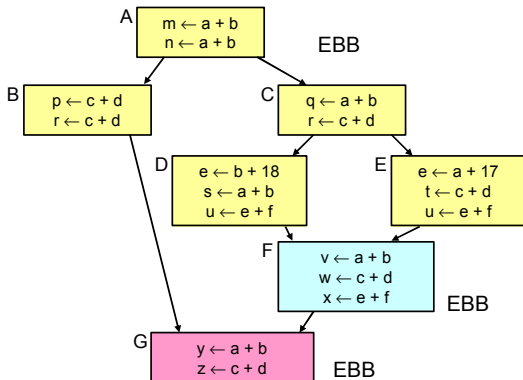


# Superlokale Methoden 1

## Arbeiten auf Extended Basic Blocks (EBBs)



- EBB  $B = \{b_1, b_2, \dots, b_n\}$ , mit BBs  $b_i$
- Nur  $b_1$  darf im CFG mehrere oder keine Vorgänger haben
- Alle anderen  $b_i$  haben genau einen Vorgänger
- EBB ist Baum aus BBs mit  $b_1$  als Wurzel

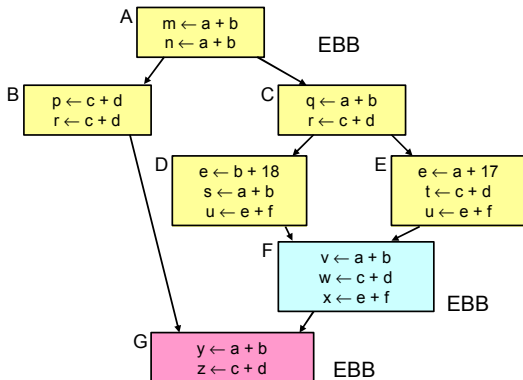


# Superlokale Methoden 1

## Arbeiten auf Extended Basic Blocks (EBBs)



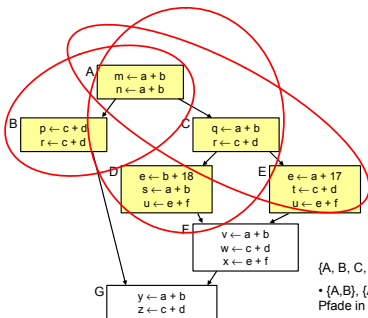
- EBB  $B = \{b_1, b_2, \dots, b_n\}$ , mit BBs  $b_i$
- Nur  $b_1$  darf im CFG mehrere oder keine Vorgänger haben
- Alle anderen  $b_i$  haben genau einen Vorgänger
- EBB ist Baum aus BBs mit  $b_1$  als Wurzel



# Superlokale Methoden 2



- Idee: Benutze **Pfade** beginnend bei  $b_1$  durch EBB wie einen BB
- Auf jedem Pfad: Genau ein Vorgänger, baue auf dessen Analysen auf



{A, B, C, D, E} ist ein EBB

- {A,B}, {A,C,D}, und {A,C,E} sind Pfade in {A, B, C, D, E}

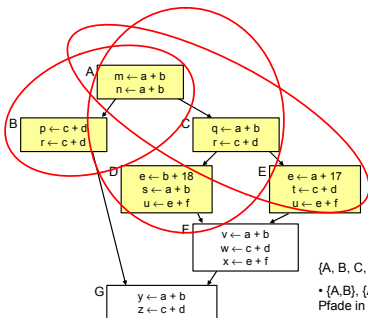
{F} und {G} sind auch EBBs

- ... haben aber nur triviale Pfade

# Superlokale Methoden 2



- Idee: Benutze **Pfade** beginnend bei  $b_1$  durch EBB wie einen BB
- Auf jedem Pfad: Genau ein Vorgänger, baue auf dessen Analysen auf



{A, B, C, D, E} ist ein EBB

- {A,B}, {A,C,D}, und {A,C,E} sind Pfade in {A, B, C, D, E}

{F} und {G} sind auch EBBs

- ... haben aber nur triviale Pfade

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

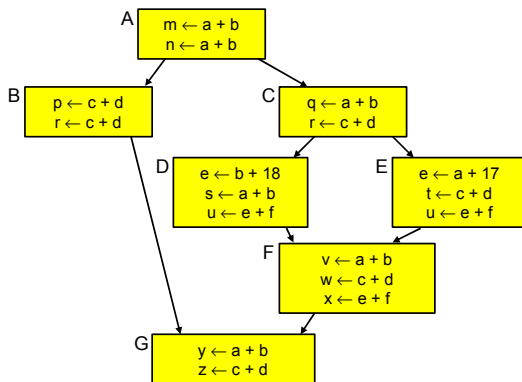
Wertnumerierung

Lokal

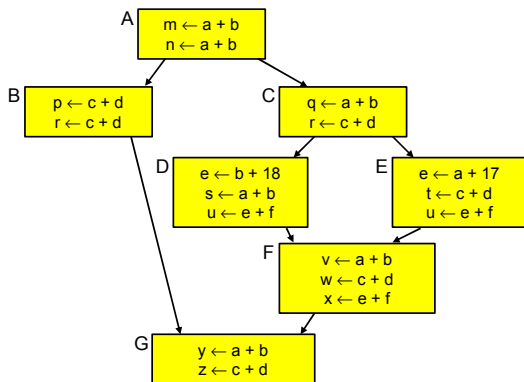
Superlokal

Regional

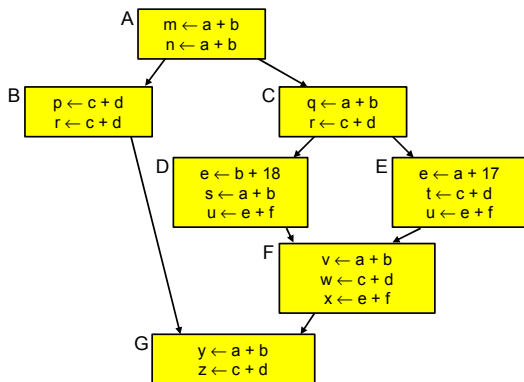
Global



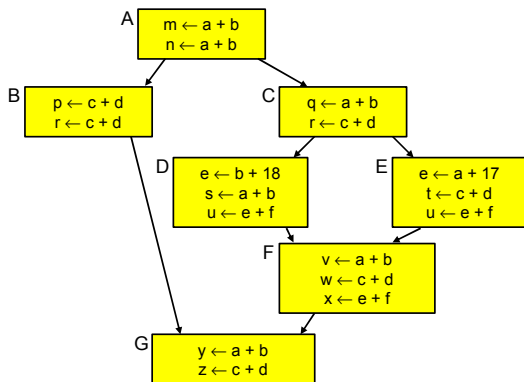
- Idee: Wende **lokale** Methode auf jeden **Pfad** an
- Hier: Bearbeite  $(A, B)$ ,  $(A, C, D)$ ,  $(A, C, E)$
- Baue auf Ergebnisse des Vorgängers auf
- **Aber:** Hilft noch nicht für F und G (woher kommt man?)



- Idee: Wende **lokale** Methode auf jeden **Pfad** an
- Hier: Bearbeite  $(A, B)$ ,  $(A, C, D)$ ,  $(A, C, E)$
- Baue auf Ergebnisse des Vorgängers auf
- **Aber:** Hilft noch nicht für F und G (woher kommt man?)



- Idee: Wende **lokale** Methode auf jeden **Pfad** an
- Hier: Bearbeite  $(A, B)$ ,  $(A, C, D)$ ,  $(A, C, E)$
- Baue auf Ergebnisse des Vorgängers auf
- **Aber:** Hilft noch nicht für F und G (woher kommt man?)



- Idee: Wende **lokale** Methode auf jeden **Pfad** an
- Hier: Bearbeite  $(A, B)$ ,  $(A, C, D)$ ,  $(A, C, E)$
- Baue auf Ergebnisse des Vorgängers auf
- **Aber**: Hilft noch nicht für F und G (woher kommt man?)





- Vermeide mehrfache Neuberechnung gleicher Daten
  - Hier A und C
- Braucht Fähigkeit, Einträge ungültig zu machen
  - Beispiel: Von (A,B) nach (A,C) müssen Daten von B entfernt werden
- Eine Realisierung:  
Symbol-Tabelle mit Geltungsbereichen
  - Siehe Kontextanalyse
  - Öffne Geltungsbereich bei Anhängen von Block an Pfad
  - Schließe Geltungsbereich bei Entfernen von Block aus Pfad



- Vermeide mehrfache Neuberechnung gleicher Daten
  - Hier A und C
- Braucht Fähigkeit, Einträge ungültig zu machen
  - Beispiel: Von (A,B) nach (A,C) müssen Daten von B entfernt werden
- Eine Realisierung:  
Symbol-Tabelle mit Geltungsbereichen
  - Siehe Kontextanalyse
  - Öffne Geltungsbereich bei Anhängen von Block an Pfad
  - Schließe Geltungsbereich bei Entfernen von Block aus Pfad



- Vermeide mehrfache Neuberechnung gleicher Daten
  - Hier A und C
- Braucht Fähigkeit, Einträge ungültig zu machen
  - Beispiel: Von (A,B) nach (A,C) müssen Daten von B entfernt werden
- Eine Realisierung:  
Symbol-Tabelle mit Geltungsbereichen
  - Siehe Kontextanalyse
  - Öffne Geltungsbereich bei Anhängen von Block an Pfad
  - Schließe Geltungsbereich bei Entfernen von Block aus Pfad



- Vermeide mehrfache Neuberechnung gleicher Daten
  - Hier A und C
- Braucht Fähigkeit, Einträge ungültig zu machen
  - Beispiel: Von (A,B) nach (A,C) müssen Daten von B entfernt werden
- Eine Realisierung:  
Symbol-Tabelle mit Geltungsbereichen
  - Siehe Kontextanalyse
  - Öffne Geltungsbereich bei Anhängen von Block an Pfad
  - Schließe Geltungsbereich bei Entfernen von Block aus Pfad



- Vermeide mehrfache Neuberechnung gleicher Daten
  - Hier A und C
- Braucht Fähigkeit, Einträge ungültig zu machen
  - Beispiel: Von (A,B) nach (A,C) müssen Daten von B entfernt werden
- Eine Realisierung:  
Symbol-Tabelle mit Geltungsbereichen
  - Siehe Kontextanalyse
  - Öffne Geltungsbereich bei Anhängen von Block an Pfad
  - Schließe Geltungsbereich bei Entfernen von Block aus Pfad



- Vermeide mehrfache Neuberechnung gleicher Daten
  - Hier A und C
- Braucht Fähigkeit, Einträge ungültig zu machen
  - Beispiel: Von (A,B) nach (A,C) müssen Daten von B entfernt werden
- Eine Realisierung:  
Symbol-Tabelle mit Geltungsbereichen
  - Siehe Kontextanalyse
  - Öffne Geltungsbereich bei Anhängen von Block an Pfad
  - Schließe Geltungsbereich bei Entfernen von Block aus Pfad



- Vermeide mehrfache Neuberechnung gleicher Daten
  - Hier A und C
- Braucht Fähigkeit, Einträge ungültig zu machen
  - Beispiel: Von (A,B) nach (A,C) müssen Daten von B entfernt werden
- Eine Realisierung:  
Symbol-Tabelle mit Geltungsbereichen
  - Siehe Kontextanalyse
  - Öffne Geltungsbereich bei Anhängen von Block an Pfad
  - Schließe Geltungsbereich bei Entfernen von Block aus Pfad



- Vermeide mehrfache Neuberechnung gleicher Daten
  - Hier A und C
- Braucht Fähigkeit, Einträge ungültig zu machen
  - Beispiel: Von (A,B) nach (A,C) müssen Daten von B entfernt werden
- Eine Realisierung:  
Symbol-Tabelle mit Geltungsbereichen
  - Siehe Kontextanalyse
  - Öffne Geltungsbereich bei Anhängen von Block an Pfad
  - Schließe Geltungsbereich bei Entfernen von Block aus Pfad





## Eingabe-Code

```
a ← x + y
* b ← x + y
 a ← 17
* c ← x + y
```

Problem: Zuordnung von Wertnummern zu Namen

- Um überschriebene Namen zu umgehen (siehe Beispiel!)
- Muß auch je Geltungsbereich aktualisiert werden

Besser

- Eindeutige Namen für jede Definition
- Nun eindeutiger Name je Wertnummer

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global



## Eingabe-Code

```
a ← x + y
* b ← x + y
a ← 17
* c ← x + y
```

## Value Numbering

```
a3 ← x1 + y2
* b3 ← x1 + y2
a4 ← 17
* c3 ← x1 + y2
```

Problem: Zuordnung von Wertnummern zu Namen

- Um überschriebene Namen zu umgehen (siehe Beispiel!)
- Muß auch je Geltungsbereich aktualisiert werden

Besser

- Eindeutige Namen für jede Definition
- Nun eindeutiger Name je Wertnummer



## Eingabe-Code

```
a ← x + y
* b ← x + y
a ← 17
* c ← x + y
```

## Value Numbering

```
a3 ← x1 + y2
* b3 ← x1 + y2
a4 ← 17
* c3 ← x1 + y2
```

## Umgeschrieben

```
a3 ← x1 + y2
* b3 ← a3
a4 ← 17
* c3 ← a3 (oops!)
```

Problem: Zuordnung von Wertnummern zu Namen

- Um überschriebene Namen zu umgehen (siehe Beispiel!)
- Muß auch je Geltungsbereich aktualisiert werden

Besser

- Eindeutige Namen für jede Definition
- Nun eindeutiger Name je Wertnummer



## Eingabe-Code

```
a ← x + y
* b ← x + y
a ← 17
* c ← x + y
```

## Value Numbering

```
a3 ← x1 + y2
* b3 ← x1 + y2
a4 ← 17
* c3 ← x1 + y2
```

## Umgeschrieben

```
a3 ← x1 + y2
* b3 ← a3
a4 ← 17
* c3 ← a3 (oops!)
```

Problem: Zuordnung von Wertnummern zu Namen

- Um überschriebene Namen zu umgehen (siehe Beispiel!)
- Muß auch je Geltungsbereich aktualisiert werden

Besser

- Eindeutige Namen für jede Definition
- Nun eindeutiger Name je Wertnummer



## Eingabe-Code

```
a ← x + y
* b ← x + y
a ← 17
* c ← x + y
```

## Value Numbering

```
a3 ← x1 + y2
* b3 ← x1 + y2
a4 ← 17
* c3 ← x1 + y2
```

## Umgeschrieben

```
a3 ← x1 + y2
* b3 ← a3
a4 ← 17
* c3 ← a3 (oops!)
```

Problem: Zuordnung von Wertnummern zu Namen

- Um überschriebene Namen zu umgehen (siehe Beispiel!)
- Muß auch je Geltungsbereich aktualisiert werden

Besser

- Eindeutige Namen für jede Definition
- Nun eindeutiger Name je Wertnummer



## Eingabe-Code

```
a ← x + y
* b ← x + y
a ← 17
* c ← x + y
```

## Value Numbering

```
a3 ← x1 + y2
* b3 ← x1 + y2
a4 ← 17
* c3 ← x1 + y2
```

## Umgeschrieben

```
a3 ← x1 + y2
* b3 ← a3
a4 ← 17
* c3 ← a3 (oops!)
```

Problem: Zuordnung von Wertnummern zu Namen

- Um überschriebene Namen zu umgehen (siehe Beispiel!)
- Muß auch je Geltungsbereich aktualisiert werden

Besser

- Eindeutige Namen für jede Definition
- Nun eindeutiger Name je Wertnummer



## Eingabe-Code

```
a ← x + y
* b ← x + y
a ← 17
* c ← x + y
```

## Value Numbering

```
a3 ← x1 + y2
* b3 ← x1 + y2
a4 ← 17
* c3 ← x1 + y2
```

## Umgeschrieben

```
a3 ← x1 + y2
* b3 ← a3
a4 ← 17
* c3 ← a3 (oops!)
```

Problem: Zuordnung von Wertnummern zu Namen

- Um überschriebene Namen zu umgehen (siehe Beispiel!)
- Muß auch je Geltungsbereich aktualisiert werden

