Karlsruher Institut für Technologie

Praktikum Compilerbau
Sitzung 10 – Codeerzeugung

Prof. Dr.-Ing. Gregor Snelting
Matthias Braun und Sebastian Buchwald

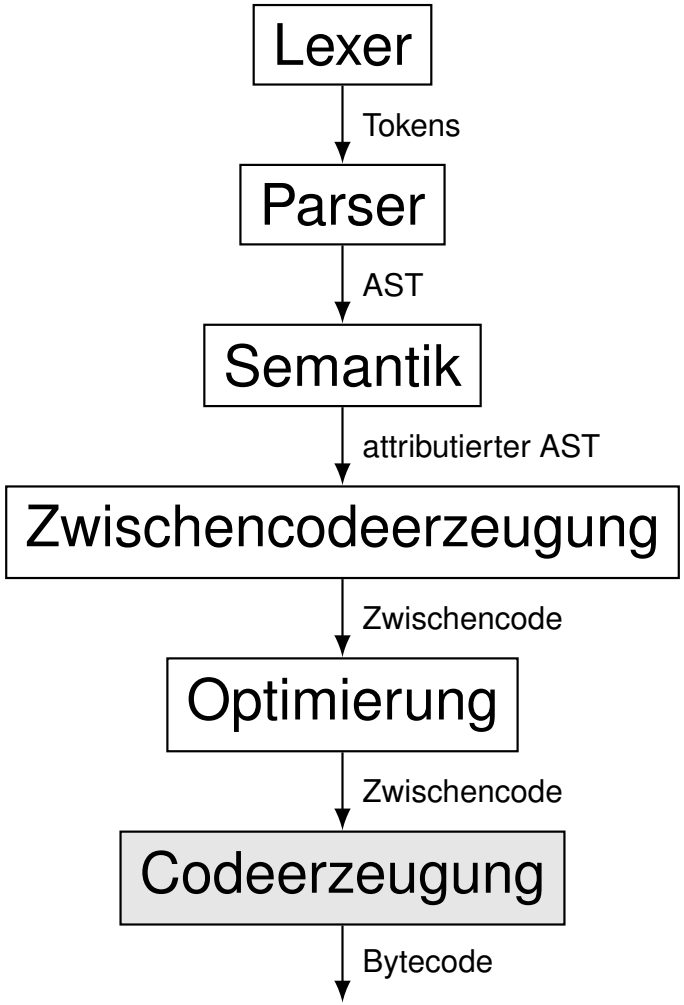
IPD Snelting, Lehrstuhl für Programmierparadigmen

1. Letzte Woche
2. Backends
3. Scheduling
4. Stackcode, Variablen, SSA-Darstellung
5. Codeausgabe, Backendschema
6. Sonstiges

Letzte Woche

- Was waren die Probleme?
- Hat soweit alles geklappt?

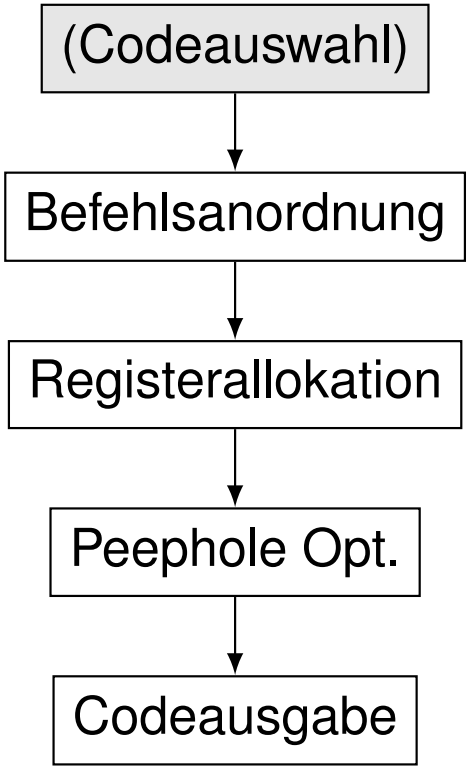
Compilerphasen



Aufbau eines Compilerbackends - Befehlsauswahl

- Abbilden von Befehlen der Zwischensprache auf Befehle der Zielmaschine. Meist werden *n* Zwischensprachbefehle auf einen Zielmaschinenbefehl abgebildet.
- Bei uns bis auf 1 Konstrukt eine 1 : 1 Abbildung. (bei welchem Konstrukt haben wir *n* : 1?)
- Deshalb bei uns keine separate Codeauswahlphase!

