Introducción a los asistentes de pruebas basados en teoría de tipos dependientes, en específico “Coq” (y resultados parciales sobre Redex $\to$ Coq)

UMA 2023 Salta

Mallku Soldevila[1], Beta Ziliani[2]

[1]FAMAF/UNC y CONICET, [2]FAMAF/UNC y Manas.Tech

Resumen

- Fundamentos (ideas de la matemática de primera mitad del siglo pasado).
- Coq.
- Nuestro trabajo:
  - Redex $\rightarrow$ Coq.

Fundamentos

(o cómo podemos representar computacionalmente

nociones de lógica matemática)

Fundamentos

Intuicionismo (L.E.J. Brouwer, desde 1908 aprox.)[1] [2]

- Basado en ideas filosóficas sobre la naturaleza de objetos matemáticos diferentes a las presentes en la matemática clásica.

- Opuesto al *platonismo*: los objetos matemáticos no existen más allá de nuestro intelecto. Son solamente construcciones de nuestra mente.

[1] Mauricio Guillermo. *Introducción a la Realizabilidad Intuicionista*. ECI'21.
[2] Harrie de Swart. *Philosophical and Mathematical Logic*. Springer, 2018.

Fundamentos: Intuicionismo

- Postura clásica: la validez o no validez de un enunciado depende de la naturaleza (objetiva) de los objetos de los que el enunciado habla, y no tanto de si disponemos o no de métodos efectivos para determinar esto.

- Intuicionismo: un enunciado es cierto sólo si tenemos una construcción matemática, una prueba, que hace evidente la validez del enunciado.

- Se enmarca dentro de lo que, más ampliamente, se conoce como *constructivismo*.

Fundamentos: Intuicionismo

Semántica de Brouwer-Heyting-Kolmogorov (BHK) (desde 1928, aprox.)

- Identifica la idea (la semántica) de enunciado verdadero con la idea de enunciado *demostrable*, y la idea de enunciado falso con la idea de enunciado *refutable*.

- La semántica de los conectivos lógicos se va a explicar en términos de la construcción de la correspondiente prueba.

Fundamentos: Semántica BHK

- $A \wedge B$: 2 piezas de información: prueba de A y prueba de B.
- $A \vee B$: 2 piezas de información: cuál es el disyunto demostrable y prueba correspondiente de ese disyunto.
- $A \to B$: método efectivo para transformar pruebas de A en pruebas de B.
- $\exists x \in A, B(x)$: testigo $x \in A$, y una prueba que demuestre $B(x)$.
- $\forall x \in A, B(x)$: método efectivo que transforma todo elemento $x \in A$ en una prueba que demuestre $B(x)$.
- No hay pruebas para $\bot$.
- $\neg A$ se define como $A \to \bot$.

Fundamentos: Semántica BHK

- Algunas consecuencias:
  - No tiene sentido asumir $\forall A, A \vee \neg A$.
    - Sí puede ser cierto $A \vee \neg A$ para cierto A específico, pero tengo que probarlo.
  - $\neg\neg A \not\to A$
- Conexión entre noción de prueba intuicionista y recursión:
  - Realizabilidad de Kleene: relación entre funciones recursivas y pruebas de la aritmética intuicionista de Heyting.

Fundamentos: Semántica BHK

Deducción natural para la lógica intuicionista (fragmento).

$$\frac{}{A \vdash_{\mathcal{L}} A}$$

$$\frac{\Gamma, A \vdash_{\mathcal{L}} B}{\Gamma \vdash_{\mathcal{L}} A \Rightarrow B} \qquad \frac{\Gamma \vdash_{\mathcal{L}} A \Rightarrow B \quad \Gamma \vdash_{\mathcal{L}} A}{\Gamma \vdash_{\mathcal{L}} B}$$

$$\frac{\Gamma \vdash_{\mathcal{L}} A \quad \Gamma \vdash_{\mathcal{L}} B}{\Gamma \vdash_{\mathcal{L}} A \wedge B} \qquad \frac{\Gamma \vdash_{\mathcal{L}} A \wedge B}{\Gamma \vdash_{\mathcal{L}} A} \qquad \frac{\Gamma \vdash_{\mathcal{L}} A \wedge B}{\Gamma \vdash_{\mathcal{L}} B}$$

Va a ser semejante a la deducción natural para lógica clásica, reemplazando la reducción al absurdo: $\frac{\Gamma, \neg A \vdash_{\mathcal{L}} \bot}{\Gamma \vdash_{\mathcal{L}} A}$, por el principio de explosión: $\frac{\Gamma \vdash_{\mathcal{L}} \bot}{\Gamma \vdash_{\mathcal{L}} A}$.

Fundamentos

λ-cálculo[3] [4]

- Alonzo Church, 1932/33.
- Teoría sobre funciones en términos de reglas de reescritura, y no como conjuntos de pares ordenados.
- Partes constitutivas: un lenguaje y relaciones con dominio en este lenguaje.
- Veremos la formulación moderna simplificada.[5]

[3] Barendregt, H. P. *The Lambda Calculus: Its Syntax and Semantics*. North Holland, Amsterdam (1984).

[4] Felleisen M., Findler R. B., Flatt M. *Semantics Engineering with PLT Redex*. The MIT Press (2009).

[5] Formulación original: Church A. *A Set of Postulates for the Foundation of Logic*.

Fundamentos: λ-cálculo

Sintaxis

$t ::= \lambda\, x.\; t \mid x \mid t\; t$

Relaciones entre términos (términos que se "comportan" del mismo modo)

Evaluación de una función (noción de reducción):
$\rightarrow_\beta$: $(\lambda\, x.\; t)\; u \rightarrow_\beta t[u/x]$ ($t[u/x]$: subst. consistente sin captura)

"Los nombres de variables no importan" (conversión):
α : renombre consistente de variables

"Una función está univocamente definida por sus evaluaciones":
$\rightarrow_\eta$: $(\lambda\, x.\; t\; x)\; \eta\; t$ (con $x \notin FV(t)$)

Fundamentos: λ-cálculo

- El cálculo original se demostró inconsistente (paradoja de Kleene-Rosser).[6]

