КОМП'ЮТЕРНІ НАУКИ

UDC 004.4'22 *doi: 10.30970/vam.2023.31.11967*

THE COQ-MODEL OF ATOMIC ENTITIES IN FORMAL THEORIES

O. Berezovskyi[1], **A. Zozulia**[2]

[1]*Ivan Franko National University of Lviv,*
1, Universytetska str., 79000, Lviv, Ukraine
e-mail: berezovskyi.oleksandr@gmail.com
[2]*V.N. Karazin Kharkiv National University,*
4, Svobody sqr, 61022, Kharkiv, Ukraine
e-mail: anna.zozulia@karazin.ua

Atomic entities are significant basic elements of the logical modeling of subject areas of software systems. Since modern software systems require guarantees of the correctness of their functioning, formal logical models become a necessary tool for ensuring software dependability.

The article proposes a conceptual basis for modeling atomic entities, developed for using it for formal verification with The Coq Proof Assistant.

Atomic entities are the smallest, indivisible units of meaning in a logical model. They are the building blocks of more complex concepts and relationships. For example, in calculi, atomic entities are constants and variables. Modern software systems are complex and often involve multiple concurrent processes. This can make it difficult to ensure that the system will always behave correctly, even in the presence of unexpected events or errors. Formal logical models can help to address this challenge by providing a way to reason about the system's behavior rigorously and systematically.

Key words: formal verification, dependent types, The Coq Proof Assistant, atomic entities, formal proof.

1. Introduction

The use of Information and Communication Technology has become ubiquitous, and this has led to an increase, in the requirements for software dependability. Modern applications are often complex and distributed. This can make it difficult to analyze their behaviour using traditional methods. Traditional dynamic analysis methods, such as debugging and testing, can be untrustworthy when applied to modern applications. The reason is the necessary tools can significantly affect the system behaviour under study. For example, a debugger can introduce delays or other artefacts that can change the way the application runs. This can raise difficulties in identifying and fixing bugs.

The mentioned situation can be solved by the use of formal verification methods and the corresponding software tools. Formal verification is an approach based on mathematically proving the correctness of a system. The approach is for verifying the correctness of hardware and software systems, protocols, and other mathematical aspects of system design. Formal verification is based on mathematical techniques for specifying and analyzing systems. Formal methods suggest creating some formal system model and analyzing it further to prove its properties specified as system requirements.

One fruitful technique for formal verification is dependent type theory, on which a series of software verification tools such as The Coq Proof Assistant [1], Isabelle [2], Agda [3], and others are based.

In the introduction of his book [4], A. Chlipala explained reasons to use The Coq Proof Assistant for the certified programming technique. Due to this argumentation, we decided to use The Coq Proof Assistant for our paper. Our decision is motivated by the use experience of The Coq Proof Assistant in a lot of projects for

1. verifying software: Coq is used to verify the correctness of software systems, such as compilers, operating systems, and security protocols. For example, the CompCert compiler (http://compcert.org/) and the CertiKOS operating system (http://flint.cs.yale.edu/certikos/) have been verified using Coq;
2. mathematical proofs: Coq is used to develop formal proofs of mathematical theorems. For example, the link http://github.com/coq-community/fourcolor refers to the repository containing Coq-scripts for prooving the Four Color Theorem and the link http://github.com/flyspeck/flyspeck refers to the repository containing Coq-scripts solving the Kepler Conjecture;
3. teaching and research: Coq is used to teach and research formal mathematics and computer science. Many universities around the world offer courses on Coq, and there is a large and active research community working on Coq and related topics.

The problem of this paper is modelling atomic entities of logical models used for specifying and analysing software. The problem is motivated by the research project that the Theoretical and Applied Computer Science Department of V.N. Karazin Kharkiv National University carries out jointly with research team KAIROS of INRIA (France). This project aims to provide formal semantics and verification methods for Clock Constraint Specification Language (CCSL) [5–7] used under design embedded and cyber-physical systems.

Such atomic entities are present in the majority of logical models. For example, such entities are known as variables in Hoare logic [8], as constants and variables in different lambda calculi [9] and so on.

