Instruktionsauswahl

Instruktionsauswahl

Ziel: Übersetzung der IR-Bäume in Assemblerinstruktionen mit (beliebig vielen) abstrakten Registern.

- Die x86 (Pentium) Architektur
- Abstrakter Datentyp für Assemblerinstruktionen
- Instruktionsauswahl als Kachelung von Bäumen
- Greedy-Algorithmus (Maximal-Munch) und optimaler Algorithmus mit dynamischer Programmierung
- Implementierungsaufgabe

x86 Assembler: Register

- Der x86 (im 32bit Modus) hat 8 Register: `%eax`, `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`, `%esp`, `%ebp`
- `%esp` ist der *stack pointer*; nicht anderweitig verwendbar.
- `%ebp` kann als *frame pointer* benutzt werden. Wird kein frame pointer benötigt, so ist `%ebp` allgemein verfügbar.
- `%eax`, `%edx`, `%ecx` sind *caller-save*
- `%ebx`, `%esi`, `%edi`, `%ebp` sind *callee-save*

x86 Assembler: Operanden

Ein *Operand* ist entweder

- ein Register, oder
- eine Konstante, oder
- eine Speicherstelle (effective address).

Eine *Konstante* wird geschrieben als n wobei n eine Zahl ist. Der *Assembler* gestattet hier auch arithmetische Ausdrücke mit Konstanten und Abkürzungen.

Eine *Speicherstelle* wird geschrieben als $[reg_1 + reg_2 * s + n]$, wobei n Zahl oder arith. Ausdr. wie oben, $s \in \{1, 2, 4, 8\}$ und $reg_{1,2}$ Register sind ($reg_2 \neq$ `%esp`). Die Art des Inhalts kann mit dem Vorsatz `BYTE/WORD/DWORD PTR` disambiguiert werden.

x86 Assembler: Verschiebebefehle

Ein Verschiebebefehl hat die Form

```
mov     dst, src
```

Er verschiebt den Inhalt von `src` nach `dst`. Hier kann `src` ein beliebiger Operand sein; `dst` darf natürlich keine Konstante sein. Außerdem dürfen `src` und `dst` nicht beide zugleich Speicherstellen sein.

```
mov     [%esp+48-4], %ebx
mov     DWORD PTR [%eax], 33
mov     %esi, [%ecx]
```

x86 Assembler: Arithmetische Befehle

Plus, Minus, Mal haben das allgemeine Format

```
op      dst, src
```

wobei `op` eines aus `add`, `sub`, `imul` ist. Der Effekt ist

$$dst \leftarrow src\ op\ dst$$

Die Operanden sind denselben Einschränkungen wie bei den Verschiebebefehlen unterworfen. Bei `imul` muss `dst` ein Register sein.

x86 Assembler: Division

Das Format des Divisionsbefehls ist:

```
cdq
idiv    src
```

Dies platziert das Ergebnis der Division `%eax` durch `src` nach `%eax`. Außerdem wird das Register `%edx` überschrieben (mit dem Rest der Division) und `src` darf keine Konstante sein.

x86 Assembler: Sonderbefehle

Als Beispiel für einen der vielen Sonderbefehle geben wir hier den nützlichen Befehl *load effective address*:

```
    lea     dst, reg * s + n
```

(Hierbei ist n eine Konstante oder ein Register plus eine Konstante und $s \in \{1, 2, 4, 8\}$). Das Ergebnis von *load effective address* ist eine simple Zuweisung

$$\mathtt{dst} \leftarrow \mathtt{reg} \times s + n$$

Anstelle eine Speicherzelle zu laden, wird hier nur ihre Adresse berechnet.

x86 Assembler: Bitverschiebebefehle

Diese haben die Form

```
sop     dst, n
```

`sop` ist entweder eine Bitverschiebung auf vorzeichenfreien, `shl` (left shift) und `shr` (right shift), oder vorzeichenbehafteten Ganzzahloperanden, `sal` (arithmetic left shift, synomym zu `shl`) und `sar` (arithmetic right shift). Der Operand op wird um `n` Bits verschoben.

Beispiele:

- Multiplikation von `%eax` mit 4:

```
shl     %eax, 2
```

- Die Instruktion `cdq` kann implementiert werden durch:

```
mov     %edx, %eax
sar     %edx, 31
```

x86: Labels und unbedingte Sprünge

Jeder Befehl kann durch Voranstellen eines Bezeichners mit Doppelpunkt markiert (*labelled*) werden.

```
f:    sub    %esp, 24
L73:  mov    %eax, 35
```

Nach dem Doppelpunkt ist ein Zeilenumbruch erlaubt.

