Static-Single-Assignment-Form

Static-Single-Assignment-Form

- besondere Darstellungsform des Programms (Zwischensprache)
- vereinfacht Datenflussanalyse und damit Optimierungstechniken des Compilers
- ist in vielen Compilern inzwischen Standard
- wird benötigt, um das MiniJava-Front-End an die LLVM-Compiler-Infrastruktur anzuschließen

Static-Single-Assignment-Form

Ein Programm ist in *SSA-Form*, wenn jede Variable nur an einer Stelle im Programm definiert wird.

- Diese Definitionsstelle kann aber beliebig oft durchlaufen werden.
- Jede Variable kann beliebig oft benutzt werden. Jede Variable wird vor ihrer ersten Benutzung definiert.

Bei Programmen in SSA-Form kann jede Benutzung einer Variablen genau einer Definition zugeordnet werden.

Vorteile

Vereinfacht Datenfluss- und Optimisierungs-Techniken deutlich:

- Jeder Variablen kann man direkt ansehen, wo sie definiert wurde. Ohne SSA-Form muss man dazu erst eine Programmanalyse (*Reaching Definitions*) durchführen.
- vereinfacht z.B. *constant propagation*:
  Entferne Zuweisungen der Form $x_{14} \leftarrow 4$
  und ersetze alle Vorkommen von x_{14} durch 4.
- oder *dead code elimination*:
  Entferne Zuweisungen der Form $x_{14} \leftarrow a \ op \ b$,
  wenn x_{14} sonst nicht vorkommt.
- SSA-Form muss nur einmal berechnet werden

Vorteile

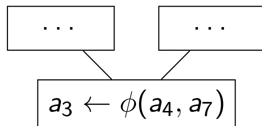
Die Benutzung der SSA-Form ist wegen seiner Vorteile weit verbreitet, z.B. in GCC, LLVM, JITs in HotSpot oder V8, ...

Betrachte LLVM als Beispiel:

- LLVM ist eine flexible Compiler-Infrastruktur, die LLVM-Assembler-Code mit verschiedenen Techniken optimiert und in verschiedene Zielplattformen übersetzt.
- LLVM kann Code-Optimierung und -Generierung im MiniJava-Compiler übernehmen, so dass wir uns ganz auf das Front-End konzentrieren können.
- LLVM-Assemblercode muss in SSA-Form sein.

Umwandlung in SSA-Form

Jedes Programm kann in SSA-Form gebracht werden, wenn auf sogenannte ϕ-Knoten zurückgegriffen wird. Ein solcher ϕ-Knoten definiert eine Variable in Abhängigkeit von der Stelle, von der der ϕ-Knoten erreicht wurde.



Ein Knoten $a_3 \leftarrow \phi(a_4, a_7)$ im Kontrollflussgraph entspricht $a_3 \leftarrow a_4$, wenn sie vom ersten Vorgänger erreicht wurde, und $a_3 \leftarrow a_7$, wenn sie vom zweiten Vorgänger erreicht wurde. Eine ϕ-Funktion hat immer so viele Argumente wie Vorgängerknoten.

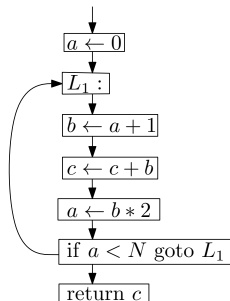
Kontrollflussgraph

Ein Kontrollflussgraph erfasst alle möglichen Läufe eines Programms.

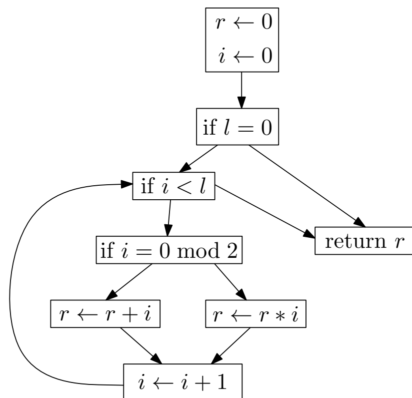
Jede Instruktion im gegebenen Programm entspricht einem Knoten des Kontrollflussgraphen.