In the paper, the principal elements of the library for operating with such entities using The Coq Proof Assistant are presented. All mentioned in the paper's formal definitions and proofs have been checked with Coq Platform v8.16 or later.

2. Core Model

Let us begin our presentation with a discussion of natural requirements for models of atomic entities called below atoms. These requirements are the following:

1. there exists an algorithm for distinguishing atoms from other entities;
2. there exists an algorithm for checking the equality of atoms;
3. at each specific moment of the use of a model, a finite set of atoms is available only;
4. at each specific moment of the use of a model, a new atom can be introduced, which becomes available after the introduction.

It is evident that the type of natural numbers is the ideal candidate for satisfying these requirements. But this type has many properties related to Peano arithmetic that are redundant for our purposes. Therefore, we introduce the new type `atom` by wrapping the natural-valued identifier.

```
(* An atom is an entity uniquely defined by a corresponding
   natural number.                                                 *)
Inductive atom : Set := a : nat -> atom.
```

For each atom, we have the manner to determine its identifier as follows

```
(* This function returns that natural number defining an atom.       *)
Definition aid : atom -> nat := fun n => let 'a idn := n in idn.
```

Easy to see that the function aid is bijective. This follows directly from the next lemmas.

```
Lemma aid_inj : forall n m : atom, aid n = aid m -> n = m.
(* Equality of atoms implies equality of natural numbers defining
   the atoms.                                                      *)
Proof (* has been verified by Coq *).
  intros. destruct n as [idn], m as [idm]. simpl in H. now rewrite H.
Qed.

Lemma aid_surj : forall n : atom, exists idn, n = a idn.
(* Any atom is determined by a natural number.                     *)
Proof (* has been verified by Coq *).
  intro. destruct n as [idn]. now exists idn.
Qed.
```

Also, these lemmas demonstrate that items 1) and 2) of the declared requirements are fulfilled due to these requirements are valid for natural numbers.

3. Finite Sets of Atoms

In this section, we focus on items 3) and 4) of the requirements mentioned in the previous section. Hence, we need to propose a manner for representing finite sets of atoms and a function for defining and manipulating such sets.

3.1. How to Represent a Finite Set of Atoms?

A finite set can be constructively defined by enlisting its elements. Therefore, a natural tool for representing such a set is a list. Unfortunately, the direct use of lists leads to anomalies: elements of a list can be duplicated, and two lists that are distinguished by the order of their elements only represent the same set. Thus, we propose to use lists of atoms sorted by increasing identifiers of their members for representing finite atom sets to eliminate these anomalies.

We define the predicate increasing as follows

```
Inductive increasing : list atom -> Prop :=
  (* a list of atoms is increasing if either                       *)
  | inc0 : increasing []           (* the list is empty or         *)
  | inc1 : forall n, increasing [n]  (* it contains only one member or *)
  | incS : forall n m ns,          (* it is built by pushing an atom *)
      aid n < aid m ->             (* whose identifier is less than  *)
      increasing (m :: ns) ->      (* the head identifier of an      *)
        increasing (n :: m :: ns).   (* increasing list into this list *)
```

This definition establishes that empty and one-element lists of atoms are increasing. A more complicated case is defined by the third clause of the inductive definition: we obtain an increasing list by adding to the head of an increasing one under the condition that the identifier of this head is less than the identifier of the adding atom.

