Lógica Computacional 2017-2, nota de clase 15
El Isomorfismo de Curry-Howard

Favio Ezequiel Miranda Perea Araceli Liliana Reyes Cabello
Lourdes Del Carmen González Huesca Pilar Selene Linares Arévalo

28 de mayo de 2017

1. Un lenguaje para expresiones aritméticas y booleanas

A continuación presentamos un lenguaje para expresiones aritméticas y booleanas que incluye las operaciones de suma, producto, orden, un test para cero y la negación booleana.
Las expresiones del lenguaje se definen como sigue:

$$\begin{aligned} e \quad ::= \quad & x \mid n \mid \mathsf{true} \mid \mathsf{false} \mid \\ & e + e \mid e * e \\ & e < e \mid \mathsf{iszero}\, e \mid \mathsf{not}\, e \end{aligned}$$

Ejemplos de expresiones del lenguaje son:

- $3 + y$
- $\mathsf{not}(\mathsf{iszero}\, 9)$
- $\mathsf{iszero}\, 3 < \mathsf{not}(2 + 4)$
- $\mathsf{not}\, \mathsf{false}$
- $\mathsf{not}(x + 1)$
- $\mathsf{iszero}\, \mathsf{false}$
- $\mathsf{not}\, 7 * y$
- $7 < x + y$

Se observa que todas son expresiones sintácticamente correctas pero algunas de ellas no tienen sentido desde el punto de vista semántico, es decir, no es posible asignarles uno de los tipos posibles para el lenguaje que son Nat o Bool. Por ejemplo $\mathsf{not}(8+2)$ o $\mathsf{iszero}\, \mathsf{false}$. Además hay expresiones en donde no resulta claro si son válidas o no, por ejemplo $x + 2$ puede o no ser válida dependiendo de si x representa a un número o a un booleano, lo mismo sucede con $\mathsf{not}(\mathsf{iszero}\, x)$ o con $x < y + 1$.

1.1. Intérprete

Un intérprete para el lenguaje es simplemente una función que devuelve un natural o un booleano que resulta de evaluar una expresión e. La definición recursiva de esta función

$$\texttt{eval} : Exp \to \mathbb{N} \cup \mathcal{B}$$

donde $\mathbb{N} = \{0, 1, \ldots\}$ y $\mathcal{B} = \{t, f\}$ es:

```
eval n = n
eval true = t
eval false = f
eval (e1 + e2) = eval e1 + eval e2
eval (e1 * e2) = eval e1 * eval e2
eval (e1 < e2) = (eval e1) < (eval e2)
eval (not e) = ¬ (eval e)
eval (iszero e) = isz (eval e)
```

donde los símbolos del lado derecho $+, *, <$ denotan las operaciones usuales mientras que del lado izquierdo son únicamente símbolos que forman parte de las expresiones. Omitímos aquí el uso de variables por simplicidad en la presentación. Para agregar variables es necesario agregar un estado como argumento de entrada al intérprete, el cual se encargará de evaluar variables.

Se observa que la función de evaluación así definida resulta una función parcial puesto que la evaluación de expresiones como $(3 * 5) + \mathsf{not\ false}$ no tiene sentido ya que no sabemos sumar $15 + \mathsf{true}$.

Una manera de evitar estos errores es verificando previamente si la expresión es coherente para lo cual podemos usar la deducción natural.

1.2. Eliminación de expresiones semánticamente incorrectas

Sea Γ el conjunto de las siguientes fórmulas:

$$\begin{array}{l} Nat(n) \\ Bool(true) \\ Bool(false) \\ \forall x \forall y (Nat(x) \wedge Nat(y) \to Nat(x + y)) \\ \forall x \forall y (Nat(x) \wedge Nat(y) \to Nat(x * y)) \\ \forall x \forall y (Nat(x) \wedge Nat(y) \to Bool(x < y)) \\ \forall x \forall y (Bool(x) \to Bool(\mathsf{not}\, x)) \\ \forall x \forall y (Nat(x) \to Bool(\mathsf{iszero}\, x)) \end{array}$$

Podemos verificar si una expresión e es semánticamente correcta si es posible **clasificarla** como natural o booleano derivando el jucio $\Gamma \vdash Nat(e)$ o $\Gamma \vdash Bool(e)$ respectivamente. Por ejemplo

$$\Gamma \vdash \mathsf{not}(\mathsf{iszero}(2 + 3 * 4)) \qquad \Gamma \nvdash Bool(\mathsf{iszero}(3 * 7) + 4)$$

Además si hay variables debemos suponer de antemano si es que representan números o booleanos. Por ejemplo

$$\Gamma, Nat(x) \vdash Bool(x < 5) \qquad \Gamma, Bool(z) \nvdash Nat(z + 6)$$

Esta clase de verificaciones son exactamente lo que hace un sistema de tipos en lenguajes de programación.

2. Sistemas de tipos

Un sistema de tipos es un formalismo lógico que impone ciertas restricciones en la formación de programas de un lenguaje de programación. Las expresiones del lenguaje se clasifican mediante tipos que dictan como pueden usarse en combinación con otras expresiones.
Intuitivamente el tipo de una expresión predice la forma de su valor, por ejemplo, la expresión $e_1 + e_2$ donde e_1 y e_2 son expresiones numéricas debe ser una expresión numérica nuevamente. Por otra parte si intentamos sumar una expresión numérica con una expresión booleana digamos $\mathsf{true} + 7$ dicha expresión debe ser prohibida ya que genera un error de tipos [1].

El uso de sistemas de tipos en el diseño de un lenguaje de programación tiene las siguientes ventajas:

- Permite descubrir errores de programación tempranamente.
- Ofrece seguridad, un programa correctamente tipado no puede funcionar mal.
- Soporta abstracción, importante para la descripción de interfaces.
- Los tipos documentan un programa de manera mas simple y manejable que los comentarios.
- Los lenguajes tipados pueden implementarse de manera más clara y eficiente.

