情報処理学会論文誌　プログラミング　Vol.5　No.2　1–15 (Mar. 2012)

Coq上でのMonadic Total Parser Combinatorの実装

上里 友弥[1,a)]

受付日 2011年9月30日, 採録日 2012年1月20日

概要：本論文では，必ず停止するパーザのみを構成できるようなパーザコンビネータライブラリ（*total parser combinator library*）の実装手法について述べる．構成されるパーザが必ず停止することの保証は，定理証明支援系のCoqによって行う．Coqは，停止する関数だけを許すプログラミング言語としても使うことができるので，Coq上で定義することができればよい．パーザおよびパーザコンビネータはmonadicに実装したい．しかし一般に，monadicな実装においては証明の段階で逐一定義を展開していかなければならず，無駄が多いという問題がある．一方，この問題に対しては，Swierstraの提案したHoare state monadが有効である．そこで，我々はパーザをmonadicに実装する手段として，Hoare state monadを一般化したHoare state monad transformerを新たに提案する．このHoare state monad transformerを用いたことで，従来のmonadicな実装をもとにしながらも，停止性を比較的容易に証明することができた．

キーワード：パーザコンビネータ，停止性，形式的検証，Coq

Implementing Monadic Total Parser Combinator on Coq

Yuya Uezato[1,a)]

Received: September 30, 2011, Accepted: January 20, 2012

Abstract: We have implemented a total parser combinator library that can constitute only a parser that terminates for all inputs. The termination is guaranteed by using the Coq proof assistant as a programming language that allows only terminating functions. If a parser is implemented with the library on the Coq, then termination of it is really guaranteed. It is desirable to implement parsers and parser combinators in the monadic style. However, when a program is implemented in the monadic style, it is usually necessary to unfold the definition of monads in the process of proving, and the unfolding makes the proofs rather cumbersome. To overcome this problem, we generalize Hoare state monads of Swierstra to Hoare state monad transformers. By using Hoare state monad transformers, we maintain the monadic style implementation, and also it is relatively easy to prove the termination of constructed parsers.

Keywords: parser combinators, termination, formal verification, Coq

1. はじめに

プログラミング言語を実装する際には，パーザの実装が必要となる．その際にパーザジェネレータを用いることも多いが，一方でパーザを手書きすることもある．パーザを手書きするにも様々な方法があるが，特にパーザコンビネータを用いると，ホスト言語の機能を多く用いることができ，単にプログラムを書くのと同じようにしてパーザを作ることができる．

パーザコンビネータを用いてパーザを構成していくためのライブラリとしてはHaskellのParsec [1] などがある．このようなライブラリを用いると，様々な再帰下降パーザを構築できる．しかし，構築に関する制約がないので，次のように左再帰的なパーザも定義できてしまう．

```
expr  = do { e ← expr; op ← minus; d ← digit;
             return (op e d) }
      <|> digit
minus = do { symb "−" ; return (−) }
```

[1] 筑波大学情報学群情報科学類
College of Information Science, University of Tsukuba, Tsukuba, Ibaraki 305–8573, Japan
[a)] s0811425@coins.tsukuba.ac.jp

このパーザは，たとえば parse expr "" のように実行されたとすると無限にループしてしまう．このような無限ループするパーザは単体でも無意味であるが，無意味なパーザを用いて新たにパーザが作られると，それもまた無意味なものになってしまう．つまりバグの温床となるのである．

プログラミング言語のパーズは入力に対し必ず停止することが期待されることからも，無限ループするようなパーザは定義できないことが望ましい．小さなパーザ群であれば，全体として必ず停止するパーザとなっているかどうかを調べることはやさしいだろうが，ある程度のサイズになるとそれも難しくなる．また，停止性を形式的に，確かに証明したいということもあるだろう．そこで，停止するパーザだけを構築できるようなパーザコンビネータライブラリ（*total parser combinator library*）が望まれる．

本論文ではそのようなライブラリを定理証明支援系 Coq [2] を用いて実装する．Coq はそもそも，停止する関数だけを許すプログラミング言語とみることもできるので，Coq 上でパーザコンビネータライブラリを実装することができれば，そのライブラリを用いて作るパーザは任意の入力に対し停止することが保証される．

パーザコンビネータはあくまでコンビネータであるから，パーズ対象の文字列が，その定義に出てくるのは望ましくない．このような場合，monadic にプログラムを書くことはよく知られたイディオム [3] である．そこで，Coq で実装する際にも monadic に実装したい．

さて，パーザ全体が停止することを示すのに，端的にいえば未処理の文字列が徐々に減少することを示すことになる．ところが，monadic に作られたプログラムの定義においては減少性をいうための文字列が出現しないため，停止性の証明においてはあえて monadic な定義を崩さなければならず，大変に無駄である．

一方，monadic な定義を採用することでかえって証明が煩わしくなるという問題に対しては，State monad を一般化した Hoare state monad が有効であることが分かっている [4]．しかし，パーザコンビネータの定義では State monad および Hoare state monad ではなく，State monad transformer と呼ばれる，State monad に対する別の一般化された構造が必要となる．

そこで本論文では，パーザコンビネータを monadic に実装するために，Hoare state monad をさらに一般化した Hoare state monad transformer を新たに提案する．提案手法である Hoare state monad transformer を用いることで，Coq 上においてもパーザが monadic なコードで単純に実装できることを示す．

本論文の構成は次のようになっている．

2 章では，Coq による定義を行うもととなるパーザコンビネータライブラリの実装とその考え方を紹介する．

3 章では，本論文の考えのもととなった，Hoare state monad について調べる．

4 章では，新たに提案する Hoare state monad transformer が Hoare state monad の一般化であること，そして State monad transformer の定義をもとに実装できることを述べる．

主題となる 5 章では，まずパーザコンビネータライブラリの実装と，State monad transformer の関係を簡単に述べる．そのうえで，Hoare state monad transformer を用いると，従来どおりの定義のままに，停止性が証明できることを示す．さらに，通常の State monad transformer を用いた場合と比較して，提案手法の有効性を示す．

6 章を関連研究の紹介と比較とし，7 章で問題点と今後の課題，8 章で結論を述べる．

2. Monadic Parser Combinator

本論文におけるパーザコンビネータライブラリのセマンティクスおよび実装は，Haskell による monadic なパーザコンビネータライブラリの実装を述べた文献 [3] をもとにする．

文献 [3] では，Parser 型を次のように定義している．

```
newtype Parser a = Parser (String →
[(a,String)])
```

この定義において Parser a は，文字列を受け取り，パーズ結果として型 a の値と，残った文字列のペアをリストで返すことを表す．特に結果がリストとなっていることについては

> Returning a list of results allows us to build parsers for ambiguous grammars, with many results being returned if the argument string can be parsed in many different ways. [3]

