Towards primitive data types for Coq: 63-bits integers and persistent arrays[1]

Maxime Dénès

Inria Sophia-Antipolis

July 22, 2013

[1] This work has been funded by the FORMATH project, nr. 243847, of the FET program within the 7th Framework program of the European Commission.

Motivation

Growing need for efficient computations in proof systems.

Motivation

Growing need for efficient computations in proof systems.

Typical uses:

- Proofs by computation: four color theorem, Kepler conjecture, certification of prime numbers

Motivation

Growing need for efficient computations in proof systems.

Typical uses:

- Proofs by computation: four color theorem, Kepler conjecture, certification of prime numbers
- Automation: deciding identities over rings, Kleene algebras, untrusted calls to external solvers

Motivation

Growing need for efficient computations in proof systems.

Typical uses:

- Proofs by computation: four color theorem, Kepler conjecture, certification of prime numbers
- Automation: deciding identities over rings, Kleene algebras, untrusted calls to external solvers
- Other uses: importing proof objects from other systems, emitting assembly code

Motivation

Growing need for efficient computations in proof systems.

Typical uses:

- Proofs by computation: four color theorem, Kepler conjecture, certification of prime numbers
- Automation: deciding identities over rings, Kleene algebras, untrusted calls to external solvers
- Other uses: importing proof objects from other systems, emitting assembly code

But also efficiency of extracted code: floating-point arithmetic,…

From efficient evaluation…

Evaluation of terms inside Coq has improved a lot over years:

From efficient evaluation…

Evaluation of terms inside Coq has improved a lot over years:

- Lazy evaluation machine

From efficient evaluation…

Evaluation of terms inside Coq has improved a lot over years:

- Lazy evaluation machine
- Bytecode-based dedicated compiler and virtual machine

From efficient evaluation…

Evaluation of terms inside Coq has improved a lot over years:

- Lazy evaluation machine
- Bytecode-based dedicated compiler and virtual machine
- Compilation to native code through OCaml

…to efficient data representations

But these efforts can be vain without adapted representations for data.

. . . to efficient data representations

But these efforts can be vain without adapted representations for data.

Especially important in a purely functional setting.
A symptomatic case: natural numbers.

. . . to efficient data representations

But these efforts can be vain without adapted representations for data.

Especially important in a purely functional setting.
A symptomatic case: natural numbers.

In the following, I will review the solutions implemented in Coq for efficient integer arithmetic and suggest some improvements.

Numbers in Coq: Peano numbers

Simple definition (`nat`), easy inductive reasoning.

Numbers in Coq: Peano numbers

Simple definition (`nat`), easy inductive reasoning.

But not very loyal to the user interested in computations.

Numbers in Coq: Peano numbers

Simple definition (`nat`), easy inductive reasoning.

But not very loyal to the user interested in computations.

```
Coq < Definition x := 10000.
Warning: Stack overflow or segmentation fault happens when
working with large numbers in nat (observed threshold may
vary from 5000 to 70000 depending on your system limits
and on the command executed).
x is defined
```

Numbers in Coq: binary numbers

```
Inductive positive : Set :=
  | xI : positive -> positive
  | xO : positive -> positive
  | xH : positive.

Inductive N : Set :=
  | N0 : N
  | Npos : positive -> N.
```

Slight complication for uniqueness of `0`, definition of positive binary numbers (`positive`) and then binary naturals (`N`).

Exponential gain in space and time, but still too limited for heavy computations.

Numbers in Coq: big naturals

Based on a representation of numbers as binary trees (`bigN`).

Numbers in Coq: big naturals

Based on a representation of numbers as binary trees (`bigN`).

Well-suited to the implementation of Karatsuba's product.

Numbers in Coq: big naturals

Based on a representation of numbers as binary trees (`bigN`).

Well-suited to the implementation of Karatsuba's product.

Performance rely critically on the inlining of operators for small trees (height ≤ 6), hence a fairly complex implementation.

Numbers in Coq: big naturals

Based on a representation of numbers as binary trees (`bigN`).

Well-suited to the implementation of Karatsuba's product.

Performance rely critically on the inlining of operators for small trees (height ≤ 6), hence a fairly complex implementation.

Leaves were first implemented by an inductive type with 256 constructors, but in recent versions they are substituted with native 31-bits integers.

