Konzepte systemnaher Programmierung

Technische Hochschule Mittelhessen

Andre Rein

— Bigint Bibliothek —

Problemstellung

Aufgabe: Stelle $\sum_{i=1}^{100} \frac{1}{i}$ als exakten Bruch dar.

- $\frac{1}{a} + \frac{1}{b} = \frac{a+b}{a*b}$ ⇒ Im Nenner würde hier $100!$ stehen.
  - $100! \approx 10^{158}$ – also eine Zahl mit 158 Stellen
  - Dies können wir nicht in unseren 32 Bit darstellen
- **Ziel**: Rechnen mit beliebig großen Zahlen

Rechnen mit beliebig großen Zahlen

Nach Donald E. Knuth

Darstellung einer Zahl zur $Basis = 10$

Beispiel: $1024 \rightarrow 1 * 10^3 + 0 * 10^2 + 2 * 10^1 + 1 * 10^0$

Verallgemeinert: Zahl z zur Basis b

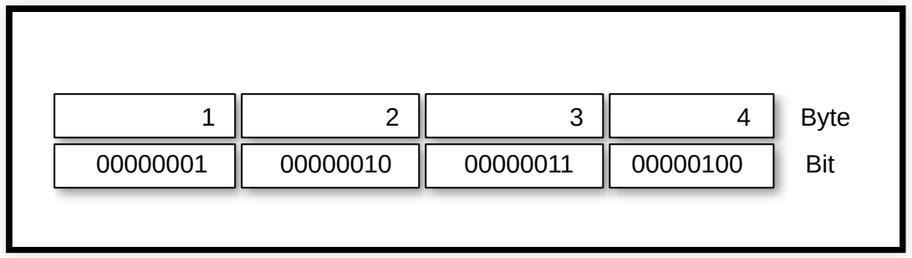
- $z = \sum_{i=0}^{n} d_i * b^i$
  - Nummerierung der Stellen: $d_n d_{n-1} d_{n-2} \dots d_0$
  - Weiterhin gilt: $0 \leq d_i < b, \forall i \in \{0, 1, \dots, n\}$
    - d_i ist also *kleiner* b und *größer gleich* 0

Die Zahlen selbst und alle Rechenoperationen auf dieser Zahlendarstellung müssen formalisiert und programmiert werden. Dies übernimmt die Bibliothek `bigint`, die zur Verfügung gestellt wird.

Zahlendarstellung der Bibliothek

Wir wählen die Basis $b = 256$, jede Ziffer belegt damit `1 Byte`.

Die Zahl: 1234_{256} (16909060_{10}) entspricht somit $1 * 256^3 + 2 * 256^2 + 3 * 256^1 + 4 * 256^0$




```c
typedef struct {
    int nd;                 // Number of digits
    unsigned char sign;     // Vorzeichen
    unsigned char digits[1]; // Size as needed
                            // mittels malloc!
} Big;
```

- Jedes Byte, d.h. `unsigned char digits[1]`, repräsentiert also eine Stelle unserer Zahl, zur Basis 256

> Der Typ `Big` wird von der Bibliothek bereitgestellt und muss nicht selbst implementiert werden. Details sind für uns nicht relevant. Für Interessierte: Entnehmen Sie die Details der Implementierung der `bigint`-Bibliothek.

Verwendung von Big in NJVM

Die bekannte Struktur, die für uns Objekte verwaltet ist `ObjRef`:

```
typedef struct {
    unsigned int size;     // # byte of payload
    unsigned char data[1]; // payload
} *ObjRef;
```

Nun werden aber nicht länger wie bisher Integer (`int`) im **Payload** (`unsigned char data[1]`) abgelegt, sondern Objekte vom Typ `Big`.

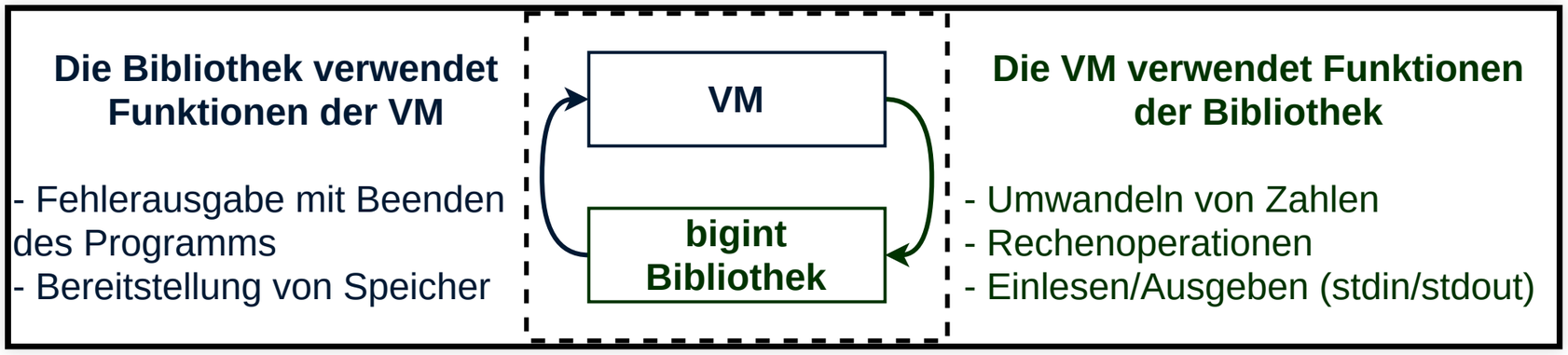
```
typedef struct {
    int nd;                  // Number of digits
    unsigned char sign;      // Vorzeichen
    unsigned char digits[1]; // Size as needed
                             // mittels malloc!
} Big;
```

