Function compose, Type cut, And the Algebra of logic

XIE Yuheng
SZDIY community
xyheme@gmail.com

Abstract

In this paper, I demonstrate the Curry-Howard correspondence of Gentzen's sequent calculus, and give a very rich algebraic structure of the corresponding deduction system. I introduce then a prototype implementation of a programming language, written in scheme, which uses sequent calculus as its dependent type system.

Keywords language design, dependent type, deduction system, sequent calculus, algebraic structure.

1. Introduction

1.1 Curry–Howard correspondence of sequent calculus

Some said that in the framework of Curry–Howard correspondence, Gentzen's sequent calculus does not correspond with a well-defined pre-existing model of computation, as it was for Hilbert-style and natural deduction. [7]

in this paper, I will show that we can get what sequent calculus corresponds to, not by designing a new model of computation, but by changing the syntax of well known model.

I will show that sequent calculus corresponds to a functional programming language, the syntax of which is optimized for function composition instead of function application.

I will also show the trade-off of this syntax design.
(1) The syntax of type and function-body are unified.
(2) We lose the implicit syntax for currying.
(3) We gain the algebraic associative law.

1.2 The Algebra of logic

Some also said that deduction system can be viewed as algebraic structure, where theorems are the elements (like elements of group), where deductive rules are the operations (like multiplication of group). [1]

In this paper, I will show that with the associative law which we obtained from function composition, the corresponding deduction system of our programming language not merely "can be viewed as" algebraic structure, but actually has a very rich algebraic structure.

1.3 Implementation

I will also introduce a prototype implementation of such a language. I call it "sequent1".

Scheme and Functional Programming Workshop September 18th, 2016, Nara, Japan

Author's homepage at: xieyuheng.github.io

2. Background

2.1 How the idea was conceived

Two years ago, for some reason, I learned the design of the stack-based language – forth. I began to write my own forth-like language, and tried to add various features to it. Soon I found another forth-like language – joy. From joy I learned the composition v.s. application trade-off in syntax.

I added more and more features to my forth-like language. Things went well, until I tried to add a type system to the it. I found that to design a good type system, a designer has to know Curry–Howard correspondence well.

Then I figured that the composition v.s. application trade-off in syntax, corresponds to the sequent-calculus v.s. natural-deduction in proof theory.

Finally I decided to write a prototype language in scheme, only to demonstrate the idea of "sequent-calculus as type system",
thus I wrote "sequent1".

2.2 Related works

There are many other works that assign computational models to sequent-calculus-like logics. For example, Frank Pfenning assigned concurrent programming to linear logic. [2]

Maybe one logic system can correspond to multiple computational models. Maybe more terminologys should be carefully coined, and new framework and theory should be wisely designed, under which the correspondences between deduction systems and computational models could be elegantly described.

3. The change of syntax

I will introduce my syntax by comparing it with the syntax of an imaginary agda-like (or idris-like) language. [5] [6]
I will mark its syntax by `<< application-language >>`,
and I will mark my new syntax by `<< composition-language >>`.

3.1 Natural number

```
;; << application-language >>

data natural : type where
  zero : natural
  succ : natural -> natural

add : natural -> natural -> natural
add zero n = n
add (succ m) n = succ (add m n)

mul : natural -> natural -> natural
mul zero n = zero
mul (succ m) n = add n (mul m n)
```

```
;; << composition-language >>

;; In the following examples
;; ‘‘~’’ can be read as ‘‘define-function’’,
;; ‘‘+’’ can be read as ‘‘define-type’’.

;; The syntax of type and function-body are unified, where
;; a type is an arrow, and a function-body is a list of arrows.

(+ natural (-> type)
   zero (-> natural)
   succ (natural -> natural))

(~ add (natural natural -> natural)
   (:m zero -> :m)
   (:m :n succ -> :m :n add succ))

(~ mul (natural natural -> natural)
   (:m zero -> zero)
   (:m :n succ -> :m :n mul :m add))

;; for example, take (:m :n succ -> :m :n mul :m add) of
;; the function-body of ‘‘mul’’
;;
;;   it's antecedent is (:m :n succ)
;;     :m   -- (-> natural)
;;     :n   -- (-> natural)
;;     succ -- (natural -> natural)
;;   compose to (-> natural natural)
;;
;;   it's succedent is (:m :n mul :m add)
;;     :m  -- (-> natural)
;;     :n  -- (-> natural)
;;     mul -- (natural natural -> natural)
;;     :m  -- (-> natural)
;;     add -- (natural natural -> natural))
;;   compose to (-> natural)
;;
;; thus the type of ‘‘mul’’ is (natural natural -> natural)
```