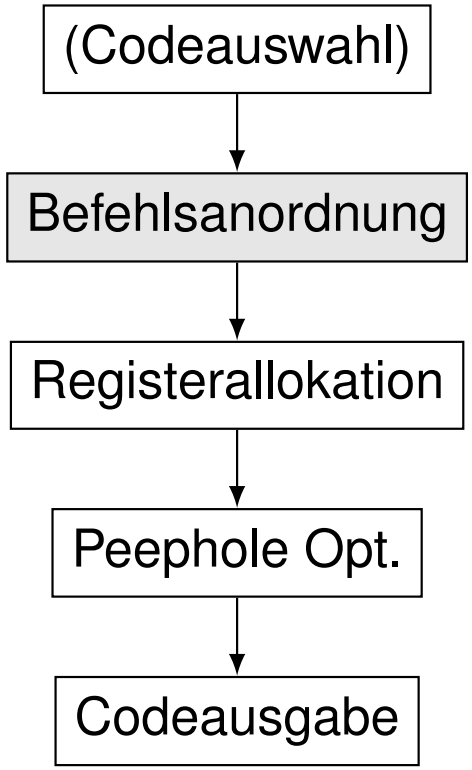
Compiler-Backend:



Aufbau eines Compilerbackends - Befehlsanordnung

- Bestimme Abhängigkeiten zwischen Befehlen und ordne diese neu an.
- Anordnungsziel: Minimiere Ressourcenbedarf und nutze Hardwareeigenschaften (Pipelining) aus.

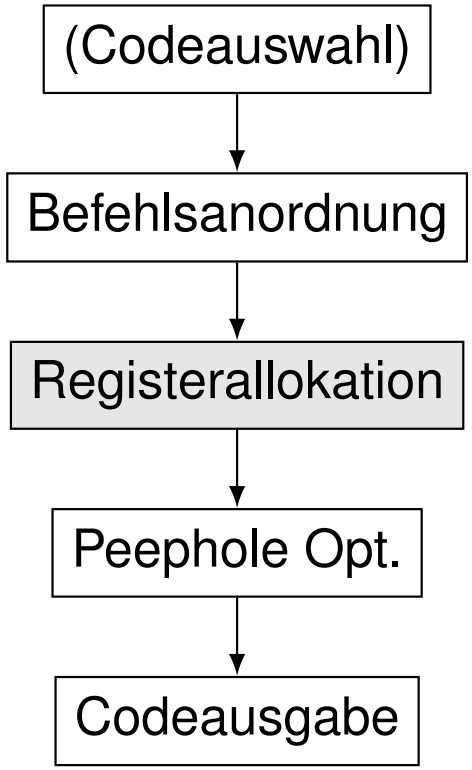
Compiler-Backend:



Aufbau eines Compilerbackends - Registerallokation

- Behandlung von Ressourcenbeschränkungen: Register, Stackframe, etc. zuteilen.
- Bei Registermangel Erzeugung von Auslagerungscode.
- Bei uns:
  - Soll Wert in Variable oder reicht Operandenstack?
  - Keine Ressourcenbeschränkungen: ausreichende Zahl Variablen

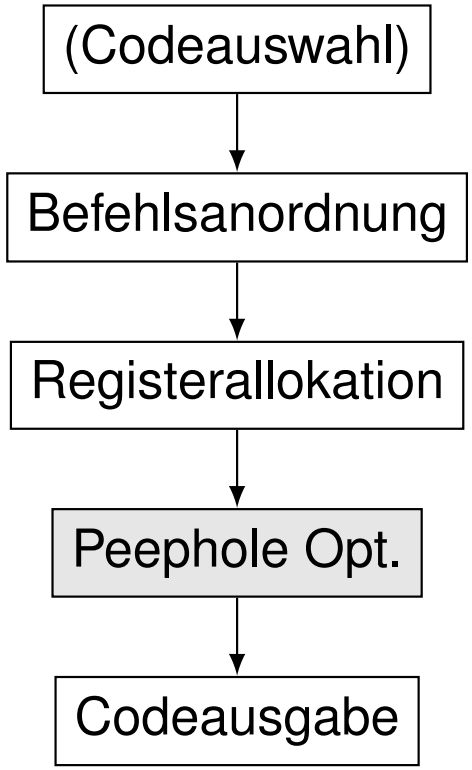
Compiler-Backend:



Aufbau eines Compilerbackends - Peephole Optimierungen

- Ersetze Muster von Zielsprachbefehlen durch billigere.
- Nicht immer in vorhergehenden Phasen möglich, da manche Optimierungen von konkreter Registerbelegung und Schedule abhängen.
- Bei uns nicht oder nur wenig nötig. Beispiele:
  - `Jmp L1; L1:` weglassen
  - `iconst 1` durch `iconst_1` ersetzen

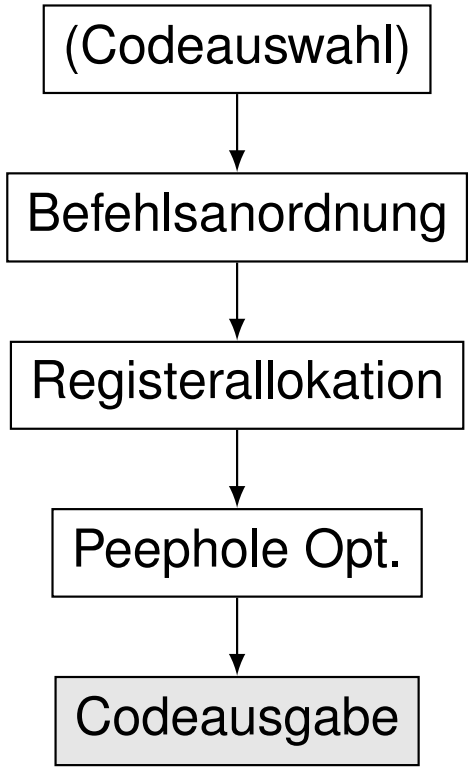
Compiler-Backend:



Aufbau eines Compilerbackends - Codeausgabe

- Ausgaben von Assembler oder direktes Erzeugen des Maschinencodes.
- Auflösen von Sprungmarken und Referenzen (beim direkten Erzeugen)

Compiler-Backend:



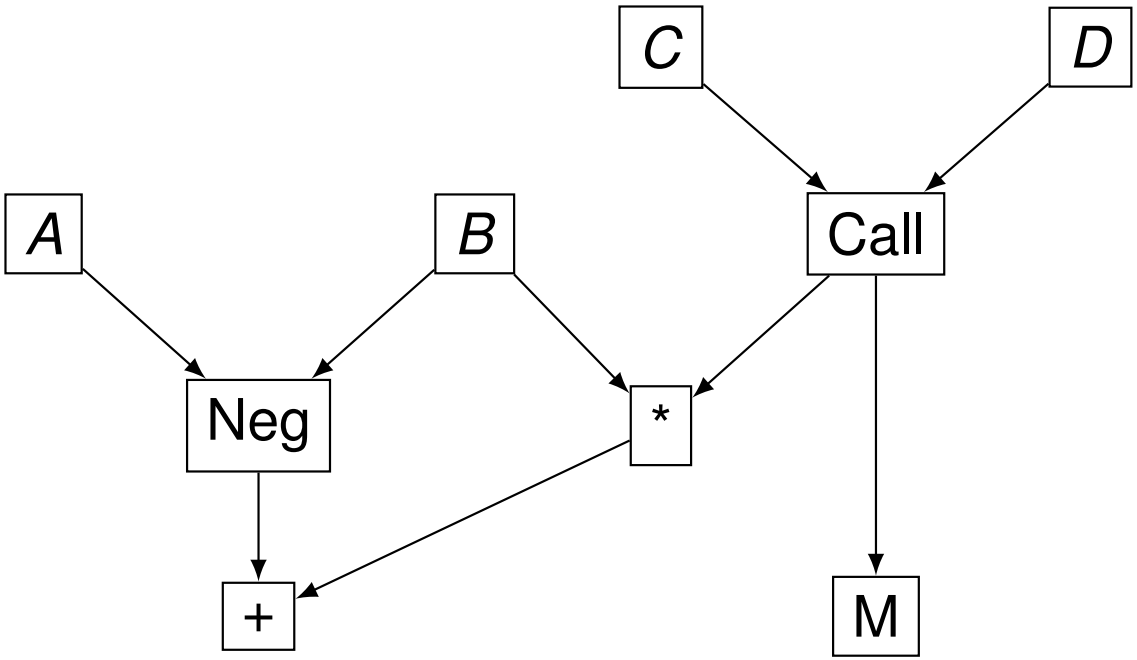
1. Letzte Woche
2. Backends
3. Scheduling
4. Stackcode, Variablen, SSA-Darstellung
5. Codeausgabe, Backendschema
6. Sonstiges