An expected property of this predicate is its decidability and the certified procedure for distinguishing increasing and non-increasing lists is the following.

```
Definition increasing_dec :
  forall lst : list atom, {increasing lst} + {~ increasing lst}.
Proof (* has been verified by Coq *).
  intro.
  (* Now, let us consider two cases: the list is empty and not empty.  *)
  destruct lst as [| n lst'].
  - (* the first case is trivial                                       *)
   left. constructor.
  - (* in the second case, we apply induction on the list lst'         *)
   revert n. induction lst' as [| m lst'' IHlst'']; intro.
    + (* proving the induction base is trivial                         *)
      left. constructor.
    + (* for proving the induction step, we need to consider all cases
         of outcomes under comparison of aid n and aid m               *)
      destruct (lt_eq_lt_dec (aid n) (aid m)) as [Hle | Hgt];
      try destruct Hle as [Hlt | Heq].
      * (* case aid n < aid m                                          *)
        { elim (IHlst'' m); intro H.
        - left. now constructor.
        - right. intro H1. apply H. now inversion_clear H1. }
      * (* case aid n = aid m                                          *)
        right. intro H. inversion_clear H. rewrite Heq in H0.
        now apply Nat.lt_irrefl with (aid m).
      * (* case aid n > aid m                                          *)
       right. intro H. inversion_clear H.
        apply Nat.lt_irrefl with (aid m).
        now apply Nat.lt_trans with (aid n).
Defined.
```

Note that the Coq script above demonstrates an object definition method through proof of its existence. Such a method is a significant The Coq Proof Assistant feature. This feature is grounded by the Curry-Howard correspondence for Calculus of Inductive Constructions, which is the mathematical base of The Coq Proof Assistant [10].

Now, we are ready to introduce the type `AtomSet` for representing finite sets of atoms.

```
Definition AtomSet : Set := {lst : list atom | increasing lst}.
Coercion toList := fun ns : AtomSet => proj1_sig ns.
```

That is, the type `AtomSet` is inhabited by lists of atoms equipped with the certificate (evidence, proof) of their increasing. The coercion `toList` ensures simplifying manipulations with such lists.

3.2. How to Build a Finite Set of Atoms?

Now we have a method of representing finite sets of atoms, but constructing a specific atom set is complicated: first, it is necessary to create an atom list; second, to sort it by increasing the identifiers of its members; third, to build the appropriate certificate; and only after that create a term of type `AtomSet`.

In contrast, we propose the certified function

```
inject : atom -> AtomSet -> AtomSet
```

for inserting an atom into a set of atoms.

We begin our construction by defining an auxiliary function

```
aux_inject : atom -> list atom -> list atom
```

that inserts an atom into an atom list before its first member whose identifier is greater than the identifier of the inserting atom.

```
Fixpoint aux_inject (n : atom) (lst : list atom) : list atom :=
    match lst with
    | []        => [n]
    | m :: lst' => match (lt_eq_lt_dec (aid n) (aid m)) with
        | inleft Hle => match Hle with
            | left _  => n :: m :: lst'
            | right _ => m :: lst'
            end
        | inright _  => m :: (aux_inject n lst')
        end
    end.
```

Now, we are ready to define the function `inject`.

```
Definition inject (n : atom) (ns : AtomSet) : AtomSet.
Proof (* has been verified by Coq *).
  (* let us destruct ns : AtomSet into the atom list and
     the certificate that it increases                                   *)
  destruct ns as (lst, H).
  (* build atom list nlst using aux_inject                               *)
  pose (aux_inject n lst) as nlst.
  (* build atom set based on nlst                                        *)
  exists nlst. subst nlst.
  (* prove that nlst increases                                           *)
  destruct lst as [| m lst'].
  - constructor.
  - simpl. destruct (lt_eq_lt_dec (aid n) (aid m)) as [Hle | Hgt];
    try destruct Hle as [Hlt | Heq].
    + now constructor.
    + assumption.
    + revert n m H Hgt. induction lst' as [| k lst'' IHlst''].
```

```
      * constructor; [assumption | constructor].
      * intros. {
        simpl. destruct (lt_eq_lt_dec (aid n) (aid k)) as [Hle | Hgt'];
        try destruct Hle as [Hlt | Heq].
        - constructor; [ assumption | constructor ]; try assumption.
          now inversion_clear H.
        - assumption.
        - inversion_clear H.
          constructor; try assumption. now apply IHlst''. }
Defined.
```

Note that inject does not insert an atom into the atom set if this atom is already in the atom set.

Now, let us present two lemmas that demonstrate the expected behaviour of the function inject.

