Übersetzung und Analyse von Programmen

Vorlesung im Wintersemester 2025/26

Prof. Dr. Stephan Diehl
Informatik, Universität Trier

Portfolio
und
Projekt 5

Bildquelle: mit KI erzeugt (Bing, DALL-E)

Übersetzung und Analyse von Programmen

- **Inhalt:**
  - Semantik von Programmiersprachen
  - Überblick über verschiedene Phasen eines Compilers
  - Lexikalische und syntaktische Analyse
  - Statische Programmanalyse
    - insbesondere Datenflussanalyse
  - Übersetzung von Java
  - Abstrakte Maschinen insbesondere JVM
  - Hybride Analysen
    - Erkennung von Code-Klonen
    - Erkennung von Entwurfsmustern
    - Transpiler, Übersetzung mit LLMs
  - Programmtransformationen
    - Programmoptimierung
    - Partielle Auswertungen

Vorlesungsbegleitendes Projekt:
- **Implementierung der TRAM**
- **Übersetzung von TRIPLA**

Prüfungsleistung

- **Portfolio**
  - 5 aus 7 Übungen (10 Punkte pro Übungsblatt) — **50**
  - 5 Teilprojekte
    - TRAM (10)
    - Lexer+Parser (20)
    - Code-Erzeugung (25)
    - Kontrollflussanalyse (10)
    - Datenflussanalyse (25)

    — **90**
  - **Reflexion (kurzer Text):**
    - unbewertet, aber **zwingend notwendig** zum Bestehen der Portfolio-Prüfung
    - Kritische Betrachtung der **eigenen Lern- und Arbeitsweise**: Was haben Sie wie ich in der Vorlesung und im Abschlussprojekt gelernt? Was würden Sie anders machen? Eigene Schwächen und Stärken, Bedeutung des Gelernten

    —

  140

- **Abgabe des Portfolio bis zum 17.3.2026 in StudIP**

Bei Fragen/Problemen → kurzfristig Termin per Email vereinbaren

Aufgabe 1 : Datenflussanalyse für TRIPLA (16 Punkt(e))

Implementieren Sie die interprozedurale Datenflussanalyse der “Reached Uses” auf dem zuvor erzeugten Kontrollflussgraphen für TRIPLA. Überlegen Sie sich eine geeignete Ausgabe der Ergebnisse der Analyse entweder auf der Konsole oder in eine Datei.

Tipp: Da es sich dabei um eine Variante der “Live Variables”-Datenflussanalyse handelt, könnten Sie zuerst diese implementieren und daraufhin erweitern. Dies hätte den Vorteil, dass sich das Debuggen einfacher gestaltet, da die Datenstrukturen für die “Live Variables”-Datenflussanalyse strukturell einfacher sind.

Rückwärtsflussproblem: Reached Uses
(Erreichte Verwendungen)

$$KILL(v) = \begin{cases} \{(p', id) \mid p' \in P\} & \text{if } v = id{=}e^{\,p} \\ \{(p', id_1) \ldots, (p', id_k) \mid p' \in P\} & \text{if } v = (\mathsf{START}, f(id_1, \ldots, id_k)^p) \\ \{(p', id_1), \ldots, (p', id_k) \mid p' \in P\} & \text{if } v = (\mathsf{END}, f(id_1, \ldots, id_k)^p) \\ \emptyset & \text{otherwise} \end{cases}$$

$$GEN(v) = \begin{cases} \{(p, id)\} & \text{if } v = id^{\,p} \\ \emptyset & \text{otherwise} \end{cases}$$

```
for all v ∈ V do
    IN(v) := GEN(v)
while there are changes do
  for all v ∈ V do
      if v == (CALL, w) then
          ∃!ret ∈ succ(v) with ret = (RET, w)
          ∃!start ∈ succ(v) with start = (START, w')
          ∃!end ∈ pred(ret) with end = (END, w')
          Δ := (OUT(end) − OUT(start)) − KILL(start)
          IN(v) := (OUT(ret) ∪ IN(start)) − Δ
      else
          OUT(v) := ⋃_{s∈succ(v)} IN(s)
          IN(v) := (OUT(v) − KILL(v)) ∪ GEN(v)
```

Analog zu Lebendige Variablen

Rückwärtsflussproblem: Reached Uses
(Erreichte Verwendungen)

$$KILL(v) = \begin{cases} \{id\} & \text{if } v = id{=}e^{\,p} \\ \{id_1, \ldots, id_k\} & \text{if } v = (\mathsf{START}, f(id_1, \ldots, id_k)^p) \\ \{id_1, \ldots, id_k\} & \text{if } v = (\mathsf{END}, f(id_1, \ldots, id_k)^p) \\ \emptyset & \text{otherwise} \end{cases}$$

$$GEN(v) = \begin{cases} \{(p, id)\} & \text{if } v = id^{\,p} \\ \emptyset & \text{otherwise} \end{cases}$$