D.h. in `data` ist ab sofort ein Objekt vom Typ `Big` enthalten.

Die `bigint` -Bibliothek sorgt dafür, dass die Daten vom Typ `Big` in unser `ObjRef` eingefügt werden. Unsere Aufgabe ist es **ausreichend Speicher** hierfür zur Verfügung zu stellen.

Bibliothekinterface

Die Bibliothek implementiert für uns diverse Funktionen, ist auf der anderen Seite aber abhängig von bestimmten Funktionen der `njvm`. Es besteht also eine wechselseitige Abhängigkeit, um die `bigint`-Bibliothek verwenden zu können.



Speicherbereitstellung für bigint

Die Bibliothek erwartet eine Funktion für die **Speicheranforderung**. Hierzu muss die entsprechende Funktion mit dem Prototyp `ObjRef newPrimObject(int dataSize)` implementiert werden. Der Parameter `dataSize` gibt an, wie viel Speicher reserviert werden muss.

- Die `bigint` -Bibliothek implementiert selbst keine Speicheranforderung, diese muss von der VM bereitgestellt werden.
- Aktuell verwenden wir zur Speicheranforderung in der VM `malloc()`
  - **Anmerkung**: Sobald wir unseren Garbagecollector implementieren, wird die Speicherverwaltung durch eine eigene Lösung ersetzt. Nur durch eine aktive und eigene Speicherverwaltung ist es überhaupt möglich einen GC zu implementieren!
  - Damit dies später umgestellt werden kann, verwendet die `bigint` -Bibliothek selbst auch kein `malloc()` und überlässt die Speicherverwaltung der VM.

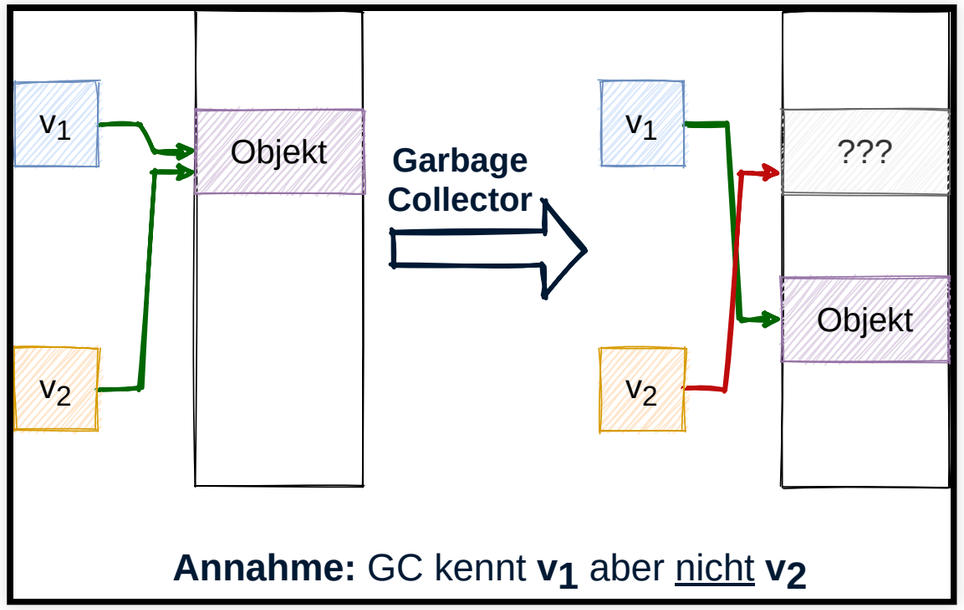
Rechenoperationen

Man könnte annehmen, dass die Rechenoperationen, die durch die `bigint`-Bibliothek bereitgestellt werden, folgendermaßen aussehen könnten:

- `ObjRef bigAdd(ObjRef op1, ObjRef op2);` — Beispiel für eine Addition, die als Parameter 2 Operanden erhält und als Ergebnis wieder ein `ObjRef` zurückgibt

Dies funktioniert unter Umständen aber nicht, da bei diesem Aufruf Speicher angefordert werden muss (für das **Ergebnis**). Wenn nicht mehr ausreichend Speicher vorhanden ist, wird die Garbagecollection ausgelöst — Dies funktioniert in diesem Fall aber nicht korrekt, da der GC die Parameter `op1` und `op2` **nicht kennt**, da es sich um Parameter einer aufgerufenen C-Funktion handelt. Davon weiß der GC jedoch nichts!

