

Introducción a la Verificación Formal

Clase 1 – 26/08/2025

Administrivia

- Única materia previa: ALP
 - MUY recomendada: IS1
 - Siempre pueden cursar. Para rendir tienen que tener ALP aprobado.
- Nos vemos todos los **jueves a las 3PM.**
 - Las clases se graban. En teoría tienen acceso con la invitación.
 - Luego de cada clase hay una parte del libro para leer y ejercicios para hacer.
- Final de materia: TP final a (semi)elección, a definir más adelante
- Clases invitadas: vayan viendo si les interesa algo en particular

Software Confiable

- Objetivamente, la ingeniería de software es la peor ingeniería
 - Sin garantías reales
 - Fallos frecuentes (evidentemente)
 - Poca sistematización
 - Testing como método de calidad fundamental
- Por otro lado, el *bloat* es innegable
 - La containerización, chequeos dinámicos, etc, no son gratis

```
hashOut.data = hashOut + SSL_SHA1_DIGEST_LEN;
hashOut.length = SSL_SHA1_DIGEST_LEN;
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;

if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

err = sslRawVerify(ctx,
                  ctx->peerPubKey,
                  dataToSign,
                  dataToSignLen,
                  signature,
                  /* plaintext */
                  /* plaintext length */);
```

--uu-:---F1 sslKeyExchange.c 30% L602 (C/l Abbrev Isearch)-----

[Apple finally fixes 'gotofail' OS X security hole - CNET](#)

Salón de la Fama de Bugs

- [1994: Bug FDIV del Pentium](#)
 - Error de cálculo en divisiones de punto flotante, sin detección por >1 año.
- [1996: Vuelo Ariane V88](#)
 - Overflow entero (+ otros factores) causa destrucción del cohete + 4 satélites.
- [2012: Heartbleed](#)
 - Falla de validación de entrada + bug de memoria filtra secretos remotamente.
- [2014: Hackeo y Bancarrota de MtGox](#)
 - Maleabilidad del protocolo de Bitcoin (+ otros factores) lleva a “robo” de 650,000 BTC.
- 2018: [Spectre](#) y [Meltdown](#)
 - Side channel en el procesador afecta aislación entre procesos. (Todavía estamos pagando las mitigaciones con pérdida de perf.)

“Prendimos fuego la casa y ¿adivinen qué?
Nosotros somos los bomberos.”
- Charles E. Leiserson

¿Qué hacemos?






- Gran parte de la industria se enfoca en mejorar la calidad del software
 - Testing sistematizado (fuzzing, property-based testing)
 - Testing de penetración
 - Analizadores estáticos
 - Chequeos dinámicos
 - Nuevos lenguajes (funcionales, DSLs, Rust)
- Esto tiene algún éxito
 - Ver este [informe de la casa blanca](#) (Feb 2024) sugiriendo el uso de lenguajes memory-safe y métodos formales.
- Pero, estos métodos **no dan garantías de correctitud**





Formal Methods

Even if engineers build with memory safe programming languages and memory safe chips, one must think about the vulnerabilities that will persist even after technology manufacturers take steps to eliminate the most prevalent classes. Given the complexities of code, testing is a necessary but insufficient step in the development process to fully reduce vulnerabilities at scale. If correctness is defined as the ability of a piece of software to meet a specific security requirement, then it is possible to demonstrate correctness using mathematical techniques called *formal methods*. These techniques, often used to prove a range of software outcomes, can also be used in a cybersecurity context and are viable even in complex environments like space. While formal methods have been studied for decades, their deployment remains limited; further innovation in approaches to make formal methods widely accessible is vital to accelerate broad adoption. Doing so enables formal methods to serve as another powerful tool to give software developers greater assurance that entire classes of vulnerabilities, even beyond memory safety bugs, are absent.

Métodos Formales

- Los métodos formales son formas **estáticas** de analizar software basadas en sistemas formales de lógica.
 - Formal: basado en reglas de inferencia. E.g. Si “ $P \rightarrow Q$ ” y “ P ” entonces “ Q ”. No puede apelarse a conocimiento sobre “qué” son P/Q ni a razonamientos fuera del sistema.
- Hay dos campos principales
 - Sistemas de tipos Fuertes (Tipado Dependiente, IFC, ...)   
 - El lenguaje de programación tiene un sistema de tipos que permite expresar propiedades más complejas
 - E.g. “**f** es una función monótona” “**n** es primo”
 - El sistema de tipos luego acepta o rechaza el programa
 - Lógicas de Programa (e.g. Lógica de Hoare, Lógica de Separación, ...)  
 - El programador escribe el programa, tal vez con anotaciones, y la herramienta computa *condiciones de verificación* (VCs) que implican la correctitud.
 - El programador luego intenta demostrar esas VCs, ya sea con métodos manuales o automáticos, o ajustando el programa y sus anotaciones
- En ambos casos el programa se desarrolla junto a su prueba
 - La prueba se **chequea por máquina**
- Se eliminan **clases** enteras de bugs posibles, con certeza
- Sigue siendo “caro”: requiere expertos y mucho tiempo

Algunos éxitos en Verificación Formal

- Teorema de los cuatro colores
 - Demostrado por Wolfgang Haken y Kenneth Appel (padre del Appel que conocen) en la primera prueba matemática por computadora. [Video](#)
- Feit-Thompson: teorema sobre grupos, ~250 páginas
- [Mathlib](#): librería de matemática en Lean
- [CompCert](#): Compilador de C verificado
- [seL4](#): Microkernel
- [DeepSpec](#)
- [HACL*](#): Criptografía verificada 
- [EverParse](#): Parsers y serializadores verificados 

