Formally Verified Electronic Voting Scheme : A Case Study

Mukesh Tiwari

A thesis submitted for the degree of
YOUR DEGREE NAME
The Australian National University

July 2019

Except where otherwise indicated, this thesis is my own original work.

Mukesh Tiwari
4 July 2019

to my xxx, yyy (yyy is the people you want to dedicated this thesis to.)

Acknowledgments

This thesis could not have been possible without the support of my supervisor, Dirk Pattinson. I really admire his ability to understand the problem, and his intuition to to make sure that I stay clear from many dead ends which I would have happily spent months. I wish I could incorporate more of his qualities, but I believe I have less optimism about my chance.

Abstract

Put your abstract here.

Contents

Acknowledgments **vii**

Abstract **ix**

1 Introduction to Coq **1**
1.1 Theoretical Foundation of Coq 2
1.2 Coq in Action (Example) 2
1.3 Thesis Outline 4

2 Background and Related Work **5**
2.1 Motivation 5
2.2 Related work 5
2.3 Summary 5

3 Design and Implementation **7**
3.1 Smart Design 7
3.2 Summary 7

4 Experimental Methodology **9**
4.1 Software platform 9
4.2 Hardware platform 9

5 Results **11**
5.1 Direct Cost 11
5.2 Summary 11

6 Conclusion **13**
6.1 Future Work 13

List of Figures

4.1 Hello world in Java and C. 10

5.1 The cost of zero initialization 12

List of Tables

4.1 Processors used in our evaluation. 9

Chapter 1

Introduction to Coq

Write about Hilbert's idea of mathematical formalism

A proof assistant is a computer program which assists users in development of mathematical proofs. The idea of developing mathematical proofs using computer goes back to Automath (automating mathematics) [cite Automath] and LCF [cite Logic for computation] project. The Automath project (1967 until the early 80's) was initiative of De Bruijn, and the aim of the project was to develop a language for expressing mathematical theories which can be verified by aid of computer. Automath was first practical project to exploit the Curry-Howard isomorphism (proofs-as-programs and formulas-as-types) [reference here]. DeBruijn was likely unaware of this correspondence, and he almost re-invented it ([Wiki entry on Curry-Howard]). Many researchers refers Curry-Howard isomorphism as Curry-Howard-DeBruijn isomorphism. Automath project can be seen as the precursor of proof assistants NuPrl [cite here] and Coq [cite coq]. Some other notable proof assistants are LCF (Logic for Computable Functions) [cite Milner?], Mizar [cite], Nqthm/ACl2 [cite], PVS [cite], HOL (a family of tools derived from LCF theorem prover), Agda [cite], and Lean [cite].

The Coq proof assistant is interactive theorem prover based on underlying theory of Calculus of Inductive Construction [Cite Pualine Mohring] which itself is a augmentation of Calculus of Construction [cite Huet and Coquand] with inductive data-type. Coq provides a highly expressive specification language Gallina for development of mathematical theories and proving the theorems about these theories. Even though Gallina is very expressive, writing proofs in Gallina is very tedious and cumbersome. In order to ease the proof development, Coq also provides tactics. The user interacting with Coq applies these tactics to build the Gallina term which otherwise would be very laborious. In this chapter, I will give a brief overview of Calculus of Construction, followed by Calculus of Inductive Construction, with a example of building proof directly using Gallina and show that how same proof can be build easily using the tactics provided by Coq. In final section, I will try to justify my decision of using Coq for verifying Schulze method.

1.1 Theoretical Foundation of Coq

Calculus of Construction is .

write here Coc + CIC and typing inference

1.2 Coq in Action (Example)

Now I give a small example which defines natural number, addition of two natural numbers, and proof that addition over natural number is commutative. We can define natural number in Coq using inductive data type (listing 1.1), addition of the natural numbers (listing 1.2), and proof that addition of natural numbers is commutative written in Gallina (listing 1.3).

```
Inductive Natural : Type :=
 | O : Natural
 | Succ : Natural -> Natural
```

Listing 1.1: Inductive Data Type for Natural Numbers

More precisely, the interpretation is that Natural is a inductive type with two constructors: i) O representing zero, and ii) Succ representing successor which takes a Natural number and gives next Natural number.