Para decidir si una expresión es semánticamente correcta o no, los lenguajes de programación emplean contextos de declaración de variables que involucran a un predicado binario infijo denotado $x : \mathsf{T}$, que relaciona a una variable x con un tipo T. Esta declaración significa que la variable x representa a un valor de tipo T. Un contexto es entonces un conjunto Γ de la forma $\Gamma = \{x_1 : \mathsf{T}_1, \ldots, x_n : \mathsf{T}_n\}$ donde todas las variables x_i son distintas y les corresponde un tipo T_i.

Por ejemplo si $\Gamma = \{x : \mathsf{Nat}, y : \mathsf{Bool}\}$ entonces las expresiones $x + 2$ y $\mathsf{not}\, y$ son aceptadas, mientras que $y < 2$ y $\mathsf{not}\, x$ son rechazadas para su evaluación. Para formalizar la verificación de tipos correctos se utiliza una relación ternaria que involucra a un contexto Γ, a una expresión cualquiera del lenguaje e y a un tipo T.
Dicha relación se escribe como $\Gamma \vdash e : \mathsf{T}$ y se lee como "la expresión e es de tipo T en el contexto Γ".

Esta relación se conoce como relación de tipado y se define recursivamente mediante las siguientes reglas de inferencia:

$$\frac{}{\Gamma, x : \mathsf{T} \vdash x : \mathsf{T}}\ (\textsc{Var}) \qquad \frac{}{\Gamma \vdash n : \mathsf{Nat}}\ (\textsc{TNum}) \qquad \frac{\Gamma \vdash e_1 : \mathsf{Nat} \qquad \Gamma \vdash e_2 : \mathsf{Nat}}{\Gamma \vdash e_1 + e_2 : \mathsf{Nat}}\ (\textsc{TSum})$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Nat} \qquad \Gamma \vdash e_2 : \mathsf{Nat}}{\Gamma \vdash e_1 * e_2 : \mathsf{Nat}}\ (\textsc{TProd}) \qquad \frac{}{\Gamma \vdash \mathsf{true} : \mathsf{Bool}}\ (\textsc{True}) \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{Bool}}\ (\textsc{False})$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Nat} \qquad \Gamma \vdash e_2 : \mathsf{Nat}}{\Gamma \vdash e_1 < e_2 : \mathsf{Bool}}\ (\textsc{Tmen}) \qquad \frac{\Gamma \vdash e : \mathsf{Nat}}{\Gamma \vdash \mathsf{iszero}\, e : \mathsf{Bool}}\ (\textsc{Tisz}) \qquad \frac{\Gamma \vdash e : \mathsf{Bool}}{\Gamma \vdash \mathsf{not}\, e : \mathsf{Bool}}\ (\textsc{Tnot})$$

Mediante estas reglas podemos cerciorarnos por ejemplo de que si $x : \mathsf{Nat}$ entonces $\mathsf{iszero}(x + 2) : \mathsf{Bool}$ como sigue:

1. $x : \mathsf{Nat} \vdash x : \mathsf{Nat}$ (Var)
2. $x : \mathsf{Nat} \vdash 2 : \mathsf{Nat}$ $(TNum)$

[1] Type-checking error

3. $x : \mathsf{Nat} \vdash x + 2 : \mathsf{Nat}$ $(TSum)$ $1, 2$

4. $x : \mathsf{Nat} \vdash \mathsf{iszero}(x + 2) : \mathsf{Nat}$ $(Tisz)$ 4

Veamos otro ejemplo, esta vez tipando la expresión $\mathsf{not}(2 + x < y)$ en el contexto $\Gamma = \{x : \mathsf{Nat}, y : \mathsf{Nat}\}$

1. $x : \mathsf{Nat}, y : \mathsf{Nat} \vdash x : \mathsf{Nat}$ (Var)

2. $x : \mathsf{Nat}, y : \mathsf{Nat} \vdash 2 : \mathsf{Nat}$ $(TNum)$

3. $x : \mathsf{Nat}, y : \mathsf{Nat} \vdash 2 + x : \mathsf{Nat}$ $(TSum)$ $1, 2$

4. $x : \mathsf{Nat}, y : \mathsf{Nat} \vdash y : \mathsf{Nat}$ (Var)

5. $x : \mathsf{Nat}, y : \mathsf{Nat} \vdash 2 + x < y : \mathsf{Bool}$ $(Tmen)$ $3, 4$

6. $x : \mathsf{Nat}, y : \mathsf{Nat} \vdash \mathsf{not}(2 + x < y) : \mathsf{Bool}$ $(Tneg)$ 5

3. Isomorfismo de Curry-Howard

Como se observa los sistemas de tipos son muy similares a los sistemas de deducción natural con contextos que ya hemos estudiado. De hecho los sistemas de deducción natural pueden verse, desde el punto de vista computacional, como un sistema de tipos para un *prototipo de lenguaje de programación funcional* mediante la llamada correspondencia de Curry-Howard, también conocida como paradigma de derivaciones como programas o proposiciones como tipos. La idea a grandes rasgos es:

- Las fórmulas de la lógica corresponden a tipos para un lenguaje de programación.
- Las pruebas o derivaciones en la lógica corresponden a los pasos que el verificador de tipos realiza para mostrar que cierta expresión o programa del lenguaje de programación está bien tipada.
- Por lo tanto, verificar si una prueba es correcta en un contexto dado corresponde a verificar si un programa tiene un tipo válido en el contexto dado.