3.2 Currying must be explicit

In type, input arguments and return values are made explicit.
instead of `(natural -> natural -> natural)`,
We write `(natural natural -> natural)`.

Thus, in function body, currying must also be explicit. We lost the implicit syntax for currying, because currying is designed as a convention for the syntax of function application.

3.3 Vector

```
;; << application-language >>

data vector : natural -> type -> type where
  null : vector zero t
  cons : t -> vector n t -> vector (succ n) t

append : vector m t -> vector n t -> vector (add m n) t
append null       l = l
append (cons e r) l = cons e (append r l)

map : (m : a -> b) -> f a -> f b
map f null       = null
map f (cons e l) = cons (f e) (map f l)


;; << composition-language >>

(+ vector (natural type -> type)
   null (-> zero :t vector)
   cons (:n :t vector :t -> :n succ :t vector))

(~ append (:m :t vector :n :t vector -> :m :n add :t vector)
   (:l null -> :l)
   (:l :r :e cons -> :l :r append :e cons))

(~ map (:n :t1 vector (:t1 -> :t2) -> :n :t2 vector)
   (null :f -> null)
   (:l :e cons :f -> :l :f map :e :f apply cons))
```

3.4 Function composition

```
;; << application-language >>

compose : {A B C : type} (A -> B) -> (B -> C) -> (A -> C)
compose f g =  x -> (f (g x))


;; << composition-language >>

;; The syntax is optimized for function composition.
;; Function composition is expressed by term concatenation.
```

3.5 Function application

```
;; << application-language >>

;; The syntax is optimized for function application.
;; Function application is expressed by term concatenation.


;; << composition-language >>

(~ apply (:a :b ... (:a :b ... -> :c :d ...) -> :c :d ...)
   (note it is implemented as a primitive-function))
```

3.6 Stack processing

Multiple return values are easily handled, and stack-processing functions can be used to help to re-order return values (without naming them) for function composition. (just like in forth & joy)

```
;; << composition-language >>

(~ drop (:t ->)
   (:d ->))

(~ dup (:t -> :t :t)
   (:d -> :d :d))

(~ over (:t1 :t2 -> :t1 :t2 :t1)
   (:d1 :d2 -> :d1 :d2 :d1))

(~ tuck (:t1 :t2 -> :t2 :t1 :t2)
   (:d1 :d2 -> :d2 :d1 :d2))

(~ swap (:t1 :t2 -> :t2 :t1)
   (:d1 :d2 -> :d2 :d1))
```

4. The correspondence

To show Curry–Howard correspondence under this syntax, is to show:
(1) How to view type as theorem?
(2) How to view function as proof?

4.1 Type as theorem

With the ability to handle multiple return values, we can express “and” as:

```
(A B -> C D) -- ‘‘(A and B) implies (C and D)’’
```

we can express “for all” and “there exist” in an unified way:

```
((:x : A) -> :x P) -- ‘‘for all x belong to A, we have P(x)’’
(-> (:x : A) :x P) -- ‘‘there exist x belong to A, such that P(x)’’
```

I call expression of form `(A B C ... -> E F G ...)` sequent, but you should note that, sequent for us, is not exactly the same as sequent for Gentzen[3]. Gentzen views succedent as “or”, while we view succedent as “and”:

```
for Gentzen -- (A B -> C D) -- ‘‘(A and B) implies (C or D)’’
for us      -- (A B -> C D) -- ‘‘(A and B) implies (C and D)’’
```

4.2 Function as proof

"function as proof" means, the way we write function body forms a language to record deduction. A record of many steps of deduction is called a proof.