Change the Addition into infix symbol + and use + in proofs. It will convey the idea more clearly

```
Fixpoint Addition (n m : Natural) : Natural :=
  match n with
  | O => m
  | Succ n' => Succ (Addition n' m)
  end.

(* Notation for Addition. Now we can use + instead of
   writing Addition *)
Notation "x␣+␣y" := (Addition x y)
            (at level 50, left associativity).
```

Listing 1.2: Addition function for Natural Numbers

We define the addition by pattern matching on first argument **n**. When **n** is O (zero), then we sum is **m**, and if **n** is **Succ n'**, then sum is successor of **n' + m**.

```
Theorem Addition_by_zero : forall (n : Natural), n + O = n.
  refine (fix IHa (n : Natural) : n + O = n :=
            match n as nz return (nz + O = nz) with
            | O => eq_refl
            | Succ n' =>
```

```
              let IHn' := IHa n' in
              eq_ind_r (fun m => Succ m = Succ n') eq_refl IHn'
            end).
Qed.


Lemma Successor_addition : forall (n m : Natural),
    Succ (n + m) = n + (Succ m).
  refine
    (fix IHn (n : Natural) : forall m : Natural,
        Succ (n + m) = n + (Succ m) :=
       match n as nz return (forall m : Natural,
                               Succ (nz + m) =
                               nz + (Succ m)) with
       | O => fun m : Natural => eq_refl
       | Succ n' =>
         fun m : Natural =>
           eq_ind (Succ (n' + m))
                  (fun t => Succ (Succ (n' + m)) = Succ t)
                  eq_refl (n' + (Succ m)) (IHn n' m)
       end).
Qed.



Theorem Addition_is_commutative :
  forall (n m : Natural), n +  m = m + n.
  refine
    (fix IHn (n : Natural) : forall m : Natural,
        n + m = m + n :=
       match n as nz return (forall m : Natural,
                               nz + m =
                               m + nz) with
       | O => fun m : Natural => eq_ind_r (fun t => m = t)
                                    eq_refl
                                    (Addition_by_zero m)
       | Succ n' =>
         fun m  =>
           eq_ind (Succ (m + n'))
                  (fun t => Succ (n' + m) = t)
                  (eq_ind_r (fun t => Succ t = Succ (m + n'))
                            eq_refl (IHn n' m))
                  (m + (Succ n'))
                  (Successor_addition m n')
```

```
        end).
Qed.
```

Listing 1.3: Addition function for Natural Numbers

We need two additional Lemma: i) *Addition_by_zero*, a proof of n + 0 = 0, and ii) *Successor_addition*, a proof of Succ (n + m) = n + (Succ m) to prove that addition on Natural is commutative (*Addition_is_commutative*),

One thing that can't escape from the reader's eyes is that the proofs written in Gallina is verbose, and they don't appear anywhere compared to a proof that would have been written by a mathematician. Well, we can lift the burden of verbosity by using tactics provided by Coq; however, there is no universally accepted solution in Coq community for second problem. There has been some research in declarative style proof writing [1], but it is not widely practised in Coq community. The proof of addition on Natural is commutative re-written using tactics (Listing 1.4)

```
Lemma Addition_by_zero : forall (n : Natural), n + O = n.
  induction n; cbn; [auto | rewrite IHn; auto].
Qed.


Lemma Successor_addition : forall (n m : Natural),
    Succ (n + m) = n + (Succ m).
Proof.
  induction n; intros m;
    cbn; [auto | rewrite <- IHn; auto].
Qed.


Theorem Addition_is_commutative :
  forall (n m : Natural), n +  m = m + n.
Proof.
  induction n; intro m;
    [rewrite Addition_by_zero |
     rewrite <- Successor_addition;
     rewrite <- IHn]; auto.
Qed.
```

Listing 1.4: Addition function for Natural Numbers

1.3 Thesis Outline

How many chapters you have? You may have Chapter 2, Chapter 3, Chapter 4, Chapter 5, and Chapter 6.

