Numerix

Big integer library

Michel Quercia
`michel.quercia@prepas.org`

version 0.19, August 24, 2001

Contents

Chapter 1

Presentation

`Numerix` is a library implementing arbitrary long signed integers and the usual arithmetic operations between those numbers. Designed for a use with the Objective-Caml language, it is also available with reduced functionalities for the Camllight, C and Pascal languages on 32 or 64 bit Unix-type computers. It is shipped in three different versions:

`Clong`:

written in standard C. The basic object is a “digit”, the length of which is half of a machine word. So, elementary operations giving a two digit result are implemented with ordinary C operations on `long` variables, and this library should be portable to every computer with a binary arithmetic and words of even bit-length not fewer than 32 bits.

`Dlong`:

also written in C, but a digit is a whole machine word. Operations between digits are handled with the datatype `long long` (a `gcc` feature) which for now restricts this library to 32 bit computers (on the 64 bit computers at the Inria, a `long long` is of the same size as a `long` and there are no double-precision operations).

`Slong`:

written partly in assembly language for Intel x86 processors and partly in C, a digit is a 32 bit word.

Concerning the speed, `Numerix` compares well to the other multi-precision libraries commonly available, especially `Big_int` (the adaptation of `BigNum` for Camllight/Ocaml) and `GMP`. Below are the computing times for the n first decimal digits of π on a Linux PC with an Athlon-650Mhz processor and a 128Mb random access memory:

n	`Slong`	`Dlong`	`Clong`	`GMP-3`	`Big_int`
10^4	0.06s	0.24s	0.33s	0.08s	0.45s
10^5	1.49s	6.49s	9.05s	4.95s	57.22s
10^6	36.27s	148.71s	208.61s	505.58s	15867s

The same algorithm is used in the five cases, derived from a series expansion from Ramanujan, and only the big integer implementation differs. The libraries `Slong`, `Clong`, `Dlong` and `GMP` were used with a main program in C, whereas `Big_int` was with a main program in Ocaml. However, the influence of the main language on the running time is negligible for this kind of program for which the main part of the computing time is spent with the operations on several-million-bit-long numbers; the running times are similar when all the libraries are used with an Ocaml main program.

Below are the computing times on a Dec workstation with an alpha-500 processor and a 256Mb memory :

n	`Clong`	`Big_int`
10^4	0.15s	0.55s
10^5	4.68s	60s
10^6	112s	8310s

This high computing speed comes from the implementation of fast algorithms for all operations, namely multiplication (Karatsuba and Schönage (FFT) algorithms), division (Burnikel-Ziegler algorithm), square root (Zimmermann algorithm), gcd (recursive Lehmer algorithm) and binary to decimal conversion. `GMP-3.1` provides similar algorithms for multiplication and division achieving a greater speed than `Numerix` on those operations for operands below one million bits long, and a smaller speed for operands above one million bits. But the other operations are implemented by quadratic algorithms resulting in a greater computing time with `GMP` than with `Numerix` for an “average” program. Moreover, `GMP` and Camllight/Ocaml each have their own memory manager and this duality results in some circumstances in an important slow-down or even a memory exhaust causing the program to be aborted. Concerning `Big_int`, all operations starting from multiplication have a quadratic or higher complexity, therefore this library is not well suited for computation with huge numbers.

On the other hand, for computations involving only “small” numbers (say with with fewer than 300 bits), `Numerix` also performs nicely compared to `Big_int` because of a lower memory management overhead, and the handling at the C or assembly level of signs and integer lengths instead of handling them at the Caml/Ocaml level. `GMP`, though having fine-tuned algorithms for small size operands, is generally slower than `Numerix` when driven by a Caml/Ocaml main program because of the double memory management problem. The Hamming problem (to find the n^{th} natural integer having only 2 and 3 and 5 as prime divisors) involves roughly $n^{1/3}\ln(n)$ operations with small integers, from 100 to 300 bits. Below are the computing times for each of the libraries linked with an Ocaml main program (on a Linux PC Athlon-650, 128Mb RAM):

with Ocaml 2.04

n	`Slong`	`Dlong`	`Clong`	`GMP-3`	`Big_int`
10^{12}	0.17s	0.28s	0.26s	2.13s	0.69s
10^{15}	1.51s	2.52s	2.44s	20.23s	6.80s
10^{18}	17.06s	29.07s	28.99s	233.62s	79.59s
10^{21}	214.33s	355.56s	370.77s	2815.57s	891.14s

with Ocaml 3.00

n	`Slong`	`Dlong`	`Clong`	`GMP-3`	`Big_int`
10^{12}	0.17s	0.27s	0.26s	2.54s	0.70s
10^{15}	1.47s	2.61s	2.57s	–	6.58s
10^{18}	17.05s	30.10s	30.77s	–	75.52s
10^{21}	213.15s	368.21s	386.83s	–	933.48s

`GMP-3` does not pass this test with $n \geqslant 10^{15}$ and Ocaml-3.00 because of a memory overflow.

Though designed to be interfaced with the functional Camllight and Ocaml languages, `Numerix` allows "in-place operations", that is to say reusing the memory already allocated to some variable for storing the result of an operation instead of allocating a new memory block for this result. If the memory block allocated to a variable is too short for storing a result then a new large enough block is allocated to this variable and the former is given back to the Camllight/Ocaml memory manager for a potential recycling. The size increase follows a "double the need" policy (allocate a block twice as large as necessary) in order to avoid a ceaseless reallocation. In-place computation with this policy lowers to almost zero the memory management overhead and its influence on the total execution time, the cost being a less intuitive programming style and a greater memory use. Below are the running times for the Hamming problem with the in-place computation:

with Ocaml 2.04

n	`Slong`	`Dlong`	`Clong`	`GMP-3`	`Big_int`
10^{12}	0.15s	0.27s	0.26s	0.85s	0.69s
10^{15}	1.33s	2.39s	2.26s	3.24s	6.74s
10^{18}	15.20s	27.35s	26.31s	23.10s	78.77s
10^{21}	187.06s	334.55s	330.44s	285.82s	978.26s

with Ocaml 3.00

n	`Slong`	`Dlong`	`Clong`	`GMP-3`	`Big_int`
10^{12}	0.17s	0.28s	0.27s	0.94s	0.76s
10^{15}	1.38s	2.58s	2.36s	3.49s	7.51s
10^{18}	15.46s	29.56s	27.30s	22.96s	87.15s
10^{21}	189.25s	360.63s	343.15s	284.75s	1074.41s

In this test `Big_int` is at a disadvantage because in-place operations are not available and they have been simulated by using references at the cost of a greater running time compared to the functional version. On the other hand, `GMP` performs better with in-place computations as the two-memory-managers problem vanishes.

Chapter 2

Use with Ocaml

2.1 Interface

Numerix was developed and tested with Ocaml-2.02 and then Ocaml-3.00. A few tests with other versions of Ocaml have not shown any problem except for Ocaml-1.05 with which the automatic activation of the pretty-printers in the toplevel `ocamlnumx` could not be implemented (manual activation with `install_printer` works, though). Therefore, `Numerix` is likely to run with any version of Ocaml between 1.05 and 3.02 inclusive.