Besser

- Eindeutige Namen für jede Definition
- Nun eindeutiger Name je Wertnummer



## Eingabe-Code

```
a ← x + y
* b ← x + y
a ← 17
* c ← x + y
```

## Value Numbering

```
a3 ← x1 + y2
* b3 ← x1 + y2
a4 ← 17
* c3 ← x1 + y2
```

## Umgeschrieben

```
a3 ← x1 + y2
* b3 ← a3
a4 ← 17
* c3 ← a3 (oops!)
```

Problem: Zuordnung von Wertnummern zu Namen

- Um überschriebene Namen zu umgehen (siehe Beispiel!)
- Muß auch je Geltungsbereich aktualisiert werden

**Besser**

- Eindeutige Namen für jede Definition
- Nun eindeutiger Name je Wertnummer





## Eingabe-Code

```
a ← x + y
* b ← x + y
a ← 17
* c ← x + y
```

## Value Numbering

```
a3 ← x1 + y2
* b3 ← x1 + y2
a4 ← 17
* c3 ← x1 + y2
```

## Umgeschrieben

```
a3 ← x1 + y2
* b3 ← a3
a4 ← 17
* c3 ← a3 (oops!)
```

Problem: Zuordnung von Wertnummern zu Namen

- Um überschriebene Namen zu umgehen (siehe Beispiel!)
- Muß auch je Geltungsbereich aktualisiert werden

**Besser**

- Eindeutige Namen für jede Definition
- Nun eindeutiger Name je Wertnummer



- Zur Compile-Zeit (also **statisch**)
- Jeder Wert wird an genau **eine eigene** Variable zugewiesen
- Jeder Operand hat somit **genau eine** Definition in BB
- Letzte Definition ist die aktuelle
- Problem: Was, wenn mehrere “letzte” Definitionen?  
(z.B. then/else-Zweige: mehrere BBs)



- Zur Compile-Zeit (also **statisch**)
- Jeder Wert wird an genau **eine eigene** Variable zugewiesen
- Jeder Operand hat somit **genau eine** Definition in BB
- Letzte Definition ist die aktuelle
- Problem: Was, wenn mehrere “letzte” Definitionen?  
(z.B. then/else-Zweige: mehrere BBs)



- Zur Compile-Zeit (also **statisch**)
- Jeder Wert wird an genau **eine eigene** Variable zugewiesen
- Jeder Operand hat somit **genau eine** Definition in BB
- Letzte Definition ist die aktuelle
- Problem: Was, wenn mehrere "letzte" Definitionen?  
(z.B. then/else-Zweige: mehrere BBs)



- Zur Compile-Zeit (also **statisch**)
- Jeder Wert wird an genau **eine eigene** Variable zugewiesen
- Jeder Operand hat somit **genau eine** Definition in BB
- Letzte Definition ist die aktuelle
- Problem: Was, wenn mehrere “letzte” Definitionen?  
(z.B. then/else-Zweige: mehrere BBs)



- Zur Compile-Zeit (also **statisch**)
- Jeder Wert wird an genau **eine eigene** Variable zugewiesen
- Jeder Operand hat somit **genau eine** Definition in BB
- Letzte Definition ist die aktuelle
- Problem: Was, wenn mehrere “letzte” Definitionen? (z.B. then/else-Zweige: mehrere BBs)



## Zusammenführen von mehreren “letzten” Definitionen

- Füge  $\phi$ -Funktionen an Zusammenführungspunkten im CFG ein
- Sogenannte *merge points*
- $\phi$ -Funktion liefert den Wert mit dem Index der Nummer der letzten genommenen Kante



Wie CFG in SSA-Form umwandeln? **Kommt noch ...**



## Zusammenführen von mehreren “letzten” Definitionen

- Füge  $\phi$ -Funktionen an Zusammenführungspunkten im CFG ein
- Sogenannte *merge points*
- $\phi$ -Funktion liefert den Wert mit dem Index der Nummer der letzten genommenen Kante



Wie CFG in SSA-Form umwandeln? **Kommt noch ...**





## Zusammenführen von mehreren “letzten” Definitionen

- Füge  $\phi$ -Funktionen an Zusammenführungspunkten im CFG ein
- Sogenannte *merge points*
- $\phi$ -Funktion liefert den Wert mit dem Index der Nummer der letzten genommenen Kante



Wie CFG in SSA-Form umwandeln? **Kommt noch ...**



## Zusammenführen von mehreren “letzten” Definitionen

- Füge  $\phi$ -Funktionen an Zusammenführungspunkten im CFG ein
- Sogenannte *merge points*
- $\phi$ -Funktion liefert den Wert mit dem Index der Nummer der letzten genommenen Kante



Wie CFG in SSA-Form umwandeln? **Kommt noch ...**



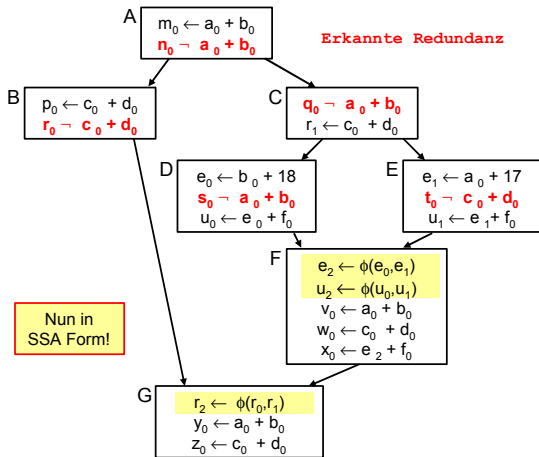
## Zusammenführen von mehreren “letzten” Definitionen

- Füge  $\phi$ -Funktionen an Zusammenführungspunkten im CFG ein
- Sogenannte *merge points*
- $\phi$ -Funktion liefert den Wert mit dem Index der Nummer der letzten genommenen Kante



Wie CFG in SSA-Form umwandeln? **Kommt noch ...**

# Beispiel Superlokale VN 1



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global



Mehr Redundanzen erkannt, aber nicht alle

- F und G eigene EBBs, hier  $a + b$  unbekannt
- $e + f$  berechnet in allen Vorgängern von F, aber mit unterschiedlichen Werten

Erweiterung der Anwendungsbereiche

- VN: Leicht von BB auf EBB erweiterbar
- Klappt aber nicht immer. z.B.

[http://www.dcc.umd.edu/~kostas/papers/ebbs.pdf](#)



Mehr Redundanzen erkannt, aber nicht alle

- F und G eigene EBBs, hier  $a + b$  unbekannt
- $e + f$  berechnet in allen Vorgängern von F, aber mit unterschiedlichen Werten

Erweiterung der Anwendungsbereiche

- VN: Leicht von BB auf EBB erweiterbar
- Klappt aber nicht immer, z.B.



Mehr Redundanzen erkannt, aber nicht alle

- F und G eigene EBBs, hier  $a + b$  unbekannt
- $e + f$  berechnet in allen Vorgängern von F, aber mit unterschiedlichen Werten

Erweiterung der Anwendungsbereiche

- VN: Leicht von BB auf EBB erweiterbar
- Klappt aber nicht immer, z.B.



Mehr Redundanzen erkannt, aber nicht alle

- F und G eigene EBBs, hier  $a + b$  unbekannt
- $e + f$  berechnet in allen Vorgängern von F, aber mit unterschiedlichen Werten

Erweiterung der Anwendungsbereiche

- VN: Leicht von BB auf EBB erweiterbar
- Klappt aber nicht immer. z.B.
  - Verändern bereits bearbeiteter Blöcke





Mehr Redundanzen erkannt, aber nicht alle

- F und G eigene EBBs, hier  $a + b$  unbekannt
- $e + f$  berechnet in allen Vorgängern von F, aber mit unterschiedlichen Werten

Erweiterung der Anwendungsbereiche

- VN: Leicht von BB auf EBB erweiterbar
- Klappt aber nicht immer. z.B.
  - Verändern bereits bearbeiteter Blöcke



Mehr Redundanzen erkannt, aber nicht alle

- F und G eigene EBBs, hier  $a + b$  unbekannt
- $e + f$  berechnet in allen Vorgängern von F, aber mit unterschiedlichen Werten

Erweiterung der Anwendungsbereiche

- VN: Leicht von BB auf EBB erweiterbar
- Klappt aber nicht immer. z.B.
  - Verändern bereits bearbeiteter Blöcke



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

**Regional**

Global

# Regionale Methoden

# Noch größere Anwendungsbereiche 1



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

**Regional**

Global

- Superlokale VN verwirft gesamte Tabelle bei Merge Point
- Nächste Verfeinerung: Über Merge Points hinweg arbeiten

➔ **Regionaler** Anwendungsbereich

- Größer als superlokal
- Kleiner als gesamte Prozedur

# Noch größere Anwendungsbereiche 1



- Superlokale VN verwirft gesamte Tabelle bei Merge Point
- Nächste Verfeinerung: Über Merge Points hinweg arbeiten

## ➔ Regionaler Anwendungsbereich

- Größer als superlokal
- Kleiner als gesamte Prozedur

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

**Regional**

Global

# Noch größere Anwendungsbereiche 1



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

**Regional**

Global

- Superlokale VN verwirft gesamte Tabelle bei Merge Point
- Nächste Verfeinerung: Über Merge Points hinweg arbeiten

➔ **Regionaler** Anwendungsbereich

- Größer als superlokal
- Kleiner als gesamte Prozedur

# Noch größere Anwendungsbereiche 1



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

**Regional**

Global

- Superlokale VN verwirft gesamte Tabelle bei Merge Point
- Nächste Verfeinerung: Über Merge Points hinweg arbeiten

## ➔ **Regionaler** Anwendungsbereich

- Größer als superlokal
- Kleiner als gesamte Prozedur



- Superlokale VN verwirft gesamte Tabelle bei Merge Point
- Nächste Verfeinerung: Über Merge Points hinweg arbeiten

## ➔ **Regionaler** Anwendungsbereich

- Größer als superlokal
- Kleiner als gesamte Prozedur





- Superlokale VN verwirft gesamte Tabelle bei Merge Point
- Nächste Verfeinerung: Über Merge Points hinweg arbeiten

## ➔ **Regionaler** Anwendungsbereich

- Größer als superlokal
- Kleiner als gesamte Prozedur



## Überlegungen am Beispiel von F: Zusammenführung von D und E

- Keine Beschränkung auf einzelne der Tabellen D oder E möglich
  - Kontrollfluß könnte anderen Pfad nehmen, dann Ausdruck nicht redundant
- Zusammenfügen der Tabellen von D und E
  - Welche Einträge streichen? (z.B.  $b+18$  und  $a+17$ )
    - *→ nicht möglich, wenn unterschiedliche Pfade*
  - Gleiche Ausdrücke in beiden Pfaden
    - *→ nicht möglich, wenn unterschiedliche Wertnumerierung*
  - Kompliziert und rechenintensiv



## Überlegungen am Beispiel von F: Zusammenführung von D und E

- Keine Beschränkung auf einzelne der Tabellen D oder E möglich
  - Kontrollfluß könnte anderen Pfad nehmen, dann Ausdruck nicht redundant
- Zusammenfügen der Tabellen von D und E
  - Welche Einträge streichen? (z.B.  $b+18$  und  $a+17$ )
  - Welche Ausdrücke sind redundant?
  - Gleiche Ausdrücke in beiden Pfaden
  - Kompliziert und rechenintensiv



## Überlegungen am Beispiel von F: Zusammenführung von D und E

- Keine Beschränkung auf einzelne der Tabellen D oder E möglich
  - Kontrollfluß könnte anderen Pfad nehmen, dann Ausdruck nicht redundant
- Zusammenfügen der Tabellen von D und E
  - Welche Einträge streichen? (z.B.  $b+18$  und  $a+17$ )
    - Liegen auf unterschiedlichen Pfaden
  - Gleiche Ausdrücke in beiden Pfaden
    - Aber mit unterschiedlichen Wertnummern ( $e+f$ )
  - **Kompliziert** und rechenintensiv



## Überlegungen am Beispiel von F: Zusammenführung von D und E

- Keine Beschränkung auf einzelne der Tabellen D oder E möglich
  - Kontrollfluß könnte anderen Pfad nehmen, dann Ausdruck nicht redundant
- Zusammenfügen der Tabellen von D und E
  - Welche Einträge streichen? (z.B. **b+18** und **a+17**)
    - Liegen auf unterschiedlichen Pfaden
    - Gleiche Ausdrücke in beiden Pfaden
      - Aber mit unterschiedlichen Wertnummern (**e+£**)
    - **Kompliziert** und rechenintensiv



## Überlegungen am Beispiel von F: Zusammenführung von D und E

- Keine Beschränkung auf einzelne der Tabellen D oder E möglich
  - Kontrollfluß könnte anderen Pfad nehmen, dann Ausdruck nicht redundant
- Zusammenfügen der Tabellen von D und E
  - Welche Einträge streichen? (z.B. **b+18** und **a+17**)
    - Liegen auf unterschiedlichen Pfaden
  - Gleiche Ausdrücke in beiden Pfaden
    - Aber mit unterschiedlichen Wertnummern (**e+£**)
  - **Kompliziert** und rechenintensiv



## Überlegungen am Beispiel von F: Zusammenführung von D und E

- Keine Beschränkung auf einzelne der Tabellen D oder E möglich
  - Kontrollfluß könnte anderen Pfad nehmen, dann Ausdruck nicht redundant
- Zusammenfügen der Tabellen von D und E
  - Welche Einträge streichen? (z.B. **b+18** und **a+17**)
    - Liegen auf unterschiedlichen Pfaden
  - Gleiche Ausdrücke in beiden Pfaden
    - Aber mit unterschiedlichen Wertnummern (**e+f**)
  - **Kompliziert** und rechenintensiv



## Überlegungen am Beispiel von F: Zusammenführung von D und E

- Keine Beschränkung auf einzelne der Tabellen D oder E möglich
  - Kontrollfluß könnte anderen Pfad nehmen, dann Ausdruck nicht redundant
- Zusammenfügen der Tabellen von D und E
  - Welche Einträge streichen? (z.B. **b+18** und **a+17**)
    - Liegen auf unterschiedlichen Pfaden
  - Gleiche Ausdrücke in beiden Pfaden
    - Aber mit unterschiedlichen Wertnummern (**e+f**)
  - Kompliziert und rechenintensiv





## Überlegungen am Beispiel von F: Zusammenführung von D und E

- Keine Beschränkung auf einzelne der Tabellen D oder E möglich
  - Kontrollfluß könnte anderen Pfad nehmen, dann Ausdruck nicht redundant
- Zusammenfügen der Tabellen von D und E
  - Welche Einträge streichen? (z.B. **b+18** und **a+17**)
    - Liegen auf unterschiedlichen Pfaden
  - Gleiche Ausdrücke in beiden Pfaden
    - Aber mit unterschiedlichen Wertnummern (**e+f**)
  - **Kompliziert** und rechenintensiv

# Noch größere Anwendungsbereiche 3



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

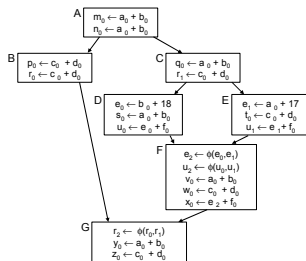
Superlokal

Regional

Global

Anderer Ansatz: Beschränke Tabelle auf Fakten, die **unabhängig** vom konkreten Pfad gelten

- Beide Pfade zu F haben gleichen **Präfix** (A,C)
- Alle Operationen in A und C sind **immer** vor Erreichen von F ausgeführt worden
- Letzter "sicherer" Stand ist also Tabelle nach C
- Als Ausgangspunkt für F benutzen



# Noch größere Anwendungsbereiche 3



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

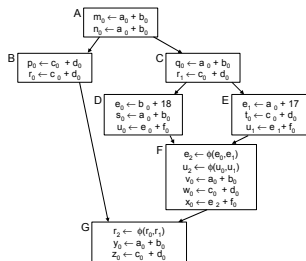
Superlokal

Regional

Global

Anderer Ansatz: Beschränke Tabelle auf Fakten, die **unabhängig** vom konkreten Pfad gelten

- Beide Pfade zu F haben gleichen **Präfix** (A,C)
- Alle Operationen in A und C sind **immer** vor Erreichen von F ausgeführt worden
- Letzter "sicherer" Stand ist also Tabelle nach C
- Als Ausgangspunkt für F benutzen



# Noch größere Anwendungsbereiche 3



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

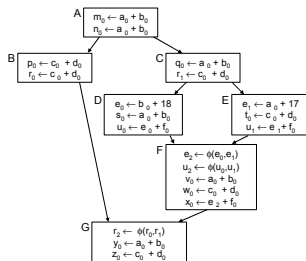
Superlokal

Regional

Global

Anderer Ansatz: Beschränke Tabelle auf Fakten, die **unabhängig** vom konkreten Pfad gelten

- Beide Pfade zu F haben gleichen **Präfix** (A,C)
- Alle Operationen in A und C sind **immer** vor Erreichen von F ausgeführt worden
- Letzter "sicherer" Stand ist also Tabelle nach C
- Als Ausgangspunkt für F benutzen



# Noch größere Anwendungsbereiche 3



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

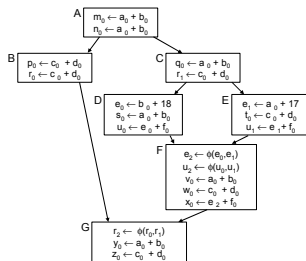
Superlokal

Regional

Global

Anderer Ansatz: Beschränke Tabelle auf Fakten, die **unabhängig** vom konkreten Pfad gelten

- Beide Pfade zu F haben gleichen **Präfix** (A,C)
- Alle Operationen in A und C sind **immer** vor Erreichen von F ausgeführt worden
- Letzter "sicherer" Stand ist also Tabelle nach C
- Als Ausgangspunkt für F benutzen



# Noch größere Anwendungsbereiche 3



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

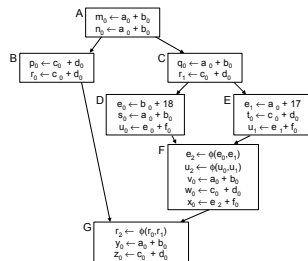
Superlokal

Regional

Global

Anderer Ansatz: Beschränke Tabelle auf Fakten, die **unabhängig** vom konkreten Pfad gelten

- Beide Pfade zu F haben gleichen **Präfix** (A,C)
- Alle Operationen in A und C sind **immer** vor Erreichen von F ausgeführt worden
- Letzter "sicherer" Stand ist also Tabelle nach C
- Als Ausgangspunkt für F benutzen





OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

**Regional**

Global

## Wie mit Zuweisungen in D und E umgehen?

- SSA-Form umgeht Verlorengehen von Werten

→ Erzeugen eines Einheitswertes

→ D und E-Werte werden als lokale Variable

Abstrahiert in globalen Werten

→ Abstraktion in globalen Werten



Wie mit Zuweisungen in D und E umgehen?

