A Coq Implementation of the Program Algebra in Jifeng He's New Roadmap for Linking Theories of Programming

Rundong Mu and Qin Li(✉)

Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China
qli@sei.ecnu.edu.cn

Abstract. Jifeng He has proposed a roadmap for linking theories of programming and presents an algebra of programs capable of generating both denotational and operational representations from the refinement relation. In this paper, we implement this algebra of programs and its refinement relation using the interactive theorem prover Coq. Encoding the algebra into CIC (Calculus of Inductive Constructions), the main formalism in Coq, facilitates machine-aided interactive proving for the properties of programs using predefined algebraic laws. The implementation of the algebra for finite programs enables us to prove that every finite program can be reduced to the normal form and to check the refinement between two finite programs. The implementation of the algebra for infinite programs supports formalizing recursive programs with one variable and checking the refinement between one finite and one infinite program. Then, we present examples of proving the refinement relationship between two finite programs and a finite program and an infinite program.

Keywords: Unifying theories of programming · Coq · Program algebra · Refinement

1 Introduction

Formal semantics for programs are usually constructed using one of two approaches, as described in [7]. The first approach is a top-down approach that starts with a denotational model and links the algebraic properties with it by establishing the soundness and completeness between them. This approach is used in works such as [2,4,14]. The second approach is a bottom-up approach that begins with an operational representation and defines a rich variety of bisimulations to identify the equivalences among programs. This approach is used in works such as [1,15]. Algebraic laws are then generated from the study of the equivalence relations like [16].

In Jifeng He's paper [7], he explores a new roadmap for linking theories of programming other than the top-down and bottom-up approaches. It begins

J. P. Bowen et al. (Eds.): *Theories of Programming and Formal Methods*, LNCS 14080, pp. 395–412, 2023.
https://doi.org/10.1007/978-3-031-40436-8_15

from an algebra of programs and generates both denotational and operational representations from the algebraic refinement relation. For the initial step of this approach, a program algebra $(\mathcal{P}, \sqsubseteq_A)$ consisting of a set of laws is presented to express the algebraic properties of programs. This program algebra is the basis of this approach, and the main criterion of the algebra is whether it is sufficient to convert every program in the domain $\mathcal{P}$ to a normal form. This criterion is stated in Theorem 2.1 in [7]. The refinement order $\sqsubseteq_A$ is defined on normal forms to support comparing the behaviors of two programs.

This paper aims to implement the program algebra in [7] with the interactive theorem prover Coq and prove the corresponding theorems relating the algebra with the aid of it. We chose Coq because of its strong type system, which can support the binding and value model required by He's model. Although efforts have been made to develop more suitable value models, such as the one proposed in [5], we opted for a deep embedding approach to gain greater control over the proof process during refinement steps. Specifically, we restrict ourselves to a monomorphic value type with a fixed representation, similar to what was done in [11]. Different from shallow embedding like [3], deep embedding allows for more accurate operations on values and enable more effective use of corresponding libraries of certain types. However, it can be less convenient for proving. By encoding in an algebraic way, we can separate the value and abstract algebra parts. For the latter, we can still leverage proof facilities to simplify the proving process. For the former, we can make assertions on values rather than providing concrete values. Additionally, deep embedding allows us to manipulate the data at a more granular level using Coq's library.

- We translated the finite operators over algebra into CIC (Calculus of Inductive Constructions) that can be accepted in Coq.
- We encoded the algebraic laws as rules that can be used for deduction between finite algebras. In doing so, we proved Theorem 2.1 in [7], which states that all finite algebras can be transformed into some normal form by giving the concrete transformation program. We proved that such transformation converts all programs to some normal form, and such transformation is only a composition of the laws outlined in He's paper.
- We provided a method to check the refinement relationship between two finite programs in mechanical proof.
- Furthermore, our work extends He's paper by providing a solution for comparing certain infinite programs and finite programs within finite steps.

The paper is organized as follows:

Section 2 briefly introduces the algebra of programs and refinement relation intended to be implemented.

Section 3 encodes the operators of the algebra into CIC and implements the algebraic laws for finite programs. The theorem that every finite program can be reduced to normal form is proved based on the implementation. Additionally, we implemented the refinement relation between finite programs and present an example of checking refinement between two finite programs.

Section 4 present our in-progress work on the infinite cases of the algebra together with an example of checking the refinement between an infinite program and a finite program.

Section 5 discusses the limitations and alternative solutions.

Section 6 concludes this article and talks about future works.

2 Preliminary

In Jifeng He's paper [7], he presents a program algebra $(\mathcal{P}, \sqsubseteq_A)$ for the following syntax of sequential programs. This algebra is built upon the foundation established by [8].

$$P, Q ::= \bot \mid var := exp \mid P \lhd bexp \rhd Q \mid P; Q \mid P \sqcap Q \mid X \mid \mu\ X \bullet P(X) \quad (1)$$

where

- $\bot$ stands for a chaotic program that does not terminate and can yield any behaviour unpredictably.
- $var := exp$ stands for the assignment statement assigning the values of a list of expressions to a list of variables.
- $P \lhd bexp \rhd Q$ stands for the conditional choice where $bexp$ is the boolean condition. It executes P when $bexp$ holds and executes Q otherwise.
- $P; Q$ stands for sequential composition.
- $P \sqcap Q$ stands for non-deterministic choice. In the following, we extend this operator to compose more than two operands. For example, $\sqcap\{P_1, P_2, ..., P_n\}$ means $P_1 \sqcap P_2 \sqcap ... \sqcap P_n$.
- X stands for a syntactic program that can be only used in the scope of a recursive program that binds it.
- $\mu X \bullet P(X)$ stands for the recursive program with X as the bounded recursive identifier.

