Paradigmas de lenguajes de programación

Alejandro Ríos

Departamento de Computación, FCEyN, UBA

"A language that doesn't affect the way you think about programming, is not worth knowing"

Epigrams in Programming, Alan Perlis (primer ganador del Turing Award), ACM SIGPLAN, Sept. 1982

15 de agosto de 2017

Cátedra y modalidad

- Modalidad: Clases teóricas y prácticas

Teoría

- Alejandro Ríos (`rios@dc.uba.ar`)
- (típicamente) Jueves: 17:00hs a 19:30hs
- **Invitados:**
  Fidel(=Pablo E. Martínez López) a una clase de PF y una clase de mónadas.
  Hernán Wilkinson a una clase de Metaprogramación.

Práctica

- Christian Roldán (JTP)
  Gabriela Steren, Edgardo Zoppi, Julián Dabbah, Daniel Álvarez, Sabrina Izcovich (Ays. 1a)
  Sebastián Vita, Iván Arcuschin (Ays. 2a)
- (típicamente) Martes: 17:30hs a 21:00hs

Recursos

Bibliografía

- Textos: no hay un texto principal, se utilizan varios, referencias en página web
- Apuntes: serán introducidos oportunamente
- Publicaciones relacionadas
- Diapositivas de teóricas y prácticas
- Guías de ejercicios

Página web

- Información al día del curso, consultar periódicamente y leer al menos una vez todas las secciones

Mailing list

- ¡Hacer todas las preguntas y consultas que quieran!

Recursos

Software

- Haskell
  - El intérprete (Hugs): `http://www.haskell.org`
  - Documentos (accesibles en el mismo sitio):
    - *A Gentle Introduction to Haskell*, Paul Hudak, John Peterson y Joseph H. Fasel.
    - *The Hugs 98 User Manual*, Mark P. Jones y John Peterson, 1999. (Manual del usuario; modestas 84 páginas)
    - *Haskell 98 Language and Libraries - The Revised Report*, Simon Peyton Jones (ed.), 2002. (La referencia definitiva sobre Hugs98, 277 páginas....)
- SWI-Prolog (programación lógica)
- Pharo (Smalltalk - programación orientada a objetos)

Paradigma

Marco filosófico y teórico de una escuela científica o disciplina en la que se formulan teorías, leyes y generalizaciones y se llevan a cabo experimentos que les dan sustento

Fuente: Merriam-Webster[1]

[1] *A philosophical and theoretical framework of a scientific school or discipline within which theories, laws, and generalizations and the experiments performed in support of them are formulated*

Lenguajes de Programación

- lenguaje usado para comunicar instrucciones a una computadora
- las instrucciones describen cómputos que llevará a cabo la computadora
- la noción de cómputo puede formalizarse de muchas maneras:
  - máquinas de Turing
  - cálculo lambda
  - funciones recursivas
  - ...

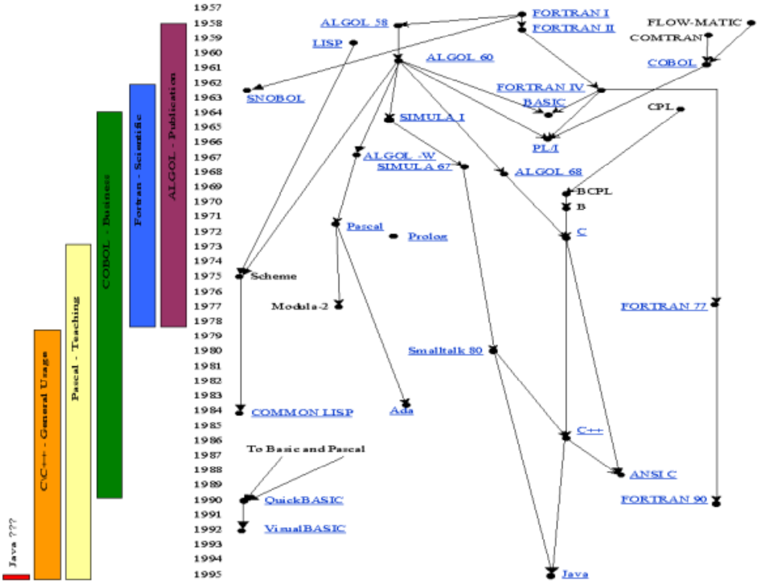
  Siempre se obtiene la misma clase de funciones computables!!
- el lenguaje es computacionalmente completo si puede expresar todas las funciones computables

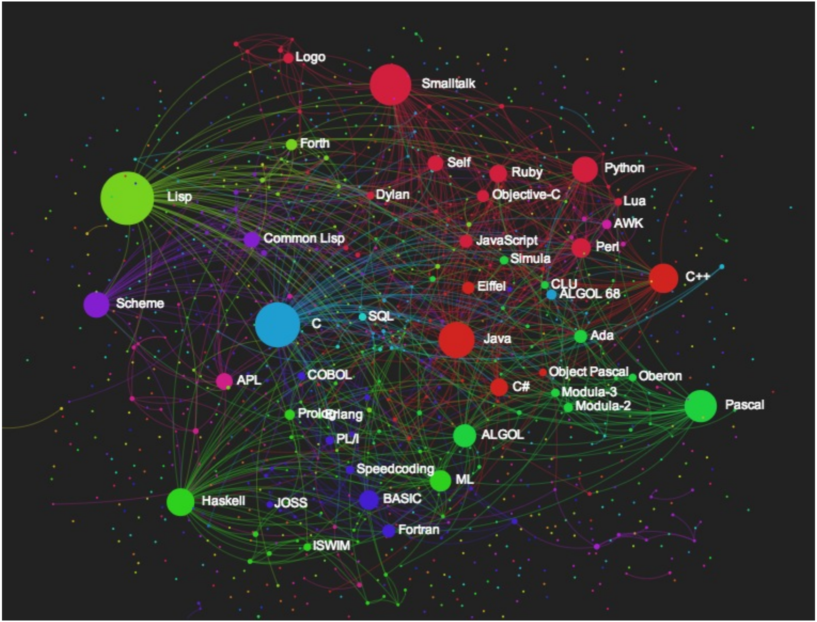
Paradigmas de lenguajes de Programación

Marco filosófico y teórico en el que se formulan soluciones a problemas de naturaleza algorítmica

- Lo entendemos como
  - un estilo de programación
  - en el que se escriben soluciones a problemas en términos de algoritmos
- El ingrediente básico es el modelo de cómputo
  - la visión que tiene el usuario de cómo se ejecutan los programas

Orígenes de los Lenguajes de Programación





exploringdata.github.io/info/programming-languages-influence-network/

Addendum

Griff: July 17, 2012 at 10:01 am How can something influence everything but be used by no one? Haskell's life force is in the things it influenced surely [snip]

Aphidman: July 19, 2012 at 2:25 pm Maybe Haskell is the Velvet Underground of programming languages. They say that the Velvet Underground only sold 10,000 records, but everyone who bought one went out and started a rock band.

`http://griffsgraphs.com/2012/07/01/programming-languages-influences/`

Objetivos del curso

Conocer los pilares conceptuales sobre los cuales se erigen los lenguajes de programación de modo de poder

- Comparar lenguajes
- Seleccionar el más adecuado para una determinada tarea
- Usar las herramientas disponibles adecuadamente
- Prepararse para lenguajes/paradigmas futuros

Enfoque del curso

1. Angulo conceptual
   - Introducción informal de conceptos a través de ejemplos.
2. Angulo de fundamentos
   - Introducir las bases rigurosas (lógicas y matemáticas) sobre las que se sustentan cada uno de los paradigmas o parte de los mismos

Algunos temas cubiertos - Angulo conceptual

- Recursión
- Valores, expresiones, currificación, funciones de alto orden, polimorfismo paramétrico, esquemas de recursión
- Asignación y efectos laterales
- Expresiones de tipo, sistema de tipos, checkeo de tipos, inferencia de tipos
- Resolución en lógica proposicional y de primer orden, Cláusulas de Horn, unificación, refutación, resolución SLD
- Objeto, clase, herencia, method dispatch estático/dinámico, polimorfismo de subclase, subtipos, sistemas de tipos invariantes

Algunos temas cubiertos - Angulo de fundamentos

- Lambda cálculo
  - Sintaxis y ejemplos de programación
  - Sistemas de tipos e inferencia
  - Semántica operacional
- Resolución
  - En lógica proposicional
  - En lógica de primer orden
  - SLD
- Programación orientada a objetos
  - Sistemas de tipos, subtipado.

Definición de un Lenguaje

- Sintaxis
- Sistema de Tipos
- Semántica

Definición de un Lenguaje

- Sintaxis
  - descripción del conjunto de secuencias de símbolos considerados como programas válidos
  - teoría de lenguajes formales bien desarrollada, comenzando a mediados de 1950 con Noam Chomsky como pionero
  - notación BNF (y EBNF) ampliamente utilizada; desarrollada por John Backus para Algol 58, modificada por Peter Naur para Algol 60
  - amplio abanico de herramientas para generar analizadores léxicos y parsers a partir de notación formal como BNF
- Sistema de Tipos
- Semántica

Definición de un Lenguaje

- Sintaxis
- Sistema de Tipos
  - propósito: prevenir errores en tiempo de ejecución
  - en general, requiere anotaciones de tipo en el código fuente
  - ejemplos: evitar sumar booleanos, aplicar función a número incorrecto de argumentos.
  - análisis de tipos en tiempo de compilación: chequeo de tipos estático
  - análisis de tipos en tiempo de ejecución: chequeo de tipos dinámico
  - veremos en detalle en el Eje de Fundamentos
- Semántica

Definición de un Lenguaje

- Sintaxis
- Sistema de Tipos
- Semántica
  - descripción del significado de instrucciones y expresiones
  - puede ser informal (eg. Castellano) o formal (basado en técnicas matemáticas); semántica formal puede ser axiomática, operacional o denotacional
  - ¿Por qué semántica formal?[a]:
    - Destructor británico H.M.S. Sheffield hundido en guerra de Malvinas. El radar de alerta de la nave estaba programado para identificar el misil Exocet como .aliado"debido a que el arsenal Inglés incluye el "homing device" de los Exocet y permitió que el misil alcanzara su blanco (el H.M.S. Sheffield).

[a] http://www.cs.tau.ac.il/~nachumd/horror.html

Definición de un Lenguaje

- Sintaxis
- Sistema de Tipos
- Semántica
  - descripción del significado de instrucciones y expresiones
  - puede ser informal (eg. Castellano) o formal (basado en técnicas matemáticas); semántica formal puede ser axiomática, operacional o denotacional
  - ¿Por qué semántica formal?
    - Votos perdidos por computadora en Toronto. El distrito de Toronto finalmente abandonó votación electrónica.

Definición de un Lenguaje

- Sintaxis
- Sistema de Tipos
- Semántica
  - descripción del significado de instrucciones y expresiones
  - puede ser informal (eg. Castellano) o formal (basado en técnicas matemáticas); semántica formal puede ser axiomática, operacional o denotacional
  - ¿Por qué semántica formal?
    - 225 de los 254 pasajeros de Korean Airlines KAL 901 en Guam fallecen en accidente. Bug descubierto en altímetro barométrico del Ground Proximity Warning System (GPWS).

Definición de un Lenguaje

- Sintaxis
- Sistema de Tipos
- Semántica
  - descripción del significado de instrucciones y expresiones
  - puede ser informal (eg. Castellano) o formal (basado en técnicas matemáticas); semántica formal puede ser axiomática, operacional o denotacional
  - ¿Por qué semántica formal?
    - Falla en el despegue del satélite Ariane 5 causado por software error en la rutina de manejo de excepciones resultante de una mala conversión de un punto flotante de 64-bit a un entero.

Paradigmas de Lenguajes

- Nos ocuparemos de los paradigmas:
  - *imperativo*
  - funcional
  - lógico
  - orientado a objetos
- Hay otros: concurrente, eventos, continuaciones, quantum, etc.
- ¡La distinción a veces no está clara!