- SSA-Form umgeht Verlorengehen von Werten
  - Eigene Variable für jeden Wert
  - D und E können **zusätzliche** Werte erzeugen
  - Aber können **keine** alten Werte streichen
  - Konflikte werden über  $\phi$ -Funktionen aufgelöst





Wie mit Zuweisungen in D und E umgehen?

- SSA-Form umgeht Verlorengehen von Werten
  - Eigene Variable für jeden Wert
  - D und E können **zusätzliche** Werte erzeugen
  - Aber können **keine** alten Werte streichen
  - Konflikte werden über  $\phi$ -Funktionen aufgelöst



Wie mit Zuweisungen in D und E umgehen?

- SSA-Form umgeht Verlorengehen von Werten
  - Eigene Variable für jeden Wert
  - D und E können **zusätzliche** Werte erzeugen
  - Aber können **keine** alten Werte streichen
  - Konflikte werden über  $\phi$ -Funktionen aufgelöst



Wie mit Zuweisungen in D und E umgehen?

- SSA-Form umgeht Verlorengehen von Werten
  - Eigene Variable für jeden Wert
  - D und E können **zusätzliche** Werte erzeugen
  - Aber können **keine** alten Werte streichen
  - Konflikte werden über  $\phi$ -Funktionen aufgelöst



Wie mit Zuweisungen in D und E umgehen?

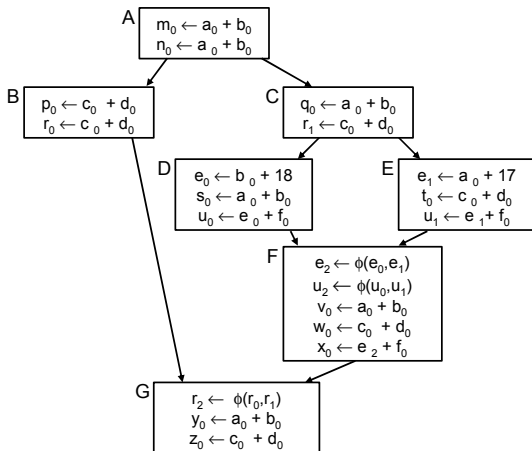
- SSA-Form umgeht Verlorengehen von Werten
  - Eigene Variable für jeden Wert
  - D und E können **zusätzliche** Werte erzeugen
  - Aber können **keine** alten Werte streichen
  - Konflikte werden über  $\phi$ -Funktionen aufgelöst

# Noch größere Anwendungsbereiche 5



Effekt der Vorgehensweise: Verwende C als Ausgangspunkt für F

- Erkennt jetzt Redundanz von  $a_0 + b_0$  und  $c_0 + d_0$  in F
- Verpasst aber  $e_2 + f_0$ , da zwischen C und F berechnet



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

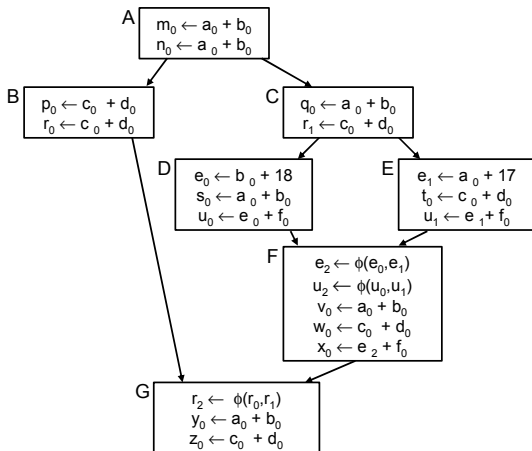


# Noch größere Anwendungsbereiche 5



Effekt der Vorgehensweise: Verwende C als Ausgangspunkt für F

- Erkennt jetzt Redundanz von  $a_0 + b_0$  und  $c_0 + d_0$  in F
- Verpasst aber  $e_2 + f_0$ , da zwischen C und F berechnet



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

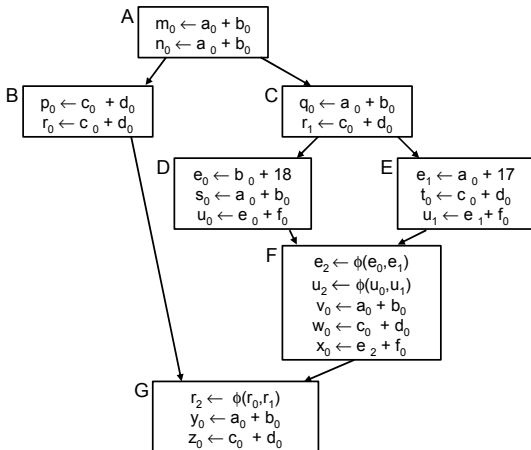


# Noch größere Anwendungsbereiche 5



Effekt der Vorgehensweise: Verwende C als Ausgangspunkt für F

- Erkennt jetzt Redundanz von  $a_0 + b_0$  und  $c_0 + d_0$  in F
- Verpasst aber  $e_2 + f_0$ , da **zwischen** C und F berechnet



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global



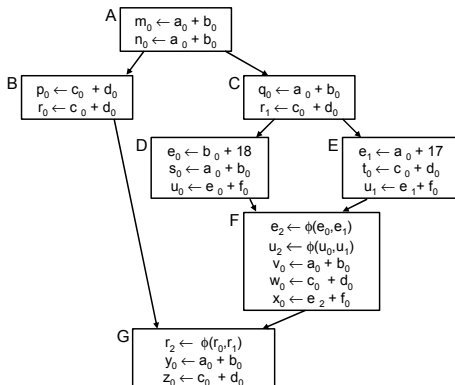
# Noch größere Anwendungsbereiche 6



Verfahren benötigt: Letzten gemeinsamen Vorfahren über alle Pfade zu einem Block

Benutze Tabelle von X bei Eintritt in Y:  $X \leftarrow Y$

- $- \leftarrow A$
- $A \leftarrow B$
- $A \leftarrow C$
- $C \leftarrow D$
- $C \leftarrow E$
- $C \leftarrow F$
- $A \leftarrow G$



➔ X muß Y dominieren

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global





## Dominanz

$x$  **dominiert**  $y$  genau dann, falls jeder Pfad vom Eingangsknoten des CFGs zum Knoten  $y$  den Knoten  $x$  enthält. Geschrieben als  $x \gg y$ .

Gilt immer:  $x \gg x$

## Strikte Dominanz

$x$  **dominiert**  $y$  **strikt**, falls  $x \gg y$  und  $x \neq y$ .  
Geschrieben als  $x \ggg y$ .



## Dominanz

$x$  **dominiert**  $y$  genau dann, falls jeder Pfad vom Eingangsknoten des CFGs zum Knoten  $y$  den Knoten  $x$  enthält. Geschrieben als  $x \gg y$ .

Gilt immer:  $x \gg x$

## Strikte Dominanz

$x$  **dominiert**  $y$  **strikt**, falls  $x \gg y$  und  $x \neq y$ .  
Geschrieben als  $x \ggg y$ .



## Dominanz

$x$  **dominiert**  $y$  genau dann, falls jeder Pfad vom Eingangsknoten des CFGs zum Knoten  $y$  den Knoten  $x$  enthält. Geschrieben als  $x \gg y$ .

Gilt immer:  $x \gg x$

## Strikte Dominanz

$x$  **dominiert**  $y$  **strikt**, falls  $x \gg y$  und  $x \neq y$ .  
Geschrieben als  $x \ggg y$ .



## Dominatoren

$\text{DOM}(y) = \{x \in \text{CFG} \mid x \gg y\}$  ist die Menge der **Dominatoren** von  $y$ .

Unmittelbarer Dominator (*immediate dominator*)

$\text{IDOM}(y)$  ist der im CFG  $y$  am **nächsten** gelegene Dominator aus  $\text{DOM}(y)$ .



## Dominatoren

$\text{DOM}(y) = \{x \in \text{CFG} \mid x \gg y\}$  ist die Menge der **Dominatoren** von  $y$ .

## Unmittelbarer Dominator (*immediate dominator*)

$\text{IDOM}(y)$  ist der im CFG  $y$  am **nächsten** gelegene Dominator aus  $\text{DOM}(y)$ .



- **Sehr nützliche Relation**

- Finden von Schleifen
- Umwandlung in SSA-Form
- Zielauswahl für Code-Bewegung

➔ **Berechnung** der Relation: Kommt noch ...



- Sehr nützliche Relation
  - Finden von Schleifen
  - Umwandlung in SSA-Form
  - Zielauswahl für Code-Bewegung

↳ **Berechnung** der Relation: Kommt noch ...



- Sehr nützliche Relation
  - Finden von Schleifen
  - Umwandlung in SSA-Form
  - Zielauswahl für Code-Bewegung

➔ **Berechnung** der Relation: Kommt noch ...





- Sehr nützliche Relation
  - Finden von Schleifen
  - Umwandlung in SSA-Form
  - Zielauswahl für Code-Bewegung

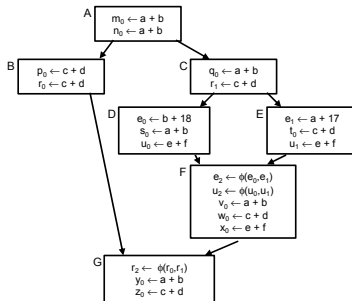
➔ **Berechnung** der Relation: Kommt noch ...



- Sehr nützliche Relation
  - Finden von Schleifen
  - Umwandlung in SSA-Form
  - Zielauswahl für Code-Bewegung

### Dominatormengen

| Block | Dom    | IDom |
|-------|--------|------|
| A     | A      | -    |
| B     | A,B    | A    |
| C     | A,C    | A    |
| D     | A,C, D | C    |
| E     | A,C, E | C    |
| F     | A,C,F  | C    |
| G     | A,G    | A    |



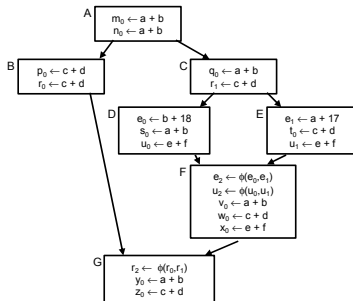
➔ Berechnung der Relation: Kommt noch ...



- Sehr nützliche Relation
  - Finden von Schleifen
  - Umwandlung in SSA-Form
  - Zielauswahl für Code-Bewegung

### Dominatormengen

| Block | Dom    | IDom |
|-------|--------|------|
| A     | A      | -    |
| B     | A,B    | A    |
| C     | A,C    | A    |
| D     | A,C, D | C    |
| E     | A,C, E | C    |
| F     | A,C,F  | C    |
| G     | A,G    | A    |



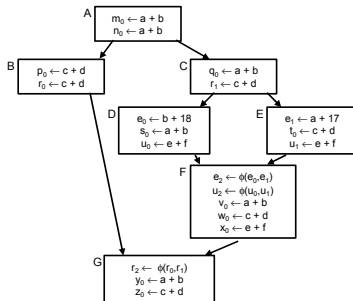
➔ Berechnung der Relation: Kommt noch ...



- Sehr nützliche Relation
  - Finden von Schleifen
  - Umwandlung in SSA-Form
  - Zielauswahl für Code-Bewegung

### Dominatormengen

| Block | Dom    | IDom |
|-------|--------|------|
| A     | A      | -    |
| B     | A,B    | A    |
| C     | A,C    | A    |
| D     | A,C, D | C    |
| E     | A,C, E | C    |
| F     | A,C,F  | C    |
| G     | A,G    | A    |



➔ **Berechnung** der Relation: Kommt noch ...



- Geben sei ein Block  $b$
- Jeder Block  $d$  in  $\text{DOM}(b) - \{b\}$  wurde **vor**  $b$  ausgeführt
- Die VN-Tabelle jedes  $d$  **könnte** zur Vorbelegung von  $b$  verwendet werden
- Welche wäre die beste Wahl?
- Die von Block  $e = \text{IDOM}(b)$  !
  - $e$  wird von allen anderen Blöcken aus  $\text{DOM}(b) - \{b\}$  dominiert
  - Kann all deren Informationen verwenden
  - Hat damit die meisten Informationen

➔ Dominator VN Technique (DVNT oder kurz DVN)



- Geben sei ein Block  $b$
- Jeder Block  $d$  in  $\text{DOM}(b) - \{b\}$  wurde **vor**  $b$  ausgeführt
- Die VN-Tabelle jedes  $d$  **könnte** zur Vorbelegung von  $b$  verwendet werden
- Welche wäre die beste Wahl?
- Die von Block  $e = \text{IDOM}(b)$  !
  - $e$  wird von allen anderen Blöcken aus  $\text{DOM}(b) - \{b\}$  dominiert
  - Kann all deren Informationen verwenden
  - Hat damit die meisten Informationen

➔ Dominator VN Technique (DVNT oder kurz DVN)



- Geben sei ein Block  $b$
- Jeder Block  $d$  in  $\text{DOM}(b) - \{b\}$  wurde **vor**  $b$  ausgeführt
- Die VN-Tabelle jedes  $d$  **könnte** zur Vorbelegung von  $b$  verwendet werden
- Welche wäre die beste Wahl?
- Die von Block  $e = \text{IDOM}(b)$  !
  - $e$  wird von allen anderen Blöcken aus  $\text{DOM}(b) - \{b\}$  dominiert
  - Kann all deren Informationen verwenden
  - Hat damit die meisten Informationen

➔ Dominator VN Technique (DVNT oder kurz DVN)



- Geben sei ein Block  $b$
- Jeder Block  $d$  in  $\text{DOM}(b) - \{b\}$  wurde **vor**  $b$  ausgeführt
- Die VN-Tabelle jedes  $d$  **könnte** zur Vorbelegung von  $b$  verwendet werden
- Welche wäre die beste Wahl?
- Die von Block  $e = \text{IDOM}(b)$  !
  - $e$  wird von allen anderen Blöcken aus  $\text{DOM}(b) - \{b\}$  dominiert
  - Kann all deren Informationen verwenden
  - Hat damit die meisten Informationen

➔ Dominator VN Technique (DVNT oder kurz DVN)