Jifeng He developed an approach to construct algebraic equivalence classes between algebras based on some predefined algebraic laws, as detailed in Appendix A of [7]. These laws employ the $=_A$ notation to represent algebraic equivalence, which is distinct from the syntactical equality $(=)$ used in Coq. Algebraic equivalence $(=_A)$ satisfies transitivity, reflexivity, and commutativity. Additionally, it adheres to a composition law that allows any subterm within an algebra to be replaced with its corresponding subterm within the same algebraic equivalence class. We ensure that the resulting term remains within the same algebraic equivalence class as the original term.

With the algebraic equivalence defined above, we can define the normal form of programs. The normal form is defined in two ways: the finite normal form (FNF) and the infinite normal form (INF).

Definition 1 (Finite Normal Form). *Let $bexp$ be a boolean condition, $v := e_i$ be a total assignment. The finite normal form is defined as follows.*

$$\bot \lhd bexp \rhd \sqcap_i (v := e_i) \quad (2)$$

The algebraic refinement relation on the finite normal forms is defined with the following two rules.

Definition 2 (Algebraic Refinement for Finite Program). *Let P and Q be programs. The refinement order $\sqsubseteq_A$ for finite programs is defined as follows.*

1. *If $P =_A \sqcap\{v := e_i \mid i \in I\}$, $Q =_A \sqcap\{v := f_j \mid j \in J\}$, then*
   $P \sqsubseteq_A Q$ iff for any $f \in \{f_j \mid j \in J\}$, we have $\forall x \bullet f(x) \in \{e_i(x) \mid i \in I\}$.
2. *If $P =_A \bot \lhd b \rhd R$, $Q =_A \bot \lhd c \rhd S$, then*
   $P \sqsubseteq_A Q$ iff $[c \Rightarrow b]$ and $[b] \vee (R \sqsubseteq_A S)$
   The notation $[bexp]$ means $\forall v \bullet bexp(v)$ where v is the list of all free variables in $bexp$.

The infinite normal form is defined as the infinite sequence of finite normal forms.

Definition 3 (Infinite Normal Form). *Let S_i be programs of finite normal form $\bot \lhd b_i \rhd Q_i$, S_i, S_{i+1} form an ascending chain $S_i \sqsubseteq_A S_{i+1}$, and for any i, j with $b_i = b_j$, $\bot \lhd b_i \rhd Q_i =_A \bot \lhd b_j \rhd Q_j$. The infinite normal form is defined as follows.*

$$\sqcup\{S_i \mid i \in N\} \tag{3}$$

where $\sqcup S$ stands for the least upper bound of the set S.

The algebraic refinement relation on the infinite normal forms is defined as following two rules.

Definition 4 (Algebraic Refinement for Infinite Program). *Let P and Q be programs. The refinement order $\sqsubseteq_A$ is defined as follows.*

1. *If $S =_A \bot \lhd b \rhd R$, $T =_A \{\bot \lhd c_i \rhd U_i \mid i \in N\}$, then*
   $S \sqsubseteq_A (\sqcup T)$ iff $[(\bigwedge c_i) \Rightarrow b]$ and $\forall i \in \mathbb{N} \bullet R \sqsubseteq_A U_i$.
2. *If $P =_A \sqcup\{S_i \mid i \in N\}$, $Q =_A \sqcup\{T_i \mid i \in N\}$, then*
   $P \sqsubseteq_A Q$ iff $\forall i \in \mathbb{N} \bullet S_i \sqsubseteq_A Q$.

As mentioned above, the infinite program relies on finite programs and their refinement relation. Therefore, it is advisable to prioritize encoding the finite programs first.

3 Encoding Algebra for Finite Programs

Jifeng He uses algebraic laws to represent the deduction of program algebra and refinement relations. In this section, we will discuss how to encode the laws related to finite algebra into Coq code. Coq employs the formalism called Calculus of Inductive Constructions [12]. This formalism replaces property verification with the check of type signatures of expression captured by Coq's special sort, `Prop`. Moreover, inductive definitions will automatically generate induction hypotheses, which assist us in our proofs.

3.1 Translating Syntax of Finite Algebra

In this section, an inductive type `Alg` is constructed to represent all finite programs that do not include recursion. The `Alg` type is composed of two types, `Atomic` and `Comp`, which correspond to different kinds of valid program syntax.

```
Section alg.
    Inductive Atomic : Type :=
    | Chaos : Atomic
    | Assn : Assign → Atomic
    | Empty : Atomic.
    Inductive Comp : Type :=
    | Seq : Comp
    | NDC : Comp
    | CDC : Boolexp → Comp.
    Inductive Alg : Type :=
    | Lift : Atomic → Alg
    | Comb : Comp → Alg → Alg → Alg.
    Definition NDCList (l : list Alg) : Alg :=
        match l with
        | [] ⇒ Lift Empty
        | h :: tl ⇒ fold_left (fun a b ⇒ (Comb NDC a b)) tl h
        end.
End alg.
```

The finite operators in program algebra correspond to the `Comp` type. These operators include sequential composition (`Seq`), non-deterministic choice (`NDC`), and conditional choice (`CDC`). On the other hand, the chaotic program (`Chaos`) and assignment statement (`Assn`) correspond to `Atomic` type.

