Expressivity of BCI algebras

Samuel Arsac
M1 student in Computer Science at ENS Paris-Saclay 2020-2021
Under the supervision of Robert Atkey

February 15th - July 16th 2021
Mathematically Structured Programming Group
University of Strathclyde, Glasgow

Contents

I warmly thank Bob Atkey for taking me as an intern, for his supervision in these difficult conditions, for being so available and for his valuable support during and after my internship.

I also thank all the attendees at my MSP101 talk for their attention and their remarks and questions on my internship. I particularly thank Conor McBride for his great help in the improvement of my Coq code.

1 Introduction

My internship revolves around the concept of BCI algebra, introduced by Moses Schönfinkel in the 1920s with his work on combinatory logic, and the related concept of linear λ-calculus (see [Hin97]). BCI algebras have the particularity of having basic combinators that do not allow for deletion or duplication of the objects they are applied to. They are also linked with Girard's geometry of interactions, as outlined in [AHS02]. Reversible computation, a model of computation in which data is linear, and as such cannot be deleted nor duplicated [JS12], was also supposed to be a central part of my internship, as it can also be linked with BCI algebras [Abr05]. In relation to this model, there are type systems that allow for the tracking of the use of resources [Atk18]. Category theory can be used to model the computing power of BCI algebras, using the work from [Hos07] with definitions from [ML71], as well as reversible computation [CS21].

My original subject was to create a reversible and dependently typed programming language using BCI algebras. As it quickly appeared that BCI algebras were too powerful for this use, I instead studied their expressivity. Since BCI combinators are very weak, as stated earlier, it is interesting to see what limit they have, if that limit is not reversible computation, as we thought at the beginning of my internship.

As writing terms with BCI combinators is very tedious and error prone, I started by implementing them in Coq, and added linear λ-calculus to make the definition of new objects easier. Although my first try proved quite difficult, a second approach to this problem allowed me to make a clean encoding for linear λ-calculus. Unfortunately, I failed to prove the correctness of my main function in both cases, but I was able to prove the correctness of the outputted objects independently.

I then tried to add booleans, first as a primitive, when trying to obtain a reversible model, and then by encoding them with the BCI combinators, which turned out to be more powerful than the few axioms added in the first attempt, while also being more general. I used encodings for linear λ-calculus from [Mai04], which are variations of the usual encodings for standard λ-calculus, and then used the Coq function previously defined to turn them into objects of a BCI algebra.

As these encoded booleans have interesting properties, I studied them from a category theory point of view to obtain a lower bound on the computing power of the model.

2 Preliminary results and definitions

The definitions and results of 2.1 and 2.2 come from [Hos07].

2.1 BCI algebra

Definition 1 (BCI algebra)**.** *A BCI algebra $\mathcal{A}$ is a pair $(A, \cdot)$ with A a set and $\cdot$ a left-associative total binary application.*
The set A contains three objects B, C and I, satisfying

$$B \cdot x \cdot y \cdot z = x \cdot (y \cdot z)$$
$$C \cdot x \cdot y \cdot z = x \cdot z \cdot y$$
$$I \cdot x = x$$

Example. *We can encode a term behaving in the same way as pairs in λ-calculus: some $Pair \in A$ such that for all $x, y, z \in A$, $Pair \cdot x \cdot y \cdot z = z \cdot x \cdot y$*

$$\begin{aligned} & B \cdot C \cdot (C \cdot I) \cdot x \cdot y \cdot z \\ = \; & C \cdot (C \cdot I \cdot x) \cdot y \cdot z \\ = \; & C \cdot I \cdot x \cdot z \cdot y \\ = \; & I \cdot z \cdot x \cdot y \\ = \; & z \cdot x \cdot y \end{aligned}$$

Definition 2 (Polynomial over $\mathcal{A}$)**.** *A polynomial over $\mathcal{A}$ is an expression generated by variables, elements of A and applications.*

In order to define useful elements of A more easily, we can define terms like in linear λ-calculus, where each variable appears exactly once.

Proposition 1. *If M is a polynomial over some BCI algebra $(A, \cdot)$ containing a variable x appearing only once, then there exists a polynomial $\lambda^* x.M$ with the same free variables as M except x, satisfying for all $a \in A$, $(\lambda^* x.M) \cdot a = M[a/x]$.*

Proof. Since it contains x as a free variable, M is either x or $N \cdot N'$ for some polynomials N and N'. We can define $\lambda^* x.M$ in the following way:

$$\lambda^* x.x = I$$

$$\lambda^* x.N \cdot N' = \begin{cases} C \cdot (\lambda^* x.N) \cdot N' & \text{if } x \text{ is a free variable in } N \\ B \cdot N \cdot (\lambda^* x.N') & \text{if } x \text{ is a free variable in } N' \end{cases}$$

It is well defined since N and N' are strictly smaller than M. It satisfies $(\lambda^* x.M) \cdot a = M[a/x]$ for $a \in A$:

- it is direct if M is x,
- if $M = N \cdot N'$ and x is in N, then

$$\begin{aligned} & C \cdot (\lambda^* x.N) \cdot N' \cdot a \\ = & (\lambda^* x.N) \cdot a \cdot N' \\ = & N[a/x] \cdot N' && \text{by induction hypothesis} \\ = & M[a/x] && \text{since } x \text{ cannot appear in } N' \end{aligned}$$

- if x is in N', $B \cdot N \cdot (\lambda^* x.N') \cdot a = N \cdot ((\lambda^* x.N') \cdot a)$, which gives us the right result, similarly to the previous case.