Imperativo

¿Qué hace este programa?

```
int main (){
  int r,s,n;
  n = 1;
  r = 1;
  s = 0;
  while (n<11) {
    r = r * n;
    s = s + r;
    n = n + 1;
  };
  printf ("%d",s);
}
```

Ayuda: imprime 4037913

Imperativo

¿Qué hace este programa?

```
int main (){
  int r,s,n;
  n = 1;
  r = 1;
  s = 0;
  while (n<11) {
    r = r * n;
    s = s + r;
    n = n + 1;
  };
  printf ("%d",s);
}
```

Ayuda: imprime 4037913

Estado global + asignación + control de flujo

Evaluación

- Computación expresada a través de modificación reiterada de estado global (o memoria)
- Variables como abstracción de celdas de memoria
- Resultados intermedios se almacenan en la memoria
- Repetición basada en iteración

Imperativo

¿Qué hace este programa?

```
int main (){
  int r,s,n;
  n = 1;
  r = 1;
  s = 0;
  while (n<11) {
    r = r * n;
    s = s + r;
    n = n + 1;
  };
  printf ("%d",s);
}
```

Ayuda: imprime 4037913

Evaluación

- Eficiente
  - Modelo ejecución - Arquit. $= 0$
- Bajo nivel de abstracción
  - Especif - Implement $= \infty$
- Matemática/lógica de programas compleja
  - Operacional vs denotacional

Imperativo - Ejemplo - Bajo nivel de abstracción

¿Qué hace este programa?

```
int g;
int f(int x)
   ++g;
   return 3;

int main ()
  g=6;
  if (g==f(2)+f(2))
    printf("Eject!");
  else
    printf("Mantener
      curso actual!");
```

Imperativo - Ejemplo - Bajo nivel de abstracción

¿Qué hace este programa?

```
int g;
int f(int x)
   ++g;
   return 3;

int main ()
  g=6;
  if (g==f(2)+f(2))
    printf("Eject!");
  else
    printf("Mantener
      curso actual!");
```

- El resultado depende del orden de evaluación de $==$.
- No habría que preocuparse por dicho orden

Funcional

```
sum (map fact [1..10])

fact 0 = 1
fact n = n * fact (n-1)
```

Funcional

```
sum (map fact [1..10])

fact 0 = 1
fact n = n * fact (n-1)
```

Evaluación

- Computación expresada a través de la aplicación y composición de funciones
- No hay un estado global
- Resultados intermedios (salida de las funciones) son pasados directamente a otras funciones como argumentos
- Repetición basada en recursión
- Expresiones tipadas

Funcional

```
sum (map fact [1..10])

fact 0 = 1
fact n = n * fact (n-1)
```

Evaluación

- Alto nivel de abstracción
  - Especificación ejecutable: *Sumar resultado de aplicar factorial a la lista de números de 1 a n*
- Declarativo
- Matemática de programas elegante
- Razonamiento algebraico posible
  - $e + e \simeq 2 * e$
  - $a == b \simeq b == a$
- Ejecución lenta (?)

Lógico

```
fact(0,1).
fact(N,Res) :- M is N-1, fact(M,Aux), Res is N*Aux.
lFact([],N,N).
lFact([H|T],M,N) :- fact(M,H), M1 is M+1, lFact(T,M1,N).
sumaFact(X,N) :- lFact(L,1,N), sumList(L,X).
```

Evaluación

- Programas son predicados
- Computación expresada a través de "proof search"
- No hay estado global
- Resultados intermedios son pasados a través de unificación
- Repetición basada en recursión

Lógico

```
fact(0,1).
fact(N,Res) :- M is N-1, fact(M,Aux), Res is N*Aux.
lFact([],N,N).
lFact([H|T],M,N) :- fact(M,H), M1 is M+1, lFact(T,M1,N).
sumaFact(X,N) :- lFact(L,1,N), sumList(L,X).
```

Evaluación

- Alto nivel de abstracción
  - Especificación ejecutable
- Declarativo
- Fundamento lógico robusto
  - Resolución
- Ejecución lenta

Orientado a Objetos

```
Numero << factorial

self = 0 ifTrue: [^ 1].
self > 0 ifTrue: [^ self * (self - 1) factorial].
self error: 'Not valid for negative integers'

Numero << sumaFactoriales

^((1 to: self) collect: [:x | x factorial]) sum
```

Evaluación

- Computación a través del intercambio de mensajes entre objetos
- Objetos se agrupan en clases, clases se agrupan en jerarquías

Orientado a Objetos

Evaluación

- Alto nivel de abstracción
  - Objetos
  - Clases
  - Mensajes
- Arquitectura extensible
  - Jerarquía de clases
  - Polimorfismo de subtipos
  - Binding dinámico
- Matemática de programas compleja

Top five: "Great Works in Programming Languages", B.Pierce

- C.A.R. Hoare. An axiomatic basis for computer programming. Communications of the ACM, 12(10):576-580 and 583, October 1969.
- Peter J. Landin. The next 700 programming languages. Communications of the ACM, 9(3):157-166, March 1966.
- Robin Milner. A theory of type polymorphism in programming. Journal of Computer and System Sciences, 17:348-375, August 1978.
- Gordon Plotkin. Call-by-name, call-by-value, and the λ-calculus. Theoretical Computer Science, 1:125-159, 1975.
- John C. Reynolds. Towards a theory of type structure. In Colloque sur la Programmation, Paris, France, volume 19 of Lecture Notes in Computer Science, pages 408-425. Springer-Verlag, 1974.

Vanguardia

Conferencias Europeas

- The European Joint Conferences on Theory and Practice of Software (ETAPS)
  - Foundations of Software Science and Computation Structures (FOSSACS)
  - Fundamental Approaches to Software Engineering (FASE)
  - European Symposium on Programming (ESOP)
  - International Conference on Compiler Construction (CC)
  - Tools and Algorithms for the Construction and Analysis of Systems (TACAS)
- International Colloquium on Automata, Languages and Programming (ICALP)
- Computer Science Logic

Vanguardia

Conferencias ACM

- Principles of Programming Languages (POPL)
- International Conference on Functional Programming (ICFP)
- Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)
- Programming Language Design and Implementation (PLDI)
- Principles and Practice of Parallel Programming (PPOPP)

PARADIGMAS DE LENGUAJES DE PROGRAMACIÓN

Programación Funcional

Pablo E. "Fidel" Martínez López
(fidel@unq.edu.ar)

“Cuando sepas reconocer la cuatrifolia en todas sus sazones, raíz, hoja y flor, por la vista y el olfato, y la semilla, podrás aprender el verdadero nombre de la planta, ya que entonces conocerás su escencia, que es más que su utilidad.”

Un mago de Terramar

Úrsula K. Le Guin

Programación

- ¿Cuáles son los dos aspectos fundamentales?
  - transformación de información
  - interacción con el medio
- Ejemplos:
  - calcular el promedio de notas de examen
  - cargar datos de un paciente en su historia clínica
- La PF se concentra en el primer aspecto.

Preguntas

- ¿Cómo saber cuándo dos programas son iguales?
- Ejemplo:
  - ¿Son equivalentes '**f(3)+f(3)**' y '**2*f(3)**'?
  - ¿Siempre?
  - ¿Sería deseable que siempre lo fueran? ¿Por qué?

Ejemplo

- ¿Qué imprime este programa?

```
Program test;
   var x : integer;
   function f(y:integer):integer;
      begin  x := x+1; f :=x+y;  end;
   begin  x := 0; writeln(f(3)+f(3));  end;
```

- ¿Y con '**2*f(3)**' en lugar de '**f(3)+f(3)**'?

Valores y Expresiones

- Valores
  - entidades (matemáticas) abstractas con ciertas propiedades
    - **Ejs:** el número dos, el valor de verdad falso.
- Expresiones
  - cadenas de símbolos utilizadas para denotar (escribir, nombrar, referenciar) valores
    - **Ejs:** 2, (1+1), False, (True && False)

Transparencia Referencial

- "El valor de una expresión depende sólo de los elementos que la constituyen."
- Implica:
  - consideración sólo del comportamiento externo de un programa (abstracción de detalles de ejecución).
  - posibilidad de demostrar propiedades usando las propiedades de las subexpresiones y métodos de deducción lógica.

Expresiones

- Expresiones atómicas
  - son las expresiones más simples
  - llamadas también formas normales
  - por abuso de lenguaje, les decimos *valores*
    - **Ejs:** 2, False, (3,True)
- Expresiones compuestas
  - se 'arman' combinando subexpresiones
  - por abuso de lenguaje, les decimos *expresiones*
    - **Ejs:** (1+1), (2==1), (4 - 1, True || False)

Expresiones

- Puede haber expresiones incorrectas ("mal formadas")
  - por errores sintácticos

    *12 (True ('a',)
  - por errores de tipo

    (2+False) (2||'a') 4 'b'
- ¿Cómo saber si una expresión está "bien formada"?
  - Reglas sintácticas
  - Reglas de asignación de tipo

Funciones

- Valores especiales, que representan "transformación de datos"
- Dos formas de entender las funciones
  - VISIÓN DENOTACIONAL
    - una función es un valor matemático que relaciona cada elemento de un conjunto (de partida) con un único elemento de otro conjunto (de llegada).
  - VISIÓN OPERACIONAL
    - una función es un mecanismo (método, procedimiento, algoritmo, programa) que dado un elemento del conjunto de partida, calcula (devuelve, retorna) el elemento correspondiente del conjunto de llegada.

Funciones

- Ejemplo: doble x = x+x
- Visión denotacional
  - a cada número x, doble le hace corresponder otro número, cuyo valor es la suma de x más x

    { (0,0), (1,2), (2,4), (3,6), ... }
- Visión operacional
  - dado un número x, doble retorna ese número sumado consigo mismo

    doble 0 $\rightarrow$ 0 doble 1 $\rightarrow$ 2

    doble 2 $\rightarrow$ 4 doble 3 $\rightarrow$ 6 ...

Funciones

- ¿Cuál es la operación básica de una función?
  - la APLICACIÓN a un elemento de su partida
- Regla sintáctica:
  - la aplicación se escribe por yuxtaposición
    - (f x) denota al elemento que se corresponde con x por medio de la función f.
    - **Ej:** (doble 2) denota al número 4

Funciones

- ¿Qué expresiones denotan funciones?
  - Nombres (variables) definidos como funciones
    - **Ej**: doble
  - Funciones anónimas (lambda abstracciones)
    - **Ej:** (\x -> x+x)
  - Resultado de usar otras funciones
    - **Ej:** doble . doble

Ecuaciones Orientadas

- Dada una expresión bien formada, ¿cómo determinamos el valor que denota?
  - Mediante ECUACIONES que establezcan su valor
- ¿Y cómo calculamos el valor de la misma?
  - Reemplazando subexpresiones, de acuerdo con las reglas dadas por las ecuaciones (REDUCCIÓN)
    - Por ello usamos ECUACIONES ORIENTADAS

Ecuaciones Orientadas

- Expresión-a-definir = expresión-definida

  e1 = e2

- Visión denotacional
  - se define que el valor denotado por e1 (su significado) es el mismo que el valor denotado por la expresión e2
- Visión operacional
  - para calcular el valor de una expresión que contiene a e1, se puede reemplazar e1 por e2

Programas Funcionales

- Definición de programa funcional (*script*):
  - Conjunto de ecuaciones que definen una o más funciones (valores).
- Uso de un programa funcional
  - Reducción de la aplicación de una función a sus datos (reducción de una expresión).

Funciones como valores

- Las funciones son valores, al igual que los números, las tuplas, etc.
  - pueden ser argumento de otras funciones
  - pueden ser resultado de otras funciones
  - pueden almacenarse en estructuras de datos
  - pueden ser estructuras de datos
- Funciones que manipulan funciones
  - Las llamamos “de alto orden”, abusando de esa nomenclatura

Funciones como valores

- Ejemplo

```
compose (f,g) = h  where h x = f (g x)
sqr x = x*x
twice f = g   where g x = f (f x)
aLaCuarta = compose (sqr,sqr)
aLaOctava = compose (sqr,aLaCuarta)
fs = [ sqr, aLaCuarta, aLaOctava, twice sqr ]
```

aLaCuarta 2 $\rightarrow$?

- ¿Será cierto que aLaCuarta = twice sqr?

Lenguaje Funcional Puro

- Definición de lenguaje funcional puro:

"lenguaje de expresiones con transparencia referencial y funciones como valores, cuyo modelo de cómputo es la reducción realizada mediante el reemplazo de iguales por iguales"

Tipos

- Toda expresión válida denota un valor
- Todo valor pertenece a un conjunto
- Los tipos denotan conjuntos
- Entonces...

  TODA EXPRESIÓN DEBERÍA TENER UN TIPO PARA SER VÁLIDA

  (si una expresión no tiene tipo, es inválida)

Asignación de Tipos

- Notación: e :: A
  - se lee "la expresión e tiene tipo A"
  - significa que el valor denotado por e pertenece al conjunto de valores denotado por A
- Ejemplos:

```
2 :: Int                    False :: Bool
'a' :: Char                 doble :: Int -> Int
[ sqr, doble ] :: [ Int -> Int ]
```

Asignación de tipos

- Se puede deducir el tipo de una expresión a partir de su constitución
- Algunas reglas
  - si e1 :: A y e2 :: B, entonces (e1,e2) :: (A,B)
  - si m, n :: Int, entonces (m+n) :: Int
  - si f :: A->B y e :: A, entonces f e :: B
  - si d = e y e :: A, entonces d :: A

Asignación de tipos

- Ejemplo: doble x = x+x
  twice f = g where g y = f (f y)
  - x+x :: Int, y entonces sólo puede ser que x :: Int
  - doble x :: Int y x :: Int, entonces sólo puede ser que doble :: Int -> Int
  - si y :: A y f :: A -> A, entonces f y :: A, f (f y) :: A
  - entonces g y :: A y g :: A -> A
  - por lo tanto twice twice :: (A->A) -> (A->A)

Asignación de tipos

- Propiedades deseables
  - que sea automática (que haya un programa)
  - que le dé tipo al mayor número posible de expresiones con sentido
  - que no le dé tipo al mayor número posible de expresiones sin sentido
  - que se conserve por reducción
  - que los tipos sean descriptivos y razonablemente sencillos de leer

Asignación de tipos

- Inferencia de tipos
  - dada una expresión e, determinar si tiene tipo o no según las reglas, y cuál es ese tipo
- Chequeo de tipos
  - dada una expresión e y un tipo A, determinar si e :: A según las reglas, o no
- Sistema de tipado fuerte (strong typing)
  - sistema que acepta una expresión si, y sólo si ésta tiene tipo según las reglas

Sistema de tipos

- ¿Para qué sirven los tipos?
  - detección de errores comunes
  - documentación
  - especificación rudimentaria
  - oportunidades de optimización en compilación
- Es una buena práctica en programación empezar dando el tipo del programa que se quiere escribir.