$$M \ominus K = \{(p, id) \mid (p, id) \in M \text{ and } id \notin K\}$$

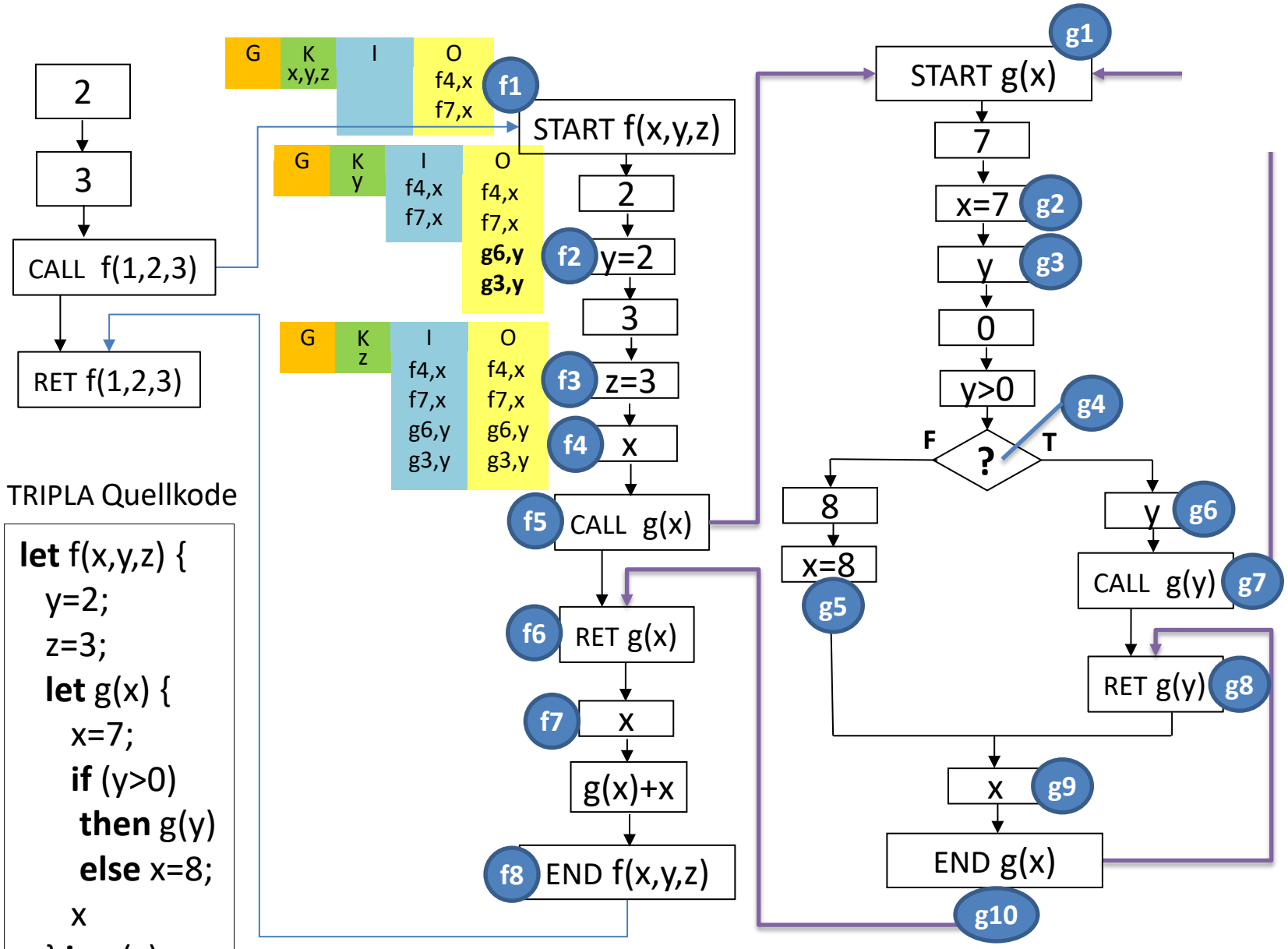
```
for all v ∈ V do
    IN(v) := GEN(v)
while there are changes do
  for all v ∈ V do
      if v == (CALL, w) then
          ∃!ret ∈ succ(v) with ret = (RET, w)
          ∃!start ∈ succ(v) with start = (START, w')
          ∃!end ∈ pred(ret) with end = (END, w')
          Δ := (OUT(end) − OUT(start)) ⊖ KILL(start)
          IN(v) := (OUT(ret) ∪ IN(start)) − Δ
      else
          OUT(v) := ⋃_{s∈succ(v)} IN(s)
          IN(v) := (OUT(v) ⊖ KILL(v)) ∪ GEN(v)
```

Analog zu Lebendige Variablen

Aufgabe 2 : Datenflussgraph (4 Punkt(e))

Fügen Sie zu ihrem Kontrollflussgraphen Datenflusskanten, also Kanten von Definitionen zu Verwendungen, hinzu. Erweitern Sie außerdem die graphische Darstellung des Graphen (DOT-Export) um diese Kanten, so dass sie sich optisch, bspw. durch unterschiedliche Farbgebung, von den anderen Kanten im Graphen unterscheiden.