Ein unbedingter Sprungbefehl hat die Form

`jmp` *label*

x86 Assembler: Bedingte Sprünge

Ein bedingter Sprung hat die Form:

`cmp` op_1, op_2
jop label

Hier ist *jop* eines von `je` (equal), `jne` (not equal), `jl` (less than), `jg` (greater than), `jle` (less or equal), `jge` (greater or equal), `jb` (unsigned less), `jbe` (unsigned less or equal), `ja` (unsigned greater), `jae` (unsigned greater or equal).

Es wird gesprungen, falls die entsprechende Bedingung zutrifft, ansonsten an der nächsten Instruktion weitergearbeitet.

x86 Assembler: Funktionsaufruf

Die Instruktion

`call` *label*

springt nach *label* und legt die Rücksprungadresse auf den Keller, subtrahiert also 4 vom Stack Pointer.
Die Instruktion

`ret`

springt zur Adresse ganz oben auf dem Keller und pop-t diese.
Rückgabewerte sollten in `%eax` übergeben werden.
Es gibt auch Instruktionen `enter` und `leave` benutzen, die den Stackframe verwalten. Diese sind jedoch weniger effizient als die direkten Befehle.

Ausführung von Assemblerprogrammen

Ein ausführbares Assemblerprogramm braucht ein Label `Lmain` und die Direktive

```
.intel_syntax
.globl Lmain
```

Bei `Lmain` wird dann die Ausführung begonnen.

Mit `call` können andere Funktionen aufgerufen werden, z.B. in C geschriebene. Argumente für diese werden in `[%esp]`, `[%esp+4]`, `[%esp+8]` übergeben.

Ein Assemblerprogramm `prog.s` kann man folgendermaßen mit einem C Programm, z.B. `runtime.c`, verbinden:

```
cc -c runtime.c
cc -o runme prog.s runtime.o
```

Dann ist `runme` eine ausführbare Datei.

Abstrakte Assemblerinstruktionen

Wir verwenden eine abstrakte Klasse `Assem` für Assemblerbefehle über Temporaries. Je nach Zielarchitektur werden davon Klassen für die konkreten Instruktionen abgeleitet.

```
public interface Assem {
  //Temporaries, welche durch die Instruktion gelesen werden
  public List<Temp> use();
  //Temporaries, welche durch die Instruktion verändert werden
  public List<Temp> def();
  //mögliche Sprungziele der Instruktion
  public List<Label> jumps();
  public boolean isFallThrough();
  public Pair<Temp, Temp> isMoveBetweenTemps();
  public Label isLabel();
  public Assem rename(Func<Temp, Temp> sigma);
}
```

Der Registerallokator greift nur über dieses Interface auf Instruktionen zu und ist somit architekturunabhängig.

Im Buch werden konkrete Unterklassen mit den folgenden Konstruktoren vorgeschlagen:

```
public OPER(String a, List<Temp> d, List<Temp> s, List<Label> j)
public OPER(String a, List<Temp> d, List<Temp> s);
public MOVE(String assem, Temp dst, Temp src);
public LABEL(String assem, Temp.Label label)
```

Im `assem` String kann man mit
`‘s0,‘s1,...,‘d0,‘d1,...,‘j0,‘j1,...` auf die Komponenten von `dst, src, jumps` Bezug nehmen.

Einen Additionsbefehl würde man so repräsentieren:

```
List<Temp> l1 = Arrays.asList(new Temp[] {t1});
List<Temp> l2 = Arrays.asList(new Temp[] {t1, t2});
emit(new OPER("add ‘d0, ‘s0\n", l1, l2));
```

Für die spätere Registerzuweisung ist es wichtig, Quell- und Zielregister richtig anzugeben. Bei Architekturen mit komplizierten Instruktionen (wie x86) ist das fehleranfällig und führt leicht zu schwer lokalisierbaren Fehlern.

Konkrete Instruktionsklassen

Wir modellieren hier die einzelnen Assemblerinstruktionen durch Unterklassen von `Assem` und Enums. (Dieser Ansatz wird auch in anderen Projekten wie LLVM verfolgt.)

Somit werden die `use`- und `def`-Listen für jede Instruktion an einer einzigen Stelle berechnet.

Beispiel: Unterklasse `AssemBinOp` mit Konstruktor

```
public AssemBinOp(Kind kind, Operand dst, Operand src);
enum Kind {MOV, ADD, SUB, SHL, SHR, SAL, SAR, AND, OR, ...}
```

Dabei ist `Operand` eine abstrakte Klasse mit drei konkreten Unterklassen `Imm` (immediate), `Reg` (register) und `Mem` (memory), welche die verschiedenen möglichen Operanden repräsentieren.