2.2 Category of assemblies

A realisability relation between a BCI algebra and a set describes which elements of the algebra implement a given element of the set.

Definition 3 (Realisability relation)**.** *A realisability relation for a BCI algebra $\mathcal{A}$ and a set X is a subset of $A \times X$.*

This can be extended to functions.

Definition 4 (Realisable function)**.** *Given two sets X and Y and their realisability relations $\vDash_X$ and $\vDash_Y$ in respect to a BCI algebra $\mathcal{A}$, a function $f : X \to Y$ is realisable if there exists a realiser $r \in A$ such that for all $x \in X$ and $a \in A$ such that $a \vDash_X x$,*

$$r \cdot a \vDash_Y f(x)$$

From this we can define a category based on the category of sets:

Definition 5 (Category of assemblies)**.** *Given a BCI algebra $\mathcal{A}$, we can define $Asm(\mathcal{A})$, the category of assemblies of $\mathcal{A}$.*

- *Its objects are pairs $(X, \vDash_X)$ with X a set and $\vDash_X \in A \times X$ the realisability relation of X.*
- *Its morphisms from $(X, \vDash_X)$ to $(Y, \vDash_Y)$ are the realisable maps from X to Y.*

We will prove that this category is symmetric monoidal closed. Here is the definition of a symmetric monoidal category, the idea behind being that the category is equipped with an associative and commutative tensor product with a unit object.

Definition 6 (Symmetric monoidal cateogry)**.** *A symmetric monoidal category is a category $\mathcal{C}$ equipped with a tensor product, which is a functor $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ and a unit object I.*

It also requires some natural isomorphisms:

- *associator $\alpha_{x,y,z} : (x \otimes y) \otimes z \to x \otimes (y \otimes z)$*
- *left unitor $\lambda_x : I \otimes x \to x$*
- *right unitor $\rho_x : x \otimes I \to x$*
- *symmetry $\sigma_{x,y} : x \otimes y \to y \otimes x$*

With the requirement that the following diagrams commute.

$$\begin{array}{ccccc}
((w \otimes x) \otimes y) \otimes z & \xrightarrow{\alpha_{w\otimes x,y,z}} & (w \otimes x) \otimes (y \otimes z) & \xrightarrow{\alpha_{w,x,y\otimes z}} & w \otimes (x \otimes (y \otimes z)) \\
& \searrow^{\alpha_{w,x,y}\otimes 1_z} & & & \nearrow_{1_w \otimes \alpha_{x,y,z}} \\
& & (w \otimes (x \otimes y)) \otimes z & \xrightarrow{\alpha_{w,x\otimes y,z}} & w \otimes ((x \otimes y) \otimes z)
\end{array}$$

$$\begin{array}{ccc}
(x \otimes I) \otimes y & \xrightarrow{\alpha_{x,I,y}} & x \otimes (I \otimes y) \\
& \searrow^{\rho_x \otimes 1_y} \quad \swarrow_{1_x \otimes \lambda_y} & \\
& x \otimes y &
\end{array}$$

$$\begin{array}{ccccc}
(x \otimes y) \otimes z & \xrightarrow{\alpha_{x,y,z}} & x \otimes (y \otimes z) & \xrightarrow{\sigma_{x,y\otimes z}} & (y \otimes z) \otimes x \\
\downarrow{\scriptstyle \sigma_{x,y}\otimes 1_z} & & & & \downarrow{\scriptstyle \alpha_{y,z,x}} \\
(y \otimes x) \otimes z & \xrightarrow{\alpha_{y,x,z}} & y \otimes (x \otimes z) & \xrightarrow{1_y \otimes \sigma_{x,z}} & y \otimes (z \otimes x)
\end{array}$$

Finally, we must have

$$\sigma_{y,x} \circ \sigma_{x,y} = 1_{x\otimes y}$$

Now we define what it means for it to be closed: the general idea is that for any objects x and y, there is an object corresponding to the set of morphisms between x and y. In the case of symmetric monoidal categories, it can equivalently be defined as the existence of currying.

Definition 7 (Symmetric closed monoidal category)**.** *A symmetric monoidal cateogry $\mathcal{C}$ is closed if for all object x, y and z, there is an isomorphism between $Hom_{\mathcal{C}}(x \otimes y, z)$ and $Hom_{\mathcal{C}}(x, hom(y, z))$.*

And we can now express the main proposition.

Proposition 2. *$Asm(\mathcal{A})$ is symmetric monoidal closed.*

Proof. This is true for the category of sets with the cartesian product, so we only have to take realisability into account.

The tensor product becomes

$$\otimes : (X, \vDash_X), (Y, \vDash_Y) \mapsto (X \times Y, \vDash_{X \times Y})$$

with $\vDash_{X \times Y} = \{(Pair \cdot a \cdot b, (x, y)) | a \vDash_X x, b \vDash_Y y\}$, where $Pair$ is the object defined in the previous example.

The unit object is $(\{\star\}, (I, \star))$ (a canonical singleton, which only object is realised by I).

All the required isomorphisms exist for sets, and can be extended to $Asm(\mathcal{A})$ by applying it to the left member of the pair and taking the corresponding realisability relation thanks to the previous definition of $\vDash_{X \times Y}$. However, we need to prove that they are realisable.