とあるように，非決定的なパージングを行うためである．

上の Parser 型が確かにモナドとなることを中心に文献 [3] は書かれているのだが，そのためには次のように適切に return と bind を定義をしなければならない．

```
parse (Parser p) = p

instance Monad Parser where
  return a = Parser (\cs → [(a,cs)])
  bind p f = Parser (\cs →
    concat [parse (f a) cs' |
            (a,cs') ← parse p cs])
```

この定義で return a は，入力をそのまま結果とする，すなわち入力をいっさい消費しない，a を値とするパーザを作る．bind p f は，パーザ p を用いて入力をパーズし，そのすべての結果を再び f でパーズするというパーザを作る．この return と bind がモナド則を満たすことは容易に確か

められる．

その他の基本的なパーザコンビネータ，たとえば，非決定的な選択子++や，単一の結果を返す決定的な選択子+++も次のように自然に定義できる．

```
p ++ q = Parser (\cs → parse p cs ++
parse q cs)

p +++ q = Parser (\cs →
case parse (p ++ q) cs of
  [] → []
  (x:xs) → [x])
```

これらを用いて，chainl1 と呼ばれるコンビネータを次のように定義できる．

```
chainl1 :: Parser a → Parser (a → a → a) →
Parser a
chainl1 p op = bind p (\a → rest a)
  where
   rest a =
   (bind op (\f → bind p (\b → rest (f a b))))
   +++ (return a)
```

この chainl1 を用いると，非形式的ではあるが次のような生成規則を得る．

```
chainl1 p op ::=
... ((p op p) op p) ... | ... | (p op p) | p
```

厳密には，非決定的な振舞いではなく op を中置演算子とみて左結合で貪欲にパーズする．chainl1 を用いれば，左再帰的な文法を右再帰的に変形しても，演算子などを左向きの結合を保ったままパーズができるようになる．たとえば 1 章であげた無限ループするパーザを変形し，次のように定義すると，マイナス演算子の左向き結合を保ちつつも必ず停止するパーザとすることができる．

```
expr = chainl1 digit minus
minus = do { symb "–" ; return (–) }
```

3. Hoare State Monad

本論文でいう *Hoare state monad* とは文献 [4] におけるものとまったく同じである．Hoare state monad は，その名のとおりではあるが，State monad をもとにした考えである．そこで，State monad から順に述べることとする．

3.1 State Monad

State monad は Coq で次のように定義できる．

```
Definition State (s a : Set) := s → (s * a).
Definition ret (s a : Set) : a → State s a
:= fun x i ⇒ (i , x).
Definition bind (s a b : Set) :
State s a → (a → State s b) → State s b
:= fun c f s1 ⇒ let (s2 , x) := c s1 in f x s2.
```

この State s a は，型 s の初期状態のもとで，型 a の値と型 s の最終状態を生成するという計算を表している．

ret x は，状態を変更せず値を x とする計算である．

bind c f は初期状態 s1 と計算 c より新しい状態 s2 と値 x を得た後，その値に依存する関数 f のもとで新たな計算 f x を得て，その初期状態として s2 を用いるという意味である．

このように定義したとき，型 State がモナドになるので State monad と呼ばれる．実際にこれらの定義でモナド則を満たすことを証明するのは容易である．

3.2 State Monad を用いた例

State monad を用いた簡単なプログラムは**図 1** のようになる．

get は，初期状態そのものを最終状態と値にする計算で，これによって現在の状態に依存する計算が定義できる．put s は，状態を s に変更する計算である．

recog c は，初期状態（ここでは文字列）の先頭文字が c と一致していれば受理，そうでなければ拒否する認識機械（recognizer）である．bind を用いて，自然に認識機械を結合し，新たな認識機械を作ることができる．たとえば combine_recog r1 r2 は，初期状態に対し r1 が受理し，かつ次の状態（すなわち残りの文字列）を r2 が受理するならば全体は受理，そうでなければ拒否とする認識機械である．

3.3 State Monad から Hoare State Monad へ

Hoare state monad は，初期状態と最終状態に制約をつけたものである．逆にいえば State monad は制約が存在しない．このような意味で Hoare state monad は State monad の一般化であるといえる．初期状態と最終状態に対する制約と，何より Hoare を冠するところからも分かるが，Hoare logic [5], [6] から発想を得ている．

ここでの Hoare state monad の定義は文献 [4] にならい，次のようにする．

```
Definition Pre (s : Type) : Type := s → Prop.
Definition Post (s : Type) (a : Type) : Type
:= s → a → s → Prop.

Program Definition HoareState (s : Type)
(pre : Pre s) (a : Type) (post : Post s a) : Type
:= ∀ (i : {t : s | pre t}) ,
     {(x , f) : a * s | post i x f}.
```

上の定義で，HoareState は，事前条件 pre を満たす初期状態 i だけを考え，かつその計算結果であるペア (x , f) が

```
Definition get (s : Set) : State s s := fun i ⇒ (i, i).
Definition put (s : Set) (new_s : s) : State s unit := fun i ⇒ (new_s, tt).

Definition recog (c : char) : State string bool :=
bind get
(fun i ⇒
  match i with
  | x :: xs ⇒
    match ascii_dec c x with
    | left _ ⇒ bind (put xs) (fun _ ⇒ ret true)
    | right _ ⇒ ret false
    end
  | nil ⇒ ret false
  end).

Definition combine_recog (r1 r2 : State string bool) : State string bool
:= bind r1 (fun v ⇒ if v then r2 else ret false).
```

図 1　State の実例
Fig. 1　Example of State.

```
Notation top s := (fun _:s ⇒ True).

Program Definition ret (s a : Type) :
∀ (x : a),HoareState s (top s) a (fun i y f ⇒ i = f ∧ y = x)
:= fun x s ⇒ (x , s).

Program Definition bind : ∀ s a b P1 P2 Q1 Q2,
(HoareState s P1 a Q1) →
(∀ (x : a) , HoareState s (P2 x) b (Q2 x)) →
HoareState s (fun s1 ⇒ P1 s1 ∧ ∀ x s2 , Q1 s1 x s2 → P2 x s2)
             b
             (fun s1 y s3 ⇒ ∃ x s2 , Q1 s1 x s2 ∧ Q2 x s2 y s3)
:= fun s a b P1 P2 Q1 Q2 c1 c2 s1 ⇒
  match c1 s1 with
  | (x , s2) ⇒ c2 x s2
  end.
```

図 2　*return* および *bind* の定義
Fig. 2　Definitions of *return* and *bind*.