Befehlsreihenfolge in einem Grundblock

- Abhängigkeiten in Graph vorhanden
- Abhängigkeiten ergeben Halbordnung der Befehle
- Bilden einer Totalordnung nötig (*Topologisches Sortieren*)
- Die einfachste Möglichkeit für DAGs: Reverse Postorder (für alle Wurzeln).

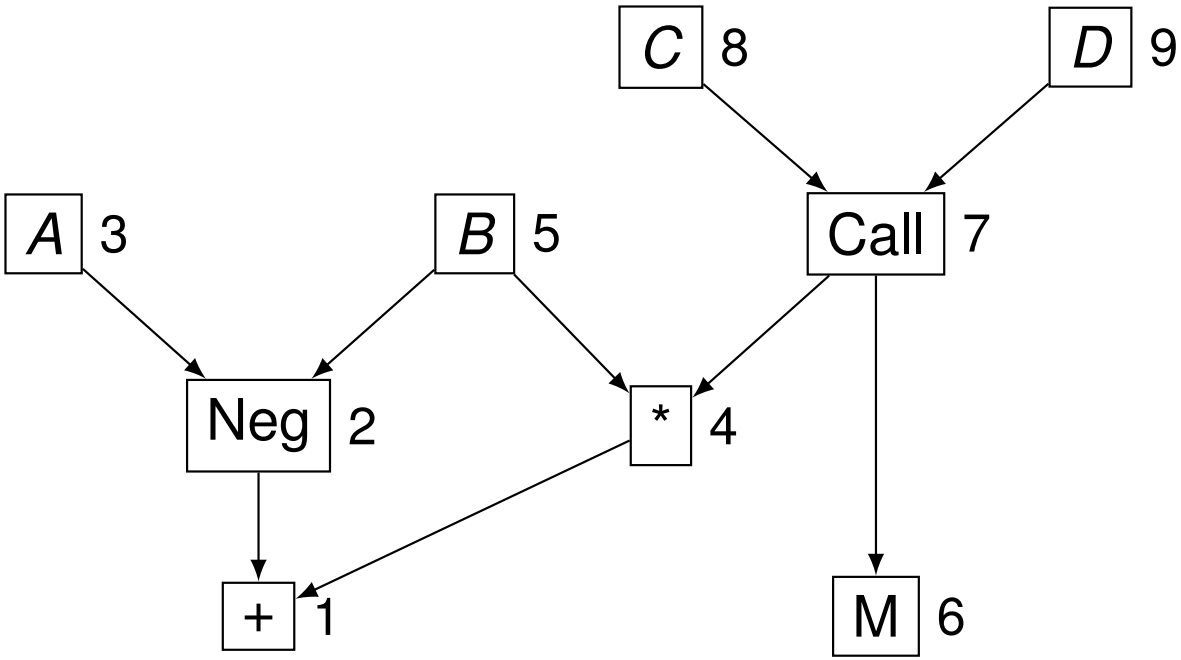
Reverse Postorder

- Für jede Wurzel Tiefensuche auf dem Graph, Nummern beim Verlassen von Knoten vergeben.