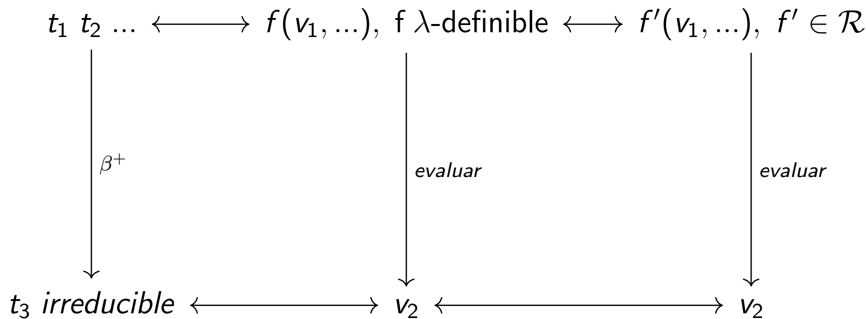
- La laxitud de las reglas de construcción del cálculo permiten cosas como "funciones aplicadas a sí mismas":
  - Por ejemplo: $\Omega \doteq (\lambda\, x.\ x\ x)\ (\lambda\, x.\ x\ x)$ (término irreducible).
  - Otro ejemplo: $Y \doteq \lambda\, f.\ (\lambda\, x.\ f\ (x\ x))\ (\lambda\, x.\ f\ (x\ x))$

- Si agregamos "tipos" (más adelante) para poder clasificar cosas como Y, todo tipo va a estar habitado (manifestación de inconsistencia en una teoría de tipos).

[6] Curry H. *The paradox of Kleene and Rosser*.

Fundamentos: λ-cálculo

Sin embargo: es posible representar todas las funciones recursivas de Gödel ($\mathcal{R}$; vía λ-definibilidad).



...y no-terminación de la reducción se corresponde con la no definición de una función sobre cierto valor.

Fundamentos: λ-cálculo

¿Por qué?

- Puedo representar información: por ej., $\mathbb{N} + 0$ usando numerales de Church.

  $0 = \lambda\ s.\ \lambda\ z.\ z$
  $1 = \lambda\ s.\ \lambda\ z.\ s\ z$
  ...
  $n = \lambda\ s.\ \lambda\ z.\ s^n\ z$

- Puedo hacer aritmética: por ej., sumar 1.

$$
\begin{aligned}
succ &= \lambda\ n.\ \lambda\ s.\ \lambda\ z.\ s\ (n\ s\ z) \\
succ\ 0 = succ\ (\lambda\ s.\ \lambda\ z.\ z) &\rightarrow_\beta \lambda\ s.\ \lambda\ z.\ s\ ((\lambda\ s.\ \lambda\ z.\ z)\ s\ z) \\
&\rightarrow_\beta \ldots \\
&\rightarrow_\beta \lambda\ s.\ \lambda\ z.\ s\ z = 1
\end{aligned}
$$

Fundamentos: λ-cálculo

- Toda función tiene punto fijo, y tengo un operador de (menor[7]) punto fijo.

  $$Y \doteq \lambda f.\,(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$$

  $g\ (Y\ g) \equiv Y\ g$, con g una abstracción λ.[8]

- ...entonces puedo definir funciones recursivas.

- Todo esto (y más) generó entusiasmo: conjetura de Church-Turing: *"hemos tenido éxito capturando la noción de función computable"*.

[7] Scott D. *Lambda Calculus: Some Models, Some Philosophy*

[8] Con $\equiv$ la clausura reflexiva-simétrica-transitiva y compatible de $\rightarrow_\beta$.

Fundamentos

λ-cálculo simplemente tipado

- Alonzo Church, 1940.
- Eliminamos términos que se “comportan mal” (ej.: funciones que se pueden aplicar a sí mismas).
- ...o, nos quedamos sólo con los términos que se comportan bien, usando teoría de tipos (este es el estilo de una teoría de tipos)

Fundamentos

Teoría de tipos[9]

- Bertrand Russell, 1903.[10]
- Desarrollada como instrumento para definir objetos evitando las paradojas en los fundamentos de la matemática que estaban siendo identificadas en la época.
- La teoría se preocupa por clasificar objetos utilizando la noción primitiva de "tipo".
- El juicio básico que le interesa formular a una teoría de tipos es: "el objeto a tiene tipo A", denotado $\mathrm{a} : \mathrm{A}$.

---

[9] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. 2013.

[10] Russell B. *The Principles of Mathematics*.

Fundamentos: Teoría de tipos

- El tipo a su vez expresa especificaciones detalladas de los objetos que clasifica. Esto da lugar a principios de razonamiento y manipulación de estos objetos.

Fundamentos: Teoría de tipos

Vamos a pensar y a definir a los objetos y a los tipos en términos de lo siguiente:

- **Reglas de formación**, que especifican cómo formar un nuevo tipo.

  Ej.: dados tipos A y B, podemos formar el tipo función $A \rightarrow B$.

- **Constructores** o **reglas de introducción**, que expliquen cómo construir habitantes del tipo.

  Ej.: utilizando el λ-cálculo, el tipo función tiene un constructor: las abstracciones λ.

- **Reglas de eliminación**, que indiquen cómo utilizar habitantes del tipo.

  Ej.: el tipo función tiene un eliminador: aplicación de funciones.

Fundamentos: Teoría de tipos

Vamos a pensar y a definir a los objetos y a los tipos en términos de lo siguiente:

- **Regla de cómputo**, que indica (define) cómo opera un eliminador sobre un constructor.

  Ej., para el tipo función, esto es lo que expresa la β-reducción.

- **Un principio de unicidad** (opcional), útil para razonar sobre enunciados que hablan de "igualdad" entre objetos.

  Ej., para el tipo función, esto es lo que expresa la η-reducción: una función está univocamente definida por sus evaluaciones.

Fundamentos: Teoría de tipos

- Teoría de tipos se va a interesar por hacer juicios sobre el tipo de las reglas de introducción y de eliminación.
- Las reglas de cómputo y los principios de unicidad son enunciados de naturaleza diferente a los juicios de tipo.

Fundamentos

λ-cálculo extendido con productos (como tipos primitivos), simplemente tipado.[11] Relación de tipado (fragmento):

$$\frac{}{x : A \vdash_{\mathcal{T}} x : A}$$

$$\frac{\Gamma, x : A \vdash_{\mathcal{T}} t : B}{\Gamma \vdash_{\mathcal{T}} \lambda\, x : A.\ t : A \to B} \qquad \frac{\Gamma \vdash_{\mathcal{T}} t : A \to B \quad \Gamma \vdash_{\mathcal{T}} u : A}{\Gamma \vdash_{\mathcal{T}} t\ u : B}$$

$$\frac{\Gamma \vdash_{\mathcal{T}} t : A \quad \Gamma \vdash_{\mathcal{T}} u : B}{\Gamma \vdash_{\mathcal{T}} (t, u) : A \times B} \qquad \frac{\Gamma \vdash_{\mathcal{T}} t : A \times B}{\Gamma \vdash_{\mathcal{T}} \pi_1\ t : A} \qquad \frac{\Gamma \vdash_{\mathcal{T}} t : A \times B}{\Gamma \vdash_{\mathcal{T}} \pi_2\ t : B}$$

[11] Wadler P. *Proofs are Programs: 19th Century Logic and 21st Century Computing*.