事後条件 post を満たさなければならないことを述べている．HoareState に関しても return と bind を定義する必要がある．これらを，**図 2** で与える．

図 2 における ret の事前条件は，top という述語を用いてすべての状態を初期状態としてとれるものとした．また，bind がこのような型を持つことの説明は文献 [4] にある．

ここで ret と bind の関数定義の本体が，それぞれ State のそれと一致していることが重要である．したがって，型だけでなく，計算的な意味もともなった一般化だということが分かる．

3.4　Hoare State Monad を用いた例

図 1 で定義した関数 recog に関する次の性質 recog_property を考える．

```
Theorem recog_property :
∀ c s , snd (recog c s) = true →
s = c :: fst (recog c s).
```

これを証明するには，以下の形のゴールを示す必要がある．ここでは，ゴールに出現する bind を *unfold*（関数の定義を展開する Coq の tactic）してやる必要がある．このように，証明の過程でいちいち ret や bind を *unfold*，すなわち monadic に書かれたプログラムをわざわざ崩さなければならず，非常に無駄が多い．

```
Program Definition recog (c : char)
: HoareState (list char) (top (list char)) bool
(fun i x f ⇒ if x then i = c :: f else i = f)
:=
bind
(P2 := fun v i ⇒ v = i)
(Q2 := fun _ i (x : bool) f ⇒ if x then (i = c :: f) else (i = f))
get
(fun i ⇒
  match i with
  | x :: xs ⇒
    match ascii_dec c x with
    | left _ ⇒ bind (put xs) (fun _ ⇒ ret true)
    | right _ ⇒ ret false
    end
  | nil ⇒ ret false
  end).
```

図 3　Hoare state monad を用いた *recog* の定義
Fig. 3　Definition of *recog* using Hoare state monad.

```
c : char , c0 : char , s : list char
H : snd (recog c (c0 :: s)) = true
============================
 c0 :: s = c :: fst (bind get (fun i : list ascii ⇒...))
```

一方 Hoare state monad においては recog の定義をする際に，その型に事後条件として同等のものを書くことができる．図 **3** で定義された recog が，元の recog と同じ定義であることに注目してほしい．さらにそのうえで，事前条件と事後条件を型で明示させたことで，単にプログラムを定義していくだけで，結果的に recog_property と同じ性質が成り立つことが示せた．

ここまでの定義で，単なる **Definition** ではなく，**Program** [7], [8] を用いた．**Program** を用いた場合の利点として，定義を単純にできることがある．実際，ret, bind, recog はきわめて自然な定義になっている．ただし，定義から証明が自動的にできなかった部分は，明示的に証明しなければならない．

4. Hoare State Monad Transformer

本章では，Hoare state monad transformer を新たに提案し，その定義をあたえる．

Hoare state monad transformer とは，Hoare state monad を Monad transformer（モナド変換子）にしたもので，その考えは陽に State monad transformer に基づいている．そこで，まず State monad transformer について述べる．

4.1 State Monad Transformer

現在 Haskell で用いられている，次のような State monad transformer [9] の定義を用いる．

```
data StateT s m a = StateT (s → m (a , s))
```

この StateT s m a において，s は状態，m は State effect が付与されるモナド，a は計算結果を意味している．

モナド変換子は，モナドからモナドを構成する射 [10] なので，StateT s m に対し return と bind が定義される．同時に，get，put のモナド変換子版も定義する．これらを，次のようにする．

```
instance Monad m ⇒ Monad (StateT s m) where
  return a = StateT (\s → return (a , s))
  bind (StateT m) k = StateT (\s →
    bind (m s) (\(a,s') →
    let (StateT m') = k a in m' s'))

get   = StateT (\s → return (s, s))
put s = StateT (\_ → return ((), s))
```

この定義のもと，StateT s m がモナド則を満たすことは容易に確かめられる．

次に StateT をもとに，HoareStateT の定義を行う．

4.2 HoareStateT

4.2.1 HoareStateT 型の定義

HoareState は，State の一般化であると述べたが，その一方で return と bind は同じ定義であったことを思い出してもらいたい．ここで，HoareStateT に関しても，StateT と同じ定義を用いたい．よって，定義は次のような形になるだろう．

```
Program Definition HoareStateT (s : Type)
```

```
s : Type , a : Type , m : Type →Type
monad : Monad m , aux : Aux m monad
x : a , i : s , H : True
============================
 get_and_pr aux (returns (x, i)) (prod_curry (fun (y : a) (f : s) ⇒i = f ∧x = y))
```

図 4　Ret の定義から生成されるゴール
Fig. 4　The goal is generated from definition of Ret.

```
m (monad : Monad m)
(pre : Pre s) (a : Type) (post : Post s a) : Type
:= ∀i:{ t : s | pre t},{ mv: m (a * s) | ... }
```

すると，上の定義中の（...）の部分をどのようにして埋めるのかという問題が生じる．この部分を埋めるべく，(a * s) 型の **m** 計算 [11]mv に対し，モナド **m** 内部で post を用いて計算し，命題を得ることにする．計算の実行方法はモナド **m** 自身が知っているはずなので，次のような関数 get_and_pr を仮定する．

```
Record Aux (m : Type → Type) (monad : Monad m)
:= mkAux {
  get_and_pr : ∀ A , m A → (A → Prop) → Prop
}.
```

直観的には，get_and_pr ma P をもって，bind ma (fun a ⇒ ...) で a に束縛される値がつねに P を満たすという命題を表したい．

なお，get_and_pr のようにモナド **m** ごとに満たすべき要求は，上記のように Coq の **Record** を用いてまとめていくこととする．モナド **m** ごとに bind による計算の実行方法が異なるため，その部分は各々に委ねるという方針である．

このとき HoareStateT を以下のように定義できる．

```
Program Definition HoareStateT ...
(aux : Aux m monad) ...
:= ∀ i : { t : s | pre t} ,
{ mv : m (a * s) |
(get_and_pr aux) mv (prod_curry (post i)) }.
```

定義中の prod_curry は，uncurrying を行う Coq の標準ライブラリにある関数である．

引き続き return と bind 相当の実装を行う．

4.2.2　return の定義

return はどのように定義するべきだろうか．まず，事前条件および事後条件は HoareState の ret と同じものにしたい．同時に，計算的にも StateT の return と等しい定義としたい．この 2 つを考慮すると，次のような定義となるだろう．

```
Program Definition Ret (s a : Type)
m monad (aux : Aux m monad) :
∀ (x : a) , HoareStateT s aux
(top s) a (fun i y f ⇒ i = f ∧ x = y)
:= fun x i ⇒ @returns m monad (a * s) (x , i).
```

上の定義は，**Program** を用いたものである．その結果として，自動的に証明できなかった**図 4** のゴールが生成される．ここで，get_and_pr (returns a) の形の式を示す術がないので，新しく仮定を入れなければならない．*return a* は値 a をモナドに含めることであるから，get_and_pr の直観的な意味を考えると，次の要求は妥当だろう．

```
aux_law1 : ∀ A a (p : A → Prop) ,
  get_and_pr A (returns a) p ↔ p a
```