4. Reglas de inferencia con codificación de pruebas

La relación de inferencia $\Gamma \vdash A$ se modifica anotando a cada prueba de un secuente $\Gamma \vdash A$ mediante una expresión t que que corresponde a un programa de un prototipo de lenguaje de programación, esta expresión t resulta ser también un **código de la prueba** cuyo secuente final es $\Gamma \vdash A$.

De esta manera obtenemos una relación $\Gamma \vdash t : A$ donde $\Gamma = \{x_1 : A_1, \ldots, x_n : A_n\}$ es un contexto de declaración de variables $x_i : A_i$ consideradas locales. En este caso no sólo suponemos una fórmula A_i sino que también suponemos que existe una prueba x_i de A_i. La nueva relación entre contextos Γ, expresiones o códigos t y fórmulas A, puede entenderse desde dos puntos de vista distintos:

- Lógicamente: $\Gamma \vdash t : A$ significa que la fórmula A es derivable a partir de las hipótesis Γ y t es un código de prueba para la derivación $\Gamma \vdash A$.
- Computacionalmente: $\Gamma \vdash t : A$ significa que t es un programa con tipo A cuyas variables locales están declaradas en Γ.

La nueva relación de inferencia se define recursivamente a partir de la regla de inicio:

$$\Gamma, x : \varphi \vdash x : \varphi \ \ (Hip)$$

mediante reglas de introducción y eliminación para cada conectivo

- Implicación: el conectivo $\to$ corresponde al tipo de funciones.

$$\frac{\Gamma, x : \varphi \vdash t : \psi}{\Gamma \vdash \mathsf{fun}(x : \varphi.t) : \varphi \to \psi} \ (\to \text{I}) \qquad \frac{\Gamma \vdash f : \varphi \to \psi \qquad \Gamma \vdash t : \varphi}{\Gamma \vdash f\, t : \psi} \ (\to \text{E})$$

- Conjunción: el conectivo $\wedge$ corresponde al tipo $\times$, es decir al producto cartesiano de dos tipos.

$$\frac{\Gamma \vdash t : \varphi \qquad \Gamma \vdash s : \psi}{\Gamma \vdash \langle t, s \rangle : \varphi \wedge \psi} \ (\wedge \text{ I}) \qquad \frac{\Gamma \vdash t : \varphi \wedge \psi}{\Gamma \vdash \mathtt{fst}\, t : \varphi} \ (\wedge_2 \text{ E}) \qquad \frac{\Gamma \vdash t : \varphi \wedge \psi}{\Gamma \vdash \mathtt{snd}\, t : \psi} \ (\wedge_1 \text{ E})$$

- Disyunción: el conectivo $\vee$ corresponde al tipo $+$, es decir a la unión disjunta de dos tipos.

$$\frac{\Gamma \vdash t : \varphi}{\Gamma \vdash \mathtt{inl}\, t : \varphi \vee \psi} \ (\vee_1 \text{ I}) \qquad \frac{\Gamma \vdash t : \psi}{\Gamma \vdash \mathtt{inr}\, t : \varphi \vee \psi} \ (\vee_2 \text{ I})$$

$$\frac{\Gamma \vdash r : \varphi \vee \psi \qquad \Gamma, x : \varphi \vdash s : \chi \qquad \Gamma, y : \psi \vdash t : \chi}{\Gamma \vdash \mathtt{case}\, r\, \mathtt{of}\, \mathtt{inl}\, x \Rightarrow s \mid \mathtt{inr}\, y \Rightarrow t \ : \chi} \ (\vee \text{ E})$$

- Falso: la constante de falsedad corresponde a un tipo vacío.

$$\frac{\Gamma \vdash r : \bot}{\Gamma \vdash \mathtt{abort}\, r : \varphi} \ (\bot \text{ E})$$

5. Ejemplos

En los siguientes ejemplos se trata de hallar el programa t que codifique la prueba correspondiente. Como ejercicios el lector debe justificar cada paso y/o agregar algunos pasos faltantes.

- $\vdash t : A \to A$
  1. $x : A \vdash x : A$
  2. $\vdash \mathsf{fun}(x : A.x) : A \to A$
- $\vdash t : A \to A \vee B$
  1. $x : A \vdash x : A$
  2. $x : A \vdash \mathsf{inl}\, x : A \vee B$
  3. $\vdash \mathsf{fun}(x : A.\, \mathsf{inl}\, x) : A \to A \vee B$
- $\vdash t : A \wedge B \to B$
  1. $x : A \wedge B \vdash x : A \wedge B$
  2. $x : A \wedge B \vdash \mathtt{snd} x : B$

3. $\vdash \mathsf{fun}(x : A \wedge B.\mathtt{snd}x) : A \wedge B \to B$

- $f : A \to B,\ g : B \to C \vdash t : A \to C$

1. $f : A \to B,\ g : B \to C, x : A \vdash f : A \to B$
2. $f : A \to B,\ g : B \to C, x : A \vdash g : B \to C$
3. $f : A \to B,\ g : B \to C, x : A \vdash x : A$
4. $f : A \to B,\ g : B \to C, x : A \vdash fx : B$
5. $f : A \to B,\ g : B \to C, x : A \vdash g(fx) : C$
6. $f : A \to B,\ g : B \to C \vdash \mathsf{fun}(x : A.g(fx)) : A \to C$

- $\vdash : t : B \to (A \to B)$

1. $x : B, y : A \vdash x : B$
2. $x : B \vdash \mathsf{fun}(y : A.x) : A \to B$
3. $\vdash \mathsf{fun}(x : B.\, \mathsf{fun}(y : A.x)) : B \to (A \to B)$

