Aktivitätsanalyse (Liveness Analysis)

Aktivitätsanalyse (Liveness Analysis)

Für eine gute Registerverteilung müssen wir an jedem Programmpunkt wissen, welche Temporaries noch gespeichert werden müssen und welche nicht mehr benötigt werden.

Diese Information berechnen wir durch Aktivitätsanalyse (Liveness Analysis).

Es handelt sich dabei eine *Datenflussanalyse*.
Andere Datenflussanalysen werden zur Programmoptimierung eingesetzt (z.B. Constant Propagation, Common Subexpression Elimination, . . .) und funktionieren ganz ähnlich.

Aktivitätsanalyse (Liveness Analysis)

Wir führen eine Aktivitätsanalyse durch, um festzustellen, welches Temporary an welchen Programmpunkten aktiv (*live*) ist.

- Aufstellen eines Kontrollflussgraphen (ähnlich wie Flussdiagramm).
- Temporary ist in einem Knoten aktiv, wenn sein aktueller Wert später gebraucht wird (genauer: werden könnte).
- Temporaries, die nie gleichzeitig aktiv sind, können auf ein einziges physikalisches Register abgebildet werden.
- Bedingungen an die Vertauschbarkeit von Instruktionen können abgelesen werden.

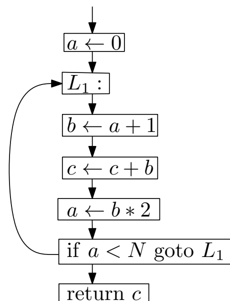
Kontrollflussgraph

Ein Kontrollflussgraph erfasst alle möglichen Läufe eines Programms.

Jede Instruktion im gegebenen Programm entspricht einem Knoten des Kontrollflussgraphen.

Für jeden Lauf des Programms gibt es einen *Pfad* im Kontrollflussgraphen. Die Umkehrung muss nicht gelten, der Kontrollflussgraph muss das Programmverhalten nur approximieren.

$a \leftarrow 0$
L_1 :
$b \leftarrow a + 1$
$c \leftarrow c + b$
$a \leftarrow b * 2$
if $a < N$ goto L_1
return c



Datenstruktur für Graphen

Für unsere Zwecke genügt eine sehr einfache Graphrepräsentation mit folgender Schnittstelle.

```
public class SimpleGraph<NodeInfo> {
  public Set<Node> nodeSet();
  public Node addNewNode(NodeInfo info);
  public void addEdge(Node src, Node dest);
  public void removeEdge(Node src, Node dest);

  public class Node {
    public NodeInfo  info();
    public Set<Node> successors();
    public Set<Node> predecessors();
    public Set<Node> neighbours();
    public int inDegree();
    public int outDegree();
    public int degree();
  }
}
```

Kontrollflussgraph

Der Kontrollflussgraph ist ein Graph, dessen Knoten mit Instruktionen beschriftet sind (`SimpleGraph<Assem>`).

Das Aufstellen des Kontrollflussgraphen bewerkstelligt eine Methode, die aus einer Liste von Instruktionen solch einen Graphen berechnet.

Dazu werden die folgenden Funktionen des Interface `Assem` benutzt:

- `boolean isFallThrough()`: Wenn diese Funktion `true` zurückgibt, dann wird eine Kante zur nachfolgenden Instruktion eingefügt.
- `List<Label> jumps()`: Füge Kanten zu allen von dieser Funktion zurückgegebenen Sprungzielen ein.
- `Label isLabel()`: Mögliche Sprungziele können mit dieser Funktion erkannt werden.

Aktivität (Liveness)

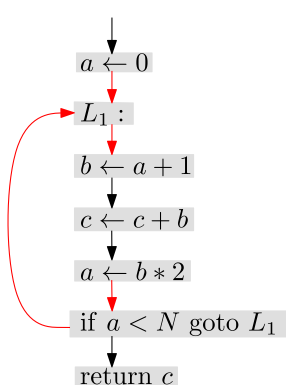
Ein Temporary t wird in einem Knoten n des Kontrollflussgraphen *definiert* (def), wenn die dort stehende Instruktion einen Wert in das Temporary t schreibt.

Ein Temporary t wird in einem Knoten n des Kontrollflussgraphen *benutzt* (use), wenn die dort stehende Instruktion das Temporary t liest.

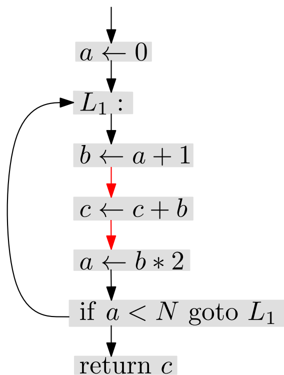
In unserem Fall (`SimpleGraph<Assem>`) sind die definierten und benutzten Temporaries durch die Methoden `Assem.def()` und `Assem.use()` erfasst.

Ein Temporary t ist entlang einer Kante (m, n) des Kontrollflussgraphen aktiv (*live*), wenn es einen Pfad im Kontrollflussgraphen $n \to n_1 \to \cdots \to n_k$ $(k \geq 0)$ gibt, der in einer Benutzung von t endet, ohne aber eine Definition von t zu enthalten (d.h. keine der Instruktionen $n, n_1, \ldots, n_k$ definiert t).