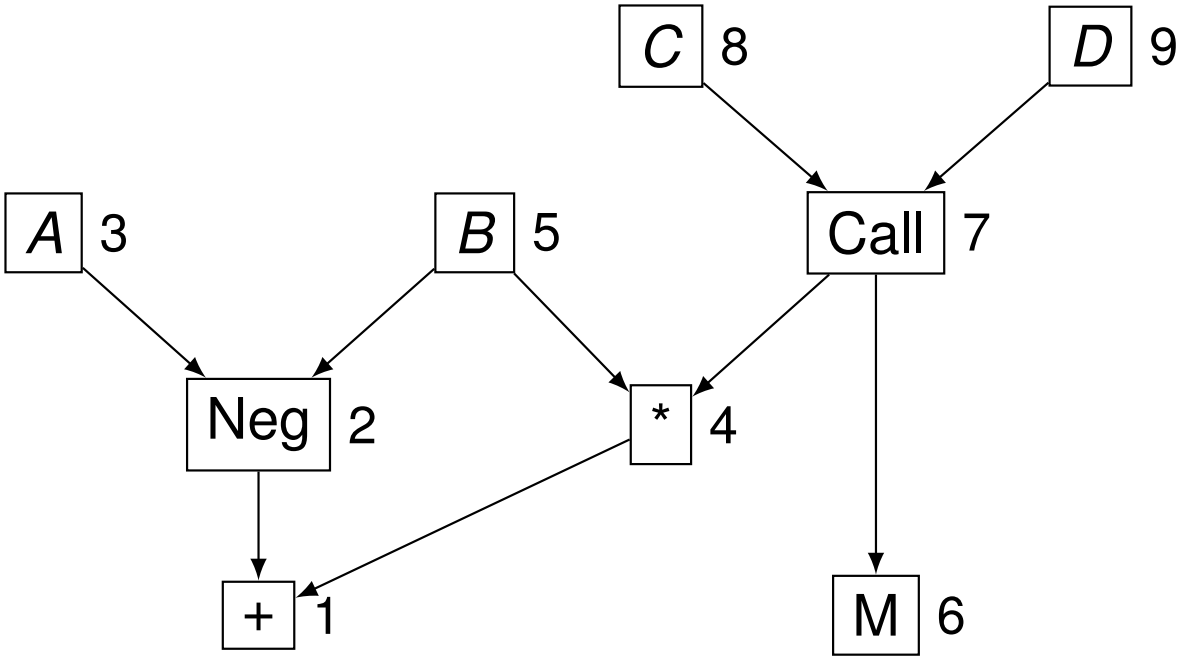
Reverse Postorder

- Für jede Wurzel Tiefensuche auf dem Graph, Nummern beim Verlassen von Knoten vergeben.



Reverse Postorder

- Für jede Wurzel Tiefensuche auf dem Graph, Nummern beim Verlassen von Knoten vergeben.



Reverse: *D*, *C*, Call, M, *B*, ∗, *A*, Neg, +

Reverse Postorder

- In JFirm: `Graph.walkTopological()`

Ressourcenverteilung

- Bytecode ist stackbasiert.
- Anordnung garantiert aber nicht, dass Operanden eines Befehls auf der Spitze des Stacks liegen. ⇒ Ausweichen auf Variablen.
- Naive Lösung: Eine Variable für jeden Firm-Knoten. Vor Operation Operanden aus Variablen laden, nach Operation Variable schreiben.

Verfeinerung: Erzeugen von Stackcode aus Bäumen

Karlsruher Institut für Technologie

Häufiger Fall: Berechnungen bilden einen Baum.