The symbol $\sqcap l_{n \in N}$ represents the non-deterministic choice of a set l, where N is the set of natural numbers. In our implementation, we represent sets using lists and encode the non-deterministic choice of the list as `NDCList` This encoding is straightforward, except for the empty set, which we represent using a special algebraic structure called `Empty`.

For the sake of readability, we will use the following notations to denote the syntax, the notations are similar to the original symbol in Jifeng He's paper.

The algebra of a single assignment statement can be represented using the following notation:

```
Notation "·{ e }" := (Lift (Assn e)) (at level 10).
```

The assignment statement e can be divided into two parts: the variable part and the expression part. The variable part is a user-defined type, which is injected through a type class. The expression part is simply a function, where the domain and range are both a list of variables.

```
Definition Exp := (list Var) → (list Var).
Record Assign := makeAssign {
    ids: (list Var); values : Exp;
}.
```

It can be represented using the following notation:

```
Notation "var :== exp" := (makeAssign var exp)(at level 51).
```

The chaotic program is represented using a notation similar to $\perp$.

```
Notation "_|_" := (Lift Chaos)(at level 10).
```

The conditional choice of program p and q is represented using the notation below.

```
Notation "p <| b |> q" := (Comb (CDC b) p q) (at level 15).
```

The boolean condition b in the branching expression is defined as a function that takes a list of variables as its input and outputs a boolean value.

```
Definition Boolexp : Type := (list Var) → bool.
```

The notation below is used to represent the sequential composition of program p and q.

```
Notation "p ;; q" := (Comb Seq p q)(at level 14, left associativity).
```

To represent the non-deterministic choice of program p and q, we use the notation below.

```
Notation "p /-\ q" := (Comb NDC p q)(at level 13, left associativity).
```

The non-deterministic choice of a set can use the following notation.

```
Notation "|-| l" := (NDCList l)(at level 10).
```

Specifically, the representation of non-deterministic choice for the empty set is denoted using the following notation.

```
Notation "-o-" := (Lift Empty)(at level 10).
```

3.2 Representing Algebraic Equivalence Relationship

The algebraic equivalence relation ($=_A$) is a property defined on two algebras. It is denoted as `rwtrel` in the following Coq code.

```
Section rwtrel.
    Parameter rwtrel : Alg → Alg → Prop.
    Axiom rwt_refl : forall (a: Alg), rwtrel a a.
    Axiom rwt_trans : forall (a b c : Alg), rwtrel a b →
        rwtrel b c → rwtrel a c.
    Axiom rwt_comm : forall (a b : Alg), rwtrel a b → rwtrel b a.
    Axiom rwt_comb : forall (a b c d : Alg) (e : Comp), rwtrel a b →
        rwtrel c d → rwtrel (Comb e a c) (Comb e b d).
End rwtrel.
Notation "a ←→ b" := (rwtrel a b) (at level 20, left associativity).
```

The relation satisfies the properties of reflexivity, transitivity, commutativity, and the composition law. These properties correspond to the following axioms: `rwt_refl`, `rwt_trans`, `rwt_comm`, and `rwt_comb`.

The notation ($\leftarrow\!\!\rightarrow$) is used to represent algebraic equivalence ($=_A$). In the code that follows, all algebraic laws for the program algebras described in [7] are expressed using `rwtrel`.

3.3 Encoding the Algebraic Laws

All algebraic laws can be categorized into three layers. The first layer concerns operations on assignments, while the second layer involves the combination of non-deterministic choices over different assignments. The third layer deals with operations on the finite normal form (without recursion). With the help of these predefined algebraic laws, we intend to establish a theorem saying that all finite programs can be reduced to their normal forms.

Due to space limitations, we refer the readers to the Appendix A of Jifeng He's paper [7] to see all the corresponding algebraic laws.

Assignment. Regarding the first layer, most laws simply require a straightforward translation into code. For example, Law A.2.(2) can be translated into code as follows:

```
Axiom Assign_Seq : forall (v : list Var) (g h : Exp),
    ·{v :== g} ;; ·{v :== h} ←→ ·{v :== (fun x ⇒ h (g x))}.
```

Law A.2.(1) states that any assignment can be extended into its corresponding total assignment.

```
Axiom Assign_extends : forall (v : Assign), ·{v} ←→ ·{extends_assign v}.
```

we interpret extending an assignment to its corresponding total assignment as an extension of the variable part to include all possible variables in the `GLOBVARS`. We then proceed to extend the expression function accordingly.

```
Definition extends_assign (v : Assign) :=
    makeAssign GLOBVARS (extends_mapping v.(ids) v.(values)).
```