- $\vdash t : (A \wedge B \to C) \to (A \to B \to C)$

1. $f : A \wedge B \to C, x : A, y : B \vdash f : A \wedge B \to C$
2. $f : A \wedge B \to C, x : A, y : B \vdash x : A$
3. $f : A \wedge B \to C, x : A, y : B \vdash y : B$
4. $f : A \wedge B \to C, x : A, y : B \vdash \langle x, y\rangle : A \wedge B$
5. $f : A \wedge B \to C, x : A, y : B \vdash f\langle x, y\rangle : C$
6. $f : A \wedge B \to C, x : A \vdash \mathsf{fun}(y : B.f\langle x, y\rangle) : B \to C$
7. $f : A \wedge B \to C \vdash \mathsf{fun}\,(x : A.\, \mathsf{fun}(y : B.f\langle x, y\rangle)) : A \to B \to C$
8. $\vdash \mathsf{fun}\left(f : A \wedge B \to C.\, \mathsf{fun}\,(x : A.\, \mathsf{fun}(y : B.f\langle x, y\rangle))\right) : (A \wedge B \to C) \to A \to B \to C$

- $f : A \to B \to C \vdash t : A \wedge B \to C$

1. $f : A \to B \to C, x : A \wedge B \vdash f : A \to B \to C$
2. $f : A \to B \to C, x : A \wedge B \vdash f : A \to B \to C$
3. $f : A \to B \to C, x : A \wedge B \vdash x : A \wedge B$
4. $f : A \to B \to C, x : A \wedge B \vdash \mathtt{fst}x : A$
5. $f : A \to B \to C, x : A \wedge B \vdash \mathtt{snd}x : B$
6. $f : A \to B \to C, x : A \wedge B \vdash f(\mathtt{fst}x) : B \to C$
7. $f : A \to B \to C, x : A \wedge B \vdash f(\mathtt{fst}x)(\mathtt{snd}x) : C$
8. $f : A \to B \to C \vdash \mathsf{fun}\,(x : A \wedge B.f(\mathtt{fst}x)(\mathtt{snd}x)) : A \wedge B \to C$

- $f : A \vee B \to C \vdash t : A \to C$

1. $f : A \vee B \to C, x : A \vdash f : A \vee B \to C$
2. $f : A \vee B \to C, x : A \vdash x : A$
3. $f : A \vee B \to C, x : A \vdash \mathsf{inl}\, x : A \vee B$

4. $f : A \vee B \to C, x : A \vdash f(\mathsf{inl}\, x) : C$
5. $f : A \vee B \to C \vdash \mathsf{fun}(x : A.f(\mathsf{inl}\, x)) : A \to C$

- $x : A \vee B, f : A \to C, g : B \to C \vdash t : C$

1. $x : A \vee B, f : A \to C, g : B \to C \vdash x : A \vee B$
2. $x : A \vee B, f : A \to C, g : B \to C, y : A \vdash fy : C$
3. $x : A \vee B, f : A \to C, g : B \to C, z : B \vdash gz : C$
4. $x : A \vee B, f : A \to C, g : B \to C \vdash (\ \mathtt{case}\ x\ \mathtt{of}\ \mathsf{inl}\, y \Rightarrow fy \mid \mathsf{inr}\, z \Rightarrow gz) : C$

- $f : A \to B, g : C \to D, x : A \vee C \vdash t : B \vee D$

1. $f : A \to B, g : C \to D, x : A \vee C \vdash x : A \vee C$
2. $f : A \to B, g : C \to D, x : A \vee C, y : A \vdash \mathsf{inl}(fy) : B \vee D$
3. $f : A \to B, g : C \to D, x : A \vee C, z : C \vdash \mathsf{inr}(gz) : B \vee D$
4. $f : A \to B, g : C \to D, x : A \vee C \vdash (\ \mathtt{case}\ x\ \mathtt{of}\ \mathsf{inl}\, y \Rightarrow \mathsf{inl}(fy) \mid \mathsf{inr}\, z \Rightarrow \mathsf{inr}(gz)) : B \vee D$

- $x : (A \vee C) \wedge (B \to C), f : (A \to B) \vdash t : C$

1. $x : (A \vee C) \wedge (B \to C), f : (A \to B) \vdash \mathtt{fst}x : A \vee C$
2. $x : (A \vee C) \wedge (B \to C), f : (A \to B) \vdash \mathtt{snd}x : B \to C$
3. $x : (A \vee C) \wedge (B \to C), f : (A \to B), y : A \vdash fy : B$
4. $x : (A \vee C) \wedge (B \to C), f : (A \to B), y : A \vdash (\mathtt{snd}x)(fy) : C$
5. $x : (A \vee C) \wedge (B \to C), f : (A \to B), z : C \vdash z : C$
6. $x : (A \vee C) \wedge (B \to C), f : (A \to B) \vdash (\ \mathtt{case}\ x\ \mathtt{of}\ \mathsf{inl}\, y \Rightarrow \mathsf{inl}(\mathtt{snd}x)(fy) \mid \mathsf{inr}\, z \Rightarrow z) : C$

- $x : A \wedge (B \vee C) \vdash t : (A \wedge B) \vee (A \wedge C)$

1. $x : A \wedge (B \vee C) \vdash \mathtt{fst}x : A$
2. $x : A \wedge (B \vee C) \vdash \mathtt{snd}x : B \vee C$
3. $x : A \wedge (B \vee C), y : B \vdash \langle \mathtt{fst}x, y \rangle : A \wedge B$
4. $x : A \wedge (B \vee C), y : B \vdash \mathsf{inl}\langle \mathtt{fst}x, y \rangle : (A \wedge B) \vee (A \wedge C)$
5. $x : A \wedge (B \vee C), z : C \vdash \mathsf{inr}\langle \mathtt{fst}x, z \rangle : (A \wedge B) \vee (A \wedge C)$
6. $x : A \wedge (B \vee C) \vdash (\ \mathtt{case}\ \mathtt{snd}x\ \mathtt{of}\ \mathsf{inl}\, y \Rightarrow \mathsf{inl}\langle \mathtt{fst}x, y \rangle \mid \mathsf{inr}\, z \Rightarrow \mathsf{inr}\langle \mathtt{fst}x, z \rangle) : (A \wedge B) \vee (A \wedge C)$