Fundamentos: λ-cálculo simplemente tipado

Propiedades:

- Reducción preserva tipos:

$$\dfrac{\dfrac{\Gamma \vdash_{\mathcal{T}} t : A \qquad \Gamma \vdash_{\mathcal{T}} u : B}{\Gamma \vdash_{\mathcal{T}} (t, u) : A \times B}}{\Gamma \vdash_{\mathcal{T}} \pi_1((t, u)) : A} \quad \to \quad \Gamma \vdash_{\mathcal{T}} t : A$$

- ...y los árboles de tipado se van reduciendo o contienen términos con tipos cada vez más "simples".

- Esto es: todo término del λ-cálculo simplemente tipado es "normalizable": lo podemos reducir hasta llegar a un término en formal normal (irreducible).

Fundamentos: λ-cálculo simplemente tipado

Propiedades:

- En este sistema de tipos no hay solución para una ecuación de tipos de la forma $A = A \rightarrow A$.
  - Para que esto sí tenga solución tenemos que introducir tipos recursivos.
- Por lo tanto, hemos retirado términos que representen funciones que se aplican sobre sí mismas, como en Ω y en Y.
  - Y en general, se puede probar que no hay operadores de punto fijo.[12]
- Tipado es decidible.

[12] Barendregt, H. P. *The Lambda Calculus: Its Syntax and Semantics*. North Holland, Amsterdam (1984).

Fundamentos

Correspondencia de Curry-Howard (fragmento).

$$\frac{}{x : A \vdash_{\mathcal{T}} x : A} \qquad \frac{}{A \vdash_{\mathcal{L}} A}$$

$$\frac{\Gamma, x : A \vdash_{\mathcal{T}} t : B}{\Gamma \vdash_{\mathcal{T}} \lambda\, x : A.\, t : A \rightarrow B} \qquad \frac{\Gamma, A \vdash_{\mathcal{L}} B}{\Gamma \vdash_{\mathcal{L}} A \Rightarrow B}$$

$$\frac{\Gamma \vdash_{\mathcal{T}} t : A \rightarrow B \qquad \Gamma \vdash_{\mathcal{T}} u : A}{\Gamma \vdash_{\mathcal{T}} t\ u : B} \qquad \frac{\Gamma \vdash_{\mathcal{L}} A \Rightarrow B \qquad \Gamma \vdash_{\mathcal{L}} A}{\Gamma \vdash_{\mathcal{L}} B}$$

$$\frac{\Gamma \vdash_{\mathcal{T}} t : A \qquad \Gamma \vdash_{\mathcal{T}} u : B}{\Gamma \vdash_{\mathcal{T}} (t,\ u) : A \times B} \qquad \frac{\Gamma \vdash_{\mathcal{L}} A \qquad \Gamma \vdash_{\mathcal{L}} B}{\Gamma \vdash_{\mathcal{L}} A \wedge B}$$

$$\frac{\Gamma \vdash_{\mathcal{T}} t : A \times B}{\Gamma \vdash_{\mathcal{T}} \pi_1\ t : A} \qquad \frac{\Gamma \vdash_{\mathcal{L}} A \wedge B}{\Gamma \vdash_{\mathcal{L}} A}$$

$$\frac{\Gamma \vdash_{\mathcal{T}} t : A \times B}{\Gamma \vdash_{\mathcal{T}} \pi_2\ t : B} \qquad \frac{\Gamma \vdash_{\mathcal{L}} A \wedge B}{\Gamma \vdash_{\mathcal{L}} B}$$

Fundamentos: Correspondencia de Curry-Howard

- Dado un término, podemos construir su derivación de tipos (y vice versa).
- Dada una derivación de tipos, podemos reconstruir la prueba (y vice versa).
- Entonces, extendiendo apropiadamente nuestro λ-cálculo, hay un isomorfismo T entre pruebas y términos tal que:

$$\Gamma \vdash_{\mathcal{L}} A \;\leftrightarrow\; \hat{\Gamma} \vdash_{\mathcal{T}} t : T(A),$$

donde $\hat{\Gamma}$ tiene hipótesis de la forma x : T(B), para B en Γ.[13]

[13] 1969, Howard, W.A. Aunque publicado recién en *The formulae-as-types notion of construction*. En *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press, 1980.

Fundamentos: Correspondencia de Curry-Howard

- "Connecting mathematical logic and computation, it ensures that some aspects of programming are absolute" (Philip Wadler)

Fundamentos

Tipos dependientes

- Martin-Löf, 1972 (y variaciones publicadas a posterior).
- Introduce tipos dependientes como forma de dar una explicación de predicados en lógica constructivista, basada en teoría de tipos.
- Tipo dependiente: una colección de tipos indexada por un cierto tipo A:

$$A \to \mathcal{U}, \text{con } \mathcal{U} \text{ un "tipo" habitado por tipos (un universo)}$$

Fundamentos: Tipos dependientes

Tipos dependientes de interés: dado un tipo A y una familia de tipos $B : A \to \mathcal{U}$:

- Tipo función dependiente, o Π-tipo:
  - Regla de formación: $\Pi_{x:A} B(x)$.
  - Regla de introducción: una abstracción λ.
  - Se va a corresponder con la noción de prueba intuicionista de una cuantificación universal.
- Tipo par dependiente, o Σ-tipo:.
  - Regla de formación: $\Sigma_{x:A} B(x)$.
  - Regla de introducción: mediante un par ordenado.
  - Se va a corresponder con la noción de prueba intuicionista de un existencial.

Coq

Coq

- Asistente de pruebas basado en un sistema de tipos dependientes (se aprovecha de la correspondencia de Curry-Howard).
- Ayuda en la construcción del término que representa a una prueba, y a su posterior verificación.
- Las pruebas que construiremos aquí no serán útiles para su comunicación a otra persona. El foco estará puesto en la verificación automática del objeto prueba.
- Ampliamente utilizado en esfuerzos de formalización de semántica de lenguajes de programación.

Coq

Cálculo de Construcciones Inductivas (CIC)[14]

- Extensión del *Cálculo de Construcciones*: un λ-cálculo con tipos dependientes.

- Incluye un mecanismo para definir tipos inductivamente, un operador de punto fijo (restringido para recursión primitiva) y encaje de patrones.

- La riqueza expresiva del sistema de tipos me permite disponer de una lógica de alto orden.

