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

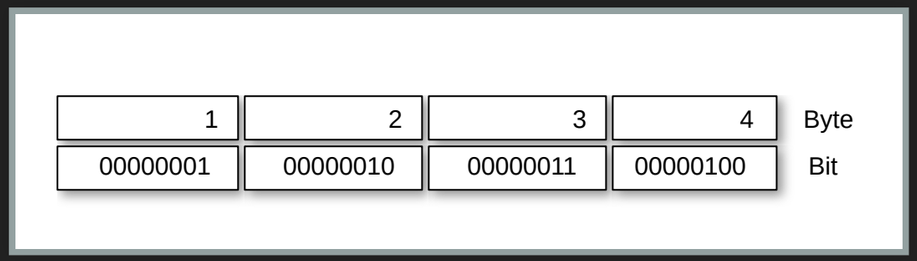
- $z = \sum_{i=0}^{n} d_i * b^i$
  - Nummerierung der Stellen: $d_n d_{n-1} d_{n-2} \dots d_0$
  - Weiterhin gilt: $0 \leq d_i < b, \forall i \in \{0, 1, \dots, n\}$
    - d_i ist also *kleiner* b und *größer gleich* 0

	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`.

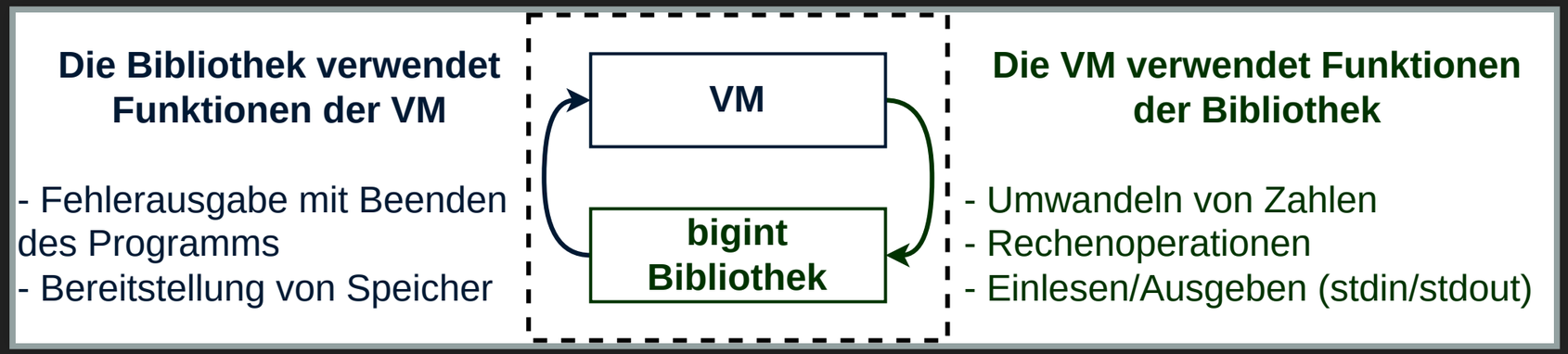
```
typedef struct {
    int nd;                   // Number of digits
    unsigned char sign;       // Vorzeichen
    unsigned char digits[1]; // Size as needed
                              // mittels malloc!
} Big;
```

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→data` 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 `void * 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.

SPEICHERADRESSE DES PAYLOADS IN OBJREF

Die Bibliothek ist so implementiert, dass sie zur Kompilierzeit noch nicht die Speicheradresse des **Payloads** (`unsigned char data[1]`) benötigt. Diese Entkopplung erlaubt es, dass Änderungen an der Struktur von `ObjRef` vorgenommen werden können (z.B. das Hinzufügen zusätzlicher Komponenten), ohne dass dies die Bibliothek beeinflusst.
Hierzu muss jedoch in der `njvm` eine zusätzliche Funktion implementiert werden, die die Speicheradresse des Payloads zur Laufzeit kennt und der Bibliothek bereitstellt. Der Prototyp der Funktion ist in der `support.h` angegeben:

```
void * getPrimObjectDataPointer(void * primObject);
```

Verwenden Sie in Ihrer `njvm` einfach die nachfolgende Implementierung der Funktion:

```
void * getPrimObjectDataPointer(void * obj){
  ObjRef oo = ((ObjRef) (obj));
  return oo->data;
}
```