この aux_law1 を用いると，ただちにゴールを示せる．

4.2.3　bind の定義

bind の定義はどうなるだろうか．Ret を定義したときと同様に，事前条件と事後条件は HoareState の bind と同じであるべきだし，計算的には StateT の bind と等しくあるべきと考える．したがって，次のように定義したい．

```
Program Definition Bind ...
(aux : Aux m monad) ...
HoareStateT s aux P1 a Q1 →
(∀ (x : a) , HoareStateT s aux (P2 x) b (Q2 x)) →
HoareStateT s aux (fun s1 ⇒ P1 s1 ∧ ∀ x s2 ,
Q1 s1 x s2 → P2 x s2)
                    b
                    (fun s1 y s3 ⇒ ∃ x s2 ,
Q1 s1 x s2 ∧ Q2 x s2 y s3)
:= fun a b P1 P2 Q1 Q2 c1 c2 s1 ⇒
bind (c1 s1)
(fun mv ⇒ match mv with
          | (v , s2) ⇒ (c2 v) s2
          end).
```

このとき，先と同様に解かなければならない**図 5** のゴールが自動的に生成される．

これは，具体的には，定義における下線部 (c2 v) s2 の関数適用にあたって満たすべき事前条件そのものである．しかしこの形ではどのようにしてもゴールを証明することができない．P2 a0 s0 を得るためには Q1 s1 a0 s0 を示せばよいのだが，Q1 に関して，すでに示されていることは仮

```
Record Aux (m : Type → Type) (monad : Monad m) := mkAux {
  get_and_pr : ∀ A , m A → (A → Prop) → Prop ;

  aux_law1 : ∀ A a (p : A → Prop) ,
    get_and_pr _ (returns a) p ↔ p a ;

  aux_law2 : ∀ A (ma : m A) (P Q : A → Prop) ,
  (∀ a : A , P a → Q a) → get_and_pr _ ma P → get_and_pr _ ma Q ;

  bind' : ∀ A B (P : A → Prop) (Q : B → Prop)
    (ma : m A) (wit : get_and_pr _ ma P)
    (f : ∀ a , P a → {m_v : m B | get_and_pr _ m_v Q})
   , {m_v : m B | get_and_pr _ m_v Q}
}.
```

図 6　レコード Aux の定義
Fig. 6　Definition of record Aux.

```
...
H : P1 s1
H0 : ∀(x : a) (s2 : s)
Q1 s1 x s2 →P2 x s2
a0 : a , s0 : s , H1 : m (a * s)
H2 : get_and_pr aux H1 (prod_curry (Q1 s1))
============================
 P2 a0 s0
```

図 5　Bind の定義から生成されるゴール
Fig. 5　The goal is generated from definition of Bind.

定 H2 のみである．

Bind の定義における，bind の計算部分 (fun mv ⇒ ...) での mv は，(a * s) 型の値になっていることに注目し，この mv に対し Q1 が成り立つことを要請する次の形の bind' を導入する．

```
bind' : ∀ A B P Q
 (ma : m A) (wit : get_and_pr _ ma P)
 (f : ∀ a , P a → {m_v : m B | get_and_pr _ m_v Q}),
 {m_v : m B | get_and_pr _ m_v Q}
```

この bind' を用いて Bind を定義しなおすと，次のようになる．これで，無事に問題は解決される．

```
Program Definition Bind ...
:= fun a b P1 P2 Q1 Q2 c1 c2 s1 ⇒
bind' aux (fun a_s ⇒ prod_curry (Q1 s1) a_s)
_ (c1 s1) _
(fun mv pr ⇒
match mv with
| (v , s2) ⇒ (c2 v) s2
end).
```

新しい Bind の定義に対して **Program** により生成される命題を証明していくと，次の形のゴールに行き当たる．

```
...
a0 : a
s0 : s
pr : Q1 s1 a0 s0
g : get_and_pr aux x (prod_curry (Q2 a0 s0))
============================
 get_and_pr aux x
   (prod_curry (fun (y : b) (s3 : s) ⇒
     ∃x0 s2, Q1 s1 x0 s2 ∧Q2 x0 s2 y s3))
```

いま，Q1 s1 a0 s0 と get_and_pr aux x (prod_curry (Q2 a0 s0)) が分かっていることに注目し，ゴールにおいて *exists* で束縛されている x0，s2 はそれぞれ a0，s0 とすればよさそうである．ただ，ゴールをこのままでは証明することはできない．そこで，次の形を仮定する．

```
aux_law2 : ∀ A (ma : m A) P Q ,
  (∀ a : A , P a →Q a) → get_and_pr _ ma P →
  get_and_pr _ ma Q
```

この aux_law2 の意味するところは，Q より強い命題 P があり，それがすでに get_and_pr のもとで証明されているならば Q についても示せるというものである．aux_law2 を現在のゴールに対して用いれば，ただちに証明を完了できる．

以上をもって，生成されたすべての命題の証明ができたので，HoareStateT に対する Bind の定義ができたことになる．また，最終的なレコード型 Aux は図 **6** のようになった．

このように，(Hoare) State effect を付与される側のモナド **m** ごとで，Aux を満たすことを条件として，Hoare state monad のモナド変換子版である Hoare state monad transformer が定義できた．

5. Coqによる Total Parser Combinator

ここまでの話で，本論文の主題である Coq を用いたパーザコンビネータライブラリを実装することができる．本章では特に文献 [3] に対応するものを実装する．

5.1 StateT を用いた Parser 型の定義

さて，すでにパーザコンビネータの考え方と，Hoare state monad およびそのモナド変換子について述べたが，これらはどのように結び付くのだろうか？

その答えは型 Parser にある．この型は

```
newtype Parser a = Parser (String → [(a,String)])
```

と 2 章で定義したが，これは

```
type Parser a = String → [(a,String)]
```

と本質的に同じものである．ところで，この型の形は前述の StateT に大変似ている．それどころか，次のように StateT をリストモナドに適用すれば Parser モナドを得ることができる．

```
type Parser a = StateT String [] a
```

この考え方自体は決して新しいものでない [10], [12]．

同じようにして，HoareStateT で Parser を定義することができる．これでパーザコンビネータライブラリと Hoare state monad transformer がつながったといえるだろう．ただし，Coq で Parser 型を定義するためには，リストモナドで要求 Aux を満たさなければならない．

5.2 リストに対する Aux の実装

先立って次のようにリストをモナドとして定義する．

```
Class Monad (m : Type → Type) := {
  returns : ∀ {A} (a : A) , m A ;
  bind    : ∀ {A B} , m A → (A → m B) → m B;
}.
Instance list_is_monad : Monad list := {|
  returns X x := x :: [] ;
  bind A B m f := flat_map f m
|}.
```

この returns と bind の定義は，Haskell におけるリストモナドの定義と一致する．

リストに関する get_and_pr は次のように定義する．

```
Definition list_get_and_pr : ∀ A , list A → (A →
Prop) → Prop
:= fun A L P⇒
  fold_right (fun (e : A) (acc : Prop) ⇒ P e ∧
acc) True L.
```