- $x : (A \wedge B) \vee (A \wedge C) \vdash t : A \wedge (B \vee C)$

1. $x : (A \wedge B) \vee (A \wedge C), y : A \wedge B \vdash \mathsf{inl}(\mathtt{snd}y) : B \vee C$
2. $x : (A \wedge B) \vee (A \wedge C), y : A \wedge B \vdash \langle \mathtt{fst}y, \mathsf{inl}(\mathtt{snd}y) \rangle : A \wedge (B \vee C)$
3. $x : (A \wedge B) \vee (A \wedge C), z : A \wedge C \vdash \langle \mathtt{fst}z, \mathsf{inr}(\mathtt{snd}z) \rangle : A \wedge (B \vee C)$
4. $x : (A \wedge B) \vee (A \wedge C) \vdash (\ \mathtt{case}\ x\ \mathtt{of}\ \mathsf{inl}\, y \Rightarrow \langle \mathtt{fst}y, \mathsf{inl}(\mathtt{snd}y) \rangle \mid \mathsf{inr}\, z \Rightarrow \langle \mathtt{fst}z, \mathsf{inr}(\mathtt{snd}z) \rangle : A \wedge (B \vee C)$

- $x : A \vee B, f : B \vee C \to D, g : A \to C \vdash t : D$

1. $x : A \vee B, f : B \vee C \to D, g : A \to C, y : A \vdash \mathsf{inr}(gy) : B \vee C$
2. $x : A \vee B, f : B \vee C \to D, g : A \to C, y : A \vdash f\, \mathsf{inr}(gy) : D$
3. $x : A \vee B, f : B \vee C \to D, g : A \to C, z : B \vdash f(\mathsf{inl}\, z) : D$
4. $x : A \vee B, f : B \vee C \to D, g : A \to C \vdash (\ \mathtt{case}\ x\ \mathtt{of}\ \mathsf{inl}\, y \Rightarrow f\, \mathsf{inr}(gy) \mid \mathsf{inr}\, z \Rightarrow f(\mathsf{inl}\, z) : D$

6. Pruebas redundantes

- Dadas las derivaciones $\Gamma \vdash \mathsf{fun}(x : A.e) : A \to B$ y $\Gamma \vdash r : A$ construir una derivación de $\Gamma \vdash ? : B$.
  - La solución que salta a la vista es $\Gamma \vdash \mathsf{fun}(x : A.e)\, r : B$
  - Sin embargo de las reglas y la prueba dada de $A \to B$, sabemos que previamente debio obtenerse la prueba $\Gamma, x : A \vdash e : B$ puesto que la regla usada fue $(\to I)$. Ahora bien, obsérvese que como ya tenemos una prueba $\Gamma \vdash r : A$, no es necesario suponer $x : A$ puesto que en la prueba codificada por e podemos sustituir cada prueba x de A por la prueba dada r, obteniendo así una prueba de B sin necesidad de usar el modus ponens, a saber $\Gamma \vdash e[x := r] : B$.
- Dada la derivación $\Gamma \vdash \langle e_1, e_2 \rangle : A \wedge B$ construir una prueba $\Gamma \vdash ? : A$.
  - La respuesta más inmediata es $\Gamma \vdash \mathtt{fst}\langle e_1, e_2 \rangle : A$.
  - Pero obsérvese que para construir la prueba codificada por $\mathtt{fst}\langle e_1, e_2 \rangle$ antes tuvo que construirse una prueba codificada por e_1 que tiene que ser $\Gamma \vdash e_1 : A$ puesto que la regla utilizada fue $(\wedge I)$.

De lo anterior se observa que las primeras pruebas son redundantes y podemos simplificarlas como sigue:

$$\mathsf{fun}(x : A.e)\, r \text{ se simplifica a } e[x := r]$$

$$\mathtt{fst}\langle e_1, e_2 \rangle \text{ se simplifica a } e_1.$$

De la misma manera podemos justificar las siguientes simplificaciones

$$\mathtt{snd}\langle e_1, e_2 \rangle \text{ se simplifica a } e_2.$$

$$\mathtt{case}\,(\mathsf{inl}\, r)\,\mathtt{of}\,\mathtt{inl}\, x \Rightarrow s \mid \mathtt{inr}\, y \Rightarrow t \text{ se simplifica a } s[x := r]$$

$$\mathtt{case}\,(\mathsf{inr}\, r)\,\mathtt{of}\,\mathtt{inl}\, x \Rightarrow s \mid \mathtt{inr}\, y \Rightarrow t \text{ se simplifica a } t[x := r]$$

Estas reglas de simplificación pueden utilizarse como reglas de evaluación de las expresiones de un lenguaje de programación. En conclusión las pruebas redundantes de la lógica y su simplificación corresponden a un proceso de evaluación de expresiones en un lenguaje de programación.

6.1. Compilación de expresiones aritméticas y booleanas

Presentamos un ejemplo de compilación del lenguaje de expresiones aritméticas y booleanas a una máquina abstracta de pila. Nos interesa en particular formalizar la prueba de la correctud del proceso de compilación mediante deducción natural.