Sistema Hindley-Milner

- Tipos básicos
  - enteros Int
  - caracteres Char
  - booleanos Bool
- Tipos compuestos
  - tuplas (A,B)
  - listas [A]
  - funciones (A->B)
- Polimorfismo

Polimorfismo

- ¿Qué tipo tendrá la siguiente función?

  id :: ??

  id x = x

  (id 3) :: Int   (id 'a') :: Char

  (id True) :: Bool   (id doble) :: Int -> Int
- ¿Es una expresión con sentido?
- ¿Debería tener un tipo?
- En realidad:

  id :: A->A, cualquiera sea A

Polimorfismo paramétrico

- Solución: ¡variables de tipo!

  id :: a -> a

  se lee: "id es una función que dado un elemento de *algún tipo* a, retorna un elemento de ese mismo tipo"

- La identidad es una función *polimórfica*
  - el tipo de su argumento puede ser *instanciado* de diferentes maneras en diferentes usos

  (id 3) :: Int y aquí id :: Int -> Int

  (id True) :: Bool y aquí id :: Bool -> Bool

Polimorfismo paramétrico

- Polimorfismo
  - Característica del sistema de tipos
  - Dada una expresión que puede ser tipada de infinitas maneras, el sistema puede asignarle un tipo que sea más general que todos ellos, y tal que en cada uso pueda transformarse en uno particular.
  - Ej: id :: a -> a ← más general
    particulares
    id :: Int -> Int   id :: [Int] -> [Int] ...
    - Reemplazando a por Int, por ejemplo, se obtiene un tipo particular
  - Se llama "paramétrico" pues a es el *parámetro*.

Polimorfismo paramétrico

- ¿Tienen tipo las siguientes expresiones? ¿Cuáles?
  (Recordar: twice f = g where g x = f (f x))

  twice :: ??

  (id doble) (id 3) :: ??

  (id twice) (id doble, id 3) :: ??

  (id id) (id doble) :: ??

  id id :: ??

  twice id :: ??


¿VOS SABÍAS QUE MI MAMÁ ES TRADUCTORA DE FRANCÉS, MANOLITO? YO TAMBIÉN SÉ FRANCÉS; SÉ DECIR "PAPÁ" EN FRANCÉS
¿SÍÍ? ¿CÓMO SE DICE, A VER?
PAPÁ
¡AH, ES FÁCIL! ¡SE DICE IGUAL!
¿FÁCIL? ¿IGUAL? ¡NADA DE ESO, EL ASUNTO ES PENSAR EN FRANCÉS! ¡TRATÁ DE DECIR "PAPÁ" PENSÁNDOLO EN FRANCÉS! ¡DALE! ¿A VER? ¡DALE!
¡ES INÚTIL! ¡JAMÁS PODRÉ HABLAR ESE MALDITO IDIOMA!

Aplicación del alto orden

- Considere las siguientes definiciones

```
suma' :: ??
suma' (x,y) = x+y

suma :: ??
suma x = f   where  f y = x+y
```

- ¿Qué tipo tienen las funciones?
- ¿Qué similitudes observa entre suma y suma'?
- ¿Qué diferencias observa entre ellas?

Aplicación del alto orden

- Similitudes
  - ambas retornan la suma de dos enteros:
    suma' (x,y) = (suma x) y, para x e y cualesquiera
- Diferencias
  - una toma un par y retorna un número;
    la otra toma un número y retorna una *función*
  - con suma se puede definir la función sucesor
    sin usar variables extra:
    succ = suma 1

Currificación

- Correspondencia entre cada función de múltiples parámetros y una de alto orden que retorna una función intermedia que completa el trabajo.
  - Por cada f’ definida como

```
f’ :: (a,b) -> c
f’ (x,y) = e
```

  siempre se puede escribir

```
f :: a -> (b -> c)
(f x) y = e
```

Currificación

- Correspondencia entre los tipos

```
(a,b) -> c      y      a -> (b -> c)

curry :: ((a,b) -> c) -> (a -> (b -> c))
curry …

uncurry :: (a -> (b -> c)) -> ((a,b) -> c)
uncurry …
```

- Se puede demostrar que

```
curry (uncurry f) = f
uncurry (curry f') = f'
```

Currificación - Sintaxis

- ¿Cómo escribimos una función currificada y su aplicación?
- Considerar las siguientes definiciones

  twice :: (Int->Int) -> (Int -> Int)

  twice_1 f = g where g x = f (f x)

  twice_2 f = \x -> f (f x)

  (twice_3 f) x = f (f x)

- ¿Son equivalentes? ¿Cuál es preferible? ¿Por qué?

Currificación

- ¿Qué pasa con un ejemplo más grande?
  - Consideremos una función para sumar 5 números

```
sum5' :: (Int, Int, Int, Int, Int) -> Int
sum5' (x,y,z,v,w) = x+y+z+v+w
```

vs.

```
sum5 :: ??
sum5 = ??
```

Currificación

- ¿Qué pasa con un ejemplo más grande? (cont.)
  - Con nombres intermedios...

```
sum5 :: Int -> (Int -> (Int -> (Int -> (Int -> Int))))
sum5 x = sum4
        where sum4 y = sum3
          where sum3 z = sum2
            where sum2 v = sum1
              where sum1 w = x+y+z+v+w
```

Currificación

- ¿Qué pasa con un ejemplo más grande? (cont.)
  - Con aplicación reiterada...

```
sum5 :: Int -> (Int -> (Int -> (Int -> (Int -> Int))))
((((sum5 x) y) z) v) w = x+y+z+v+w
```

vs.

```
sum5' :: (Int, Int, Int, Int, Int) -> Int
sum5' (x,y,z,v,w) = x+y+z+v+w
```

Currificación

- ¿Cómo podemos evitar usar paréntesis? Convenciones de notación
  - La aplicación de funciones asocia a izquierda
  - El tipo de las funciones asocia a derecha

suma :: Int ->Int ->Int	suma :: Int -> (Int ->Int)
suma x y = x+y	(suma x) y = x+y

Currificación

- Por abuso de lenguaje

  suma :: Int ->Int ->Int

  suma x y = x+y

  suma es una función que toma dos enteros y retorna otro entero.

en lugar de

  suma :: Int -> (Int -> Int)

  (suma x) y = x+y

  suma es una función que toma un entero y devuelve una función, la cual toma un entero y devuelve otro entero.

Currificación

- Ventajas.
  - Mayor expresividad

    ```
    derive :: (Int -> Int) -> (Int -> Int)
    derive f x = (f (x+h) - f x) / h   where h = 0.0001
    ```

  - Aplicación parcial

    ```
    derive f            ( = \x -> (f (x+h) - f x) / h )
    ```

  - Modularidad para tratamiento de código
    - Al inferir tipos
    - Al transformar programas

Aplicación Parcial

- Definir un función que calcule la derivada n-ésima de una función

```
deriveN :: Int -> (Int -> Int) -> (Int -> Int)
deriveN 0 f = f
deriveN n f = deriveN (n-1) (derive f)
```

Aplicación parcial de derive.

- ¿Cómo lo haría con derive'?

Expresividad

- Definir un función que calcule la aplicación n veces de una función

```
many :: Int -> (a -> a) -> (a -> a)
many 0 f x = x
many n f x = many (n-1) f (f x)
```

- Se pueden definir muchas ideas ya vistas...

```
twice = many 2
deriveN n = many n derive
```

Currificación

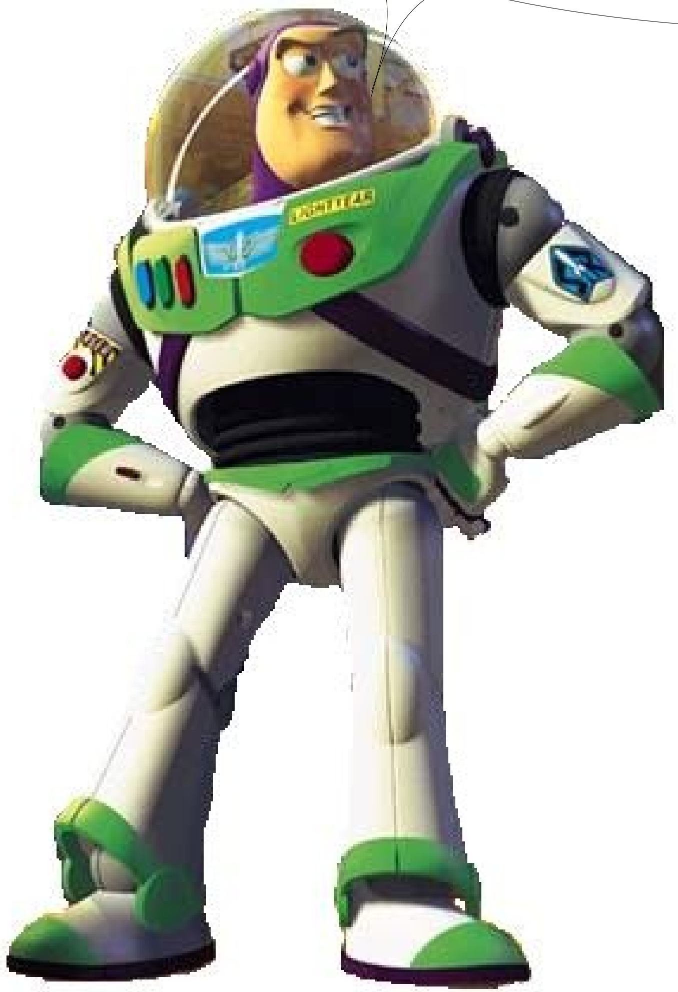
- Decir que algo está currificado es una CUESTIÓN DE INTERPRETACIÓN

```
movePoint :: (Int, Int) -> (Int, Int)
movePoint (x,y) = (x+1,y+1)

distance :: (Int, Int) -> Int
distance (x,y) = sqrt (sqr x + sqr y)
```

- ¿Están currificadas? ¿Por qué?

To infinity and beyond!



Inducción/Recursión

- Para solucionar los tres problemas, usaremos INDUCCIÓN
- La inducción es un mecanismo que nos permite:
  - Definir conjuntos infinitos
  - Probar propiedades sobre sus elementos
  - Definir funciones recursivas sobre ellos, con garantía de terminación

Inducción estructural

- Una definición inductiva de un conjunto $\mathfrak{R}$ consiste en dar condiciones de dos tipos:
  - reglas base ($z \in \mathfrak{R}$)
    - que afirman que algún elemento simple x pertenece a $\mathfrak{R}$
  - reglas inductivas ($y_1 \in \mathfrak{R}, \ldots, y_n \in \mathfrak{R} \Rightarrow y \in \mathfrak{R}$)
    - que afirman que un elemento compuesto y pertenece a $\mathfrak{R}$ siempre que sus partes $y_1, \ldots, y_n$ pertenezcan a $\mathfrak{R}$ (e y no satisface otra regla de las dadas)

  y pedir que $\mathfrak{R}$ sea el menor conjunto (en sentido de la inclusión) que satisfaga todas las reglas dadas.

Funciones recursivas

- Sea S un conjunto inductivo, y T uno cualquiera. Una definición recursiva *estructural* de una función f :: S -> T es una definición de la siguiente forma:
  - por cada elemento base z, el valor de (f z) se da directamente usando valores previamente definidos
  - por cada elemento inductivo y, con partes inductivas y1, ..., yn, el valor de (f y) se da usando valores previamente definidos ***y*** los valores (f y1), ..., (f yn).

Principio de inducción

- Sea S un conjunto inductivo, y sea P una propiedad sobre los elementos de S. ***Si se cumple que***:
  - para cada elemento $z \in$ S tal que z cumple con una regla base, P(z) es verdadero, y
  - para cada elemento $y \in$ S construído en una regla inductiva utilizando los elementos y_1, ..., y_n, si P(y_1), ..., P(y_n) son verdaderos entonces P(y) lo es,

entonces P(x) se cumple para todos los $x \in$ S.

$\forall x$.P(x) se demostró por *inducción estructural* en x

“...de más está decir que rehusarse a explotar este poder de las matemáticas concretas equivale a suicidio intelectual y tecnológico. La moraleja de la historia es: traten a todos los elementos de un conjunto ignorándolos y trabajando con la definición del conjunto.”