Numbers in Coq: 31-bits integers

In the current version of Coq, 31-bits integers are represented by an inductive type:

```
Inductive digits : Type := D0 | D1.

Definition digits31 t :=
  Eval compute in nfun digits 31 t.

Inductive int31 : Type := I31 : digits31 int31.
```

Numbers in Coq: 31-bits integers

In the current version of Coq, 31-bits integers are represented by an inductive type:

```
Inductive digits : Type := D0 | D1.

Definition digits31 t :=
  Eval compute in nfun digits 31 t.

Inductive int31 : Type := I31 : digits31 int31.
```

But during an evaluation or a conversion using the VM, it is substituted with native machine arithmetic.

Numbers in Coq: 31-bits integers

In the current version of Coq, 31-bits integers are represented by an inductive type:

```
Inductive digits : Type := D0 | D1.

Definition digits31 t :=
  Eval compute in nfun digits 31 t.

Inductive int31 : Type := I31 : digits31 int31.
```

But during an evaluation or a conversion using the VM, it is substituted with native machine arithmetic.

This machinery is called “retroknowledge”.

Retroknowledge vs primitive data types

An alternative to "retroknowledge" would be to introduce in the formalism a primitive type `int`, with operators and axiomatized equational theory.

Retroknowledge vs primitive data types

An alternative to “retroknowledge” would be to introduce in the formalism a primitive type `int`, with operators and axiomatized equational theory.

With a major gain: benefits from machine arithmetic are not limited to the conversion test, but available in the whole system.

Retroknowledge vs primitive data types

An alternative to “retroknowledge” would be to introduce in the formalism a primitive type `int`, with operators and axiomatized equational theory.

With a major gain: benefits from machine arithmetic are not limited to the conversion test, but available in the whole system.

But a few drawbacks:

- Requires to enrich the formalism
- Does not give computational meaning to the axioms encoding the equational theory

Retroknowledge vs primitive data types

Still, we propose to replace “retroknowledge” by some primitive data types, because it is sometimes unavoidable:

- No guarantee that an inductive type can be defined to reflect the computational behavior
- Explicit constructions can be too costly to typecheck (outside conversion), or even to allocate!

Primitive persistent arrays

Persistent arrays have a functional interface, but internally use destructive arrays.

They maintain a history of changes made to elements in the array.

Primitive persistent arrays

Persistent arrays have a functional interface, but internally use destructive arrays.

They maintain a history of changes made to elements in the array.

We ported them as a primitive data type:

```
Register array : Type -> Type as array_type.
Register get : forall {A:Type}, array A -> int -> A
   as array_get.
Register set : forall {A:Type}, array A -> int -> A
   -> array A as array_set.
Register length : forall {A:Type}, array A -> int as
    array_length.
Register init : forall {A:Type}, int -> (int -> A)
   -> A -> array A as array_init.
Register map : forall {A B:Type}, (A -> B) -> array
   A -> array B as array_map.
```

Primitive 63-bits integers

Unlike OCaml, modular arithmetic libraries in Coq aim at being portable (i.e. have the same behavior on different architectures).

Primitive 63-bits integers

Unlike OCaml, modular arithmetic libraries in Coq aim at being portable (i.e. have the same behavior on different architectures).

In the previous setting, 31-bit arithmetic was native on 32-bits architectures and emulated (using 63-bits arithmetic) on 64-bits machines.

Primitive 63-bits integers

Unlike OCaml, modular arithmetic libraries in Coq aim at being portable (i.e. have the same behavior on different architectures).

In the previous setting, 31-bit arithmetic was native on 32-bits architectures and emulated (using 63-bits arithmetic) on 64-bits machines.

We propose to do it the other way around, and provide two implementations of an interface of unsigned 63-bits arithmetic:

- One relying on OCaml's int type, to be used on 64-bits architectures

Primitive 63-bits integers

Unlike OCaml, modular arithmetic libraries in Coq aim at being portable (i.e. have the same behavior on different architectures).

In the previous setting, 31-bit arithmetic was native on 32-bits architectures and emulated (using 63-bits arithmetic) on 64-bits machines.

We propose to do it the other way around, and provide two implementations of an interface of unsigned 63-bits arithmetic:

- One relying on OCaml's int type, to be used on 64-bits architectures
- The other using OCaml's `Int64` module to emulate 63-bits integers, to be used on 32-bits architectures

Primitive 63-bits integers

Unlike OCaml, modular arithmetic libraries in Coq aim at being portable (i.e. have the same behavior on different architectures).