- α: if we have $(X, \vDash_X)$, $(Y, \vDash_Y)$ and $(Z, \vDash_Z)$ three objects, x, y and z and a, b and c such that $a \vDash_X x$, $b \vDash_Y y$ and $c \vDash_Z z$, then $(Pair \cdot a \cdot (Pair \cdot b \cdot c)) \vDash_{X \times (Y \times Z)} (x, (y, z))$.

  $$\begin{aligned} & Pair \cdot a \cdot (Pair \cdot b \cdot c) \cdot (\lambda^* a\ m.\ m \cdot (\lambda^* b\ c\ x.\ x \cdot a \cdot b \cdot c)) \cdot (\lambda^* a\ b\ c\ x.\ x \cdot (\lambda^* y.y \cdot a \cdot b) \cdot c) \\ =& (\lambda^* x.\ x \cdot a \cdot b \cdot c) \cdot (\lambda^* a\ b\ c\ x.\ x \cdot (\lambda^* y.y \cdot a \cdot b) \cdot c) \\ =& \lambda^* x.x \cdot (\lambda^* y.y \cdot a \cdot b) \cdot c \\ =& Pair \cdot (Pair \cdot a \cdot b) \cdot c \end{aligned}$$

  Hence α is realised by

  $$\lambda^* e.\ e \cdot (\lambda^* a\ m.\ m \cdot (\lambda^* b\ c\ x.\ x \cdot a \cdot b \cdot c)) \cdot (\lambda^* a\ b\ c\ x.\ x \cdot (\lambda^* y.y \cdot a \cdot b) \cdot c)$$

- λ is realised by $\lambda^* x.\ x \cdot (\lambda^* a\ b.\ b \cdot a)$
- ρ is realised by $\lambda^* x.\ x \cdot (\lambda^* a\ b.\ a \cdot b)$
- σ is realised by $\lambda^* x.\ x \cdot (\lambda^* a\ b\ y.\ y \cdot b \cdot a)$

As for closure, we need to prove that for all objects a, b and c, there is a natural bijection between $\mathrm{Hom}(a \otimes b, c)$ and $\mathrm{Hom}(a, \mathrm{hom}(b, c))$.

If we have a morphism f from $(X \times Y, \vDash_{X \times Y})$ to $(Z, \vDash_Z)$ then we have a morphism g from $(X, \vDash_X)$ to $(Y \multimap Z, \vDash_{Y \multimap Z})$, with $Y \multimap Z$ the set of realisable morphisms from Y to Z, and $\vDash_{Y \multimap Z} = \{(a, s) \in \mathcal{A} \times (Y \multimap Z) | a \text{ realises } s\}$. It associates to each element x of X a function h_x from Y to Z such that for all $y \in Y$, $h_x(y) = f(x, y)$.

We need to prove that g and h_x are realisable. We name a a realiser of x, b a realiser of y and c a realiser of $f(x, y)$. If f is realised by r, we have $r \cdot (Pair \cdot a \cdot b) = c$, hence h_x is realised by $\lambda^* b.\ r \cdot (Pair \cdot a \cdot b)$ and g is realised by $\lambda^* a\ b.\ r \cdot (Pair \cdot a \cdot b)$.

The other way around, if we have a morphism g as described in the previous case, we can associate it to the morphism f such that $f(x, y) = g(x)(y)$. if g is realised by r and $g(x)$ by r', then f is realised by $r' \cdot r$.

3 Implementing BCI algebras in Coq

We can define BCI algebras as a record type, with the necessary objects and properties.

```
Record BCI :=
  {Carrier: Type;
  App: Carrier -> Carrier -> Carrier
    where "A @ B" := (App A B);
  B: Carrier;
  C: Carrier;
  I: Carrier;
  B_ax: forall x y z: Carrier, B @ x @ y @ z = x @ (y @ z);
  C_ax: forall x y z: Carrier, C @ x @ y @ z = x @ z @ y;
  I_ax: forall x: Carrier, I @ x = x;
  }.
```

Note: the names of the elements of the record are simplified. These are different in the file, as they require a `BCI` as an argument to be used, then the use of notations to obtain clear names in the proofs. For instance, `B` is named `B_combinator` and a `Parameter bci: BCI` is used to define `Notation "'B'" := (B_combinator bci)`.

3.1 Implementing λ^*

We now want to define a `lambda_s` function, computing the `Carrier` object corresponding to a λ^* term. I took the encoding of λ-calculus from [BHKM12] as a model for the beginning.

Implementing λ^* requires a type `Exp` for polynomials, with free variables. It also requires a way to go from polynomials back to a `Carrier`.

This type will be dependent on a list of free variables. The type of free variables is a variable `Ty: Type`, with the environment being defined as

```
Definition Env := list Ty.
```

The `Exp` type will have to satisfy the two following properties:

- The order of variables must not matter: `Exp E1 = Exp E2` if `E1` and `E2` are equal up to permutation. Otherwise, if we were to define a function of the form

  ```
  Definition Double_bind: _ -> Exp [] := fun e => lambda_s x (lambda_s y e)
  ```

  (where `lambda_s` is simplified, it will require more arguments), `e` would either be typed `Exp [x; y]` or `Exp [y; x]`, hence the need for these two types to contain all the terms with two free variables `x` and `y`, appearing in any order.

- If the term is an application $N \cdot N'$, there must be a way to split the list into two lists containing the free variables of N and N'.

The `Exp` type should look like this:

```
Inductive Exp: Env -> Type :=
| CAR: Carrier -> Exp []
| VAR: forall (v: Ty), Exp [v]
| APP: forall {E E_fun E_arg}, Exp E_fun -> Exp E_arg -> ... -> Exp E
```

The `VAR` and `CAR` cases are straightforward, and satisfy the first property. The difficult part is the missing type in the `APP` case. How can we link `E_fun`, `E_arg` and `E`?