The first lemma demonstrate that an atom is a member of an atom set after injecting it into this atom set.

```
Lemma post_inject : forall n ns, In n (inject n ns).
Proof (* has been verified by Coq *).
  intros. revert n.
  destruct ns as (lst, H).
  induction lst as [| m lst' IHlst']; intro.
  - now left.
  - simpl.
    destruct (lt_eq_lt_dec (aid n) (aid m)) as [Hle | Hgt];
    try destruct Hle as [Hlt | Heq].
    + now left.
    + left. now apply aid_inj.
    + right.
      assert (increasing lst'). {
        inversion_clear H; [ constructor | assumption ]. }
      now apply IHlst'.
Qed.
```

The second lemma demonstrates that an atom belonging to the atom set obtained by injecting some atom into an atom set is either equal to the injected atom or belongs to the original atom set.

```
Lemma post_inject_discr : forall n m ns,
  In m (inject n ns) -> m = n \/ In m ns.
Proof (* has been verified by Coq *).
  intros until ns. revert m n.
  destruct ns as (lst, H).
  induction lst as [| k lst' IHlst']; intros * H1.
  - elim H1; intro H2; [ now left | contradiction ].
  - simpl in H1 |-*.
    assert (H2 : increasing lst'). {
      inversion_clear H; [ constructor | assumption ]. }
    simpl in IHlst'. pose (IH := IHlst' H2).
    destruct (lt_eq_lt_dec (aid n) (aid k)) as [Hle | Hgt];
    try destruct Hle as [Hlt | Heq].
    + elim H1; intro H3.
      * left. now symmetry.
      * destruct H3 as [HL | HR]; right; [ now left | now right ].
    + destruct (lt_eq_lt_dec (aid k) (aid m)) as [Hle | Hgt];
      try destruct Hle as [Hlt | Heq'];
      try (right; inversion_clear H1; [ now left | now right ]).
    + inversion_clear H1.
      * right. now left.
      * elim (IH m n H0); intro; [ now left | right ]; now right.
Qed.
```

Thus, inject is a universal tool for constructing any finite atom set by sequentially inserting the required atoms. Using inject allows you to not worry about the order of inserting members into the resulting atom set.
However, sometimes we need to form atom sets whose member identifiers are sequential finite series of natural numbers. The function

```
segment : nat -> nat -> AtomSet
```

where the first argument is the identifier of the first member of the series and the second argument is the length of the series solves this problem.

To define this function, we first introduce an auxiliary function

```
Fixpoint aux_segment (base len : nat) {struct len} : list atom :=
  match len with
  | 0      => []
  | S len' => (a base) :: aux_segment (S base) len'
  end.
```

Then we prove that this function forms a sorted atom list.

```
Lemma aux_segment_inc : forall base len, increasing (aux_segment
 base len).
Proof (* has been verified by Coq *).
  intros. revert base.
  induction len as [| len' IHlen']; intro.
  - constructor.
  - simpl. destruct len' as [| len''].
    + constructor.
    + assert (increasing (aux_segment (S base) (S len''))). {
        pose (IHlen' (S base)). assumption. }
      assert (base < S base). { constructor. }
      simpl. constructor.
      * assumption.
      * simpl in H. assumption.
Qed.
```

And finally, we define the required function.

```
Definition segment (base len : nat) : AtomSet.
Proof (* has been verified by Coq *).
  exists (aux_segment base len).
  apply aux_segment_inc.
Defined.
```

This function satisfies the following two properties, which we state without proofs since the corresponding proofs are sufficiently cumbersome.

```
Lemma in_segment : forall base len n,
  In n (segment base len) -> base <= (aid n) /\ (aid n) < base + len.

Lemma segment_in : forall base len n,
  base <= aid n -> aid n < base + len -> In n (segment base len).
```

These lemmas demonstrate the correctness of the definition of `segment`.

4. Conclusion

Thus, in the paper, the model of atomic entities for formal theories has been presented. The model has been developed using The Coq Proof Assistant. The obtained results can be organised as Coq-Library for further use in developing formal models for verifying software systems.