---

[14] Paulin-Mohring C. *Introduction to the Calculus of Inductive Constructions*

Coq: Cálculo de Construcciones Inductivas

- Se sabe que este cálculo posee propiedades deseables:
  - Normalización fuerte.
  - Canonicidad.
  - Consistencia relativa con respecto a ZFC.

Coq: Cálculo de Construcciones Inductivas

Sorts

- Los tipos habitan *universos*, llamados *sorts*.

- En CIC tenemos una jerarquía infinita de sorts: $\{Prop\} \cup \{Set\} \cup \bigcup_{i \in \mathbb{N}} \{Type_i\}$.
  - Sort *Prop* está habitado por los tipos que representan proposiciones *proof-irrelevant*.
  - Sort *Set* está habitado por los tipos computacionalmente relevantes.
  - $Prop : Type_1$ y $Set : Type_1$.
  - $Type_i : Type_{i+1}$, para evitar inconsistencias que surgen en sistemas donde existe un único sort *Type*.

Coq

Ejemplo: números naturales como un tipo inductivo.

```
Inductive nat : Set :=
| O : nat
| S : nat → nat.
```

- Formador o constructor de tipos:
  - `nat` (que va a habitar el sort `Set`).
  - `nat` será el menor punto fijo de la función *generadora* (o *funcional*) definida por los constructores `O` y `S`.
- Reglas de introducción: constructores `O` y `S`.
- Reglas de eliminación: encaje de patrones (forma sintáctica más adelante) o *análisis por casos*.

Coq

Ejemplo: números naturales como un tipo inductivo.

```
Inductive nat : Set :=
| O : nat
| S : nat → nat.
```

- Reglas de cómputo: deriva de la regla de reducción del encaje de patrones sobre un natural.

- Análisis por casos:
  - *Normalización fuerte* +
  - *Canonicidad* +
  - *Valores construidos con constructores diferentes, son diferentes*

Coq

Ejemplo: números naturales como un tipo inductivo.

```
Inductive nat : Set :=
| O : nat
| S : nat → nat.
```

- Principios de inducción: es una función! (donde Sort ∈ {Prop, SProp, Set, Type}):

```
forall P : nat → Sort,
  P 0 →
  (forall n : nat, P n → P (S n)) →
  forall n : nat, P n
```

- Lógica de alto orden: basta con un principio cuantificado sobre todas las proposiciones posibles.

Coq

Ejemplo: orden sobre los números naturales.

```
Inductive le (n : nat) : nat → Prop :=
| le_n : le n n
| le_S : forall m : nat, le n m → le n (S m).
```

- `le` es la menor relación que satisface:
  - $\forall n : \mathbb{N} \cup \{0\}$, `le` $n\ n$
  - $\forall n, m : \mathbb{N} \cup \{0\}$, `le` $n\ m \Rightarrow$ `le` $n\ (m + 1)$
- `le_n` y `le_S` definen un sistema formal completo y correcto para construir pruebas de cosas de la forma $n \leq m$.

Coq

Ejemplo: tipo habitado por los primeros n números naturales.

```
Inductive fin (n : nat) : Set :=
| elem_fin : forall (m : nat), le m n → fin n.
```

- Al construir habitantes de `fin`, es posible automatizar la generación de pruebas de `le m n`: *typeclasses*.

Coq

Conectivos lógicos y sus análogos vía Curry-Howard.

- Conjunción:

```
Inductive and (A B : Prop) : Prop := conj : A → B → A ∧ B.

Inductive prod (A B : Type) : Type := pair : A → B → A * B.
```

- Disyunción:

```
Inductive or (A B : Prop) : Prop :=
| or_introl : A → A ∨ B
| or_intror : B → A ∨ B.

Inductive sum (A B : Type) : Type :=
| inl : A → A + B
| inr : B → A + B.
```

Coq

Conectivos lógicos y sus análogos vía Curry-Howard.

- Cuantificación existencial:

```
Inductive ex (A : Type) (P : A → Prop) : Prop :=
ex_intro : forall x : A, P x → exists y, P y.

Inductive sigT (A : Type) (P : A → Type) : Type :=
existT : forall x : A, P x → {x : A & P x}.
```

- Cuantificación universal e implicación:

$$\frac{\Gamma, x : A \vdash_{\mathcal{T}} t : B}{\Gamma \vdash_{\mathcal{T}} \textbf{fun } x : A \Rightarrow t : A \rightarrow B} \qquad \frac{\Gamma, x : A \vdash_{\mathcal{T}} t : B(x)}{\Gamma \vdash_{\mathcal{T}} \textbf{fun } x : A \Rightarrow t : \Pi_{x:A}, B(x)}$$

$$\frac{\Gamma, x : A \vdash_{\mathcal{T}} B : Prop}{\Gamma \vdash_{\mathcal{T}} \Pi_{X:A}, B : Prop} \qquad \frac{\Gamma, x : A \vdash_{\mathcal{T}} B : Type_i \quad \Gamma \vdash_{\mathcal{T}} A : Type_i}{\Gamma \vdash_{\mathcal{T}} \Pi_{X:A}, B : Type_i}$$

Coq

Conectivos lógicos y sus análogos vía Curry-Howard.

- Se corresponden con la noción intuicionista de prueba, y el tipo de los constructores se corresponden con las reglas de introducción del sistema de deducción natural.

Coq

Forma general de una definición inductiva.

```
Inductive I pars : Ar := ...
| c : Π(x1 : A1)...(xn : An), I pars u1...up
| ...
```

- pars are called the parameters of the inductive definition and will be the same for all definitions.
- Ar is called the arity. Tiene la forma $\Pi(y_1 : B_1)...(y_p : y_p), s$, con s un Sort.
- u_i is an index. Los tipos pueden estar indexados por cualquier expresión del lenguaje, además de permitir indexación por tipos
- $\Pi(x_1 : A_1)...(x_n : A_n)$, I pars $u_1...u_p$ is a type of constructor.
- A_i is a type of argument of constructor.

Coq

Forma general de una definición inductiva.

```
Inductive I pars : Ar := ...
| c : Π(x1 : A1)...(xn : An), I pars u1...up
| ...
```

Hay condiciones (sintácticas) que se imponen para poder considerar una definición dada como *bien formada*.

- Para garantizar predicatividad.
- Para garantizar normalización fuerte.

Coq

Construcciones restantes de CIC: operador de punto fijo para definir funciones recursivas.

`Fixpoint` `f` $(x_1 : A_1)...(x_n : A_n)$ $\{$`struct` $x_i\}$: `B` := `t`.

- `f` : $\Pi(x_1 : A_1)...(x_n : A_n), B$.
- Sólo recursión primitiva: evaluaciones recursivas de `f` sólo sobre un término *estructuralmente* más pequeño que x_n.