Program Proofs in F* for Billions of Unsuspecting Users

Automated parsing
untrusted data with proofs
in Hyper-V/VMSwitch



Cloud infrastructure

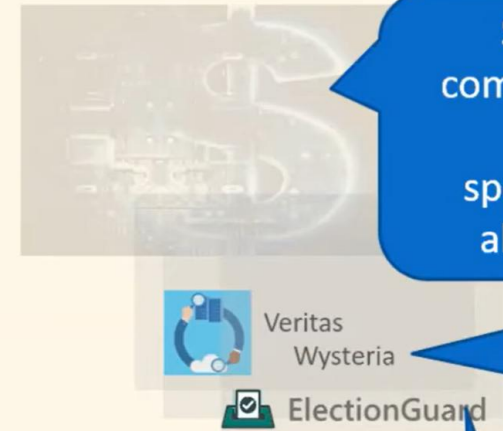
Verified Merkle trees
for Enterprise
blockchains

Verified
cryptography in the
Linux kernel



Secure communications

Quic transport,
MSQuic in Windows,
Verified crypto in Firefox,
mbedTLS,
Signal in Wasm,
Wireguard, ...

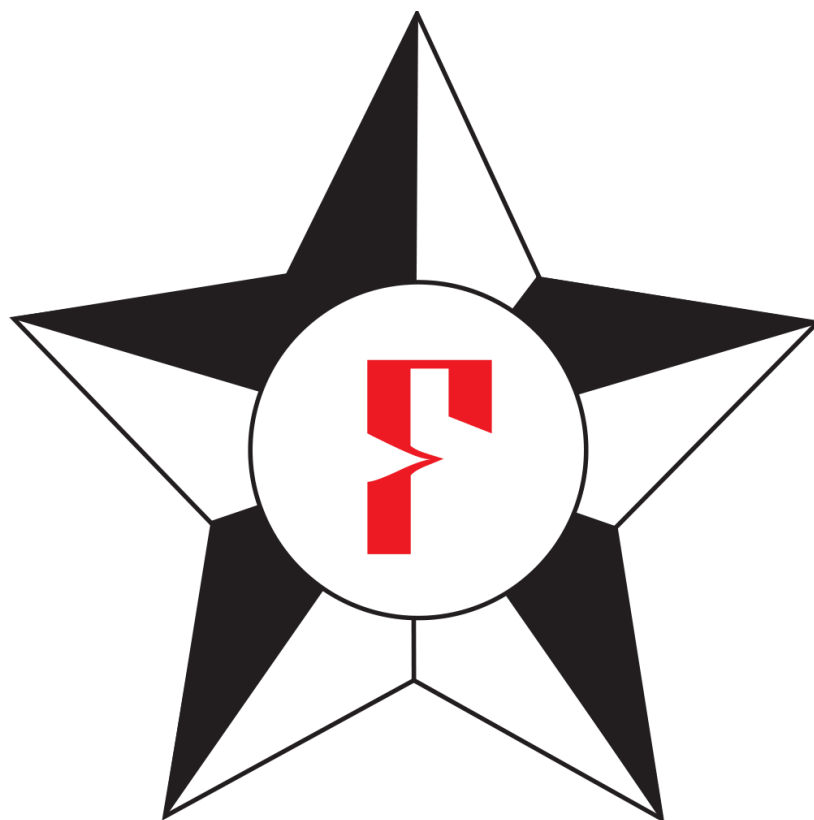


High-value domains:
Fintech, verifiable
computing, ...

Some Fintech
companies rewriting
core trading
specifications and
algorithms in F*

Correctness of
Verifiable
computations

Core crypto in
ElectionGuard



F*: Programación orientada a las pruebas



- F* es un lenguaje funcional de la familia ML
 - Estricto (lo opuesto a lazy)
 - Alto orden (lambdas)
 - Polimórfico, con inferencia de tipos (Hindley-Milner)
- Lo nuevo:
 - Refinamientos
 - Tipos dependientes
 - Descarga de VCs por SMT solver
 - Extensionalidad
 - Sistema de efectos

```
(** The type of non-negative integers *)  
type nat = i: int{i >= 0}
```

Sintaxis básica

- Similar a OCaml
 - **val**: declaración de tipo
 - **let**: definición
 - **list a**: tipo de listas de **a**
 - **match** para pattern matching (en lugar de ecuaciones como en Haskell/Agda)
- Recursión explícita con **let rec**
 - El archivo va en orden.
- No hay comprensión de listas

```
val map : (int -> bool) -> list int -> list bool
let rec map f xs =
  match xs with
  | [] -> []
  | x::xs -> f x :: map f xs
```

Factorial en F*

```
let rec fac (x:nat) : nat =  
  if x = 0 then 1 else x * fac (x-1)
```

- Parece simple, pero requiere:
 - Demostrar que 1 es natural
 - Demostrar que si **x** es natural y no es cero (condición de rama), **x-1** es natural
 - Si **x** es natural y **fac (x-1)** es natural, entonces **x * fac (x-1)** es natural
 - Terminación: la recursión eventualmente termina
- La mayor parte de este trabajo lo realiza [Z3, un SMT solver](#).