- Geben sei ein Block  $b$
- Jeder Block  $d$  in  $\text{DOM}(b) - \{b\}$  wurde **vor**  $b$  ausgeführt
- Die VN-Tabelle jedes  $d$  **könnte** zur Vorbelegung von  $b$  verwendet werden
- Welche wäre die beste Wahl?
- Die von Block  $e = \text{IDOM}(b)$  !
  - $e$  wird von allen anderen Blöcken aus  $\text{DOM}(b) - \{b\}$  dominiert
  - Kann all deren Informationen verwenden
  - Hat damit die meisten Informationen

➔ Dominator VN Technique (DVNT oder kurz DVN)



- Geben sei ein Block  $b$
- Jeder Block  $d$  in  $\text{DOM}(b) - \{b\}$  wurde **vor**  $b$  ausgeführt
- Die VN-Tabelle jedes  $d$  **könnte** zur Vorbelegung von  $b$  verwendet werden
- Welche wäre die beste Wahl?
- Die von Block  $e = \text{IDOM}(b)$  !
  - $e$  wird von allen anderen Blöcken aus  $\text{DOM}(b) - \{b\}$  dominiert
  - Kann all deren Informationen verwenden
  - Hat damit die meisten Informationen

➔ Dominator VN Technique (DVNT oder kurz DVN)



- Geben sei ein Block  $b$
- Jeder Block  $d$  in  $\text{DOM}(b) - \{b\}$  wurde **vor**  $b$  ausgeführt
- Die VN-Tabelle jedes  $d$  **könnte** zur Vorbelegung von  $b$  verwendet werden
- Welche wäre die beste Wahl?
- Die von Block  $e = \text{IDOM}(b)$  !
  - $e$  wird von allen anderen Blöcken aus  $\text{DOM}(b) - \{b\}$  dominiert
  - Kann all deren Informationen verwenden
  - Hat damit die meisten Informationen

➔ Dominator VN Technique (DVNT oder kurz DVN)



- Geben sei ein Block  $b$
- Jeder Block  $d$  in  $\text{DOM}(b) - \{b\}$  wurde **vor**  $b$  ausgeführt
- Die VN-Tabelle jedes  $d$  **könnte** zur Vorbelegung von  $b$  verwendet werden
- Welche wäre die beste Wahl?
- Die von Block  $e = \text{IDOM}(b)$  !
  - $e$  wird von allen anderen Blöcken aus  $\text{DOM}(b) - \{b\}$  dominiert
  - Kann all deren Informationen verwenden
  - Hat damit die meisten Informationen

➔ Dominator VN Technique (DVNT oder kurz DVN)



- Geben sei ein Block  $b$
- Jeder Block  $d$  in  $\text{DOM}(b) - \{b\}$  wurde **vor**  $b$  ausgeführt
- Die VN-Tabelle jedes  $d$  **könnte** zur Vorbelegung von  $b$  verwendet werden
- Welche wäre die beste Wahl?
- Die von Block  $e = \text{IDOM}(b)$  !
  - $e$  wird von allen anderen Blöcken aus  $\text{DOM}(b) - \{b\}$  dominiert
  - Kann all deren Informationen verwenden
  - Hat damit die meisten Informationen

➔ Dominator VN Technique (DVNT oder kurz DVN)



- Geben sei ein Block  $b$
- Jeder Block  $d$  in  $\text{DOM}(b) - \{b\}$  wurde **vor**  $b$  ausgeführt
- Die VN-Tabelle jedes  $d$  **könnte** zur Vorbelegung von  $b$  verwendet werden
- Welche wäre die beste Wahl?
- Die von Block  $e = \text{IDOM}(b)$  !
  - $e$  wird von allen anderen Blöcken aus  $\text{DOM}(b) - \{b\}$  dominiert
  - Kann all deren Informationen verwenden
  - Hat damit die meisten Informationen

➔ Dominator VN Technique (DVNT oder kurz DVN)



## Regionaler Algorithmus

- Rechne superlokalen Algorithmus auf EBBs, verwendet dabei
  - Verschachtelte Hash-Tabellen für Geltungsbereiche
  - SSA-Form
- **Neu:** Initialisiere Tabelle für Knoten  $x$ 
  - Mit Tabelle von  $IDOM(x)$
  - Trage so Wissen über Merge Points hinweg
- Wie vorher möglich:
  - Constant folding
  - Ausnutzung algebraischer Eigenschaften

➡ Größerer Anwendungsbereich sollte zu besseren Ergebnissen führen



## Regionaler Algorithmus

- Rechne superlokalen Algorithmus auf EBBs, verwendet dabei
  - Verschachtelte Hash-Tabellen für Geltungsbereiche
  - SSA-Form
- **Neu:** Initialisiere Tabelle für Knoten  $x$ 
  - Mit Tabelle von  $IDOM(x)$
  - Trage so Wissen über Merge Points hinweg
- Wie vorher möglich:
  - Constant folding
  - Ausnutzung algebraischer Eigenschaften

➡ Größerer Anwendungsbereich sollte zu besseren Ergebnissen führen





## Regionaler Algorithmus

- Rechne superlokalen Algorithmus auf EBBs, verwendet dabei
  - Verschachtelte Hash-Tabellen für Geltungsbereiche
  - SSA-Form
- **Neu:** Initialisiere Tabelle für Knoten  $x$ 
  - Mit Tabelle von  $IDOM(x)$
  - Trage so Wissen über Merge Points hinweg
- Wie vorher möglich:
  - Constant folding
  - Ausnutzung algebraischer Eigenschaften

➡ Größerer Anwendungsbereich sollte zu besseren Ergebnissen führen



## Regionaler Algorithmus

- Rechne superlokalen Algorithmus auf EBBs, verwendet dabei
  - Verschachtelte Hash-Tabellen für Geltungsbereiche
  - SSA-Form
- **Neu:** Initialisiere Tabelle für Knoten  $x$ 
  - Mit Tabelle von  $IDOM(x)$
  - Trage so Wissen über Merge Points hinweg
- Wie vorher möglich:
  - Constant folding
  - Ausnutzung algebraischer Eigenschaften

➡ Größerer Anwendungsbereich sollte zu besseren Ergebnissen führen



## Regionaler Algorithmus

- Rechne superlokalen Algorithmus auf EBBs, verwendet dabei
  - Verschachtelte Hash-Tabellen für Geltungsbereiche
  - SSA-Form
- **Neu:** Initialisiere Tabelle für Knoten  $x$ 
  - Mit Tabelle von  $IDOM(x)$
  - Trage so Wissen über Merge Points hinweg
- Wie vorher möglich:
  - Constant folding
  - Ausnutzung algebraischer Eigenschaften

➡ Größerer Anwendungsbereich sollte zu besseren Ergebnissen führen



## Regionaler Algorithmus

- Rechne superlokalen Algorithmus auf EBBs, verwendet dabei
  - Verschachtelte Hash-Tabellen für Geltungsbereiche
  - SSA-Form
- **Neu:** Initialisiere Tabelle für Knoten  $x$ 
  - Mit Tabelle von  $IDOM(x)$
  - Trage so Wissen über Merge Points hinweg
- Wie vorher möglich:
  - Constant folding
  - Ausnutzung algebraischer Eigenschaften

➡ Größerer Anwendungsbereich sollte zu besseren Ergebnissen führen



## Regionaler Algorithmus

- Rechne superlokalen Algorithmus auf EBBs, verwendet dabei
  - Verschachtelte Hash-Tabellen für Geltungsbereiche
  - SSA-Form
- **Neu:** Initialisiere Tabelle für Knoten  $x$ 
  - Mit Tabelle von  $IDOM(x)$
  - Trage so Wissen über Merge Points hinweg
- Wie vorher möglich:
  - Constant folding
  - Ausnutzung algebraischer Eigenschaften

➡ Größerer Anwendungsbereich sollte zu besseren Ergebnissen führen



## Regionaler Algorithmus

- Rechne superlokalen Algorithmus auf EBBs, verwendet dabei
  - Verschachtelte Hash-Tabellen für Geltungsbereiche
  - SSA-Form
- **Neu:** Initialisiere Tabelle für Knoten  $x$ 
  - Mit Tabelle von  $IDOM(x)$
  - Trage so Wissen über Merge Points hinweg
- Wie vorher möglich:
  - Constant folding
  - Ausnutzung algebraischer Eigenschaften

➡ Größerer Anwendungsbereich sollte zu besseren Ergebnissen führen



## Regionaler Algorithmus

- Rechne superlokalen Algorithmus auf EBBs, verwendet dabei
  - Verschachtelte Hash-Tabellen für Geltungsbereiche
  - SSA-Form
- **Neu:** Initialisiere Tabelle für Knoten  $x$ 
  - Mit Tabelle von  $IDOM(x)$
  - Trage so Wissen über Merge Points hinweg
- Wie vorher möglich:
  - Constant folding
  - Ausnutzung algebraischer Eigenschaften

➡ Größerer Anwendungsbereich sollte zu besseren Ergebnissen führen



## Regionaler Algorithmus

- Rechne superlokalen Algorithmus auf EBBs, verwendet dabei
  - Verschachtelte Hash-Tabellen für Geltungsbereiche
  - SSA-Form
- **Neu:** Initialisiere Tabelle für Knoten  $x$ 
  - Mit Tabelle von  $IDOM(x)$
  - Trage so Wissen über Merge Points hinweg
- Wie vorher möglich:
  - Constant folding
  - Ausnutzung algebraischer Eigenschaften

➡ Größerer Anwendungsbereich sollte zu besseren Ergebnissen führen





## Regionaler Algorithmus

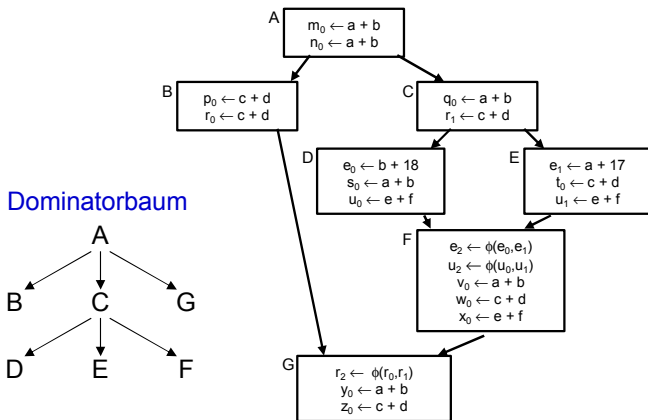
- Rechne superlokalen Algorithmus auf EBBs, verwendet dabei
  - Verschachtelte Hash-Tabellen für Geltungsbereiche
  - SSA-Form
- **Neu:** Initialisiere Tabelle für Knoten  $x$ 
  - Mit Tabelle von  $IDOM(x)$
  - Trage so Wissen über Merge Points hinweg
- Wie vorher möglich:
  - Constant folding
  - Ausnutzung algebraischer Eigenschaften

➔ Größerer Anwendungsbereich sollte zu besseren Ergebnissen führen



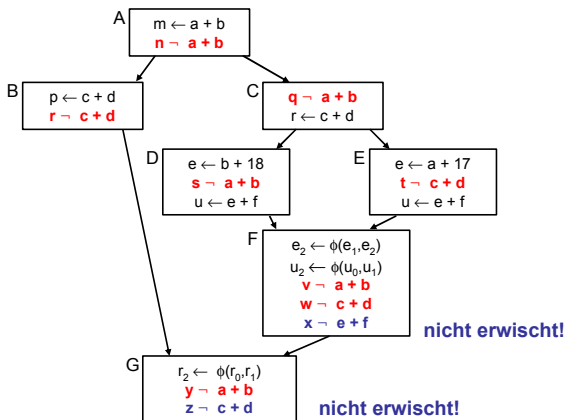
Arbeitet in Prefix-Reihenfolge auf **Dominatorbaum**

- Stellt sicher, das  $IDOM(x)$  vor  $x$  bearbeitet ist



Mögliche Reihenfolge: A, B, G, C, D, E, F

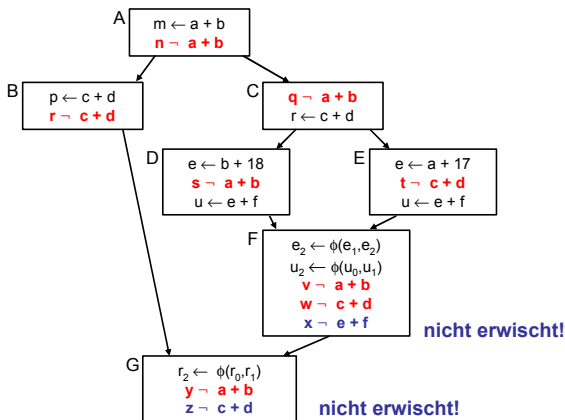
# Beispiel DVNT 3



- Erkennt nochmehr Redundanz
- Aber immer noch nicht alle Möglichkeiten
- Scheitert z.B. bei Schleifen

● Rückwärtskanten im CFG

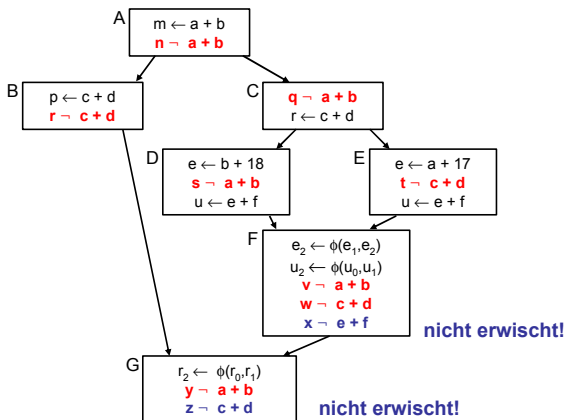
# Beispiel DVNT 3



- Erkennt nochmehr Redundanz
- Aber immer noch nicht alle Möglichkeiten
- Scheitert z.B. bei Schleifen

• Rückwärtskanten im CFG

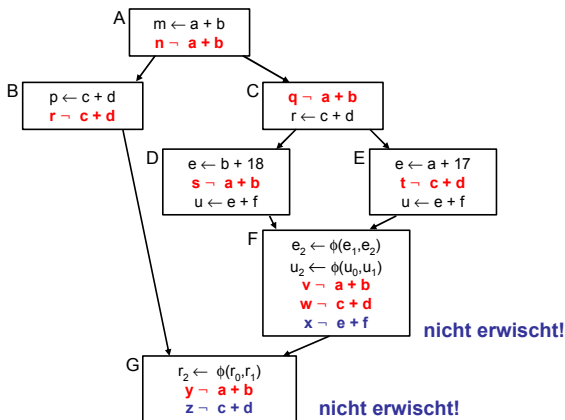
# Beispiel DVNT 3



- Erkennt nochmehr Redundanz
- Aber immer noch nicht alle Möglichkeiten
- Scheitert z.B. bei Schleifen

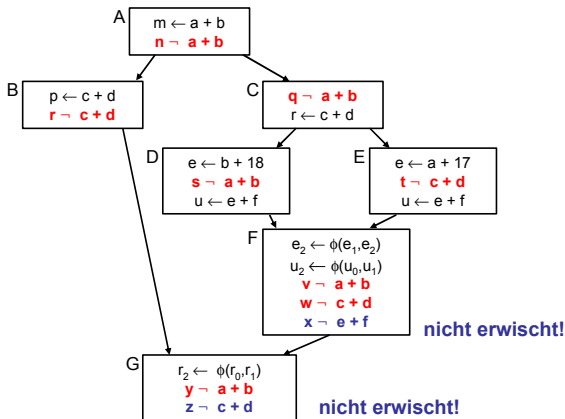
• Rückwärtskanten im CFG

# Beispiel DVNT 3



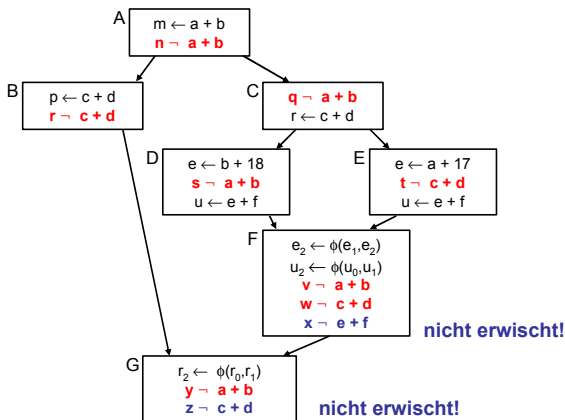
- Erkennt nochmehr Redundanz
- Aber immer noch nicht alle Möglichkeiten
- Scheitert z.B. bei Schleifen
  - Rückwärtskanten im CFG