Coq

Construcciones restantes de CIC: operador de punto fijo para definir funciones recursivas.

Fixpoint f $(x_1 : A_1)...(x_n : A_n)$ {struct x_i} : B := t.

- ...recursión general interactúa con el sistema de tipos: todo tipo termina estando habitado por términos superfluos (sin información).

```
Fixpoint mal (n : nat) : P := mal n.
(* para cualquier n y P, mal n : P *)
```

- ...inclusive Falso:

```
Inductive False : Prop := .
```

- Sólo vamos a permitir funciones totales.

Coq

Construcciones restantes de CIC: encaje de patrones.

```
Fixpoint sum (n : nat) : nat :=
  match n with
  | 0    => 0
  | S n' => S n' + sum n'
  end.
```

- Funciones totales: tenemos que contemplar las 2 posibles formas de construir un natural.
- Implementa el principio de eliminación para tipos inductivos (positivos): me permite extraer la información con la que se construyó un valor.
- Puedo razonar por casos sobre un valor de cualquier tipo inductivo.

Coq

Ejemplo de prueba:

```
Lemma sum_expr : forall (n : nat), sum n = (n * (n + 1)) / 2.
```

- Luego del punto final del comando `Lemma` entramos en modo interactivo de edición de prueba:

```
1 goal (ID 2)

______________________________________
forall n : nat, sum n = n * (n + 1) / 2
```

- Esto es el *proof state*: la colección de hipótesis que manejamos + el enunciado que tenemos que probar.

- Manipulamos el estado de la prueba mediante comandos llamados *tácticas*.

Coq

Ejemplo de prueba:

`Lemma sum_expr : forall (n : nat), sum n = (n * (n + 1)) / 2.`

- *Razonamiento hacia atrás*: partimos de la conclusión del árbol de prueba que queremos construir, y vamos hacia arriba, hacia las hojas del árbol.
- Coq nos va a ayudar a construir el objeto que representa a la prueba, llamado *proof term*.

Coq

Ejemplo de prueba:

```
Require Import
        Arith.Arith
        Lia.

Fixpoint sum (n : ℕ) : ℕ :=
  match n with
  | 0    ⇒ 0
  | S n' ⇒ S n' + sum n'
  end.

Lemma sum_expr : ∀ (n : ℕ), sum n = (n * (n + 1)) / 2.
Proof.
```

```
1 goal (ID 2)

  ______________________________________
  ∀ n : ℕ, sum n = n * (n + 1) / 2

U:%%-  *goals*   All   (3,12)   (Coq Goals +2)
```

Coq

Ejemplo de prueba:

- Razonamos por inducción en `n`. Podemos aplicar la función que representa el principio de inducción generado para `nat`, `nat_ind`:

```
forall P : nat → Prop,
 P 0 →
 (forall n : nat, P n → P (S n)) →
 forall n : nat, P n
```

- La prueba será entonces un término de la forma:
  `fun n : nat ⇒ nat_ind P ... ... n.`

- Podemos utilizar la táctica `induction n`:

Coq

Ejemplo de prueba:

```
Require Import
        Arith.Arith
        Lia.

Fixpoint sum (n : ℕ) : ℕ :=
  match n with
  | 0    ⇒ 0
  | S n' ⇒ S n' + sum n'
  end.

Lemma sum_expr : ∀ (n : ℕ), sum n = (n * (n + 1)) / 2.
Proof.
  induction n as [ | n' HI].
```

```
2 goals (ID 7)

  ______________________________________
  sum 0 = 0 * (0 + 1) / 2

goal 2 (ID 10) is:
 sum (S n') = S n' * (S n' + 1) / 2

U:%%-  *goals*       All   (4,0)       (Coq Goals +2)
```

Coq

Ejemplo de prueba:

```
Require Import
        Arith.Arith
        Lia.

Fixpoint sum (n : ℕ) : ℕ :=
  match n with
  | 0     ⇒ 0
  | S n' ⇒ S n' + sum n'
  end.

Lemma sum_expr : ∀ (n : ℕ), sum n = (n * (n + 1)) / 2.
Proof.
  induction n as [ | n' HI].
  - (* caso base *)
```

```
1 goal (ID 7)

  ______________________________________
  sum 0 = 0 * (0 + 1) / 2

U:%%-  *goals*        All   (4,0)        (Coq Goals +2)
```

Coq

Ejemplo de prueba:

```
Require Import
        Arith.Arith
        Lia.

Fixpoint sum (n : ℕ) : ℕ :=
  match n with
  | 0    ⇒ 0
  | S n' ⇒ S n' + sum n'
  end.

Lemma sum_expr : ∀ (n : ℕ), sum n = (n * (n + 1)) / 2.
Proof.
  induction n as [ | n' HI].
  - (* caso base *)
    (* son términos definicionalmente iguales *)
    reflexivity.
```

```
1 goal

goal 1 (ID 10) is:
 sum (S n') = S n' * (S n' + 1) / 2
```

```
U:%%-  *goals*      All    (4,0)       (Coq Goals +2)
This subproof is complete, but there are s
unfocused goals.
Focus next goal with bullet -.
```

Coq

Ejemplo de prueba:

```
Require Import
        Arith.Arith
        Lia.

Fixpoint sum (n : ℕ) : ℕ :=
  match n with
  | 0    ⇒ 0
  | S n' ⇒ S n' + sum n'
  end.

Lemma sum_expr : ∀ (n : ℕ), sum n = (n * (n + 1)) / 2.
Proof.
  induction n as [ | n' HI].
  - (* caso base *)
    (* son términos definicionalmente iguales *)
    reflexivity.
  - (* caso inductivo *)
```

```
1 goal (ID 10)

n' : ℕ
HI : sum n' = n' * (n' + 1) / 2
______________________________________
sum (S n') = S n' * (S n' + 1) / 2

U:%%-  *goals*       All   (6,0)       (Coq Goals +2)
```

Coq

Ejemplo de prueba:

```
Require Import
        Arith.Arith
        Lia.

Fixpoint sum (n : ℕ) : ℕ :=
  match n with
  | 0    ⇒ 0
  | S n' ⇒ S n' + sum n'
  end.

Lemma sum_expr : ∀ (n : ℕ), sum n = (n * (n + 1)) / 2.
Proof.
  induction n as [ | n' HI].
  - (* caso base *)
    (* son términos definicionalmente iguales *)
    reflexivity.
  - (* caso inductivo *)
    (* reducimos sum (S n) para poder usar la H.I. *)
    unfold sum.
    fold sum.
```

```
1 goal (ID 13)

  n' : ℕ
  HI : sum n' = n' * (n' + 1) / 2
  ______________________________________
  S n' + sum n' = S n' * (S n' + 1) / 2

U:%%-  *goals*     All   (6,0)      (Coq Goals +2)
```