On the cruelty of really teaching computing science
(EWD 1036)
Edsger W. Dijkstra

Ejemplo: LISTAS

- Dado un tipo cualquiera a, definimos inductivamente al conjunto [a] con las siguientes reglas:
  - [] :: [a]
  - si x :: a y xs :: [a] entonces x:xs :: [a]
- ¿Qué elementos tiene [Bool]? ¿Y [Int]?
- Notación:

  $[\ x_1, x_2, x_3\ ] = (x_1 : (x_2 : (x_3 : [\ ])))$

Ejemplo: LISTAS

- Definir por recursión una función len que cuente los elementos de una lista.

```
len :: [a] -> Int
  -- Por inducción en la estructura de la lista xs
len [ ]      = ...
len (x:xs)   = ...  len xs ...
```

Caso base

Caso inductivo

Aplicación inductiva de len

Ejemplo: LISTAS

- Definir por recursión una función len que cuente los elementos de una lista.

```
len :: [a] -> Int
  -- Por inducción en la estructura de la lista xs
len [ ]       = 0
len (x:xs)  = 1+ len xs
```

Aplicación inductiva de len

Caso base

Caso inductivo

Funciones sobre listas

- Siguiendo el patrón de recursión

```
len :: [a] -> Int
len [ ]      = 0
len (x:xs) = 1 + len xs

append :: [a] -> [a] -> [a]
append [ ]      ys = ys
append (x:xs) ys = x : (append xs ys)

(++) = append
```

Funciones sobre listas

- Sin seguir el patrón de recursión

```
head :: [a] -> a
head (x:xs) = x

tail :: [a] -> [a]
tail (x:xs) = xs

null :: [a] -> Bool
null [ ]      = True
null (x:xs)  = False
```

Funciones sobre listas

- Más funciones siguiendo el patrón de recursión

```
sum :: [Int] -> Int
sum [ ]      = 0
sum (n:ns) = n + sum ns

prod :: [ Int ] -> Int
prod [ ]      = 1
prod (n:ns) = n * prod ns
```

- ¿Por qué se puede definir (sum [ ]) y (prod [ ]) de esta manera?

Funciones sobre listas

- Más funciones siguiendo el patrón de recursión

```
upperl :: [Char] -> [Char]
upperl [ ]       = [ ]
upperl (c:cs) = (upper c) : (upperl cs)

novacias :: [ [ a ] ] -> [ [ a ] ]
novacias [ ]            = [ ]
novacias (xs:xss)  = if null xs then novacias xss
                                    else xs : novacias xss
```

Funciones sobre listas

- Siguiendo otro patrón de recursión

```
maximum :: [ a ] -> a
maximum [ x ]    = x
maximum (x:xs) = x `max` maximum xs

last :: [ a ] -> a
last [ x ]    = x
last (x:xs) = last xs
```

- ¿puede establecer cuál es el patrón?
- ¿por qué (maximum [ ]) no está definida?

Funciones sobre listas

- Otras funciones

```
reverse :: [ a ] -> [ a ]
reverse [ ]      = [ ]
reverse (x:xs) = reverse xs ++ [ x ]

insert :: a -> [ a ] -> [ a ]
insert x [ ]      = [ x ]
insert x (y:ys) = if x <= y  then x : (y : ys)
                             else y : (insert x ys)
```

“'Detrás de cada acontecimiento se esconde un truco de espejos. Nada es, todo parece. Escóndase, si quiere. Espíe por las ranuras. Alguien estará preparando otra ilusión. Las diferencias entre las personas son las diferencias entre las ilusiones que perciben.'

(Consejero, 121:6:33)”

El Fondo del Pozo
Eduardo Abel Giménez

Parametrización

- ¿Qué es un parámetro? Consideremos

$$f\ x = x + \boxed{1}$$

$$g\ x = x + \boxed{17}$$

$$h\ x = x + \boxed{42}$$

  - Sólo las partes recuadradas son distintas ¿podremos aprovechar ese hecho?

Parametrización

- ¿Qué es un parámetro?

$$
\begin{array}{lcl}
\texttt{f x = x+}\boxed{\texttt{1}} & & \texttt{f = genF 1} \\
\texttt{g x = x+}\boxed{\texttt{17}} \xrightarrow{\text{Abstracción}} & \texttt{(genF y) x = x+}\boxed{\texttt{y}} & \xrightarrow{\text{Instanciación}} \texttt{g = genF 17} \\
\texttt{h x = x+}\boxed{\texttt{42}} & & \texttt{h = genF 42}
\end{array}
$$

  - Sólo las partes recuadradas son distintas ¿podremos aprovechar ese hecho?
  - Técnica de los "recuadros"
  - Parámetro: valor que cambia en cada uso

Esquemas de funciones

- Probemos con funciones sobre listas
  - Escribir las siguientes funciones:

```
succl :: [ Int ] -> [ Int ]
-- suma uno a cada elemento de la lista
upperl :: [ Char ] -> [ Char ]
-- pasa a mayúsculas cada carácter de la lista
test :: [ Int ] -> [ Bool ]
-- cambia cada número por un booleano que
-- dice si el mismo es cero o no
```

  - ¿Observa algo en común entre ellas?

Esquemas de funciones

- Solución:

```
succl [ ]        = ...
succl (n:ns)     = ... n ... succl ns ...

upperl [ ]       = ...
upperl (c:cs)    = ... c ... upperl cs ...

test [ ]         = ...
test (x:xs)      = ... x ...  test xs ...
```

  - Usamos el esquema de recursión estructural sobre listas

Esquemas de funciones

- Solución:

```
succl [ ]        = [ ]
succl (n:ns)     = (n+1) :  succl ns

upperl [ ]       = [ ]
upperl (c:cs)    = upper c :  upperl cs

test [ ]         = [ ]
test (x:xs)      = (x==0) :  test xs
```

- Sólo las partes recuadradas son distintas... pero los círculos rojos “molestan”

Esquemas de funciones

- Técnica de los “recuadros” (extendida)

```
succl [ ]       = [ ]
succl (n:ns)    = (\n' -> n'+1) n : succl ns
upperl [ ]      = [ ]
upperl (c:cs)   = (\c' -> upper c') c : upperl cs
test [ ]        = [ ]
test (x:xs)     = (\n -> n ==0) x : test xs
```

  - Reescribimos los recuadros (azules) para que no dependan del contexto (círculos rojos)

Esquema de map

- La respuesta es sí:

```
map :: ??
map f [ ]      =  [ ]
map f (x:xs) = f x : map f xs
```

- Y entonces

```
succl'   = map (\n' -> n'+1)
upperl' = map upper
test'     = map (==0)
```

- ¿Podría probar que succl' = succl? ¿Cómo?

Esquema de map

- **Demostración**
  - Por principio de extensionalidad, probamos que para toda lista finita xs, succl' xs = succl xs por inducción en la estructura de la lista.
  - Caso base: xs = [ ]
    - Usar succl', map.1, y succl.1
  - Caso inductivo: xs = x:xs'
    - Usar succl', map.2, succl', HI, y succl.2
- ¡Observar que no estamos contemplando el caso ⊥ ni el de listas no finitas, o con elementos ⊥!

Esquemas de funciones

- Una vez más, con otras funciones
  - Escribir las siguientes funciones:

```
masQueCero :: [ Int ] -> [ Int ]
-- retorna la lista que sólo contiene los números
-- mayores que cero, en el mismo orden

digitos :: [ Char ] -> [ Char ]
-- retorna los caracteres que son dígitos

noVacias :: [ [a] ] -> [ [a] ]
-- retorna sólo las listas no vacías
```

  - ¿Observa algo en común entre ellas?

Esquemas de funciones

- Solución:

```
digitos [ ]       = ...
digitos (c:cs)  =
            ... c ... digitos cs ...


noVacias [ ]         = ...
noVacias (xs:xss) =
            ... xs ... noVacias xss ...
```

  - Siempre recursión estructural

Esquemas de funciones

- Solución:

```
digitos [ ]      = [ ]
digitos (c:cs)  =
   if (isDigit c) then c : digitos cs
                  else digitos cs

noVacias [ ]        = [ ]
noVacias (xs:xss) =
   if (null xs) then noVacias xss
               else xs : noVacias xss
```

  - Otra vez, técnica de los “recuadros” extendida

Esquemas de funciones

- Solución:

```
digitos [ ]        = [ ]
digitos (c:cs)  =
   if (\c' -> isDigit c') c   then c : digitos cs
                              else digitos cs

noVacias [ ]          = [ ]
noVacias (xs:xss) =
   if (\xs' -> not (null xs')) xs   then xs : noVacias xss
                                    else noVacias xss
```

- Observar el cambio en el if de noVacias para que ambas funciones se parezcan

Esquema de filter

- La respuesta es sí:

```
filter :: ??
filter p [ ]        = [ ]
filter p (x:xs)  = if  (p x)  then x : filter p xs
                                else filter p xs
```

- Y entonces

```
masQueCero' = filter (>0)
digitos'         = filter isDigit
noVacias'       = filter (not . null)
```

- ¿Podría probar que noVacias' = noVacias?

Esquemas de funciones

- Una vez más, con más complejidad
  - Escribir las siguientes funciones:

```
sonCincos :: [ Int ] -> Bool
-- dice si todos los elementos son 5
cantTotal :: [ [a] ] -> Int
-- dice cuántos elementos de tipo a hay en total
concat :: [ [a] ] -> [ a ]
-- hace el append de todas las listas en una
```

  - ¿Observa algo en común entre ellas?
    ¿Qué es?

Esquemas de funciones

- Solución:

```
sonCincos [ ]       = ...
sonCincos (n:ns) =
            … n ... sonCincos ns ...

concat [ ]          = ...
concat (xs:xss)  =
            ... xs ...  concat xss ...
```

  - Recursión estructural

Esquemas de funciones

- Aplicando la técnica de las cajas

```
sonCincos [ ]      = True
sonCincos (n:ns) =
        n ==5 && sonCincos ns

concat [ ]          = [ ]
concat (xs:xss)  =
        xs ++ concat xss
```

  - Las cajas son más complicadas, pero la técnica es la misma

Esquemas de funciones

- Aplicando la técnica de las cajas

```
sonCincos [ ]      = True
sonCincos (n:ns) =
   (\x b -> x==5 && b) n (sonCincos ns)

concat [ ]          = [ ]
concat (xs:xss)  =
   (\ys zs -> ys ++ zs) xs ( concat xss )
```

  - Las cajas son más complicadas, pero la técnica es la misma

Esquema de recursión (fold)

- ¿Podemos aprovecharlo?

```
foldr :: ??
foldr f z [ ]      = z
foldr f z (x:xs)  = f x (foldr f z xs)
```

- Y entonces

```
sonCincos' = foldr check True
             where check x b = (x==5) && b
cantTotal' = foldr ((+) . len) 0
concat' = foldr (++) [ ]
```

- ¿Podría probar que concat' = concat?

Esquemas de funciones

- ¿Qué ventajas tiene trabajar con esquemas?

  Permite
  - definiciones más concisas y modulares
  - reutilizar código
  - demostrar propiedades generales
- ¿Qué requiere trabajar con esquemas?
  - Familiaridad con funciones de alto orden
  - Detección de características comunes (¡ABSTRACCIÓN!)

Esquemas y alto orden

- ¿Cómo definir append con foldr?

```
append :: [a] -> ([a] -> [a])
append []     = \ys -> ys
append (x:xs) = \ys -> x : append xs ys
```

- Expresado así, es rutina:

```
\ys -> x : append xs ys =
        (\x' h -> \ys -> x' : h ys) x (append xs)
```

y entonces

```
append = foldr (\x h ys -> x : h ys) id
       = foldr (\x h -> (x:) . h) id = foldr ((.) . (:)) id
```

Esquemas y alto orden

- ¿Cómo definir take con foldr?

```
take :: Int -> [a] -> [a]
take _ []      = []
take 0 (x:xs) = []
take n (x:xs) = x : take (n-1) xs
```

¡El n cambia en cada paso!

- Primero debo cambiar el orden de los argumentos

```
take' :: [a] -> (Int -> [a])
take' []     = \_ -> []
take' (x:xs) = \n -> case n of 0 -> []
                               _ -> x : take' xs (n-1)
```

Esquemas y alto orden

- ¿Cómo definir take con foldr? (Cont.)

```
take' :: [a] -> (Int -> [a])
take' = foldr g (const [])
  where g _ _ 0 = []
        g x h n = x : h (n-1)
```

y entonces

```
take :: Int -> [a] -> [a]
take = flip take'

flip f x y = f y x
```

Esquemas y alto orden

- Un ejemplo más: la función de Ackerman

  (¡con notación unaria!)

```
data One = One

ack :: Int -> Int -> Int
ack n m = u2i (ack' (i2u n) (i2u m))
            where i2u n = repeat n One
                  u2i = length

ack' :: [ One ] -> [ One ] -> [ One ]
ack' []       ys     = One : ys
ack' (x:xs) []       = ack' xs [ One ]
ack' (x:xs) (y:ys) = ack' xs (ack' (x:xs) ys)
```

Esquemas y alto orden

- La función de Ackerman (cont.)

```
ack' :: [ One ] -> [ One ] -> [ One ]
ack' []       = \ys -> One : ys
ack' (x:xs) = g
   where g []       = ack' xs [ One ]
            g (y:ys) = ack' xs (g ys)
```

- Reescribimos ack' (x:xs) = g como un foldr

```
ack' (x:xs) = foldr (\_ -> ack' xs) (ack' xs [ One ])
```

Esquemas y alto orden

- Y finalmente podemos definir ack' con foldr

```
ack' :: [ One ] -> [ One ] -> [ One ]
ack' = foldr (const g) (One :)
        where g h = foldr (const h) (h [ One ])
```

- Con esto podemos ver que la función de Ackerman termina para todo par de números naturales.

Esquemas en otros tipos

- Los esqumas de recursión también se pueden definir para otros tipos.
- Los naturales son un tipo inductivo.

```
foldNat :: (b -> b) -> b -> Nat -> b
foldNat s z 0 = z
foldNat s z n = s (foldNat s z (n-1))
```

- Los casos de la inducción son cero y el sucesor de un número, y por eso los argumentos del foldNat.

