Lenguajes y Compiladores: Laboratorio opcional en Coq

Objetivos: Introducirse en la mecanización de resultados matemáticos utilizando asistentes de prueba basados en tipos dependientes, particularmente formalizando los teoremas de Coincidencia y Substitución para expresiones enteras y booleanas.

1. Instalación

Para la formalización utilizamos el compilador de Coq 8.9.0 junto con la librería de matemática SSreflect. Además, si bien hay diferentes opciones, usamos Emacs junto con ProofGeneral y Company-coq como entorno de desarrollo.

1. Instalar emacs > 25.
2. Instalar Coq-8.9.0' `https://coq.inria.fr/news/coq-890-is-out.html`.
3. Instalar SSreflect 1.6.1: `http://math-comp.github.io/math-comp`.
4. Instalar ProofGeneral: `https://proofgeneral.github.io/download`.
5. Instalar Company-coq: `https://github.com/cpitclaudel/company-coq`.

2. Tipos dependientes

Para empezar a programar con tipos dependientes está bueno antes entender un poco de la teoría que implementan este tipo de lenguajes con el objetivo particular de "convencerse" sobre la siguiente correspondencia conocida como el isomorfismo de Curry-Howard que relaciona fórmulas lógicas con tipos.

Lógica	Haskell[1]	Coq
$true$	*data* `True = I`	*Inductive* `True : Prop := I : True`
$false$	*data* `False`	*Inductive* `False : Prop :=` .
$A \wedge B$	$(A\,,\,B)$	$A \,/\backslash\, B$
$A \vee B$	`Either` $A\,B$	$A \,\backslash/\, B$
$A \Rightarrow B$	A `->` B	A `->` B
$\forall(a \in A),\, P(a)$	———	$forall\,(a : A),\, P\,a$
$\exists(a \in A),\, P(a)$	———	$exists\,(a : A),\, P\,a$

Este isomorfismo además nos habla de una equivalencia entre pruebas y programas. Por ejemplo, si recordamos la regla de la conjunción esta nos define que dadas pruebas de A y B entonces tenemos una prueba de $A \wedge B$:

$$\frac{A \qquad B}{A \wedge B}$$

[1] Haskell no posee de un sistema de tipos dependientes.

Pensando en programas es equivalente a decir que dado dos programas $(a :: A)$ y $(b :: B)$, es decir un programa a cuyo tipo es A y un programa b cuyo tipo es B entonces podemos construir el programa (a, b) cuyo tipo será (A, B) (todo esto programando en Haskell). Tomando un ejemplo más concreto podemos dar una prueba de $\frac{true \quad true}{true \wedge true}$ donde su programa equivalente en Haskell será `(I,I)` :: `(True, True)` dado que `I` :: `True`. Notar que no deberíamos nunca poder dar una prueba de $false$, por lo tanto es esperable que no podamos dar un programa cuyo tipo sea `False`; razón por la cual la definición del tipo no tiene constructores.

Es recomendable entonces darle una leída al capítulo 1 del libro Homotopy Type Theory: Univalent Foundations of Mathematics donde se explica con mucho más detalle estos conceptos de manera teórica. Otra lectura recomendada es el libro Software Foundations que introduce estos conceptos utilizando Coq.

3. Mecanizando teoremas de Coincidencia y Substitución

3.1. Módulos.

- CaseTactic.v: Algunas tácticas útiles.
- Semantics.v: Definición de la semántica denotacional del lenguaje.
- Syntax.v: Definición de la sintaxis del lenguaje.
- Coincidence.v: Enunciados de coincidencia para expresiones enteras y booleanas.
- Substitution.v: Enunciados de substitución para expresiones enteras y booleanas.
- Examples.v: Definición de notaciones y algunos ejemplos.

La formalización en general sigue muy de cerca las definiciones vistas en el teórico de la materia; en muchos casos las podemos encontrar escritas de manera muy similar a como las escribimos en la "hoja" gracias al uso de company-coq. Sin embargo el módulo Syntax presenta la sintaxis de los lenguajes utilizando índices de De Bruijn en lugar de variables con nombre. Esto significa que en lugar de nombres de variables (x, y, z, etc) utilizamos números naturales para representar variables, donde cada número denota la cantidad de ocurrencias ligadoras que existen entre él y su ocurrencia ligadora, así por ejemplo la expresión $\forall x.\, x = x$ se escribe como $\forall\, 0 = 0$ debido a que entre la cuantificación $\forall x$ y la expresión $x = x$ no existe ningún otro cuantificador. En cambio, si consideramos la expresión $\forall x.\, \forall y.\, x = x$, entre la cuantificación $\forall x$ y la expresión $x = x$ encontramos la cuantificación $\forall y$ luego el índice correspondiente a x será 1, por lo tanto escribimos $\forall\, \forall\, 1 = 1$.