Coq

Ejemplo de prueba:

```
Require Import
        Arith.Arith
        Lia.

Fixpoint sum (n : ℕ) : ℕ :=
  match n with
  | 0    ⇒ 0
  | S n' ⇒ S n' + sum n'
  end.

Lemma sum_expr : ∀ (n : ℕ), sum n = (n * (n + 1)) / 2.
Proof.
  induction n as [ | n' HI].
  - (* caso base *)
    (* son términos definicionalmente iguales *)
    reflexivity.
  - (* caso inductivo *)
    (* reducimos sum (S n) para poder usar la H.I. *)
    unfold sum.
    fold sum.
    (* aplicamos H.I. *)
    rewrite HI.
```

```
1 goal (ID 14)

  n' : ℕ
  HI : sum n' = n' * (n' + 1) / 2
  ______________________________________
  S n' + n' * (n' + 1) / 2 = S n' * (S n'
2

U:%%-  *goals*     All   (6,0)     (Coq Goals +2)
```

Coq

Ejemplo de prueba:

```
Require Import
        Arith.Arith
        Lia.

Fixpoint sum (n : ℕ) : ℕ :=
  match n with
  | 0    ⇒ 0
  | S n' ⇒ S n' + sum n'
  end.

Lemma sum_expr : ∀ (n : ℕ), sum n = (n * (n + 1)) / 2.
Proof.
  induction n as [ | n' HI].
  - (* caso base *)
    (* son términos definicionalmente iguales *)
    reflexivity.
  - (* caso inductivo *)
    (* reducimos sum (S n) para poder usar la H.I. *)
    unfold sum.
    fold sum.
    (* aplicamos H.I. *)
    rewrite HI.
    (* simple reducción no me ayuda a probar el objetivo *)
    replace (n' + 1) with (S n') by lia. (* por def. de + *)
```

```
1 goal (ID 18)

- n' : ℕ
- HI : sum n' = n' * (n' + 1) / 2
  ______________________________________
  S n' + n' * S n' / 2 = S n' * (S n' + 1)

U:%%-  *goals*     All   (6,0)     (Coq Goals +2)
```

Coq

Ejemplo de prueba:

```
Require Import
        Arith.Arith
        Lia.

Fixpoint sum (n : ℕ) : ℕ :=
  match n with
  | 0    ⇒ 0
  | S n' ⇒ S n' + sum n'
  end.

Lemma sum_expr : ∀ (n : ℕ), sum n = (n * (n + 1)) / 2.
Proof.
  induction n as [ | n' HI].
  - (* caso base *)
    (* son términos definicionalmente iguales *)
    reflexivity.
  - (* caso inductivo *)
    (* reducimos sum (S n) para poder usar la H.I. *)
    unfold sum.
    fold sum.
    (* aplicamos H.I. *)
    rewrite HI.
    (* simple reducción no me ayuda a probar el objetivo *)
    replace (n' + 1) with (S n') by lia. (* por def. de + *)
    (* Nat.add_comm: forall n m : nat, n + m = m + n *)
    rewrite Nat.add_comm.
```

```
1 goal (ID 39)

- n' : ℕ
- HI : sum n' = n' * (n' + 1) / 2
  ______________________________________
  n' * S n' / 2 + S n' = S n' * (S n' + 1)

U:%%-  *goals*     All   (6,0)      (Coq Goals +2)
```

Coq

Ejemplo de prueba:

```
- (* caso base *)
  (* son términos definicionalmente iguales *)
  reflexivity.
- (* caso inductivo *)
  (* reducimos sum (S n) para poder usar la H.I. *)
  unfold sum.
  fold sum.
  (* aplicamos H.I. *)
  rewrite HI.
  (* simple reducción no me ayuda a probar el objetivo *)
  replace (n' + 1) with (S n') by lia. (* por def. de + *)
  (* Nat.add_comm: forall n m : nat, n + m = m + n *)
  rewrite Nat.add_comm.
  (* Nat.div_add:
     forall a b c : nat, c <> 0 -> (a + b * c) / c = a / c + b *)
  rewrite ← Nat.div_add.
```

```
2 goals (ID 40)

- n' : ℕ
- HI : sum n' = n' * (n' + 1) / 2
  ____________________________________
  (n' * S n' + S n' * 2) / 2 = S n' * (S n
/ 2

goal 2 (ID 41) is:
 2 ≠ 0

U:%%-  *goals*      All   (6,0)      (Coq Goals +2)
```

Coq

Ejemplo de prueba:

```
- (* caso base *)
  (* son términos definicionalmente iguales *)
  reflexivity.
- (* caso inductivo *)
  (* reducimos sum (S n) para poder usar la H.I. *)
  unfold sum.
  fold sum.
  (* aplicamos H.I. *)
  rewrite HI.
  (* simple reducción no me ayuda a probar el objetivo *)
  replace (n' + 1) with (S n') by lia. (* por def. de + *)
  (* Nat.add_comm: forall n m : nat, n + m = m + n *)
  rewrite Nat.add_comm.
  (* Nat.div_add:
     forall a b c : nat, c <> 0 -> (a + b * c) / c = a / c + b *)
  rewrite ← Nat.div_add.
  + (* (n' * S n' + S n' * 2) / 2 = S n' * (S n' + 1) / 2 *)
```

```
1 goal (ID 40)

- n' : ℕ
- HI : sum n' = n' * (n' + 1) / 2
  ______________________________________
  (n' * S n' + S n' * 2) / 2 = S n' * (S r
/ 2

U:%%-  *goals*   All  (6,0)   (Coq Goals +2)
```

Coq

Ejemplo de prueba:

```
- (* caso base *)
  (* son términos definicionalmente iguales *)
  reflexivity.
- (* caso inductivo *)
  (* reducimos sum (S n) para poder usar la H.I. *)
  unfold sum.
  fold sum.
  (* aplicamos H.I. *)
  rewrite HI.
  (* simple reducción no me ayuda a probar el objetivo *)
  replace (n' + 1) with (S n') by lia. (* por def. de + *)
  (* Nat.add_comm: forall n m : nat, n + m = m + n *)
  rewrite Nat.add_comm.
  (* Nat.div_add:
     forall a b c : nat, c <> 0 -> (a + b * c) / c = a / c + b *)
  rewrite ← Nat.div_add.
  + (* (n' * S n' + S n' * 2) / 2 = S n' * (S n' + 1) / 2 *)
    replace (n' * S n' + S n' * 2) with (S n' * (S n' + 1)) by lia.
```

```
1 goal (ID 45)

- n' : ℕ
- HI : sum n' = n' * (n' + 1) / 2
  ______________________________________
  S n' * (S n' + 1) / 2 = S n' * (S n' + 1

U:%%-  *goals*      All    (6,0)      (Coq Goals +2)
```