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

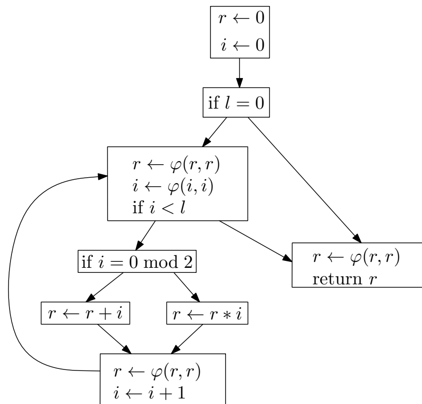


Umwandlung in SSA-Form



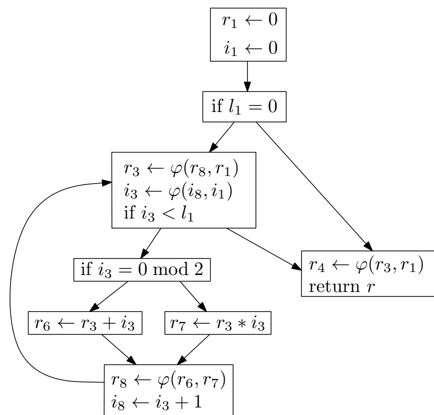
1. Kontrollflussgraph berechnen

Umwandlung in SSA-Form



2. Instruktionen der Form $a \leftarrow \phi(a, a)$ an Zusammenflüssen einfügen, wenn a abhängig vom Pfad verschiedene Werte haben kann

Umwandlung in SSA-Form



3. Für jede Definition von a neue Variable a_i verwenden und Benutzungen von a umbenennen

Umwandlung eines Programms in SSA-Form

Wir gehen im Folgenden von einem Kontrollflussgraphen aus, der einen eindeutigen Anfangsknoten s_0 hat. Außerdem werden der Einfachheit halber gleich am Anfang alle Variablen pseudo-definiert (z.B. mit $a \leftarrow \mathit{uninitialized}$).

Ein *Join-Knoten* im Kontrollflussgraph ist ein Knoten, der mehr als einen Vorgänger hat, an dem also zwei Pfade zusammenlaufen.

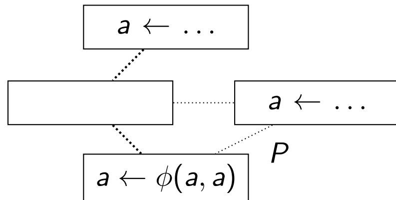
Einfügen von ϕ-Knoten

Am einfachsten ist es, an allen Join-Punkten ϕ-Knoten der Form $v \leftarrow \phi(v, v)$ für alle Variablen v einzufügen, und dann die Variablen in ihre jeweiligen Versionen umzubenennen.

Dadurch werden im Allgemeinen aber viel zu viele ϕ-Knoten eingefügt, was die auf SSA-Form aufbauenden Algorithmen wieder verlangsamt.

Intelligenteres Einfügen von ϕ-Knoten

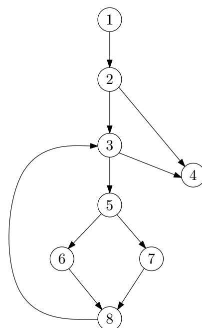
Stattdessen sollte ein ϕ-Knoten für v genau dann an einem Join-Punkt eingefügt werden, wenn es einen Pfad P von einer Definition von v zu dem Join-Punkt gibt, der nicht notwendigerweise durchlaufen werden muss, d.h. wenn ein von P disjunkter Pfad von s_0 zu dem Join-Punkt existiert.

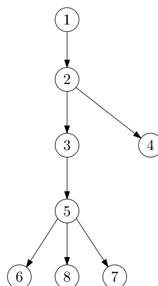


Solche Join-Punkte können mit der *Dominanzrelation* auf dem Kontrollflussgraphen ermittelt werden.