Using `E = E_fun ++ E_arg` is not enough, as it would break the first property: for example, $x \cdot y$ could only be encoded by `APP (VAR x) (VAR y) eq_refl`, which would always be typed `Exp [x; y]`.

3.2 First attempt: using permutations

Using permutations allows the elements of `E` to be reordered into `E_fun ++ E_arg`.

3.2.1 Permutations

Permutations are already present in the standard library, however they are defined as a `Prop`, which means that they only encode the equivalence relation "equal up to permutation", and Coq doesn't remember the underlying permutation. I had to rewrite part of that library, changing this `Prop` into a `Type`, which allowed me to follow the elements of the lists through permutations.

Here is the encoding for permutations:

```
Inductive Permutation: list A -> list A -> Type :=
| perm_nil: Permutation [] []
| perm_skip x l l': Permutation l l' -> Permutation (x :: l) (x :: l')
| perm_swap x y l: Permutation (y :: x :: l) (x :: y :: l)
| perm_trans l l' l'': Permutation l l' -> Permutation l' l'' -> Permutation l l''.
```

This allows for the encoding of permutations as compositions of transpositions of adjacent items. `perm_nil` is the base case, `perm_skip` encodes a fixed point and `perm_swap` encodes a transposition of adjacent elements. Finally, `perm_trans` is used for composition.

We will use the notation `l1 >< l2` for the type of permutations between `l1` and `l2`.

The `APP` constructor of `Exp` is now

```
| APP: forall {E E_fun E_arg}, Exp E_fun -> Exp E_arg -> E >< E_fun ++ E_arg
  -> Exp E
```

3.2.2 The `lambda_s` function

The `lambda_s` function takes as arguments the name of the variable to be bound, an environment and an expression in that environment, followed by a proof that the variable is in the environment. It returns an expression in the environment stripped of the bound variable.

```
Definition lambda_s (v: Ty) E (e: Exp E):
    forall E1 E2, E >< E1 ++ v :: E2 -> Exp (E1 ++ E2).
  induction e.
    ...
```

The induction cases are the same as the ones in the proof of Proposition 1.
The `VAR` case of the induction is just `I_exp`, since the variable can only be the one to be bound.
The `CAR` case is absurd, since it gives an expression with an empty environment.
The `APP` case requires more work, which will be detailed in the following section.

3.2.3 The `APP` case

The problem is that we have to follow `v` to find out whether it belongs in `E_fun` or in `E_arg`. In order to do that, we first define a type holding this information:

```
Inductive APP_split: Ty -> Env -> Env -> Env -> Env -> Type :=
  | head_split: forall {x: Ty} {E1 E2 E_fun E_arg E_fun1 E_fun2: Env},
      E_fun >< E_fun1 ++ x :: E_fun2
      -> E_fun1 ++ E_fun2 ++ E_arg >< E1 ++ E2
      -> APP_split x E1 E2 E_fun E_arg
  | tail_split: forall {x: Ty} {E1 E2 E_fun E_arg E_arg1 E_arg2: Env},
      E_arg >< E_arg1 ++ x :: E_arg2
      -> E_fun ++ E_arg1 ++ E_arg2 >< E1 ++ E2
      -> APP_split x E1 E2 E_fun E_arg.
```

`head_split` means the variable is in `E_exp` and `tail_split` means the variable is in `E_fun`. In both cases, we provide a permutation of the related list showing that the variable is present, as well as a new global permutation without the variable.

We must then obtain objects of that type to use them in `lambda_s`. The following function does exactly that, by induction on the permutation it takes as an argument.

```
Definition APP_splitter: forall (x: Ty) (E1 E2 E_fun E_arg: Env),
  E_fun ++ E_arg >< E1 ++ x :: E2 -> APP_split x E1 E2 E_fun E_arg.
```

And with this element, `lambda_s` can be defined.

3.2.4 A major drawback

One issue with this encoding is that it is very redundant. Indeed, one could expect the type `[] >< []` to be only inhabited by `perm_nil`. However, with `perm_trans` there is an infinite number of inhabitants, as you can always use transitivity to add more empty permutations.

This adds a layer of complexity to the proofs, which can require difficult inductions even in cases that should be trivial.

In order to remove part of this problem, we can define an extensional equality on permutations, which is based on the movements of the elements in the list.

In order to track the elements in the list, we will use the following type:

```
Inductive Contains (a: A): list A -> Type :=
  | Here : forall l, Contains a (a :: l)
  | Next : forall l a', Contains a l -> Contains a (a' :: l).
```

`Contains` not only shows that an object is part of a list, but also keeps its position (the index in the list is the amount of `Next` before the `Here`).

Then we can change these objects according to a permutation:

```
Lemma Permutation_contains: forall {l l': list A} {a},
  l >< l' -> Contains a l -> Contains a l'.
```

Finally, our equality will be based on the similarity of these movements, in the following way:

```
Definition Permutation_eq {l1 l2: list A} (p1 p2: l1 >< l2) :=
  forall (x: A) (Ct: Contains x l1),
    Permutation_contains p1 Ct = Permutation_contains p2 Ct.
```

We can now state properties with permutations that behave like `perm_nil` (or any other permutation), which can help to obtain easier proofs.

3.2.5 Correctness

The next step is to prove the correctness of the `lambda_s` function, that is the fact that it respects $(\lambda^* x.M) \cdot N = M[N/x]$. We need a way to express the right term of the equation, which can be done by defining environments for free variables, and then substitute all the free variables with their value in the environment.

Environments In order to do that, we have the following type:

```
Definition Bci_env E := forall v: Ty, Contains v E -> Carrier.
```

`Contains` allows `Bci_env` to reference each variable by its position in the environment.