Let us summarize deductive rules in sequent calculus in our language. I will omit some explicit contexts variables in the deductive rules, because in our language contexts can be implicit.

```
f : (A -> B)
g : (B -> C)
-------------- cut
f g : (A -> C)

f : (A -> C)
------------------- left-weakening
drop f : (A B -> C)

f : (A A -> B)
---------------- left-contraction
dup f : (A -> B)

f : (A -> B B)
----------------- right-contraction
f drop : (A -> B)

f : (A B -> C)
------------------- left-permutation
swap f : (B A -> C)

f : (A -> B C)
------------------- right-permutation
f swap : (A -> C B)

f : (A -> C)
------------------- left-and-1
drop f : (A B -> C)

f : (B -> C)
------------------------ left-and-2
swap drop f : (A B -> C)

f : (A -> B)
g : (C -> D)
--------------------------- right-and
g swap f swap : (A C -> B D)

f : (A -> B)
------------------- right-or-1
f : (A -> (B or C))

f : (A -> C)
------------------- right-or-2
f : (A -> (B or C))

f : (A -> B)
g : (C -> D)
---------------------------- left-or
(case (:x {:x : A} -> :x f)
(:y {:y : C} -> :y g))
: ((A or C) -> (B or D))

f : (A -> B)
g : (C -> D)
------------------------- left-implies
(:a :h -> :a f :h apply g)
: (A (B -> C) -> D)

f : (A B -> C)
---------------------- right-implies
(:x -> (:y -> :x :y f))
: (A -> (B -> C))
```

4.3 Example proofs

```
;; have-equal-human-rights

;; in the following example,
;; ``*'' can be read as ``define-hypothesis''

(* rich-human (:x is-rich -> :x is-human))
(* poor-human (:x is-poor -> :x is-human))
(* human-have-equal-human-rights
   (:x is-human :y is-human -> :x :y have-equal-human-rights))

(~ rich-and-poor-have-equal-human-rights
   (:x is-rich :y is-poor -> :x :y have-equal-human-rights)
   (:ri :po -> :ri rich-human
               :po poor-human
               human-have-equal-human-rights))


;; map/has-length

(+ list (type -> type)
   null (-> :t list)
   cons (:t list :t -> :t list))

(~ map (:t1 list (:t1 -> :t2) -> :t2 list)
   (null :f -> null)
   (:l :e cons :f -> :l :f map :e :f apply cons))

(+ has-length (:t list natural -> type)
   null/has-length (-> null zero has-length)
   cons/has-length (:l :n has-length ->
                    :l :a cons :n succ has-length))

(~ map/has-length (:l :n has-length -> :l :f map :n has-length)
   (null/has-length -> null/has-length)
   (:h cons/has-length -> :h map/has-length cons/has-length))


;; natural-induction

(+ natural (-> type)
   zero (-> natural)
   succ (natural -> natural))

(~ natural-induction
   ((:p : (natural -> type))
    zero :p apply
    ((:k : natural) :k :p apply -> :k succ :p apply)
    (:x : natural) -> :x :p apply)
   (:q :q/z :q/s zero -> :q/z)
   (:q :q/z :q/s :n succ ->
       :n
       :q :q/z :q/s :n natural-induction
       :q/s apply))
```

5. The Algebra of logic

A concrete (not abstract) algebraic structure is rich when:
(1) its elements have practical meaning.
(2) it is equipped with many algebraic laws, which you can use to transform equations.

A good example of such rich concrete algebraic structure is the field of multivariate rational function (i.e. quotient (or fraction) of multivariate polynomials), which is studied in algebraic geometry.

Since function composition already satisfies associative law, we have the opportunity to demonstrate a rich algebraic structure, the elements of which are formal theorems.

We will try to define those algebraic operations that are closed in the set of derivable theorems. Hopefully we will be able to capture all deductions by algebraic operations.