- Especificación del lenguaje objeto: se trata de un lenguaje de instrucciones primitivas

  $$i ::= n \mid t \mid f \mid + \mid * \mid < \mid \mathsf{not} \mid \mathsf{iszero}$$

  de manera que una instrucción es un valor natural o booleano o bien un operador del lenguaje fuente de expresiones aritméticas y booleanas
- Programas: un programa es una lista[2] de instrucciones $p = [i_1, \ldots, i_n]$
- Memoria: la memoria es una lista de valores $s = [v_1, \ldots, v_n]$ donde $v_i \in \{n, t, f\}$

[2]En realidad es una pila y elegimos implementarla como una lista, lo mismo sucede con la memoria

- Ejecución de una instrucción: la función de ejecución de una instrucción `ej` recibe una instrucción `i` y una memoria `s` devolviendo la memoria resultante al ejecutar la instrucción.

```
ej n s = (n:s)
ej true s = (t:s)
ej false s = (f:s)
ej + (v1:v2:s) = (v1+v2) : s
ej * (v1:v2:s) = (v1*v2) : s
ej < (v1:v2:s) = (v1<v2) : s
ej not (v:s) = (¬ v) : s
ej iszero (v:s) = (isz v) : s
```

  donde del lado derecho de las ecuaciones nos referimos a las operaciones binarias $+, * :: \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, $<: \mathbb{N} \times \mathbb{N} \to \mathcal{B}$ y a las operaciones unarias $\neg : \mathcal{B} \to \mathcal{B}$, $isz : \mathbb{N} \to \mathcal{B}$

- Ejecución de un programa: ejecutar un programa p en la memoria dada s devuelve la memoria obtenida al ejecutar en orden todas instrucciones de p.

```
ejp nil s = s
ejp (i:p) s = ejp p (ej i s)
```

- Compilación de una expresión en un programa: el proceso de compilación convierte una expresión e en un programa.

```
comp n = [n]
comp true = [t]
comp false = [f]
comp (e1+e2) = comp e2 ++ comp e1 ++ [+]
comp (e1*e2) = comp e2 ++ comp e1 ++ [*]
comp (e1<e2) = comp e2 ++ comp e1 ++ [<]
comp (not e) = comp e ++ [not]
comp (iszero e) = comp e ++ [iszero]
```

  donde $+\!\!+$ es la función de concatenación de listas. ¿ Porqué en los casos de operadores binarios queda el programa resultado de compilar la segunda expresión e_2 al inicio de la lista?

- Correctud del compilador: la memoria resultado de ejecutar el programa p obtenido al compilar la expresión e con la memoria vacía coincide con la memoria cuyo único valor es la evaluación de la expresión e

```
∀ e (ejp (comp e) nil = [eval e])
```

  Por supuesto en todas las definiciones anteriores estamos suponiendo que las expresiones de entrada son coherentes en el sentido de que cualquier operación estará bien definida. Es decir, estamos suponiendo que el proceso de verificación previa fue hecho por el sistema de tipos. Por lo que el cuantificador $\forall e$ se refiere únicamente a todas las expresiones semánticamente correctas.

6.2. Prueba de la correctud (informal)

Para probar la correctud del compilador vamos a probar algo más general:

$$\forall\ \texttt{e}\ \forall\ \texttt{p}\ \forall\ \texttt{s}\ (\texttt{ejp}\ (\ \texttt{comp e}\ +\!\!+\ \texttt{p})\ \texttt{s} = \texttt{ejp p (eval e:s)})$$

de esta propiedad la correctud del compilador resulta un corolario tomando $p = nil, s = nil$.

La prueba es por inducción sobre las expresiones. Analizamos aquí un caso base y dos casos inductivos:

- Base $e = n$: P.D. $\forall p \forall s(\texttt{ejp}\ (\texttt{comp}\, n +\!\!+ p)\ s = \texttt{ejp}\ p\ (\texttt{eval}\, n : s))$.

$$\begin{array}{rcl} \texttt{ejp}\ (\texttt{comp}\, n +\!\!+ p)\ s & = & \texttt{ejp}\ ([n] +\!\!+ p)\ s \\ & = & \texttt{ejp}\ (n : p)\ s \\ & = & \texttt{ejp}\ p\ (\texttt{ej}\, n\ s) \\ & = & \texttt{ejp}\ p\ (n : s) \\ & = & \texttt{ejp}\ p\ (\texttt{eval}\, n : s) \end{array}$$

- Paso inductivo $e = e_1 * e_2$.
  - I.H.1.: $\forall p \forall s(\texttt{ejp}\ (\texttt{comp}\, e_1 +\!\!+ p)\ s = \texttt{ejp}\ p\ (\texttt{eval}\, e_1 : s))$
  - I.H.2.: $\forall p \forall s(\texttt{ejp}\ (\texttt{comp}\, e_2 +\!\!+ p)\ s = \texttt{ejp}\ p\ (\texttt{eval}\, e_2 : s))$

  Queremos demostrar que:

$$\forall p \forall s(\texttt{ejp}\ (\texttt{comp}(e_1 * e_2) +\!\!+ p)\ s = \texttt{ejp}\ p\ (\texttt{eval}(e_1 * e_2) : s))$$