Beispiel:

```
emit(new AssemBinaryOp(MOV, new Reg(eax), new Reg(l)));
emit(new AssemUnaryOp(IMUL, new Reg(r)));
emit(new AssemBinaryOp(MOV, new Reg(t), new Reg(eax)));
```

Instruktionsauswahl

Ziel: Auswahl einer Liste von Assemblerinstruktionen, die einen gegebenen `TreeExp`- oder `TreeStm`-Ausdruck implementiert.

Für jeden gegebenen Ausdruck gibt es viele Möglichkeiten, diesen durch Assemblerinstruktion zu implementieren. Wir suchen eine möglichst effiziente Implementierung.

Ansätze zur Erzeugung effizienten Maschinencodes:

1. Ordne den Assemblerinstruktionen Baummuster zu und zerteile den Baum in diese Muster.
2. Übersetze den Baum auf naive Weise in Assemblerinstruktionen und wende dann lokale Optimierungen auf dem Code an (peephole optimization).

Beide Ansätze sind in der Praxis erfolgreich. Wir betrachten hier den ersten.

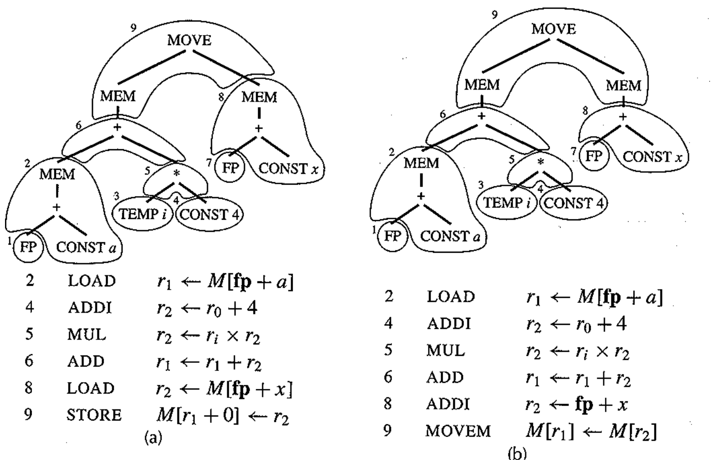
Instruktionsauswahl durch Baumkachelung

Ordne jeder Assemblerinstruktionen Baummustern zu:

Name	*Effect*	*Trees*
—	r_i	TEMP
ADD	$r_i \leftarrow r_j + r_k$	+ (two children)
MUL	$r_i \leftarrow r_j \times r_k$	* (two children)
SUB	$r_i \leftarrow r_j - r_k$	- (two children)
DIV	$r_i \leftarrow r_j / r_k$	/ (two children)
ADDI	$r_i \leftarrow r_j + c$	+(_, CONST); +(CONST, _); CONST
SUBI	$r_i \leftarrow r_j - c$	-(_, CONST)
LOAD	$r_i \leftarrow M[r_j + c]$	MEM(+(_, CONST)); MEM(+(CONST, _)); MEM(CONST); MEM(_)
STORE	$M[r_j + c] \leftarrow r_i$	MOVE(MEM(+(_, CONST)), _); MOVE(MEM(+(CONST, _)), _); MOVE(MEM(CONST), _); MOVE(MEM(_), _)
MOVEM	$M[r_j] \leftarrow M[r_i]$	MOVE(MEM(_), MEM(_))

Instruktionsauswahl durch Baumkachelung

Übersetze einen gegebenen Baum durch Kachelung mit den Baummustern. Schreibe die Ergebnisse jeweils in Temporaries.



(a) (b)

Instruktionsauswahl durch Baumkachelung

Eine sinnvolle Wahl der Baummuster sollte es ermöglichen, jeden Baum damit zu kacheln.

Bei Prozessoren mit eingeschränkten Instruktionen wie x86 kann es nötig sein, zusätzliche Kopierinstruktionen zu benutzen.

Instruktionen	Baummuster
`mov r, t1` `add r, t2`	$+(t1, t2)$

`mov %eax, t1`
`imul t2`
`mov r, %eax`
	$*(t1, t2)$

`mov r, t1` `add r, DWORD PTR [t2 + 4*t3 - 14]`	$+(t1, +(t2, -(*(4, t3), 14)))$

Algorithmen zur Baumkachelung

Ziel ist nun, eine möglichst gute Kachelung für einen gegebenen IR-Baum zu berechnen.

Der zu minimierende Preis einer Kachelung sei hier zunächst einfach die Anzahl der Kacheln darin.

Maximal Munch

- Greedy-Algorithmus
- Wähle die größte Kachel, die an der Wurzel des Baumes passt.
- Wiederhole diese Auswahl für Teilbäume an den Blättern der Kacheln.

Dieser Algorithmus findet eine Kachelung, die sich nicht durch Ersetzung zweier benachbarter Kacheln durch eine einzige verbessern lässt.

Die so gefundene Kachelung muss aber nicht optimal sein.