BIBLIOTHEK-INTERNE VERARBEITUNG DER DATEN

```
void * getPrimObjectDataPointer(void * obj){
  ObjRef oo = ((ObjRef) (obj));
  return oo->data;
}
```

Die BigInt-Bibliothek verwendet **intern** ein Objekt mit dem Namen `BigObjRef`. Schaut man sich jedoch die Definition dieses Objektes an (`typedef void* BigObjRef;`), sieht man, dass es sich eigentlich nur um einen **alternativen Namen** für einen `void`-Zeiger, also einen Zeiger auf eine Speicheradresse (`void *`), handelt.

Konkret heißt dies, dass an jeder Stelle der Bibliothek, wo `BigObjRef` steht, genau so gut `void *` stehen könnte. Damit man aber das Objekt eindeutiger identifizieren kann, wurde der **Name** `BigObjRef` eingeführt und verwendet.

Achtung: Ein Objekt vom Typ `void *` kann in `C` nicht dereferenziert werden, weil es im engeren Sinne kein wirklicher Typ ist, mit dem `C` etwas konkretes assoziieren kann. Um also auf die Daten zuzugreifen, auf die der `void`-Zeiger zeigt, muss zuerst ein *cast* auf einen konkreten Datentyp erfolgen. Erst dann kann auf die Daten zugegriffen werden.

BIBLIOTHEK-INTERNE VERARBEITUNG DER DATEN

Die nachfolgenden 2 Folien beschreiben, wie die bigint-Bibliothek die beschriebenen Datenstrukturen **intern** verwendet. Dies soll Ihnen dabei helfen zu verstehen, warum dies so gemacht wird und wie diese Daten dort verwendet werden. Auf der Seite der `njvm` verwenden Sie ausschließlich den Datentyp `ObjRef` (den Sie bereits haben sollten). Generell können Sie sich merken:

- Die `njvm` verwendet ausschließlich `ObjRef` und kennt und arbeitet **nicht** mit dem Typ `BigObjRef`!
- Die BigInt-Bibliothek verwendet als Typ `BigObjRef` (was ein alternativer Name für `void *` ist) und kennt `ObjRef` nicht.

BIBLIOTHEK-INTERNE VERARBEITUNG DER DATEN

Der *cast* auf einen konkreten Datentyp erfolgt innerhalb der bigint Bibliothek im Makro:

```
#define BIG_PTR(bigObjRef) ((Big *) (getPrimObjectDataPointer(bigObjRef)))
```

Hierbei wird nun dem `void`-Zeiger der Typ "***Zeiger auf Big***" (`(Big *) void *` → `Big *`) zugeordnet, was letztendlich dem *cast* entspricht. Wird nun auf die Adresse zugegriffem, dann ist bekannt, dass auf Daten der als `Big`-definierten Struktur zugegriffen werden soll, die folgendermaßen definiert ist:

```
typedef struct {
  int nd;
  unsigned char sign;
  unsigned char digits[1];
} Big;
```

Der Zugriff auf konkrete Daten, z.B. `int nd`, erfolgt dann abermals über Makros:

```
#define GET_ND(bigObjRef) (BIG_PTR(bigObjRef)->nd)
// #define GET_ND(bigObjRef) ((Big *) (getPrimObjectDataPointer(bigObjRef))->nd) // Alternativ ausgeschrieben
```

Also: Umwandlung vom Typ `void *` zu `Big *`, Dereferenzierung der Adresse (`→`) und Zugriff auf den Wert `nd`, was einem Integer entspricht.

BIBLIOTHEK-INTERNE VERARBEITUNG DER DATEN

- **Die interne Verarbeitung der Daten in der Bibliothek selbst, sind aus Sicht der `njvm` nicht von Interesse**. Trotzdem ist es wichtig zu verstehen, wie die Daten zwischen `njvm` und BigInt-Bibliothek ausgetauscht werden und wie darauf zugegriffen wird.
- Bei C handelt es sich um eine **schwach typisierte Programmiersprache**, in der eine beliebige Umwandlung von Typen (*cast*) erlaubt ist. Man kann also jeden Datentyp in einen beliebigen Datentyp umwandeln, ohne das geprüft wird, ob diese Umwandlung auch sinnvoll ist oder nicht (z.B. in Java geht das so nicht).
- Ist eine Umwandlung nicht sinnvoll, dann ist der Zugriff auf die Daten u.U. fehlerhaft, da die Daten nicht korrekt interpretiert werden.