5.1 To mimic fraction of natural number

Let us view theorem `(A -> B)` as fraction, `A` as denominator, `B` as numerator. – Just like `(A \ B)`. (note that, we are using reverse-slash instead of slash, to maintain the order of `A B` in `(A -> B)`)

5.2 Multiplication

To multiply two theorems `(A -> B)` and `(C -> D)`, we get `(A C -> B D)`.
– Just like `(A \ B) (C \ D) = (A C \ B D)`.

```
(* r (A -> B))
(* s (C -> D))

(~ r/s/mul (A C -> B D)
   (:x :y -> :x r :y s))

;; abstract it to a combinator
(~ general/mul
   ((:a -> :b) (:c -> :d) -> (:a :c -> :b :d))
   (:r :s -> (lambda (:a :c -> :b :d)
               (:x :y -> :x :r apply :y :s apply))))
```

Theorems under multiplication is an Abelian group. Identity element is `(->)`. Inverse of `(A -> B)` is `(B -> A)`.

5.3 First definition of addition

This definition recalls the fraction of natural number.
To add two theorems `(A -> B)` and `(C -> D)`,
we get `(A B -> (B C or A D))`.
– Just like `(A \ B) + (C \ D) = (A C \ (B C + A D))`.

```
(* r (A -> B))
(* s (C -> D))

(~ r/s/fraction-add (A C -> (B C or A D))
   (:x :y -> :x r :y)
   (:x :y -> :x :y s))

;; abstract it to a combinator
(~ general/fraction-add
   ((:a -> :b) (:c -> :d) -> (:a :c -> (:b :c or :a :d)))
   (:r :s -> (lambda (:a :c -> (:b :c or :a :d))
               (:x :y -> :x :r apply :y)
               (:x :y -> :x :y :s apply))))
```

Distributive is just like fraction of natural number, because the way we define addition is just like the addition of fraction of natural number.

Theorems under addition is an Abelian semigroup. We do not have identity element, and we do not have inverse. (to make our algebraic structure more like fraction of natural number, we could introduce a "zero-theorem" (a theorem that we can never prove) as the identity element of addition)

Under this definition of addition, one may call the algebraic structure "natural field", to recall its similarites between the fraction of natural number. (note that, term like "semi-field" is ambiguous, because it does not inform us whether we mean addition is semi or multiplication is semi)

5.4 Second definition of addition

To add two theorems `(A -> B)` and `(C -> D)`,
we get `((A or B) -> (C or D))`.

```
(* r (A -> B))
(* s (C -> D))

(~ r/s/mul-like-add ((A or C) -> (B or D))
   (:x {:x : A} -> :x r)
   (:y {:y : C} -> :y s))

;; abstract it to a combinator
(~ general/mul-like-add
   ((:a -> :b) (:c -> :d) -> ((:a or :c) -> (:b or :d)))
   (:r :s -> (lambda ((:a or :c) -> (:b or :d))
               (:x {:x : :a} -> :x :r apply)
               (:y {:y : :c} -> :y :s apply))))
```

Distributive also hold under this definition of addition, because `(-> A (B or C))` is the same as `(-> (A B or A C))`.

Theorems under addition is an Abelian semigroup. Identity element is `(->)`, but we do not have inverse.

5.5 Term-lattice, and cut as weaken

This is where we must take term-lattice into account.

term	lattice
unification (uni)	meet
anti-unification (ani)	join
cover (or match)	greater-or-equal

(note that, "equal" can be defined by "greater-or-equal")

term-lattice is also called "subsumption lattice" by other authors. I call it "term-lattice", because I want to make explicit its relation with term-rewriting-system (I will address the detail of term-lattice in another paper).

If we have `(A -> B)` and `(C -> D)`,
we can cut them only when `(C cover B)`.
for example, when:

```
(1) C = B
(2) C = (B or E)
(3) C = :x :y P
    B = :x :x P
```

cut can be viewed as an important way to weaken a theorem. We can first multiply `(A -> B)` and `(C -> D)` to `(A C -> B D)`, then weaken it to `(A -> D)`, provided that `(C cover B)`.

We can also extend the lattice operations to cedent (antecedent and succedent), because cedent is Cartesian product of term.