Algorithmen zur Baumkachelung

Optimale Kachelung durch dynamisches Programmieren

- Lege Tabelle an, in der für jeden Knoten des gegebenen IR-Baumes eine optimale Kachelung (mit minimalen Kosten) abgelegt werden soll.
- Fülle Tabelle bottom-up oder rekursiv: Blätter sind klar. Bei inneren Knoten werden alle passenden Kacheln probiert. Die optimalen Kachelungen der Unterbäume an den Blättern der Kacheln sind schon in der Tabelle eingetragen.
- Die Kosten können für jede Kachel einzeln festgelegt werden. Es wird eine Kachelung mit minimalen Kosten gefunden.
- Das Prinzip ist recht leicht; die Implementierung aber etwas aufwendig, weswegen es auch für diese Aufgabe codeerzeugende Werkzeuge gibt („code generator generator“). Diese lösen sogar eine etwas allgemeinere Aufgabe, die sich stellt, wenn Register nicht allgemein verwendbar sind.

Maximal Munch

Wir schreiben zwei wechselseitig rekursive Methoden:

```
Temp munchExp(TreeExp exp);
void munchStm(TreeStm stm);
```

Diese erzeugen als Seiteneffekt eine Liste von Assemblerinstruktionen (mit `void emit(Assem a);`).

Das von `munchExp` zurückgegebene Temporary enthält nach Ausführung der mit `emit` erzeugten Instruktionssequenz den Wert des Ausdrucks `exp`.

Die Funktionen `munchExp` und `munchStm` wählen jeweils diejenige Kachel aus, die von der Wurzel des übergebenen Baums das meiste abdeckt. Die beiden Funktionen rufen dann rekursiv die entsprechende Funktion für Teilbäume an den Blättern der Kachel auf. Damit werden Assemblerinstruktionen für die Teilbäume erzeugt. Danach werden die Assemblerinstruktionen der gewählten Kachel ausgegeben.

Instruktionsauswahl im MiniJava-Compiler

Erweitere die abstrakte Klasse der maschinenspezifischen Funktionen wie folgt:

```
public abstract class MachineSpecifics {
  ...

  public abstract
  Fragment<List<Assem>> codeGen(Fragment<List<TreeStm>> frag);

  public abstract
  String printAssembly(List<Fragment<List<Assem>>> frags);
}
```

Beachte: `codeGen` kann Prozeduren noch nicht mit dem vollständigen Entry/Exit-Code ausstatten, da die Größe des Frames vor der Registerverteilung noch nicht bekannt ist. Dieser Code wird erst in `printAssembly` hinzugefügt.

Spezielle Register

Für die Maschinenregister `%eax`, `%ebx`, `%ecx`,... sollten spezielle Temp-Objekte reserveriert werden.

Diese sollen zunächst nur benutzt werden, wenn dies absolut unvermeidbar ist, z.B. weil `idiv` sein Ergebnis immer in `%eax` zurückliefert.

Ansonsten verwenden wir immer frische Temporaries. Ergebnisse, die in speziellen Maschinenregistern zurückgegeben werden, werden sofort in Temporaries umkopiert.

Erst bei der Registerverteilung werden diese Temporaries dann auf die entsprechenden Machinenregister abgebildet. Unnötige Zuweisungen können dort entfernt werden.

Funktionsaufrufe

Funktionsaufrufe können als `call` Instruktion übersetzt werden.

Wir nehmen an, dass alle Temporaries, die keine Maschinenregister sind, bei einem Funktionsaufruf gespeichert werden.

Die Speicherung von Caller-Save-Registern überlassen wir dem Registerverteiler: Diese Register werden wir (später) in die `def`-Liste von `call` eintragen. Daran erkennt der Registerverteiler, dass er diese Register speichern muss, wenn sie nach dem Funktionsaufruf den gleichen Wert haben sollen wie vorher.

Die Callee-Save-Register sollten im Funktionsrumpf in Temporaries gespeichert und am Ende wiederhergestellt werden. Das ermöglicht es dem Registerverteiler, diese Register im Bedarfsfall auch anderweitig zu nutzen (es werden nur Temporaries auf Machinenregister verteilt, nicht aber Machinenregister umkopiert).

Programmieraufgabe

Implementierung des einer konkreten `MachineSpecifics`-Klasse für die x86-Architektur:

- Implementierung von `codeGen` mit Maximal Munch-Verfahren.
- Implementierung einer Frame-Klasse für x86.

Klassen zur Repräsentierung von x86-Assemblerinstruktionen finden Sie auf der Praktikumshomepage.

Sie sollten somit in der Lage sein, x86-Assembler mit unendlich vielen Registern zu erzeugen. Einen Simulator für solche Assemblerprogramme finden Sie ebenfalls auf der Homepage.