Reached Uses



TRIPLA Quellkode

```
let f(x,y,z) {
  y=2;
  z=3;
  let g(x) {
    x=7;
    if (y>0)
     then g(y)
     else x=8;
    x
  } in g(x)+x
} in f(2,3)
```

Aufgabe 3 : Optimierung (5 Punkt(e))

Implementieren Sie in Ihrem Compiler eine Optimierung für Zuweisungen, die das Ergebnis der “Reached Uses”-Datenflussanalyse ausnutzt.

Optimierung

- Live Variables (und auch andere Datenflussanalysen)
  - Nur die Zuweisung selbst entfernen, nicht die Auswertung der rechten Seite
  - Es könnten Seiteneffekte bestehen

```
let f(x, y)
{
  let g(x)
  {
     y = x + 7
  } in x = g(5); y
} in f(1, 2)
```

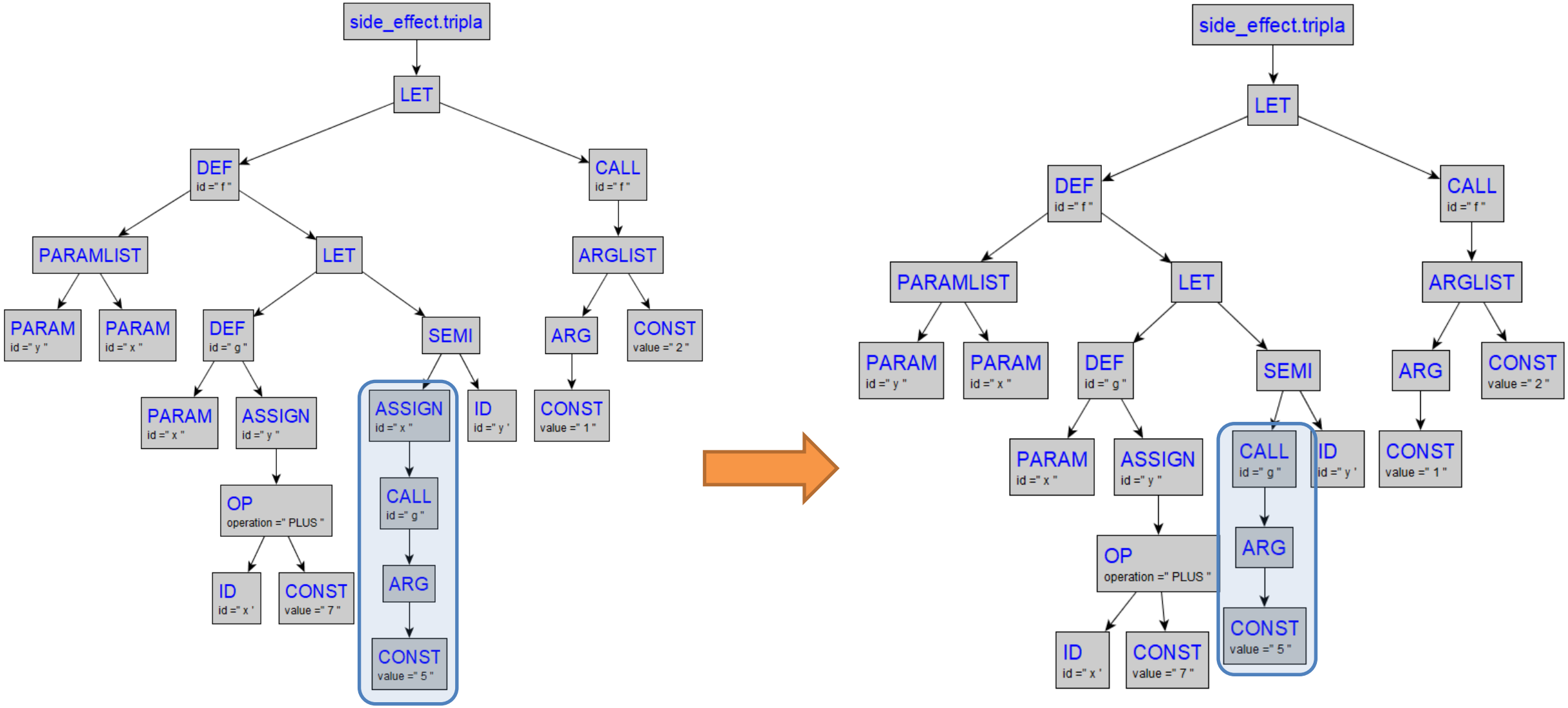
Seiteneffekt, dass `g(x)` aktiv `y` verändert!

Optimierung

⇒

```
let f(x, y)
{
  let g(x)
  {
     y = x + 7
  } in g(5); y
} in f(1, 2)
```

Optimierung im AST



Code without tests is like a ship without a compass