Verificación Formal
Primer Proyecto

Ciro Iván García López

Octubre 2020

Resumen

En el presente documento se exponen los avances del primer proyecto del curso de verificación formal. El objetivo de este proyecto es mecanizar el articulo de Bas van den Heuvel y Jorge Pérez Session Type Systems based on Linear Logic en Coq.

Introducción

Heuvel y Pérez [vdHP20] introducen el sistema de tipos π-ULL para el cálculo π [SW03], este sistema busca establecer una relación Curry-Howard entre un fragmento de la lógica lineal y el cálculo π. La relación Curry-Howard entre estos sistemas también ha sido estudiada por Pfenning, Wadler y otros, e incluso existen trabajos que buscan abordar el problema de mecanizar dicha relación por medio de sistemas de verificación [CFY20, Zal19]. No obstante, como es expuesto por Castro et. al. [CFY20] verificar formalmente dichas relaciones cobra importancia dado que las pruebas involucradas son tediosas y en muchos casos omitidas.

Abordar la tarea de mecanizar este tipo de teorías involucra adicionalmente el manejo de abstracciones, variables ligadas y α-equivalencias; por lo general trabajar con estos tres conceptos no es una labor simple ya que no es sencillo trabajar sobre clases de equivalencia en los sistemas de verificación formal. Los indices de De Bruijn constituyen una primera aproximación a la mecanización de variables, marco de trabajo en el cual todas las variables obtienen un índice numérico y se manipulan usando operaciones de corrimiento [Cha12]. Sin embargo no son una solución óptima, debido a que razonar sobre los términos usando las funciones de corrimiento se vuelve una labor bastante compleja y propensa a errores.

Por otro lado, inspirado en los índices de Bruijn Charguéraud [Cha12] introduce la *representación local libre de nombres* una segunda aproximación en la mecanización de variables. En este marco de trabajo se hace distinción entre variable libre y ligada, las primeras representadas mediante nombres y las segundas mediante índices numéricos. Esta idea elimina la noción de α-equivalencia en variables ligadas ya que los términos representan de manera única. El precio a pagar por la simplicidad de la representación local libre de nombres es el incremento en las reglas de la gramática, por ende de términos *basura*, y de proposiciones auxiliares en el sistema para el manejo de los términos.

1. ULL en Coq

En el artículo de Huevel y Pérez la primera definición corresponde es la noción de proposición ULL (πULL), Definición 2.1 [vdHP20, p. 2].

Definición 1. Las proposiciones de ULL son generados por la siguiente gramática:

$$A, B := 1 \mid \bot \mid A \otimes B \mid A \parr B \mid A \multimap B \mid !A \mid ?A \tag{1}$$

Una explicación detallada de cada operador de la definición anterior se encuentra en la Tabla 1 [vdHP20, p. 3]. En Coq la definición de una proposición es:

```
Inductive Proposition : Type :=
  | ONE : Proposition
  | ABS : Proposition
  | TEN (A : Proposition) (B : Proposition) : Proposition
  | PAR (A : Proposition) (B : Proposition) : Proposition
  | EXP (A : Proposition) : Proposition
  | MOD (A : Proposition) : Proposition.
Hint Constructors Proposition : core.

Notation "\P" := ONE.
Notation "⊥" := ABS.
Notation "A ⊗ B" := (TEN A B)(at level 70, right associativity).
Notation "A ⅋ B" := (PAR A B)(at level 70, right associativity).
(* Notation "A ⊸ B" := (ULLT_IMP A B)(at level 50, left associativity). *)
Notation "! A" := (EXP A)(at level 60, right associativity).
Notation "? A" := (MOD A)(at level 60, right associativity).
```

Observe que en la definición no se incluye el operador $\multimap$, para éste operador se utiliza la observación la anotación de la Definición 2.2 [vdHP20, p. 3] del artículo y se define el operador de la siguiente manera:

```
Definition ULLT_IMP (A : Proposition) (B : Proposition) : Proposition := (A^⊥) ⅋ B.
```