Coq

Ejemplo de prueba:

```
- (* caso base *)
  (* son términos definicionalmente iguales *)
  reflexivity.
- (* caso inductivo *)
  (* reducimos sum (S n) para poder usar la H.I. *)
  unfold sum.
  fold sum.
  (* aplicamos H.I. *)
  rewrite HI.
  (* simple reducción no me ayuda a probar el objetivo *)
  replace (n' + 1) with (S n') by lia. (* por def. de + *)
  (* Nat.add_comm: forall n m : nat, n + m = m + n *)
  rewrite Nat.add_comm.
  (* Nat.div_add:
     forall a b c : nat, c <> 0 -> (a + b * c) / c = a / c + b *)
  rewrite ← Nat.div_add.
  + (* (n' * S n' + S n' * 2) / 2 = S n' * (S n' + 1) / 2 *)
    replace (n' * S n' + S n' * 2) with (S n' * (S n' + 1)) by lia.
    reflexivity.
```

```
1 goal

goal 1 (ID 41) is:
 2 ≠ 0

U:%%-  *goals*      All    (4,0)      (Coq Goals +2)
This subproof is complete, but there are s
unfocused goals.
Focus next goal with bullet +.
```

Coq

Ejemplo de prueba:

```
- (* caso base *)
  (* son términos definicionalmente iguales *)
  reflexivity.
- (* caso inductivo *)
  (* reducimos sum (S n) para poder usar la H.I. *)
  unfold sum.
  fold sum.
  (* aplicamos H.I. *)
  rewrite HI.
  (* simple reducción no me ayuda a probar el objetivo *)
  replace (n' + 1) with (S n') by lia. (* por def. de + *)
  (* Nat.add_comm: forall n m : nat, n + m = m + n *)
  rewrite Nat.add_comm.
  (* Nat.div_add:
     forall a b c : nat, c <> 0 -> (a + b * c) / c = a / c + b *)
  rewrite ← Nat.div_add.
  + (* (n' * S n' + S n' * 2) / 2 = S n' * (S n' + 1) / 2 *)
    replace (n' * S n' + S n' * 2) with (S n' * (S n' + 1)) by lia.
    reflexivity.
  + (* 2 <> 0 *)
```

```
1 goal (ID 41)

  n' : ℕ
  HI : sum n' = n' * (n' + 1) / 2
  ______________________________________
  2 ≠ 0

U:%%-  *goals*    All   (6,0)    (Coq Goals +2)
```

Coq

Ejemplo de prueba:

```
- (* caso base *)
  (* son términos definicionalmente iguales *)
  reflexivity.
- (* caso inductivo *)
  (* reducimos sum (S n) para poder usar la H.I. *)
  unfold sum.
  fold sum.
  (* aplicamos H.I. *)
  rewrite HI.
  (* simple reducción no me ayuda a probar el objetivo *)
  replace (n' + 1) with (S n') by lia. (* por def. de + *)
  (* Nat.add_comm: forall n m : nat, n + m = m + n *)
  rewrite Nat.add_comm.
  (* Nat.div_add:
     forall a b c : nat, c <> 0 -> (a + b * c) / c = a / c + b *)
  rewrite ← Nat.div_add.
  + (* (n' * S n' + S n' * 2) / 2 = S n' * (S n' + 1) / 2 *)
    replace (n' * S n' + S n' * 2) with (S n' * (S n' + 1)) by lia.
    reflexivity.
  + (* 2 <> 0 *)
    lia.
```

```
U:%%-  *goals*        All   (1,0)      (Coq Goals +2)
No more goals.
```

Coq

Ejemplo de prueba:

```
- (* caso base *)
  (* son términos definicionalmente iguales *)
  reflexivity.
- (* caso inductivo *)
  (* reducimos sum (S n) para poder usar la H.I. *)
  unfold sum.
  fold sum.
  (* aplicamos H.I. *)
  rewrite HI.
  (* simple reducción no me ayuda a probar el objetivo *)
  replace (n' + 1) with (S n') by lia. (* por def. de + *)
  (* Nat.add_comm: forall n m : nat, n + m = m + n *)
  rewrite Nat.add_comm.
  (* Nat.div_add:
    forall a b c : nat, c <> 0 -> (a + b * c) / c = a / c + b *)
  rewrite ← Nat.div_add.
  + (* (n' * S n' + S n' * 2) / 2 = S n' * (S n' + 1) / 2 *)
    replace (n' * S n' + S n' * 2) with (S n' * (S n' + 1)) by lia.
    reflexivity.
  + (* 2 <> 0 *)
    lia.
Qed.
```

```
U:%%-  *goals*        All   (1,0)        (Coq Goals +2)
```

Coq

Ejemplo de prueba: vemos el término construido.

```
sum_expr = fun n : nat ⇒
             nat_ind
             (fun n0 : nat ⇒ sum n0 = n0 * (n0 + 1) / 2)
             eq_refl
             (fun (n' : nat) (HI : sum n' = n' * (n' + 1) / 2) ⇒ ...)
             n
```

Nuestro trabajo

Nuestro trabajo

Qué hacemos

- Formalización de la semántica dinámica y estática de lenguajes de programación *reales*.
  - Existen compiladores/intérpretes (demasiado complejos).
  - Hay manuales de referencia (informales).
- Queremos elaborar un modelo del lenguaje sobre el cual sea posible:
  - Verificar propiedades de corrección de la semántica informal (Ej., todo error durante la ejecución es reconocido por la semántica dinámica; un programa correctamente tipado se ejecuta sin errores de tipo).
  - Desarrollar herramientas de análisis de código.

Nuestro trabajo: Qué hacemos

Formalismo:

- Semántica operacional:
  - Útil para demostrar propiedades de sistemas de tipos.
  - Próxima a la forma en la que informalmente se piensa y se presenta la semántica de un lenguaje (ej., en manuales de referencia).
- Semántica de reducciones con contextos de evaluación:
  - Enfoque inspirado en ideas del λ-cálculo (Peter Landin).
  - Definiciones concisas, semántica modular.
  - Empleada (parcialmente) en esfuerzos de formalización de lenguajes varios (Python, Scheme, varias versiones de JavaScript, Lua 5.1 y 5.2).

Nuestro trabajo: Qué hacemos

PLT Redex

- Lenguaje de dominio específico implementado sobre Racket.
- Útil para mecanizar semánticas de reducciones con contextos de evaluación.