Refinamientos

- Subtipos definidos por una base y una formula lógica
- Para chequear que un **x** tiene tipo **(y:t{phi})**
 - Primero, **x** debe tener tipo **t** (que puede ser un refinamiento)
 - La proposición **phi[x/y]** debe ser cierta
- Dualmente, si tenemos **x** de tipo **(y:t{phi})**, podemos asumir ambos puntos
 - Esto implica que, por ejemplo, siempre puede usarse un natural donde se espera un entero. No hacen falta coerciones (vs Coq, Agda).

```
(** The type of non-negative integers *)
type nat = i: int{i >= 0}

(** The type of positive integers *)
type pos = i: int{i > 0}

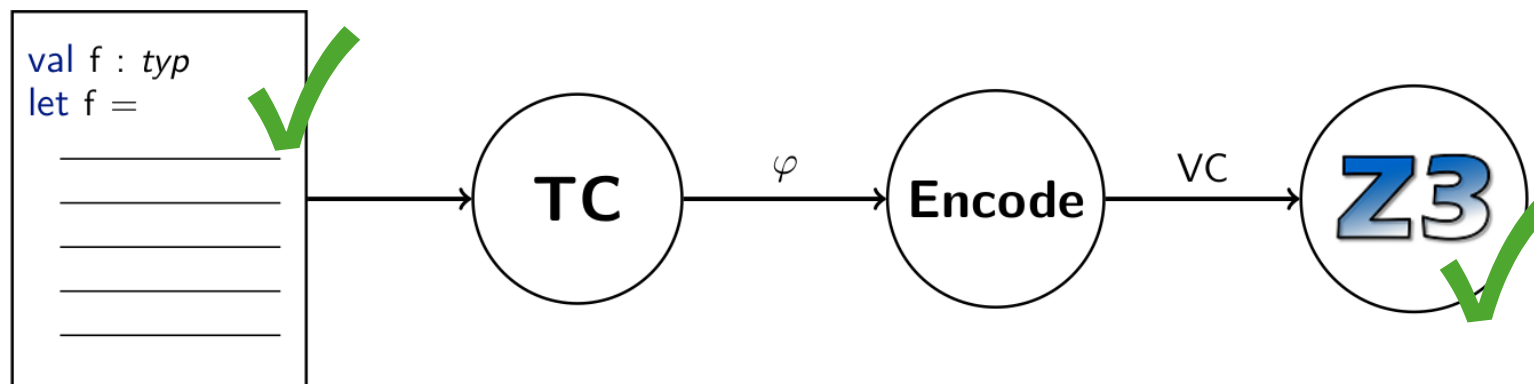
(** The type of non-zero integers *)
type nonzero = i: int{i <> 0}
```

```
let addnat (x y : nat) : int = x + y
```

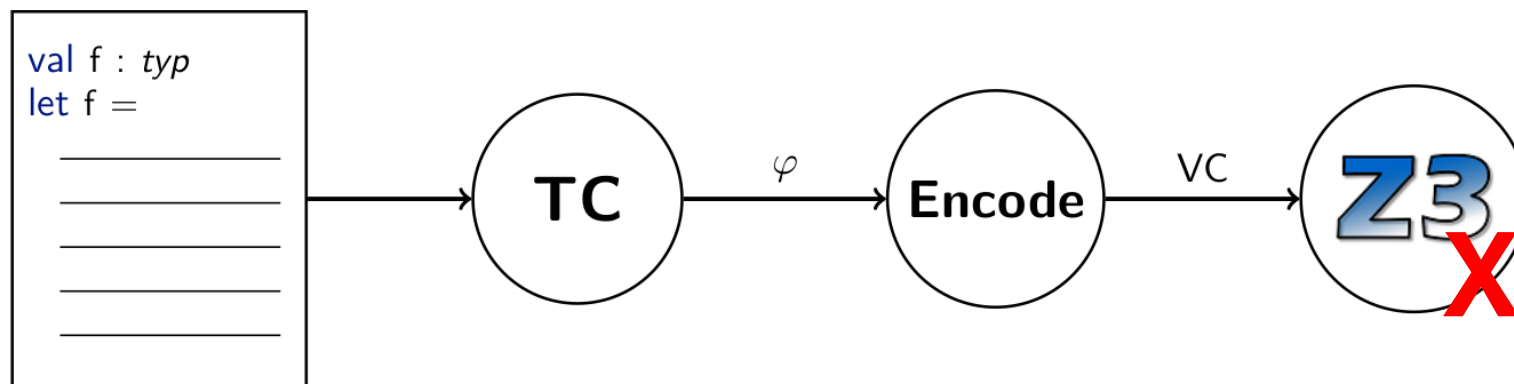
```
let debilitar (f : nat -> nat) : int -> int
```

```
let debilitar (f : int -> nat) : nat -> int = f
```

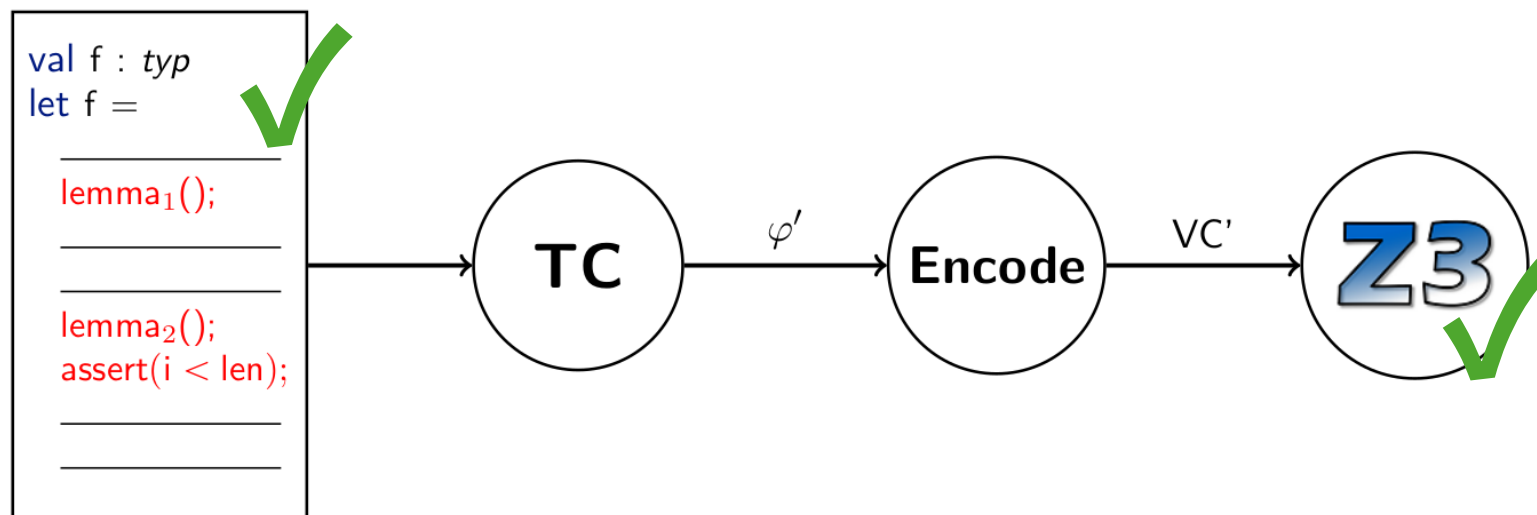
Verificación “auto-activa”