In the previous setting, 31-bit arithmetic was native on 32-bits architectures and emulated (using 63-bits arithmetic) on 64-bits machines.

We propose to do it the other way around, and provide two implementations of an interface of unsigned 63-bits arithmetic:

- One relying on OCaml's int type, to be used on 64-bits architectures
- The other using OCaml's `Int64` module to emulate 63-bits integers, to be used on 32-bits architectures

The implementation is chosen when Coq is compiled.

Primitive 63-bits integers

We plugged the standard kernel conversion and the reduction by compilation to native code to this native 63-bits arithmetic (the VM will follow).

As a byproduct, we started to re-think the design of the `BigN` library.

A new library of big numbers?

With the use of 63-bits integers, typical applications require less machine words to represent numbers that then used to.

A new library of big numbers?

With the use of 63-bits integers, typical applications require less machine words to represent numbers that then used to.
The tree representation underlying `BigN` may no longer be relevant.

A new library of big numbers?

With the use of 63-bits integers, typical applications require less machine words to represent numbers that then used to.
The tree representation underlying `BigN` may no longer be relevant.

We implemented a prototype library based on lists of 63-bits integers:

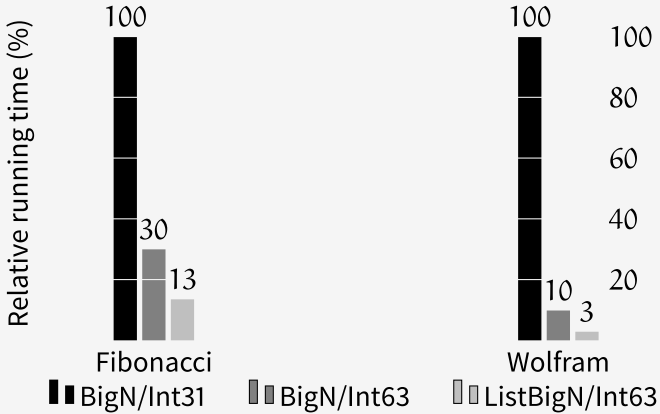
A new library of big numbers?

With the use of 63-bits integers, typical applications require less machine words to represent numbers that then used to.
The tree representation underlying `BigN` may no longer be relevant.

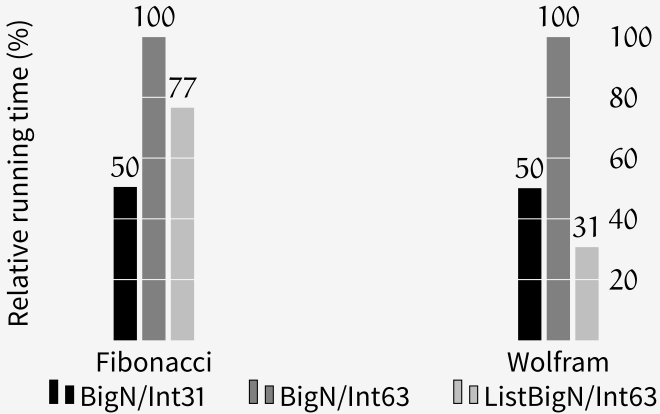
We implemented a prototype library based on lists of 63-bits integers:

```
Definition big_nat := list int.
Fixpoint add_big (a b : big_nat) (c : bool) :=
  match a, b with
   | nil, nil => if c then 1 :: nil else nil
   | i :: is, nil => if c then succ a else a
   | nil, j :: js => if c then succ b else b
   | i :: is, j :: js => if c then
     let r := i + j + 1 in r :: add_big is js (r <= i)
     else let r := i + j in r :: add_big is js (r < i)
  end.
Definition add (a b : big_nat) := add_big a b false.
```

Benchmarks (64-bits architecture)



Benchmarks (32-bits architecture)



Conclusion

Three proposals:

- Replace “retroknowledge” with primitive data types
- Switch from 31-bits to 63-bits arithmetic
- Design a simpler and more efficient library for big numbers

Moral of the story:

- Better verify a computation with a slightly bigger trusted base than not verify it at all
- Often valuable to periodically re-think the design of parts of a system (here Coq) to follow the evolution of hardware and software components

Conclusion

Remaining issues:

- Modular and flexible extension of the trusted base
- Unverified ad-hoc code in the parser and the printer for big numbers

Thank you!