Recursión Primitiva (Listas)

- No toda función sobre listas es definible con foldr.
- Ejemplos:

```
tail :: [a] -> [a]
tail (x:xs) = xs

insert :: a -> [a] -> [a]
insert x []       = [x]
insert x (y:ys) = if x<y then (x:y:ys) else (y:insert x ys)
```

  - (**Nota:** en listas es complejo de observar. La recursión primitiva se observa mejor en árboles.)

Recursión Primitiva (Listas)

- El problema es que, además de la recursión sobre la cola, ¡utilizan la misma cola de la lista!
- Solución

```
recr :: b -> (a -> [a] -> b -> b) -> [a] -> b
recr z f []     = z
recr z f (x:xs) = f x xs (recr z f xs)
```

- Entonces

```
tail = recr (error “Lista vacía”) (\_ xs _ -> xs)
insert x = recr [x] (\y ys zs -> if x<y then (x:y:ys)
                                         else (y:zs))
```

Recursión Primitiva (Nats)

- Recursión primitiva sobre naturales

```
recNat :: b -> (Nat -> b -> b) -> Nat -> b
recNat z f 0 = z
recNat z f n = f (n-1) (recNat z f (n-1))
```

- Ejemplos (no definibles como foldNat)

```
fact = recNat 1 (\n p -> (n+1)*p)
```

-- fact n = $\prod_{i=1}^{n} i$

```
sumatoria f = recNat 0 (\x y -> f (x+1) + y)
```

-- sumatoria f n = $\sum_{i=1}^{n} f\ i$

“We claim that advanced data structures and algorithms can be better taught at the functional paradigm than at the imperative one.”

"A Second Year Course on Data Structures Based on Functional Programming"

M. Núñez, P. Palao y R. Peña

Functional Programming Languages in Education, LNCS 1022

Definición de Tipos 1

- Para definir un tipo de datos podemos:
  - establecer qué *forma* tendrá cada *elemento*, y
  - dar un *mecanismo único* para *inspeccionar* cada elemento
  - entonces: TIPO ALGEBRAICO

ó

  - determinar cuáles serán las *operaciones* que manipularán los elementos, SIN decir cuál será la forma exacta de éstos o aquéllas
  - entonces: TIPO ABSTRACTO

Tipos Algebraicos

- ¿Cómo damos en Haskell la forma de un elemento de un tipo algebraico?
  - Mediante **constantes** llamadas *constructores*
    - nombres con mayúsculas
    - no tienen asociada una regla de reducción
    - pueden tener argumentos
- Ejemplos:

False :: Bool　　　　True :: Bool

Tipos Algebraicos

- La cláusula data
  - introduce un nuevo tipo algebraico
  - introduce los nombres de los constructores
  - define los tipos de los argumentos de los constructores
- Ejemplos:

```
data Sensacion = Frio | Calor
data Shape = Circle Float | Rect Float Float
```

Tipos Algebraicos

- data Shape = Circle Float | Rect Float Float

Ejemplos de elementos:

```
c1 = Circle 1.0
c2 = Circle (4.0-3.0)
r1 = Rect 2.5  3.0
```

Ejemplos de funciones que arman Shapes:

```
circuloPositivo x = Circle (abs x)
cuadrado x = Rect x x
```

Tipos Algebraicos

- data Shape = Circle Float | Rect Float Float

  Ejemplo de alto orden:

```
construyeShNormal :: (Float -> Shape) -> Shape
construyeShNormal c = c 1.0
```

Uso de funciones de alto orden:

```
c3 = construyeShNormal circuloPositivo
c4 = construyeShNormal cuadrado
c5 = construyeShNormal (Rect 2.0)
```

- ¿Cuál es el tipo de Circle? ¿Y el de Rect?

Pattern Matching

- ¿Cuál es el mecanismo único de acceso?
  - *Pattern matching* (correspondencia de patrones (?))
- Pattern: expresión especial
  - sólo con constructores y variables sin repetir
  - argumento en el lado izquierdo de una ecuación
- Matching: operación asociada a un pattern
  - inspecciona el valor de una expresión
  - puede fallar o tener éxito
  - si tiene éxito, liga las variables del pattern

Pattern Matching

- Ejemplos:

```
area :: Shape -> Float
area (Circle radio) = pi * radio^2
area (Rect base altura) = base * altura

isCircle :: Shape -> Bool
--isCircle1 (Circle radio) = True
--isCircle1 (Rectangle base altura) = False
isCircle (Circle _) = True
isCircle _ = False
```

Pattern Matching

- Uso de pattern matching:
  - Al evaluar (area (circuloPositivo (-3.0)))
    - se reduce (circuloPositivo (-3.0)) a (Circle 3.0)
    - luego se verifica cada ecuación, para hacer el matching
    - si lo hace, la variable toma el valor correspondiente
  - radio se liga a 3.0, y la expresión retorna 28.2743
- ¿Cuánto valdrá (area (cuadrado 2.5))?
- ¿Y (area c2)?

Tuplas

- Son tipos algebraicos con sintaxis especial

```
fst :: (a,b) -> a
fst (x,y) = x

snd :: (a,b) -> b
snd (x,y) = y

distance :: (Float, Float) -> Float
distance (x,y) = sqrt (x^2 + y^2)
```

- ¿Cómo definir distance sin usar pattern matching?

```
distance p = sqrt ((fst p)^2 + (snd p)^2)
```

Tipos Algebraicos

- Pueden tener argumentos de tipo
- Ejemplo:

  ```
  data Maybe a = Nothing | Just a
  ```

- ¿Qué elementos tiene (Maybe Bool)? ¿Y (Maybe Int)?
- En general:
  - tiene los mismos elementos que el tipo a (pero con Just adelante) más uno adicional (Nothing)

Tipos Algebraicos

- ¿Para qué se usa el tipo `Maybe`?
- Ejemplo:

```
buscar :: clave -> [(clave,valor)] -> valor
buscar k [] = error "La clave no se encontró"
            -- Única elección posible con polimorfismo!
buscar k ((k',v):kvs) = if k==k'
                        then v
                        else buscar k kvs
```

- ¿La función `buscar` es total o parcial?

Tipos Algebraicos

- ¿Para qué se usa el tipo Maybe?
- Ejemplo:

```
lookup :: clave -> [(clave,valor)] -> Maybe valor
lookup k []               = Nothing
lookup k ((k',v):kvs) = if k==k'
                          then Just v
                          else lookup k kvs
```

- ¿La función lookup es total o parcial?

Tipos Algebraicos

- El tipo Maybe
  - permite expresar la posibilidad de que el resultado sea erróneo, sin necesidad de usar 'casos especiales'
  - evita el uso de ⊥ hasta que el programador decida, permitiendo controlar los errores

```
sueldo :: Nombre -> [Empleado] -> Int
sueldo nombre empleados
      = analizar (lookup nombre empleados)
analizar Nothing = error "No es de la empresa!"
analizar (Just s) = s
```

Tipos Algebraicos

- El tipo Maybe (cont.)
  - evita el uso de ⊥ hasta que el programador decida, permitiendo controlar los errores

```
sueldoGUI :: Nombre -> [Empleado] -> GUI Int
sueldoGUI nombre empleados =
    case (lookup nombre empleados) of
      Nothing -> ventanaError "No es de la empresa!"
      Just s    -> mostrarInt “El sueldo es ” s
```

Tipos Algebraicos

- Otro ejemplo:

  data Either a b = Left a | Right b

- ¿Qué elementos tiene (Either Int Bool)?
- En general:
  - representa la unión disjunta de dos conjuntos (los elementos de uno se identifican con Left y los del otro con Right)

Tipos Algebraicos

- ¿Para qué sirve Either?
  - Para mantener el tipado fuerte y poder devolver elementos de distintos tipos
    - Ejemplo: [Left 1, Right True] :: [Either Int Bool]
  - Para representar el origen de un valor
    - Ejemplo: lectora de temperaturas

```
data Temperatura = Celsius Int | Fahrenheit Int

convertir :: Either Int Int -> Temperatura
convertir (Left t)    = Celsius t
convertir (Right t) = Fahrenheit t
```

Tipos Algebraicos

- ¿Por qué se llaman tipos algebraicos?
- Por sus características:
  - toda combinación válida de constructores y valores es elemento de un tipo algebraico (y sólo ellas lo son)
  - dos elementos de un tipo algebraico son iguales si y sólo si están construídos utilizando los mismos constructores aplicados a los mismos valores

Tipos Algebraicos

- Expresividad: números complejos
  - Toda combinación de dos flotantes es un complejo
  - Dos complejos son iguales si tienen las mismas partes real e imaginaria

```
data Complex = C Float Float

realPart, imagePart :: Complex -> Float
realPart (C r i) = r
imagePart (C r i) = i

mkPolar :: Float -> Float -> Complex
mkPolar r theta = C (r * cos theta) (r * sin theta)
```

Tipos Algebraicos

- Expresividad: números racionales
  - No todo par de enteros es un número racional (R 1 0)
  - Hay racionales iguales con distinto numerador y denominador (R 4 2 = R 2 1)

```
data NoRacional = R Int Int
numerador, denominador :: NoRacional -> Int
numerador (R n d) = n
denominador (R n d) = d
```

- ¡No se puede representar a los racionales como tipo algebraico!

Tipos Algebraicos

- Expresividad: ejemplos
  - Se pueden armar tipos ad-hoc, combinando las ideas

```
data Helado = Vasito Gusto
            | Cucurucho Gusto Gusto (Maybe Baño)
            | Capelina Gusto Gusto [Agregado]
            | Pote Gusto Gusto Gusto
data Gusto = Chocolate | …
data Agregado = Almendras | Rocklets | …
data Baño = Blanco | Negro
```

- Así se pueden expresar elementos de dominios específicos

Tipos Algebraicos

- Expresividad: ejemplos
  - Se pueden armar funciones por pattern matching

```
precio :: Helado -> Float
precio h = costo h * 0.3 + 5

costo :: Helado -> Float
costo (Vasito g) = 1 + costoGusto g
costo (Cucurucho g1 g2 mb) = 2 + costoGusto g1
                               + costoGusto g2
                               + costoBaño mb
...
```

Tipos Algebraicos

- Expresividad: ejemplos
  - Se pueden armar funciones por pattern matching (cont.)

```
costoGusto :: Gusto -> Float
costoGusto Chocolate = 2
…

costoBaño :: Maybe Baño - > Float
costoBaño Nothing         = 0
costoBaño (Just Negro)   = 2
costoBaño (Just Blanco)  = 1
```

Tipos Algebraicos

- Podemos clasificarlos en:
  - Enumerativos (Sensacion, Bool)
    - Sólo constructores sin argumentos
  - Productos (Complex, Tuplas)
    - Un único constructor con varios argumentos
  - Sumas (Shape, Maybe, Either)
    - Varios constructores con argumentos
  - Recursivos (Listas)
    - Utilizan el tipo definido como argumento

Tipos de Datos Recursivos

- Un tipo algebraico recursivo
  - tiene al menos uno de los constructores con el tipo que se define como argumento
  - es la concreción en Haskell de un conjunto definido inductivamente
- Ejemplos:

```
data N = Z | S N
data BE = TT | FF | AA BE BE | NN BE
```

- ¿Qué elementos tienen estos tipos?

Tipos de Datos Recursivos

- Cada constructor define un caso de una definición inductiva de un conjunto.
  - Si tiene al tipo definido como argumento, es un *caso inductivo*, si no, es un *caso base*.
- El pattern matching
  - provee análisis de casos
  - permite acceder a los elementos inductivos que forman a un elemento dado
- Por ello, se pueden definir funciones recursivas

Tipos de Datos Recursivos

- Ejemplo: data N = Z | S N
    - Asignación de significado

```
evalN :: N -> Int
evalN Z       = ...
evalN (S x) = ... evalN x ...
```

    - ¡Usamos recursión estructural!