References

1. The Coq Development Team. The Coq Proof Assistant. – Access mode: http://coq.inria.fr/
2. Naraschewski W. Isabelle/HOL: a proof assistant for higher-order logic / W. Naraschewski, T. Nipkow. – Access mode: http://www.cl.cam.ac.uk/research/hvg/Isabelle/
3. The Agda Team. The Agda Wiki. – Access mode: https://wiki.portal.chalmers.se/agda/Main/HomePage
4. Chlipala A. Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant / A. Chlipala. – The MIT Press, 2022.
5. Mallet F. The clock constraint specification language for building timed causality models / F. Mallet // Innovations in Systems and Software Engineering. – 2010. – Vol. 6. – P. 99–106.
6. Mallet F. Coalgebraic Semantic Model for the Clock Constraint Specification Language / F. Mallet, G. Zholtkevych, C. Artho, P. "Olveczky //Formal Techniques for Safety-Critical Systems. FTSCS 2014. Communications in Computer and Information Science. Springer, Cham. – 2014. – Vol. 476. – P. 174–188.
7. Mallet F. Time: It is only Logical! / F. Mallet, J.P. Bowen, Q. Li, Q. Xu // Theories of Programming and Formal Methods. Lecture Notes in Computer Science. Springer, Chem. – 2023. – 14080. – P. 323–347.
8. Hoare C.A.R. An axiomatic basis for computer programming. / C.A.R. Hoare // CACM. – 1969. – Vol. 12 (10). – P. 576–580.
9. Hindley J.R. Lambda-Calculus and Combinators: An Introduction / J.R. Hindley, J.P. Seldin. – Cambridge University Press, 2008, 2ed.

10. Pierce B.C. Logical Foundations. Series "Software Foundations", vol. 1 / B.C. Pierce. – Electronic textbook, 2023. – Access mode: http://softwarefoundations.cis.upenn.edu

Article: received 18.09.2023
revised 04.10.2023
printing adoption 25.10.2023

COQ-МОДЕЛЬ АТОМАРНИХ СУТНОСТЕЙ У ФОРМАЛЬНИХ ТЕОРІЯХ

О. Березовський[1], **А. Зозуля**[2]

[1]*Львівський національний університет імені Івана Франка,*
вул. Університетська 1, Львів, 79000
e-mail: berezovskyi.oleksandr@gmail.com
[2]*Харківський національний університет імені В.Н. Каразіна,*
майдан Свободи 4, Харків, 61022
e-mail: anna.zozulia@karazin.ua

Атомарні сутності є важливими базовими елементами логічного моделювання предметних областей програмних систем. Оскільки сучасні програмні системи потребують гарантій коректності їх функціонування, то формальні логічні моделі стають необхідним інструментом для забезпечення надійності програмного забезпечення.

Концептуальну основу для моделювання атомарних об'єктів, яку розробили для використання формальної перевірки за допомогою The Coq Proof Assistant.

Атомарні сутності є найменшими неподільними одиницями значення в логічній моделі. Вони є будівельними блоками складніших концепцій і відносин. Наприклад, в обчисленнях атомарні сутності є константами та змінними. Сучасні програмні системи складні і часто охоплюють кілька одночасних процесів. Через це може бути важко гарантувати, що система завжди працюватиме правильно, навіть за наявності неочікуваних подій або помилок. Формальні логічні моделі можуть допомогти у вирішенні цієї проблеми, надаючи спосіб чіткого та систематичного міркування про поведінку системи.

Такий підхід використовують, наприклад, для формальної верифікації мови специфікації годинникових обмежень (CCSL), яка призначена для опису обмежень поведінки вбудованих і кіберфізичних систем. Атомарними сутностями цієї мови є джерела гомоменних подій у системі, які можна розглядати як подійні типи і які називають годинниками. Запропонована модель виникає як спроба побудови формального базису семантичного фреймворка для формальної верифікації специфікацій годинникових обмежень.

Ключові слова: формальна перевірка, залежні типи, помічник доказів Coq, атомарні сутності, формальний доказ.