En donde $A^{\perp}$ representa la noción de dual, Definición 2.2 [vdHP20, p. 3].

```
Fixpoint Dual_Prop ( T : Proposition ) : Proposition
```

Adicional a las definiciones sobre proposiciones se prueban algunos de los resultados mencionados, tal vez la propiedad más importante es la relacionada con el doble dual.

```
Proposition Doble_Duality_ULLT :
forall A : Proposition,
(A^⊥)^⊥ = A.
Proof.
...
Qed.
```

2. Procesos

La etapa de la mecanización corresponde con implementar la Definición 2.3 [vdHP20, p. 3] sobre procesos.

Definición 2. Un proceso está generado por la siguiente gramática:

$$P := \mathbf{0} \mid [x \leftrightarrow y] \mid (\nu y)P \mid P|Q \mid x\langle y\rangle.P \mid x(y).P \mid !x(y).P \mid x\langle\rangle.\mathbf{0} \mid x().P \tag{2}$$

No obstante al usar la definición surge la problemática de mecanizar las variables libres y ligadas, labor que no es trivial y ha sido estudiada con anterioridad. Existen dos maneras canónicas de resolver este problema, la primera basada en los índices de Bruijn y la segunda la representación local libre de nombres.

En principio gracias a las observaciones dadas por Chargeroud [Cha12] sobre las funciones de cambio (shift) de índices bajo en el marco de trabajo de Bruijn y algunas de las tareas asignadas en el curso, se decide utilizar la representación libre de nombres para la mecanización en Coq propuesta.

2.1. Representación Local Libre de Nombres

La idea es introducida por Chargeroud [Cha12]. En este marco de trabajo se hace una distinción entre las variables libres y ligadas; para las primeras se siguen representando por cadenas, para las segundas se remplazan los nombres por índices numéricos que van a indicar su posición relativa respecto su operador cerradura. Para ejemplificar la representación considere el siguiente ejemplo.

Ejemplo 1. Sea $\lambda x.(x\lambda y.xyz)$ un término del cálculo λ, su representación libre de nombres está dada por: $\lambda,0(\lambda.(1)(0)z)$.

Observe que los acopladores (Binder) aparecen, sin variable, al cambiar a la representación libre de nombres. De acuerdo a Sangiorgi [SW03] en $(\nu y)P$, $x(y).P$, $!x(y).P$ el nombre y está ligado, por lo que se diseña una representación libre de nombres para los procesos siguiendo las ideas expuestas por Charguéraud [Cha12] y de esta manera se obtiene la siguiente gramática modificada:

$$N := FName(x) \mid BName(i)$$
$$P := \mathbf{0} \mid [N \leftrightarrow N] \mid \nu P \mid P|Q \mid N\langle N\rangle.P \mid N.P \mid !N.P \mid N\langle\rangle.\mathbf{0} \mid N().P$$

Siendo N un nombre libre o ligado, x es una cadena e i un índice entero. En Coq la gramática modificada está representada por las definiciones inductivas:

```
Inductive Name : Type

Inductive Prepro : Type
```

Observe que los términos generados por la gramática modificada son llamados preprocesos (*Prepro*), la razón es que esta nueva gramática genera términos que no tienen sentido en la original, por ejemplo $[BVar(0) \leftrightarrow BVar(1)]$. Por lo que entonces es necesario determinar cuáles términos realmente están bien formados, para esto se definen las siguientes reglas (*Process*) que determinan los términos bien formados.