The five libraries `Clong`, `Dlong`, `Slong`, `GMP` and `Big_int` are declared as submodules of the module `Numerix` with the same signature of type `Int_type` in which the data types for big integers and mutable big integers (for in-place operations) are abstract. This allows the programmer to write a library-independent code and test his code with any of the big integer implementations and without the need to recompile the program, thanks to the module system of Ocaml and to the possibility of defining parametric modules (functors). You will find the public interface, part of the file `numerix.mli`, below:

```
module type Int_type = sig
type t     (* abstract integer         *)
type tref  (* abstract mutable integer *)

(* a,b,c,... denote the arguments of functions in this order *)
(* unary and binary operations *)
val add        :  t → t → t              (* a + b                  *)
val sub        :  t → t → t              (* a − b                  *)
val mul        :  t → t → t              (* a ∗ b                  *)
val quo        :  t → t → t              (* a div b                *)
val modulo     :  t → t → t              (* a mod b                *)
val quomod     :  t → t → t*t            (* (a div b, a mod b)     *)
val abs        :  t → t                  (* |a|                    *)
val neg        :  t → t                  (* −a                     *)
val sqr        :  t → t                  (* a^2                    *)
val sqrt       :  t → t                  (* ⌊√a⌋                   *)
val powmod     :  t → t → t → t          (* a^b mod c, b ⩾ 0       *)
```

```
val gcd         :  t → t → t                 (* gcd(a,b)                                *)
val gcd_ex      :  t → t → t*t*t             (* Bézout coefficients and gcd             *)
val cfrac       :  t → t → t*t*t*t*t         (* reduced integers, Bézout coef. and gcd *)
val eq          :  t → t → bool              (* a = b ?                                 *)
val neq         :  t → t → bool              (* a ≠ b ?                                 *)
val inf         :  t → t → bool              (* a < b ?                                 *)
val infeq       :  t → t → bool              (* a ⩽ b ?                                 *)
val sup         :  t → t → bool              (* a > b ?                                 *)
val supeq       :  t → t → bool              (* a ⩾ b ?                                 *)
val sgn         :  t → int                   (* sign(a)      (1,−1,0)                   *)
val cmp         :  t → t → int               (* sign(a − b) (1,−1,0)                    *)
val max         :  t → t → t                 (* max(a,b)                                *)
val min         :  t → t → t                 (* min(a,b)                                *)

(* mixed operations *)
val add_1       :  t → int → t               (* a + b                                   *)
val sub_1       :  t → int → t               (* a − b                                   *)
val mul_1       :  t → int → t               (* a ∗ b                                   *)
val quo_1       :  t → int → t               (* a div b                                 *)
val mod_1       :  t → int → int             (* a mod b                                 *)
val quomod_1    :  t → int → t*int           (* (a div b, a mod b)                      *)
val eq_1        :  t → int → bool            (* a = b ?                                 *)
val neq_1       :  t → int → bool            (* a ≠ b ?                                 *)
val inf_1       :  t → int → bool            (* a < b ?                                 *)
val infeq_1     :  t → int → bool            (* a ⩽ b ?                                 *)
val sup_1       :  t → int → bool            (* a > b ?                                 *)
val supeq_1     :  t → int → bool            (* a ⩾ b ?                                 *)
val cmp_1       :  t → int → int             (* sign(a − b) (1,−1,0)                    *)
val pow         :  t → int → t               (* a^b                                     *)
val root        :  t → int → t               (* ⌊ b√a ⌋                                 *)
val fact        :  int → t                   (* a!                                      *)
val random_bits :  int → t                   (* pseudo-random number                    *)

(* access to binary representation *)
val nbits       :  t → int                   (* bit size of |a|                         *)
val lowbits     :  t → int                   (* bits 0..30 of |a|                       *)
val highbits    :  t → int                   (* bits n − 31..n − 1, n =nbits(a)         *)
val nth_bit     :  t → int → bool            (* bit b of |a|                            *)
val nth_word    :  t → int → int             (* bits 16b..16b + 15 of |a|               *)
val shr         :  t → int → t               (* sgn(a) ∗ (|a| div 2^b)                  *)
val shl         :  t → int → t               (* a ∗ 2^b                                 *)
val split       :  t → int → t*t             (* sgn(a) ∗ (|a| div 2^b, |a| mod 2^b)     *)
val join        :  t → t → int → t           (* a + b ∗ 2^c                             *)

(* conversion functions *)
val of_int      :  int → t
val of_string   :  string → t
val int_of      :  t → int
val string_of   :  t → string
```

```
(* in-place operations *)
val add_in       :  tref → t → t → unit          (* a ← b + c                    *)
val sub_in       :  tref → t → t → unit          (* a ← b − c                    *)
val mul_in       :  tref → t → t → unit          (* a ← b ∗ c                    *)
val quo_in       :  tref → t → t → unit          (* a ← b div c                  *)
val quomod_in    :  tref → tref → t → t → unit   (* division and remainder       *)
val abs_in       :  tref → t → unit              (* a ← |b|                      *)
val neg_in       :  tref → t → unit              (* a ← −b                       *)
val sqr_in       :  tref → t → unit              (* a ← b^2                      *)
val sqrt_in      :  tref → t → unit              (* a ← ⌊√b⌋                     *)
val powmod_in    :  tref → t → t → t → unit      (* a ← b^c mod d, c ⩾ 0         *)
val add_1_in     :  tref → t → int → unit        (* a ← b + c                    *)
val sub_1_in     :  tref → t → int → unit        (* a ← b − c                    *)
val mul_1_in     :  tref → t → int → unit        (* a ← b ∗ c                    *)
val quomod_1_in  :  tref → t → int → int         (* a ← b div c, returns b mod c *)
val pow_in       :  tref → t → int → unit        (* a ← b^c                      *)
val root_in      :  tref → t → int → unit        (* a ← ⌊b^(1/c)⌋                *)
val fact_in      :  tref → int → unit            (* a ← b!                       *)
val shl_in       :  tref → t → int → unit        (* a ← b shl c                  *)
val shr_in       :  tref → t → int → unit        (* a ← b shr c                  *)
val split_in     :  tref→tref→t→int→unit         (* (a, b) ← c split d           *)
val join_in      :  tref → t → t → int → unit    (* a ← b + c ∗ 2^d              *)
val gcd_in       :  tref → t → t → unit          (* a ← pgcd(b, c)               *)
val gcd_ex_in    :  tref→tref→tref→t→t→unit      (* Bézout coef. and gcd         *)
val cfrac_in     :  tref → tref → tref → tref → tref → t → t → unit
                                        (* reduced integers, Bézout coef. and gcd *)

(* reference handling *)
val make_ref     :  t → tref                     (* ref(copy a)                  *)
val copy_in      :  tref → t → unit              (* a ← b                        *)
val copy_out     :  tref → t                     (* copy(!a)                     *)
val look         :  tref → t                     (* !a                           *)

(* miscellaneous *)
val name         :  unit -> string               (* name of module               *)
val toplevel_print :  t → unit                   (* printing with Format         *)

end (* module type Int_type *)

                        (* infix notation *)

module Infixes(E : Int_type) :  sig
open E

val ( + )        :  t → t → t                    (* add                          *)
val ( - )        :  t → t → t                    (* sub                          *)
val ( * )        :  t → t → t                    (* mul                          *)
val ( / )        :  t → t → t                    (* quo                          *)
val ( % )        :  t → t → t                    (* modulo                       *)
val ( /% )       :  t → t → t*t                  (* quomod                       *)
```

```
val ( << )     :  t → int → t                (* shl                          *)
val ( >> )     :  t → int → t                (* shr                          *)
val ( ** )     :  t → int → t                (* pow                          *)

val ( +.  )    :  t → int → t                (* add_1                        *)
val ( -.  )    :  t → int → t                (* sub_1                        *)
val ( *.  )    :  t → int → t                (* mul_1                        *)
val ( /.  )    :  t → int → t                (* quo_1                        *)
val ( %.)      :  t → int → int              (* mod_1                        *)
val ( /%.)     :  t → int → t*int            (* quomod_1                     *)

val ( +!  )    :  int → int → int            (* add                          *)
val ( -!  )    :  int → int → int            (* sub                          *)
val ( *!  )    :  int → int → int            (* mul                          *)
val ( /!  )    :  int → int → int            (* quo                          *)
val ( %!)      :  int → int → int            (* mod                          *)

val ( = )      :  t → t → bool               (* eq (Ocaml < 3.00)            *)
val ( <> )     :  t → t → bool               (* neq (Ocaml < 3.00)           *)
val ( < )      :  t → t → bool               (* inf (Ocaml < 3.00)           *)
val ( <= )     :  t → t → bool               (* infeq (Ocaml < 3.00)         *)
val ( > )      :  t → t → bool               (* sup (Ocaml < 3.00)           *)
val ( >= )     :  t → t → bool               (* supeq (Ocaml < 3.00)         *)

val ( =.  )    :  t → int → bool             (* eq_1                         *)
val ( <>.  )   :  t → int → bool             (* neq_1                        *)
val ( <.  )    :  t → int → bool             (* inf_1                        *)
val ( <=.  )   :  t → int → bool             (* infeq_1                      *)
val ( >.  )    :  t → int → bool             (* sup_1                        *)
val ( >=.  )   :  t → int → bool             (* supeq_1                      *)

val ( =!  )    :  'a → 'a → bool             (* eq                           *)
val ( <>!  )   :  'a → 'a → bool             (* neq                          *)
val ( <!  )    :  'a → 'a → bool             (* inf                          *)
val ( <=!  )   :  'a → 'a → bool             (* infeq                        *)
val ( >!  )    :  'a → 'a → bool             (* sup                          *)
val ( >=!  )   :  'a → 'a → bool             (* supeq                        *)

val ( ~ )      :  tref → t                   (* look (Ocaml < 3.00)          *)
val ( ~~ )     :  tref → t                   (* look                         *)

end (* Infixes functor *)
                          (* available modules *)

module Big     :  Int_type                   (* Ocaml bigints                *)
module Slong   :  Int_type                   (* assembly longints            *)
module Clong   :  Int_type                   (* C longints                   *)
module Dlong   :  Int_type                   (* C longints with double precision *)
module Gmp     :  Int_type                   (* GMP bigints                  *)
```

```
                                  (* timing and run-time selection of a module *)
external chrono : string → unit = "chrono"

module type Main_type = sig
  val main : string list → unit
end
module Start(Main : functor(E : Int_type) → Main_type) : sig
  val start : unit → unit
end
```

2.2 Details

Modules `Big`, `Slong`, `Clong`, `Dlong` or `Gmp` are available only on those computers for which the hardware allows compilation and also, concerning `Big` and `Gmp`, if the `BigNum` and `GMP` libraries are installed. As a rule, all modules of type `Int_type` are interchangeable, that is to say any given operation with valid arguments returns a result semantically independent of the module used and of the machine word size. The only exceptions are the `gcd_ex`, `gcd_ex_in`, `cfrac`, `cfrac_in`, `look` operations and the hash, serialize and deserialize functions (see below). On the contrary, the exceptions raised in case of invalid arguments may differ between modules.

Division

Division operations `quo`, `quo_1`, `modulo`, `mod_1`, `quomod` and `quomod_1` and their in-place counterparts follow the rule:

`quomod` a b and `quomod_1` a b return the pair (q, r) such that $a = bq + r$ with $0 \leqslant |r| < |b|$, r having same sign as b. b may not be zero.

`quo` a b and `quo_1` a b return only q.

`modulo` a b and `mod_1` a b return only r.

Gcd

The operations related to the greatest common divisor are `gcd`, `gcd_ex`, `cfrac` and their in-place counterparts.

`cfrac` a b returns a tuple (p, q, u, v, d) such that:

1. d is the positive gcd of a and b;
2. $ua - vb = d$;
3. $pu - qv = 1$;
4. $pa = qb$ is the lcm of a and b with same sign as ab.

The above conditions ensure uniqueness of p, q, d when a or b is non null, but the Bézout coefficients u and v are not unique and may differ between modules. Ensuring their uniqueness is possible as p and q are known, but it would have involved additional potentially time-expensive divisions.

gcd_ex a b returns the triple (u, v, d).

gcd a b returns d.

Comparisons

The sgn, cmp and cmp_1 functions always return one of the 0, 1 or −1 integers so it is possible to filter the result of any of those functions using the match construct.

The infix comparison symbols (<, <=, =, <>, >= and >) have a different status depending in which Ocaml version is used and on the opening or non opening of a module built with the Infixes functor.

- With Ocaml versions lower than 3.00, if no Infixes(E) is opened, the <,...,> symbols denote the usual Ocaml polymorphic comparison operators, of type 'a->'a->bool. These operators raise an

  ```
  Invalid_argument("equal: abstract value")
  ```

  exception when called with big integers.

- With Ocaml versions lower than 3.00, if an Infixes(E) module is opened, the <,...,> symbols denote the big integer comparison operators, of type E.t->E.t. The polymorphic comparison operators remain available with the <!,...,>! symbols.

- Starting with Ocaml 3.00, the <,...,> symbols denote the usual Ocaml polymorphic comparison operators, regardless of a possible opening of an Infixes(E) module, they are no more redefined by the Infixes functor. The Clong, Dlong, Slong and Gmp modules provide an interface with the generic comparison function of Ocaml, and the polymorphic comparison operators can be used with big integers from these modules without raising an exception. Moreover, big integers from these modules can be stored into ordered structures such as the sets from the Set module or the association tables from the Map module with the compare generic comparison function.

  On the contrary, the Bignum library does not provide an interface with the generic comparison function, therefore the polymorphic comparison functions of Ocaml cannot be used with data of type Big.t. If one wishes to compare such data with the <,...,> infix symbols, one needs to redefine these symbols on his own, or to modify the Bignum library in order to include an interface with the generic comparison function of Ocaml. Such a modification is described in the **Installation** chapter, section **Modifying the Bignum library**, page 42.

  Last, the <!,...,>! symbols remain binded to the polymorphic comparison operators by the Infixes functor, in order to preserve the compatibility with the previous Numerix versions.

Binary representation

Functions giving access to the binary representation of big integers follow the rules:

`nbits` a returns the number of significant bits of $|a|$; `nbits` $0 = 0$.

`lowbits` a and `highbits` a return the 31 lowest or highest bits of $|a|$ if a is non null and return 0 if $a = 0$. For `highbits`, when $0 < |a| < 2^{30}$, null bits are pushed at the right end in order to get a 31 bits word with highest bit set. Mathematically speaking, for $a \neq 0$ and n integer such that $2^{n-1} \leqslant |a| < 2^n$:

$$\texttt{highbits}(a) = (|a| * 2^{\max(0,31-n)}) \text{ div } 2^{\max(n-31,0)}.$$

`nth_bit` $a\ n$ and `nth_word` $a\ n$ return the n^{th} bit or the n^{th} 16 bits word of $|a|$, with least significant bit or word having index zero. When n is negative or greater than the length of $|a|$ those functions return 0.

`shl` $a\ n$ returns $a * 2^n$ if $n \geqslant 0$ and $\texttt{sgn}(a) * (|a| \text{ div } 2^{-n})$ if $n < 0$. `shr` $a\ n$ is equivalent to `shl` $a\ (-n)$.

`split` $a\ n$ splits $|a|$ into two parts and attaches the sign of a to each. n may not be negative.

`join` $a\ b\ n$ returns the number $a + b * 2^n$ even if a and b have different signs or overlap. n may not be negative.

Pseudo-random integers

`random_bits` n returns a pseudo-random integer between 0 and $2^n - 1$ inclusive, n may not be negative. The pseudo-random generator used is the Ocaml one, so the numbers returned by `random_bits` are identical for all runs of the same program not modifying the random generator seed, and are identical for all `Numerix` big integer modules available.

Mutable integers

Type `tref` values behave semantically as references to type `t` values, but their implementation may be more efficient and remains abstract.

op`_in` r *args* computes *op args* and stores the result in the memory referenced by r. If the result does not fit in the memory available, then a new block is allocated for storing the result and r is updated in order to reference this new block. For operations returning tuples: `quomod`, `gcd_ex`, `cfrac` and `split` the in-place counterpart stores the components of the tuple in the memory blocks referenced by the first arguments. Those references must be distinct.

`make_ref` v performs a physical copy of v and returns a reference to this copy.

`copy_in` $r\ v$ performs a physical copy of v in the memory block referenced by r.

`copy_out` r returns a physical copy of the memory referenced by r.

`look` r returns the value referenced by r. The result is volatile, that is to say it is guaranteed to be valid only until next in-place operation involving r. Read-modify-write operations, for instance *op*`_in` r (`look` r) x, are handled correctly. The `Infixes` functor defines two abbreviations for `look`: `~` valid for all version of Ocaml up to 2.99 and `~~` valid for all version of Ocaml up to 3.00 (`~` became a reserved word with Ocaml 3.00).

Timing

`chrono` *msg* prints on the standard error stream the CPU time in seconds since the beginning of the process, the difference with the previous time and the *msg* string. The inclusion of a few calls to `chrono` within a program informs the user of the approximate running times of the various phases in this program.