ここで，list_get_and_pr A L P の意味は，A 型のリスト L のメンバすべてが命題 P を満たすということである．

続いて，リストモナドが要求 Aux を満たすことを次のように定義する．

```
Program Definition list_aux:(Aux list list_is_monad)
:= {|
  get_and_pr := list_get_and_pr ;
  bind' A B P Q ma wit f
  := concat' B Q (map' _ _ ma P f wit)
|}.
```

上の定義中に出現する map' と concat' はそれぞれ次のような型を持つ．

```
Program Fixpoint map' A B (ma : list A) :
∀ P (f : ∀ (a : A) , P a → B)
(wit : list_get_and_pr _ ma P) , list B.
Program Fixpoint concat' A P
(l : list { ma : list A |
            list_get_and_pr _ ma P }) :
{ ma : list A | list_get_and_pr _ ma P }.
```

ここで，map' は，いわゆるリストの map 関数と計算的に同じ意味を持つ．concat' についても，リストのリストからリストを作る通常の concat 関数と計算的には同じである．したがって，bind' の定義は，計算としては通常のリストモナドの bind の定義と一致する．

以上により list_aux が定義できたので，次のように Parser 型を定義する．

```
Notation Parser := (HoareStateT string list_aux).
```

5.3 パーザコンビネータ群の定義

本節では，5.2 節で定義した Parser 型を用いていくつかのパーザを定義する．

5.3.1 パーザ item

最初にパーザ item を以下のように定義する．item は基本的なパーザであり，入力が空文字列でなければ，1 文字目を消費し，結果として返すものである．一方，入力が空文字列であれば失敗する．

```
Notation Top := (fun _:string ⇒ True).

Program Definition item :
Parser Top char
       (fun i x f⇒x :: f = i ∧ length f <
        length i)
:= fun i ⇒match i with
| [] ⇒ []
| (x :: xs) ⇒ [ (x , xs) ]
```

```
Notation "X ← A ; B" := (Bind (aux := list_aux) A (fun X ⇒ B))

Program Fixpoint Many A
(p : Parser Top A (fun i _ f ⇒ length f < length i))
(inp : string) { measure (length inp) } :
Parser
 (fun i ⇒ length i <= length inp) (list A)
 (fun i _ f ⇒ length f <= length i) :=
do _ _ (
 (v ← p ;
  inp' ← get (fun inp' ⇒ length inp' < length inp) ;
  vs ← @Many A p inp' _ ;
  Ret (v :: vs)) +++
 (Ret [])
).

Program Definition many A
(p : Parser Top A (fun i _ f ⇒ length f < length i)) :
Parser Top (list A) (fun i _ f ⇒ length f <= length i)
:= fun i ⇒ (Many A p i) i.
```

図 7　Many および many の定義
Fig. 7　Definitions of Many and many.

end.

上の item の定義は，文献 [3] によるものに等しい．また，この **Program** を用いた定義は，特別に証明を与えることなく Coq が受理する．

ここで，item の型に注目する．任意の入力に対して使いたいパーザなので，事前条件として，制約を課さない述語 Top を用いている．次に，事後条件は以下の 2 つのことを表している．

- x :: f = i : 得られる値 x と残り f を結合したものが入力 i である
- length f < length i : 先頭文字を消費するので，残り f は入力 i よりも短い

1 つ目は，このパーザに対して期待する振舞いである．2 つ目は，停止性を証明するために重要な性質である．

5.3.2　パーザコンビネータ many

次に，再帰的なパーザコンビネータである many を定義する．many は，パーザを引数としてとり，そのパーザを失敗するまで繰り返し用い，結果をリストとして返すようなパーザを生成する．

many は再帰的な関数となるから，再帰呼び出しするごとに入力文字列が何らかの形で減少するようになっていなければならない．たとえば，many (Ret 1) のように，文字を消費せずに成功するパーザを many に渡した結果得られるパーザは，どのような入力に対しても無限ループすることは明らかだろう．そこで，many に渡すことのできるパーザは必ず文字を消費するように要請する．たとえば前述の item のように，事後条件として fun i x f ⇒ length f < length i が成り立てばよい．

一方，停止性を示すために，many の補助関数となる Many を定義することとする（**図 7**）．

Many では，引数 inp を明示的にとるようにしている．再帰呼び出しするごとに，この引数部分が長さに関して短くなることを利用して停止性を保証する．そのために引数 inp と，実際の入力の関係が必要になるが，これには事前条件を用いて fun i ⇒ length i ≤ length inp とした．入力の長さ（length i）がたかだか length inp 以下であるものが実行可能ということを意味している．

この補助関数 Many を用いれば，通常の many をただちに定義することができる．

また，図 7 では，分かりやすさのために Bind を直接用いるのではなく，Haskell の do 記法に似せたものを用いている．加えて，do，get，+++を新たに用いている．その各々は，以下のとおりである．

```
Program Definition do s m monad (aux :
Aux m monad) :
∀ a (P1 P2 : Pre s) (Q1 Q2 : Post s a)
(str : ∀ i , P2 i → P1 i)
(wkn : ∀ i x f , Q1 i x f → Q2 i x f) ,
HoareStateT s aux P1 a Q1 →
HoareStateT s aux P2 a Q2 :=
fun a P1 P2 Q1 Q2 str wkn c ⇒ c.
Next Obligation.
```

```
destruct c ; simpl in *.
refine (aux_law2 aux x0 _ _ _ g).
intros ; destruct a0; firstorder.
Defined.

Program Definition get
m monad (aux : Aux m monad) pre :
HoareStateT string aux
  pre {s : string | pre s}
  (fun i x f ⇒ i = f ∧ proj1_sig x = i).

Program Definition DChoice :
∀ A pre1 pre2 post1 post2
(p1 : Parser pre1 A post1)
(p2 : Parser pre2 A post2) ,
Parser (fun i ⇒ pre1 i ∧ pre2 i) A
       (fun i a f ⇒ post1 i a f ∨ post2 i a f).

Notation "A +++ B" := (DChoice A B).
```

do [4], [13] は，Hoare logic でいう帰結規則（consequence rule）に対応している．ただし，計算的には何もしない，すなわち恒等射となっている．do を用いると，事前条件と事後条件に関する強弱の証明に帰着させることができるので，**Program** の行う自動証明を手助けできる．実際，HoareStateT 上では do の定義に aux_law2 が必要で，本質的にモナド $\mathbf{m}$ に依存する．逆にいえば，do を使わない場合は aux_law2 を使わなければ証明できないゴールが生成されてしまうということである．

get は，StateT における同名関数とまったく同じ役割を果たすが，事前条件をパラメータ化するという一般化を行っている．これによって，Many の定義途中で出てくる中間的な文字列に関する性質 length inp' < length inp を記述できるようになる．

+++は，選択演算子のことである．

5.4　通常のモナドによる定義と比較

前節では，提案手法のもとでどのようにパーザを定義すればよいかについて述べた．一方，本節では通常のモナド，すなわち StateT を用いた従来の定義で何が起こるかを述べ，提案手法と比較する．

まず，StateT のもとで item を定義すると，**図 8** のようになるだろう．次に，item_is_consuming という定理の形で，item が必ず文字列を消費することを示さなくてはならない．この証明の出だしは，**図 9** のようになる．はじめに unfold を行い，続けて入力文字列 i に対する場合分け（destruct）をし，証明全体を進めていく．難しいものではないが，一手間要するために面倒ではあるだろう．逆

```
Program Definition item : Parser char :=
fun cs ⇒
match cs with
| [] ⇒ []
| (c :: cs) ⇒ [(c,cs)]
end.

Definition consume a (p : Parser a) :=
∀ (i : string) ,
match p i with
| [] ⇒ True
| l  ⇒ ∀ m , In m l → length (snd m) < length i
end.

Theorem item_is_consuming : consume item.
Proof with (simpl in * ; auto || (try omega)).
```

図 8　StateT を用いた item の定義
Fig. 8　Definition of item using StateT.

