KSP Aufgabe 8

Statten Sie Ihre VM mit einem kompaktierenden Garbage-Collector nach dem "Stop & Copy"-Verfahren aus!

Vorbereitungen

1. Legen Sie beim Programmstart den Stack dynamisch mit `malloc()` an. Sehen Sie eine Kommandozeilenoption `--stack n` vor, die einen Stack mit `n` KiB bereitstellt (**ACHTUNG**: `n * 1024` Bytes, NICHT Stack Slots, NICHT `n * 1000` Bytes). Der Defaultwert soll n = 64 sein, falls diese Kommandozeilenoption nicht angegeben wird. Hinweis: Sie werden möglicherweise Ihren Test auf Stackoverflow anpassen müssen!
2. Legen Sie beim Programmstart mit `malloc()` dynamisch einen eigenen Heap an, dessen Größe man durch die Kommandozeilenoption `--heap n` festlegen kann. Dabei soll n die Gesamtgröße des Heaps in KiB sein (ACHTUNG: `n * 1024` Bytes, NICHT `n * 1000` Bytes). Der Defaultwert soll `n` = `8192` sein, falls diese Kommandozeilenoption nicht angegeben wird.
3. Teilen Sie den Heap in zwei gleich große Hälften. Ändern Sie das Anlegen von Objekten so ab, dass Sie den benötigten Speicherplatz fortlaufend von einer Hälfte des Heaps nehmen. Brechen Sie Ihr Programm ab, wenn diese Hälfte erschöpft ist.

Garbage Collector

1. Programmieren Sie einen kompaktierenden Garbage-Collector nach dem Verfahren *Stop & Copy*, der anstelle des Programmabbruchs einen Sammeldurchlauf ausführt. Steht nach dem Durchlauf immer noch nicht genügend Platz für die verlangte Allokation zur Verfügung, sollte Ihre VM mit einer passenden Fehlermeldung terminieren.
   - Die Zeiger zu den "Root-Objekten" findet man an allen Stellen in unserer VM, die Objektreferenzen halten: der statische Datenbereich, das Return-Value-Register, diejenigen Stack-Slots, die auf Objekte verweisen, und - nicht vergessen! - die vier Komponenten des Big-Integer-Prozessors `bip`.

> **NOTE**
>
> Als "Broken Heart Flag" lässt sich sehr gut das zweithöchste Bit der `size`-Komponente benutzen. Der eigentliche "Forward Pointer" (= Byte-Offset in den Zielhalbspeicher, wo sich das kopierte Objekt befindet) belegt die restlichen 30 Bits. Das geht problemlos, da die `size`-Komponente bei bereits kopierten Objekten nicht mehr benötigt wird.
>
> Alternativen zur Speicherung in der `size`-Komponente — also die Verwendung von zusätzlichen Komponenten im struct `ObjRef` — wurden in der Veranstaltung besprochen.

2. Da man beim Integrieren eines Garbage-Collectors leicht einmal einen eigentlich zu verfolgenden Zeiger vergessen kann, stellen Sie eine Kommandozeilenoption `--gcpurge` zur Verfügung, die bewirkt, dass direkt nach einem Sammeldurchlauf der alte Halbspeicher mit Nullen überschrieben wird. So sollten sich Fehler der genannten Art schneller bemerkbar

machen.

3. Instrumentieren Sie Ihren Garbage-Collector so, dass man mit der Kommandozeilenoption `--gcstats` beim Sammeldurchlauf die folgenden Werte angezeigt bekommt:
   - Anzahl der seit dem letzten Durchlauf angelegten Objekte (sowie die davon belegte Zahl von Bytes),
   - Anzahl lebender Objekte (sowie die davon belegte Zahl von Bytes), freier Platz nach dem Sammeldurchlauf (Bytes im momentan benutzten Halbspeicher).
   - **Außerdem**: Sie sollten nach dem Ausführen der `halt`-Instruktion den Garbage-Collector einmal explizit aufrufen, damit man auch für kleine Programme mit wenig Speicherbedarf, die den GC sonst nicht triggern würden, diese statistischen Angaben erhält.
4. Der Instruktionssatz ist gegenüber Aufgabe 7 unverändert; auch der Compiler und der Assembler bleiben exakt gleich (bis auf die Versionsnummern).
5. Hier nun die endgültige Version der Referenzimplementierung: njvm

Testen der Implementierung

Bevor Sie den GC implementieren sollten sie sicherstellen, dass die u.a. Testprogramme (Heap Test, Tree und Factor) **ohne den GC** lauffähig sind. Erst dann macht es in einem letzten Schritt Sinn, den GC zu implementieren. Die Fehlersuche mit implementiertem GC gestaltet sich u.U. wesentlich schwerer also ohne. Daher macht es in jedem Fall Sinn, den GC nur mit einer gut getesteten und funktionierenden Implementierung anzufangen.