The function `extends_mapping` maps the variable in the domain of the original assignment to its original range while leaving all other variables unchanged. This can be achieved with the help of `extends_mapping_help` function.

```
Definition extends_mapping (us : list Var) (m : (list Var) → (list Var)) :=
    fun k ⇒ (extends_mapping_help us (m us) k).
```

The function `extends_mapping_help` allows for the extension of a target expression mapping's domain. Specifically, it maps elements in the range of us to their corresponding values in $m(us)$. Any element that is not in the range of us but is within the range of k remains unchanged. The function utilizes the `lookup_help` function to determine whether a target variable exists within an assignment's domain.

```
Fixpoint extends_mapping_help (us rs k : (list Var)) : (list Var) :=
  match k with
  | []  ⇒ []
  | v:: vl ⇒
      lookup_help v us rs :: extends_mapping_help us rs vl
  end.
```

If the variable a is within the domain vs, `lookup_help` will return its corresponding value within the range us. Otherwise, the variable a remains unchanged.

```
Fixpoint lookup_help (a: Var) (vs rs: (list Var)) : Var :=
  match vs, rs with
  | _, []  ⇒ a
  | [], _ ⇒ a
  | v:: vl, r:: rl ⇒
      if (eqb a v) then r else lookup_help a vl rl
  end.
```

Non-deterministic Choice. Law A.3 in [7] states the absorption properties of non-deterministic choice of total assignments. It relies on syntax checking whether a program is in the form of non-deterministic choices over total assignments, which we denote as CH. We define the following function CH to achieve that.

```
Definition CH (p : list Alg) : Prop :=
  forall (x : Alg), In x p → exists y, x = ·{y} ∧ Total_Assign y.
```

where the function Total_Assign checks whether a target assigning is total.

```
Definition Total_Assign (a : Assign) :=
  forall v:Var, In v GLOBVARS → In v a.(ids).
```

Therefore, the law of the conditional operation over CHs (Law A.3.(2)) is defined as follows.

```
Axiom Cond_over_Choice : forall (a b : list Alg) (bexp : Boolexp),
  CH a → CH b → (|-| a) <| bexp |> (|-| b) ← →
    |-| (map (fun g ⇒ (fst g) <| bexp |> (snd g)) (list_prod a b)).
```

Finite Normal Form. The laws on the absorption properties of finite normal forms (Law A.4) can be similarly defined. For instance, the law of the non-deterministic operation over finite normal forms (Law A.4.(1)) is defined as follows.

```
Axiom NF_over_Choice : forall (a b : list Alg) (c d : Boolexp),
  CH a → CH b → (((_|_) <| c |> (|-| a)) /-\ ((_|_) <| d |> (|-| b))) ← →
    ((_|_) <| (fun g ⇒ orb (c g) (d g)) |> ((|-|a) /-\ (|-| b))).
```

The proof of the following theorem relies on the laws defined above.

3.4 Proof of Finite Normal Form Reduction

In this part, we would use the implications given above to prove the key Theorem 2.1 in [7], which states that every finite program can be reduced to FNF. The corresponding theorem is presented in Coq as follows.

```
Theorem FNF_closure : forall (P : Alg),
    exists Q, P ←→ Q ∧ FNF Q.
```

where the function FNF (Definition 1) is defined to check whether a program is in the finite normal form.

```
Definition FNF (P : Alg): Prop :=
    exists bexp R, P = (_|_) <| bexp |> (|−| R) ∧ CH R.
```

We proved this theorem through the implementation of a program that converts any input program to its normal form, referred to as Normal. In order to prove the above, we imposed two crucial rules.

The first law states that the resulting program must conform to the normal form condition, which can be expressed as follows:

```
Theorem NormalisNF : forall x, FNF (Normal x).
```

Listing 1.1. Normal is in normal form

The second law, which states that all finite programs subjected to the transformation should still yield algebraic equivalent outcomes, can be formalized as the following theorem:

```
Theorem NormalRWT : forall x, x ←→ Normal x.
```

Listing 1.2. Normal is algebraic equivalent

The transformation function Normal that satisfies the above conditions is implemented as follows:

```
Fixpoint Normal (a : Alg) : Alg :=
    match a with
    | Lift e ⇒
        match e with
        | Assn a ⇒ (_|_) <| false_stat |> |−|[·{extends_assign a}]
        | Empty ⇒ (_|_) <| false_stat |> |−|[]
        | Chaos ⇒ (_|_) <| true_stat |> |−|[·{empty_assn}]
        end
    | Comb s p q ⇒
        match s with
        | Seq ⇒ Normal_comb_Seq (Normal p) (Normal q)
        | CDC b ⇒ Normal_comb_CDC (Normal p) (Normal q) b
        | NDC ⇒ Normal_comb_NDC (Normal p) (Normal q)
        end
    end.
```

When a program belongs to Atomic, it can be translated into its corresponding normal form directly. However, if it contains any operators belonging to the

Comp, it must then be divided into two sub-programs for translation. The sub-programs are subsequently translated individually and then combined to form a new program that is also in its normal form.