$$\begin{array}{rcl} \texttt{ejp}\ (\texttt{comp}(e_1 * e_2) +\!\!+ p)\ s & = & \texttt{ejp}\ ((\texttt{comp}\, e_2 +\!\!+ \texttt{comp}\, e_1 +\!\!+ [*]) +\!\!+ p)\ s \\ & = & \texttt{ejp}\ ((\texttt{comp}\, e_2 +\!\!+ (\texttt{comp}\, e_1 +\!\!+ [*] +\!\!+ p))\ s \\ & =_{Asoc +\!\!+} & \texttt{ejp}\ ((\texttt{comp}\, e_2 +\!\!+ (\texttt{comp}\, e_1 +\!\!+ [*] +\!\!+ p))\ s \\ & =_{I.H.2} & \texttt{ejp}\ (\texttt{comp}\, e_1 +\!\!+ [*] +\!\!+ p))\ (\texttt{eval}\, e_2 : s) \\ & =_{Asoc +\!\!+} & \texttt{ejp}\ (\texttt{comp}\, e_1 +\!\!+ ([*] +\!\!+ p))\ (\texttt{eval}\, e_2 : s) \\ & =_{I.H.1} & \texttt{ejp}\ ([*] +\!\!+ p)\ (\texttt{eval}\, e_1 : \texttt{eval}\, e_2 : s) \\ & = & \texttt{ejp}\ (* : p)\ (\texttt{eval}\, e_1 : \texttt{eval}\, e_2 : s) \\ & = & \texttt{ejp}\ p\ (\texttt{ej}\ [*]\ (\texttt{eval}\, e_1 : \texttt{eval}\, e_2 : s)) \\ & = & \texttt{ejp}\ p\ (\texttt{eval}\, e_1 * \texttt{eval}\, e_2 : s) \\ & = & \texttt{ejp}\ p\ (\texttt{eval}(e_1 * e_2) : s) \end{array}$$

- Paso inductivo $e = \texttt{not}\, e_1$.
  - I.H: $\forall p \forall s(\texttt{ejp}\ (\texttt{comp}\, e_1 +\!\!+ p)\ s = \texttt{ejp}\ p\ (\texttt{eval}\, e_1 : s))$

  Queremos demostrar que:

$$\forall p \forall s(\texttt{ejp}\ (\texttt{comp}(\texttt{not}\, e_1) +\!\!+ p)\ s = \texttt{ejp}\ p\ (\texttt{eval}(\texttt{not}\, e_1) : s))$$

$$
\begin{array}{lll}
\texttt{ejp}\ (\texttt{comp}(\mathsf{not}\, e_1) +\!\!+ p)\ s & = & \texttt{ejp}\ ((\texttt{comp}\, e_1 +\!\!+ [\neg]) +\!\!+ p)\ s \\
& =_{Asoc +\!\!+} & \texttt{ejp}\ ((\texttt{comp}\, e_1 +\!\!+ ([\neg] +\!\!+ p))\ s \\
& =_{I.H} & \texttt{ejp}\ ([\neg] +\!\!+ p)\ (\texttt{eval}\, e_1 : s) \\
& = & \texttt{ejp}\ (\neg : p)\ (\texttt{eval}\, e_1 : s) \\
& = & \texttt{ejp}\ p\ (\texttt{ej}\ [\neg]\ (\texttt{eval}\, e_1 : s)) \\
& = & \texttt{ejp}\ p\ (\neg(\texttt{eval}\, e_1) : s) \\
& = & \texttt{ejp}\ p\ (\texttt{eval}(\mathsf{not}\, e_1) : s)
\end{array}
$$

A continuación damos una idea de la formalización de esta prueba en el sistema de deducción natural.

6.3. Bosquejo de la prueba formal (dentro del sistema de deducción natural)

Queremos probar que $\Gamma \vdash \forall e P(e)$ donde Γ es un conjunto de premisas adecuado y $P(e)$ es la fórmula que formaliza la propiedad requerida.

- La fórmula a probar es $P(e)$ definida como

$$P(e) =_{def} \forall p \forall s(\, \texttt{ejp}\ (\texttt{comp}\, e +\!\!+ p)\ s = \texttt{ejp}\ p\ (\texttt{eval}\, e : s))$$

- Γ se define como la siguiente unión de conjuntos de fórmulas:

$$\Gamma = \{\mathcal{I}_P\} \cup \mathbb{F} \cup \mathbb{L} \cup \mathbb{E}$$

  donde

- $\mathcal{I}_P$ es el principio de inducción para expresiones para el caso de P:

$$
\begin{array}{lll}
\mathcal{I}_P & =_{def} & P(n) \wedge P(\mathsf{true}) \wedge P(\mathsf{false}) \wedge \\
& & \forall e_1 \forall e_2 (P(e1) \wedge P(e_2) \to P(e_1 + e_2)) \wedge \\
& & \forall e_1 \forall e_2 (P(e1) \wedge P(e_2) \to P(e_1 * e_2)) \wedge \\
& & \forall e_1 \forall e_2 (P(e1) \wedge P(e_2) \to P(e_1 < e_2)) \wedge \\
& & \forall e (P(e) \to P(\mathsf{not}\, e)) \wedge \\
& & \forall e (P(e) \to P(\mathsf{iszero}\, e)) \\
& & \to \forall e P(e)
\end{array}
$$

- $\mathbb{F}$ consta de las cerraduras universales de las ecuaciones que definen a las funciones $\texttt{eval}, \texttt{ej}, \texttt{ejp}, \texttt{comp}$.

$$
\begin{array}{lll}
\mathbb{F} & = \{ & \forall n (\texttt{eval}\, n = n), \ldots, \\
& & \forall e_1 \forall e_2 (\, \texttt{eval}(e1 + e2) = \texttt{eval}\, e1 + \texttt{eval}\, e2), \ldots \\
& & \forall v1 \forall v2 \forall s (\, \texttt{ej} < (v1 : v2 : s) = (v1 < v2) : s), \ldots \\
& & \vdots \\
& & \forall e (\, \texttt{comp}(\mathsf{iszero}\, e) = \texttt{comp}\, e +\!\!+ [\mathsf{iszero}]) \\
& \} &
\end{array}
$$

- $\mathbb{L}$ consta de las propiedades requeridas de las operaciones de listas:
  - Concatenación de lista unitaria: $\forall x \forall \ell ([x] +\!\!+ \ell = x : \ell)$.