Dominanzknoten

Ein Knoten *d* *dominiert* einen Knoten *n*, wenn jeder Pfad von s_0 nach *n* durch *d* läuft. Jeder Knoten dominiert sich selbst.





d dominiert *n*, wenn ein Pfad von *d* zu *n* existiert. Jeder Knoten hat höchstens einen *direkten* Dominanzknoten (immediate dominator), weshalb man von einem *Dominanzbaum* spricht.

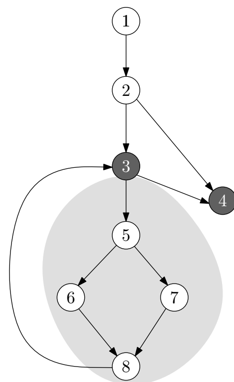
Dominanzgrenze

Die *Dominanzgrenze* (dominance frontier) eines Knotens x besteht aus allen Knoten w, für die *eine* der folgenden beide Eigenschaften gilt:

- x dominiert einen Vorgänger von w, nicht aber w selbst.
- $w = x$ und x dominiert einen Vorgänger von w.

Intuitiv heißt das, dass man von s_0 zu w nicht notwendigerweise über einen Pfad von x nach w muss (und w der „erste“ Knoten auf dem Pfad mit dieser Eigenschaft ist).

Die Dominanzgrenze kann man als „Grenze“ zwischen dominierten und nicht-dominierten Knoten verstehen.



Dominanzgrenze

Liegt w in der Dominanzgrenze von x, gibt es disjunkte Pfade von s_0 zu w und von x zu w.

Wenn also Knoten x eine Definition von a ist, muss an jedem Knoten w in der Dominanzgrenze von x ein ϕ-Knoten $a \leftarrow \phi(a, a, ..., a)$ eingefügt werden. (Dieser ϕ-Knoten ist selbst wieder eine Definition von a, weshalb der Algorithmus iterativ durchgeführt werden muss.)

Umbenennung

Umbenennung: Wenn v in Knoten n definiert wird, verwende einen neuen Namen v_n in der Definition. Wenn v in Knoten n benutzt wird, bennene v in v_u um, wobei u die unmittelbar vorangehende Definition von v ist (d.h. die *reaching definition* von v an der Stelle u).

Im Buch werden verschiedene effiziente Algorithmen zur Berechnung von Dominanzrelation und -grenzen sowie zur Umbennenung der Variablen vorgestellt, auf die hier nicht weiter eingangen wird.

Rückumwandlung in ausführbaren Code

Nach der Optimierung können wir das Programm wieder in ein ausführbares Programm umwandeln, indem die entsprechenden Zuweisungsoperationen hinter den Vorgängern eingefügt werden und die ϕ-Knoten entfernt werden.

LLVM

- Assemblersprache hauptsächlich als Zwischensprache für C- und C++-Compiler entwickelt
  - enthält auch komplexere Datenstrukturen wie Arrays und Verbünde (structs), mit entsprechenden Zugriffsmethoden
- LLVM-Compiler führt Optimierungen darauf durch und generiert Code für verschiedene Zielarchitekturen
  - z.B. x86, x86-64, PowerPC, ARM, Sparc, C, ...
- Ziel: Modularisierung von wiederkehrenden Optimierungs- und Generierungsaufgaben
  - Optimierungsphasen und Zielarchitekturen können flexibel hinzgefügt werden
  - Compiler für neue Sprachen können sich auf das Front-End konzentrieren, wenn sie nach LLVM übersetzen
- `www.llvm.org`

LLVM Assemblercode

In SSA-Form, stark getypt, beliebig viele Register

```
define i32 @factorial(i32 %X) {
entry:
        %0 = add i32 1, 0
        %1 = icmp sgt i32 %X, 0
        br i1 %1, label %rec_call, label %exit
rec_call:
        %2 = sub i32 %X, 1
        %3 = call i32 @factorial(i32 %2)
        %4 = mul i32 %X, %3
        br label %exit
exit:
        %5 = phi i32 [ %4, %rec_call ], [ %0, %entry ]
        ret i32 %5
}
```

LLVM im MiniJava-Compiler

Aus Sicht unseres MiniJava-Compilers kann LLVM als weitere Zielarchitektur eingebettet werden, mit eigener Frame-Klasse, Code-Generierung, usw. Dazu muss vor oder während der Code-Generierung das Programm in SSA-Form umgewandelt werden. Außerdem muss die MiniJava-Zwischensprache um Typinformationen erweitert werden, weil diese im LLVM-Code zwingend benötigt wird.