Verificación “auto-activa”



Verificación “auto-activa”



(demo)

Extracción: ejecutando F*

- (No vamos a ver mucho.)
- Los programas en F* se pueden *extraer* a OCaml, y luego usar una toolchain normal para compilar y ejecutar.
 - Uso de garbage collector
- Low*: un DSL de “bajo-nivel” embebido en F*
 - Código de primer orden (e.g. sin lambdas)
 - Pruebas arbitrariamente complejas (total se borran)
 - Manejo manual de memoria
 - Extrae a C via [Karamel](#), sin garbage collector
- Steel y Pulse:
 - Lógica de separación en F*
 - De alguna forma la evolución de Low*, pero más escalable
 - Steel extrae a C. Pulse extrae a OCaml/C/Rust

Disclaimer: TCB

¿Es correcta la lógica de F*?

¿Es correcta la *implementación* de F*?

¿Es correcto el SMT solver?

¿Es correcto el proceso de extracción?

- Vamos a asumir que **sí**. Responder estas preguntas está fuera del alcance de la materia.
- Esto pone a F* y Z3 en nuestra *trusted computing base* (TCB). Si algo en la TCB no cumple su especificación, perdemos las garantías sobre el programa. En general la TCB también incluye al hardware, librerías, runtime, etc.
- Así y todo, es extremadamente raro que un fallo en, e.g., la lógica de F* se alinee perfectamente para secondar un bug *real*. Las garantías que se obtienen en la práctica siguen siendo altas aun sin verificar las herramientas.

- [Self-certification: Bootstrapping certified typecheckers in F* with Coq](#)
- [A Verified Implementation of the DPLL Algorithm in Dafny](#)
- [MetaCoq | Website of the MetaCoq Project](#)
- [Gagallium : How I found a bug in Intel Skylake processors](#)

Tareas

- Preparar VS Code siguiendo instrucciones
 - [mtzguido/intro-verif-25](https://mtzguido.github.io/intro-verif-25)
- Leer: capítulos 1 y 2 del [libro](#)
- Completar archivo Clase1.fst

Introducción a la Verificación Formal

Clase 2 – 04/09/2025

¿Cuál es el tipo de printf?

```
printf "%s\n" : string -> unit
```

```
printf "%d\n" : int -> unit
```

```
printf "%d + %d = %f\n" : int -> int -> float -> unit
```

Entonces printf : string -> ...?

Tipado simple

$$\begin{array}{c} t ::= c \mid t \rightarrow t \end{array} \qquad \begin{array}{c} \text{T-ABS-ST} \\ \frac{\Gamma, x : t \vdash e : s}{\Gamma \vdash (\lambda(x : t).e) : t \rightarrow s} \end{array} \qquad \begin{array}{c} \text{T-APP-ST} \\ \frac{\Gamma \vdash f : t \rightarrow s \quad \Gamma \vdash e : t}{\Gamma \vdash fe : s} \end{array}$$

- Tipado en STLC
- Los tipos son *cerrados*: no dependen del contexto Γ
- Si $f : t \rightarrow s$, entonces f e debe tener tipo s

Sistema F: polimorfismo paramétrico

$$\begin{array}{c} t ::= X \mid c \mid t \rightarrow t \mid \forall X.t \\[10pt] \text{T-ABS} \quad \frac{\Gamma, x:t \vdash e : s}{\Gamma \vdash (\lambda(x:t).e) : t \rightarrow s} \quad \text{T-APP} \quad \frac{\Gamma \vdash f : t \rightarrow s \quad \Gamma \vdash e : t}{\Gamma \vdash fe : s} \\[10pt] \text{T-TABS} \quad \frac{\Gamma, X \vdash e : s}{\Gamma \vdash (\Lambda X.e) : \forall X.s} \quad \text{T-TAPP} \quad \frac{\Gamma \vdash e : \forall X.t \quad \Gamma \vdash A : \text{Type}}{\Gamma \vdash eA : t[A/X]} \end{array}$$

- Algo de “dependencia”: los términos pueden depender de tipos
 - a.k.a. polimorfismo, y en este caso es *paramétrico*
- Los tipos ahora *tienen variables ligadas*
 - Deben estar bien formados
- Dos “tipos” de aplicación y abstracción
 - Sintaxis distinguida

[Wadler 1989 – Theorems for Free!](#)