We can now turn an expression into a BCI object with the following function.

```
Fixpoint Exp_to_BCI {E} (e: Exp E): Bci_env E -> Carrier
```

The proof The statement for the correctness of `lambda_s` is the following:

```
Theorem lambda_s_correctness:
  forall E (e: Exp E) E1 E2 (v: Ty) (p: E >< E1 ++ v :: E2)
    (x: Bci_env (E1 ++ E2)) (c: Carrier),
  exists p', Permutation_eq p p' /\
  Exp_to_BCI e (Bci_env_convert _ _ p' (Bci_env_insert_mid E1 E2 x v c))
  = Exp_to_BCI (lambda_s v E e E1 E2 p') x @ c.
```

where `Bci_env_convert` and `Bci_env_insert_mid` are just used to create the environment giving the variable a value.

The `CAR` and `VAR` cases are not a problem. However, I have not succeeded in proving the `APP` case, even with the help of extensional equality and already many simplification lemmas.

The main problem is the absence of link between the input and output permutations in some functions, which makes them appear as unrelated permutations in the proof, whereas they should be closely related.

3.3 Second attempt: using partitions

This idea was given to me by Conor McBride, as an alternative to permutations. We can split directly `E` into `E_fun` and `E_arg` using an encoding of partitions in two lists.

3.3.1 Partitions

Here is the encoding for partitions:

```
Inductive Partition: list A -> list A -> list A -> Type :=
| part_nil: Partition [] [] []
| part_skipl x {l lleft lright}: Partition l lleft lright
  -> Partition (x :: l) (x :: lleft) lright
| part_skipr y {l lleft lright}: Partition l lleft lright
  -> Partition (y :: l) lleft (y :: lright).
```

`part_nil` is the base case, and the two others are used to send the first element of the list to be partitionned to one of the two sublists. This encoding has the advantage of not being redundant, unlike that for permutations.

The following notation will be used from now on:

```
Notation "ll <! l !> lr" := (Partition l ll lr)
```

We now have the following for the `APP` constructor:

```
| APP: forall E E_fun E_arg, Exp E_fun -> Exp E_arg -> E_fun <! E !> E_arg
  -> Exp E
```

3.3.2 The `lambda_s` function

This representation yields a much simplified version of `lambda_s`:

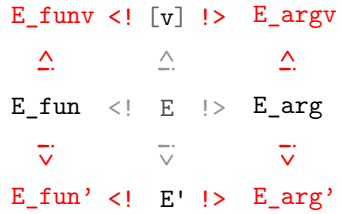
```
Definition lambda_s: forall v E E', [v] <! E !> E' -> Exp E -> Exp E'.
```

in which we directly take the variable `v` out of `E` to obtain `E'`.

The base cases are similar to the previous version, however the `APP` case is not.

3.3.3 The `APP` case

In this case, we have two partitions of `E`:
`E_fun <! E !> E_arg` and `[v] <! E !> E'`. We can combine them to obtain a partition for `E_fun`, `E_arg`, `[v]` and `E'`:



We can then look at the partition `E_funv <! [v] !> E_argv` to figure out where `v` belongs:

- if the partition is `part_skipl v part_nil`, then `v` is in `E_fun`
- else, the partition is `part_skipr v part_nil`, and `v` is in `E_arg`.

We can now define the `lambda_s` function, with much less efforts than in the previous attempt.

3.3.4 Substitution

Like in the previous case, we want a way to implement $M[N/x]$, however this time I tried a cleaner approach, with the following substitution function:

```
Lemma substitute: forall {x: Var} {EM EM' EMN EN},
  [x] <! EM !> EM' -> EM' <! EMN !> EN -> Exp EM -> Exp EN -> Exp EMN.
```

The first partition shows that x is free in M and the second is used to show that the expression outputted contains the free variables of M and N with the exception of x.

3.3.5 Correctness

Using substitution, we now have a more elegant expression for correctness:

```
Theorem lambda_s_correctness: forall v E E' E'' E''' (p1: [v] <! E !> E')
                                    (p2: E' <! E'' !> E''')
                                    (e1: Exp E) (e2: Exp E'''),
  APP (lambda_s v p1 e1) e2 p2 = substitute p1 p2 e1 e2.
```

Even though this method is cleaner than the one with permutations, I still failed to prove this theorem. This is mostly caused by some difficulties with proofs involving `substitute`, and since I did this implementation at the very end of my internship, I had not enough time to look into this problem in depth.

3.4 Comparing the two approaches

Even though my main objective when using partitions was to prove the correctness of `lambda_s`, which I was not able to do, the second method has its advantages.

The main one is that it is much shorter. The file defining permutations and the lemmas about them is about three times longer as the one for partitions. Moreover, `lambda_s` itself requires less work, which reduces even more the total size of the code on the partitions side.

This is mostly due to the fact that partitions are a much more effective way to do the case split on `APP`. Indeed, `APP_splitter`, which is very long, is replaced by the use of the - much simpler - lemma giving the square described in section 3.3.3. When writing for example that `x` is in `E`, writing `[x] <! E !> E'` is much more effective than `E >< E1 ++ x :: E2`, since concatenation adds another layer of complexity. I tried to use permutations more efficiently with `E >< x :: E'`, but I didn't manage to code `lambda_s` again using that, since it reduces the liberty given by the presence of the two sublists `E1` and `E2`.

4 Adding Booleans

We have no datatypes for now, so we can try to implement booleans to add one.