PLT Redex

Definición de una gramática en Redex

```
#lang racket

(require redex)

(define-language core-lang
  ; terms as compiled from Lua code
  [sing \;
        break
        (return e ...)
        ($statFCall e (e ...))
        ($statFCall e : Name (e ...))
        (var_1 var_2 ... = e ...)
        (do s end)
        (if e then s else s end)
        (while e do s end)
        ($iter e do s end)
        (local Name_1 Name_2 ... = e ... in s end)]
```

PLT Redex

Definición de una relación semántica en Redex

```
#lang racket
; Expressions that don't interact with some store

(require redex
         "../grammar.rkt"
         "../Meta-functions/grammarMetaFunctions.rkt"
         "../Meta-functions/delta.rkt")

(define terms-rel
  (reduction-relation
   ext-lang
   #:arrow -->s/e

   ; tuples
   [-->s/e (in-hole Et (< v_1 v_2 ... >))
           (in-hole Et v_1)
           E-TruncateNonEmptyTuple]

   [-->s/e (in-hole Et (< >))
           (in-hole Et nil)
           E-TruncateEmptyTuple]

   [-->s/e (in-hole Ea (< v_1 ... >))
           (fix_unwrap Ea (v_1 ...))
           E-UnwrapNonEmptyTuple]
```

PLT Redex

Definición de una relación mediante reglas de inferencia

```
; constraints for a list of expression (for example, for actual parameters of
; a fun call)
(define-judgment-form
  core-lang-typed
  #:mode (cons_gen_el I I O O O)
  #:contract (cons_gen_el γ (e ...) (τ ...) γ Cs)

  [(cons_gen γ_1 e τ γ_2 Cs)
   --------------------------------------------------------
   (cons_gen_el γ_1 (e) (τ) γ_2 Cs)]

  [(cons_gen γ_1 e_1 τ_1 γ_2 Cs_1)
   (cons_gen_el γ_2 (e_2 e_3 ...) (τ_2 ...) γ_3 Cs_2)
   (where Cs_3 (cons_un Cs_1 Cs_2))
   -----------------------------------------------------------
   (cons_gen_el γ_1 (e_1 e_2 e_3 ...) (τ_1 τ_2 ...) γ_3 Cs_3)]
  )
```

PLT Redex

- Otras facilidades de Redex
  - Testeo aleatorio de propiedades.
  - Visualización de trazas de reducción.
  - Facilidades para implementar suite de tests.
- Qué no puede hacer Redex:
  - Asistir en la demostración de propiedades.
  - Ofrecer garantías estáticas de corrección.

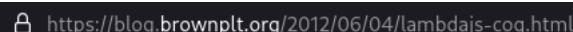
PLT Redex

Problema

- No hay facilidades para exportar un modelo Redex hacia un asistente de pruebas.
- Nos vemos en la obligación de descartar un modelo ya testeado e implementarlo desde 0 en un asistente de pruebas.

PLT Redex

La experiencia λ_{JS}:



Redex can also generate random tests to exercise your semantics. Random testing caught several more bugs in λ_{JS}.

Coq: A Machine-Checked Proof

Testing is not enough. We shipped λ_{JS} with a bug that breaks the soundness theorem above. We didn't discover it for a year. David van Horn and Ian Zerny both reported it to us independently. We'd missed a case in the semantics, which caused certain terms to get "stuck". It turned out to be a simple fix, but we were left wondering if anything else was left lurking.

To gain further assurance, we mechanized λ_{JS} with the Coq proof assistant. The soundness theorem now has a machine-checked proof of correctness. You still need to read the Coq definition of λ_{JS} and ensure it matches your intuitions. But once that's done, you can be confident that the proofs are valid.

Doing this proof was surprisingly easy, once we'd read Software Foundations and Certified Programming with Dependent Types. We'd like to thank Benjamin Pierce and his co-authors, and Adam Chlipala, for putting their books online.

PLT Redex

Propuesta:

- Implementar en Coq la semántica de Redex, basándonos en lo propuesto en *A Semantics for Context-Sensitive Reduction Semantics*, de Klein et. al.
- Extender el modelo con facilidades ausentes.
- Implementar una librería de tácticas para facilitar la demostración de propiedades.
- Implementar una rutina de traducción de un modelo Redex hacia un (idealmente) equivalemente semánticamente en Coq.

PLT Redex

Logros parciales

- Mecanizamos el modelo presentado en la bibliografía citada.
  - Algoritmo de encaje de patrones original no es una recursión primitiva.
  - Definimos una versión más general del algoritmo.
  - Definimos una relación *bien fundada* sobre los valores de entrada del algoritmo.
  - Probamos la buena fundación de la relación.
  - Mecanizamos el algoritmo mediante *recursión bien fundada*: recursión primitiva sobre la estructura de la prueba de accesibilidad de cada elemento en la relación.

PLT Redex

Logros parciales

- Reprodujimos la prueba de completitud del paper original, para nuestro modelo mecanizado en Coq.

- Extendimos el modelo con una noción de *clausura de kleene*.

PLT Redex

Logros parciales: ¡testeando, encontramos errores en una versión reciente de Redex!

https://github.com/racket/redex/commit/817c18d61e72b9d2f6b58de65d97181ac8da2009#diff-11c6448c00a1872fd2062f76244

fix bug in matcher

Thanks to Mallku Ernesto Soldevila Raffa for finding the bug

master
v8.10 ... v8.4

rfindler committed on Dec 21, 2021

Showing **2 changed files** with **29 additions** and **1 deletion**.

- redex-lib/redex/private
  - matcher.rkt
- redex-test/redex/tests
  - tl-language.rkt

redex-lib/redex/private/matcher.rkt

```
@@ -728,7 +728,7 @@ See match-a-pattern.rkt for more details
728 728   (define (build-compiled-pattern proc names lang-α-equal?)
729 729     (make-compiled-pattern
730 730      proc
731     -    (null? names)
    731 +    (not (null? names))
732 732
733 733      ;; none of the names are duplicated
734 734      (and (equal? names (remove-duplicates names))
```

PLT Redex

Logros parciales: hacia una teoría mecanizada sobre la decibilidad de predicados de semántica de reducciones estilo Redex.

- Comenzamos a experimentar con formas de automatizar casos particulares del problema de intersección de lenguajes, para el caso de lenguajes definidos con el formalismo de Redex.

- Construimos tipos finitos de lenguajes y patrones de tamaño acotado. Nos interesaría generalizar la técnica empleada a tipos con constructores como los utilizados en nuestro caso para definir lenguajes y patrones.

¡Gracias!