Übungsblatt 12

Aufgabe 12.1: Registervergabe mittels Graphfärben

Gegeben sei das folgende Programmfragment in der bekannten Zwischencodedarstellung:

```
a   = mov $49
b   = mov $3
%v0 = add a, b
%v1 = mul a, b
%v2 = add %v0, %v1
%v3 = div a, b
%v4 = add %v2, %v3
%v5 = div b, a
%v6 = add %v4, %v5
%v7 = mul a, b
%v8 = div $1, %v7
%v9 = add %v6, %v8
f   = mov %v9
ret f
```

Wenden Sie die aus der Vorlesung bekannte *Registervergabe mittels Graphfärben* an:

a) Ermitteln Sie zunächst die *Lebensspannen* der Variablen und virtuellen Register.

b) Erstellen Sie den zugehörigen *Interferenzgraphen* für eine Architektur mit vier Allzweckregistern `r[1-4]`.

c) Bilden Sie die Variablen und virtuellen Register mit Hilfe des *Graphfärbealgorithmus* auf die real zur Verfügung stehenden Register ab.

Aufgabe 12.2: Vergabe von Stack-Positionen durch `javac`

Übersetzen Sie die folgenden beiden Java-Methoden mittels `javac`:

```
public static final void foo(final int p) {
  if (p != 13) {
    int a = someMethod();
    System.out.println(a);
  } else {
    int b = someOtherMethod();
    System.out.println(b);
  }
}

public static final void bar(final int p) {
  int a = someMethod();
  if (p != 13) {
    System.out.println(a);
  } else {
    int b = someOtherMethod();
    System.out.println(b);
  }
}
```

Bestimmen Sie mit Hilfe des Disassemblers `javap` die Stack-Positionen, die der Java-Übersetzer für die jeweiligen lokalen Variablen vergeben hat. Nach welcher Regel scheint `javac` die Stack-Positionen zu vergeben?

Könnte der Speicherverbrauch für die lokalen Variablen in `bar()` reduziert werden? Wie?

Optionale Projektübung 8 (nach Bearbeitung bitte kurze Mail schreiben ☺)

Mit *LLVM* existiert ein sehr mächtiges Werkzeug, das u.a. als Backend für Übersetzer unterschiedlichster Sprachen verwendet werden kann. LLVM definiert dazu mit *LLVM IR* eine assemblerartige Zwischensprache, die als gemeinsame Schnittstelle dient. Statt Assemblercode für eine „echte“ Zielarchitektur zu erzeugen, gibt der Übersetzer Code in dieser Zwischensprache aus. Dieser wird anschließend von LLVM in die gewünschte Zielsprache übersetzt. Die wichtigsten Vorteile für uns Compiler-Entwickler sind dabei:

- Das Abstraktionsniveau von LLVM IR ist – zumindest im Vergleich zu Maschinencode – relativ hoch, was die Übersetzung in diese Sprache vereinfacht. Beispielsweise lassen sich die Instruktionen des `e2`-Zwischencodes im Wesentlichen 1-zu-1 auf die entsprechenden LLVM-Instruktionen abbilden.
- LLVM enthält (gute!) Code-Generatoren für unterschiedlichste Zielarchitekturen (x86, ARM, MIPS, PowerPC, ...). Haben wir einmal die Übersetzung nach LLVM IR implementiert, kann unser Übersetzer mit LLVMs Hilfe Code für alle diese Architekturen erzeugen.
- LLVM beinhaltet eine ganze Reihe von Optimierungsläufen, die den Code in LLVM IR so transformieren, dass die erwarteten Kosten (Laufzeit, Speicherverbrauch) des resultierenden Programms verringert werden. Die implementierten Algorithmen sind dabei so gut, dass sie auch naiv generierten Code aggressiv optimieren können – wir müssen uns also bei der Übersetzung nach LLVM IR nicht viel Mühe geben, um ein gutes Ergebnis zu erhalten... ☺

Ein *„Hallo, Welt“*-Programm in LLVM IR könnte beispielsweise wie folgt aussehen:

```
; Deklaration der Bibliotheksfunktion writeChar()
declare i64 @writeChar(i64)

; Definition der main()-Funktion
define i64 @main() {
  call i64 @writeChar(i64 72)  ; "H"
  call i64 @writeChar(i64 97)  ; "a"
  call i64 @writeChar(i64 108) ; "l"

  ; ...

  ret i64 0
}
```