int_to_string.c

```c
#include <stdio.h>

void main(void) {
    int a = 0x67680061;
    int *a_ptr = &a;
    char * b = (char *) &a;
    printf("int          : [%x]\n", a);
    printf("int -> string: [%s]\n", b);
}
```

Ausgabe

```
$ gcc int_to_string_cast.c
$ ./a.out
int          : [67680061]
int -> string: [a]
```

Frage: Warum wird hier nur `[a]` ausgegeben?

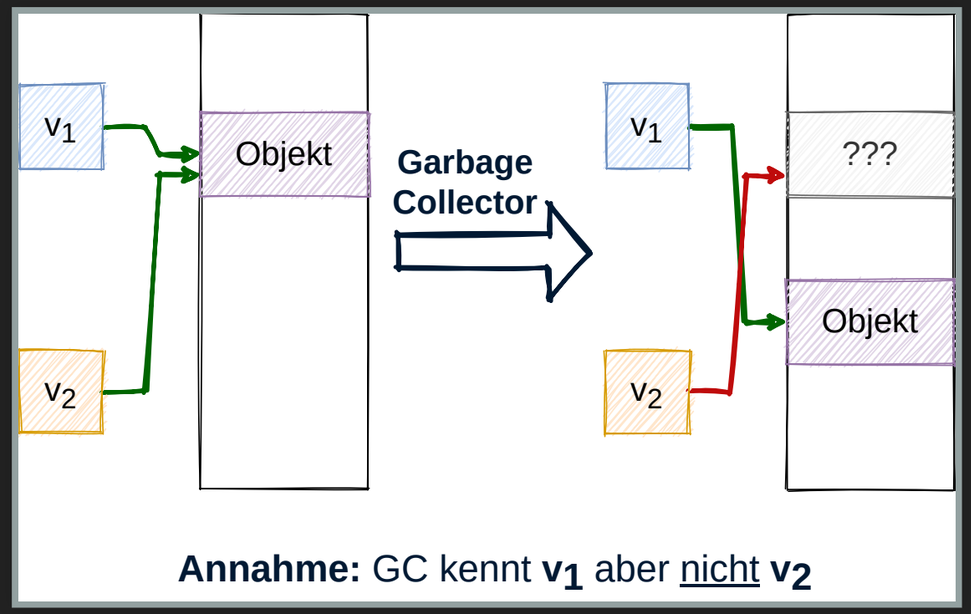
RECHENOPERATIONEN

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

- `BigObjRef bigAdd(BigObjRef op1, BigObjRef op2);` — Beispiel für eine Addition, die als Parameter 2 Operanden erhält und als Ergebnis wieder ein `BigObjRef` 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 eines GCs 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. (*Bedenken Sie, dass es sich hier wieder um eine Bibliotheks-interne Darstellung handelt. Deswegen wird hier `BigObjRef` verwendet.*)

```
typedef struct {
    BigObjRef op1;   /* first (or single) operand */
    BigObjRef op2;   /* second operand (if present) */
    BigObjRef res;   /* result of operation */
    BigObjRef 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 für die **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:
    - `void * newPrimObject(int dataSize);`
    - `void * getPrimObjectDataPointer(void * primObject);`
    - `void fatalError(char *msg);`
  - `bigint.h` — (was die Bibliothek bietet) und
  - `bigint.c` — (die Implementierung)

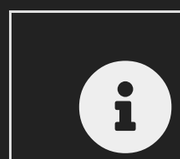
```
$ gcc -g -Wall -o bigint.o -c bigint.c
```

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/hausuebung]$ pwd
/home/ar/KSP_public/hausuebung
ar@lunar:[~/KSP_public/hausuebung]$ cd njvm/src/bigint/
ar@lunar:[~/KSP_public/hausuebung/njvm/src/bigint]$ make
ar@lunar:[~/KSP_public/hausuebung/njvm/src/bigint]$ cd ../
ar@lunar:[~/KSP_public/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/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/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
```