4.1 As a primitive

We can define booleans as a primitive: in addition to B, C and I, A contains two elements $\top$ and $\bot$. The idea when I worked on this was to extend the BCI algebra and to try to have a basis for a reversible programming language. Since it turned out that BCI algebras were not a model of reversible computation, this attempt could not succeed.

The booleans can be implemented in Coq as two new entries in the `BCI` record type, named `T_bool` and `F_bool`. Since these two objects cannot do anything by themselves, we need more axioms for them to be of any use.

4.1.1 If-Then-Else

In order to keep in line with the idea of reversibility, we define combinators which do not delete or duplicate variables in their definition. Since an If-Then-Else has two branches, with only one being used, applying the combinator to both its branches would carry the unused branch for the rest of the computation.

We can however assume that A (the carrier set) contains for all $x, y \in A$ an element $ITE\ x\ y$, which is the combinator corresponding to these two branches. It will then only be applied to the boolean.

Another thing we don't want to do is to discard the boolean the combinator is applied to. We can thus define the two axioms of ITE in the following way:

$$(ITE\ x\ y) \cdot \top = x \cdot \top$$

$$(ITE\ x\ y) \cdot \bot = y \cdot \bot$$

However, in this case we cannot even define a clean negation, since the boolean given as an argument will have to stay. However this can be fixed by integrating negation into the two axioms, which become:

$$(ITE\ x\ y) \cdot \top = x \cdot \bot$$

$$(ITE\ x\ y) \cdot \bot = y \cdot \top$$

With negation being defined as

$$Not = ITE\ I\ I$$

Even without the fact that this addition does not help the absence of reversibility (which will be shown in the next subsection), having ITE take two arguments and give an object of A makes λ^* unable to bind variables inside those arguments. I tried adapting λ^*, with the constraint of having the same free variables in both branches, but I failed to do so, and didn't look further when we realised that reversibility was not achievable.

4.2 By adapting the encoding for λ-calculus

Another approach to the addition of booleans is to encode them directly with the BCI combinators. Since booleans can be encoded in λ-calculus, we can try to make an encoding for linear λ-calculus.

Definition 8 (Booleans in λ-calculus)**.**

- *True is encoded as $\lambda xy.x$.*

- *False is encoded as $\lambda xy.y$.*

This encoding is not linear, hence the need to adapt it. The following encodings are from [Mai04]. All of them are defined in the `exp_def` file, and their correctness proofs are in `bci_proof` (see appendices).

Definition 9 (Booleans in linear λ-calculus)**.**

- $\top := Pair.$
- $\bot := \lambda^* xy.Pair \cdot y \cdot x.$

In this encoding, booleans are like $Pair$, but $\bot$ swaps the two arguments.

4.2.1 Discarding booleans

One problem with this encoding will arise when defining conjunction and disjunction: it will not be possible to obtain a single boolean as the output without deleting booleans. However we can define the following object to discard booleans, which shows that BCI algebras are not a model for reversible computation.

Definition 10 (Eliminator for booleans)**.** *The idea is to provide the boolean with two I (which erases the difference between True and False), which gives us $Pair \cdot I \cdot I$, and provide it with yet another I in order to obtain $I \cdot I \cdot I$, which is equal to I.*

$$Elim_B := \lambda^* b.b \cdot I \cdot I \cdot I$$

We have, as proven in the Coq file, $Elim_B \cdot \top = I$, and the same for $\bot$.

4.2.2 Boolean functions

Definition 11 (Negation)**.** *Since the difference between True and False is that False swaps the arguments, negation is obtained by swapping the arguments:*

$$Not := \lambda^* bxy.b \cdot y \cdot x$$

Definition 12 (Conjunction)**.** *The first boolean is applied to $\bot$, the second boolean and the eliminator for booleans. The idea is that the first boolean will determine which of $\bot$ and the second boolean will be affected by the eliminator.*

$$And := \lambda^* b_1 b_2.b_1 \cdot \bot \cdot b_2 \cdot Elim_B$$

Definition 13 (Disjunction)**.** *The idea is the same as for conjunction.*

$$Or := \lambda^* b_1 b_2.b_1 \cdot b_2 \cdot \top \cdot Elim_B$$

4.2.3 Copying booleans

Another useful thing we can do is to duplicate booleans, that is to go from a boolean b to a $Pair \cdot b \cdot b$.

Definition 14 (Duplicator)**.**

$$\begin{aligned} Copy_B = &\lambda^* b.b \cdot (Pair \cdot \top \cdot \top) \\ &\cdot (Pair \cdot \bot \cdot \bot) \\ &\cdot (\lambda^* UV.U \cdot (\lambda^* u_1 u_2.V \cdot (\lambda^* v_1 v_2.Pair \cdot (Elim_B \cdot v_1 \cdot u_1) \\ &\qquad \cdot (Elim_B \cdot v_2 \cdot u_2)))) \end{aligned}$$

If b is a boolean, it is either equal to $\top$ or to $\bot$, hence the result should either be $Pair \cdot \top \cdot \top$ or $Pair \cdot \bot \cdot \bot$. The variables U and V will hold these pairs, with b setting which variable holds which pair. The pair contained in V is then deleted, keeping only the one in U.

5 Considering discardable and duplicable objects

The fact that one can encode objects which can be discarded and duplicated is interesting considering that in the basic equations of BCI algebras all variables are present in the same numbers on both sides. We will look in this part at these objects in particular.

5.1 Eliminators and duplicators

We first have to define a general concept of eliminator and duplicator.

