q-類似の Coq による形式化

中村薫　名古屋大学

Coqgen project とは

- OCaml の型推論の正しさを Coq で確認することを目標とするプロジェクト.
- 具体的には OCaml から Coq へのコンパイラの作成, およびコンパイル後のプログラムに関する証明を行っている.
- 以下, Coqgen project で得られた成果の一部をまとめ, 説明する.

モナド

- OCaml における store や例外を Coq でも表現することができるようにモナドを利用する.
- store を Env, 例外を Exn で表し, これらをまとめて次のようにモナドを定義する.

```
Definition M T := Env → Env * (T + Exn).
```

Type translation

OCaml の型はデータ型として定義されていて, Coq に翻訳される.

```
Fixpoint coq_type (T : ml_type) : Type := ...
```

今までの Coqgen

- core ML（λ計算 + 多相型, 再帰関数, データ型）
- 参照型, 例外
- 変換の例

```
let rec fact_rec n =
  if n ≤ 1 then 1 else n * fact_rec (n - 1)

Fixpoint fact_rec (h : nat) (n : coq_type ml_int) : M (coq_type ml_int) :=
  if h is h.+1 then (* 停止性のためにガスを減らす *)
    do v <- ml_le h ml_int n 1%int63;
    if v then Ret 1%int63 else
      do v <- fact_rec h (Int63.sub n 1%int63); Ret (Int63.mul n v)
  else FailGas.
```

while loop

- OCaml のプログラム

```
let fact n =
  let i = ref n in let v = ref 1 in
  while !i > 0 do v := !v * !i; i := !i - 1 done;  !v
```

- Coqgen のライブラリ

```
Fixpoint whileloop (h : nat) (f : M bool) (b : M unit) : M unit :=
  if h is h.+1 then
    do v <- f; if v then (do _ <- b; whileloop h f b) else Ret tt
  else FailGas.
```

- コンパイル結果

```
Definition fact (h : nat) (n : coq_type ml_int) : M (coq_type ml_int) :=
  do i <- newref ml_int n;
  do v <- newref ml_int 1%int63;
  do _ <-
  whileloop h (do v_1 <- getref ml_int i; ml_gt h ml_int v_1 0%int63)
    (do _ <-
    (do v_1 <-
      (do v_1 <- getref ml_int i;
      do v_2 <- getref ml_int v; Ret (Int63.mul v_2 v_1));
      setref ml_int v v_1);
    do v_1 <- (do v_1 <- getref ml_int i; Ret (Int63.sub v_1 1%int63));
    setref ml_int i v_1);
  getref ml_int v.
```

lazy

- OCaml のプログラム

```
let lazy_id = lazy (fun x → x) (* 多相型  (`a → `a)lazy_t *)

let lazy_next c = lazy (incr c; !c)

let c = ref 0 in let m = lazy_next c in let n = lazy_next c in
let n = Lazy.force n in let m = Lazy.force m in [m; n]
```

- Coqgen のライブラリ

```
Inductive lazy_val (a : Type) :=
  LzVal of a | LzThunk of M a | LzExn of ml_exns.
Inductive lazy_t a a1 := Lval of a | Lref of (loc (ml_lazy_val a1)).

Definition force a (lz : coq_type (ml_lazy a)) : M (coq_type a) := ...
Definition make_lazy a (b : M (coq_type a)) : M (coq_type (ml_lazy a)) :=
  do x <- newref (ml_lazy_val a) (LzThunk _ b); Ret (Lref _ _ x).
Definition make_lazy_val a (b : coq_type a) : coq_type (ml_lazy a) :=
  Lval (coq_type a) a b.
```

- コンパイル結果

```
Definition lazy_id (T : ml_type) : coq_type (ml_lazy (ml_arrow T T)) :=
  make_lazy_val (ml_arrow T T) (fun x : coq_type T ⇒ Ret x).
```

make_lazy_val を使うことで多相型にできる.

```
Definition lazy_next (c : coq_type (ml_ref ml_int))
  : M (coq_type (ml_lazy ml_int)) :=
     make_lazy ml_int (do _ <- incr c; getref ml_int c).

Definition it_2 := Eval compute in
  Restart it_1
    (do c <- newref ml_int 0%int63;
     do m <- lazy_next c; do n <- lazy_next c;
     do n_1 <- force ml_int n; do m_1 <- force ml_int m;
     Ret (m_1 :: n_1 :: @nil (coq_type ml_int))).
Print it_2.
it_2 = (..., inl [:: 2%sint63; 1%sint63]) : Env * (seq int + Exn)
```

force されて初めて計算が実行されるという lazy の性質が表現できている.