# Beispiel DVNT 3



- Erkennt nochmehr Redundanz
- Aber immer noch nicht alle Möglichkeiten
- Scheitert z.B. bei Schleifen
  - Rückwärtskanten im CFG

# Beispiel DVNT 3



- Erkennt nochmehr Redundanz
- Aber immer noch nicht alle Möglichkeiten
- Scheitert z.B. bei Schleifen
  - Rückwärtskanten im CFG





OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

**Global**

# Globale Methoden



- Über CFG der **kompletten** Prozedur
- Allgemeine Vorgehensweise: Schritte trennen
- **Analyse** sammelt Informationen, auch über Zyklus hinweg
- **Transformation** erst, wenn alles fertig analysiert



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

- Über CFG der **kompletten** Prozedur
- Allgemeine Vorgehensweise: Schritte trennen
- **Analyse** sammelt Informationen, auch über Zyklus hinweg
- **Transformation** erst, wenn alles fertig analysiert



- Über CFG der **kompletten** Prozedur
- Allgemeine Vorgehensweise: Schritte trennen
- **Analyse** sammelt Informationen, auch über Zyklus hinweg
- **Transformation** erst, wenn alles fertig analysiert



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

- Über CFG der **kompletten** Prozedur
- Allgemeine Vorgehensweise: Schritte trennen
- **Analyse** sammelt Informationen, auch über Zyklus hinweg
- **Transformation** erst, wenn alles fertig analysiert



## Am Beispiel **Global Common Subexpression Elimination**

- Nun wieder **lexikalischer Vergleich von Namen**
- **Keine** Wertnummern mehr
- Keine SSA-Form mehr
- Demonstriert das wichtige Mittel der **Datenflussanalyse**



## Am Beispiel **Global Common Subexpression Elimination**

- Nun wieder **lexikalischer** Vergleich von **Namen**
- **Keine** Wertnummern mehr
- Keine SSA-Form mehr
- Demonstriert das wichtige Mittel der **Datenflussanalyse**



## Am Beispiel **Global Common Subexpression Elimination**

- Nun wieder **lexikalischer** Vergleich von **Namen**
- **Keine** Wertnummern mehr
- Keine SSA-Form mehr
- Demonstriert das wichtige Mittel der **Datenflussanalyse**





## Am Beispiel **Global Common Subexpression Elimination**

- Nun wieder **lexikalischer** Vergleich von **Namen**
- **Keine** Wertnummern mehr
- Keine SSA-Form mehr
- Demonstriert das wichtige Mittel der **Datenflussanalyse**



## Am Beispiel **Global Common Subexpression Elimination**

- Nun wieder **lexikalischer** Vergleich von **Namen**
- **Keine** Wertnummern mehr
- Keine SSA-Form mehr
- Demonstriert das wichtige Mittel der **Datenflussanalyse**



- **Global Value Numbering**
- Graphenbasierte Vorgehensweise
- Idee: Kongruente Wertgraphen in allen Zweigen des CFG finden
- Hier aber nicht weiter verfolgt
  - ... aus Zeitmangel
  - Siehe Muchnick Abschnitt 13.1.2



- **Global Value Numbering**
- **Graphenbasierte Vorgehensweise**
- Idee: Kongruente Wertgraphen in allen Zweigen des CFG finden
- Hier aber nicht weiter verfolgt
  - ... aus Zeitmangel
  - Siehe Muchnick Abschnitt 13.1.2



- **Global Value Numbering**
- Graphenbasierte Vorgehensweise
- Idee: Kongruente Wertgraphen in allen Zweigen des CFG finden
- Hier aber nicht weiter verfolgt
  - ... aus Zeitmangel
  - Siehe Muchnick Abschnitt 13.1.2



- **Global Value Numbering**
- Graphenbasierte Vorgehensweise
- Idee: Kongruente Wertgraphen in allen Zweigen des CFG finden
- Hier aber nicht weiter verfolgt
  - ... aus Zeitmangel
  - Siehe Muchnick Abschnitt 13.1.2



- **Global Value Numbering**
- Graphenbasierte Vorgehensweise
- Idee: Kongruente Wertgraphen in allen Zweigen des CFG finden
- Hier aber nicht weiter verfolgt
  - ... aus Zeitmangel
  - Siehe Muchnick Abschnitt 13.1.2



- **Global Value Numbering**
- Graphenbasierte Vorgehensweise
- Idee: Kongruente Wertgraphen in allen Zweigen des CFG finden
- Hier aber nicht weiter verfolgt
  - ... aus Zeitmangel
  - Siehe Muchnick Abschnitt 13.1.2





- Eine Auswertung eines Ausdrucks  $e$  an der Stelle  $p$  ist redundant
- ... genau dann, wenn
  - Jeder Pfad vom Prozedurstart zu  $p$  enthält eine Auswertung von  $e$
  - und die Werte der Teilausdrücke von  $e$  haben sich von dort zu  $p$  nicht geändert

➔ Auswertung  $e$  an  $p$  liefert **gleichen** Wert wie frühere Auswertung(en)

Wie diese redundanten (Teil)ausdrücke finden?



- Eine Auswertung eines Ausdrucks  $e$  an der Stelle  $p$  ist redundant
- ... genau dann, wenn
  - Jeder Pfad vom Prozedurstart zu  $p$  enthält eine Auswertung von  $e$
  - **und** die Werte der Teilausdrücke von  $e$  haben sich von dort zu  $p$  nicht geändert

➔ Auswertung  $e$  an  $p$  liefert **gleichen** Wert wie frühere Auswertung(en)

Wie diese redundanten (Teil)ausdrücke finden?



- Eine Auswertung eines Ausdrucks  $e$  an der Stelle  $p$  ist redundant
- ... genau dann, wenn
  - Jeder Pfad vom Prozedurstart zu  $p$  enthält eine Auswertung von  $e$
  - **und** die Werte der Teilausdrücke von  $e$  haben sich von dort zu  $p$  nicht geändert

➔ Auswertung  $e$  an  $p$  liefert **gleichen** Wert wie frühere Auswertung(en)

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Wie diese redundanten (Teil)ausdrücke finden?



- Eine Auswertung eines Ausdrucks  $e$  an der Stelle  $p$  ist redundant
- ... genau dann, wenn
  - Jeder Pfad vom Prozedurstart zu  $p$  enthält eine Auswertung von  $e$
  - **und** die Werte der Teilausdrücke von  $e$  haben sich von dort zu  $p$  nicht geändert

➔ Auswertung  $e$  an  $p$  liefert **gleichen** Wert wie frühere Auswertung(en)

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Wie diese redundanten (Teil)ausdrücke finden?



- Eine Auswertung eines Ausdrucks  $e$  an der Stelle  $p$  ist redundant
- ... genau dann, wenn
  - Jeder Pfad vom Prozedurstart zu  $p$  enthält eine Auswertung von  $e$
  - **und** die Werte der Teilausdrücke von  $e$  haben sich von dort zu  $p$  nicht geändert

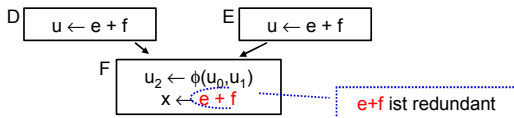
➔ Auswertung  $e$  an  $p$  liefert **gleichen** Wert wie frühere Auswertung(en)

Wie diese redundanten (Teil)ausdrücke finden?



- Eine Auswertung eines Ausdrucks  $e$  an der Stelle  $p$  ist redundant
- ... genau dann, wenn
  - Jeder Pfad vom Prozedurstart zu  $p$  enthält eine Auswertung von  $e$
  - **und** die Werte der Teilausdrücke von  $e$  haben sich von dort zu  $p$  nicht geändert

➔ Auswertung  $e$  an  $p$  liefert **gleichen** Wert wie frühere Auswertung(en)

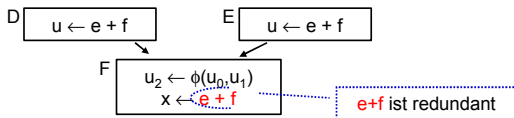


Wie diese redundanten (Teil)ausdrücke finden?



- Eine Auswertung eines Ausdrucks  $e$  an der Stelle  $p$  ist redundant
- ... genau dann, wenn
  - Jeder Pfad vom Prozedurstart zu  $p$  enthält eine Auswertung von  $e$
  - **und** die Werte der Teilausdrücke von  $e$  haben sich von dort zu  $p$  nicht geändert

➔ Auswertung  $e$  an  $p$  liefert **gleichen** Wert wie frühere Auswertung(en)

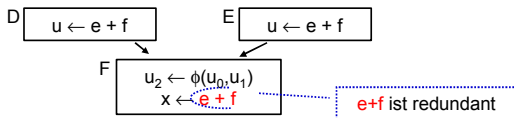


Wie diese redundanten (Teil)ausdrücke finden?



- Eine Auswertung eines Ausdrucks  $e$  an der Stelle  $p$  ist redundant
- ... genau dann, wenn
  - Jeder Pfad vom Prozedurstart zu  $p$  enthält eine Auswertung von  $e$
  - **und** die Werte der Teilausdrücke von  $e$  haben sich von dort zu  $p$  nicht geändert

➔ Auswertung  $e$  an  $p$  liefert **gleichen** Wert wie frühere Auswertung(en)



Wie diese redundanten (Teil)ausdrücke finden?





## Definition

Ein Ausdruck  $e$  ist **definiert** an einer Stelle  $p$  im CFG falls sein Wert an der Stelle  $p$  berechnet wird.  $p$  ist damit eine **Definitionsstelle** von  $e$ .



## Auslöschung

Ein Ausdruck  $e$  wird **ausgelöscht** an einer Stelle  $p$  im CFG falls ein oder mehrere seiner Operanden an der Stelle  $p$  definiert werden.  $p$  ist damit eine **Auslöschungsstelle** von  $e$ .



## Verfügbarkeit

Ein Ausdruck  $e$  ist **verfügbar** an einer Stelle  $p$  wenn jeder zu  $p$  führende Pfad im CFG, beginnend beim Prozeduranfang, eine vorhergehende Definition von  $e$  enthält und  $e$  nicht zwischen dieser Definitionsstelle und  $p$  ausgelöscht wird.

# Identifikation von Ausdrücken



Hier lexikalisch, **nicht** mehr über Wertnummern!

```
x := e + f; // Definition von e+f
e := 5; // Ausloeschung e+f
y := e + f; // nicht redundant!
```

- Identifikation von Variablen über **Namen**
- Identifikation von Ausdrücken über
  - Eindeutigen Hash-Wert, berechnet über gesamten Ausdruck
  - Operatoren werden über Operationen
  - Argumente Hash-Werte ihrer aufgerufenen durch
- Anzahlen
  - Minimal Anzahl von Parametern Argumenten
  - Minimal Anzahl von Parameterausdrücken im Arg.

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

# Identifikation von Ausdrücken



Hier lexikalisch, **nicht** mehr über Wertnummern!

```
x := e + f; // Definition von e+f
e := 5; // Ausloeschung e+f
y := e + f; // nicht redundant!
```

- Identifikation von Variablen über Namen
- Identifikation von Ausdrücken über
  - Eindeutigen Hauptoperator, Bereichsoperator, Operator
  - Operatoren und Operanden
  - Ausdrücke Hauptoperator, Bereichsoperator, Operator
- Anzahlen
  - Hauptoperator, Bereichsoperator, Operator
  - Operatoren und Operanden
  - Ausdrücke Hauptoperator, Bereichsoperator, Operator

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

# Identifikation von Ausdrücken



Hier lexikalisch, **nicht** mehr über Wertnummern!

```
x := e + f; // Definition von e+f
e := 5; // Ausloeschung e+f
y := e + f; // nicht redundant!
```

- Identifikation von Variablen über **Namen**
- Identifikation von Ausdrücken über
  - Eindeutigen Hash-Wert, berechnet über gesamten Ausdruck
  - Operandennamen und Operatoren
  - Numeriere Hash-Werte dann aufsteigend durch
- Anzahlen
  - Minimal: Anzahl Variablen plus Anzahl Konstanten
  - Maximal: Anzahl von (Teil)Ausdrücken im CFG

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

# Identifikation von Ausdrücken



Hier lexikalisch, **nicht** mehr über Wertnummern!

```
x := e + f; // Definition von e+f
e := 5; // Ausloeschung e+f
y := e + f; // nicht redundant!
```

- Identifikation von Variablen über **Namen**
- Identifikation von Ausdrücken über
  - Eindeutigen Hash-Wert, berechnet über gesamten Ausdruck
  - Operanden**namen** und Operatoren
  - Numeriere Hash-Werte dann aufsteigend durch
- Anzahlen
  - Minimal: Anzahl Variablen plus Anzahl Konstanten
  - Maximal: Anzahl von (Teil)Ausdrücken im CFG

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

# Identifikation von Ausdrücken



Hier lexikalisch, **nicht** mehr über Wertnummern!

```
x := e + f; // Definition von e+f
e := 5; // Ausloeschung e+f
y := e + f; // nicht redundant!
```

- Identifikation von Variablen über **Namen**
- Identifikation von Ausdrücken über
  - Eindeutigen Hash-Wert, berechnet über gesamten Ausdruck
  - Operandennamen und Operatoren
  - Numeriere Hash-Werte dann aufsteigend durch
- Anzahlen
  - Minimal: Anzahl Variablen plus Anzahl Konstanten
  - Maximal: Anzahl von (Teil)Ausdrücken im CFG

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global





Hier lexikalisch, **nicht** mehr über Wertnummern!

```
x := e + f; // Definition von e+f
e := 5; // Ausloeschung e+f
y := e + f; // nicht redundant!
```

- Identifikation von Variablen über **Namen**
- Identifikation von Ausdrücken über
  - Eindeutigen Hash-Wert, berechnet über gesamten Ausdruck
  - Operanden**namen** und Operatoren
  - Numeriere Hash-Werte dann aufsteigend durch
- Anzahlen
  - Minimal: Anzahl Variablen plus Anzahl Konstanten
  - Maximal: Anzahl von (Teil)Ausdrücken im CFG



Hier lexikalisch, **nicht** mehr über Wertnummern!

```
x := e + f; // Definition von e+f
e := 5; // Ausloeschung e+f
y := e + f; // nicht redundant!
```

- Identifikation von Variablen über **Namen**
- Identifikation von Ausdrücken über
  - Eindeutigen Hash-Wert, berechnet über gesamten Ausdruck
  - Operanden**namen** und Operatoren
  - Numeriere Hash-Werte dann aufsteigend durch
- Anzahlen
  - Minimal: Anzahl Variablen plus Anzahl Konstanten
  - Maximal: Anzahl von (Teil)Ausdrücken im CFG



Hier lexikalisch, **nicht** mehr über Wertnummern!

```
x := e + f; // Definition von e+f
e := 5; // Ausloeschung e+f
y := e + f; // nicht redundant!
```

- Identifikation von Variablen über **Namen**
- Identifikation von Ausdrücken über
  - Eindeutigen Hash-Wert, berechnet über gesamten Ausdruck
  - Operanden**namen** und Operatoren
  - Numeriere Hash-Werte dann aufsteigend durch
- Anzahlen
  - Minimal: Anzahl Variablen plus Anzahl Konstanten
  - Maximal: Anzahl von (Teil)Ausdrücken im CFG



Hier lexikalisch, **nicht** mehr über Wertnummern!

```
x := e + f; // Definition von e+f
e := 5; // Ausloeschung e+f
y := e + f; // nicht redundant!
```

- Identifikation von Variablen über **Namen**
- Identifikation von Ausdrücken über
  - Eindeutigen Hash-Wert, berechnet über gesamten Ausdruck
  - Operanden**namen** und Operatoren
  - Numeriere Hash-Werte dann aufsteigend durch
- Anzahlen
  - Minimal: Anzahl Variablen plus Anzahl Konstanten
  - Maximal: Anzahl von (Teil)Ausdrücken im CFG



Hier lexikalisch, **nicht** mehr über Wertnummern!

```
x := e + f; // Definition von e+f
e := 5; // Ausloeschung e+f
y := e + f; // nicht redundant!
```

- Identifikation von Variablen über **Namen**
- Identifikation von Ausdrücken über
  - Eindeutigen Hash-Wert, berechnet über gesamten Ausdruck
  - Operanden**namen** und Operatoren
  - Numeriere Hash-Werte dann aufsteigend durch
- Anzahlen
  - Minimal: Anzahl Variablen plus Anzahl Konstanten
  - Maximal: Anzahl von (Teil)Ausdrücken im CFG



## Ziel

Wenn ein Ausdruck  $e$  in einem Block **verfügbar** ist, braucht er dort nicht neu berechnet zu werden.