Tipos de Datos Recursivos

- Ejemplo: data N = Z | S N (cont.)
  - Asignación de significado

```
evalN :: N -> Int
evalN Z      = 0
evalN (S x) = 1 + evalN x
```

  - El tipo N es notación unaria para expresar números enteros (Int)

Tipos de Datos Recursivos

- Ejemplo: data N = Z | S N (cont.)
  - Manipulación simbólica

```
addN :: N -> N -> N
addN Z     m = ...
addN (S n) m = ... (addN n m) ...
```

  - Otra vez usamos recursión estructural

Tipos de Datos Recursivos

- Ejemplo: data N = Z | S N (cont.)
  - Manipulación simbólica

```
addN :: N -> N -> N
addN Z      m  = m
addN (S n) m  = S (addN n m)
```

  - No hay significado en sí mismo en esta manipulación

Tipos de Datos Recursivos

- Ejemplo: data N = Z | S N (cont.)
  - Coherencia de significación con manipulación
  - ¿Puede probarse la siguiente propiedad?
    Sean n,m::N finitos, cualesquiera; entonces

    evalN (addN n m) = evalN n + evalN m
  - ¿Cómo? ...
  - La propiedad captura el vínculo entre el significado y la manipulación simbólica

Tipos de Datos Recursivos

- Por inducción estructural
  (pues el tipo representa a un conjunto inductivo)
- **Demostración**: por inducción en la estructura de n
  - <u>Caso base</u>: n = Z
    - Usar addN.1, 0 neutro de (+) y evalN.1
  - <u>Caso inductivo</u>: n = S n'
  - HI: size (addN n' m) = size n' + size m
    - Usar addN.2, evalN.2, HI, asociatividad de (+), y evalN.2

Listas

- Una definición equivalente a la de listas

  data List a = Nil | Cons a (List a)

- La sintaxis de listas es equivalente a la de esta definición:
  - [ ] es equivalente a Nil
  - (x:xs) es equivalente a (Cons x xs)
- Sin embargo, (List a) y [a] son tipos distintos

Listas

- Considerar las definiciones

```
sum :: [Int] -> Int              -- Significado
sum [ ]        = 0
sum (n:ns)     = n + sum ns

(++) :: [a] -> [a] -> [a]        -- Manipulación simbólica
[ ] ++ ys      = ys
(x:xs) ++ ys   = x : (xs ++ ys)
```

- Coherencia entre significado y manipulación:
  Demostrar que para todas xs, ys listas finitas vale que:

  sum (xs ++ ys) = sum xs + sum ys

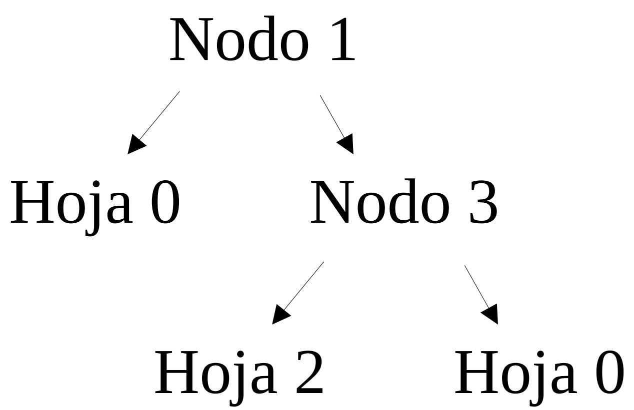
Árboles

- Un árbol es un tipo algebraico tal que al menos un elemento compuesto tiene dos componentes inductivas
- Se pueden usar TODAS las técnicas vistas para tipos algebraicos y recursivos
- Ejemplo:
  data Arbol a = Hoja a | Nodo a (Arbol a) (Arbol a)
- ¿Qué elementos tiene el tipo (Arbol Int)?

Árboles

- Si representamos elementos de tipo `Arbol Int` mediante diagramas jerárquicos

```
aej = Nodo 1 (Hoja 0)
             (Nodo 3 (Hoja 2) (Hoja 0))
```



Árboles

- ¿Cuántas hojas tiene un (Arbol a)?

```
hojas :: Arbol a -> Int
hojas (Hoja x)          = ...
hojas (Nodo x t1 t2) = ... hojas t1 ... hojas t2 ...
```

- ¿Y cuál es la altura de un (Arbol a)?

```
altura :: Arbol a -> Int
altura (Hoja x)          = ...
altura (Nodo x t1 t2) = ... altura t1 ... altura t2 ...
```

- Observar el uso de recursión estructural

Árboles

- ¿Cuántas hojas tiene un (Arbol a)?

```
hojas :: Arbol a -> Int
hojas (Hoja x)          = 1
hojas (Nodo x t1 t2) = hojas t1 + hojas t2
```

- ¿Y cuál es la altura de un (Arbol a)?

```
altura :: Arbol a -> Int
altura (Hoja x)          = 0
altura (Nodo x t1 t2) = 1+ (altura t1 `max` altura t2)
```

- ¿Puede mostrar que para todo árbol finito a, hojas a ≤ 2^(altura a)? ¿Cómo?

Árboles

- ¿Cómo reemplazamos una hoja?
- Ej: Cambiar los 2 en las hojas por 3.

```
cambiar2 :: Arbol Int -> Arbol Int
cambiar2 (Hoja n) = …


cambiar2 (Nodo n t1 t2) =
            … (cambiar2 t1) … (cambiar2 t2) …
```

- ¡Más recursión estructural!

Árboles

- ¿Cómo reemplazamos una hoja?
- Ej: Cambiar los 2 en las hojas por 3.

```
cambiar2 :: Arbol Int -> Arbol Int
cambiar2 (Hoja n) = if n==2
                     then Hoja 3
                     else Hoja n
cambiar2 (Nodo n t1 t2) =
          Nodo n (cambiar2 t1) (cambiar2 t2)
```

- ¿Cómo trabaja cambiar2? Reducir (cambiar2 aej)

Árboles

- Más funciones sobre árboles

```
duplA :: Arbol Int -> Arbol Int
duplA (Hoja n) = Hoja (n*2)
duplA (Nodo n t1 t2) =
          Nodo (n*2) (duplA t1) (duplA t2)
sumA :: Arbol Int -> Int
sumA (Hoja n) = n
sumA (Nodo n t1 t2) = n + sumA t1 + sumA t2
```

- ¿Cómo evalúa la expresión (duplA aej)?
- ¿Y (sumA aej)?

Árboles

- Recorridos de árboles
  - Transformación de un árbol en una lista (estructura no lineal a estructura lineal)

```
inOrder, preOrder :: Arbol a -> [ a ]
inOrder (Hoja n) = [ n ]
inOrder (Nodo n t1 t2) =
        inOrder t1 ++ [ n ] ++ inOrder t2

preOrder (Hoja n) = [ n ]
preOrder (Nodo n t1 t2) =
        n : (preOrder t1 ++ preOrder t2)
```

  - ¿Cómo sería posOrder?

Expresiones Aritméticas

- Definimos expresiones aritméticas
  - constantes numéricas
  - sumas y productos de otras expresiones

```
data ExpA = Cte Int | Suma ExpA ExpA
                    | Mult ExpA ExpA
```

- Ejemplos:
  - 2 se representa (Cte 2)
  - (4*4) se representa (Mult (Cte 4) (Cte 4))
  - ((2*3)+4) se representa
    Suma (Mult (Cte 2) (Cte 3)) (Cte 4)

Expresiones Aritméticas

- Definimos expresiones aritméticas
  - alternativa más simbólica...

```
data ExpS = CteS N | SumaS ExpS ExpS
                   | MultS ExpS ExpS
```

- Ejemplos:
  - 2 se representa (CteS (S (S Z)))
  - comparar con (Cte 2), donde Int representa números como semántica y no como símbolos (como lo hace N)

Expresiones Aritméticas

- ¿Cómo dar el significado de una ExpA?

```
evalEA :: ExpA -> Int
evalEA (Cte n)          = ...
evalEA (Suma e1 e2) = evalEA e1 ... evalEA e2
evalEA (Mult e1 e2)  = evalEA e1 ... evalEA e2
```

  - Observar el uso de recursión estructural

Expresiones Aritméticas

- ¿Cómo dar el significado de una ExpA?

```
evalEA :: ExpA -> Int
evalEA (Cte n)         = n
evalEA (Suma e1 e2)  = evalEA e1 + evalEA e2
evalEA (Mult e1 e2)   = evalEA e1 * evalEA e2
```

- Reduzca:

```
evalEA (Suma (Mult (Cte 2) (Cte 3)) (Cte 4))

evalEA (Mult (Cte 2) (Suma (Cte 3) (Cte 4)))
```

Expresiones Aritméticas

- Comparar con la versión más simbólica

```
evalES :: ExpS -> Int
evalES (CteS n)          = evalN n
evalES (SumaS e1 e2)     = evalES e1 + evalES e2
evalES (MultS e1 e2)     = evalES e1 * evalES e2
```

  - Se observa el uso de la función de asignación semántica (evalN) a los números representados como N (símbolos)

Expresiones Aritméticas

- ¿Cómo simplificar una ExpA?
  Manipulación simbólica

```
simplEA :: ExpA -> ExpA
simplEA (Cte n)          = ...
simplEA (Suma e1 e2) = ... (simplEA e1) ...
                             (simplEA e2) ...
simplEA (Mult e1 e2)    = ... (simplEA e1) (simplEA e2)
```

  - Observar otra vez el uso de recursión estructural

Expresiones Aritméticas

- ¿Cómo simplificar una ExpA?
  Manipulación simbólica

```
simplEA :: ExpA -> ExpA
simplEA (Cte n)          = Cte n
simplEA (Suma e1 e2) = armarSuma (simplEA e1)
                                  (simplEA e2)
simplEA (Mult e1 e2)   = Mult (simplEA e1) (simplEA e2)

armarSuma (Cte 0) e  = e
armarSuma e (Cte 0)  = e
armarSuma e1 e2       = Suma e1 e2
```

Expresiones Aritméticas

- ¿La manipulación es correcta?
  - Coherencia entre
    significado y manipulación simbólica
  - Expresado mediante la siguiente propiedad

    para todo e se cumple que

    evalEA (simplEA e) = evalEA e

Expresiones con variables

- ¿Cómo agregar variables a las expresiones?
  - Nuevo constructor

```
type Variable = String

data NExp = Vble Variable
          | NCte Int
          | Add NExp NExp | Sub NExp NExp
          | Mul NExp NExp
          | Div NExp NExp  | Mod NExp NExp
```

  - Además agregamos nuevas operaciones

Expresiones con variables

- ¿Y para asignarles significado?
  - Necesitamos conocer el valor de las variables

    evalNExp :: NExp -> (Memoria -> Int)

  - ¡El significado es una función!
  - Observar el uso del alto orden y la currificación
  - ¿Qué es la memoria?

Expresiones con variables

- Tipo abstracto para representar la memoria

```
data Memoria                -- Tipo abstracto de datos
enBlanco :: Memoria
  -- Una memoria vacía, que no recuerda nada
cuantoVale :: Memoria -> Variable -> Maybe Int
  -- Retorna el valor recordado para la variable dada
recordar :: Memoria -> Variable -> Int -> Memoria
  -- Recuerda un valor paraa una variable
variables :: Memoria -> [ Variable ]
  -- Retorna las variables que recuerda
```

Expresiones con variables

- Semántica de expresiones con variables

```
evalN :: NExp -> (Memoria -> Int)
evalN (Vble x)      mem  =
       case (cuantoVale mem x) of
         Nothing   -> error ("variable "++x++" indefinida")
         Just v     -> v
evalN (NCte n)      mem   = n
evalN (Add e1 e2)  mem   = evalN e1 mem + evalN e2 mem
        ...
```

- Observar
  - que las variables complican la semántica
  - el uso de la currificación para pasar la memoria

Definición de LIS

- Definimos un Lenguaje Imperativo Simple
  - asignación de expresiones numéricas
  - sentencias if y while sobre expresiones booleanas
  - secuencia de sentencias
- Ejemplo:

```
a := n; facn := 1
while (a > 0)
  { facn := a * facn; a := a – 1 }
```

Definición de LIS (cont.)

- Definimos tipos algebraicos para representar un programa LIS

```
data Program = P Bloque
type Bloque = [Comando]
data Comando  = Skip
              | Assign Variable NExp
              | If BExp Bloque Bloque
              | While BExp Bloque
```

Definición de LIS (cont.)

- Usamos las NExp anteriores, y agregamos expresiones booleanas

```
data BExp = BCte Bool      | Not BExp
          | And BExp BExp  | Or BExp BExp
          | RelOp ROp NExp NExp

data ROp = Equal           | NotEqual
         | Greater         | Lower
         | GreaterEqual    | LowerEqual
```

Definición de LIS

- ¿Cómo queda el programa de ejemplo?

```
a := n; facn := 1
while (a > 0)
  { facn := a * facn; a := a – 1 }
```

se expresa como

```
P [ Assign "a" (Vble "n")
  , Assign "facn" (NCte 1)
  , While (RelOp Greater (Vble "a") (NCte 0))
      [ Assign "facn" (Mul (Vble "a") (Vble "facn"))
      , Assign "a" (Sub (Vble "a") (NCte 1))
      ]
  ]
```

Evaluador de LIS (cont.)

- Semántica de expresiones booleanas

```
evalB :: BExp -> (Memoria -> Bool)
evalB (BCte b) mem = ...
evalB (RelOp rop e1 e2) mem
   = ... rop ... (evalN e1 mem) ... (evalN e2 mem) ...
evalB (And e1 e2) mem
   = ... evalB e1 mem ... evalB e2 mem ...
...
```

- ¿Por qué hace falta la memoria para dar significado a una expresión booleana?

Evaluador de LIS (cont.)

- Semántica de expresiones booleanas

```
evalB :: BExp -> (Memoria -> Bool)
evalB (BCte b) mem = b
evalB (RelOp rop e1 e2) mem
   = evalROp rop (evalN e1 mem) (evalN e2 mem)
evalB (And e1 e2) mem
   = evalB e1 mem && evalB e2 mem
…
```

- Nuevamente, observar el uso de currificación
- ¿Y la función auxiliar evalROp?

Evaluador de LIS (cont.)

- Semántica de expresiones booleanas (cont.)

```
evalROp :: ROp -> (Int -> Int -> Bool)
evalROp Equal     = (==)
evalROp NotEqual  = (!=)
evalROp Greater   = (>)
…
```

- ¡Observar que el significado se define directamente como una función!

Evaluador de LIS (cont.)

- Semántica de programas LIS

```
evalP :: Program -> (Memoria -> Memoria)
evalP (P bloque) = evalBloque bloque

evalBloque [ ] = \mem -> mem
evalBloque (c:cs) =
        \mem -> let mem' = evalCom c mem
                in evalP cs mem'
```

- ¡¡El significado es una función!!
- ¡Observar cómo la secuencia de comandos ALTERA la memoria antes de proseguir!