にいえば，Hoare state monad transformer を用いた新しい Parser 型による定義の方が無駄が少ないともいえる．

面倒という点では，状況はより悪くなる．たとえば，item を用いて以下のようなパーザ sat を定義したとする．パーザ sat は，入力文字列の先頭文字が条件 pred を満たすならば，それを消費するようなパーザである．ただし，先頭文字が条件を満たさなかった場合は失敗する．

```
Definition sat (pred : char → bool) :
Parser char :=
c ← item ;
match pred c with
| true ⇒ Ret c
| false ⇒ Fail
end.

Theorem sat_is_consuming : ∀ p , consume (sat p).
```

sat は，間違いなく consume を満たす．しかし，sat_is_consuming を証明しようとすると**図 10** のようになる．StateT の Bind は，リストモナドの bind を用いて定義されているので，素直に証明をしようと思うとこのようになってしまう．

一方，提案手法では，

```
Program Definition sat (pred : char → bool) :
Parser Top char (fun i x f ⇒ length f < length i
∧ pred x = true) :=
do _ _ (
c ← item ;
match pred c with
| true ⇒ gen_ret c (fun c ⇒ pred c = true) _
```

```
item_is_consuming < unfold consume ; unfold item.
1 subgoal
  ============================
   ∀ i : string,
   match
     match i as cs return (cs = i →list (char * string)) with
     | nil ⇒fun _ : [] = i ⇒[]
     | c :: cs ⇒fun _ : c :: cs = i ⇒[(c, cs)]
     end eq_refl
   with
   | nil ⇒True
   | p :: l0 ⇒
       ∀ m : char * string, In m (p :: l0) →length (snd m) < length i
   end
item_is_consuming < intro i ; destruct i.
...
```

図 9　item_is_consuming の証明
Fig. 9　Proof of item_is_consuming.

```
sat_is_consuming < unfold consume ; unfold sat.
1 subgoal

  p : char →bool
  ============================
   ∀ i : string,
   match (c ←item; (if p c then Ret c else Fail)) i with
    ...
sat_is_consuming < unfold Bind ; unfold bind.
1 subgoal

  p : char →bool
  ============================
   ∀ i : string,
   match
     (let (returns0, bind0, _, _, _) := list_is_monad in bind0)
       (char * string) (char * string) (item i)
       (fun v : char * string ⇒
        (let (v0, s2) as v0 return (v0 = v →list (char * string)) := v in
         fun _ : (v0, s2) = v ⇒(if p v0 then Ret v0 else Fail) s2) eq_refl)
   with
    ...
```

図 10　sat_is_consuming の証明
Fig. 10　Proof of sat_is_consuming.

```
  | false ⇒ Fail
  end).
```

として定義できる．

最後に，従来どおりの StateT を用いた Parser 型での many コンビネータの定義を試みる．

図 7 をもとに書き直したとすると，次のようなゴールを解かなければならない．しかし，many の引数となるパーザが入力文字列を消費できたことが表現できないので，これを示すことができない．

```
  ...
  inp : string , inp' : string
  many : ∀(a : Type) (p : Parser a),
         consume p →
         ∀ inp0 : string, length inp0 < length inp →
Parser (list a)
  v : a
```

```
============================
 length inp' < length inp
```

一方，次のようにCoqの証明モードで，定義したい項に対応するように証明を書くといった方法がある．

```
Definition many a (p : Parser a) (wit : consume p)
: Parser (list a).
intro ; revert H.
apply
  (well_founded_induction_type (well_founded_ltof _ length)).
 intros ; case_eq (p x).
...
```

ただし，証明モードによる定義は込み入ったものになりがちで，手間がかかる．何よりHoare state monad transformerを用いて定義した素直なものに比べて可読性に劣る．さらに，manyを定義したうえでこれが必ず文字を消費するパーザとなっているか否かを証明しなくてはならない．同様の手間はitemやsatを定義する際にも発生したが，Hoare state monad transformerを用いた場合はmanyを定義するだけで事後条件をもっていうことができる．

5.5 実例

本節では実例として，文献[3]で例として用いられている以下の文法[*1]に対するパーザを実装する．

```
expr ::= expr addop term | term
term ::= term mulop factor | factor
factor ::= digit | ( expr )
digit ::= 0 | 1 | ... | 9
addop ::= + | -
mulop ::= *
```

左再帰な文法は，再帰下降パージングでは扱うことができないので，まず右再帰形に変形する．

```
expr ::= term (addop term)*
term ::= factor (mulop factor)*
```

この文法に対応するパーザは，Haskellでは次のように定義される．

```
expr = chainl1 term addop
term = chainl1 factor mulop
factor = digit +++
         do {symb "(" ; n ← expr ; symb ")";
             return n}
digit = do {x ← token (sat isDigit);
            return (ord x − ord '0')}
addop = do {symb "+" ; return (+)} +++
        do {symb "−"; return (−)}
mulop = do {symb "∗" ; return (∗)}
```

Coqではこれをもとに，どのように定義することができるだろうか．特に，停止性を念頭に置く必要がある．そこで，manyを定義するときに用いた補助関数Manyで行ったように，文字列を明示的に引数にとるようにし，停止性証明のために用いたい．したがって，次のようなgrammar関数が定義できればよい．

```
Program Fixpoint grammar (nt : NT) (inp : string) :
Parser
 (fun i ⇒ length i <= length inp) (nt_type nt)
 (fun i x f ⇒ length f < length i)
```

ただし，非終端記号を各々実装しようとするとただちに問題に直面する．たとえば，factorの場合を次のように実装したとする．

```
match nt with
| factor ⇒ (grammar digit inp) +++
  (p_char '(' ;
   inp' ← get (fun inp' ⇒ length inp' <
length inp) ;
   e ← grammar expr inp' ;
   p_char ')' ;
   Ret e)
```