Alternativ kann man auch gleich in der Translate-Phase den abstrakten Syntaxbaum von Java in LLVM-Code übersetzen. Hierbei ist zu überlegen, welche LLVM-Sprachfeatures sich am besten für die Modellierung von Hochsprachen-Konstrukten (wie Objekten) anbieten.

LLVM im MiniJava-Compiler

Möglichkeiten zur Erzeugung von LLVM-Code im Minijava-Compiler:

1. Implementierung des Verfahrens zur Erzeugung von SSA-Code:
   - Generierung von LLVM-Code ohne ϕ-Knoten, d.h. noch nicht in SSA-Form.
   - Berechnung des Kontrollflussgraphen und des Dominanzbaums.
   - Einfügen der ϕ-Knoten an der Dominanzgrenze von Definitionen.
   - Umbenennung der Variablen, so dass SSA-Code entsteht.
2. Simple SSA-Form durch stack-allokierte Variablen
   (LLVM hat einen Optimierungspass für solchen Code:
   `mem2reg`;
   in unseren Experimenten war das Ergebnis jedoch langsamer
   als der direkt erzeugte SSA-Code.)

Simple SSA-Form für LLVM

Lege für jedes Temporary eine Speicherstelle an:

```
%__t = alloca i32
store i32 0, i32* %__t
%t = ptrtoint i32* %__t to i32
```

Bei jedem lesenden Zugriff auf `t` wird der Inhalt der Speicherstelle in ein frisches Temporary gelesen:

```
%t63 = inttoptr i32 %t to i32*
%t64 = load i32* %t63
```

Bei jedem schreibenden Zugriff auf `t` wird der Inhalt der Speicherstelle überschrieben:

```
%t65 = add i32 0, 0
%t66 = inttoptr i32 %t to i32*
store i32 %t65, i32* %t66
```

Man erhält LLVM-Code in SSA-Form, da jede Variable (inklusive `%t`) nur einmal definiert wird.

Beispiel

```
define i32 @LFac$ComputeFac(i32 %_t2, i32 %_t3) {
  // LABEL(L$$6)
  //   MOVE(TEMP(t6), CONST(0))
  //   CJUMP(<, TEMP(t5), CONST(1), L$$1, L$$2)
  //   ...

  %__t2 = alloca i32
  store i32 %_t2, i32* %__t2
  %t2 = ptrtoint i32* %__t2 to i32
  %__t3 = alloca i32
  store i32 %_t3, i32* %__t3
  %t3 = ptrtoint i32* %__t3 to i32
  %__t5 = alloca i32
  %t5 = ptrtoint i32* %__t5 to i32
  br label %L$$6
L$$6:
  %t9 = add i32 0, 0
  %t62 = inttoptr i32 %t5 to i32*
  store i32 %t9, i32* %t62
  %t63 = inttoptr i32 %t3 to i32*
  %t21 = load i32* %t63
  %t22 = add i32 1, 0
  %t23 = icmp slt i32 %t21, %t22
  br i1 %t23, label %L$$0, label %L$$1
  ...
```

Simple SSA-Form für LLVM

In einem MiniJava-Backend kann man mit folgenden Instruktionen auskommen:

- `alloca`, `load`, `store`, `inttoptr`, `ptrtoint`
- `icmp`, `br`
- `call`, `ret`
- `add`, `sub`, `mul`, `sdiv`, `shl`, `shr`, `ashr`, `and`, `or`, `xor`