Evaluador de LIS (cont.)

- Semántica de sentencias LIS

```
evalCom :: Comando -> (Memoria -> Memoria)
evalCom Skip  = ...
evalCom (Assign x ne)
        = ... (evalN ne mem) ...
evalCom (If be bl1 bl2)
        = ... (evalB be mem)
                  ... (evalBloque bl1 mem)
                  ... (evalBloque bl2 mem)
evalCom (While be p)
        = ...??
```

Evaluador de LIS (cont.)

- Semántica de sentencias LIS

```
evalCom :: Comando -> (Memoria -> Memoria)

evalCom Skip  = \mem -> mem

evalCom (Assign x ne)
         = \mem -> recordar mem x (evalN ne mem)

evalCom (If be bl1 bl2)
         = \mem -> if (evalB be mem)
                      then (evalBloque bl1 mem)
                      else (evalBloque bl2 mem)

evalCom (While be p)
       = evalCom (If be (p ++ [While be p]) [Skip])
```

¡No es estructural!

Manipulacion simbólica

- ¿Qué pasa con programas como éste?

```
p :: Program
p = P [ Assign “x” (Add (NCte 10) (NCte 7))
      , Assign “y” (Add (Sub (NCte 59) (Var “x”))
                        (Sub (NCte 2) (NCte 2)) ]
```

- ¿Se podría hacer más eficiente ANTES de ejecutarlo?
  - Constant folding, simplification

```
p = P [ Assign “x” (NCte 17))
      , Assign “y” (Sub (NCte 59) (Var “x”)) ]
```

Manipulación simbólica (cont.)

- Expresiones sin variables

```
groundNExp :: NExp -> Bool
```

- Simplificación y evaluación

```
simplifyNExp :: NExp -> NExp
evalGroundNExp :: NExp -> Int
  -- PRECOND: el argumento es ground
```

  - ¡simplify debería usar evalGroundNExp!

- Análisis simbólico del programa

```
optimize :: Program -> Program
```

Manipulación simbólica (cont.)

- Expresiones sin variables

```
groundNExp :: NExp -> Bool
groundNExp ...
```

Manipulación simbólica (cont.)

- Simplificación y evaluación

```
simplifyNExp :: NExp -> NExp

simplifyNExp …
    -- OBS: usa evalGroundNExp

evalGroundNExp :: NExp -> Int
  -- PRECOND: el argumento es ground
evalGroundNExp …
```

Manipulación simbólica (cont.)

- Análisis simbólico del programa

```
optimize :: Program -> Program
optimize ...
```

“'Todo es pasajero. La verdad depende del momento. Baje los ojos. Incline la cabeza. Cuente hasta diez. Descubrirá otra verdad.'

(Consejero, 74:96:3)”

El Fondo del Pozo
Eduardo Abel Giménez

Esquemas en árboles

- Esquema de map en árboles:

```
data Arbol a = Hoja a | Nodo a (Arbol a) (Arbol a)

mapArbol :: (a -> b) -> Arbol a -> Arbol b
mapArbol f (Hoja x)          = Hoja (f x)
mapArbol f (Nodo x t1 t2)  =
    Nodo (f x) (mapArbol f t1) (mapArbol f t2)
```

- ¿Cómo definiría la función que multiplica por 2 cada elemento de un árbol? ¿Y la que los eleva al cuadrado?

Esquemas en árboles

- Solución:

```
dupArbol :: Arbol Int  -> Arbol Int
dupArbol = mapArbol (*2)

cuadArbol :: Arbol Int -> Arbol Int
cuadArbol = mapArbol (^2)
```

- ¿Podría definir, usando mapArbol, una función que aplique dos veces una función dada a cada elemento de un árbol? ¿Cómo?

Esquemas en árboles

- La función foldr expresa el patrón de recursión estructural sobre listas como función de alto orden
- Todo tipo algebraico recursivo tiene asociado un patrón de recursión estructural
- ¿Existirá una forma de expresar cada uno de esos patrones como una función de alto orden?
- ¡Sí, pero los argumentos dependen de los casos de la definición!

Esquemas en árboles

- Ejemplo:

```
foldArbol :: (a->b) -> (a->b->b->b) -> Arbol a -> b
foldArbol f g (Hoja x)          = f x
foldArbol f g (Nodo x t1 t2) =
          g x (foldArbol f g t1) (foldArbol f g t2)
```

- ¿Cuál es el tipo de los constructores?

```
   Hoja :: a -> Arbol a
   Nodo :: a -> Arbol a -> Arbol a -> Arbol a
```

- ¿Qué similitudes observa con el tipo de foldArbol?

Esquemas en árboles

- Defina una función que sume todos los elementos de un árbol

```
sumArbol :: Arbol Int -> Int
sumArbol = foldArbol id (\n n1 n2 -> n1 + n + n2)
```

- ¿Podría identificar las llamadas recursivas?
- ¿Y si expandimos la definición de foldArbol?

```
sumArbol (Hoja x)          = id x
sumArbol (Nodo x t1 t2) =
                  sumArbol t1 + x + sumArbol t2
```

Esquemas en árboles

- Defina, usando foldArbol una función que:
  - cuente el número de elementos de un árbol

```
sizeArbol = foldArbol (\x->1) (\x s1 s2 -> 1+s1+s2)
```

  - cuente el número de hojas de un árbol

```
hojas = foldArbol (const 1) (\x h1 h2 -> h1+h2)
```

  - calcule la altura de un árbol

```
altura = foldArbol (\x->0) (\x a1 a2 -> 1 + max a1 a2)
```

  - ¿Puede identificar los llamados recursivos?
  - ¿Por qué el primer argumento es una función?

Árboles alfa-beta

- Considere la siguiente definición

```
data AB a b = Leaf b | Branch a (AB a b) (AB a b)
```

- Defina una función que cuente el número de bifurcaciones de un árbol

```
bifs :: AB a b -> Int
bifs (Leaf x)           = ...
bifs (Branch y t1 t2)   = ... bifs t1 ... bifs t2 ...
```

- Completamos con el significado...

Árboles alfa-beta

- Considere la siguiente definición

  ```
  data AB a b = Leaf b | Branch a (AB a b) (AB a b)
  ```

- Defina una función que cuente el número de bifurcaciones de un árbol

  ```
  bifs :: AB a b -> Int
  bifs (Leaf x)          = 0
  bifs (Branch y t1 t2)  = 1 + bifs t1 + bifs t2
  ```

- ¿Cómo sería el esquema de recursión asociado a un árbol AB?

Árboles alfa-beta

- ¡Utilizamos el esquema de recursión!

```
foldAB :: ??
foldAB f g (Leaf x)           = f x
foldAB f g (Branch y t1 t2) =
           g y (foldAB f g t1) (foldAB f g t2)
```

- ¿Cómo representaría la función bifs?

```
bifs' = foldAB (const 0) (\x n1 n2 -> 1+n1+n2)
```

- ¿Puede probar que bifs' = bifs?

Árboles alfa-beta

- Ejemplo de uso

```
type AExp = AB BOp Int
data BOp = Suma | Producto
```

- ¿Cómo definimos la semántica de AExp usando foldAB?

```
evalAE :: AExp -> Int
evalAE = foldAB id binOp

binOp :: BOp -> Int -> Int -> Int
binOp Suma      = (+)
binOp Producto = (*)
```

Árboles alfa-beta

- Ejemplo de uso

```
type Decision s a = AB (s->Bool) a
```

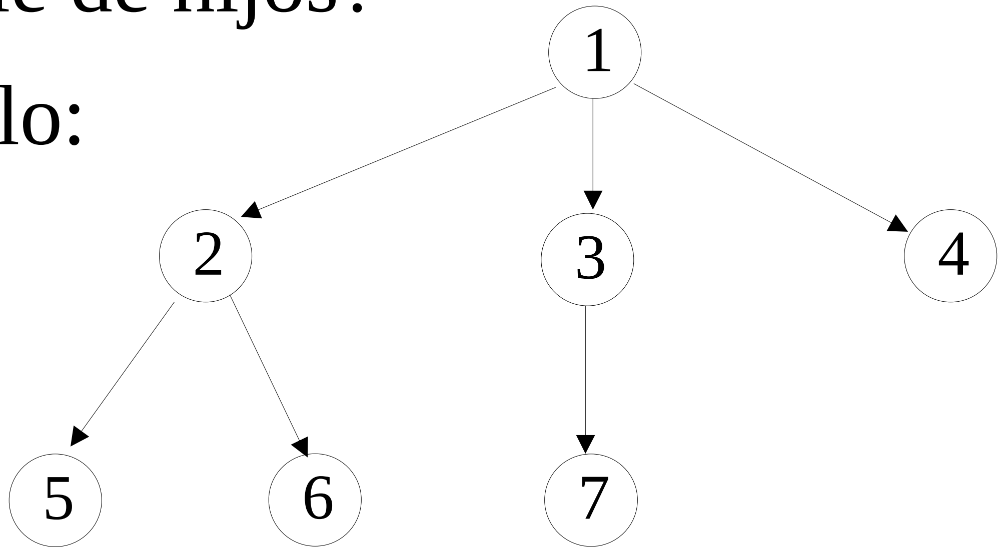
- Definamos una función que dada una situación, decida qué acción tomar basada en el árbol

```
decide :: situation -> Decision situation action -> action
decide s = foldAB id  (\f a1 a2 -> if (f s) then a1 else a2)

ej = Branch f1 (Leaf "Huya")
        (Branch f2 (Leaf "Trabaje") (Leaf "Quédese manso"))
       where f1 s = (s==Fuego) || (s==AtaqueExtraterrestre)
             f2 s = (s==VieneElJefe)
```

Árboles Generales

- ¿Cómo representar un árbol con un número variable de hijos?
- Ejemplo:



- Idea: ¡usar una lista de hijos!

Árboles Generales

- Ello nos lleva a la siguiente definición:

  data AG a = GNode a [AG a]

- Pero, ¿tiene caso base? ¿cuál?
  - Un árbol sin hijos...
- ¡Se basa en el esquema de recursión de listas!
  - O sea, el caso base es (GNode x [ ]); por ejemplo:

```
GNode 1  [  GNode 2 [ GNode 5 [ ], GNode 6 [ ] ]
         ,  GNode 3 [ GNode 7 [ ] ]
         ,  GNode 4 [ ]
         ]
```

Árboles Generales

- Definir una función que sume los elementos

  sumAG :: AG Int -> Int

- ¿Cómo la definimos?
  - ¡Usando funciones sobre listas!

  sumAG (GNode x ts) = x + sum (map sumAG ts)

- Y esto, ¿es estructural?
  - Sí, pues se basa en la estructura de las listas
- Se ve la utilidad de funciones de alto orden...

Árboles Generales

- ¿Cómo sería el esquema de recursión?
  Hay varias posibilidades
  - Según la receta de una función por constructor

    ```
    foldAG0 :: (a->[b]->b) -> AG a -> b
    foldAG0 h (GNode x ts) = h x (map (foldAG0 h) ts)
    ```

    y entonces, la función sumAG queda

    ```
    sumAG0 = foldAG0 (\x ns -> x + sum ns)
    ```

  - ¡El problema es que no es recursión estructural!

Árboles Generales

- ¿Cómo sería el esquema de recursión? (2)
  - Completamente estructural

```
foldAG1 :: (a->c->b) -> (b->c->c) -> c -> AG a -> b
foldAG1 g f z (GNode x ts) =
        g x (foldr f z (map (foldAG1 g f z) ts))
```

  y entonces, la función sumAG queda

```
sumAG1 = foldAG1 (+) (+) 0  -- sum = foldr (+) 0
```

  - Siempre termina, porque es estructural
  - ¡El problema es que es difícil de pensar!

Árboles Generales

- ¿Cómo sería el esquema de recursión? (3)
  - Opción intermedia entre ambas

```
foldAG :: (a->c->b) -> ([b]->c) -> AG a -> b
foldAG g k (GNode x ts) =
        g x (k (map (foldAG g k) ts))
```

y entonces, la función sumAG queda

```
sumAG = foldAG (+) sum
```

  - No es estructural, pero es bastante clara

Árboles Generales

- ¿Cuál es mejor? Depende del uso y el gusto

```
sumAG0 = foldAG0 (\x ns -> x + sum ns)

sumAG1 = foldAG1 (+) (+) 0  -- sum = foldr (+) 0

sumAG' = foldAG (+) sum
```

- Otras funciones sobre árboles generales:

```
depthAG = foldAG (\x d -> 1+d) (maxWith 0)
    where maxWith x []   = x
          maxWith x xs = maximum xs

mirrorAG = foldAG GNode reverse
```

“La tarea de un pensador no consistía para Shevek en negar una realidad a expensas de otra, sino en integrar y relacionar. No era una tarea fácil.”

Los Desposeídos
Úrsula K. Le Guin

“Enseñen a los niños a ser preguntones para que pidiendo el por qué de lo que se les manda, se acostumbren a obedecer a la razón, no a la autoridad como los limitados, ni a la costumbre como los estúpidos.”

Simón Rodríguez,
maestro del Libertador, Simón Bolívar.