Run-time selection of a module

The `Start` functor allows the user to select a particular implementation of big integers at run-time and to execute the program with this implementation. The argument of `Start` is a `Main` functor receiving an implementation of the big integers as an argument and returning an implementation for the `main:string list` → `unit` function which constitutes the actual entry point of the program. The argument of `main` is the list of command-line parameters except those specifying which implementation of big integers to use. The choice of a particular implementation is done in the `start` function which parses the command-line, searching for a `-e` *module* option where *module* is the lower-case name of a big integer module available. When several `-e` *module* options are given on the command-line, the last one prevails; when none is, the first module available is selected as a default. When an `-e` option is not followed by a valid module name, an error message is printed showing the list of available modules and the program is aborted.

Hashing, serialization and deserialization

Starting with Ocaml 3.00, the `Clong`, `Dlong`, `Slong` and `Gmp` modules provide interfaces with the generic hashing function and with the serialization and deserialization functions of Ocaml, therefore the big integers from these modules can be stored into hash tables using the `Hashtbl.hash` function, and can be exported or imported through the `output_value` and `input_value` functions. The `Bignum` library provides interfaces with the serialization and deserialization functions of Ocaml, but no interface with the generic hash function, so data of type `Big.t` can be exported or imported with the `output_value` and `input_value` functions, but they cannot be stored into hash tables using the `Hashtbl.hash` generic hash function. It is possible to modify the `Bignum` library so as to include an interface with the generic hash function, please refer to chapter **Installation**, section **Modifying the Bignum library**, page 42 for such a modification.

Concerning the `Hashtbl.hash` generic hash function, the hash keys returned by this function can differ between modules for semantically equal big integers, due to different binary representations.

Concerning the export/import of big integers (functions `output_value` and `input_value`) and the encoding of big integers as sequences of bytes (functions from the `Marshal` module), typing must be preserved between export or encoding and import or decoding, that is to say it is not possible to turn a big integer from one module into a big integer from another module with these functions.

2.3 Additional modules

`Numerix` provides two modules allowing one to make new modules of type `Int_type` from existing modules: `Numcmp` and `Numcount`.

2.3.1 `Numcmp`: comparison between two modules

The `Numcmp` module contains two functors:

```
module M(A:Int_type)(B:Int_type) : Int_type

module Start(Main : functor(E:Int_type) -> Main_type) : sig
  val start : unit -> unit
end
```

`Numcmp.M` receives as arguments two modules A and B of type `Int_type` and returns a module C of the same type, in which each *op* operation is done with calls to $A.op$ and $B.op$ followed by semantic comparison between the results. When a comparison fails, that is to say when $A.op$ and $B.op$ return semantically different results with supposed semantically identical arguments, an exception is raised and the arguments and results in each module are printed on the standard error stream. This functor was used to debug the under-development modules by comparison with a reliable one, and except for this situation, its use is not recommended as duplication of calculations and comparison of results are time-consuming. Concerning operations `gcd_ex`, `gcd_ex_in`, `cfrac` and `cfrac_in`, Bézout coefficients are not compared, those from B are translated into values of type $A.$`t` in order to build results of type $C.$`t`.

`Numcmp.Start` is a functor similar to `Numerix.Start` allowing one to choose at run-time which big integer modules to compare through an `-e` *mod1_mod2* option. The module pairs available are:

`big_clong`, `big_dlong`, `big_slong`, `big_gmp`
`gmp_clong`, `gmp_dlong`, `gmp_slong`.

There is no default pair. When an incorrect `-e` option is given, the program is aborted and the list of available module pairs is printed on the standard error stream.

2.3.2 Numcount: counting operations

The `Numcount` module contains two functors :

```
module M(A:Int_type) : sig

  type statelt = {
    mutable n:float;  (* number of calls       *)
    mutable s:float;  (* sum of operand sizes  *)
    mutable m:int     (* maximal operand size  *)
  }

  val cadd  : statelt (* add      sub                                *)
  val cmul  : statelt (* mul      sqr                                *)
  val cquo  : statelt (* quo      modulo      quomod                 *)
  val cpow  : statelt (* pow      powmod      fact                   *)
  val croot : statelt (* sqrt     root                               *)
  val cgcd  : statelt (* gcd      gcd_ex      cfrac                  *)
  val cbin  : statelt (* shr      shl         split       join       *)
                      (* nbits    lowbits     highbits    nth_bit    *)
                      (* nth_word random_bits                        *)
  val cmisc : statelt (* abs      neg         make_ref    copy_in    *)
                      (* copy_out comparisons  conversions           *)

  val clear_stats : unit -> unit (* reset stats             *)
  val print_stats : unit -> unit (* display stats on stderr *)

  module Int : Int_type (* big integer module *)
end

module Start(Main : functor(E:Int_type) -> Main_type) : sig
  val start : unit -> unit
end
```

The `Numcount.M` functor receives as argument a module A of type `Int_type` and returns a module B`.int` of the same type, in which each *op* operation is done by calling $A.op$ and updating counter depending on *op*. The purpose of this functor is to provide statistics on the number of big integer operations performed by a program. The big integer operations are merged into eight categories each category being associated with a different counter:

- `nadd` counts additions and subtractions;
- `nmul` counts multiplications and squares;
- `nquo` counts divisions;
- `npow` counts exponentiations and factorials;
- `nroot` counts square roots and p-th roots;
- `ngcd` counts greatest common divisor and associated operations;
- `nbin` counts operations on the binary representations;
- `nmisc` counts all other operations.

Each counter B.cxxx has three fields:

B.cxxx.n : number of calls to one of the functions associated with cxxx ;

B.cxxx.s : sum of the bit sizes of the arguments for all calls ;

B.cxxx.m : maximum the bit sizes of the arguments for all calls.

For each call to a function in B.Int, field n of the associated counter is increased by one, field s is incremented by the average bit size of big integer operands and field m is updated so as to hold the maximum bit size of an operand for any function associated with this counter. Operands of type tref, int or string are not taken into account in size computations. Please note that the operations done with B.Int are slower than their counterpart in A because of the counter updates and the indirect calls overhead.

One can read the values of the counters and modify them at will, so as to determine how many operations of each category have been performed since the last reset. The function B.clear_stats resets all counters and the function B.print_stats displays on the standard error stream the statistics relative to each counter (number of calls, average operand bit size, maximum operand bit size).

Numcount.Start is a functor similar to Numerix.Start allowing one to choose at run-time which big integer module E to use through an -e *module* option with the same convention as for Numerix.Start. If the -count option is given on the command line then the function

Main(Numcount.M(E).Int).main

is called with the list of remaining command line arguments and the function Numcount.M(E).print_stats is called at program termination. If there is no -count option then the function Main(E).main is called as with Numerix.Start, and no statistics is printed. In the second case, there is no slow-down with respect to the use of Numerix.Start.

Below is an example of the use or not use of the -count option with the program computing digits of π described in section 6.6.4:

```
exemples/pi/ocaml/pi-opt -e slong -count 1000000 -d -skip
    0.01      0.01 Count(Slong), surplace
    0.36      0.35 puiss 5
    3.79      3.43 sqrt
   26.48     22.69 série lb=6875847 ls=6875855
   31.93      5.45 quotient
   39.11      7.18 conversion
   op      count   avg.size   max.size
  add     424091        240    6875855
  mul     636133        323    6875847
  quo          3    2845643   10197784
  pow          3         13         19
 root          1    6643863    6643863
```

```
  gcd          0          -          -
  bin          1    4643867    4643867
 misc    1767125        353    6875855
3.
14159 26535  89793 23846  26433 83279  50288 41971  69399 37510
... (19998 lignes omises)
56787 96130  33116 46283  99634 64604  22090 10610  57794 58151

exemples/pi/ocaml/pi-opt -e slong 1000000 -d -skip
    0.00     0.00 Slong, surplace
    0.36     0.36 puiss 5
    3.83     3.47 sqrt
   23.96    20.13 série lb=6875847 ls=6875855
   29.41     5.45 quotient
   36.60     7.19 conversion
3.
14159 26535  89793 23846  26433 83279  50288 41971  69399 37510
... (19998 lignes omises)
56787 96130  33116 46283  99634 64604  22090 10610  57794 58151
```

Computing the first million digits of π thus required 424091 additions/subtractions on operands having an average length of 240 bits, 636133 multiplications/squares on operands having an average length of 323 bits, 3 divisions between operands having an average length of 2845643, etc. The time overhead for computing those statistics is approximately 2.5 seconds.

2.4 Compilation

Ocaml programs using `Numerix` must be compiled with the following commands:

```
ocamlc -custom options nums.cma numerix.cma source files \
       -cclib -lmlnumx -cclib -lnums -cclib -lgmp

ocamlopt options nums.cmxa numerix.cmxa source files \
       -cclib -lmlnumx -cclib -lnums -cclib -lgmp
```

The `nums.cma`, `nums.cmxa`, `numerix.cma` and `numerix.cmxa` files contain the Ocaml part of the `BigNum` and `Numerix` llibraries in a compiled form (`nums.cma` and `nums.cmxa` are included in the Ocaml distribution). You may need to add a `-I` *path* option in order to inform the compilers about the location of the `numerix.cma` and `numerix.cmxa` files.

The `-cclib -lmlnumx`, `-cclib -lnums` and `-cclib -lgmp` options add the `libmlnumx.a`, `libnums.a` and `libgmp.a` files to the list of libraries in which the linker searches for external functions. `libnums.a` contains, in a compiled form, the C part of `BigNum`, `libmlnumx.a` the C part of `Numerix` and `libgmp.a` the C part of `GMP`. You may need to add `-ccopt -L`*path* options in order to inform the linker about the location of these `libxxx.a` files. If `GMP` is not installed or if its interface with Ocaml is not included in your Numerix version, then the option -cclib -lgmp must be omitted. If the BigNum library is not installed, then the parameters nums.cmxa and -cclib -lnums must be omitted.

Starting with Ocaml-3.00, the -custom, -cclib -lxxx and -ccopt xxx are recorded at build time into the numerix.cma and numerix.cmxa files and don't need to be given on compile command lines for programs using Numerix..

2.5 Example

```
                (* +----------------------------------------------+
                   |  simple.ml file: Use of Numerix with Ocaml  |
                   +----------------------------------------------+ *)

(* computing (sqrt(3) + sqrt(2))/(sqrt(3)-sqrt(2)) with n digits *)

open Numerix

module Main(E:Int_type) = struct
  module I = Infixes(E)
  open E
  open I

  (* functional version *)
  let calc_func n =
    let ten_n  = (of_int(5) ** n) << n in   (* 10^n  *)
    let ten_2n = sqr(ten_n)           in   (* 10^2n *)
    let sqrt_2 = sqrt(ten_2n *. 200)   in   (* sqrt(2) with n+1 digits *)
    let sqrt_3 = sqrt(ten_2n *. 300)   in   (* sqrt(3) with n+1 digits *)
    (ten_n * (sqrt_3 + sqrt_2)) / (sqrt_3 - sqrt_2)

  (* in-place version *)
  let calc_in n =
    let ten_n  = make_ref(of_int 0) in
    let ten_2n = make_ref(of_int 0) in
    let sqrt_2 = make_ref(of_int 0) in
    let sqrt_3 = make_ref(of_int 0) in
    let x      = make_ref(of_int 0) in
    let y      = make_ref(of_int 0) in

    pow_in   ten_n    (of_int 5)  n;        (* ten_n <- 5^n ...       *)
    shl_in   ten_n    ~~ten_n     n;        (*         ... * 2^n      *)
    sqr_in   ten_2n   ~~ten_n;              (* ten_2n <- 10^(2n)      *)
    mul_1_in sqrt_2   ~~ten_2n    200;      (* sqrt_2 <- ...          *)
    sqrt_in  sqrt_2   ~~sqrt_2;             (* .. sqrt(2*10^(2n+2))  *)
    mul_1_in sqrt_3   ~~ten_2n    300;      (* sqrt_3 <- ...          *)
    sqrt_in  sqrt_3   ~~sqrt_3;             (* .. sqrt(3*10^(2n+2))  *)
    add_in   x        ~~sqrt_3    ~~sqrt_2; (* x <- sqrt_2 + sqrt_2  *)
    sub_in   y        ~~sqrt_3    ~~sqrt_2; (* y <- sqrt_3 - sqrt_2  *)
```

```
    mul_in   x        ~~ten_n     ~~x;       (* x <- trunc(10^n*x ... *)
    quo_in   x        ~~x         ~~y;       (*               ... /y) *)
    copy_out x                               (* result               *)

  (* main function *)
  let main arglist =

    let rec parse (n,f) = function
      | []          -> (n,  f)
      | "-test"::s  -> parse (30, calc_func) s
      | "-func"::s  -> parse (n,  calc_func) s
      | "-in"  ::s  -> parse (n,  calc_in)   s
      | "-n"::x::s  -> parse (int_of_string x, f) s
      | _ -> prerr_endline "syntax : simple [-e module] [-test] [-func|-in] [-n <n>]";
             exit(1)
    in
    let (n,f) = parse (30,calc_func) arglist in
    print_endline(string_of(f n))

end (* Main *)

module S = Numcount.Start(Main)
let _ = S.start()
```