ここでp_charは与えられた文字を，入力文字列の先頭から消費するパーザであるとする．したがって，+++の右項でのgrammarの再帰呼び出しについては，引数inp'がinpよりも短くなっているので問題ない．一方，左項はどうだろうか．このとき，引数が短くなっていることはいえない．これより，入力文字列だけを停止性のよりどころとするには不十分であることが分かる．

そこで，grammarのもう1つの引数である非終端記号に関係を入れることにする．ただし，停止性証明のよりどころとする関係であるから，整礎性を満たさなければならない．かつ，直感的なものを導入したい．そこで，再帰下降パージングをすると仮定したときに，非終端記号Nの定義において文字列をいっさい消費せずに到達できる非終端記号N'の間に$N' < N$となるように関係を入れることとする．

このように関係を入れたときに，整礎性が成り立たないならば，文法が（間接）左再帰的になっていることは明らかである．逆に，整礎性が成り立つならば，grammarの定義は入力文字列が減少するときはそれを用いて，入力文字列が減少しない場合は非終端記号の関係を用いるようにすると定義できる．すなわち，入力文字列の減少性と，非終端記号の関係の辞書式関係（**図11**）を用いる．また，これ

[*1] 元の文献[3]では，文法に除算'/'を含むが，Coqでの整数除算は単純ではないので省いている．

```
Inductive NT : Set := expr | term | factor | digit : NT.

Definition NT_lt : ∀ (x:string) (a b : NT) , Prop := (* a < b *)
fun _ a b ⇒
match (a , b) with
| (term , expr) | (factor , term) | (digit , factor) ⇒ True
| (_ , _) ⇒ False
end.

Definition strlen_lt (a b : string) : Prop := length a < length b.

Theorem well_founded_NT_lt : ∀ x : string , well_founded (NT_lt x).
Theorem well_founded_strlen : well_founded strlen_lt.

Definition lexico_R := lexprod _ _ strlen_lt NT_lt.
Theorem well_founded_lexico_R : well_founded lexico_R.
```

図 11　grammar のための辞書式関係
Fig. 11　Lexicographic relation for grammar.

```
Program Fixpoint grammar (nt : NT) (inp : string)
{measure (existT _ inp nt) (lexico_R)} :
Parser (fun i ⇒ length i <= length inp)
       (nt_type nt)
       (fun i x f ⇒ length f < length i) :=
match nt with
| expr ⇒ do _ _ (chainl1 _ inp (@grammar term inp _) addop)
| term ⇒ do _ _ (chainl1 _ inp (@grammar factor inp _) mulop)
| factor ⇒ do _ _ (
(@grammar digit inp _) +++
(_ ← p_char "(" ;
inp' ← get (fun inp' ⇒ length inp' < length inp) ;
n ← @grammar expr inp' _;
_ ← p_char ")" ;
Ret n)
)
| digit ⇒ do _ _ (p_digit)
end.
```

図 12　grammar の定義
Fig. 12　Definition of grammar.

らをふまえた grammar の実装が，図 **12** となる．

6. 関連研究

Coq によるパーザの話では，PEG インタプリタの実装について述べた TRX [14] がある．インタプリタのセマンティクスは再帰下降的な自然なものであり，その対象とする文法は well-formed [15] なものに限るといった形をとっている．

本論文の手法がパーザコンビネータという DSL の浅い埋め込みであるとすると，TRX は PEG を DSL とした深い埋め込みである．本手法は浅い埋め込みであるので，Coq の従来のプログラムとほぼ同じように DSL を書くことができるが，全体にわたって証明を付けなければならない．一方，深い埋め込みである TRX は Coq の中で，PEG を用いて文法を定義しなければならない（ただしセマンティックアクションは自然に Coq の関数で書ける）が，DSL そのものに対する証明を付けているので，ユーザが，たとえば停止性に関する証明を付ける必要はない．

埋め込みの違いによる長短はあるのだが，たとえば chainl1 コンビネータのようなものを新たに TRX 上に加えようとした場合，定義や証明を大幅に変更しなくてはならない．このような変更に対する身軽さは，浅い埋め込みによく見られるものである．

その他の定理証明系によるパーザの話では，Agda [16] を

用いた文献 [17], [18] がある．

文献 [17] では，左再帰的なパーザが定義できるライブラリを実装している．左再帰的なパーザは，本論文でのアプローチでは直接書くことはできない．実際，5.5 節でも左再帰的な文法を右再帰的な文法に変換した後，chainl1 を用いて演算子の結合則を保つように定義を考えたうえで実装を行った．これがパーザを定義するうえでの最も大きな違いだろう．

実装は，Agda の機能である *Mixed Induction and Coinduction* [19], [20] を巧みに用いた形になっている．基本的なパーザコンビネータを Parser 型の構成子として定義し，構成子の解釈が帰納的になるように定義をしている．ただし，*Mixed Induction and Coinduction* は Coq に組み込まれている機能ではないため，現状の Coq では容易に，文献 [17] の手法を摸倣することはできない．

一方，本論文では Coq の余帰納的な機能はいっさい使わずに，Hoare state monad transformer および **Program** 機能を用いて従来どおりのパーザコンビネータの定義ができるようになっている．前述のとおり，浅い埋め込みを用いたのでパーザを定義するごとに証明を書かなければならない．それでも，各々の証明は比較的少量で済む．しかし，Agda では現状，Hoare state monad transformer を用いるだけでは多量の証明を行わなければならない．この意味で，本論文では **Program** という Coq の機能をうまく用いたといえる．文献 [17] と本論文では，各々の定理証明系の長所を最大限に利用したところが共通しているといえるだろう．しかし，これが 2 つのアプローチを大きく分けるものになっているともいえる．

文献 [18] では帰納的な手法と余帰納的な手法を並行させ（ただし *mixed* ではない），それらを合わせる手法になっている．

Hoare State の考え方は特別新しいものではない．すでに述べたとおり古くは Hoare logic まで遡るが，それよりは，文献 [4] でも書かれていることだが，Hoare Type Theory [13], [21], [22] を参考にしているといえる．

7. 問題点と今後の課題

Coq において State monad transformer を用いたある程度の規模のプログラムを書く際には，本論文の手法を用いることができると思う．ただし，Coq 上での軽量な実装にすぎないので，単なる Hoare state monad としてではなく，より詳しく調べられ，正確に形式化されたライブラリや実装があれば（Coq でいえば Ynot [13] などがある），そちらを使った方がよいだろう．

さて，本論文の実装にはかかわらなかったものの，未解決の問題点がいくつかある．

7.1 すべての Monad は Aux を満たすか？

本論文では，特にリストモナドに対する定義を述べたが，代表的なモナドである Identity モナド，Maybe モナド，Either モナド，Writer モナド，Reader モナドは Aux を満たすような定義ができる．では Continuation（継続）モナドはどうだろうか．

Continuation モナドそのものは次のような定義になっている．

```
newtype Cont r a = Cont {
    runCont :: (a → r) → r
}
instance Monad (Cont r) where
    return a = Cont (\k → k a)
    m >>= k  = Cont (\c → runCont m (\a →
runCont (k a) c))
```