Definición 3. Un *Process* es un término que se construye bajo las siguientes reglas:

$$\frac{\forall\ x}{Process(FVar(x))} \qquad \frac{Process(FVar(x)) \qquad Process(P)}{Process(FVar(x)().P)}$$

$$\frac{}{Process(\mathbf{0})} \qquad \frac{Process(FVar(x)) \qquad Process(FVar(y)) \qquad Process(P)}{Process(FVar(x)\langle FVar(y)\rangle P)}$$

$$\frac{Process(P) \qquad Process(Q)}{Process(P|Q)} \qquad \frac{\exists L\ \forall\ x \notin L \qquad Process(P^x)}{Process(\nu P)}$$

$$\frac{Process(FVar(x)) \qquad Process(FVar(y))}{Process([FVar(x) \leftrightarrow FVar(y)])} \qquad \frac{\exists L\ \forall\ x \notin L \qquad FVar(x) \qquad Process(P^x)}{Process(FVar(x).P)}$$

$$\frac{Process(FVar(x))}{Process(FVar(x)\langle\rangle,0)} \qquad \frac{\exists L\ \forall\ x \notin L \qquad FVar(x) \qquad Process(P^x)}{Process(!FVar(x).P)}$$

Siendo L una lista finita de cadenas, en Coq se materializan mediante las definiciones:

```
Inductive Process_Name : Name → Prop

Inductive Process : Prepro → Prop
```

Observe que en la definición de proceso no se especifico el significado de P^x, ésta es conocida como la función de apertura, cuya tarea es remplazar nombres ligados por nombres abiertos.

```
(* Apertura para los nombres*)
Definition Open_Name ( k : nat )(z N : Name ) : Name :=
match N with
  | FName x ⇒ FName x
  | BName i ⇒ if ( k =? i ) then z else (BName i)
end.

(* Apertura para los preprocesos *)
Fixpoint Open_Rec (k : nat)( z : Name )( T : Prepro ) {struct T} : Prepro
```

De manera dual existe una función de cerradura que remplaza nombres abiertos por cerrados

```
(* Cerradura para los nombres*)
Definition Close_Name ( k : nat )( z N : Name ) : Name :=
match N with
  | FName n0 ⇒ match z with
                 | FName x0 ⇒ if String.eqb n0 x0 then BName k else N
                 | BName i0 ⇒ N
               end
  | BName i ⇒ N
end.

(* Cerradura de preprocesos *)
Fixpoint Close_Rec (k : nat)( z : Name )( T : Prepro ) {struct T} : Prepro
```

Otro conceptos que emerge de la definición de un proceso es el concepto de cuerpo (*Body*), informalmente un cuerpo es un preproceso P para el cual $\nu.P$, $x.P$, $!x.P$ son procesos para todas las cadenas x salvo un número finito L.

```
Inductive Body : Prepro → Prop :=
  is_body : forall (P : Prepro), (forall (x : Name)(L : list Name),
  Process_Name x → ∼ (In x L) → Process ( ({ 0 ∼> x }P) )) → Body(P).
```

Ahora se definen las operaciones de cambio de nombre libre, para efectos de la mecanización se va a suponer que si x se cambia por y, entonces y no aparece en el término, esta es una suposición usual en textos sobre cálculo π.

```
Definition Subst_Name ( x y N : Name ) : Name :=
match N with
  | FName n0 ⇒ match x with
                 | FName x0 ⇒ if String.eqb n0 x0 then y else N
                 | BName i0 ⇒ N
               end
  | BName i ⇒ N
end.

Fixpoint Subst ( x y : Name )( T : Prepro ) {struct T} : Prepro
```

Por último se definen la congruencia estructural (Definición 2.4 [vdHP20, p. 4]) y la reducción de procesos (Definición 2.5 [vdHP20, p. 4]). Una observación importante de implementación la eliminación de algunas reglas que pierden sentido bajo la representación libre de nombres, para algunas otras es necesario adicionar hipotesis de estructura sobre el término, por ejemplo pedir que sea un cuerpo o un proceso.

```
Reserved Notation "R '= = =' S" (at level 60).
Inductive Congruence : Prepro → Prepro → Prop :=
...
| Con_abs_restriction : forall (P Q : Prepro),
       Process P → Body Q → (P ↓ (ν Q)) = = = ν (P ↓ Q)
...

Reserved Notation "R '-- >' S" (at level 60).
Inductive Reduction : Prepro → Prepro → Prop :=
...
| Red_parallel_fuse : forall ( x y : Name) ( P : Prepro),
    Process P → ( (P ↓ [x ↔ y]) —→ (Subst x y P) )
...
```