Tipado dependiente

Sintaxis uniforme para expresiones y tipos

$t ::= x \mid c \mid (x : t) \rightarrow t$

$(t \rightarrow t')$ es azúcar para $(x : t) \rightarrow t'$ con $x \notin FV(t')$

x puede aparecer en **s**

T-ABS-DEP

$\Gamma, x : t \vdash e : s$

$\Gamma \vdash (\lambda(x : t).e) : (x : t) \rightarrow s$

T-APP-DEP

$\Gamma \vdash f : (x : t) \rightarrow s \quad \Gamma \vdash e : t$

$\Gamma \vdash fe : s[e/x]$

Sustitución por el argumento real

- En tipado dependiente, los tipos pueden depender de valores
 - Ya vimos algunos ejemplos encubiertos:

```
val incr''': (x:nat) -> y:int{y = x+1}
let incr''' (x:nat) : y:int{y = x+1} = x+1
```

Polimorfismo en F*

- Como en otros lenguajes con tipos dependientes, las funciones polimórficas son simplemente funciones que toman un tipo como argumento

```
val id : (a:Type) -> a -> a  
let id a x = x
```

```
let _ = assert (id int 42 == 42)  
let _ = assert (id string "hola" == "hola")
```

```
let _ = assert (id Type string == string)  
let _ = assert (id (id Type string) "hola" == "hola")
```

Al margen: terminología

$$\begin{aligned} & \Pi_{x:A} B \\ & (x : A) \rightarrow B \\ & \Pi(x : A) B(x) \end{aligned}$$

“Producto dependiente” (función dependiente, flecha dependiente, tipo Π)

$$\begin{aligned} \mathbf{Bool} \rightarrow A & \cong A \times A \\ \mathbf{Nat} \rightarrow A & \cong A \times A \times \cdots \times A \\ \Pi_{x:\mathbf{Nat}} A(x) & \cong A(0) \times A(1) \times \cdots \times A(n) \times \cdots \end{aligned}$$

“Suma dependiente” (par dependiente, tupla dependiente, tipo Σ)

$$\begin{aligned} \mathbf{Bool} \times A & \cong A + A \\ \mathbf{Nat} \times A & \cong A + A + \cdots + A \\ \Sigma_{x:\mathbf{Nat}} A(x) & \cong A(0) + A(1) + \cdots + A(n) + \cdots \end{aligned}$$

(demo)



**AND NOW FOR SOMETHING
COMPLETELY DIFFERENT**

Lógica formal

- Presentación “a la Gentzen”
 - Hipótesis “a descargar”
- Reglas de introducción y eliminación
- Una prueba es una derivación correcta de una proposición
- “Formal”: las pruebas pueden chequearse mecánicamente

$$\begin{array}{c}
 \Rightarrow\text{-INTRO} \\
 \frac{A}{\vdots} \\
 \frac{B}{A \Rightarrow B}
 \end{array}
 \qquad
 \begin{array}{c}
 \Rightarrow\text{-ELIM} \\
 \frac{A \Rightarrow B \quad A}{B}
 \end{array}$$

$$\begin{array}{c}
 \wedge\text{-INTRO} \\
 \frac{A \quad B}{A \wedge B}
 \end{array}
 \qquad
 \begin{array}{c}
 \wedge\text{-ELIM-L} \\
 \frac{A \wedge B}{A}
 \end{array}
 \qquad
 \begin{array}{c}
 \wedge\text{-ELIM-R} \\
 \frac{A \wedge B}{B}
 \end{array}$$

Lógica formal - Secuentes

- Secuentes: explicitan las hipótesis
- Prop de Eliminación de cortes:
 - Corte: introducción seguida de eliminación para un mismo conectivo
 - Toda prueba puede simplificarse a una prueba sin cortes
- Todo corte puede reducirse, eso es trivial
- ¿El proceso termina?
 - Sí, Gentzen demostró que si se toman los cortes más externos, siempre termina

$$\begin{array}{c}
 \text{Ax} \\
 \hline
 \Gamma, A, \Gamma' \vdash A
 \end{array}
 \quad
 \begin{array}{c}
 \Rightarrow\text{-INTRO} \\
 \Gamma, A \vdash B \\
 \hline
 \Gamma \vdash A \Rightarrow B
 \end{array}$$

$$\begin{array}{c}
 \Rightarrow\text{-ELIM} \\
 \Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A \\
 \hline
 \Gamma \vdash B
 \end{array}$$

$$\begin{array}{c}
 \wedge\text{-INTRO} \\
 \Gamma \vdash A \quad \Gamma \vdash B \\
 \hline
 \Gamma \vdash A \wedge B
 \end{array}
 \quad
 \begin{array}{c}
 \wedge\text{-ELIM-L} \\
 \Gamma \vdash A \wedge B \\
 \hline
 \Gamma \vdash A
 \end{array}
 \quad
 \begin{array}{c}
 \wedge\text{-ELIM-R} \\
 \Gamma \vdash A \wedge B \\
 \hline
 \Gamma \vdash B
 \end{array}$$

$$\begin{array}{c}
 [\Gamma, A \vdash A] \\
 \vdots \\
 \hline
 \Gamma, A \vdash^1 B
 \end{array}
 \quad
 \Gamma \vdash^2 A$$

$$\frac{\Gamma, A \vdash^1 B}{\Gamma \vdash A \Rightarrow B}
 \quad
 \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash^2 A}{\Gamma \vdash B}
 \rightsquigarrow
 \frac{[\Gamma \vdash^2 A]}{\Gamma \vdash^1 B}$$

$$\frac{\text{Ax}}{\overline{\Gamma, A, \Gamma' \vdash A}} \quad \frac{\begin{array}{c} \Rightarrow\text{-INTRO} \\ \Gamma, A \vdash B \end{array}}{\overline{\Gamma \vdash A \Rightarrow B}}$$

$$\frac{\begin{array}{c} \Rightarrow\text{-ELIM} \\ \Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A \end{array}}{\Gamma \vdash B}$$