In the above definition, the function Normal_comb_Seq transforms two sub-programs combined in normal form with 'Seq' into a new program in its normal form. Firstly, it combines the subprograms as Law A.4.(5) dictates. However, the right part of the resulting program is not assignment sequences, so we need to transform it accordingly.

```
Definition Normal_comb_Seq (p q : Alg) :=
  match p, q with
  | Comb x _ a, Comb y _ b ⇒
    match x, y with
    | CDC c, CDC d ⇒ (_|_) <| (fun g ⇒ orb (c g)
        (CH_over_Boolexp (Alg_to_CH a) d)) |>
    |−| (CH_comb_Seq (Alg_to_CH a) (Alg_to_CH b))
    | _, _ ⇒ −o−
    end
  | _, _ ⇒ −o−
  end.
```

The function Alg_to_CH converts the algebra that consists of assignments linked by non-deterministic choices into a list format.

```
Fixpoint Alg_to_CH (a : Alg) : list Alg :=
  match a with
  | Lift e ⇒ match e with
             | Assn a ⇒ [·{a}]
             | _ ⇒ []
             end
  | Comb s p q ⇒ match s with
                 | NDC ⇒ (Alg_to_CH p) ++ (Alg_to_CH q) % list
                 | _ ⇒ []
                 end
  end.
```

The function Alg_to_CH must meet the following condition to ensure its correctness.

```
Lemma Alg_to_CH_id : forall l, CH l → Alg_to_CH (|−| l) = l.
```

The function CH_comb_Seq combines two lists of assignments together in the manner described by Law A.3.(3).

```
Definition CH_comb_Seq (a b : list Alg) :=
  (map (fun g ⇒ Assign_comb_Seq (fst g) (snd g)) (list_prod a b)).
```

The function Assign_comb_Seq applies Law A.2.(2) to transform a program consisting of two assignments combined with Seq into a single assignment statement.

```
Definition Assign_comb_Seq_help (a b : Assign) :=
  a.( ids) :== fun x ⇒ b.(values) (a.(values) x).
```

```
Definition Assign_comb_Seq (a b : Alg) :=
    match a, b with
    | Lift x, Lift y ⇒
        match x, y with
        | Assn s, Assn t ⇒ ·{(Assign_comb_Seq_help
            (extends_assign s) (extends_assign t))}
        | _, _ ⇒ −o−
        end
    | _, _ ⇒ −o−
    end.
```

The function `Normal_comb_CDC` and the function `Normal_comb_UDC` is defined similarly. Upon completion of the definition of the function `Normal`, we need to ensure that it satisfies the conditions outlined in Listing 1.1 and Listing 1.2.

Listing 1.1 states that the program after transformation should be in normal form. The process of proving can be divided into two types of sub-goals. The first type involves only operators in the Atomic group, which we can prove directly.

The proof of Listing 1.2 requires the use of induction hypotheses. The process of proving is similar to that of the previous theorem. For subgoals involving only operators in the Atomic group, we prove them directly by applying laws. For subgoals involving induction hypotheses, we first ensure that the condition part is correctly constructed before moving on to the assignment part. Since list operations are involved, we cannot apply the reducing law directly. Instead, we must define a new lemma that connects the reducing equivalence relation between individual elements with the reducing equivalence relation across the entire list.

```
Lemma rwt_ext_Forall : forall A (f g : A → Alg) (l : list A),
  Forall (fun x ⇒ f x ←→ g x) l → |−|(map f l) ←→ |−|(map g l).
```

The complete proof can be found at the following link on GitHub[1]. In addition, the techniques for proving the above theorem can help us to convert any program to its normal form.

3.5 Definition of Refinement on Finite Programs

The refinement relation defined in Definition 2 can be implemented in Coq for finite programs by comparing two assignment expressions for equality, and checking if one non-deterministic choice of total assignments is a subset of another.

```
Definition Refine (P Q : Alg) :=
    exists bexp cexp U V,
        (P = (_|_) <| bexp |> (|−| U) ∧ CH U)
        ∧ (Q = (_|_) <| cexp |> (|−| V) ∧ CH V)
        ∧ (Constraints → ((cexp GLOBVARS = false ∧ (RefineCH U V))
        ∨ (bexp GLOBVARS = true))).
```

[1] https://github.com/DonnotPanic/Program-Algebra-in-Jifeng/blob/main/ProgramAlgebra.v.

The function `Refine` corresponds to the second case of Definition 2 where two programs are in `FNF`. With the introduction of `Constraints`, we can specify certain limitations on the variables in GLOBVARS, which determines the possible range of variables.

```
Definition RefineCH (A : list Alg) (B : list Alg) :=
    forall x, In x B → exists y , In y A ∧ subAssn x y.
```

The function `RefineCH` encodes the first case of Definition 2 where two programs are both non-deterministic choices of total assignments.

```
Definition subAssn (x : Alg) (y : Alg) :=
    match x, y with
    | Lift e, Lift f ⇒
        match e, f with
        | Assn m, Assn n ⇒ subEval (extends_assign m) (extends_assign n)
        | _,_ ⇒ False
        end
    | _, _ ⇒ False
    end.
```

The function `subAssn` is defined to find whether two assignments form a subset relation, i.e., $x \subseteq y$.

```
Definition subEval (x y : Assign) :=
    forall a, In a (x.( values) x.(ids)) → In a (y.(values) y.(ids)).
```

The `subEval` function is designed to determine whether a given set of variables x, represented as a list, is a subset of another set y.