... nun über Basis-Blockgrenzen und Verzweigungen hinweg!



## Ziel

Wenn ein Ausdruck  $e$  in einem Block **verfügbar** ist, braucht er dort nicht neu berechnet zu werden.

... nun über Basis-Blockgrenzen und Verzweigungen hinweg!



## 1. Schritt: Analyse

- Formuliere **Gleichungssystem** über den CFG der Prozedur
- Löse Gleichungssystem, um Menge **verfügbarer** Ausdrücke zu bestimmen

$AVAIL(b)$

$AVAIL(b)$  sei Menge der am Anfang von Block  $b$  verfügbaren Ausdrücke.





## 1. Schritt: Analyse

- Formuliere **Gleichungssystem** über den CFG der Prozedur
- Löse Gleichungssystem, um Menge **verfügbarer** Ausdrücke zu bestimmen

$AVAIL(b)$

$AVAIL(b)$  sei Menge der am Anfang von Block  $b$  verfügbaren Ausdrücke.



## 1. Schritt: Analyse

- Formuliere **Gleichungssystem** über den CFG der Prozedur
- Löse Gleichungssystem, um Menge **verfügbarer** Ausdrücke zu bestimmen

$AVAIL(b)$

$AVAIL(b)$  sei Menge der am Anfang von Block  $b$  verfügbaren Ausdrücke.



## 1. Schritt: Analyse

- Formuliere **Gleichungssystem** über den CFG der Prozedur
- Löse Gleichungssystem, um Menge **verfügbarer** Ausdrücke zu bestimmen

$AVAIL(b)$

$AVAIL(b)$  sei Menge der am Anfang von Block  $b$  verfügbaren Ausdrücke.



## 1. Schritt: Analyse

- Formuliere **Gleichungssystem** über den CFG der Prozedur
- Löse Gleichungssystem, um Menge **verfügbarer** Ausdrücke zu bestimmen

$AVAIL(b)$

$AVAIL(b)$  sei Menge der am Anfang von Block  $b$  verfügbaren Ausdrücke.



## Sicherheit

- $x + y \in \text{AVAIL}(b)$  **beweist**, dass eine **vorherige** Auswertung von  $x + y$  existiert
- Spätere Transformation muss über einen **Namen** auf diesen Wert zugreifen können
  - ... auf verschiedene Weisen realisierbar

## Profitabilität

- Verursacht keine **zusätzlichen** Evaluationen
- Fügt aber **Kopieroperationen** ein



## Sicherheit

- $x + y \in \text{AVAIL}(b)$  **beweist**, dass eine **vorherige** Auswertung von  $x + y$  existiert
- Spätere Transformation muss über einen **Namen** auf diesen Wert zugreifen können
  - ... auf verschiedene Weisen realisierbar

## Profitabilität

- Verursacht keine **zusätzlichen** Evaluationen
- Fügt aber **Kopieroperationen** ein



## Sicherheit

- $x + y \in \text{AVAIL}(b)$  **beweist**, dass eine **vorherige** Auswertung von  $x + y$  existiert
- Spätere Transformation muss über einen **Namen** auf diesen Wert zugreifen können
  - ... auf verschiedene Weisen realisierbar

## Profitabilität

- Verursacht keine **zusätzlichen** Evaluationen
- Fügt aber **Kopieroperationen** ein



## Sicherheit

- $x + y \in \text{AVAIL}(b)$  **beweist**, dass eine **vorherige** Auswertung von  $x + y$  existiert
- Spätere Transformation muss über einen **Namen** auf diesen Wert zugreifen können
  - ... auf verschiedene Weisen realisierbar

## Profitabilität

- Verursacht keine **zusätzlichen** Evaluationen
- Fügt aber **Kopieroperationen** ein





## Sicherheit

- $x + y \in \text{AVAIL}(b)$  **beweist**, dass eine **vorherige** Auswertung von  $x + y$  existiert
- Spätere Transformation muss über einen **Namen** auf diesen Wert zugreifen können
  - ... auf verschiedene Weisen realisierbar

## Profitabilität

- Verursacht keine **zusätzlichen** Evaluationen
- Fügt aber **Kopieroperationen** ein
  - Im Prinzip billig
  - Viele können auch entfernt werden
  - Haben aber Einfluß auf Lebenszeiten (später ...)



## Sicherheit

- $x + y \in \text{AVAIL}(b)$  **beweist**, dass eine **vorherige** Auswertung von  $x + y$  existiert
- Spätere Transformation muss über einen **Namen** auf diesen Wert zugreifen können
  - ... auf verschiedene Weisen realisierbar

## Profitabilität

- Verursacht keine **zusätzlichen** Evaluationen
- Fügt aber **Kopieroperationen** ein
  - Im Prinzip billig
  - Viele können auch entfernt werden
  - Haben aber Einfluß auf Lebenszeiten (später ...)



## Sicherheit

- $x + y \in \text{AVAIL}(b)$  **beweist**, dass eine **vorherige** Auswertung von  $x + y$  existiert
- Spätere Transformation muss über einen **Namen** auf diesen Wert zugreifen können
  - ... auf verschiedene Weisen realisierbar

## Profitabilität

- Verursacht keine **zusätzlichen** Evaluationen
- Fügt aber **Kopieroperationen** ein
  - Im Prinzip billig
  - Viele können auch entfernt werden
  - Haben aber Einfluß auf Lebenszeiten (später ...)



## Sicherheit

- $x + y \in \text{AVAIL}(b)$  **beweist**, dass eine **vorherige** Auswertung von  $x + y$  existiert
- Spätere Transformation muss über einen **Namen** auf diesen Wert zugreifen können
  - ... auf verschiedene Weisen realisierbar

## Profitabilität

- Verursacht keine **zusätzlichen** Evaluationen
- Fügt aber **Kopieroperationen** ein
  - Im Prinzip billig
  - Viele können auch entfernt werden
  - Haben aber Einfluß auf Lebenszeiten (später ...)



## Sicherheit

- $x + y \in \text{AVAIL}(b)$  **beweist**, dass eine **vorherige** Auswertung von  $x + y$  existiert
- Spätere Transformation muss über einen **Namen** auf diesen Wert zugreifen können
  - ... auf verschiedene Weisen realisierbar

## Profitabilität

- Verursacht keine **zusätzlichen** Evaluationen
- Fügt aber **Kopieroperationen** ein
  - Im Prinzip billig
  - Viele können auch entfernt werden
  - Haben aber Einfluß auf Lebenszeiten (später ...)

# Berechnung von $AVAIL(b)$ 1



## Über zwei Hilfsfunktionen

### $EXPRKILL(b)$

Die Menge der bei Ende des Blocks  $b$  ausgelöschten Ausdrücke.

### $DEEXPR(b)$ (*downward exposed*)

Menge der im Block  $b$  definierten Ausdrücke, die bis zum Ende des Blocks nicht ausgelöscht werden.

## Weiterleitung von Ausdrücken:

- Wenn Ausdruck  $e$  bei Eintritt in Block  $b$  verfügbar ist, und **nicht** in  $EXPRKILL(b)$  ist, dann ist  $e$  auch nach Ende von  $b$  verfügbar.

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

# Berechnung von $AVAIL(b)$ 1



Über zwei Hilfsfunktionen

$EXPRKILL(b)$

Die Menge der bei Ende des Blocks  $b$  ausgelöschten Ausdrücke.

$DEEXPR(b)$  (*downward exposed*)

Menge der im Block  $b$  definierten Ausdrücke, die bis zum Ende des Blocks nicht ausgelöscht werden.

Weiterleitung von Ausdrücken:

- Wenn Ausdruck  $e$  bei Eintritt in Block  $b$  verfügbar ist, und **nicht** in  $EXPRKILL(b)$  ist, dann ist  $e$  auch nach Ende von  $b$  verfügbar.

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

# Berechnung von $AVAIL(b)$ 1



Über zwei Hilfsfunktionen

$EXPRKILL(b)$

Die Menge der bei Ende des Blocks  $b$  ausgelöschten Ausdrücke.

$DEEXPR(b)$  (*downward exposed*)

Menge der im Block  $b$  definierten Ausdrücke, die bis zum Ende des Blocks nicht ausgelöscht werden.

Weiterleitung von Ausdrücken:

- Wenn Ausdruck  $e$  bei Eintritt in Block  $b$  verfügbar ist, und **nicht** in  $EXPRKILL(b)$  ist, dann ist  $e$  auch nach Ende von  $b$  verfügbar.

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global



# Berechnung von $AVAIL(b)$ 1



Über zwei Hilfsfunktionen

$EXPRKILL(b)$

Die Menge der bei Ende des Blocks  $b$  ausgelöschten Ausdrücke.

$DEEXPR(b)$  (*downward exposed*)

Menge der im Block  $b$  definierten Ausdrücke, die bis zum Ende des Blocks nicht ausgelöscht werden.

Weiterleitung von Ausdrücken:

- Wenn Ausdruck  $e$  bei Eintritt in Block  $b$  verfügbar ist, und **nicht** in  $EXPRKILL(b)$  ist, dann ist  $e$  auch nach Ende von  $b$  verfügbar.

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

# Berechnung von $AVAIL(b)$ 2



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Damit nun definierbar

$$AVAIL(b) = \bigcap_{a \in \text{pred}(b)} \text{DEEXPR}(a) \cup (AVAIL(a) \cap \overline{\text{EXPRKILL}(a)})$$

$$AVAIL(b_0) = \emptyset$$

mit:

$\text{pred}(b)$ : Vorgängerblöcke von  $b$  im CFG

$b_0$  Startblock des CFG

➔ Datenflußproblem, lösen mit Standardverfahren

# Berechnung von $AVAIL(b)$ 2



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Damit nun definierbar

$$AVAIL(b) = \bigcap_{a \in \text{pred}(b)} \text{DEEXPR}(a) \cup (AVAIL(a) \cap \overline{\text{EXPRKILL}(a)})$$

$$AVAIL(b_0) = \emptyset$$

mit:

$\text{pred}(b)$ : Vorgängerblöcke von  $b$  im CFG

$b_0$  Startblock des CFG

➔ Datenflußproblem, lösen mit Standardverfahren

# Berechnung von $AVAIL(b)$ 2



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Damit nun definierbar

$$AVAIL(b) = \bigcap_{a \in \text{pred}(b)} \text{DEEXPR}(a) \cup (AVAIL(a) \cap \overline{\text{EXPRKILL}(a)})$$

$$AVAIL(b_0) = \emptyset$$

mit:

$\text{pred}(b)$ : Vorgängerblöcke von  $b$  im CFG

$b_0$  Startblock des CFG

➔ Datenflußproblem, lösen mit Standardverfahren



Vor den Details erstmal einen Schritt zurück: Wie soll es weitergehen?

- 1 Berechne  $AVAIL(b)$  für alle Blöcke  $b$
- 2 Vergebe dann eindeutige, CFG-globale Bezeichner für Ausdrücke in  $AVAIL(b)$
- 3 Innerhalb der Blöcke  $b$  dann **lokales Value Numbering**
  - Initialisiert block-lokale Tabelle mit Ausdrücken aus  $AVAIL(b)$



Vor den Details erstmal einen Schritt zurück: Wie soll es weitergehen?

- 1 Berechne  $AVAIL(b)$  für alle Blöcke  $b$
- 2 Vergebe dann eindeutige, CFG-globale Bezeichner für Ausdrücke in  $AVAIL(b)$
- 3 Innerhalb der Blöcke  $b$  dann **lokales Value Numbering**
  - Initialisiert block-lokale Tabelle mit Ausdrücken aus  $AVAIL(b)$



Vor den Details erstmal einen Schritt zurück: Wie soll es weitergehen?

- 1 Berechne  $AVAIL(b)$  für alle Blöcke  $b$
- 2 Vergebe dann eindeutige, CFG-globale Bezeichner für Ausdrücke in  $AVAIL(b)$
- 3 Innerhalb der Blöcke  $b$  dann **lokales Value Numbering**
  - Initialisiert block-lokale Tabelle mit Ausdrücken aus  $AVAIL(b)$



Vor den Details erstmal einen Schritt zurück: Wie soll es weitergehen?

- 1 Berechne  $AVAIL(b)$  für alle Blöcke  $b$
- 2 Vergebe dann eindeutige, CFG-globale Bezeichner für Ausdrücke in  $AVAIL(b)$
- 3 Innerhalb der Blöcke  $b$  dann **lokales Value Numbering**
  - Initialisiert block-lokale Tabelle mit Ausdrücken aus  $AVAIL(b)$



# Berechnung von AVAIL( $b$ ) 3



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

## Baut auf Berechnung von DEEXPR und EXPRKILL auf

*assume a block  $b$  with operations  $o_1, o_2, \dots, o_k$*

$VARKILL \leftarrow \emptyset$

$DEEXPR(b) \leftarrow \emptyset$

**for**  $i = k$  **to** 1

*assume  $o_i$  is " $x \rightarrow y + z$ "*

add  $x$  to  $VARKILL$

**if** ( $y \notin VARKILL$ ) **and** ( $z \notin VARKILL$ ) **then**

add " $y + z$ " to  $DEEXPR(b)$

$EXPRKILL(b) \leftarrow \emptyset$

**For each** expression  $e$

**for each** variable  $v \in e$

**if**  $v \in VARKILL(b)$  **then**

$EXPRKILL(b) \leftarrow EXPRKILL(b) \cup \{e\}$

# Berechnung von $AVAIL(b)$ 3



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

## Baut auf Berechnung von $DEEXPR$ und $EXPRKILL$ auf

assume a block  $b$  with operations  $o_1, o_2, \dots, o_k$

$VARKILL \leftarrow \emptyset$

$DEEXPR(b) \leftarrow \emptyset$

Rückwärts durch Block

for  $i = k$  to  $1$ :

assume  $o_i$  is " $x \rightarrow y + z$ "

add  $x$  to  $VARKILL$

if ( $y \notin VARKILL$ ) and ( $z \notin VARKILL$ ) then

add " $y + z$ " to  $DEEXPR(b)$

}  $O(k)$  Schritte

$EXPRKILL(b) \leftarrow \emptyset$

For each expression  $e$

for each variable  $v \in e$

if  $v \in VARKILL(b)$  then

$EXPRKILL(b) \leftarrow EXPRKILL(b) \cup \{e\}$

}  $O(N)$  Schritte  
 $N$  ist # von  
Operationen

# Berechnung von $AVAIL(b)$ 4



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

- “**Foreach** expression  $e$ ”  
über alle Ausdrücke der **Prozedur**
- Potentiell sehr langsam!
- Abhilfe
  - Hash-Map  $M$  von  $v \rightarrow E$  bildet Variable ab auf benutzende Ausdrücke
  - Dann mit  $v$  über  $VARKILL(b)$  iterieren
  - Für jede Variable  $v$  via  $M(v)$  Ausdrücke  $E$  bestimmen
  - $v$  und  $E$  in  $EXPRKILL(b)$  aufnehmen

# Berechnung von $AVAIL(b)$ 4



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

- “**Foreach** expression  $e$ ”  
über alle Ausdrücke der **Prozedur**
- Potentiell sehr langsam!
- Abhilfe
  - Hash-Map  $M$  von  $v \rightarrow E$  bildet Variable ab auf benutzende Ausdrücke
  - Dann mit  $v$  über  $VARKILL(b)$  iterieren
  - Für jede Variable  $v$  via  $M(v)$  Ausdrücke  $E$  bestimmen
  - $v$  und  $E$  in  $EXPRKILL(b)$  aufnehmen

# Berechnung von $AVAIL(b)$ 4



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

- “**Foreach** expression  $e$ ”  
über alle Ausdrücke der **Prozedur**
- Potentiell sehr langsam!
- Abhilfe
  - Hash-Map  $M$  von  $v \rightarrow E$  bildet Variable ab auf benutzende Ausdrücke
  - Dann mit  $v$  über  $VARKILL(b)$  iterieren
  - Für jede Variable  $v$  via  $M(v)$  Ausdrücke  $E$  bestimmen
  - ... und  $E$  in  $EXPRKILL(b)$  aufnehmen