$$\frac{\begin{array}{c} \wedge\text{-INTRO} \\ \Gamma \vdash A \quad \Gamma \vdash B \end{array}}{\Gamma \vdash A \wedge B} \quad \frac{\begin{array}{c} \wedge\text{-ELIM-L} \\ \Gamma \vdash A \wedge B \end{array}}{\Gamma \vdash A} \quad \frac{\begin{array}{c} \wedge\text{-ELIM-R} \\ \Gamma \vdash A \wedge B \end{array}}{\Gamma \vdash B}$$

$$\frac{\begin{array}{c} \vee\text{-INTRO-L} \\ \Gamma \vdash A \end{array}}{\Gamma \vdash A \vee B} \quad \frac{\begin{array}{c} \vee\text{-INTRO-R} \\ \Gamma \vdash B \end{array}}{\Gamma \vdash A \vee B} \quad \frac{\begin{array}{c} \vee\text{-ELIM} \\ \Gamma \vdash A \vee B \end{array} \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}$$

$$\frac{}{\Gamma \vdash \top} \quad \frac{\begin{array}{c} \perp\text{-ELIM} \\ \Gamma \vdash \perp \end{array}}{\overline{\Gamma \vdash A}}$$

(PD: Consistencia)

- ¿Cómo se demuestra que la lógica es *consistente*?
 - i.e. tiene al menos una prop P tal que **no** ocurre $\vdash P$
- 1. Se demuestra que la lógica tiene la propiedad de eliminación de cortes
 - Esta es la parte difícil, en general por mostrar que el procedimiento termina
- 2. No hay pruebas sin cortes de Falso
 - Una prueba sin cortes tiene que terminar con una introducción
 - Entonces esto es trivial, porque falso no tiene introducciones.

$\not\vdash \perp$

Correspondencia Curry-Howard

- a.k.a. “propositions as types”

$$\begin{array}{c}
 \text{Ax} \\
 \hline
 \Gamma, A, \Gamma' \vdash A
 \end{array}
 \quad
 \begin{array}{c}
 \Rightarrow\text{-INTRO} \\
 \Gamma, A \vdash B \\
 \hline
 \Gamma \vdash A \Rightarrow B
 \end{array}
 \quad
 \begin{array}{c}
 \text{Ax} \\
 \hline
 \Gamma, x : A, \Gamma' \vdash x : A
 \end{array}
 \quad
 \begin{array}{c}
 \rightarrow\text{-INTRO} \\
 \Gamma, x : A \vdash e : B \\
 \hline
 \Gamma \vdash (\lambda x. e) : A \rightarrow B
 \end{array}$$

$$\begin{array}{c}
 \Rightarrow\text{-ELIM} \\
 \Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A \\
 \hline
 \Gamma \vdash B
 \end{array}
 \quad
 \begin{array}{c}
 \rightarrow\text{-ELIM} \\
 \Gamma \vdash f : A \rightarrow B \quad \Gamma \vdash e : A \\
 \hline
 \Gamma \vdash fe : B
 \end{array}$$

$$\begin{array}{c}
 \wedge\text{-INTRO} \\
 \Gamma \vdash A \quad \Gamma \vdash B \\
 \hline
 \Gamma \vdash A \wedge B
 \end{array}
 \quad
 \begin{array}{c}
 \wedge\text{-ELIM-L} \\
 \Gamma \vdash A \wedge B \\
 \hline
 \Gamma \vdash A
 \end{array}
 \quad
 \begin{array}{c}
 \wedge\text{-ELIM-R} \\
 \Gamma \vdash A \wedge B \\
 \hline
 \Gamma \vdash B
 \end{array}$$

$$\begin{array}{c}
 \times\text{-INTRO} \\
 \Gamma \vdash x : A \quad \Gamma \vdash y : B \\
 \hline
 \Gamma \vdash (x, y) : A \times B
 \end{array}
 \quad
 \begin{array}{c}
 \times\text{-ELIM-L} \\
 \Gamma \vdash p : A \times B \\
 \hline
 \Gamma \vdash \mathbf{fst} \, p : A
 \end{array}
 \quad
 \begin{array}{c}
 \times\text{-ELIM-R} \\
 \Gamma \vdash A \times B \\
 \hline
 \Gamma \vdash \mathbf{snd} \, p : B
 \end{array}$$

Relacionado: interpretación Brouwer-Heyting-Kolmogorov de lógica intuicionista

Curry-Howard

- Tomamos una proposición (tipo) P como cierta si existe un habitante de P
- Las pruebas **son programas**, reducen y tienen contenido computacional
 - Una prueba de $A \vee B$ decide cuál caso vale
 - Cuando tengamos existenciales: son programas que computan un testigo

$$\frac{\frac{}{p : A \times B \vdash p : A \times B}}{p : A \times B \vdash \mathbf{snd} \, p : B} \quad \frac{\frac{}{p : A \times B \vdash p : A \times B}}{p : A \times B \vdash \mathbf{fst} \, p : A}}{p : A \times B \vdash (\mathbf{snd} \, p, \mathbf{fst} \, p) : B \times A}$$

$$\vdash \lambda p. (\mathbf{snd} \, p, \mathbf{fst} \, p) : A \times B \rightarrow B \times A$$

$$\frac{\frac{}{A \times B \vdash A \times B}}{A \times B \vdash B} \quad \frac{\frac{}{A \times B \vdash A \times B}}{A \times B \vdash A}}{A \times B \vdash B \times A}$$

$$\vdash A \times B \rightarrow B \times A$$

$$\frac{\frac{}{A \wedge B \vdash A \wedge B}}{A \wedge B \vdash B} \quad \frac{\frac{}{A \wedge B \vdash A \wedge B}}{A \wedge B \vdash A}}{A \wedge B \vdash B \wedge A}$$

$$\vdash A \wedge B \implies B \wedge A$$

Curry-Howard

- El término codifica la prueba
 - Si el tipado es dirigido por sintaxis, es inmediato construir un tipado para un término dado
 - Chequear una prueba es fácil = chequear el tipo de un término es fácil
 - Encontrar una prueba es difícil = encontrar un habitante de un tipo es difícil
- Es un principio general, hay una correspondencia para cada lógica
 - STLC \approx Lógica proposicional
 - System F \approx Lógica de segundo orden
 - Tipado dependiente (tipos dependen de valores) \approx Lógica de predicados
 - Tipado dependiente + polimorfismo \approx lógica de alto orden
 - Y más...