3.6 Example of Refinement on Finite Programs

In this part, we will use Coq to prove the refinement on two finite programs T_1 and T_2 presented as follows.

$$\begin{aligned} T_1 =_{def} & (\{a,b,c := a+1, b+1, c+1\}; \\ & (\{a,b,c := a,b,c\} \triangleleft\ (a \geq 20)\ \triangleright \{a,b,c := a-1, b-1, c-1\})) \\ & \triangleleft\ (a \leq 10)\ \triangleright \bot \end{aligned} \tag{4}$$

$$\begin{aligned} T_2 =_{def}\ & \bot \triangleleft (a > 10) \triangleright \{a,b,c := a+1, b+1, c+1\}; \\ & ((\{a,b,c := a+1, b+1, c+1\} \sqcap \{a,b,c := a-1, b-1, c-1\})) \end{aligned} \tag{5}$$

Since the refinement relation is defined on normal forms. The proof is conducted by first reducing the two programs T_1, T_2 to their corresponding normal forms N_1, N_2 and then show that $N_2 \sqsubseteq_A N_1$.

In order to encode the two programs with our Coq implementation, we need to first instantiate the parameters of our formalism.

```
Instance myParams : UserParams :=
    Build_UserParams MyVar GLOBVARS eqbVar Constraints.
```

The instantiation involves providing the concrete type of each variable, and a function that decides whether two variables are equal.

We set the type of variables to be a tuple consisting of a string and a natural number with `MyVar`.

```
Record MyVar := mkVar {
    id: string;
    val : nat;
}.
```

The function `eqbVar` determines whether two variables have the same name and value of natural numbers.

`GLOBVARS` is a user-defined parameter that keeps track of all variables used in the relevant programs. It functions as a dictionary that enables us to convert arbitrary assignments into total assignments. Initializing `GLOBVARS` with concrete values is not strictly necessary. Instead, we use `Constraints` to specify the properties that `GLOBVARS` must satisfy. Typically, this means including all possible variables that could appear. In our case, we instantiate `GLOBVARS` as $\{a, b, c\}$ where a, b, and c are different variables.

The programs T_1 and T_2 are encoded as Coq instances `testAlg` and `testAlg2` respectively.

```
Definition testAlg := ((·{ascassn}) ;;
    ((· {empty_assn}) <| hdge2 |> (·{dscassn}))) <| hdle1 |> (_|_).
Definition testAlg2 := (_|_) <| (fun x ⇒ negb (hdle1 x)) |>
    (· {ascassn}) ;; ((|−|[ · {ascassn};·{dscassn}])).
```

In the code above, `hdge2` represents the condition $a \geq 20$, while `hdle1` represents $a \leq 10$. The program `empty_assn` is the assignment statement that keeps all variables' values unchanged. On the other hand, `ascassn` is the assignment statement that increases all variables' values by 1, while `dscassn` decreases all variables' values by 1.

After completing the pre-work, the only work that remains to be done is to prove the following property.

```
Example testrefine : Refine (Normal testnf2) (Normal testnf).
```

The proof consists of three steps. Firstly, we pattern match `Normal testnf` and `Normal testnf2`. Let us denote the boolean expression of `Normal testnf` as b_1, and its assignment list as l_1. Similarly, let the boolean expression and assignment list of `Normal testnf2` be denoted by b_2 and l_2, respectively.

In the second step, we categorize the possible values of variables. We ensure that there is no condition where b_2 is false and b_1 is true; in other words, either b_2 is true or b_1 is false.

For the third step, we simplify l_1 and l_2 based on the condition that b_1 is false. `lia`, a tactic for linear integer arithmetic, is used to simplify conditional functions in expressions. Then, by substituting the variables in expressions, we check if all possible values in l_1 exist in l_2. This process involves rewriting by substituting the variables in the hypothesis into the goals.

The full process of this proof can be found in GitHub[2].

4 Encoding Algebra for Infinite Programs

In this section, we will delve into the intricacies of handling infinite programs and draw comparisons with their finite counterparts. Specifically, our focus will be on analyzing the infinite program generated by recursive functions with a single variable. This particular structure allows for comparisons between finite and infinite programs, without the added complexity of navigating through the expanding order of the recursive function.

4.1 Representing Infinite Programs

Throughout our discussion, we will focus specifically on recursive functions that take only a single variable. To generate infinite series for analysis, we will use a function that maps from one finite algebra to another. Specifically, we will be working with a datatype called `Stream`, which is an infinitely long list composed of two parts: the current element which is its head, and the rest of the infinite list.

Using the `CoFixpoint`, we can define an infinite list called `Recur` in such a way that every element in the list is the result of applying the function f to the previous element and the first element of the list is a.

```
Variable f : Alg → Alg.
Definition AlgStr := Stream Alg.
CoFixpoint Recur (a : Alg) : AlgStr := Cons a (Recur (f a)).
```

We can define the normal form for a given algebra stream $\{S_i\}$ by verifying that $\forall i \in \mathbb{N} \bullet S_i \sqsubseteq S_{i+1}$ as defined in Definition 3, A stream satisfying this property is said to be in its normal form using the following Coq code, where h and m are the first two elements of the stream s. The use of `Forall` ensures that the property holds for all suffixes of the given stream.

```
Definition FNFPres(P Q : Alg) :=
    exists R S, (P ←→ R ∧ FNF R) ∧ (Q ←→ S ∧ FNF S) ∧ Refine R S.
Definition AlgPresStep (s : AlgStr) :=
    let h := Streams.hd s in
    let m := Streams.hd (Streams.tl s) in
    FNFPres h m.
Definition AlgPres := Streams.ForAll AlgPresStep.
```

Coq's automatic tactic for infinite structures has a limitation in generating proper induction laws automatically. Therefore, we need to define the induction law ourselves.