2.2. Reducción de Procesos

Una consecuencia importante de trabajar con los procesos es el hecho que al operar sobre éstos, en general no se obtiene como resultado un proceso. Aquí es importante estudiar algunas afirmaciones

auxiliares que permitan caracterizar los resultados de las operaciones definidas anteriormente. El primer paso para lograr el objetivo es definir algunos axiomas que relacionan las operaciones, su significado es intuitivo a la luz del supuesto que todas las aperturas, cerraduras o sustituciones se hacen con cadenas diferentes.

```
Axiom Ax_Alpha :
forall (x y: Name)( P : Prepro),
({y \ x} P ) = P.

Axiom Ax_Process_Name :
forall ( x : Name),
Process_Name x.
```

Los dos primeros resultados relevantes se refieren a la cerradura y sustitución. El primer lema nos indica que al aplicar la función de cerradura a un proceso se obtiene un cuerpo, y el segundo nos expresa que hacer un cambio de nombre no afecta sustancialmente la estructura del proceso.

```
Lemma Close_Is_Body:
forall (x : Name)(P : Prepro),
Process_Name x → Process P → Body( Close x P ).

Lemma Subst_Process_Is_Process :
forall ( x y : Name ) (P : Prepro),
Process P → Process_Name x → Process_Name y → Process ( { y \ x } P ).

Lemma Open_Process_Is_Process :
forall (x : Name)(P : Prepro),
Process_Name x → Process P → Process ( {0 ~> x} P ).
```

El último y más importante de los teoremas mecanizados es que la reducción está bien definida para los procesos. En otras palabras, al reducir un proceso se obtiene un proceso.

```
Theorem Congruence_WD :
forall P Q : Prepro,
(P = = = Q) → Process(P) → Process(Q).

Theorem ProcessReduction_WD :
forall P Q : Prepro,
(P —→ Q) → Process(P) → Process(Q).
```

3. Tipos

Siguiendo la exposición hecha por Heuvel y Pérez sobre inferencia de tipos [vdHP20, p. 4], se hace necesario introducir las nociones de asignación y colección de asignaciones.

```
Inductive Assignment : Type

Inductive Assig : Assignment → Prop

Inductive Collect : list Assignment → Prop
```

Posteriormente se definen las reglas de inferencia [vdHP20, p. 5] para el cálculo ULL propuesto. Obsérvese que es necesario modificar algunas de las reglas para que conserven el sentido original, al eliminar las variables ligadas es entonces necesario primero cerrar el término.

```
Reserved Notation "D ';;;' F '!-' P ':::' G" (at level 60).
Inductive Inference : Prepro → list Assignment → list Assignment → list Assignment → Prop
...
  | reccr : forall ( D F G: list Assignment ) ( y x : Name)( A B : Proposition)(P : Prepro ),
    Collect D → Collect F → Collect G → Process_Name x → Process_Name y → Process P →
    ( D ;;;  F !— P :::  ( (cons (x:B) (cons (y:A) nil) ) ++ G ) ) →
```

```
    ( D ;;;  F !— (x . (Close y P)) :::  ((cons (x:(A⅋B) ) nil) ++ G ) )
...
where "D ';;;' F '!-' P ':::' G" := (Inference P D F G).
```

La sección de tipos finaliza con la prueba de la propiedad de solidez (Soundness) para las reglas de inferencia.

```
Theorem Soundness :
forall (P : Prepro)(D F G : list Assignment),
  ( D ;;;  F !— P :::  G ) →
  (forall (Q: Prepro), (P —→ Q) → ( D ;;; F !— Q ::: G )).
Proof.
...
Qed.
```