Testprogramme zusammgesetzte Objekte

Das Testprogramm Heap Test legt viele zusammengesetzte Objekte an. Hiermit wird u.a. die Korrektheit der Implementierung hinsichtlich Heap Speicher im zusammenspiel mit zusammengesetzen Objekten funktioniert.

Das Testprogramm Tree stellt einen arithmetischen Ausdruck dar, bzw. gibt ihn aus. Auch hierbei wird wieder überprüft, ob sich zusammengesetzte Objekte korrekt verhalten.

Komplexere Testprogramme

Die folgenden zwei etwas größeren Testprogramme muss Ihre VM (bei verschiedenen Heapgrößen) fehlerlos ausführen; dann wird sie mit guter Wahrscheinlichkeit auch als Hausübung akzeptiert.

Die Faktorisierung großer Zahlen

Das Testprogramm Factor hilft, die folgende Frage zu beantworten: Gegeben sind die Zahlen 10^{n+1} bei $n = 1, \ldots, 30$. Was sind die Primfaktoren jeder Zahl?

Bemerkung: Man muss hier zwischen Tests auf Zusammengesetztheit, Primalitätsbeweisen und Methoden zur Faktorisierung unterscheiden. Es gibt verschiedene Algorithmen in jedem der drei Gebiete. Ein gutes Buch, das auch diese Themen behandelt, ist [Henry Cohen: *A Course in Computational Algebraic Number Theory*, Springer 1993].

Ein kleines Computeralgebra-System

Computeralgebra-Systeme werden in ihrem Kern oft als Interpreter in einer LISP-ähnlichen Sprache implementiert. Deshalb wird hier njlisp zur Verfügung gestellt, eine Re-Implementierung des alten muLISP-80-Systems in Ninja. Der Makefile dient zum Zusammenbauen und Laufenlassen. Da das System auf mehreren Ebenen interpretiert abläuft, und diese Interpreter alle geladen werden müssen, folgt eine kurze Erklärung der Makefile-Targets, jeweils zusammen mit einem kleinen Beispiel:

1) `make` → erzeugt die Binärdatei `njlisp.bin`.

2) `make run` → lässt `njlisp` laufen. Probieren Sie als Eingaben:

a. `(PLUS 3 4)`

b. `(TIMES (PLUS 3 4) (DIFFERENCE 10 7))`

c. `(PUTD (QUOTE SQUARE) (QUOTE (LAMBDA (X) (TIMES X X))))`

d. `(SQUARE 12345)`

e. `(SQUARE (SQUARE (SQUARE 12345)))`

f. `(OBLIST)`

3) `make musimp` → lässt `njlisp` laufen und lädt das LISP-Programm `musimp.lsp`, das die mathematische Notation von Ausdrücken erlaubt und eine eigene Programmiersprache (mit gefälligerer Syntax als LISP) zur Verfügung stellt. Es werden dann interaktiv Eingaben erwartet, die berechnet und ausgegeben werden. Probieren Sie als Eingaben:

a. `123456789*987654321;`

b. `1-2*3+4*5;`

c. `FUNCTION F(N), WHEN N=0 OR N=1, N EXIT, F(N-1)+F(N-2) ENDFUN;`

d. `F(10);`

e. `FUNCTION G(N), A:0, B:1, LOOP WHEN N=0, A EXIT, H:A+B, A:B, B:H, N:N-1 ENDLOOP ENDFUN;`

f. `G(100);`

4) `make mumath`: lädt nach `musimp.lsp` auch noch die Mathematik-Module `arith.mus` (rationale Arithmetik) und `algebra.ari` (elementare Algebra). Probieren Sie als Eingaben:

a. `5/9 + 7/12;`

b. `((236 - 3*127) * -13) ^ 16;`

c. `GCD(861, 1827);`

d. `(-24) ^ (1/3);`

e. `(-4) ^ (1/2);`

f. `#E ^ (3 * #I * #PI / 2);`

g. `5*X^2/X - 3*X^1;`

h. `(5*X)^3 / X;`

i. `EXPD((3*Y^2 - 2*Y + 5)^3);`

j. `FCTR(6*X^2*Y - 4*X*Y^2/Z);`

NOTE

Das ist nur ein kleiner Ausschnitt aus dem Funktionsumfang des damaligen Systems, das auf Mikrocomputern mit maximal 64 KiB (!) Hauptspeicher lief. Es gab ca. 15 Pakete (wie die oben benutzten `arith.mus` und `algebra.ari`), die Aufgaben aus den Bereichen Matrizen, Gleichungen, Trigonometrie, Logarithmen, Differential- und Integralrechnung sowie Summen- und Grenzwertbildung lösen konnten.