[2] https://github.com/DonnotPanic/Program-Algebra-in-Jifeng/blob/main/testAlt.v.

```
Lemma AlgPresInd : forall y, FNFPres y (f y) →
    (forall x, FNFPres (f x) (f (f x))) → AlgPres (Recur y).
Proof.
    intros. unfold AlgPres. intros. apply HereAndFurther.
    unfold AlgPresStep. auto. simpl. generalize y. cofix Pres.
    intros. apply HereAndFurther.
    − unfold AlgPresStep. simpl. apply H0.
    − simpl. apply Pres.
Qed.
```

4.2 Refinement Between Finite and Infinite Program

According to the first case of Definition 4, to find out whether an infinite program refines a finite program is to find if there exists some item in the infinite program normal form series that is strong enough to refine the given finite algebra. To find such an item, we can either use the `Str_nth` function defined in Stream library to trace the nth item, or we can define a `SthExists` function to find whether the given algebra exists (or in the same deducing-closed class with the item) in the series.

```
Definition SthStep (a : Alg) (s : AlgStr) :=
    let h := Streams.hd s in a ←− → h.
Definition SthExists (a : Alg) := Streams.Exists (SthStep a).
```

With the definition given above, we can prove $F \sqsubseteq G$.

$$F =_{def} (\{a, b := a, b\} \sqcap \{a, b := b \bmod a, a\} \triangleleft\ (a = 0)\ \triangleright \bot \tag{6}$$

$$G =_{def} \lambda X \bullet (\{a, b := a, b\} \triangleleft\ (a = 0)\ \triangleright (\{a, b := b \bmod a, a\}; X)) \tag{7}$$

F is a finite program that can be encoded as follows:

```
Definition falg := |−| [skip;GCDAssn] <| hdeqz |>(_|_).
```

G is a program that uses the Euclidean algorithm to solve for the greatest common divisor. The Coq encoding of G (`GCDStr`) is shown below:

```
Definition GCDStep (a : Alg) : Alg :=
    skip <| hdeqz |> (GCDAssn ;; a).
Definition GCDStr := Recur GCDStep (_|_).
```

In the above definition, `hdeqz` is used to determine whether a is equal to zero. The program `skip` denotes an assignment that does not change anything. Finally, `GCDAssn` updates the value of a and b according to the Euclidean algorithm: $a, b := b \bmod a, a$.

```
Definition empty_assn := makeAssign GLOBVARS refl_exp.
Definition skip := ·{ empty_assn }.
Definition GCDAssn := ·{ makeAssign GLOBVARS GCDFunc}.
```

In this case we will initialize GLOBVARS as {a;b}. First of all, we would like to know if such a sequence is in its normal form.

```
Lemma GcdStrPres : AlgPres GCDStr.
```

After that, we want to find some item in the series that can be deduced to GCDRes.

```
Definition GCDRes := (_|_) <| (fun x ⇒ negb (orb (hdeqz x)
        (Assign_over_Boolexp hdeqz GCDAssn))) |>
    · { GLOBVARS :== exp_Cond refl_exp GCDFunc hdeqz}.
Lemma GcdReachRes : SthExists GCDRes GCDStr.
```

GCDRes is picked up to represent GCDStr. Our goal now is to demonstrate that GCDRes refines finite program falg.

```
Lemma refinegcd : exists r s, (r ← → falg ∧ FNF r) ∧
    (s ← → GCDRes ∧ FNF s) ∧ Refine r s.
```

Since both GCDStr and falg are finite programs, we can perform a finite comparison between them. We can use the proving techniques introduced in Sect. 3.6.

The full process of this proof can be found in GitHub[3].

5 Discussion

In this paper, we implemented Jifeng He's approach to establishing program equivalence and refinement relations using axioms, and encoded it in Coq. Our approach is based on axiomatic semantics, which distinguishes it from the [6] project that utilizes the denotational model and builds alphabetized predicates. To facilitate program comparison, we transform each program into a normal form, separating the abstract program part from the concrete evaluation part. This approach accommodates diverse computational models. In this paper, we utilize a simple model that applies abstract variables to functions directly, making comparisons between functions challenging.

We have also made progress in automating the proving process by utilizing Coq's mechanics. We have successfully automated the transformation of the abstract algebra part to its normal form. However, there remain challenges in the refinement process. The proof can be verbose, as issues may arise with unifying the type of variables in our library and the type of user-defined variables when importing our library.

This paper focuses on a special case of recursive programs with one variable, which serves as the foundation for more general recursive programs that can eventually be transformed to some recursive program with a composite variable. We are currently working on extending our work to this area.