La prueba de esta afirmación se basa en descartar casi todas las reglas de inferencia salvo las reglas de corte. Para todas las pruebas que no son de corte es posible demostrar que la hipótesis de reducción conduce a un absurdo, dichas pruebas se encuentran como lemas auxiliares.

```
Lemma No_Red_AX4 :
forall (x : Name)(P Q: Prepro),
~( (x . P) —→ Q ).
Proof.
  unfold not.
  intros.
  inversion H.
Qed.
```

Para el caso de las reglas de corte, se procede a analizar el término correspondiente a P para poder deducir las condiciones que debe cumplir y poder establecer las reducciones válidas.

```
Lemma Char_Red_Chanres2 :
forall (P Q : Prepro)(x : Name),
Process P → (ν (Close x P) —→ Q ) →
exists (Q0 : Prepro), ( Q = (ν (Close x Q0)) ∧ P —→ Q0).
Proof.

Qed.
```

No obstante es prudente mencionar que para llevar a cabo estas reducciones fue necesario introducir los siguientes lemas, cuya idea intuitiva es cierta.

```
Lemma Close_Rec_Eq_Subst :
forall (P Q : Prepro)(x y : Name)( i: nat),
Process P → Process Q → Close_Rec i x P = Close_Rec i y Q →
P = {x\y} Q.
Admitted.

Lemma Eq_Change_Var_Close :
forall (P : Prepro)(x y : Name),
Close x P = Close y ({y\x}P).
Admitted.
```

La prueba de estos hechos descansa en poder determinar el nombre de las variables libres de P y saber que la función de cerradura se aplica siempre sobre términos que tienen la variable indicada. Estos dos hechos no han sido sencillos de formalizar en Coq y son un tema de discusión amplia.

4. Conclusiones

El nivel de formalidad con el que se expresan los resultados del cálculo π en la mayoría de artículos y textos [SW03, vdHP20, Hon93] hace de la mecanización una labor extensa, ya que es necesario capturar la esencia de esta baja formalidad y en algunos casos conlleva plantear marcos de trabajo lo suficientemente

amplios. En este sentido, hay muchos resultados pendientes por explorar y quedan bastantes resultados por mecanizar.

La representación libre de nombres brinda beneficios a la hora de trabajar con variables ligadas, es sencillo expresar los términos bajo la representación y sus resultados son naturales. No obstante en la práctica se evidencia que no suprime la necesidad de las α equivalencias, lo cual implica asumir algunos resultados no deseados.

5. Repositorio

El enlace al repositorio del proyecto es:

`https://github.com/cigarcial/VF2020II`

Y contiene los siguientes archivos:

- Defs_Proposition : Definiciones relativas a la noción de proposición.
- Props_Proposition : Pruebas de algunos hechos que involucran los proposición.
- Defs_Process : Definiciones relativas a la noción de proceso.
- Props_Process : Pruebas de algunos hechos que involucran los procesos.
- Defs_Typing : Definiciones relativas al sistema de tipos.
- Props_Typing : Pruebas de algunos hechos que involucran el sistema de tipos.

Referencias

[CFY20] David Castro, Francisco Ferreira, and Nobuko Yoshida. Emtst: Engineering the meta-theory of session types. In Armin Biere and David Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 278–285. Springer International Publishing, 2020.

[Cha12] Arthur Charguéraud. The locally nameless representation. *Journal of Automated Reasoning - JAR*, 49:1–46, 10 2012.

[CPT12] Luís Caires, Frank Pfenning, and Bernardo Toninho. Towards concurrent type theory. *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, 01 2012.

[Gir93] Jean-Yves Girard. Linear logic: A survey. In Friedrich L. Bauer, Wilfried Brauer, and Helmut Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 63–112, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[Hon93] Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR'93*, pages 509–523, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[SW03] D. Sangiorgi and D. Walker. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2003.

[vdHP20] Bas van den Heuvel and Jorge A. Pérez. Session type systems based on linear logic: Classical versus intuitionistic. *Electronic Proceedings in Theoretical Computer Science*, 314:1–11, Apr 2020.

[Zal19] Uma Zalakain. Type-checking session-typed π-calculus with coq. *University of Glasgow*, 2019.