Compilation and execution (with Ocaml 3.00):

```
> ocamlc -o simple-byte nums.cma numerix.cma simple.ml
> ./simple-byte -e slong
9898979485566356196394568149410
>
> ocamlopt -o simple-opt nums.cmxa numerix.cmxa simple.ml
> ./simple-opt -e gmp -n 50 -count
   op      count   avg.size   max.size
  add          2        170        171
  mul          4        250        333
  quo          1        253        338
  pow          1          3          3
 root          2        340        341
  gcd          0          -          -
  bin          1        117        117
 misc          2         85        170
989897948556635619639456814941178278393189496131335
```

2.6 Customized toplevel

`ocamlnumx` is a customized toplevel derived from `ocaml` and linked against the `numerix.cma` object file and the `libnums.a`, `libmlnumx.a` and `libgmp.a` static libraries. It allows to use any `Numerix` module, the choice of a big integer implementation being done through an appropriate `open` directive.

```
> ocamlnumx
ocamlnumx : Objective Caml with big integer libraries
Numerix submodules : Big Clong Dlong Slong Gmp
Numerix version    : 0.19 (24 Aout 2001)

        Objective Caml version 3.02

# open Numerix;;
# open Big;;
# fact(30);;
- : Numerix.Big.t = 265252859812191058636308480000000
# module I = Infixes(Big) open I;;
module I :
  sig
    val ( + ) : Numerix.Big.t -> Numerix.Big.t -> Numerix.Big.t
    val ( - ) : Numerix.Big.t -> Numerix.Big.t -> Numerix.Big.t
    val ( * ) : Numerix.Big.t -> Numerix.Big.t -> Numerix.Big.t
    ...
    val ( ~~ ) : Numerix.Big.tref -> Numerix.Big.t
  end
# (of_int 1) << 100;;
- : Numerix.Big.t = 1267650600228229401496703205376
# #quit;;
```

Chapter 3

Use with Camllight

The Camllight interface for `Numerix` is derived from the Ocaml one by removing or adapting the functionalities specific to the Ocaml language. Please refer to the previous chapter to see the list of the available functions, only the differences with the Ocaml version will be explained here. This interface was successfully tested with Camllight versions 0.73, 0.74 and 0.75.

3.1 Modules

Camllight's module system is rather limited and provides neither sub-modules nor parametric modules. Nevertheless, it is possible to write library-independent code by using the short names of functions, the long names being inferred by the compiler with the help of `#open` directives included in the source file. Doing this way, just modifying those directives (possibly with a preprocessor) and recompiling the source file will change the big integer implementation used.

The available modules have the same names as in the Ocaml interface but without initial capital letters: `big`, `clong`, `dlong`, `slong` and `gmp`. There are no equivalents to the Ocaml modules built with the `Numcmp.M` and `Numcount.M` functors. Infix notations are available by opening the `infxxx` module where `xxx` is the name of the module implementing big integers.

3.2 Functions

All functions described in the Ocaml `Int_type` signature have been interfaced to Camllight with only two differences:

Division without remainder is named `quo` in Ocaml but `div` in Camllight. The reason for this difference is the infix status of the `quo` identifier in Camllight. The other names derived from `quo`: `quomod`, `quo_1`, `quomod_1`, `quo_in`, `quomod_in` and `quomod_1_in` are unchanged.

Accessing the value held by a reference is written `look` or `~` in Ocaml, but `look` or `?` in Camllight. There are two reasons for this difference: the identifier `?` is reserved in Ocaml and the identifier `~` has a prefix status in Ocaml, but an infix one in Camllight.

3.3 Compilation

Caml programs using `Numerix` must be compiled with the following command:

`camlc -custom` *options* `nums.zo numerix.zo` *source files* `-lclnumx -lnums -lgmp`

In their compiled form, the `nums.zo` and `numerix.zo` files contain the Caml part of the libraries `nums` and `Numerix` (`nums.zo` is included in the Camllight distribution). You may need to add a `-I` *path* option in order to inform the compiler about the location of the `numerix.zo` file.

The `-lclnumx`, `-lnums` and `-lgmp` options add the `libclnumx.a`, `libnums.a` and `libgmp.a` files to the list of libraries in which the linker searches for external functions. `libnums.a` contains the C part of `BigNum`, `libclnumx.a` the C part of `Numerix` and `libgmp.a` the C part of `GMP` in a compiled form. You may need to add `-ccopt -L`*path* options in order to inform the linker about the location of these files `libxxx.a`. If `GMP` is not installed or if its interface with Camllight is not included in your `Numerix` version, then the `-lgmp` option must be omitted. Accordingly, if `BigNum` is not installed, then the parameters `nums.zo` and `-cclib -lnums` must be omitted.

3.4 Example

```
              (* +------------------------------------------------+
                 |  simple.ml file: Use of Numerix with Camllight  |
                 +------------------------------------------------+ *)

(* computing (sqrt(3) + sqrt(2))/(sqrt(3)-sqrt(2)) with n digits *)

#open "slong";;
#open "infslong";;

  (* functional version *)
  let calc_func n =
    let ten_n  = (of_int(5) ** n) << n in   (* 10^n  *)
    let ten_2n = sqr(ten_n)            in   (* 10^2n *)
    let sqrt_2 = sqrt(ten_2n *. 200)   in   (* sqrt(2) with n+1 digits *)
    let sqrt_3 = sqrt(ten_2n *. 300)   in   (* sqrt(3) with n+1 digits *)
    (ten_n * (sqrt_3 + sqrt_2)) / (sqrt_3 - sqrt_2);;

  (* in-place version *)
  let calc_in n =
    let ten_n  = make_ref(of_int 0) in
    let ten_2n = make_ref(of_int 0) in
    let sqrt_2 = make_ref(of_int 0) in
    let sqrt_3 = make_ref(of_int 0) in
    let x      = make_ref(of_int 0) in
    let y      = make_ref(of_int 0) in
```

```
    pow_in   ten_n    (of_int 5)  n;        (* ten_n <- 5^n ...       *)
    shl_in   ten_n    ?ten_n      n;        (*         ... * 2^n      *)
    sqr_in   ten_2n   ?ten_n;               (* ten_2n <- 10^(2n)      *)
    mul_1_in sqrt_2   ?ten_2n     200;      (* sqrt_2 <- ...          *)
    sqrt_in  sqrt_2   ?sqrt_2;              (* .. sqrt(2*10^(2n+2))  *)
    mul_1_in sqrt_3   ?ten_2n     300;      (* sqrt_3 <- ...          *)
    sqrt_in  sqrt_3   ?sqrt_3;              (* .. sqrt(3*10^(2n+2))  *)
    add_in   x        ?sqrt_3     ?sqrt_2;  (* x <- sqrt_3 + sqrt_2  *)
    sub_in   y        ?sqrt_3     ?sqrt_2;  (* y <- sqrt_3 - sqrt_2  *)
    mul_in   x        ?ten_n      ?x;       (* x <- trunc(10^n*x ... *)
    quo_in   x        ?x          ?y;       (*               ... /y) *)
    copy_out x;;                            (* result                *)

  (* main function *)
  let main arglist =

    let rec parse (n,f) = function
      | []          -> (n,  f)
      | "-test"::s  -> parse (30, calc_func) s
      | "-func"::s  -> parse (n,  calc_func) s
      | "-in"  ::s  -> parse (n,  calc_in)   s
      | "-n"::x::s  -> parse (int_of_string x, f) s
      | _ -> prerr_endline "syntax : simple [-test] [-func|-in] [-n <n>]";
             exit(1)
    in
    let (n,f) = parse (30,calc_func) arglist in
    print_endline(string_of(f n));
    flush stdout;;

main (tl(list_of_vect sys__command_line));;
```

Compilation and execution:

```
> camlc -custom -o simple nums.zo numerix.zo simple.ml \
        -lclnumx -lnums -lgmp
> ./simple
9898979485566356196394568149410
```

Please note that all three `libclnumx.a`, `libnums.a` and `libgmp.a` libraries must be given to the linker though only the `slong` module is actually used, because the other modules are included in `numerix.zo` and refer to functions from those libraries.

3.5 Customized toplevel

A customized toplevel is available as it was in the Ocaml version:

```
> camllight ~/lib/camlnumx
```

```
>       Caml Light version 0.74

camlnumx : Caml toplevel with big integer libraries
Numerix submodules : big clong dlong slong gmp
Numerix version    : 0.19 (24 Aout 2001)

##open "big";;
##open "infbig";;
#fact(30);;
- : t = 265252859812191058636308480000000
#(of_int 1) << 100;;
- : t = 1267650600228229401496703205376
#quit();;
```

Chapter 4

Use with C

The C interface of Numerix was derived from the Ocaml one by adding a simple memory manager in order to cope with the lack of Ocaml GC, and by restricting the interface to in-place operations. The main purpose of this interface is to allow a fair comparison between Numerix and GMP (designed to be used with C), and to allow the compilation and the execution of test programs on computers where Ocaml is not installed.

4.1 Interface

The three `clong`, `dlong` and `slong` modules are available, as far as the C compiler and the computer hardware allow compilation. The following `#define` directives must be given to specify the size of a machine word and which module to use:

`#define bits_32` and `#define bits_64` specify the machine word size. One and only one of those directives must be specified.

`#define use_clong`, `#define use_dlong` and `#define use_slong` select the module implementing big integers. Not more than one of these directives may be specified, the default module selected when none is specified is `Clong`.

Those directives may be written at the beginning of each source file or transmitted to the preprocessor with `-Dxxx` options on the compile command-line.

The file `c-long_int.h` defines the datatype `xint` representing a big integer and contains the prototypes for all the public functions dealing with big integers. The function names are prefixed with a three character string identifying the `Clong`, `Dlong` or `Slong` module to which they belong and the abstraction level at which they operate:

`cn_nnn`: function `nnn` from `Clong` module operating on natural integers (digit arrays);

`cz_nnn`: function `nnn` from `Clong` module operating on signed integers (digit arrays with length and sign beside);

`cx_nnn`: function `nnn` from `Clong` module operating on big integers (signed integers extensible and relocatable).

`dn_nnn`, `dz_nnn`, `dx_nnn`, same for `Dlong` module.

`sn_nnn`, `sz_nnn`, `sx_nnn`, same for `Slong` module.

Normally, the programmer should only care about type `xint` big integers, and use only `cx_nnn` or `dx_nnn` or `sx_nnn` as functions operating on the big integers. The operations at `n` or `z` level are for internal use only and are not documented here. In order to allow writing module-independent code, the `c-long_int.h` file defines an `xx` macro which has `cx_` or `dx_` or `sx_` as expansion code depending on which symbol among `use_clong`, `use_dlong` and `use_slong` is defined. The recommended syntax to add `a` and `b` into `x` is:

```
xx(add)(&x,a,b);
```

this code being translated by the preprocessor into:

`cx_add(&x,a,b);` or `dx_add(&x,a,b);` or `sx_add(&x,a,b);`

The programmer is advised to systematically use the `xx` macro, and never to directly code the expanded identifiers. Then, just modifying the `#define use_xxx` directive will be the only change to do before a recompilation with another big integer implementation. Nevertheless, the functions of one module cannot operate on the data of another module, and there is no mechanism for distinguishing the `xint` datatype according to a specific module.