Rechenoperationen und Garbagecollection



- Der GC passt nur Zeiger für Variablen und Objekte an, die ihm bekannt sind (z.B. v_1).
- Ist eine Variable dem GC unbekannt (z.B. v_2), dann zeigt diese unbekannte Variable nach dem Durchlauf auf einen Speicherbereich, der nicht mehr gültig ist.
  - Das bedeutet, dass der Speicherinhalt, auf den diese Variable zeigt, möglicherweise nicht mehr den aktuellen Wert (der Variablen) repräsentiert oder gänzlich andere Daten beinhaltet.

Rechenoperationen

Die Konsequenz aus der Verwendung des GC ist, dass weitere **Register** eingeführt werden. Die `bigint` -Bibliothek stellt diese Register in Form von einer globalen Variable `bip` vom Typ `BIP` (*Big Integer Processor*) zur Verfügung.

```
typedef struct {
    ObjRef op1;   /* first (or single) operand */
    ObjRef op2;   /* second operand (if present) */
    ObjRef res;   /* result of operation */
    ObjRef rem;   /* remainder in case of division */
} BIP;
```

- Diese Register, also `bip.op1`, `bip.op2`, `bip.res` und `bip.rem` sind dem GC später bekannt und somit können alle Objektreferenzen korrekt aktualisiert werden, ohne das Informationen verloren gehen.

Der Funktionsprototyp der Rechenoperation Addition ändert sich demnach auf: `void bigAdd(void);` — weitere Funktionen sind in der Datei `bigint.h` aufgeführt.

Einbinden der Bibliothek

- Erzeugen der Bibliothek
  - `support.h` — was die Bibliothek fordert (z.B. `newPrimObject(int dataSize);`)
  - `bigint.h` — (was die Bibliothek bietet) und
  - `bigint.c` — (die Implementierung)

```
$ gcc -g -Wall -o bigint.o -c bigint.c
```


Mit `-c` wir nur die Objektdatei erzeugt.

Einbinden der Bibliothek

- Erzeugen eine *statischen* Bibliothek (static library)

```
$ ar -crs libbigint.a bigint.o
```


Die Konvention ist, dass jede *statische* Bibliothek mit den Zeichen `lib` beginnt und `.a` endet.

Benutzen der Bibliothek

- **1.** Header Dateien verwenden — **Annahme**: Header Dateien befinden sich im Pfad `./bigint/build/include`

```
$ gcc -I./bigint/build/include ...
```

- **2.** Bibliothek einbinden — **Annahme**: Die Bibliothek mit dem Namen `libbigint.a` wurde erzeugt und befindet sich, zusammen mit den Header Dateien, im Pfad `./bigint/build/lib`

```
$ gcc -L./bigint/build/lib ... -lbigint
```

- **3.** `njvm` erzeugen — **Annahme** `njvm.c` befindet sich im aktuellen Arbeitsverzeichnis

```
$ gcc -I./bigint/build/include -L./bigint/build/lib njvm.c -lbigint -o njvm
```

Beispiel: Einbinden und Nutzung

```
ar@lunar:[~/KSP_public_WS20_21/hausuebung]$ pwd
/home/ar/KSP_public_WS20_21/hausuebung
ar@lunar:[~/KSP_public_WS20_21/hausuebung]$ cd njvm/src/bigint/
ar@lunar:[~/KSP_public_WS20_21/hausuebung/njvm/src/bigint]$ make
ar@lunar:[~/KSP_public_WS20_21/hausuebung/njvm/src/bigint]$ cd ../
ar@lunar:[~/KSP_public_WS20_21/hausuebung/njvm/src]$ ls
bigint  helper.h  Makefile  njvm.c  operations.c  operations.h  stack.c  stack.h  vm.c  vm.h
ar@lunar:[~/KSP_public_WS20_21/hausuebung/njvm/src]$ gcc -g -Wall -std=c99 -pedantic \
-I./bigint/build/include -L./bigint/build/lib njvm.c operations.c stack.c vm.c -lbigint -o njvm
ar@lunar:[~/KSP_public_WS20_21/hausuebung/njvm/src]$ ./njvm --help
usage: ./njvm [option] [option] ...
  --help        show this help and exit
  --version     show version and exit
  --debug       start the ninja vm in debugger mode
```