$$\begin{array}{c}
 \frac{}{p : A \times B \vdash p : A \times B} \quad \frac{}{p : A \times B \vdash p : A \times B} \\
 \frac{}{p : A \times B \vdash \mathbf{snd} \, p : B} \quad \frac{}{p : A \times B \vdash \mathbf{fst} \, p : A} \\
 \hline
 p : A \times B \vdash (\mathbf{snd} \, p, \mathbf{fst} \, p) : B \times A \\
 \hline
 \vdash \lambda p. (\mathbf{snd} \, p, \mathbf{fst} \, p) : A \times B \rightarrow B \times A
 \end{array}$$

$$\begin{array}{c}
 \frac{}{A \times B \vdash A \times B} \quad \frac{}{A \times B \vdash A \times B} \\
 \frac{}{A \times B \vdash B} \quad \frac{}{A \times B \vdash A} \\
 \hline
 A \times B \vdash B \times A \\
 \hline
 \vdash A \times B \rightarrow B \times A
 \end{array}$$

$$\begin{array}{c}
 \frac{}{A \wedge B \vdash A \wedge B} \quad \frac{}{A \wedge B \vdash A \wedge B} \\
 \frac{}{A \wedge B \vdash B} \quad \frac{}{A \wedge B \vdash A} \\
 \hline
 A \wedge B \vdash B \wedge A \\
 \hline
 \vdash A \wedge B \implies B \wedge A
 \end{array}$$

Correspondencia Curry-Howard (2)

- La correspondencia se extiende a la simplificación de pruebas
 - Tipo \approx proposición o predicado
 - Prueba \approx programa/algoritmo
 - “cut-elimination” de la implicancia \approx beta-reducción
 - “cut-elimination” de pares \approx reducción de proyecciones ($\text{fst } (x,y) \rightsquigarrow x$)
 - Prueba en forma normal \approx término en forma normal
 - La lógica tiene eliminación de cortes \approx el lenguaje es débilmente normalizante (en general... hay casos borde)

$$\frac{
 \begin{array}{c}
 [\Gamma, A \vdash A] \\
 \vdots \\
 \hline
 \Gamma, A \vdash^1 B
 \end{array}
 \quad
 \Gamma \vdash^2 A
 }{
 \Gamma \vdash A \implies B \quad \Gamma \vdash^2 A
 }
 \rightsquigarrow
 \frac{
 \begin{array}{c}
 [\Gamma \vdash^2 A] \\
 \vdots \\
 \hline
 \Gamma \vdash^1 B
 \end{array}
 }{
 }$$

["Propositions as Types" by Philip Wadler - YouTube](#)

Intuicionismo / constructivismo

- Para capturar la lógica clásica, nos falta una regla importante

$$\overline{\Gamma \vdash A \vee \neg A}$$

- En lógica intuicionista, se rechaza este axioma
 - Implicaría un procedimiento de decisión para cada prop A (en lógicas suf. expresivas contradice a Gödel y Turing a la vez)
 - No tiene interpretación computacional
 - Rompe propiedades como $(\vdash A \vee B) \implies (\vdash A) \vee (\vdash B)$
- Hay más axiomas clásicos, muchos son equivalentes
 - Notoriamente la eliminación de doble negación: $\neg\neg A \implies A$
 - Otros: ver práctica

Prueba no constructivas

Veamos un ejemplo. *Existen irracionales a y b , tales que a^b es racional.*

Demostración (por el absurdo): consideremos $\sqrt{2}^{\sqrt{2}}$. Por el principio del tercero excluido, este número será racional o irracional. En el primer caso, basta tomar $a = b = \sqrt{2}$; en el segundo caso, basta tomar $a = \sqrt{2}^{\sqrt{2}}$ y $b = \sqrt{2}$, pues $\sqrt{2}^{\sqrt{2}^{\sqrt{2}}} = 2$.

La demostración es impecable, pero no sabemos si $\sqrt{2}^{\sqrt{2}}$ es racional o no.

De la charla “Continuaciones: La Venganza del Goto” de Guido Macchi, JCC 2007

(demo)

Pureza / totalidad

- En un lenguaje de programación, ¿por qué no podemos demostrar cualquier P mediante recursion?

```
let rec prueba () : p = prueba ()  
let prueba () : p = let x = 1/0 in ...  
let prueba () : p = raise “☺”
```

- En la presencia de **efectos** (no terminación, parcialidad, excepciones, etc), la correspondencia se rompe. En general, sólo vale para lenguajes puros
- En F* tenemos definiciones puras y *efectuosas*
 - El sistema de efectos se encarga de separar las efectuosas del fragmento puro
 - Implica chequear terminación, completitud de pattern match, etc...