Definition 15 (Eliminator)**.** *An* eliminator *of an object x of a BCI algebra is an object $Elim$ such that*

$$Elim \cdot x = I$$

Definition 16 (Duplicator)**.** *A* duplicator *of an object x is an object $Copy$ such that*

$$Copy \cdot x = Pair \cdot x \cdot x$$

As an example, we can use eliminators to define an If-Then-Else, as a generalisation of the boolean *And* and *Or*.

Definition 17 (If-Then-Else)**.** *If we have an eliminator E for x and y:*

$$ITE := \lambda^* bxyE.b \cdot y \cdot x \cdot E$$

This works in the same way as And and Or, by discarding the right object, between x and y.

Lemma 1. *I has a duplicator and an eliminator.*

Proof. I is an eliminator for I, as $I \cdot I = I$.
$\lambda^* x.x \cdot (Pair \cdot I \cdot I)$ is a duplicator for I:

$$\begin{aligned} &(\lambda^* x.x \cdot (Pair \cdot I \cdot I)) \cdot I \\ =&I \cdot (Pair \cdot I \cdot I) \\ =&Pair \cdot I \cdot I \end{aligned}$$

An interesting result is that we can get an eliminator for a pair of objects given eliminators for the elements of the pair, and a duplicator for a pair provided we have a duplicator for each element. The proofs in Coq are in the `bci_proof` files.

Lemma 2. *If x and y have eliminators, then $Pair \cdot x \cdot y$ has one.*

Proof. if we have x_{elim} and y_{elim} eliminators for x and y respectively, we can apply each of them to the right component of the pair:

$$\lambda^* p.p \cdot (\lambda^* xy.(x_{elim} \cdot x) \cdot (y_{elim} \cdot y))$$

Lemma 3. *If x and y have duplicators, then $Pair \cdot x \cdot y$ has one.*

Proof. If we have x_{copy} and y_{copy} duplicators for x and y respectively, we can apply each to the right component of the pair and then reorganise the obtained objects.

$$\begin{aligned}\lambda^* p.p \cdot (\lambda^* xy.(x_{copy} \cdot x) \\ \cdot (\lambda^* x_1 x_2.(y_{copy} \cdot y) \\ \cdot (\lambda^* y_1 y_2.Pair \cdot (Pair \cdot x_1 \cdot y_1) \\ \cdot (Pair \cdot x_2 \cdot y_2))))\end{aligned}$$

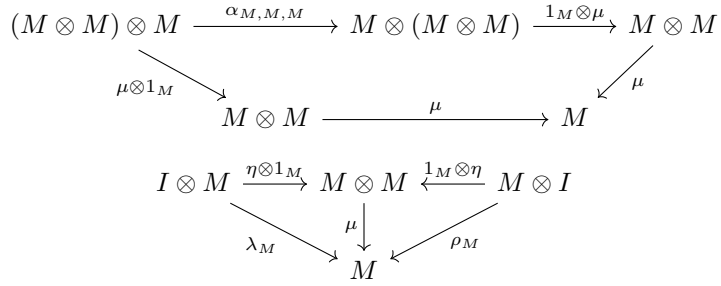
5.2 A cartesian subcategory of $Asm(\mathcal{A})$

We consider the category based on objects of $Asm(\mathcal{A})$ for which there exists an eliminator and a duplicator. To such an object $(X, \vDash_X)$ we associate the object $(X, \vDash_X, E_X, D_X)$, D_X and E_X being respectively a duplicator and an eliminator for the realisers of X. The morphisms are then the ones between the original objects of $Asm(\mathcal{A})$.

5.2.1 Commutative comonoids

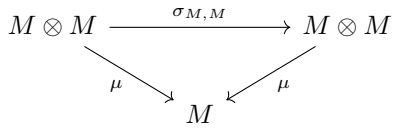
It is worth noting that this category corresponds to the category of commutative comonoids of $Asm(\mathcal{A})$, $CComon(Asm(\mathcal{A}))$.

Definition 18 (Monoid object)**.** *A monoid object in a monoidal category $(\mathcal{C}, \otimes, I)$ is an object M with a multiplication $\mu : M \otimes M \to M$ and a unit $\eta : I \to M$ such that the two following diagrams commute:*



Where α, λ and μ are the morphisms defined in definition 6.

Definition 19 (Commutative monoid)**.** *A monoid object M in a symmetric monoidal category is commutative if the following diagram commutes:*



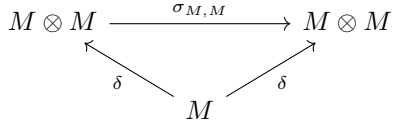
Where σ is the symmetry of definition 6.

Definition 20 (Comonoid object)**.** *A comonoid object in a monoidal category $\mathcal{C}$ is a monoidal object in the category $\mathcal{C}^{op}$.*

A comonoid M is thus equipped with a duplication $\delta : M \to M \otimes M$ and elimination $\varepsilon : M \to 1$.

Definition 21 (Commutative comonoid)**.** *Like in the previous definition, a commutative comonoid in $\mathcal{C}$ is a commutative monoid in $\mathcal{C}^{op}$.*

Alternatively, a comonoid M is commutative if the following diagram commutes:



which is equivalent to the first definition.

Proposition 3. *The category described earlier corresponds to $CComon(Asm(\mathcal{A}))$.*

Proof. It is clear from their definition that the comonoids of $Asm(\mathcal{A})$ do have an eliminator and a duplicator.

Conversely, if we have an object $(X, \vDash_X, D_X, E_X)$, then

$$\delta : \begin{array}{ccc} X & \to & X \times X \\ x & \mapsto & (x, x) \end{array}$$

is realised by D_X and

$$\varepsilon : \begin{array}{ccc} X & \to & I \\ x & \mapsto & \star \end{array}$$

is realised by E_X.