Below is a part of the file `c-long_int.h` showing the public functions prototypes:

```
/* public functions prefix  */
#if defined(use_slong)
#define xx(nom) sx_##nom
#elif defined(use_dlong)
#define xx(nom) dx_##nom
#else
#define xx(nom) cx_##nom
#endif

/* extensible integer */
typedef struct {...} * xint;

/* allocation/deallocation */
xint xx(new)(void);
void xx(free)(xint *x);

/* initialization from a long, a big integer or a string */
xint xx(of_int)     (long a);
xint xx(of_xint)    (xint a);
```

```
xint xx(of_string)  (char *s);

/* modification of an integer already initialized */
void xx(copy)       (xint *x, xint a);               /* x ← a          */
void xx(copy_int)   (xint *x, long a);               /* x ← a          */
void xx(copy_string)(xint *x, char *s);              /* x ← val(s)     */

/* usual operations */
void xx(add)   (xint *x, xint a, xint b);            /* x ← a+b        */
void xx(sub)   (xint *x, xint a, xint b);            /* x ← a-b        */
void xx(mul)   (xint *x, xint a, xint b);            /* x ← ab         */
void xx(quo)   (xint *x, xint a, xint b);            /* x ← a div b    */
void xx(quomod)(xint *x, xint *y,xint a, xint b);    /* x ← a div b, y ← a mod b */
void xx(gcd)   (xint *z, xint a, xint b);            /* x ← gcd(a,b)   */
void xx(powmod)(xint *x, xint a, xint b, xint c);    /* x ← a^b mod c */

/* extended gcd : z ← gcd(a,b), xa-by = z, xr-ys = 1 */
void xx(gcd_ex)(xint *x, xint *y, xint *z, xint a, xint b);
void xx(cfrac) (xint *r, xint *s, xint *x, xint *y, xint *z, xint a, xint b);

/* unary operations */
void xx(abs)   (xint *x, xint a);                    /* x ← |a|        */
void xx(neg)   (xint *x, xint a);                    /* x ← -a         */
void xx(sqr)   (xint *x, xint a);                    /* x ← a^2        */
void xx(sqrt)  (xint *x, xint a);                    /* x ← trunc(a^1/2) */

/* mixed operations */
void xx(add_1) (xint *x, xint a, long b);            /* x ← a+b        */
void xx(sub_1) (xint *x, xint a, long b);            /* x ← a-b        */
void xx(mul_1) (xint *x, xint a, long b);            /* x ← ab         */
long xx(quomod_1) (xint *x, xint a, long b);         /* x ← a div b, returns a mod b */
void xx(shl)   (xint *x, xint a, long b);            /* x ← a*2^b      */
void xx(shr)   (xint *x, xint a, long b);            /* x ← a div 2^b */
void xx(split) (xint *x, xint *y,xint a, long b);    /* x ← a mod 2^b, y ← a div 2^b */
void xx(join)  (xint *x, xint a, xint b, long c);    /* x ← a + b*2^c */
void xx(pow)   (xint *x, xint a, long b);            /* x ← a^b        */
void xx(root)  (xint *x, xint a, long b);            /* x ← trunc(a^1/b) */
void xx(fact)  (xint *x, long a);                    /* x ← a!         */

/* comparison */
int xx(eq)     (xint a, xint b);                     /* a = b ?        */
int xx(neq)    (xint a, xint b);                     /* a ≠ b ?        */
int xx(inf)    (xint a, xint b);                     /* a < b ?        */
int xx(infeq)  (xint a, xint b);                     /* a ⩽ b ?        */
int xx(sup)    (xint a, xint b);                     /* a > b ?        */
int xx(supeq)  (xint a, xint b);                     /* a ⩾ b ?        */
int xx(cmp)    (xint a, xint b);                     /* sign(a-b)      */
int xx(sgn)    (xint a);                             /* sign(a)        */

int xx(eq_1)   (xint a, long b);                     /* a = b ?        */
```

```
int xx(neq_1)  (xint a, long b);                        /* a ≠ b ?         */
int xx(inf_1)  (xint a, long b);                        /* a < b ?         */
int xx(infeq_1)(xint a, long b);                        /* a ⩽ b ?         */
int xx(sup_1)  (xint a, long b);                        /* a > b ?         */
int xx(supeq_1)(xint a, long b);                        /* a ⩾ b ?         */
int xx(cmp_1)(xint a, long b);                          /* sign(a-b)       */

/* access to the binary representation, lowbits, highbits and int_of return
  31 bits only for compatibility with Ocaml */
long xx(nbits)   (xint a);                              /* size of a       */
long xx(lowbits) (xint a);                              /* see Ocaml       */
long xx(highbits)(xint a);                              /* see Ocaml       */
long xx(nth_word)(xint a, long n);                      /* bits 16b..16b+15*/
long xx(int_of)  (xint a);                              /* bits 0..30      */

/* conversion integer -> string, release the string with free_string */
char *xx(string_of)(xint a);
void xx(free_string)(char *s);

/* timing */
void chrono(char *msg);
```

4.2 Details

A variable of `xint` type is a pointer to a data structure handled by the `Numerix` built-in memory manager. The initialization of an `x` variable of `xint` type is usually done in two steps:

1. initialization of the pointer `x` with a call to `xx(new)()`;
2. assignment of a value by giving the address of `x`, `&x`, as a result parameter for an operation, for instance `xx(add)(&x,a,b)`.

The `xx(of_int)`, `xx(of_xint)` and `xx(of_string)` functions do the initialization of the pointer and the assignment of a value from an ordinary integer, another big integer or a string in a single step. The instruction:

    xint x = xx(of_int)(i);

is formally equivalent to the two instructions:

    xint x = xx(new)(); xx(copy_int)(&x,i);

and similarly for `xx(of_xint)` and `xx(of_string)`. The execution of an operation with a call such as:

    xx(quo)(&x,a,b);

is valid if and only if the following requirements are fulfilled:

1. the `a` and `b` variables are initialized and point to valid values for that operation (`a` points to any value and `b` points to a non zero value);

2. the `x` variable contains an initialized pointer.

It is not necessary for `x` to point to some value before this operation. If it does, then the memory block storing this value will be overwritten with the result if the block is large enough, otherwise a new memory block will be allocated to store the result, the `x` pointer will be directed towards this new block and the former one will be deallocated.

Read-modify-write operations for which a same variable is given as operand and result are handled correctly. On the contrary, for operations returning several results (`quomod`, `gcd_ex`, `cfrac` and `split`) a single variable cannot be used for receiving more than one result. For instance the following instruction is *not* allowed:

```
xx(quomod)(&x,&x,a,b); /* illegal */
```

The `xx(free)` function releases the memory block allocated to a no more useful value. The instruction:

```
xx(free)(&x);
```

releases the value pointed by `x` if there is one and resets the `x` pointer. After this instruction, the `x` variable is still operational and may receive a new value.

Conversion of a big integer into a string is done with `xx(string_of)` which allocates a string of appropriate length and copies into it the decimal representation of its argument. This string must be released with a call to the `xx(free_string)` function.

Last, the `chrono` function prints the CPU time in seconds since the beginning of the process, the difference with the preceding time and the message given as an argument on the standard error stream.

4.3 Compilation

C programs using `Numerix` must be compiled with the following command:

`gcc` *options* `-Dbits_xx -Duse_yyy` *source files* `-lcnumx`

`-Dbits_xx` specifies the bit length of a machine word, `xx` = 32 or 64. `-Duse_yyy` specifies which module to use, `yyy` = `clong` or `dlong` or `slong`. `-lcnumx` adds the `libcnumx.a` file to the list of libraries in which the linker searches for external functions. You may need to add a `-L`*path* option to specify the location of this `libcnumx.a` file, and a `-I`*path* option to inform the preprocessor about the location of the `c-long_int.h` header file.

4.4 Example

```c
                /* +---------------------------------------+
                   |  File simple.c: Use of Numerix with C  |
                   +---------------------------------------+ */

/* computing (sqrt(3) + sqrt(2))/(sqrt(3)-sqrt(2)) with n digits */

#include <stdlib.h>
#include <stdio.h>
#include "c-long_int.h"

void compute(xint *res, long n) {
   xint ten_n  = xx(new)();
   xint ten_2n = xx(new)();
   xint sqrt_2 = xx(new)();
   xint sqrt_3 = xx(new)();
   xint x      = xx(new)();
   xint y      = xx(new)();

   xx(copy_int)(&ten_n,5);
   xx(pow)  (&ten_n,  ten_n,  n);            /* ten_n <- 5^n ...       */
   xx(shl)  (&ten_n,  ten_n,  n);            /*        ... * 2^n       */
   xx(sqr)  (&ten_2n, ten_n);                /* ten_2n <- 10^(2n)      */
   xx(mul_1)(&sqrt_2, ten_2n, 200);          /* sqrt_2 <- ...          */
   xx(sqrt) (&sqrt_2, sqrt_2);               /* .. sqrt(2*10^(2n+2))   */
   xx(mul_1)(&sqrt_3, ten_2n, 300);          /* sqrt_3 <- ...          */
   xx(sqrt) (&sqrt_3, sqrt_3);               /* .. sqrt(3*10^(2n+2))   */
   xx(add)  (&x,      sqrt_3, sqrt_2);       /* x <- sqrt_3 + sqrt_2   */
   xx(sub)  (&y,      sqrt_3, sqrt_2);       /* y <- sqrt_3 - sqrt_2   */
   xx(mul)  (&x,      ten_n,  x);            /* res <- trunc(10^n*x...*/
   xx(quo)  (res,     x,      y);            /*              ... /y) */

   xx(free)(&ten_n);
   xx(free)(&ten_2n);
   xx(free)(&sqrt_2);
   xx(free)(&sqrt_3);
   xx(free)(&x);
   xx(free)(&y);
}


int main(int argc, char **argv) {
  long n = 30;
  xint x = xx(new)();
  char *s;

  for(argc--, argv++; (argc); argc--, argv++) {
    if      (strcmp(*argv,"-test") == 0) {n = 30;}
    else if ((strcmp(*argv,"-n") == 0) && (argc > 0)) {
```

```
      argc--, argv++;
      n = atol(*argv);
    } else {
      fprintf(stderr,"syntax: simple [-test] [-n <n>]\n");
      fflush(stderr);
      exit(1);
    }
  }

  compute(&x,n);
  s = xx(string_of)(x);
  fprintf(stdout,"%s\n",s);
  fflush(stdout);

  xx(free)(&x);
  xx(free_string)(s);
  return(0);
}
```

Compilation and execution:

```
> gcc -O2 -Wall -Dbits_32 -Duse_slong -o simple simple.c -lcnumx
> ./simple
9898979485566356196394568149410
```

Chapter 5

Use with Pascal

The Pascal interface of Numerix is built on top of the C interface and provides the same functionalities. It was developed on a Linux PC (Athlon processor) with the Free Pascal version 1.0.2 compiler.