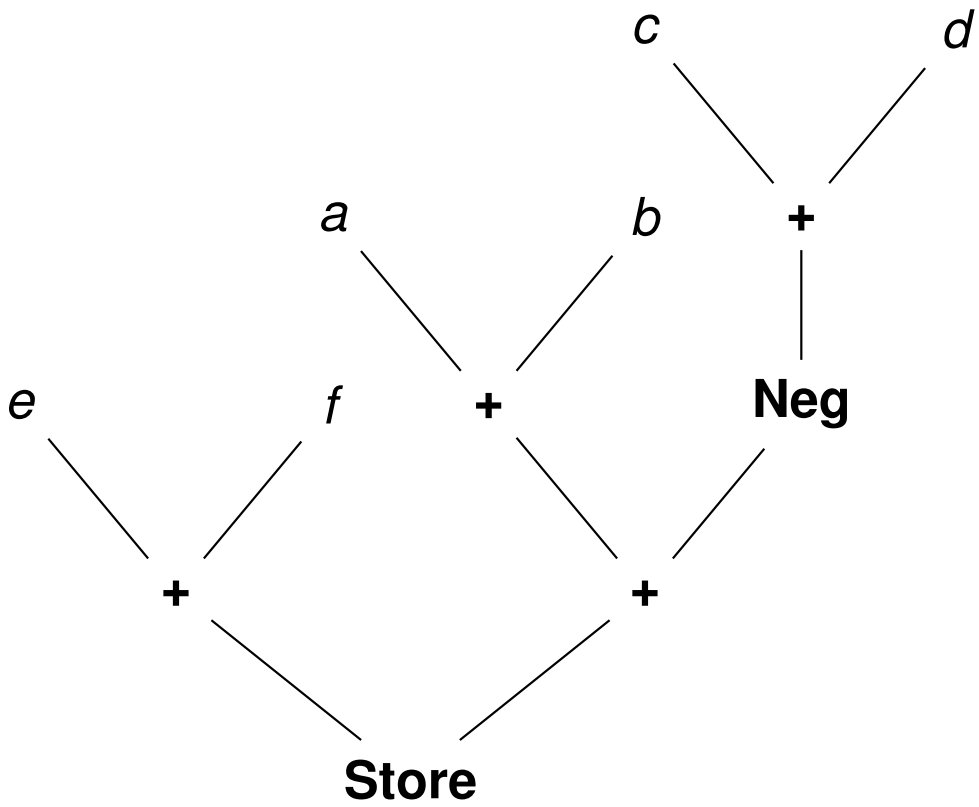
- Bei Codeerzeugung aus dem AST: Expressions
- In Firm: Induzierter Teilgraph bei dem jeder Wert nur einmal benutzt wird.

dann gilt:

- Stackcode ist genau für **Berechnungsbäume** effizient.
- Postfixform ist gültige Berechnungsvorschrift
- Erzeugen: Für jeden Knoten, rufe Erzeugerfunktion rekursiv für alle Kinder auf, erzeuge dann die Operation.

Beispiel: Stackcode aus Baum erzeugen

Beispiel:



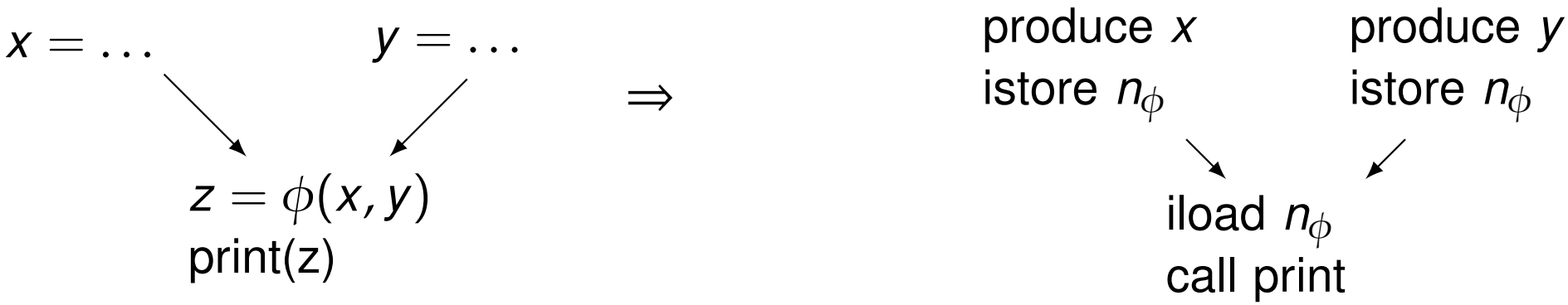
Schematisch:

e
f
add
a
b
add
c
d
add
neg
add
store

1. Letzte Woche
2. Backends
3. Scheduling
4. Stackcode, Variablen, SSA-Darstellung
5. Codeausgabe, Backendschema
6. Sonstiges

ϕ-Knoten Behandlung

- Teile Variablen für ϕ-Knoten zu.
- Am Ende eines Vorgängerblocks zu einem ϕ-Knotens speichere Argumente in die entsprechende Variable.