Das Programm wird mit Hilfe des LLVM-Werkzeugs `llc`[1] in ein Objektmodul übersetzt, das anschließend wie bekannt mittels `ld` zusammen mit der `e2`-Bibliothek zu einem ausführbaren Programm gebunden wird:

```
$> llc hallo_welt.ll -o hallo_welt.o -filetype=obj
$> ld hallo_welt.o e2lib.a -o hallo_welt
$> ./hallo_welt
```

Im Rahmen dieser (optionalen!) Projektübung erweitern Sie Ihren `e2`-Übersetzer um ein LLVM-Backend, das den `e2`-Zwischencode in (menschenlesbaren) LLVM-Code übersetzt:

a) Schreiben Sie einen Zwischencodebesucher `e2c.backend.codegen.llvm.CodeGenerationLLVM`, der die eigentliche Übersetzung implementiert. Informationen zu LLVM IR finden Sie in der offiziellen Dokumentation[2] sowie der entsprechenden Vorlesung in *Ausgewählte Kapitel aus dem Übersetzerbau* (UE3)[3].

Hinweis: LLVM IR hat im Vergleich zu unserem `e2`-Zwischencode zwei wesentliche Besonderheiten:

[1] Je nach Distribution und LLVM-Version hat das Kommdozeilenwerkzeug u.U. ein Suffix mit der Versionsnummer, z.B. `llc-3.8`.
[2] https://llvm.org/docs/LangRef.html
[3] https://www2.cs.fau.de/teaching/WS2017/UE3/secure/ue3-llvm.pdf, Benutzername und Passwort wie in UE1

- Der LLVM-Code verwendet die sog. *SSA-Form* (*Static Single Assignment*, siehe auch *Optimierungen in Übersetzern* im Sommer). Dies bedeutet grob, dass Variablen und virtuellen Registern jeweils nur einmalig ein Wert zugewiesen werden darf. Recherchieren Sie, wie Sie mit Hilfe der `alloca`-Instruktion diese Beschränkung umgehen können (die UE3-Vorlesung entählt Informationen dazu).
- Der LLVM-Code verwendet ausschließlich *explizite Grundblockenden*:
  - Bedingte Sprünge haben *zwei* Sprungziele.
  - Vor jeder Sprungmarke muss eine Sprunginstruktion stehen.
  - Nach jeder Sprunginstruktion muss eine Sprungmarke stehen.

  Um die Übersetzung nach LLVM IR zu vereinfachen, sollten Sie zuvor den `e2`-Zwischencode entsprechend transformieren, also zusätzliche Sprunginstruktionen und -marken einfügen.

Tipp: Auf der Internetseite zur Übung finden Sie die von der „Musterlösung“ generierte Übersetzung des Testprogramms `examples/fib_mem.e2` nach LLVM IR, an der Sie sich ggf. orientieren können.

b) Analog zur voherigen Projektübung: Erstellen Sie eine neue Klasse `e2c.backend.codegen.llvm.BackendLLVM` als Unterklasse von `e2c.backend.Backend` und implementieren Sie die notwendigen Methoden. Verwenden Sie das Werkzeug `llc` (wie oben gezeigt) als Assemblierer, um den erzeugten LLVM-Code in ein Objektmodul zu übersetzen, sowie `ld` zum abschließenden Binden. **Wichtig**: Stellen Sie sicher, dass die von LLVM verwendete Aufrufkonvention zu der von Ihner Standardbibliothek verwendeten passt!

Tipp: Die Optimierungen in LLVM müssen bei der Übersetzung explizit angestoßen werden... ☺

„Registrieren“ Sie außerdem das neue Backend in `e2c.backend.BackendSelector`. Sorgen Sie bitte dafür, dass standardmäßig weiterhin Ihr natives Backend verwendet wird, das LLVM-Backend hingegen nur bei entsprechend gesetzter Kommandozeilenoption.

c) Testen Sie Ihre Implementierung. Sie können dazu die Testprogramme zu Meilenstein 5 verwenden, indem Sie *eine* der folgenden Optionen wählen:

- Sie sorgen *vorübergehend* dafür, dass standardmäßig das LLVM-Backend verwendet wird.
- Sie fügen *vorübergehend* die notwendigen Kommandozeilenoptionen zur Auswahl des LLVM-Backends beim Aufruf im Startskript `e2c` ein.

Dokumentieren Sie bitte in der Datei `README.md`, mit welcher Kommandozeilenoption das LLVM-Backend ausgewählt werden kann und welche LLVM-Version Sie verwendet haben. Schicken Sie uns nach Bearbeitung bitte außerdem eine kurze E-Mail, damit wir uns Ihre Implementierung kurz ansehen können ☺