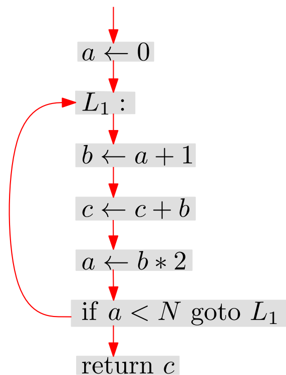
Beispiel



Aktivität von a



Aktivität von b



Aktivität von c

Aktivität in Knoten

Ein Temporary r ist in einem Knoten n des Kontrollflussgraphen *ausgangsaktiv* (*live-out*), wenn n eine ausgehende Kante hat, entlang der r aktiv ist.

Ein Temporary r ist in einem Knoten n des Kontrollflussgraphen *eingangsaktiv* (*live-in*), wenn r entlang einer beliebigen, den Knoten n erreichenden Kante aktiv ist.

Berechnung der Aktivität

Man kann für jedes Temporary r die Menge derjenigen Knoten, in denen es *ausgangsaktiv* ist, direkt aus der Definition berechnen.

Hierzu verfolgt man, ausgehend von einer Benutzung von r, die Kanten im Kontrollflussgraphen *rückwärts* und nimmt alle besuchten Knoten hinzu, bis man schließlich auf eine Definition von r stößt.

Für diese Rückwärtsverfolgung lässt sich die Tiefensuche verwenden.

Natürlich muss dies für jede Benutzung von r separat gemacht werden, wodurch sich möglicherweise ein recht hoher Aufwand ergibt.

Eine Alternative liegt in der gleichzeitigen Berechnung der eingangs- und ausgangsaktiven Temporaries für jeden Knoten.

Berechnung der Aktivität

Man kann für jeden Knoten die Menge der eingangs- und ausgangsaktiven Temporaries aufgrund folgender Beobachtungen iterativ berechnen:

- Wird ein Temporary r in n *benutzt*, so ist es in n eingangsaktiv.
- Ist r in einem Nachfolger von n eingangsaktiv, so ist es in n selbst ausgangsaktiv.
- Ist r ausgangsaktiv in n und wird r in n nicht definiert, so ist r auch eingangsaktiv in n.

In Formeln:

$$out[n] = \bigcup_{s \in succ[n]} in[s]$$
$$in[n] = use[n] \cup (out[n] \setminus def[n])$$

Die $in[n]$ und $out[n]$ Mengen bilden die kleinste Lösung dieser Gleichung und können mit deren Hilfe iterativ berechnet werden:

Algorithmus

$$
\begin{array}{l}
\text{for each } n \quad in[n] \leftarrow \emptyset; \quad out[n] \leftarrow \emptyset; \\
\text{repeat} \\
\quad \text{for each } n \\
\qquad in'[n] \leftarrow in[n]; \qquad out'[n] \leftarrow out[n]; \\
\qquad out[n] \leftarrow \bigcup_{s \in succ[n]} in[s]; \\
\qquad in[n] \leftarrow use[n] \cup (out[n] \setminus def[n]); \\
\text{until } in'[n] = in[n] \text{ and } out'[n] = out[n] \text{ for all } n
\end{array}
$$

Es empfiehlt sich, die „for each" Schleifen so zu durchlaufen, dass Nachfolgerknoten möglichst vor ihren Vorgängern verarbeitet werden. Durch Tiefensuche im umgedrehten Kontrollflussgraphen kann solch eine Reihenfolge bestimmt werden.

Interferenzgraph

Aus der Aktivitätsinformation wird ein Interferenzgraph berechnet.

Zwei Temporaries t_1, t_2 *interferieren*, wenn es nicht zulässig ist, sie auf ein und dasselbe physikalische Register abzubilden.

Gründe für Interferenz:

1. t_1 wird in einem Knoten n definiert und gleichzeitig ist t_2 in n ausgangsaktiv.
2. t_1 ist ein Maschinenregister und t_2 darf aus anderen Gründen nicht auf t_1 abgebildet werden (z.B. weil eine Maschineninstruktion ihr Ergebnis nur in bestimmten Registern liefern kann).

Von der ersten Bedingung gibt es noch eine wichtige Ausnahme: t_1 und t_2 interferieren nicht, wenn ihre Aktivitätsbereiche *nur* wegen einer Verschiebeinstruktion $t_1 \leftarrow t_2$ oder $t_2 \leftarrow t_1$ überlappen. Dann kann man sie nämlich doch zusammenfassen und die Verschiebeinstruktion entfernen!

Die zweite Bedingung ist für uns weniger relevant.

Algorithmus

Diese Betrachtungen führen auf folgenden Algorithmus:

for each n
  if *n enthält keine Verschiebeinstruktion*
    for each $t \in \mathit{def}[n]$
      for each $u \in \mathit{out}[n]$
        füge Interferenzkante (t, u) *ein*
  else if *n enthält Verschiebeinstruktion* $t \leftarrow v$
      for each $u \in \mathit{out}[n], \; u \neq v$
        füge Interferenzkante (t, u) *ein*

Temp. t inferiert nicht mit v; beide könnten auf das selbe Register abgebildet werden – dann fällt die Verschiebeinstruktion weg.

Programmieraufgabe

Für die Registerverteilung benötigen wir einen Interferenzgraphen für jede Prozedur. Dieser soll in drei Schritten berechnet werden:

1. Berechnung eines Kontrollflussgraphen
2. Berechnung der Aktivitätsmengen
3. Berechnung des Interferenzgraphen

Zur Implementierung dieser Schritte müssen die Funktionen `isFallThrough`, `isLabel`, `jumps`, `def`, `use` und `isMoveBetweenTemps` in allen Klassen, die `Assem` implementieren, vervollständigt werden.