5.1 Interface

Three units are defined : `clong`, `dlong` and `slong`. Each unit contains the declaration of the corresponding big integer type and the declarations of the associated procedures and functions. The exported identifiers are the same in each unit, which permits to write user programs in a unit independent fashion, only the `uses` clause needs to be changed in order to change the big integer implementation.

```
unit clong; (* or dlong or slong *)
interface

(* extensible integer  *)
type xint = ...;

(* allocation/deallocation *)
function  xnew : xint;
procedure xfree(var x : xint);

(* initialization from a long, a big integer or a string *)
function  of_int     (a : longint) : xint;
function  of_xint    (a : xint   ) : xint;
function  of_string  (s : pchar  ) : xint;

(* modification of an integer already initialized *)
procedure copy       (var x : xint; a : xint);                (* x ← a        *)
procedure copy_int   (var x : xint; a : longint);             (* x ← a        *)
procedure copy_string(var x : xint; s : pchar);               (* x ← val(s)   *)

(* usual operations *)
```

```
procedure add     (var x   : xint; a,b   : xint);             (* x ← a+b       *)
procedure sub     (var x   : xint; a,b   : xint);             (* x ← a-b       *)
procedure mul     (var x   : xint; a,b   : xint);             (* x ← ab        *)
procedure quo     (var x   : xint; a,b   : xint);             (* x ← a div b   *)
procedure quomod  (var x,y : xint; a,b   : xint);             (* x ← a div b,  *)
                                                              (* y ← a mod b   *)
procedure gcd     (var z   : xint; a,b   : xint);             (* x ← gcd(a,b)  *)
procedure powmod  (var x   : xint; a,b,c : xint);             (* x ← a^b mod c *)

(* extended gcd : z ← pgcd(a,b), xa-by = z, xr-ys = 1 *)
procedure gcd_ex  (var x,y,z     : xint; a,b : xint);
procedure cfrac   (var r,s,x,y,z : xint; a,b : xint);

(* unary operations *)
procedure abs     (var x   : xint; a     : xint);             (* x ← |a|       *)
procedure neg     (var x   : xint; a     : xint);             (* x ← -a        *)
procedure sqr     (var x   : xint; a     : xint);             (* x ← a^2       *)
procedure sqrt    (var x   : xint; a     : xint);             (* x ← trunc(a^1/2) *)

(* mixed operations *)
procedure add_1   (var x   : xint; a     : xint; b : longint); (* x ← a+b       *)
procedure sub_1   (var x   : xint; a     : xint; b : longint); (* x ← a-b       *)
procedure mul_1   (var x   : xint; a     : xint; b : longint); (* x ← ab        *)
function  quomod_1(var x   : xint; a     : xint; b : longint) : longint;
                                                              (* x ← a div b,  *)
                                                              (* returns a mod b *)
procedure shiftl  (var x   : xint; a     : xint; b : longint); (* x ← a*2^b     *)
procedure shiftr  (var x   : xint; a     : xint; b : longint); (* x ← a div 2^b *)
procedure split   (var x,y : xint; a     : xint; b : longint); (* x ← a mod 2^b,*)
                                                              (* y ← a div 2^b *)
procedure join    (var x   : xint; a,b   : xint; c : longint); (* x ← a + b*2^c *)
procedure pow     (var x   : xint; a     : xint; b : longint); (* x ← a^b       *)
procedure root    (var x   : xint; a     : xint; b : longint); (* x ← trunc(a^1/b) *)
procedure fact    (var x   : xint; a     : xint);             (* x ← a!        *)

(* comparison *)
function eq   (a,b : xint) : boolean;                         (* a = b ?       *)
function neq  (a,b : xint) : boolean;                         (* a ≠ b ?       *)
function inf  (a,b : xint) : boolean;                         (* a < b ?       *)
function infeq(a,b : xint) : boolean;                         (* a ⩽ b ?       *)
function sup  (a,b : xint) : boolean;                         (* a > b ?       *)
function supeq(a,b : xint) : boolean;                         (* a ⩾ b ?       *)
function cmp  (a,b : xint) : longint;                         (* sign(a-b)     *)
function sgn  (a   : xint) : longint;                         (* sign(a)       *)

function eq_1    (a : xint; b : longint) : boolean;           (* a = b ?       *)
function neq_1   (a : xint; b : longint) : boolean;           (* a ≠ b ?       *)
function inf_1   (a : xint; b : longint) : boolean;           (* a < b ?       *)
function infeq_1(a : xint; b : longint) : boolean;            (* a ⩽ b ?       *)
function sup_1   (a : xint; b : longint) : boolean;           (* a > b ?       *)
```

```
function supeq_1(a : xint; b : longint) : boolean;             (* a ⩾ b ?        *)
function cmp_1  (a : xint; b : longint) : longint;             (* sign(a-b)       *)

(* access to the binary representation, lowbits, highbits and int_of return
  31 bits only for compatibility with Ocaml *)
function nbits   (a : xint)              : longint;             (* size of a       *)
function lowbits (a : xint)              : longint;             (* see Ocaml       *)
function highbits(a : xint)              : longint;             (* see Ocaml       *)
function nth_word(a : xint; n : longint) : longint;             (* bits 16b..16b+15*)
function int_of  (a : xint)              : longint;             (* bits 0..30      *)

(* conversion integer -> string *)
function string_of(x : xint) : ansistring;

(* timing *)
procedure chrono(msg : pchar);

(* errors *)
procedure failwith(msg : pchar);
```

5.2 Details

A variable of `xint` type is a pointer to a data structure handled by the `Numerix` built-in memory manager. The initialization of an `x` variable of `xint` type is usually done in two steps:

1. initialization of the pointer `x` with a call to `xnew`;
2. assignment of a value as the result of some computation, for instance `add(x,a,b)`.

The `of_int`, `of_xint` and `of_string` functions do the initialization of the pointer and the assignment of a value from an ordinary integer, another big integer or a string in a single step. The instruction:

`x := of_int(`i`);`

is formally equivalent to the two instructions:

`x := xnew; copy_int(x,`i`);`

and similarly for `of_xint` et `of_string`. The execution of an operation with a call such as:

```
quo(x,a,b);
```

is valid if and only if the following requirements are fulfilled:

1. the `a` and `b` variables are initialized and point to valid values for that operation (`a` points to any value and `b` points to a non zero value);
2. the `x` variable contains an initialized pointer.

It is not necessary for `x` to point to some value before this operation. If it does, then the memory block storing this value will be overwritten with the result if the block is large enough, otherwise a new memory block will be allocated to store the result, the `x` pointer will be directed towards this new block and the former one will be deallocated.

Read-modify-write operations for which a same variable is given as operand and result are handled correctly. On the contrary, for operations returning several results (`quomod`, `gcd_ex`, `cfrac` and `split`) a single variable cannot be used for receiving more than one result. For instance the following instruction is *not* allowed:

```
quomod(x,x,a,b); (* illegal *)
```

The `xfree` procedure releases the memory block allocated to a no more useful value. The instruction:

```
xfree(x);
```

releases the value pointed by `x` if there is one and resets the `x` pointer. After this instruction, the `x` variable is still operational and may receive a new value.

Conversion of a big integer into a string is done with `string_of` which allocates a string of appropriate length and copies into it the decimal representation of its argument. This string will be automatically released by the Pascal memory manager as soon as it will not be referenced anymore.

The `chrono` procedure prints the CPU time in seconds since the beginning of the process, the difference with the preceding time and the message given as an argument on the standard error stream.

Last, the `failwith` procedure prints its argument on the standard error stream and stops the program.

5.3 Compilation

Pascal programs using `Numerix` must be compiled with the following command:

`ppc386` *options* `-Fu`*ppu_path* `-Fl`*lib_path source files*

`-Fu`*ppu_path* specifies the directory containing the compiled `clong.ppu`, `clong.o`, `dlong.ppu`, `dlong.o`, `slong.ppu` and `slong.o` files. `-Fl`*lib_path* specifies the directory containing the `libcnumx.a` library. Those directives may be omitted if the files are stored in directories that are automatically searched by the Pascal compiler.

5.4 Example

```
program simple;
```

```
 (* +--------------------------------------------+
    |  File simple.p: Use of Numerix with Pascal  |
    +--------------------------------------------+ *)

(* computing (sqrt(3) + sqrt(2))/(sqrt(3)-sqrt(2)) with n digits *)

uses clong; (* or dlong or slong *)

procedure calcul(var res : xint; n : longint);
var ten_n, ten_2n, sqrt_2, sqrt_3, x, y : xint;
begin
   ten_n  := xnew;
   ten_2n := xnew;
   sqrt_2 := xnew;
   sqrt_3 := xnew;
   x      := xnew;
   y      := xnew;

   copy_int(ten_n,5);
   pow     (ten_n,  ten_n,  n);       (* ten_n <- 5^n ...       *)
   shiftl  (ten_n,  ten_n,  n);       (*        ... * 2^n       *)
   sqr     (ten_2n, ten_n);           (* ten_2n <- 10^(2n)      *)
   mul_1   (sqrt_2, ten_2n, 200);     (* sqrt_2 <- ...          *)
   sqrt    (sqrt_2, sqrt_2);          (* .. sqrt(2*10^(2n+2))  *)
   mul_1   (sqrt_3, ten_2n, 300);     (* sqrt_3 <- ...          *)
   sqrt    (sqrt_3, sqrt_3);          (* .. sqrt(3*10^(2n+2))  *)
   add     (x,      sqrt_3, sqrt_2);  (* x <- sqrt_3 + sqrt_2  *)
   sub     (y,      sqrt_3, sqrt_2);  (* y <- sqrt_3 - sqrt_2  *)
   mul     (x,      ten_n,  x);       (* res <- trunc(10^n*x...*)
   quo     (res,    x,      y);       (*               ... /y) *)

   xfree(ten_n);
   xfree(ten_2n);
   xfree(sqrt_2);
   xfree(sqrt_3);
   xfree(x);
   xfree(y);

end;


(* main program *)
var n,i : longint;
    c   : word;
    x   : xint;
begin

   n := 30;
   x := xnew;
```

```
   i := 1;
   while i <= paramcount do begin
      if  paramstr(i) = '-test' then n := 30
      else if (paramstr(i) = '-n') and (i < paramcount) then begin
         i := i+1;
         val(paramstr(i),n,c);
      end
      else failwith('syntax: simple [-test] [-n <n>]');
      i := i+1;
   end;

   calcul(x,n);
   writeln(string_of(x));

   xfree(x);

end.
```

Compilation and execution:

```
> ppc386 -Fu/home/quercia/lib -Fl/home/quercia/lib simple.p
Free Pascal Compiler version 1.0.2 [2001/03/12] for i386
Copyright (c) 1993-2000 by Florian Klaempfl
Target OS: Linux for i386
Compiling simple.p
Assembling simple
Linking simple
69 Lines compiled, 0.1 sec
> ./simple
98989794855663561963945681494l0
```

Chapter 6

Installation

6.1 Downloading

`Numerix` is available at the following URL:

```
http://pauillac.inria.fr/~quercia/cdrom/bibs/numerix.tar.gz
```

You will need the `gcc` C compiler to compile the C and assembly parts of the library, any recent version of `gcc` should fit for that. The library was successfully compiled on a Linux PC with `egcs-2.91.66` as well as on a Dec workstation with `gcc 2.7.2.3` for the `Clong` module only.

For Ocaml you need a not less than `1.05` version and for Camllight a not less than `0.73` version. Ocaml and Camllight are available at the URL:

```
http://pauillac.inria.fr/caml/index-eng.html
```

If you want to include the `Gmp` module in the interfaces for Ocaml and Camllight then you need `GMP` installed on your computer. `GMP` is available at the URL:

```
http://www.swox.com/gmp/
```

The Pascal interface can be compiled only on a Linux PC with the Free Pascal compiler which is available at the URL:

```
http://www.freepascal.org/
```

Please note that the Pascal interface requires the `libcnumx.a` library, therefore this interface can be compiled only if the C interface is also compiled.

6.2 Automatic configuration

Extract the `numerix.tar.gz` archive in a temporary directory and run the configuration script at the root.

On a Linux PC with the `bash` shell launch:

```
./configure   2>&1 | tee conflog
```

On a Dec workstation with the `csh` shell launch:

```
./configure       |& tee conflog
```

This script checks which parts of `Numerix` can be compiled on your computer and creates a `Makefile` file and a `numconf` file suited for your configuration. The `configure` script accepts the following options:

- -prefix=*dir*

  Set the common root for installation directories:

  `INSTALL_LIB     =` *dir*`/lib`,

  `INSTALL_BIN     =` *dir*`/bin`,

  `INSTALL_INCLUDE =` *dir*`/include`.

  The default prefix is `$HOME`.

`--libdir=`*dir,* `--bindir=`*dir,* `--includedir=`*dir*

Set one of `INSTALL_LIB`, `INSTALL_BIN` and `INSTALL_INCLUDE` directory regardless of the others.

`--enable-c`, `--disable-c`

Select or un-select the C interface. The default is to select it.

`--enable-caml`, `--disable-caml`

Select or un-select the Camllight interface. The default is to select it when Camllight is found on the computer.

`--enable-ocaml`, `--disable-ocaml`

Select or un-select the Ocaml interface. The default is to select it when Ocaml is found on the computer.

`--enable-pascal`, `--disable-pascal`

Select or un-select the Pascal interface. The default is to select it when `ppc386` is found on the computer.

`--enable-clong`, `--disable-clong`

Select or un-select the `Clong` module. The default is to select it.