[1] www-verimag.imag.fr/ corbinea/ftp/publis/bricks-poster.pdf

Chapter 2

Background and Related Work

At the begging of each chapter, please introduce the motivation and high-level picture of the chapter. You also have to introduce sections in the chapter.

Section 2.1 xxxx.

Section 2.2 yyyy.

2.1 Motivation

2.2 Related work

You may reference other papers. For example: Generational garbage collection [Lieberman and Hewitt, 1983; Moon, 1984; Ungar, 1984] is perhaps the single most important advance in garbage collection since the first collectors were developed in the early 1960s. (doi: "doi" should just be the doi part, not the full URL, and it will be made to link to dx.doi.org and resolve. shortname: gives an optional short name for a conference like PLDI '08.)

2.3 Summary

Summary what you discussed in this chapter, and mention the story in next chapter. Readers should roughly understand what your thesis takes about by only reading words at the beginning and the end (Summary) of each chapter.

Chapter 3

Design and Implementation

Same as the last chapter, introduce the motivation and the high-level picture to readers, and introduce the sections in this chapter.

3.1 Smart Design

3.2 Summary

Same as the last chapter, summary what you discussed in this chapter and be the bridge to next chapter.

Chapter 4

Experimental Methodology

4.1 Software platform

4.2 Hardware platform

Table 4.1 shows how to include tables and Figure 4.1 shows how to include codes.

Architecture	Pentium 4	Atom D510	i7-2600
Model	P4D 820	Atom D510	Core i7-2600
Technology	90nm	45nm	32nm
Clock	2.8GHz	1.66GHz	3.4GHz
Cores × SMT	2 × 2	2 × 2	4 × 2
L2 Cache	1MB × 2	512KB × 2	256KB × 4
L3 Cache	none	none	8MB
Memory	1GB DDR2-400	2GB DDR2-800	4GB DDR3-1066

Table 4.1: Processors used in our evaluation.

```c
int main(void)
{
  printf("Hello World\n");
  return 0;
}
```

(a)

```java
void main(String[] args)
{
  System.out.println("Hello World");
}
```

(b)

Figure 4.1: Hello world in Java and C.