Since these two are simple functions on sets, they directly fill the required properties, including the one for commutativity.

5.2.2 A cartesian subcategory

Lemma 4. *$CComon(Asm(\mathcal{A}))$ is symmetric monoidal.*

Proof. The tensor product is still well defined, as lemmas 2 and 3 give us that the cartesian product is still realisable.
Lemma 1 gives us that the unit object is also in $CComon(Asm(\mathcal{A}))$.

Proposition 4. *$CComon(Asm(\mathcal{A}))$ is cartesian.*

Proof. Since it is symmetric monoidal, we need to show that there are two monoidal natural transformations: $\delta_X : X \to X \otimes X$ and $\varepsilon_X : X \to I$ such that

$$\lambda_X \circ (\varepsilon_X \otimes 1_X) \circ \delta_x = 1_X$$

$$\rho_X \circ (1_X \otimes \varepsilon_X) \circ \delta_x = 1_X$$

This is a result that can be found in nlab, or in the following lecture notes [HV12], along with the definition of monoidal natural transformation.

Deletion and copy filling these requirements exist for sets, thus we only need to show that they are realisable, which has been done in proposition 3.

5.2.3 A non linear core

Since $CComon(Asm(\mathcal{A}))$ is a subcategory of $Asm(\mathcal{A})$, we have at least booleans and expressions made of pairs and booleans which can be eliminated and duplicated. This forms a core of powerful types, in addition to the linear part corresponding to the rest of the category.

This is only a lower bound of the computing power of BCI algebras, I have for example not determined if some morphisms could be treated the same way, wich would give a cartesian closed category.

6 Conclusion

Although I failed to prove the correctness of my main function in Coq, which remains as a future work, we have an encoding using partitions which is quite concise and was efficient enough for the uses I had of it. Complexity will probably be an issue with bigger terms, but optimising Coq code is out of my reach for now.

The exact computing power of BCI algebras remains to be determined, as well as the limitations coming from the fact that the cartesian category is not closed in the final structure. It is still an interesting result, as it shows that BCI algebras are powerful enough to implement types which objects can be discarded or copied, and thus have a language with a core of powerful types.

An interesting follow-up would be to examine the relation of the structure we obtained to linear-non-linear models [Ben95], which are models of intuitionistic linear logic with a non linear core.

References

[Abr05] Samson Abramsky. A structural approach to reversible computation. *Theoretical Computer Science*, 2005. doi:10.1016/j.tcs.2005.07.002.

[AHS02] Samson Abramsky, Esfandiar Haghverdi, and Philip Scott. Geometry of interaction and linear combinatory algebras. *Mathematical Structures in Computer Science*, 2002. doi:10.1017/S0960129502003730.

[Atk18] Robert Atkey. Syntax and semantics of quantitative type theory. In *Proceedings of the 33rd Annual ACM/IEE Symposium on Logic in Computer Science*, 2018. doi:10.1145/3209108.3209189.

[Ben95] P. N. Benton. A mixed linear and non-linear logic: Proofs, terms and models. In *Computer Science Logic*, 1995. doi:10.1007/BFb0022251.

[BHKM12] Nick Benton, Chung-Kil Hur, Andrew J. Kennedy, and Conor McBride. Strongly typed term representations in coq. *Journal of Automated Reasoning*, 2012. doi:10.1007/s10817-011-9219-0.

[CS21] Chao-Hong Chen and Amr Sabry. A computational interpretation of compact closed categories: Reversible programming with negative and fractional types. *Proceedings of the ACM on Programming Languages*, 2021. doi:10.1145/3434290.

[Hin97] J. Roger Hindley. *Basic Simple Type Theory*. Cambridge University Press, 1997. doi:10.1017/CBO9780511608865.

[Hos07] Naohiko Hoshino. Linear realizability. In *Computer Science Logic*, 2007. doi:10.1007/978-3-540-74915-8_32.

[HV12] Chris Heunen and Jamie Vicary. Lectures on categorical quantum mechanics, 2012. URL: https://www.cs.ox.ac.uk/files/4551/cqm-notes.pdf.

[JS12] Roshan P. James and Amr Sabry. Information effects. *ACM SIGPLAN Notices*, 2012. doi:10.1145/2103621.2103667.

[Mai04] Harry G. Mairson. Functional pearl linear lambda calculus and ptime-completeness. *Journal of Functional Programming*, 2004. doi:10.1017/S0956796804005131.

[ML71] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer, New York, NY, 1971. doi:10.1007/978-1-4612-9839-7.

7 Appendices

7.1 Code

The Coq code is stored in the following repository, with some documentation: `https://gitlab.com/SamuelArsac/m1-internship`

The code is divided into two folders:

- The code for permutations, with only the encoding of BCI algebras, the file for permutations, and the file with the `lambda_s` function and some definitions to showcase its use.
- The code for partitions, with the same contents as the other one, with the addition of files containing the definition of all objects with BCI combinators, the proofs for their correctness and the proofs of their equality with the terms defined using `lambda_s`.

7.2 Meta-information

This internship was done remotely due to the sanitary conditions.

Approximate timeline of the internship:

- First month/month and a half: bibliography work.
- Next two months: implementation of BCI algebras and λ^* in Coq and work on implementing booleans as a primitive.
- Last month and a half: worked on boolean encodings, the cartesian subcategory and their encoding in Coq.
- Very end of the internship: implementation of λ^* in Coq using partitions.