`--enable-dlong`, `--disable-dlong`

Select or un-select the `Dlong` module. The default is to select it when the C compiler supports the double precision integer datatype (`long long`) and when this datatype length is twice the length of a `long`.

`--enable-slong`, `--disable-slong`

Select or un-select the `Slong` module. The default is to select it when the processor is an Intel x86 one and when the C compiler correctly supports the double precision integer datatype.

--enable-gmp, --disable-gmp

Select or un-select the Gmp module. The default is to select it when the GMP library is found on the computer.

--enable-caml_bignum, --disable-caml_bignum

--enable-ocaml_bignum, --disable-ocaml_bignum

Select or un-select the Big module independently for Camllight and for Ocaml. The default is to select it when the corresponding libnums library is found on the computer.

--disable-lang

Un-select all the languages not explicitely selected with a --enable-xxx option.

--disable-modules

Un-select all the modules not explicitely selected with a --enable-xxx option.

--disable-all

Un-select all the module/language combinations not explicitely selected with a --enable-xxx option.

6.3 Editing the numconf file

Normally the configure script should make a suitable numconf file. In case of trouble you can edit this file and change the values chosen by configure. After modifying the numconf file you need to recreate a Makefile file in order to take into account the changes and rewrite the build rules. To update your Makefile file just do:

```
make -f numconf makerules
```

In the numconf file, most of the parameters should receive a boolean value with the convention 0 = false and 1 = true. In case of trouble, have a look at the logs/pentium/numconf and logs/alpha/numconf files which contain the settings that were used for tests on Pentium and Digital alpha processors.

```
BITS = 32
```

Specify the size of a machine word, this should be 32 or 64.

```
USE_CLONG        = 1
USE_DLONG        = 1
USE_SLONG        = 1
USE_GMP          = 1
USE_CAML_BIGNUM  = 1
USE_OCAML_BIGNUM = 1
```

Specify which modules you want to compile, several modules may be selected. `Dlong` and `Slong` modules compile only on computers with an x86 Intel processor. Modules `Gmp` and `Big` compile only if you have `GMP` and `BigNum` installed.

```
MAKE_C_LIB      = 1
MAKE_OCAML_LIB  = 1
MAKE_CAML_LIB   = 1
MAKE_PASCAL_LIB = 1
```

Specify which interfaces you want to compile.

```
GCC = gcc -O2 -Wall
AR  = ar -rc
RANLIB = ranlib
```

Specify the commands to launch to call the C compiler and the librarian. You can add `-Ixxx` and `-Lxxx` directives if the compiler or the linker fail to find some header files or libraries.

```
OCAML_VERSION = 204
OCAML_LIBDIR  = /usr/local/lib/ocaml
CAML_VERSION  = 074
CAML_LIBDIR   = /usr/local/lib/caml-light
```

Specify without any period which versions of Ocaml and Camllight you have and in which directories they are installed. Those variables are taken into account only if interfaces for Ocaml and Camllight have to be compiled.

```
CAML       = camlc
CAMLLIBR   = camllibr
CAMLMKTOP  = camlmktop

OCAML      = ocamlc
OCAMLOPT   = ocamlopt
OCAMLMKTOP = ocamlmktop
```

Specify which commands `make` must launch to call the Camllight compiler, the Camllight librarian, the Camllight toplevel compiler, the Ocaml compiler, the Ocaml optimizing compiler, and the Ocaml toplevel compiler. The values above should be correct, except possibly for path prefixes.

```
PPC = ppc386
```

Specify which command `make` must launch to call the Pascal compiler. The value above should be correct, except possibly for path prefix and library paths.

```
INSTALL_LIB     = $(HOME)/lib
INSTALL_INCLUDE = $(HOME)/include
INSTALL_BIN     = $(HOME)/bin
```

Specify in which directories the compiled libraries, the header files (for C interface only) and the binaries should be installed. The value of `INSTALL_LIB` variable will be hard-coded into the customized toplevels `ocamlnumx` for Ocaml $\geqslant$ 1.07 and `camlnumx` for Camllight $\geqslant$ 0.74 so that those toplevels can find the `numerix.cmi` and `numerix.zi` compiled interfaces by themselves. Previous versions of Ocaml and Camllight do not provide such a possibility, and you will have to specify the path to those compiled interfaces each time you launch a toplevel with a `directory` directive. In the same way, the activation of pretty-printers is done automatically only with Ocaml-1.07 and Camllight-0.74 or greater; for previous versions you will have to activate them manually with the `install_printer` directive.

6.4 Modifying the Bignum library

Starting with Ocaml-3.00, data external to Ocaml can be compared, hashed, exported and imported through the generic functions of Ocaml, provided that the external library implements interfaces with these generic functions. The `Clong`, `Dlong`, `Slong` and `Gmp` modules implement such interfaces, but the `Big_int` module shipped with Ocaml implements only the interfaces with the export and import functions, it lacks the interfaces with the comparison and hashing functions.

The `lib/ocaml` directory of `Numerix` contains a patch file, `bignum.patch`, implementing these missing interfaces, it can be used to modify the `Bignum` library so as to make the generic comparison and hashing functions available within the `Big` module.

Note: the modifications described below are optional. If you do not intend to use the `Big` *module, or if you can use it without generic comparisons and generic hashing, then you are advised against making these modifications; you can skip to the next section,* **Compilation***, then.*

You need Ocaml sources. Unpack them in a temporary directory, move to the topmost directory of Ocaml distribution (directory `ocaml-`*x.yy*) and launch:

```
patch -b -p0 <.../numerix-0.19/lib/ocaml/bignum.patch
```

where `...` denote the directory containing `Numerix` sources. The `patch` program must report with a message such as:

On a Linux PC

```
patching file otherlibs/num/nat_stubs.c
```

On a Dec workstation

```
Hmm...  Looks like a new-style context diff to me...
The text leading up to this was:
--------------------------
|*** otherlibs/num/nat_stubs.c.orig     Fri Aug 24 15:08:51 2001
|--- otherlibs/num/nat_stubs.c  Fri Aug 24 15:46:58 2001
```

```
--------------------------
Patching file otherlibs/num/nat_stubs.c using Plan A...
Hunk #1 succeeded at 11.
Hunk #2 succeeded at 30.
Hunk #3 succeeded at 335.
done
```

If there are no errors, you can rebuild and install the modified Ocaml system.

6.5 Compilation

After editing and adapting the `numconf` file you can start the compilation. The targets are:

`lib`:
: compiles the libraries and the interface files;

`examples`:
: compiles the examples;

`test`:
: executes every example program with the `-test` option;

`install`:
: copies the libraries, the header files and the binaries in the directories specified by the `INSTALL_xxx` variables;

`clean`:
: removes all compiled files.

On a Linux PC with the `bash` shell successively launch:

```
make lib      2>&1 | tee liblog
make exemples 2>&1 | tee exlog
make test     2>&1 | tee testlog
```

On a Dec workstation with the `csh` shell successively launch:

```
make lib      |& tee liblog
make exemples |& tee exlog
make test     |& tee testlog
```

There should be neither compile error nor warning. If there are some and if you cannot solve the problem on your own, please send the `liblog`, `exlog` and `testlog` log files to `michel.quercia@prepas.org` for diagnosis. If you have faced some problems that you have been able to fix alone, please let me know so that I may modify the faulty files. The `logs/alpha/xxxlog` and logs/pentium/xxxlog files shipped with the distribution contain the compilation logs for a Linux PC and a Dec workstation, have a look at them in case of trouble.

If the compilation and the tests have been successful, you can install the Numerix library with one of the commands:

```
make install  2>&1 | tee inslog
make install  |& tee inslog
```

This will copy the files:

```
big.zi       dlong.o      infgmp.zi    numcount.cmi slong.ppu
camlnumx     dlong.zi     infslong.zi  numerix.a    slong.o
clong.ppu    gmp.zi       libclnumx.a  numerix.cma  slong.zi
clong.o      infbig.zi    libcnumx.a   numerix.cmi
clong.zi     infclong.zi  libmlnumx.a  numerix.cmxa
dlong.ppu    infdlong.zi  numcmp.cmi   numerix.zo
```

in the $(INSTALL_LIB) directory,

```
c-long_int.h  long_int.h
```

in the $(INSTALL_INCLUDE) directory and

```
ocamlnumx
```

in the $(INSTALL_BIN) directory or only some of them depending on which modules and interfaces you asked for.

Now the installation is finished and you can enjoy the multi-precision programming. The user guide you are presently reading is available in the doc subdirectory in Postscript (numerix-eng.ps) and LaTeX (numerix-eng.tex) formats.

6.6 Description of the examples

The subdirectory exemples (this is the French spelling for examples) contain various programs using Numerix with C, Camllight, Ocaml or Pascal. Each example is in a separate directory with c, caml, ocaml and pascal subdirectories containing the relevant source files.

There are two versions for the C-written programs: one using the Numerix library (file xxx-long.c), the other the GMP library (file xxx-gmp.c). Compiled binaries have a name like: xxx-c, xxx-d, xxx-s and xxx-g, the last letter showing which module it uses.

Programs written in Camllight have a single xxx.mlp source file which is compiled into the xxx-b, xxx-c, xxx-d, xxx-s and xxx-g binaries, the last letter showing which module it uses.

Programs written in Ocaml have a single `xxx.ml` source file which is compiled into the `xxx-byte` (compiled with `ocamlc`) and `xxx-opt` (compiled with `ocamlopt`) binaries. Each of those binaries contain all the compiled modules and a particular module is selected at run-time with the `-e` *module* option, as described in section 2.2.

Programs written in Pascal have a single `xxx.p` source file which is compiled into the `xxx-c`, `xxx-d` and `xxx-s` binaries, the last letter showing which unit it uses.

6.6.1 chrono

Speed measurement of the various modules (source in C only). This program chooses n and $2n$ bits random integers and measures the running time for various operations between those integers:

- `mul` : multiplication n bits by n bits;
- `sqr` : square of a number n bit long;
- `quomod`: division with remainder $2n$ bits by n bits;
- `quo` : division without remainder $2n$ bits by n bits;
- `sqrt` : square root of a $2n$ bit number;
- `gcd` : gcd of two n bit numbers;
- `gcd_ex`: gcd and Bézout coefficients for two n bit numbers.
- `all` : all operations above.

Specify on the command-line the value of n and which operations to do among `mul`, `sqr`, `quomod`, `quo`, `sqrt`, `gcd` and `gcd_ex`. You can specify a repeat count with a `-r` r option. For instance:

```
> exemples/chrono/c/chrono-g -all 1000000
    0.11      0.11 début
    1.15      1.04 mul
    1.84      0.69 sqr
    6.02      4.18 quomod
   10.23      4.21 quo
   40.09     29.86 sqrt
  242.48    202.39 gcd
  698.63    456.15 gcd_ex
> exemples/chrono/c/chrono-s -all 1000000
    0.23      0.23 début
    1.11      0.88 mul
    1.65      0.54 sqr
    6.50      4.85 quomod
   10.62      4.12 quo
   13.93      3.31 sqrt
   45.59     31.66 gcd
   86.28     40.69 gcd_ex
```

So, on the test computer (Pentium-233Mhz) and with the `Gmp-3` module, the time for multiplying two one million bit numbers is 1.04 seconds, the time for squaring a one million bit one is 0.69 seconds, etc. The second run shows the corresponding timings for the `Slong` module.