„Swap”-Problem

Achtung: Semantik von ϕ-Funktionen erfordert gleichzeitige Auswertung. Klassisches Beispiel:

x_0 = ...
y_0 = ...
while (**true**) {
  x_1 = phi(x_0, y_1)
  y_1 = phi(y_0, x_1)
  print(x_1, y_1)
}

Teilgraphen mit Baumstruktur

- DAGs haben aufspannende Bäume als Teilgraphen. („Bäume mit Querkanten ;-)”)
- Codeerzeugung also durch Tiefensuche/Postfixform mit Sonderbehandlung der Kanten die nicht zum Baum gehören.
- Diese Knoten haben mehrere Benutzer oder sind Wurzel eines Teilbaums. Variablen nur für Knoten an einer solchen Kanten zuweisen.

1. Letzte Woche
2. Backends
3. Scheduling
4. Stackcode, Variablen, SSA-Darstellung
5. Codeausgabe, Backendschema
6. Sonstiges

Vorgehen

- Suche alle Klassen im Programm:

```
for(Type type : Program.getTypes()) {
  if (! (type instanceof ClassType))
    continue; /* found a class */
}
```

- Pro Klasse:
  - Gib Klassenheader und Standardkonstruktor aus.
  - Gib alle Felder, dann alle Methoden aus:

```
for(Entity entity : classType.getMembers()) {
  if (entity.getType() instanceof MethodType) { continue; }
  emitField(entity);
}
for(Entity entity : classType.getMembers()) {
  if (! (entity.getType() instanceof MethodType)) { continue; }
  emitMethod(entity.getGraph());
}
```

Methoden: Erzeugen von Befehlslisten

Karlsruher Institut für Technologie

- Benutze `Graph.walkToplogical` um Graph in Reverse Postorder zu durchlaufen. (Firm-Graph ist bereits in umgekehrter Reihenfolge)
- Ordne Befehle dabei in Befehlslisten ein. Erzeuge eine *Liste pro Block*.
- *Achtung:* Sprungbefehle sind in Firm nicht geordnet, müssen bei der Ausgabe aber als Letztes im Grundblock erscheinen.

Methoden: Baumwurzeln markieren

Zuteilung von Variablennummern an:

- Alle Knoten mit mehreren (Daten-)Verwendern (`BackEdge.getNOuts(node)>1`) „Baumwurzeln”.
- Knoten mit Verwendern aus anderen Grundblöcken
- Paramter-Projs – Paramter liegen in (vorgegebenen) Variablen
- ϕ-Knoten benötigen eine Variable.
- Zuteilung an Rückgabewert von `Call`-Knoten (empfohlen)
- Vorsicht bei `Proj`-Knoten von `Call`/`Load`/`Store`: Diese erzeugen nicht wirklich Code/Werte. Zuteilung an den Vorgängerknoten und nicht an das `Proj`.

Schema createValue

```
/**
 * Wert berechnen und auf Spitze des Stacks legen.
 */
private void createValue(Node node) {
   switch (node.getOpCode()) {
   case iro_Const: /* ... */ break;
   case iro_Add:
      Add add = (Add) node;
      pushValue(add.getLeft());
      pushValue(add.getRight());
      println("\tiadd");
      break;
   }
}
```

Schema pushValue

```
/**
 * Wert auf Spitze des Stacks legen.
 * Falls bereits berechnet, aus Variable laden.
 */
private void pushValue(Node node) {
   if (varAssigned(node)) {
      /* emit aload/iload varnumber */
      return;
   }
   createValue(node);
}
```

Ausgabe

Karlsruher Institut für Technologie

Für jeden Grundblock:

- `BlockXX:` ausgeben
- Befehlsliste durchgehen. Für folgende Knoten Code ausgeben:
  - Für Store:

    ```
    pushValue(skipSel(store.getPtr()));
    pushValue(store.getValue());
    printf("putfield %s\n", getFieldSpec(store.getPtr()));
    ```

  - Knoten mit Variablennummer (außer ϕ, Parameter-`Proj`):

    ```
    createValue(node);
    String mode = node.getMode().isReference() ? "a" : "i";
    printf("%sstore %d\n", mode, varnum);
    ```

  - `Call`-Befehle
  - Sprungbefehle

1. Letzte Woche
2. Backends
3. Scheduling
4. Stackcode, Variablen, SSA-Darstellung
5. Codeausgabe, Backendschema
6. Sonstiges

Feedback! Fragen? Probleme?

Karlsruher Institut für Technologie

- Anmerkungen?
- Probleme?
- Fragen?