# Berechnung von $AVAIL(b)$ 4



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

- “**Foreach** expression  $e$ ”  
über alle Ausdrücke der **Prozedur**
- Potentiell sehr langsam!
- Abhilfe
  - Hash-Map  $M$  von  $v \rightarrow E$  bildet Variable ab auf benutzende Ausdrücke
  - Dann mit  $v$  über  $VARKILL(b)$  iterieren
  - Für jede Variable  $v$  via  $M(v)$  Ausdrücke  $E$  bestimmen
  - ... und  $E$  in  $EXPRKILL(b)$  aufnehmen

# Berechnung von $AVAIL(b)$ 4



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

- “**Foreach** expression  $e$ ”  
über alle Ausdrücke der **Prozedur**
- Potentiell sehr langsam!
- Abhilfe
  - Hash-Map  $M$  von  $v \rightarrow E$  bildet Variable ab auf benutzende Ausdrücke
  - Dann mit  $v$  über  $VARKILL(b)$  iterieren
  - Für jede Variable  $v$  via  $M(v)$  Ausdrücke  $E$  bestimmen
  - ... und  $E$  in  $EXPRKILL(b)$  aufnehmen



- “**Foreach** expression  $e$ ”  
über alle Ausdrücke der **Prozedur**
- Potentiell sehr langsam!
- Abhilfe
  - Hash-Map  $M$  von  $v \rightarrow E$  bildet Variable ab auf benutzende Ausdrücke
  - Dann mit  $v$  über  $VARKILL(b)$  iterieren
  - Für jede Variable  $v$  via  $M(v)$  Ausdrücke  $E$  bestimmen
  - ... und  $E$  in  $EXPRKILL(b)$  aufnehmen





- “**Foreach** expression  $e$ ”  
über alle Ausdrücke der **Prozedur**
- Potentiell sehr langsam!
- Abhilfe
  - Hash-Map  $M$  von  $v \rightarrow E$  bildet Variable ab auf benutzende Ausdrücke
  - Dann mit  $v$  über  $VARKILL(b)$  iterieren
  - Für jede Variable  $v$  via  $M(v)$  Ausdrücke  $E$  bestimmen
  - ... und  $E$  in  $EXPRKILL(b)$  aufnehmen

# Berechnung von $AVAIL(b)$ 5



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Nun Anwendung eines iterativen Algorithmus zum Finden eines Fixpunktes

$Worklist \leftarrow \{ \text{all blocks in CFG} \}$

**while** ( $Worklist \neq \emptyset$ )

remove a block  $b$  from  $Worklist$

recompute  $AVAIL(b)$  as

$$AVAIL(b_i) = \bigcap_{x \in pred(b)} (DEEXPR(x) \cup (AVAIL(x) \cap \overline{EXPRKILL(x)}))$$

**if**  $AVAIL(b)$  changed **then**

$Worklist \leftarrow Worklist \cup successors(b)$

Beweis der Terminierung: Später ...

# Berechnung von $AVAIL(b)$ 5



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Nun Anwendung eines iterativen Algorithmus zum Finden eines Fixpunktes

$Worklist \leftarrow \{ \text{all blocks in CFG} \}$

**while** ( $Worklist \neq \emptyset$ )

  remove a block  $b$  from  $Worklist$

  recompute  $AVAIL(b)$  as

$$AVAIL(b_i) = \bigcap_{x \in pred(b)} (DEEXPR(x) \cup (AVAIL(x) \cap \overline{EXPRKILL(x)}))$$

**if**  $AVAIL(b)$  changed **then**

$Worklist \leftarrow Worklist \cup successors(b)$

Beweis der Terminierung: Später ...

# Berechnung von $AVAIL(b)$ 5



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

Nun Anwendung eines iterativen Algorithmus zum Finden eines Fixpunktes

$Worklist \leftarrow \{ \text{all blocks in CFG} \}$

**while** ( $Worklist \neq \emptyset$ )

remove a block  $b$  from  $Worklist$

recompute  $AVAIL(b)$  as

$$AVAIL(b_i) = \bigcap_{x \in pred(b)} (DEEXPR(x) \cup (AVAIL(x) \cap \overline{EXPRKILL(x)}))$$

**if**  $AVAIL(b)$  changed **then**

$Worklist \leftarrow Worklist \cup successors(b)$

Beweis der Terminierung: Später ...

# Beispiel mit $AVAIL(b)$ -Mengen



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

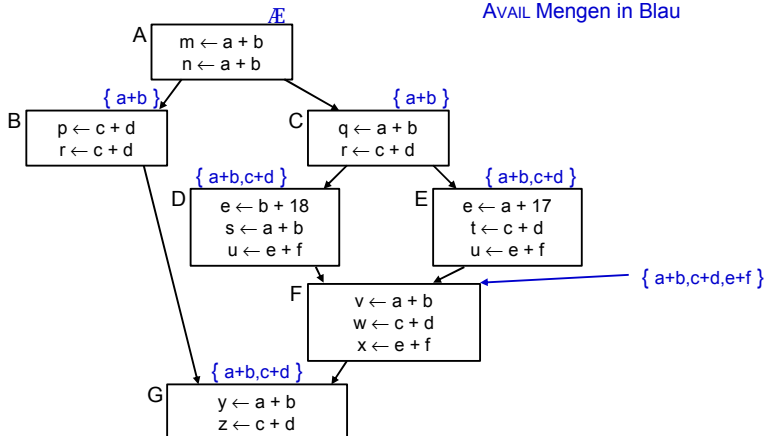
Wertnumerierung

Lokal

Superlokal

Regional

Global





- 1 Berechne  $AVAIL(b)$  für alle Blöcke  $b$ 
  - **Erledigt!**
- 2 Vergebe dann eindeutige, CFG-globale Bezeichner für Ausdrücke in  $AVAIL(b)$
- 3 Innerhalb der Blöcke  $b$  dann **lokales Value Numbering**
  - Initialisiert block-lokale Tabelle mit Ausdrücken aus  $AVAIL(b)$

Nun eindeutige Namen  $t_i$  für global bekannte (Teil-)ausdrücke  $e_i$  vergeben

➔ Hashing über Ausdrücke und Durchnumerieren mit  $i$



- 1 Berechne  $AVAIL(b)$  für alle Blöcke  $b$ 
  - **Erledigt!**
- 2 Vergebe dann eindeutige, CFG-globale Bezeichner für Ausdrücke in  $AVAIL(b)$
- 3 Innerhalb der Blöcke  $b$  dann **lokales Value Numbering**
  - Initialisiert block-lokale Tabelle mit Ausdrücken aus  $AVAIL(b)$

Nun eindeutige Namen  $t_i$  für global bekannte (Teil-)ausdrücke  $e_i$  vergeben

➔ Hashing über Ausdrücke und Durchnummerieren mit  $i$



- 1 Berechne  $AVAIL(b)$  für alle Blöcke  $b$ 
  - **Erledigt!**
- 2 Vergebe dann eindeutige, CFG-globale Bezeichner für Ausdrücke in  $AVAIL(b)$
- 3 Innerhalb der Blöcke  $b$  dann **lokales Value Numbering**
  - Initialisiert block-lokale Tabelle mit Ausdrücken aus  $AVAIL(b)$

Nun eindeutige Namen  $t_i$  für global bekannte (Teil-)ausdrücke  $e_i$  vergeben

➔ Hashing über Ausdrücke und Durchnumerieren mit  $i$





- 1 Berechne  $AVAIL(b)$  für alle Blöcke  $b$ 
  - **Erledigt!**
- 2 Vergebe dann eindeutige, CFG-globale Bezeichner für Ausdrücke in  $AVAIL(b)$
- 3 Innerhalb der Blöcke  $b$  dann **lokales Value Numbering**
  - Initialisiert block-lokale Tabelle mit Ausdrücken aus  $AVAIL(b)$

Nun eindeutige Namen  $t_i$  für global bekannte (Teil-)ausdrücke  $e_i$  vergeben

➔ Hashing über Ausdrücke und Durchnumerieren mit  $i$



- 1 Berechne  $AVAIL(b)$  für alle Blöcke  $b$ 
  - **Erledigt!**
- 2 Vergebe dann eindeutige, CFG-globale Bezeichner für Ausdrücke in  $AVAIL(b)$
- 3 Innerhalb der Blöcke  $b$  dann **lokales Value Numbering**
  - Initialisiert block-lokale Tabelle mit Ausdrücken aus  $AVAIL(b)$

Nun eindeutige Namen  $t_i$  für global bekannte (Teil-)ausdrücke  $e_i$  vergeben

➔ Hashing über Ausdrücke und Durchnumerieren mit  $i$



- 1 Berechne  $AVAIL(b)$  für alle Blöcke  $b$ 
  - **Erledigt!**
- 2 Vergebe dann eindeutige, CFG-globale Bezeichner für Ausdrücke in  $AVAIL(b)$
- 3 Innerhalb der Blöcke  $b$  dann **lokales Value Numbering**
  - Initialisiert block-lokale Tabelle mit Ausdrücken aus  $AVAIL(b)$

Nun eindeutige Namen  $t_i$  für global bekannte (Teil-)ausdrücke  $e_i$  vergeben

➔ Hashing über Ausdrücke und Durchnummerieren mit  $i$



- 1 Berechne  $AVAIL(b)$  für alle Blöcke  $b$ 
  - **Erledigt!**
- 2 Vergebe dann eindeutige, CFG-globale Bezeichner für Ausdrücke in  $AVAIL(b)$
- 3 Innerhalb der Blöcke  $b$  dann **lokales Value Numbering**
  - Initialisiert block-lokale Tabelle mit Ausdrücken aus  $AVAIL(b)$

Nun eindeutige Namen  $t_i$  für global bekannte (Teil-)ausdrücke  $e_i$  vergeben

➡ Hashing über Ausdrücke und Durchnumerieren mit  $i$



- 1 Berechne  $AVAIL(b)$  für alle Blöcke  $b$ 
  - **Erledigt!**
- 2 Vergebe dann eindeutige, CFG-globale Bezeichner für Ausdrücke in  $AVAIL(b)$
- 3 Innerhalb der Blöcke  $b$  dann **lokales Value Numbering**
  - Initialisiert block-lokale Tabelle mit Ausdrücken aus  $AVAIL(b)$

Nun eindeutige Namen  $t_i$  für global bekannte (Teil-)ausdrücke  $e_i$  vergeben

➔ Hashing über Ausdrücke und Durchnumerieren mit  $i$

# Beispiel mit benannten globalen CSEs



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

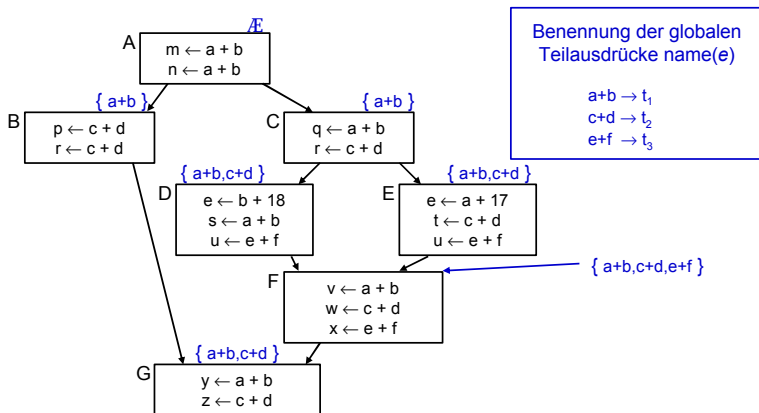
Wertnumerierung

Lokal

Superlokal

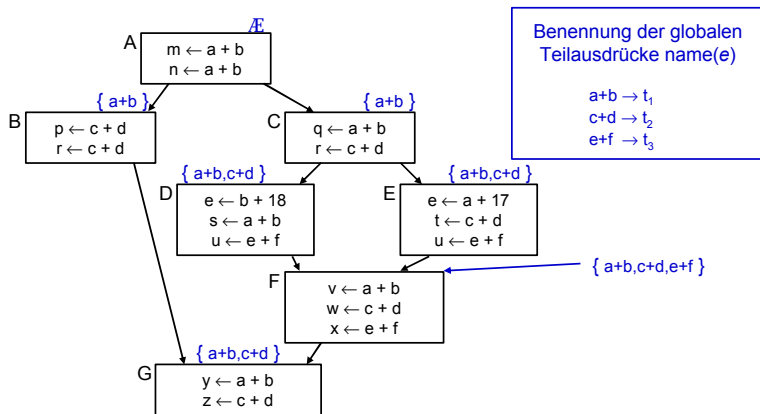
Regional

Global



Damit jetzt Schritt 3 (Transformation) vornehmen!

# Beispiel mit benannten globalen CSEs



Damit jetzt Schritt 3 (Transformation) vornehmen!



## Block-lokales Value Numbering in Block $b$

### 1. Phase: Finde Wiederbenutzungen eines Ausdrucks

- 1. Initialisiere Hash-Tabelle mit  $AVAIL(b)$

- 1.1. Für alle  $V \in AVAIL(b)$  setze  $AVAIL(V) = \emptyset$

- 1.2. Für alle  $e \in AVAIL(b)$  setze  $AVAIL(e) = \emptyset$

- 1.3. Setze  $i = 1$

- 2. Wenn Wiederverwendung  $x := e_i$  erkannt

- 2.1. Ersetze  $x$  durch  $V_{AVAIL(x)}$  auf dem rechten Operanden

- 2.2. Setze  $i = i + 1$

- 2.3. Wenn Wiederverwendung durch  $V_{AVAIL(x)}$  ist





## Block-lokales Value Numbering in Block $b$

### 1. Phase: Finde Wiederbenutzungen eines Ausdrucks

- 1. Initialisiere Hash-Tabelle mit  $AVAIL(b)$

• Für jeden Wert  $v$  in  $AVAIL(b)$  setze  $HT[v] = \emptyset$

• Für jeden Ausdruck  $e$  in  $b$  setze  $HT[e] = \emptyset$

•  $HT[e] = \{v \mid v \in AVAIL(b)$

$\wedge v \text{ ist Wert von } e\}$

- 2. Wenn Wiederverwendung  $x := e$ , erkannt

• Ersetze  $x$  durch  $HT[e]$  mit beliebigen Namen  $v$

•  $HT[e] = \emptyset$

• Keine Wiederverwendung durch  $HT[e] = \emptyset$



## Block-lokales Value Numbering in Block $b$

### 1. Phase: Finde Wiederbenutzungen eines Ausdrucks

- 1 Initialisiere Hash-Tabelle mit  $AVAIL(b)$ 
  - Falls VN Versionsnummern benutzt:  
Ausdrücke aus  $AVAIL(b)$  umformen  
z.B.  $e + f \rightarrow e_0 + f_0$
- 2 Wenn Wiederverwendung  $x := e_i$  erkannt
  - Ersetze  $e$  durch Verweis auf bekannten Namen  $t_i$ :  
 $x := t_i$
  - Merke Wiederverwendung durch  $USED[e] := true$



## Block-lokales Value Numbering in Block $b$

### 1. Phase: Finde Wiederbenutzungen eines Ausdrucks

- 1 Initialisiere Hash-Tabelle mit  $AVAIL(b)$ 
  - Falls VN Versionsnummern benutzt:  
Ausdrücke aus  $AVAIL(b)$  umformen  
z.B.  $e + f \rightarrow e_0 + f_0$
- 2 Wenn Wiederverwendung  $x := e_i$  erkannt
  - Ersetze  $e$  durch Verweis auf bekannten Namen  $t_j$ :  
 $x := t_j$
  - Merke Wiederverwendung durch  $USED[e] := true$



## Block-lokales Value Numbering in Block $b$

### 1. Phase: Finde Wiederbenutzungen eines Ausdrucks

- 1 Initialisiere Hash-Tabelle mit  $AVAIL(b)$ 
  - Falls VN Versionsnummern benutzt:  
Ausdrücke aus  $AVAIL(b)$  umformen  
z.B.  $e + f \rightarrow e_0 + f_0$
- 2 Wenn Wiederverwendung  $\mathbf{x} := e_i$  erkannt
  - Ersetze  $e$  durch Verweis auf bekannten Namen  $t_i$ :  
 $\mathbf{x} := t_i$
  - Merke Wiederverwendung durch  $USED[e] := true$