Tareas

- Completar Clase02.fst
- Leer capítulos 3 y 6
- Otras fuentes:
 - ["Propositions as Types" by Philip Wadler - YouTube](#)
 - [Cayenne – a language with dependent types \(Augustsson 1998\)](#)
 - El tipo de printf: [printf* \(fstarlang.github.io\)](#)
 - [Continuations and Logic.pdf \(cmu.edu\)](#)
 - [Guido Macchi – Continuaciones la Venganza del Goto \(JCC 2007\)](#)

Introducción a la Verificación Formal

Clase 3 – 11/09/2025

- Repaso Clase 2

Funciones recursivas

- Al definir `let rec f (x1:t1) : t = ... f ...`
 - Dentro de la definición, el cuerpo se chequea *asumiendo* que `f` tiene tipo $(x1:t1) \rightarrow t$
(como es usual)

Pero, tenemos que chequear terminación, con lo cual hay un pequeño ajuste:

$$(x1':t1\{x1' \ll x1\}) \rightarrow t$$

es decir: la función recursivamente ligada sólo está definida para `x1'` “menores” al `x1` original

- La relación `<<` es un orden bien fundado (sin cadenas infinitamente descendientes) sobre todos los valores de F^* . En particular:
 - Para x, y naturales $x \ll y \iff x < y$
 - Dado un constructor C , tenemos $xi \ll (C \ x1 \ \dots \ xi \ \dots \ xn)$

Claúsula decreases

- Siempre debe haber una métrica de terminación.
 - Si no se explicita, F* toma el orden lexicográfico de todos los argumentos

```
let rec add (m n : nat) : nat =  
  if n = 0 then m  
  else add (m+1) (n-1)
```

- Para darla explícitamente se usa la cláusula decreases:

```
let rec add (m n : nat) : Tot nat (decreases n) =  
  if n = 0 then m  
  else add (m+1) (n-1)
```

Orden lexicográfico

- Orden sobre tuplas a partir de órdenes para los componentes

$$(x_1, y_1, z_1) < (x_2, y_2, z_2) \iff x_1 < x_2 \vee$$

$$(x_1 == x_2 \wedge y_1 < y_2) \vee$$

$$(x_1 == x_2 \wedge y_1 == y_2 \wedge z_1 < z_2)$$

- La función de Ackermann termina porque respeta el orden lexicográfico en (m, n)

$$A(0, n) = n + 1$$

$$A(m + 1, 0) = A(m, 1)$$

$$A(m + 1, n + 1) = A(m, A(m + 1, n))$$

Tipos inductivos

- Tipos de datos generados por constructores, que pueden mencionar recursivamente al tipo.
 - Soportan pattern matching, como es usual
- En la definición de listas, la variable a es un *parámetro* del tipo inductivo
 - Es una elección “externa” que no cambia.
 - Hay distintas “versiones” del tipo lista para cada a

```
type list (a:Type) =  
  | Nil : list a  
  | Cons : hd:a -> tl:list a -> list a
```

```
Nil?      : #a:Type -> xs:list a -> bool  
Cons?     : #a:Type -> xs:list a -> bool  
Cons?.hd  : #a:Type -> xs:list a{Cons? xs} -> a  
Cons?.tl  : #a:Type -> xs:list a{Cons? xs} -> list a
```


Tipos indexados

- Además, un tipo puede tener *índices*, que pueden variar de constructor en constructor
 - El pattern matching es consciente de los índices (el match de la imagen es completo)
 - Es la idea principal de GADTs en Haskell

```
type t : bool -> Type =  
  | A : nat -> t true  
  | B : string -> t false  
  
let f (x : t true) : nat =  
  match x with  
  | A n -> n
```

GADT para un mini lenguaje

```
type l_ty =  
  | Int  
  | Bool  
  
type expr : l_ty -> Type =  
  | EInt : int -> expr Int  
  | EBool : bool -> expr Bool  
  | EAdd : expr Int -> expr Int -> expr Int  
  | EEq : expr Int -> expr Int -> expr Bool  
  | EIf :  
    #ty:_ ->  
    expr Bool -> expr ty -> expr ty -> expr ty
```

```
EEq : expr Int -> expr Int -> expr Bool  
EI - Expected type "expr Bool"; but "EInt 0" has type "expr Int"  
#t  
ex  
Clase3.EInt:  
_0: int -> expr Int  
  
View Problem (Alt+F8) No quick fixes available  
let te  
EIf (EInt 0) (EInt 1) (EInt 2)
```

```
let lift (ty : l_ty) : Type =  
  match ty with  
  | Int -> int  
  | Bool -> bool  
  
val eval (#ty:l_ty) (e : expr ty) : Tot (lift ty)
```

Vectores indexados por longitud

- El tipo `vec` está indexado por un natural que indica su longitud
 - Esto permite implementar `head/tail` de forma segura
 - Idem `append`, etc, que preservan la longitud en el tipo.

```
type vec (a:Type) : nat -> Type =  
  | VNil : vec a 0  
  | VCons : #n:nat -> hd:a -> tl:vec a n -> vec a (n+1)  
  
let vhd (#a:Type) (#n:pos) (xs : vec a n) : a =  
  match xs with  
  | VCons hd tl -> hd
```

Positividad

- No podemos aceptar cualquier definición de tipo inductivo

```
noeq
type ni =
| Mk : f:(ni -> bool) -> ni

let self_app (x : ni) : bool =
  match x with
  | Mk f -> f x

let boxed : ni =
  Mk self_app

let raro () : bool =
  self_app boxed
```

```
raro ()
~> self_app boxed
~> self_app (Mk self_app)
~> match Mk self_app with
    | Mk f -> f (Mk self_app)
~> self_app (Mk self_app)
~> ....
```

Tareas

- Completar Clase03.*.fst
- Leer capítulos 7, 12, 13
- Otras fuentes:
 - [Conor McBride - Faking it Simulating dependent types in Haskell](#)