このとき，get_and_pr はどのように定義すればよいのだろうか？すなわち Cont r に対して

```
Definition cont_get_and_pr r : ∀ A , Cont r A →
(A → Prop) → Prop.
```

この関数を定義しなければならない．型を見ると，継続計算の最終的な型が r として固定されていることが問題となっている．

また，リストモナドをはじめとする，Aux を満たすように定義ができるモナドについても**どのように定義するべきか**ということについてはあまり考えなかった．今後は Aux を満たすのに，モナドであることに加えて何が必要なのかということ，そしてモナドの定義に基づいて，どのように定義されるべきなのかを調べたい．さらにいえば，Aux より良い要求があるのか，そもそも HoareState, HoareStateT の定義を改良もしくは使いやすい形に一般化できないかも調べたい．

7.2 非終端記号間の関係

本論文での実装にあたっては，非終端記号間の関係を手書きした．この関係が実際に整礎であるならば，次のようにして証明を自動化することができる．

```
Theorem well_founded_NT_lt : ∀ x : string ,
well_founded (NT_lt x).
intros.
repeat (constructor; intros; destruct y; firstorder).
Qed.
```

しかし肝心の関係そのものの導入を（半）自動的に行うことはできないであろうか？実際のサイズの文法を考えると，当然文法からの導出ができた方がよいだろう．

8. おわりに

本論文では，Coq によって total parser combinator library を monadic に実装した．その実装のために，Hoare state monad transformer という，Hoare state monad のモナド変換子版と，その定義を提案した．

Haskell のモナドをそのまま用いた monadic な実装では，Coq での証明と相性が悪いことは述べたとおりである．一方，Hoare state monad と，Hoare state monad transformer を用いれば，monadic な定義はそのままに，諸性質を型で表現することができるので，証明がしやすい．

また，本論文は，Haskell による monadic なパーザコンビネータの実装を述べた文献 [3] を，Coq で実装し直したというようにも見ることができる．もとの Haskell による実装は，当然停止性を考慮するものではないが，とにかくエレガントで単純な実装である．このような参照実装を，できるだけ（プログラムとして）保ったままに，定理証明系で実装しなおすための良い技法というのは少なく，なおかつ広く知られていないと考える．

このような状況で，Hoare logic を媒介に State monad を一般化した Hoare state monad を提案し，さらに Coq の **Program** 機能を用いて，その利用を現実的なものにした Swierstra によるテクニック [4] は，単純でありながらも実用的な，大変優れたものであるといえるだろう．本論文では，この手法がパーザのような実用的なプログラムを定義する際にも十分に用いることができることを明らかにした．

参考文献

[1] Leijen, D. and Meijer, E.: Parsec: Direct Style Monadic Parser Combinators for the Real World, Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht (2001).

[2] INRIA: The Coq Proof Assistant, INRIA (online), available from ⟨http://coq.inria.fr/⟩ (accessed 2011-09-27).

[3] Hutton, G. and Meijer, E.: Monadic Parsing in Haskell, *Journal of Functional Programming*, Vol.8, No.4, pp.437–444 (1998).

[4] Swierstra, W.: A Hoare Logic for the State Monad (Proof Pearl), *Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science, Vol.5674, pp.440–451, Springer-Verlag (2009).

[5] Hoare, C.A.R.: An axiomatic basis for computer programming, *Comm. ACM*, Vol.12, pp.576–580 (1969).

[6] 林　晋：プログラム検証論，情報数学講座 8，共立出版 (1995).

[7] Sozeau, M.: Program-ing finger trees in Coq, *Proc. 12th ACM SIGPLAN international conference on Functional programming*, *ICFP'07*, pp.13–24, ACM (2007).

[8] INRIA: The Coq Proof Assistant (Chapter 22 PROGRAM), INRIA (online), available from ⟨http://coq.inria.fr/refman/Reference-Manual028.html⟩ (accessed 2011-09-27).

[9] Jones, M.P.: Functional Programming with Overloading and Higher-Order Polymorphism, *Advanced Functional Programming, 1st International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pp.97–136, Springer-Verlag (1995).

[10] Benton, N., Hughes, J. and Moggi, E.: Monads and Effects, *Applied Semantics*, Lecture Notes in Computer Science, Vol.2395, pp.923–952, Springer Berlin/Heidelberg (2002).

[11] Moggi, E.: Computational Lambda-Calculus and Monads, *LICS*, pp.14–23 (1989).

[12] Hinze, R.: Monadic-style backtracking, Technical Report IAI-TR-96-9, nstitut fr Informatik III, Universitt Bonn (1996).

[13] Nanevski, A., Morrisett, G., Shinnar, A., Govereau, P. and Birkedal, L.: Ynot: dependent types for imperative programs, *Proc. 13th ACM SIGPLAN international conference on Functional programming*, *ICFP'08*, pp.229–240, ACM (2008).

[14] Koprowski, A. and Binsztok, H.: TRX: A Formally Verified Parser Interpreter, *Logical Methods in Computer Science* (*LMCS*), Vol.7, No.2 (2011).

[15] Ford, B.: Parsing expression grammars: A recognition-based syntactic foundation, *Proc. 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, *POPL'04*, pp.111–122, ACM (2004).

[16] The Agda team: The Agda wiki, (online), available from ⟨http://wiki.portal.chalmers.se/agda/pmwiki.php⟩ (accessed 2011-09-27).

[17] Danielsson, N.A.: Total parser combinators, *Proc. 15th ACM SIGPLAN international conference on Functional programming*, *ICFP'10*, pp.285–296, ACM (2010).

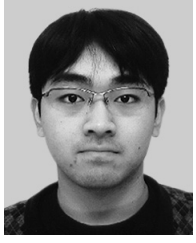
[18] Danielsson, N.A. and Norell, U.: Structurally Recursive Descent Parsing, (online), available from ⟨http://www.cse.chalmers.se/~nad/publications/danielsson-norell-parser-combinators.html⟩ (2008).

[19] Danielsson, N. and Altenkirch, T.: Subtyping, Declaratively, *Mathematics of Program Construction*, Lecture Notes in Computer Science, Vol.6120, pp.100–118, Springer Berlin/Heidelberg (2010).

[20] Danielsson, N.A.: Beating the Productivity Checker Using Embedded Languages, *PAR*, pp.29–48, EPTCS (2010).

[21] Nanevski, A., Morrisett, G. and Birkedal, L.: Polymorphism and separation in hoare type theory, *Proc. 11th ACM SIGPLAN international conference on Functional programming*, *ICFP'06*, pp.62–73, ACM (2006).

[22] Nanevski, A. and Morrisett, G.: Dependent Type Theory of Stateful Higher-Order Functions, Technical Report TR-24-05, Harvard University, Cambridge, MA, USA (2005).



上里 友弥

平成元年生．平成 20 年筑波大学情報学群情報科学類入学．