When dealing with infinite cases, we encountered limitations due to the difficulty of representing any proposition that is a finite and terminating structure of

[3] https://github.com/DonnotPanic/Program-Algebra-in-Jifeng/blob/main/testGCD.v.

an infinitely recursive program series because it is impossible to compute infinite loops. We found two approaches to address this issue. The first involves simplifying the problem into some finite cases, where we found that comparing infinite programs to finite ones can be simplified by unrolling the infinite series a finite number of times. This results in a computable process. The second approach involves translating the problem of infinite computing to a continuity problem that we can symbolically reason about. We are still working on finding a general method for this.

6 Conclusion and Future Work

In this paper, we present our implementation of the program algebra introduced by Jifeng He in Coq. We have translated the formalism of the algebra into Coq syntax and implemented the algebraic laws and refinement relation defined by He. Using our framework, we provide machine-aided proofs for key theorems that demonstrate every finite program can be reduced to its normal form, and we give a concrete transformation program. Additionally, we provide examples to illustrate how our implementation can be used to check refinement relationships between two finite programs or a finite program and an infinite program in a theorem-proving manner.

In the future, we intend to improve our work in the following aspects.

- Determining the most appropriate way to express infinite programs still require further exploration. We will try to develop a suitable model to represent the algebra between infinite structures.
- The value model in this paper needs further improvement to meet the need of actual use.
- The refinement proof process can be verbose, but there may be techniques available to simplify it such as developing automatic tactics to extract variables hypotheses and substitute them into goals, rewrapping the expression type to simplify the comparisons between functions, changing the lazy evaluation of the expression to eager one and so on.

Furthermore, our framework can serve as a foundation for several works based on process algebra, such as probability programs [10], parallel programs [13], quantum programs [9], and more. These works can potentially be extended using our framework.

Acknowledgment. We would like to express our sincere gratitude to Simon Foster for his exceptional contribution to this paper. His valuable insights and expert guidance have greatly enhanced the quality of our work, and we are truly appreciative of his dedication and commitment to this project. Without his suggestions and feedback, the paper would not have been as comprehensive and insightful as it is now.

References

1. Ngondi, G.E., Koutavas, V., Butterfield, A.: Translation of CCS into CSP, correct up to strong bisimulation. In: Calinescu, R., Păsăreanu, C.S. (eds.) SEFM 2021.

LNCS, vol. 13085, pp. 243–261. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-92124-8_14
2. Ekembe Ngondi, G.: Denotational semantics of channel mobility in UTP-CSP. Formal Aspects Comput. **33**(4), 803–826 (2021)
3. Feliachi, A., Gaudel, M.-C., Wolff, B.: Unifying theories in Isabelle/HOL. In: Qin, S. (ed.) UTP 2010. LNCS, vol. 6445, pp. 188–206. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16690-7_9
4. Foster, S.: Hybrid relations in Isabelle/UTP. In: Ribeiro, P., Sampaio, A. (eds.) UTP 2019. LNCS, vol. 11885, pp. 130–153. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31038-7_7
5. Foster, S., Baxter, J., Cavalcanti, A., Woodcock, J., Zeyda, F.: Unifying semantic foundations for automated verification tools in Isabelle/UTP. Sci. Comput. Program. **197**, 102510 (2020)
6. Foster, S., Zeyda, F., Woodcock, J.: Isabelle/UTP: a mechanised theory engineering framework. In: Naumann, D. (ed.) UTP 2014. LNCS, vol. 8963, pp. 21–41. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-14806-9_2
7. He, J., Li, Q.: A new roadmap for linking theories of programming and its applications on GCL and CSP. Sci. Comput. Program. **162**, 3–34 (2018)
8. Hoare, C.A.R., et al.: Laws of programming. Commun. ACM **30**(8), 672–686 (1987)
9. Jorrand, P., Lalire, M.: Toward a quantum process algebra. In: Proceedings of the 1st Conference on Computing Frontiers, pp. 111–119 (2004)
10. Morgan, C., McIver, A., Seidel, K., Sanders, J.W.: Refinement-oriented probability for CSP. Formal Aspects Comput. **8**(6), 617–647 (1996). https://doi.org/10.1007/BF01213492
11. Oliveira, M., Cavalcanti, A., Woodcock, J.: Unifying theories in ProofPower-Z. In: Dunne, S., Stoddart, B. (eds.) UTP 2006. LNCS, vol. 4010, pp. 123–140. Springer, Heidelberg (2006). https://doi.org/10.1007/11768173_8
12. Paulin-Mohring, C.: Introduction to the calculus of inductive constructions (2014)
13. Woodcock, J., Hughes, A.: Unifying theories of parallel programming. In: George, C., Miao, H. (eds.) ICFEM 2002. LNCS, vol. 2495, pp. 24–37. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-36103-0_5
14. Xu, X., Zhan, B., Wang, S., Talpin, J.P., Zhan, N.: A denotational semantics of simulink with higher-order UTP. J. Logical Algebraic Methods Program. **130**, 100809 (2023)
15. Yan, G., Jiao, L., Li, Y., Wang, S., Zhan, N.: Approximate bisimulation and discretization of hybrid CSP. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 702–720. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_43
16. Zhu, H., He, J., Qin, S., Brooke, P.J.: Denotational semantics and its algebraic derivation for an event-driven system-level language. Formal Aspects Comput. **27**, 133–166 (2015)