- Asociatividad de la concatenación:

$$\forall x \forall y \forall z (x +\!\!+ (y +\!\!+ z) = (x +\!\!+ y) +\!\!+ z)$$

- $\mathbb{E}$ contiene los llamados axiomas de igualdad:
  - Reflexividad: $\forall x(x = x)$
  - Simetría $\forall x \forall y(x = y \to y = x)$
  - Transitividad: $\forall x \forall y \forall z(x = y \wedge y = z \to x = z)$
  - Sustitución (Leibniz): $\forall x \forall y(x = y \wedge A(x) \to A(y))$ donde A es una fórmula cualquiera[3]

  Alternativamente en vez de usar estas fórmulas directamente para derivar nuevas fórmulas o ecuaciones podemos utilizar la lógica ecuacional discutida antes, es decir, usar las siguientes reglas: para la igualdad son admisibles y en adelante usar directamente estas reglas adicionales para la igualdad: aquí Γ es cualquier contexto

$$\Gamma \vdash t = t\ (Refl) \qquad \frac{\Gamma \vdash s = t}{\Gamma \vdash t = s}\ (Sim) \qquad \frac{\Gamma \vdash s = t \quad \Gamma \vdash t = r}{\Gamma \vdash s = r}\ (Trn)$$

$$\frac{\Gamma \vdash s = t \quad \Gamma \vdash A(s)}{\Gamma \vdash A(t)}\ (Rewrite)$$

donde A es cualquier fórmula.

A continuación bosquejamos la prueba formal $\Gamma \vdash \forall e P(e)$.

- Para esto basta probar el antecedente del axioma de inducción $\mathcal{I}_P$ y usar modus ponens.
- Como dicho antecedente es una conjunción basta probar cada parte por separado y usar la regla $(\wedge I)$.
- Cada parte se prueba formalizando las pruebas informales, dado que el razonamiento ecuacional está permitido por las hipótesis de $\mathbb{E} \subseteq \Gamma$ y por las propiedades de listas $\mathbb{L} \subseteq \Gamma$. Como ejemplo formalizemos el caso base para números naturales:
  - Base P.D. $\Gamma \vdash P(n)$, es decir, $\Gamma \vdash \forall p \forall s(\texttt{ejp}\ (\texttt{comp}\, n +\!\!+ p)\ s = \texttt{ejp}\ p\ (\texttt{eval}\, n : s))$.

1. $\Gamma \vdash$	$\texttt{ejp}\ (\texttt{comp}\, n +\!\!+ p)\ s = \texttt{ejp}\ (\texttt{comp}\, n +\!\!+ p)\ s$	$(Refl)$
2. $\Gamma \vdash$	$\texttt{ejp}\ (\texttt{comp}\, n +\!\!+ p)\ s = \texttt{ejp}\ ([n] +\!\!+ p)\ s$	$(Rewrite)$ 1 $def.\, \texttt{comp}\, n$
3. $\Gamma \vdash$	$\texttt{ejp}\ (\texttt{comp}\, n +\!\!+ p)\ s = \texttt{ejp}\ (n : p)\ s$	$(Rewrite)$ 2 *prop. de listas*
4. $\Gamma \vdash$	$\texttt{ejp}\ (\texttt{comp}\, n +\!\!+ p)\ s = \texttt{ejp}\ p\ (\texttt{ej}\, n\ s)$	$(Rewrite)$ 3 $def.\, \texttt{ejp}$
5. $\Gamma \vdash$	$\texttt{ejp}\ (\texttt{comp}\, n +\!\!+ p)\ s = \texttt{ejp}\ p\ (n : s)$	$(Rewrite)$ 4 $def.\, \texttt{ej}$
6. $\Gamma \vdash$	$\texttt{ejp}\ (\texttt{comp}\, n +\!\!+ p)\ s = \texttt{ejp}\ p\ (\texttt{eval}\, n : s)$	$(Rewrite)$ 5 $def.\, \texttt{eval}$
7. $\Gamma \vdash$	$\forall s(\texttt{ejp}\ (\texttt{comp}\, n +\!\!+ p)\ s = \texttt{ejp}\ p\ (\texttt{eval}\, n : s))$	$(\forall I)$ 6, $s \notin FV(\Gamma)$
8. $\Gamma \vdash$	$\forall p \forall s(\texttt{ejp}\ (\texttt{comp}\, n +\!\!+ p)\ s = \texttt{ejp}\ p\ (\texttt{eval}\, n : s))$	$(\forall I)$ 7, $p \notin FV(\Gamma)$

[3] Basta con agregar la instancia de esta fórmula para las fórmulas A que necesitemos

6.4. ¿Para qué una prueba formal?

Una prueba formal como la que acabamos de bosquejar no es una argumentación en algún lenguaje natural, como el español, o bien semiformal como la mezcla de español y matemáticas, la cual es susceptible de errores díficiles de encontrar en casos de aplicación real. Por ejemplo la construcción de un compilador correcto (altamente confiable) de un lenguaje real como C (ver `http://compcert.inria.fr`). Las pruebas formales constan de un proceso de cálculo preciso que sigue reglas bien definidas, en nuestro caso las reglas de deducción natural. Una ventaja de construir esta clase de pruebas es que debido a su naturaleza precisa son rigurosas, libres de ambigüedades y susceptibles de construirse y verificarse mecánicamente por programas llamados asistentes de prueba como Coq (ver `http://coq.inria.fr`). Esta verificación a la vez produce una certificación que garantiza que ciertas propiedades se cumplen, como por ejemplo la correctud de un compilador. Esto es deseable en sistemas cuyo funcionamiento incorrecto causa errores graves como sistemas de radiación para enfermos de cancer y sistemas de navegación de vuelo.
El uso de sistemas lógicos para desarrollar aplicaciones con propiedades certificadas mediante prueba formales ha generado una nueva área dentro de las ciencias de la computación conocida como Métodos Formales.