Una de las ventajas principales de utilizar índices de De Bruijn es que nos evitamos tener el *problema de la captura* al substituir. Notar además que para toda expresión su conjunto de variables libres está acotado por algún natural n, es decir, dada cualquier expresión existe un n tal que todo índice (variable) libre en ella pertenece al conjunto $\{0, 1, 2, \ldots, n-1\}$. En particular podemos indicar para cada índice i cuáles son los naturales n que caracterizan a los

Nombres	De Bruijn
$\forall x.\,(x = x)$	$\forall\,(0 = 0)$
$\forall x.\,(x < x + \overline{1})$	$\forall\,(0 < 0 + \overline{1})$
$\forall x.\,\forall y.\,(x = x)$	$\forall\forall\,(1 = 1)$
$\forall y.\,\forall x.\,(x = x)$	$\forall\forall\,(0 = 0)$
$\forall y.\,\forall x.\,(x = y)$	$\forall\forall\,(0 = 1)$
$\forall y.\,\forall x.\,(\forall z.\,z = z) \wedge (y = y)$	$\forall\forall\,(\forall\,0 = 0) \wedge (1 = 1)$
$\forall x.\,\forall y.\,\forall z.\;\; (x = z) \wedge (z = y) \Rightarrow (x = y)$	$\forall\forall\forall\;\; (2 = 0) \wedge (0 = 1) \Rightarrow (2 = 1)$

Ejemplos de equivalencia entre Nombres y De Bruijn

conjuntos de variables libres a los cuales pertenece, si i es 0 entonces pertenece al conjunto $\{0\}$ que lo podemos caracterizar con el natural 1, pero también pertenece a $\{0, 1\}$ que lo caracteriza el 2, etc. Por lo tanto cualquier natural mayor estricto que 0 caracteriza al conjunto de variables libres que el índice 0 puede pertenecer. Si i es mayor que 0 luego pertenece al conjunto caracterizado por $i + 1$, $i + 2$, etc. Esto en Coq lo podemos expresar con el siguiente tipo que define a su vez los índices.

```
Inductive Var : ℕ → Type :=
| ZVAR : ∀ n, Var (n+1)
| SVAR : ∀ n, Var n → Var (n+1)
.
```

El índice `ZVAR` (es decir, el 0) se va a encontrar en el subconjunto de variables libres definido para cualquier n tal que $n > 0$, es decir `ZVAR` tendrá tipo `Var n` para cualquier `n` tal que `n` > 0. Luego cualquier otro índice i deberá estar en al menos un entorno definido para $i + 1$; notar por ejemplo, el índice `SVAR (SVAR (ZVAR))` (es decir, el 2) tendrá tipo al menos `Var 3` pero también puede tener tipo `Var 20`, no así `Var 1` (Pregunta: ¿por qué no?).

```
Inductive IntExp (n : ℕ) : Type :=
| VAR  : Var n    → IntExp n
| NAT  : nat      → IntExp n
| Neg  : IntExp n → IntExp n
⋮
.
Inductive Assert (n : ℕ) : Type :=
| BOOL   : bool         → Assert n
| Not    : Assert n     → Assert n
| Forall : Assert (n+1) → Assert n
⋮
.
```

Utilizamos entonces un natural para definir el tipo de las construcciones sintácticas para las expresiones enteras y booleanas de manera que el número `NAT 1` (no confundir con el índice `VAR (SVAR ZVAR)`) tendrá tipo `IntExpr n` tal que $0 \leq$ `n`. Sin embargo, cualquier índice `VAR i` tendrá tipo `IntExpr n` tal que `i` tiene tipo `Var n`. El caso interésate ocurre para el caso del para todo donde dada la expresión `ae : Assert n+1`, es decir con al menos `n+1` variables libres, nos construimos la expresión `Forall ae` que tendrá una variable libre menos.

3.2. Tareas.

El código Coq a completar puede descargarse de acá. Las tareas concretas a realizar son completar las pruebas (escribir los correspondientes programas) para los enunciados de coincidencia y substitución que mostramos a continuación y que se pueden encontrar en sus correspondientes módulos. Estos significa que las construcciones sintácticas y su correspondiente semántica ya están formalizadas, así como también varios lemas que servirán para completar las pruebas de estos teoremas.

```
Theorem Coincidence_IntExp : ∀ (E : Env) (ie : IntExp E) (σ σ' : Σ(E)),
    (∀ (w : Var E), σ ! w = σ' ! w) → ⟦ie⟧_i σ = ⟦ie⟧_i σ'.

Theorem Coincidence_Assert : ∀ (E : Env) (ae : Assert E) (σ σ' : Σ(E)),
    (∀ (w : Var E), σ ! w = σ' ! w) → ⟦ae⟧_a σ = ⟦ae⟧_a σ'.

Theorem Substitution_IntExp : ∀ (E E': Env) (ie : IntExp E)
                                (σ : Σ(E')) (σ' : Σ(E)) (δ : Δ E E'),
    (∀ (w : Var E), ⟦δ w⟧_i σ' = σ' ! w) → ⟦ie //_i δ⟧_i σ = ⟦ie⟧_i σ'.

Theorem Substitution_Assert : ∀ (E E': Env) (ae : Assert E)
                                (σ : Σ(E')) (σ' : Σ(E)) (δ : Δ E E'),
    (∀ (w : Var E), ⟦δ w⟧_i σ' = σ' ! w) → ⟦ae //_a δ⟧_a σ = ⟦ae⟧_a σ'.
```

Recomendaciones: 1. puede ser útil utilizar el módulo de ejemplos para ganar alguna intuición sobre la utilización de índices de De Bruijn. 2. es muy recomendable tener las pruebas de cada teorema hechas en papel antes de empezar con la formalización de manera que sirvan como guía.

4. Apéndice

Ayudas rápidas:

- Emacs+ProofGeneral
- company-coq