## Block-lokales Value Numbering in Block $b$

### 1. Phase: Finde Wiederbenutzungen eines Ausdrucks

- 1 Initialisiere Hash-Tabelle mit  $AVAIL(b)$ 
  - Falls VN Versionsnummern benutzt:  
Ausdrücke aus  $AVAIL(b)$  umformen  
z.B.  $e + f \rightarrow e_0 + f_0$
- 2 Wenn Wiederverwendung  $\mathbf{x} := e_i$  erkannt
  - Ersetze  $e$  durch Verweis auf bekannten Namen  $t_i$ :  
 $\mathbf{x} := t_i$
  - Merke Wiederverwendung durch  $USED[e] := true$



## Block-lokales Value Numbering in Block $b$

### 1. Phase: Finde Wiederbenutzungen eines Ausdrucks

- 1 Initialisiere Hash-Tabelle mit  $AVAIL(b)$ 
  - Falls VN Versionsnummern benutzt:  
Ausdrücke aus  $AVAIL(b)$  umformen  
z.B.  $e + f \rightarrow e_0 + f_0$
- 2 Wenn Wiederverwendung  $\mathbf{x} := e_i$  erkannt
  - Ersetze  $e$  durch Verweis auf bekannten Namen  $t_i$ :  
 $\mathbf{x} := t_i$
  - Merke Wiederverwendung durch  $USED[e] := true$



## 2. Phase: Kopien der benutzten Werte unter bekanntem Namen erzeugen

- Für alle Ausdrücke  $e$  im Basisblock  $b$   
•  $W$  = Menge  $e \in \text{DEF}(b)$  und  $U = \text{USE}(e)$



## 2. Phase: Kopien der benutzten Werte unter bekanntem Namen erzeugen

- Für alle Ausdrücke  $e$  im Basisblock  $b$ 
  - Wenn  $e \in \text{DEEXPR}(b)$  und  $\text{USED}[e]$ 
    - Füge nach letzter Definition von  $e$  in  $b$  ein:  $t_i := e$





## 2. Phase: Kopien der benutzten Werte unter bekanntem Namen erzeugen

- Für alle Ausdrücke  $e$  im Basisblock  $b$ 
  - Wenn  $e \in \text{DEEXPR}(b)$  und  $\text{USED}[e]$ 
    - Füge nach letzter Definition von  $e$  in  $b$  ein:  $t_i := e$



## 2. Phase: Kopien der benutzten Werte unter bekanntem Namen erzeugen

- Für alle Ausdrücke  $e$  im Basisblock  $b$ 
  - Wenn  $e \in \text{DEEXPR}(b)$  und  $\text{USED}[e]$ 
    - Füge nach letzter Definition von  $e$  in  $b$  ein:  $t_i := e$



- **Lokale** Redundanzen durch lokales VN beseitigt
- **Globale** Redundanzen durch AVAIL-Mengen beseitigt
- Nicht ganz identischer Effekt, findet
  - Lokale Redundanzen durch Wertgleichheit
  - Globale Redundanzen durch gleiche Schreibweise



- **Lokale** Redundanzen durch lokales VN beseitigt
- **Globale** Redundanzen durch AVAIL-Mengen beseitigt
- Nicht ganz identischer Effekt, findet
  - Lokale Redundanzen durch Wertgleichheit
  - Globale Redundanzen durch gleiche Schreibweise



- **Lokale** Redundanzen durch lokales VN beseitigt
- **Globale** Redundanzen durch AVAIL-Mengen beseitigt
- Nicht ganz identischer Effekt, findet
  - Lokale Redundanzen durch Wertgleichheit
  - Globale Redundanzen durch gleiche Schreibweise

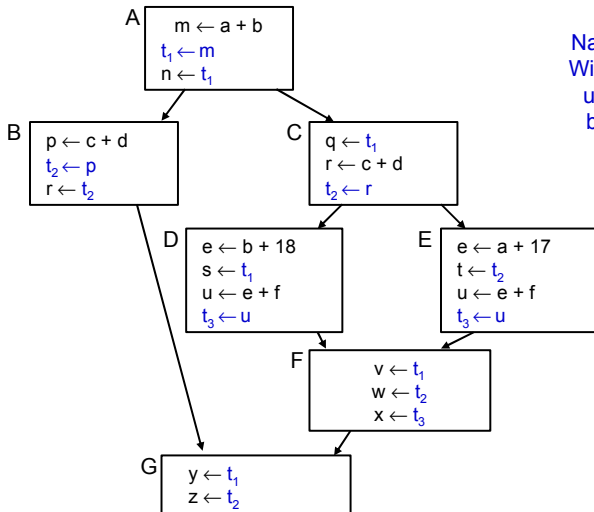


- **Lokale** Redundanzen durch lokales VN beseitigt
- **Globale** Redundanzen durch AVAIL-Mengen beseitigt
- Nicht ganz identischer Effekt, findet
  - Lokale Redundanzen durch Wertgleichheit
  - Globale Redundanzen durch gleiche Schreibweise



- **Lokale** Redundanzen durch lokales VN beseitigt
- **Globale** Redundanzen durch AVAIL-Mengen beseitigt
- Nicht ganz identischer Effekt, findet
  - Lokale Redundanzen durch Wertgleichheit
  - Globale Redundanzen durch gleiche Schreibweise

# Effekt von globalem CSE im Beispiel



Nach Auffinden von  
Wiederbenutzungen  
und Kopieren der  
benutzten Werte

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

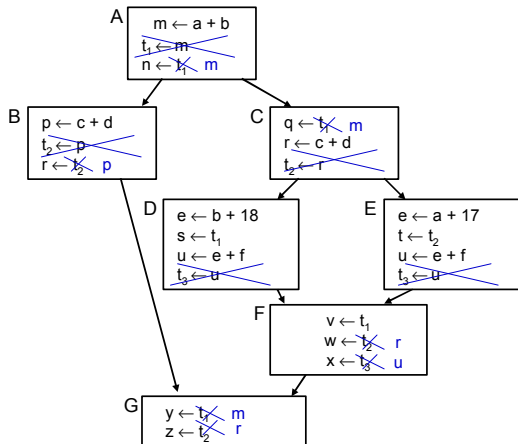
Superlokal

Regional

Global



# Unnötige Kopien



Viele der Kopien unnötig,  
hier sogar **alle**.

Können später aber entfernt  
werden (copy propagation,  
copy coalescing)

OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

# Gegenbeispiel



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

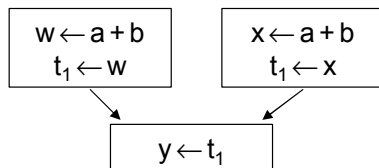
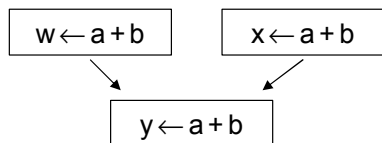
Lokal

Superlokal

Regional

Global

Hier wird Kopie gebraucht:



hier kann nicht "w oder x"  
geschrieben werden

Unterschied zu letztem Beispiel:

In beiden Zweigen nur **eine** Variable ( $u$ ).

# Gegenbeispiel



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

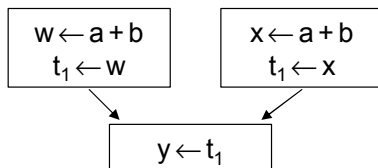
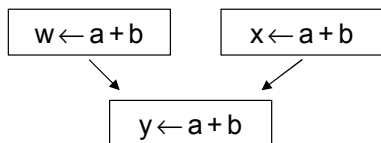
Lokal

Superlokal

Regional

Global

Hier wird Kopie gebraucht:



hier kann nicht "w oder x"  
geschrieben werden

Unterschied zu letztem Beispiel:

In beiden Zweigen nur **eine** Variable ( $u$ ).

# Gegenbeispiel



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

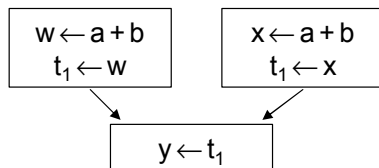
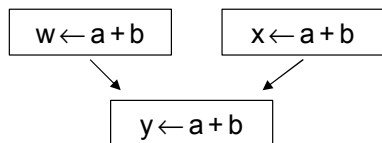
Lokal

Superlokal

Regional

Global

Hier wird Kopie gebraucht:



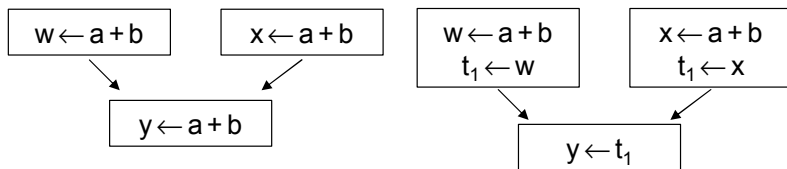
hier kann nicht "w oder x"  
geschrieben werden

Unterschied zu letztem Beispiel:

In beiden Zweigen nur **eine** Variable ( $u$ ).



Hier wird Kopie gebraucht:



hier kann nicht "w oder x"  
geschrieben werden

Unterschied zu letztem Beispiel:  
In beiden Zweigen nur **eine** Variable (u).

# Vergleich der Methoden 1



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

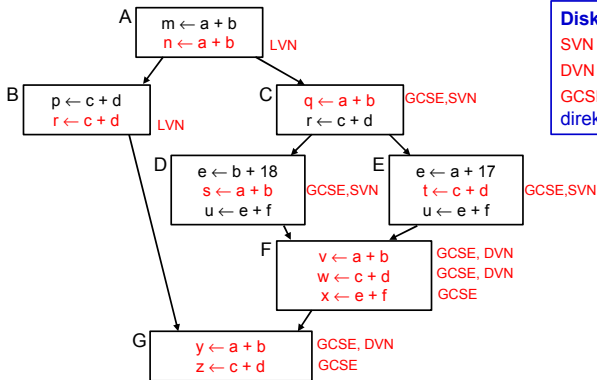
Wertnumerierung

Lokal

Superlokal

Regional

Global



## Diskussion:

SVN umfasst LVN

DVN umfasst SVN

GCSE & xVN sind nicht  
direkt vergleichbar

# Vergleich der Methoden 2



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

- GCSE ist **nicht** zwangsläufig die mächtigste Methode
- Hätte im Beispiel zwar alles gefunden
- Hat aber auch Schwächen gegenüber xVN
- Arbeit mit lexikalischem Vergleich
- Kann z.B. nicht erkennen:  
 $(a + b) = (c + d)$ , wenn  $a = d$  und  $b = c$
- GCSE versucht auf **lokaler** Ebene zu kompensieren
  - Verwendung von LVN innerhalb von Basisblöcken

# Vergleich der Methoden 2



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

- GCSE ist **nicht** zwangsläufig die mächtigste Methode
- Hätte im Beispiel zwar alles gefunden
- Hat aber auch Schwächen gegenüber xVN
- Arbeit mit lexikalischem Vergleich
- Kann z.B. nicht erkennen:  
 $(a + b) = (c + d)$ , wenn  $a = d$  und  $b = c$
- GCSE versucht auf **lokaler** Ebene zu kompensieren
  - Verwendung von LVN innerhalb von Basisblöcken



# Vergleich der Methoden 2



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

- GCSE ist **nicht** zwangsläufig die mächtigste Methode
- Hätte im Beispiel zwar alles gefunden
- Hat aber auch Schwächen gegenüber xVN
- Arbeit mit lexikalischem Vergleich
- Kann z.B. nicht erkennen:  
 $(a + b) = (c + d)$ , wenn  $a = d$  und  $b = c$
- GCSE versucht auf **lokaler** Ebene zu kompensieren
  - Verwendung von LVN innerhalb von Basisblöcken

# Vergleich der Methoden 2



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

- GCSE ist **nicht** zwangsläufig die mächtigste Methode
- Hätte im Beispiel zwar alles gefunden
- Hat aber auch Schwächen gegenüber xVN
- Arbeit mit lexikalischem Vergleich
- Kann z.B. nicht erkennen:  
 $(a + b) = (c + d)$ , wenn  $a = d$  und  $b = c$
- GCSE versucht auf **lokaler** Ebene zu kompensieren
  - Verwendung von LVN innerhalb von Basisblöcken

# Vergleich der Methoden 2



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

- GCSE ist **nicht** zwangsläufig die mächtigste Methode
- Hätte im Beispiel zwar alles gefunden
- Hat aber auch Schwächen gegenüber xVN
- Arbeit mit lexikalischem Vergleich
- Kann z.B. nicht erkennen:  
 $(a + b) = (c + d)$ , wenn  $a = d$  und  $b = c$
- GCSE versucht auf **lokaler** Ebene zu kompensieren
  - Verwendung von LVN innerhalb von Basisblöcken

# Vergleich der Methoden 2



OptComp

A. Koch

Orga

Intro

Redundanz

CFGs

Wertnumerierung

Lokal

Superlokal

Regional

Global

- GCSE ist **nicht** zwangsläufig die mächtigste Methode
- Hätte im Beispiel zwar alles gefunden
- Hat aber auch Schwächen gegenüber xVN
- Arbeit mit lexikalischem Vergleich
- Kann z.B. nicht erkennen:  
 $(a + b) = (c + d)$ , wenn  $a = d$  und  $b = c$
- GCSE versucht auf **lokaler** Ebene zu kompensieren
  - Verwendung von LVN innerhalb von Basisblöcken



- GCSE ist **nicht** zwangsläufig die mächtigste Methode
- Hätte im Beispiel zwar alles gefunden
- Hat aber auch Schwächen gegenüber xVN
- Arbeit mit lexikalischem Vergleich
- Kann z.B. nicht erkennen:  
 $(a + b) = (c + d)$ , wenn  $a = d$  und  $b = c$
- GCSE versucht auf **lokaler** Ebene zu kompensieren
  - Verwendung von LVN innerhalb von Basisblöcken



- **Charakteristika von Optimierungen**
- Redundante Ausdrücke
- Versionen von Variablen ( $\rightarrow$  SSA)
- Value Numbering
- Lokal, super-lokal, regional
- Dominatoren
- Global Common Subexpression Elimination
- Datenflußanalyse
- Vergleich der Techniken



- Charakteristika von Optimierungen
- Redundante Ausdrücke
- Versionen von Variablen ( $\rightarrow$  SSA)
- Value Numbering
- Lokal, super-lokal, regional
- Dominatoren
- Global Common Subexpression Elimination
- Datenflußanalyse
- Vergleich der Techniken



- Charakteristika von Optimierungen
- Redundante Ausdrücke
- Versionen von Variablen ( $\rightarrow$  SSA)
- Value Numbering
- Lokal, super-lokal, regional
- Dominatoren
- Global Common Subexpression Elimination
- Datenflußanalyse
- Vergleich der Techniken





- Charakteristika von Optimierungen
- Redundante Ausdrücke
- Versionen von Variablen ( $\rightarrow$  SSA)
- Value Numbering
- Lokal, super-lokal, regional
- Dominatoren
- Global Common Subexpression Elimination
- Datenflußanalyse
- Vergleich der Techniken



- Charakteristika von Optimierungen
- Redundante Ausdrücke
- Versionen von Variablen ( $\rightarrow$  SSA)
- Value Numbering
- Lokal, super-lokal, regional
- Dominatoren
- Global Common Subexpression Elimination
- Datenflußanalyse
- Vergleich der Techniken



- Charakteristika von Optimierungen
- Redundante Ausdrücke
- Versionen von Variablen ( $\rightarrow$  SSA)
- Value Numbering
- Lokal, super-lokal, regional
- Dominatoren
- Global Common Subexpression Elimination
- Datenflußanalyse
- Vergleich der Techniken



- Charakteristika von Optimierungen
- Redundante Ausdrücke
- Versionen von Variablen ( $\rightarrow$  SSA)
- Value Numbering
- Lokal, super-lokal, regional
- Dominatoren
- Global Common Subexpression Elimination
- Datenflußanalyse
- Vergleich der Techniken



- Charakteristika von Optimierungen
- Redundante Ausdrücke
- Versionen von Variablen ( $\rightarrow$  SSA)
- Value Numbering
- Lokal, super-lokal, regional
- Dominatoren
- Global Common Subexpression Elimination
- Datenflußanalyse
- Vergleich der Techniken



- Charakteristika von Optimierungen
- Redundante Ausdrücke
- Versionen von Variablen ( $\rightarrow$  SSA)
- Value Numbering
- Lokal, super-lokal, regional
- Dominatoren
- Global Common Subexpression Elimination
- Datenflußanalyse
- Vergleich der Techniken