insertion sort

- OCaml のプログラム

```
let rec insert a l =
  match l with
  | [] → [a]
  | b :: l' → if a ≤ b then a :: l else b :: insert a l'
let rec isort l =
  match l with
  | [] → []
  | a :: l' → insert a (isort l')
```

- コンパイル結果

```
Fixpoint insert (h : nat) (T_1 : ml_type) (a : coq_type T_1)
  (l : coq_type (ml_list T_1)) : M (coq_type (ml_list T_1)) :=
  if h is h.+1 then
    match l with
    | @nil _ ⇒ Ret (a :: @nil (coq_type T_1))
    | b :: l' ⇒
      do v <- ml_le h T_1 a b;
      if v then Ret (a :: l) else
        do v <- insert h T_1 a l'; Ret (@cons (coq_type T_1) b v)
    end
  else FailGas.
```

insertion sort 続

```
Fixpoint isort (h : nat) (T_1 : ml_type) (l_1 : coq_type (ml_list T_1))
  : M (coq_type (ml_list T_1)) :=
  if h is h.+1 then
    match l_1 with
    | @nil _ ⇒ Ret (@nil (coq_type T_1))
    | a :: l' ⇒ do v <- isort h T_1 l'; insert h T_1 a v
    end
  else FailGas.
```

insertion sort の証明

整列している状態 sorted の定義

```
Fixpoint sorted l :=
  if l is a :: l' then all (le a) l' && sorted l' else true.
```

示したいことは, isort で作られた列が上の意味で整列していることである.

```
Theorem isort_ok h l l' : isort h ml_int l = Ret l' → sorted l'.
Proof.
  elim: h l l' ⇒ [_ _ |h IH [|a l] l'] /(happly empty_env) //=.
  - by move⇒ [] <-.
  - destruct h ⇒ //.
    case: (isort_pure h.+1 l) ⇒ -[l'' H].
    + rewrite H bindretf.
      case: (insert_pure h.+1 a l'') ⇒ -[l0 H'].
      - rewrite H' ⇒ -[] <-.
        by apply /(insert_ok H') /(IH l l'').
      - by rewrite H'.
    + by rewrite H bindfailf.
Qed.
```

補題の先頭の isort h ml_int l = Ret l' は「計算が成功すれば」という条件である.

証明中に使われている補題は以下のとおりである.

```
Lemma happly [A B] [f g : A → B] x : f = g → f x = g x.
Lemma bindretf A B (a : A) (f : A → M B) : Ret a >>= f = f a.
Lemma bindfailf A B e (g : A → M B) : @Fail A e >>= g = @Fail B e.

Definition pure [T] (m : M T) :=
  (∃ r, m = Ret r) \/ (∃ e, m = Fail e).
Lemma insert_pure h b l : pure (insert h ml_int b l).
Lemma isort_pure h l : pure (isort h ml_int l).
Lemma insert_ok h a l l' : insert h ml_int a l = Ret l' →
  sorted l → sorted l'.
```

直接 Coq に書く場合との違い

コンパイルされたプログラムはモナドに覆われているため, 一般に証明は煩雑になる. しかし, 上のような bindretf や insert_pure などのモナドに関する補題を用意しておくことで, 直接 Coq に書いた場合の証明に近づけることができる.

例えば, isort_ok であれば, 直接書くと

```
Theorem isort_ok l : sorted (isort l).
Proof. elim: l ⇒ //= a l IH; by rewrite insert_ok. Qed.
```

となるが, モナドに関する部分を除けば本質が同じであることが見て取れる. 今後はさらに近づける方法を考えたい.

Coqgen に関する情報

https://github.com/COCTI/ocaml/pull/3

本研究はTezos財団及びJSPS科研費JP22K11902の助成を受けたものです.