5.6 Equality of theorem

We can define `A == B`, as `(A -> B)` and `(B -> A)`.

5.7 Constructiveness

In our language, we have the following keywords that make definitions:

keyword	read as	function-body
+	define-type an data	trivial
~	proof, define-function or theorem	non-trivial
*	assume, define-hypothesis	non

Whenever we have function-body, be it trivial or non-trivial, we can use it to rewrite data.

For example, the function-body of `succ` is trivial, it rewrites `zero` to `zero succ` (i.e. merely add a symbol to the data), while the function-body of `add` is non-trivial, it rewrites `zero succ zero succ` to `zero succ succ`.

Whenever we use "*" to introduce a hypothesis, the constructiveness of function is lost, because there is no function-body. Although we still can use it to define functions and type check the definitions, we can not use it to rewrite data. (but abstractiveness is gained, I will address the detail of the balance between constructiveness and abstractiveness in another paper)

5.8 Algebraic extension

When defining a new types by "+", we provide a type-constructor, and a list of data-constructors.

By introducing such constructors, we will extend our algebraic structure. (just like extending a field by the roots of equations)

6. Implementation

I made an attempt to implement a prototype of the language, project page at `http://xieyuheng.github.io/sequent1`. The implementation is a interpreter written in scheme.

6.1 Implementation-tech

During writing the prototype language, I noticed the language is not necessarily stack-based, and we have the following relations:

```
| implementation-tech     | evaluation strategy |
|-------------------------+---------------------|
| stack-based computation | call-by-value       |
| term-rewriting-system   | call-by-name        |
| graph-rewriting-system  | call-by-need        |
```

First few versions of sequent1 is implemented as a stack-based language, only later, changed to term-rewriting-system, because we have to handle lazy-trunk in the language, and in a term-rewriting-system, I can handle lazy-trunk in an unified implicit way.

6.2 Mistakes in my implementation

This prototype has the following mistakes to be fixed in the next versions of the prototype.

(1) I fail to far see that the structure of reports, which returned by various checkers, must be highly structured data instead of string, thus I fail to print useful reports when checkers find mistakes in code.

(2) I know graph-rewriting-system is needed, but I did not implement the language by it, because I want to keep the prototype simple.

(3) The prototype can not handle mutual recursive function.

(4) The prototype can not handle un-named "or".

(5) The meaning of equality is not fully understood.

(6) I have not yet designed a satisfactory mechanism to handle abstractiveness.

7. Further work

I plan to:

(1) develop further the algebra of logic.

(2) handle "equality" carefully, and use the language as a concrete tool to investigate algebraic topology.

Acknowledgments

I would like to thank my friend Jason Hemann, for his very valuable advice about this paper. I also would like to thank SZDIY community, for it provided me a place to work and live last year.

References

[1] The Univalent Foundations Program (2013), *Homotopy Type Theory – Univalent Foundations of Mathematics*.

[2] Frank Pfenning, Joint work with Lus Caires, Bernardo Toninho, and Dennis Griffith.
From Linear Logic to Session-Typed Concurrent Programming,
Tutorial at POPL 2015, Mumbai, India, January 2015

[3] Gentzen Gerhard (1969), M. E., Szabo, ed.,
Collected Papers of Gerhard Gentzen,
Paper #3: *INVESTIGATIONS INTO LOGICAL DEDUCTION (1935)*.
– In this paper the deduction rules of sequent calculus are formed.

[4] Paul Hertz (1922), *On Axiomatic Systems for Arbitrary Systems of Sentences, Part I, Sentences of the First Degree (On Axiomatic Systems of the Smallest Number of Sentences and the Concept of the Ideal Element)*.
– This paper inspired Gentzen to design his sequent calculus.

[5] Ulf Norell (2007), *Towards a practical programming language based on dependent type theory*.
– This paper describe the design of agda.

[6] Edwin Brady (2013), *Idris, a General Purpose Dependently Typed Programming Language: Design and Implementation*,
Journal of Functional Programming, August 2013.

[7] The wikipedia page of "Curry–Howard correspondence",
at the time when this paper is written.