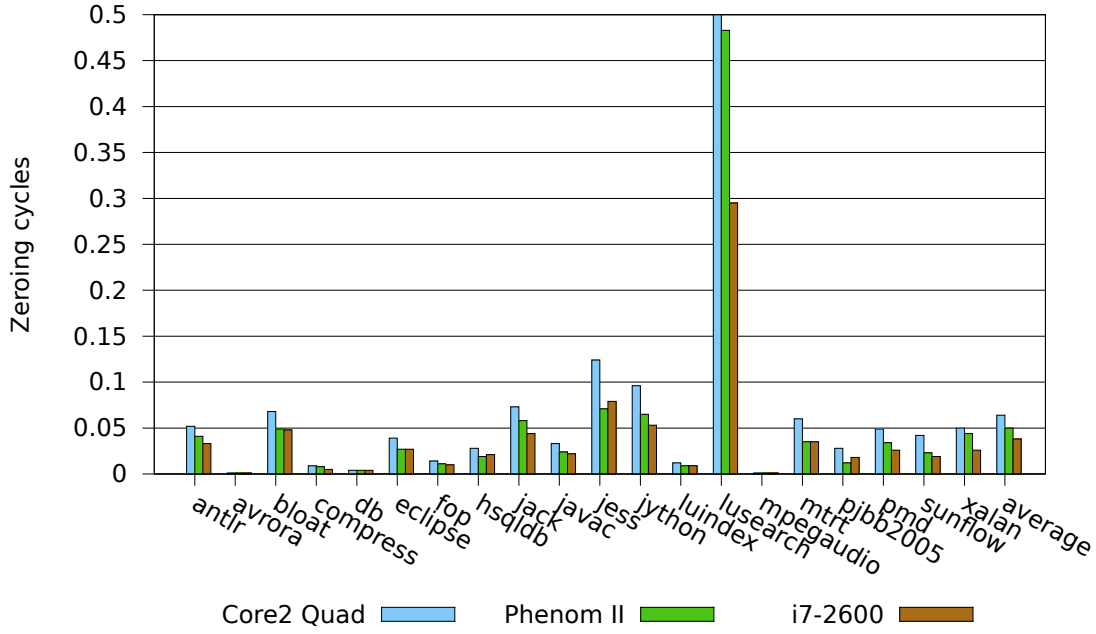
Chapter 5

Results

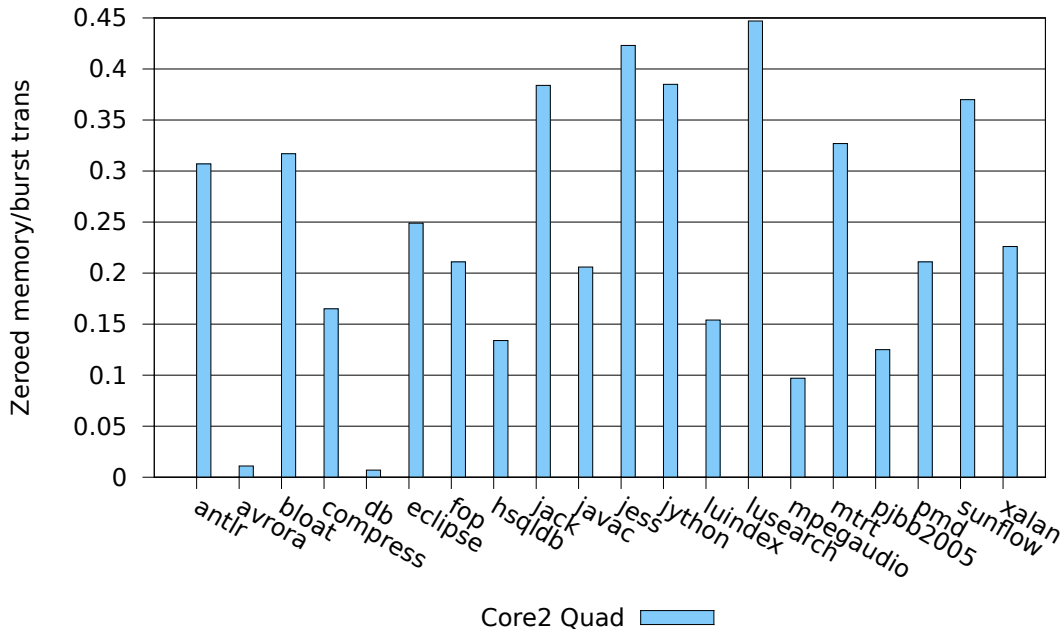
5.1 Direct Cost

Here is the example to show how to include a figure. Figure 5.1 includes two subfigures (Figure 5.1(a), and Figure 5.1(b));

5.2 Summary



(a) Fraction of cycles spent on zeroing



(b) BytesZeroed / BytesBurstTransactionsTransferred

Figure 5.1: The cost of zero initialization

Chapter 6

Conclusion

Summary your thesis and discuss what you are going to do in the future in Section 6.1.

6.1 Future Work

Good luck.

Bibliography

Geuvers, H., 2009. Proof assistants: History, ideas and future. *Sadhana*, 34, 1 (Feb 2009), 3–25. doi:10.1007/s12046-009-0001-5. https://doi.org/10.1007/s12046-009-0001-5.

Lieberman, H. and Hewitt, C., 1983. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26, 6 (Jun. 1983), 419–429. doi:10.1145/358141.358147. (cited on page 5)

Moon, D. A., 1984. Garbage collection in a large LISP system. In LFP '84: *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming* (Austin, Texas, USA, Aug. 1984), 235–246. ACM, New York, New York, USA. doi:10.1145/800055.802040. (cited on page 5)

Ungar, D., 1984. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In SDE 1: *Proceedings of the 1st ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Pittsburgh, Pennsylvania, USA, Apr. 1984), 157–167. ACM, New York, New York, USA. doi:10.1145/800020.808261. (cited on page 5)