6.6.2 hamming

Computing of the k^{th} Hamming number or the rank of a Hamming number (source in Ocaml only). The Hamming problem and various algorithms that solve it are described in the French-written documents:

```
http://pauillac.inria.fr/~quercia/papers/hamming.ps.gz
http://pauillac.inria.fr/~quercia/papers/hamming-new.ps.gz
```

The program coming with the `Numerix` examples implements an algorithm unpublished with complexity $O(k^{1/3}\ln(k))$ (`hamming-new` implements a faster algorithm with complexity $O(\ln(k)^3)$). The command-line syntax is:

`hamming-xxx (`*k* `|` *x y z*`) [-d] [-v` *volume*`] [-e` *integers*`]`

where k is the rank of a Hamming number to be found and x, y, z are the exponents of $2, 3, 5$ in a Hamming number for which the rank is to be searched. `-d` asks for an output of the different program phases with timings, `-v` *volume* specifies the method for computing tetrahedron volumes (in-place = `surplace`, functional = `func`) and `-e` *integers* specifies which big integer module to use. For instance:

```
> exemples/hamming/ocaml/hamming-opt -e slong -d -v func 1T
    0.03     0.03 entiers    Slong
    0.03     0.00 volume     func
    0.03     0.00 Paramètres 12081378335  14699735488
    0.03     0.00            19148531618  23298529519
    0.03     0.00            28052091781  34131728817
    0.03     0.00 Approx     338940570505262 -> 1000267456452
    0.03     0.00 Approx     338910358662417 -> 1000000023838
    0.03     0.00 Approx     338910355969313 -> 1000000000001
    0.37     0.34 Exact      338910355969313 -> 1000000000001
    0.40     0.03 n2         338910355969346
    0.42     0.02 n0         338910355969286
    0.50     0.08 n1         338910355969190
    0.51     0.01 Dichotomie 8295 2426 6853
x = 8295  y = 2426  z = 6853
```

6.6.3 nextpp

Search for pseudo-prime numbers with the Rabin-Miller test (source in Ocaml, Caml, C and Pascal). This program receives two n and a integers (in this order) on the command-line and searches for the smallest $n + 1$ digits integer which is pseudo-prime in base a. For instance:

```
> time exemples/nextpp/c/nextpp-s 1000 2
453

real    0m32.111s
user    0m31.980s
sys     0m0.000s
> time exemples/nextpp/c/nextpp-g 1000 2
453

real    0m30.184s
user    0m30.050s
sys     0m0.010s
```

which means that $10^{1000} + 453$ is the smallest integer with 1001 digits pseudo-prime in base 2. This is the only program among all examples for which `Gmp-3` runs faster than `Slong`, because of a higher speed for computing the modular exponentiation.

6.6.4 pi

Computing the n first decimal digits of π (sources in C, Camllight, Ocaml and Pascal). This program implements the calculation of π described in the `BigNum` reference manual (*The Caml Numbers Reference Manual*, Inria, RT-0141) with a binary summation scheme to speed up the computing of the series (terms are added two by two, then 4 by 4, then 8 by 8 and so on). For instance:

```
> exemples/pi/ocaml/pi-opt -e slong -d 1000000 -skip
    0.03     0.03 Slong, surplace
    1.67     1.64 puiss 5
   19.75    18.08 sqrt
  122.44   102.69 série lb=6875847 ls=6875855
  150.43    27.99 quotient
  186.94    36.51 conversion
3.
14159 26535  89793 23846  26433 83279  50288 41971  69399 37510
... (19998 lignes omises)
56787 96130  33116 46283  99634 64604  22090 10610  57794 58151
```

6.6.5 simple

Simple demonstration program (sources in C, Camllight, Ocaml and Pascal). This program shows how to use the `Numerix` interfaces. It computes the n first decimal digits of $(\sqrt{3}+\sqrt{2})/(\sqrt{3}-\sqrt{2})$. For instance:

```
> exemples/simple/ocaml/simple-opt -e slong -n 50
9898979485566356196394568149411782783931894961313335
```

Chapter 7

Technical informations

7.1 Algorithms

7.1.1 Multiplication

`Numerix` implements four algorithms for multiplying two big numbers, each adapted to a specific operand size range.

`mul_2`: multiplication of a big integer by a single or double precision one. This multiplication performs a simple loop with a complexity $O(n)$ where n denotes the bit size of the big integer.

`mul_n2`: multiplication of two big integers, the sorter having at most `klim` digits. This multiplication performs a double loop with complexity $O(np)$ where n and p denote the operand bit sizes.

`karamul`: multiplication of two big integers of digit lengths between `klim` and `flim` with Karatsuba's "divide and conquer" algorithm. Let $n \leqslant p$ be the operand sizes: if $n \leqslant p/2$ one splits the p-digit operand into two smaller ones with lengths n and $p-n$ and computes recursively the corresponding products ($n \times n$ and $n \times (p-n)$). When $n = p$ Karatsuba's multiplication complexity is $O(n^{\ln(3)/\ln(2)})$.

`fftmul`: multiplication of two big integers of digit lengths n, p when $p \geqslant$ `flim` and `klim` $< n \leqslant p$ with Schönage's modular fast Fourier transform algorithm. The operands and the product are viewed as polynomials with coefficients belonging to $\mathbb{Z}/(2^{2^\ell}+1)\mathbb{Z}$ and the product of the two input polynomials is computed from its values at points 2^k, $0 \leqslant k < 2^{\ell+1}$ the set of which is a cyclic multiplicative subgroup of $(\mathbb{Z}/(2^{2^\ell}+1)\mathbb{Z})^*$. The parameter ℓ chosen depends on the size of the product to be computed as a *number* so as to minimize a rough estimate of the computing time. Usually $5 \leqslant \ell \leqslant 15$. The theoretical complexity of Schönage's algorithm is $O(n\ln(n))$ for a n digit product if ℓ can fit into a machine word and $O(n\ln(n)\ln(\ln n))$ otherwise (this case is unconceivable with today's computers). However the implementation in `Numerix` follows a more complicated scheme because of the way of choosing ℓ. Experiments show that the computing time is almost linear with respect to n up to ten million bits, above this limit the performance decreases because of memory problems.

The values of `klim` and `flim` were chosen by trial and error. For a 32 bits computer with digits of length 32 bits, the values are `klim` = 32 digits, `flim` = 937 digits that is to say that one switches from the naïve $O(n^2)$-multiplication to Karatsuba's one above 1024 bits and from Karatsuba's multiplication to Schönage's one above $30,000$ bits.

The algorithms for the naïve multiplication and Karatsuba's one are described in numerous books, for instance *Knuth, The art of computer programming, vol. 2.* Schönage's modular FFT multiplication is described in an article written by Paul Zimmermann:

`http://www.loria.fr/~zimmerma/bignum/schon.ps.gz`

These algorithms are also used for squaring, taking into account the simplifications resulting from the equality of the two multiplicands.

7.1.2 Division

`Numerix` implements three division algorithms each adapted to a specific operand size range.

`quo_2`: division of a big integer by a single or double precision one with a simple loop of complexity $O(n)$ where n denotes the dividend size.

`quo_n2`: division of two big integers with the "schoolbook" algorithm when the divisor digit size or the quotient one is smaller than `klim`. This division has complexity $O(np)$ where n denotes the divisor size and p denotes the quotient one.

`bzdiv`: division of two big integers with Burnikel and Ziegler algorithm, which is a "divide and conquer" version of the schoolbook algorithm. With this algorithm the time needed to divide a $2n$ bit number by an n bit one is approximately equal to twice the time of a n bits by n bits multiplication within Karatsuba's multiplication range, and approximately equal to $C\ln(n)$ n bits by n bits multiplications within the FFT multiplication range. Experiments show that up to ten million bits the time of a $2n$ bits by n bits division is at most five times greater than the time of a n bits by n bits multiplication.

Computing the integer square root of a big integer is done with similar algorithms: to compute $b = \lfloor\sqrt{a}\rfloor$ one divides a by b, the resulting quotient being b, $b+1$ or $b+2$. In order to perform this division one only needs to know the highest digit of b, the other digits are computed while doing the division. This highest digit of b is computed in an independent way with the Newton method. The time needed to compute the square root of a $2n$ bit number with this method is theoretically equal to half the time needed to perform a $2n$ bits by n bits division, that is one n bits par n bits multiplication within Karatsuba's multiplication range (confirmed by experiments), and $C\ln(n)/2$ n bits by n bits multiplications within the FFT multiplication range (time observed for 10^7 bits: 3 multiplications).

The conversion of a big integer into a decimal string can be done with successive divisions by 10, but this leads to an algorithm with complexity $O(n^2)$ for an n bit operand. Therefore `Numerix` uses another algorithm which consists in computing by successive squarings a power of 10: 10^{2^k} the bit length of which lies between $n/2$ and n, then in dividing the input number by 10^{2^k} and in converting recursively the quotient and the remainder with the powers of 10 already computed. Doing this way, the time of conversion is approximately twice the time of an $2n$ bits by n bits division within Karatsuba's multiplication range and $O(\ln(n))$ n bits by n bits multiplications within the FFT multiplication range.

Réferences

Schoolbook division: *Knuth, The art of computer programming, vol. 2*

Burnikel and Ziegler division: *Fast recursive division, Research report MPI-I-98-1-022, Max Planck Institut Für Informatik, octobre 1998*

Recursive square root (Zimmermann): *Karatsuba square root, INRIA research report no 3805,* `http://www.inria.fr/RRRT/RR3805.html`

Iterative implementations of Burnikel-Ziegler and Zimmermann algorithms:

Division et racine carrée rapides,

`http://pauillac.inria.fr/~quercia/papers/divrap.ps.gz`

7.1.3 Gcd

Much less optimization effort has been made in the computing of two integers gcd, only one algorithm is implemented in `Numerix` : a "divide and conquer" version of Lehmer's algorithm.

Lehmer's algorithm, described in Knuth's book, computes the gcd and the Bézout coefficients of two big integers a and b by expanding into a continued fraction the number a/b, the quotients are those of Euclide's algorithm applied to a and b. The quotients are computed from the leading digits α and β of a and b by simultaneously expanding into continued fractions $\alpha/(\beta+1)$ and $(\alpha+1)/\beta$. As $\alpha/(\beta+1) \leqslant a/b \leqslant (\alpha+1)/\beta$ the quotients for a/b are the same as those of $\alpha/(\beta+1)$ and $(\alpha+1)/\beta$ as long as the quotients of the two surrounding fractions are equal. When the first difference appears, one applies the corresponding homographic transform to a and b and goes on with the new numbers. This algorithm has the same asymptotic complexity as Euclide's algorithm, namely $O(n^2)$ for n bit operands, but most of the operations are performed on single precision numbers and it is approximately 20 times faster than Euclide's algorithm for big operands.

`Numerix` implements a “divide and conquer” version of Lehmer’s algorithm with α and β being the upper halves of a and b instead of their leading digits which results in a theoretical complexity of 18 n bits par n bits multiplication within Karatsuba’s multiplication range (confirmed by experiments), and $O(\ln(n))$ n bits by n bits multiplications within the FFT multiplication range (time observed for 10^6 bits: 45 multiplications). This recursive version of Lehmer’s algorithm is not described in any of the documents that I read, however Schönage gave a similar algorithm, called *half-gcd* or *descent*, described in:

Equation Solving in Terms of Computational Complexity,
Proceedings of the International Congress of Mathematicians, 1987

7.2 Differences between versions 0.13 and 0.17 of `Numerix`

`Numerix`-0.13 was the first public version, released in February 2000. The following versions 0.14, 0.15 and 0.16 were small fixes to compile problems pointed to me by some users, and this version, `Numerix`-0.17, is actually the second public version.

Kernel:

Complete rewrite of the C and assembly kernel so as to facilitate subsequent upgrades.

Bug fix in the fast division code (overflow in a subtraction, pointed to me by YAMAGATA yoriyuki).

Implementation of Schönage’s multiplication.

Caml/Ocaml interface:

Update for Ocaml-3.00.

Add integer p-th root (recursive Newton’s algorithm).

Add modular exponentiation.

The module `Cmp` is no longer a sub-module of `Numerix`, it is now an independent one: `Numcmp`.

Add module `Numcount`.

Merge the compiled code of `Numerix`, `Numcmp` and `Numcount` into the archives `numerix.cma`, `numerix.cmxa` instead of `numerix.cmo` and `numerix.cmx`.

7.3 Differences between versions 0.17 and 0.18 of `Numerix`

Kernel:

Bug fix for the division with a two digit divisor.

Modular exponentiation is now coded in C, it is still slower than `GMP` when the base is a big number though.

Fix the compile rules for OpenBSD-2.8.

C interface:

Add the `root` function.

Pascal interface:

First public release of the Pascal interface.

7.4 Differences between versions 0.18 and 0.19 of `Numerix`

Ocaml interface:

Remove the comparison operators between big integers from the `Infixes` module, now Ocaml usual polymorphic comparison operators can be used.

Add hash, serialization and deserialization functions.