“La persona que toma lo banal y lo ordinario y lo ilumina de una nueva forma, puede aterrorizar. No deseamos que nuestras ideas sean cambiadas. Nos sentimos amenazados por tales demandas. «¡Ya conocemos las cosas importantes!», decimos. Luego aparece el Cambiador y echa a un lado todas nuestras ideas.

-El Maestro Zensunni”

Casa Capitular: Dune
Frank Herbert

“Un mago sólo puede dominar lo que está cerca, lo que puede nombrar con la palabra exacta.”

Un mago de Terramar

Úrsula K. Le Guin

“ - Maestro - dijo Ged -, no soy tan vigoroso como para arrancarte el nombre por la fuerza, ni tan sabio como para sacártelo por la astucia. Me contento pues, con quedarme aquí y aprender o servir, lo que tú prefieras; a menos que consintieras, por ventura, a responder a una pregunta mía.

- Hazla.

- ¿Qué nombre tienes?

El Portero sonrió, y le dijo el nombre.”

Un mago de Terramar
Úrsula K. Le Guin

Cálculo Lambda Tipado (1/3)

Alejandro Ríos

Departamento de Computación, FCEyN, UBA

"There may, indeed, be other applications of the system other than its use as a logic", Alonzo Church, 1932

24 de agosto de 2017

¿Qué es el Cálculo Lambda?

- Modelo de computación basado en funciones
  - da origen a la programación funcional
- Introducido por Alonzo Church en 1934
- Computacionalmente completo (i.e. Turing completo)
- También considerado como modelo fiable de lenguajes de programación en general
  - The next 700 programming languages, Peter Landin, 1966
  - Lema: Usar Cálculo Lambda para probar nuevos conceptos de programación

¿Por qué Cálculo Lambda?

- En un lenguaje "industrial-strength" es difícil determinar con precisión/rigurosidad
  - propiedades básicas sobre semántica y sistema de tipos
  - efecto de extender el lenguaje con nuevas construcciones
  - la relación con otros lenguajes o paradigmas
- Es conveniente restringir el lenguaje a un subconjunto que sea
  - representativo (del paradigma o área de problemas)
  - conciso
  - reducido en cuanto a primitivas
  - riguroso en su formulación

¿Qué vamos a estudiar sobre Cálculo Lambda?

- La formulación original es sin tipos
- Dado nuestro interés en lenguajes de programación, vamos a estudiar el Cálculo Lambda Tipado (A. Church, 1941)
- En el marco del Cálculo Lambda Tipado vamos a presentar
  1. Tipos, términos, tipado, evaluación (Clase 1/3 y 2/3)
  2. Inferencia de tipos (Clase 3/3)
- Comenzaremos con Cálculo Lambda Tipado con expresiones booleanas y luego iremos enriqueciendo el lenguaje con otras construcciones

Expresiones de tipos de λ^b

Las expresiones de tipos (o simplemente tipos) de λ^b son

$$\sigma, \tau \quad ::= \quad \mathit{Bool} \,|\, \sigma \to \tau$$

Descripción informal:

- *Bool* es el tipo de los booleanos,
- $\sigma \to \tau$ es el tipo de las funciones de tipo σ en tipo τ

Términos de λ^b

Sea $\mathcal{X}$ un conjunto infinito enumerable de variables y $x \in \mathcal{X}$. Los términos de λ^b están dados por

$$
\begin{array}{rcl}
M, N, P, Q & ::= & x \\
& | & \mathit{true} \\
& | & \mathit{false} \\
& | & \mathit{if}\ \ M\ \ \mathit{then}\ \ P\ \ \mathit{else}\ \ Q \\
& | & \lambda x : \sigma.M \\
& | & M\,N
\end{array}
$$

Términos de λ^b

Descripción informal:

- x es una variable de términos,
- *true* y *false* son las constantes de verdad,
- *if* M *then* P *else* Q es el condicional,
- $\lambda x : \sigma.M$ es una función cuyo parámetro formal es x y cuyo cuerpo es M
- $M\,N$ es la aplicación de la función denotada por el término M al argumento N.

Ejemplos

- $\lambda x : Bool.x$
- $\lambda x : Bool.if\ \ x\ \ then\ \ false\ \ else\ \ true$
- $\lambda f : \sigma \to \tau.\lambda x : \sigma.f\ x$
- $(\lambda f : Bool \to Bool.f\ true)(\lambda y : Bool.y)$
- $x\ y$

Variables libres

Una variable puede ocurrir libre o ligada en un término. Decimos que "x" ocurre libre si no se encuentra bajo el alcance de una ocurrencia de "λx". Caso contrario ocurre ligada.

- $\lambda x : Bool.if \underbrace{x}_{ligada} then\ true\ else\ false$
- $\lambda x : Bool.\lambda y : Bool.if\ true\ then \underbrace{x}_{ligada} else \underbrace{y}_{ligada}$
- $\lambda x : Bool.if \underbrace{x}_{ligada} then\ true\ else \underbrace{y}_{libre}$
- $(\lambda x : Bool.if \underbrace{x}_{ligada} then\ true\ else\ false) \underbrace{x}_{libre}$

Variables libres: Definición formal

$$
\begin{aligned}
FV(x) &\stackrel{\text{def}}{=} \{x\} \\
FV(true) = FV(false) &\stackrel{\text{def}}{=} \emptyset \\
FV(if\ \ M\ \ then\ \ P\ \ else\ \ Q) &\stackrel{\text{def}}{=} FV(M) \cup FV(P) \cup FV(Q) \\
FV(M\,N) &\stackrel{\text{def}}{=} FV(M) \cup FV(N) \\
FV(\lambda x : \sigma . M) &\stackrel{\text{def}}{=} FV(M) \setminus \{x\}
\end{aligned}
$$

Sustitución

$$M\{x \leftarrow N\}$$

- *"Sustituir todas las ocurrencias libres de x en el término M por el término N"*
- Operación importante que se usa para darle semántica a la aplicación de funciones (entre otras)
- Es sencilla de definir pero requiere cuidado en el tratamiento de los ligadores de variables (i.e. con "λx")

Sustitución

$$
\begin{aligned}
x\{x \leftarrow N\} &\stackrel{\text{def}}{=} N \\
a\{x \leftarrow N\} &\stackrel{\text{def}}{=} a \quad \text{si } a \in \{\mathit{true}, \mathit{false}\} \cup \mathcal{X} \setminus \{x\} \\
(\mathit{if}\ M\ \mathit{then}\ P\ \mathit{else}\ Q)\{x \leftarrow N\} &\stackrel{\text{def}}{=} \mathit{if}\ M\{x \leftarrow N\} \\
&\qquad \mathit{then}\ P\{x \leftarrow N\} \\
&\qquad \mathit{else}\ Q\{x \leftarrow N\} \\
(M_1\, M_2)\{x \leftarrow N\} &\stackrel{\text{def}}{=} M_1\{x \leftarrow N\}\, M_2\{x \leftarrow N\} \\
(\lambda y : \sigma . M)\{x \leftarrow N\} &\stackrel{\text{def}}{=} \textbf{?}
\end{aligned}
$$

Captura de variables

"Sustituir la variable x por el término z"

$$(\lambda z : \sigma.x)\{x \leftarrow z\} = \lambda z : \sigma.z$$

- ¡Hemos convertido a la función constante $\lambda z : \sigma.x$ en la función identidad!
- El problema: "$\lambda z : \sigma$" capturó la ocurrencia libre de z
- Hipótesis: los nombres de las variables ligadas no son relevantes
  - la ecuación de arriba debería ser comparable con
    $$(\lambda w : \sigma.x)\{x \leftarrow z\} = \lambda w : \sigma.z$$
- Conclusión: Para definir $(\lambda y : \sigma.M)\{x \leftarrow N\}$ asumiremos que la variable ligada y se renombró de tal manera que no ocurre libre en N

α-equivalencia

- Dos términos M y N que difieren solamente en el nombre de sus variables ligadas se dicen *α-equivalentes*
- α-equivalencia es una relación de equivalencia
- De aquí en más identificaremos términos α-equivalentes.

- $\lambda x : Bool.x =_\alpha \lambda y : Bool.y$
- $\lambda x : Bool.y =_\alpha \lambda z : Bool.y$
- $\lambda x : Bool.y \neq_\alpha \lambda x : Bool.z$
- $\lambda x : Bool.\lambda x : Bool.x \neq_\alpha \lambda y : Bool.\lambda x : Bool.y$

Sustitución - Revisada

$$
\begin{array}{rcl}
x\{x \leftarrow N\} & \stackrel{\text{def}}{=} & N \\
a\{x \leftarrow N\} & \stackrel{\text{def}}{=} & a \quad \text{si } a \in \{\mathit{true}, \mathit{false}\} \cup \mathcal{X} \setminus \{x\} \\
(\mathit{if}\ M\ \mathit{then}\ P\ \mathit{else}\ Q)\{x \leftarrow N\} & \stackrel{\text{def}}{=} & \mathit{if}\ M\{x \leftarrow N\}\ \mathit{then}\ P\{x \leftarrow N\} \\
 & & \qquad\qquad\qquad \mathit{else}\ Q\{x \leftarrow N\} \\
(M_1\, M_2)\{x \leftarrow N\} & \stackrel{\text{def}}{=} & M_1\{x \leftarrow N\}\, M_2\{x \leftarrow N\} \\
(\lambda y : \sigma.M)\{x \leftarrow N\} & \stackrel{\text{def}}{=} & \lambda y : \sigma.M\{x \leftarrow N\}\ x \neq y,\ \ y \notin FV(N)
\end{array}
$$

1. NB: la condición $x \neq y,\ \ y \notin FV(N)$ siempre puede cumplirse renombrando apropiadamente
2. Técnicamente, la sust. está definida sobre clases de α-equivalencia de términos

Sistema de tipado

- Sistema formal de deducción (o derivación) que utiliza axiomas y reglas de tipado para caracterizar un subconjunto de los términos llamados tipados.
  - Los axiomas de tipado establecen que ciertos juicios de tipado son derivables.
  - Las reglas de tipado establecen que ciertos juicios de tipado son derivables siempre y cuando ciertos otros lo sean.
- Motivar juicios de tipado:
  - ¿Qué tipo le asignaría a *true*?
  - ¿Y a *if x then false else true*?

Sistema de tipado

Un contexto de tipado es un conjunto de pares $x_i : \sigma_i$, anotado $\{x_1 : \sigma_1, \dots, x_n : \sigma_n\}$ donde los $\{x_i\}_{i \in 1..n}$ son distintos. Usamos letras $\Gamma, \Delta, \dots$ para contextos de tipado.

Un juicio de tipado es una expresión de la forma $\Gamma \rhd M : \sigma$ que se lee:

> "*el término M tiene tipo σ asumiendo el contexto de tipado Γ*"

Sistema de tipado

- El significado de $\Gamma \rhd M : \sigma$ se establece a través de la introducción de axiomas y reglas de tipado.
- Si $\Gamma \rhd M : \sigma$ puede derivarse usando los axiomas y reglas de tipado decimos que es derivable.
- Decimos que M es tipable si el juicio de tipado $\Gamma \rhd M : \sigma$ puede derivarse, para algún Γ y σ.
- A continuación presentaremos los axiomas y reglas de tipado de λ^b

Axiomas de tipado de λ^b

$$\frac{x : \sigma \in \Gamma}{\Gamma \rhd x : \sigma}\ (\textsc{T-Var})$$

$$\frac{}{\Gamma \rhd \mathit{true} : \mathit{Bool}}\ (\textsc{T-True}) \qquad \frac{}{\Gamma \rhd \mathit{false} : \mathit{Bool}}\ (\textsc{T-False})$$

Reglas de tipado de λ^b

$$\frac{\Gamma \rhd M : Bool \quad \Gamma \rhd P : \sigma \quad \Gamma \rhd Q : \sigma}{\Gamma \rhd \mathit{if}\ M\ \mathit{then}\ P\ \mathit{else}\ Q : \sigma}\ (\textsc{T-If})$$

$$\frac{\Gamma, x : \sigma \rhd M : \tau}{\Gamma \rhd \lambda x : \sigma . M : \sigma \to \tau}\ (\textsc{T-Abs}) \qquad \frac{\Gamma \rhd M : \sigma \to \tau \quad \Gamma \rhd N : \sigma}{\Gamma \rhd M\, N : \tau}\ (\textsc{T-App})$$

Ejemplos de derivaciones de juicios de tipado

Vamos a mostrar que los siguientes juicios de tipado son derivables:

1. $\triangleright \lambda x : Bool.\lambda f : Bool \to Bool.f\ x :$
   $Bool \to (Bool \to Bool) \to Bool$
2. $x : Bool, y : Bool \triangleright \mathit{if}\ x\ \mathit{then}\ y\ \mathit{else}\ y : Bool$
3. $\triangleright \lambda f : \rho \to \tau.\lambda g : \sigma \to \rho.\lambda x : \sigma.f(g\, x) :$
   $(\rho \to \tau) \to (\sigma \to \rho) \to \sigma \to \tau$
4. ¿Existen Γ y σ tal que $\Gamma \triangleright x\, x : \sigma$?

Resultados básicos

Unicidad de tipos

Si $\Gamma \triangleright M : \sigma$ y $\Gamma \triangleright M : \tau$ son derivables, entonces $\sigma = \tau$

Weakening+Strengthening

Si $\Gamma \triangleright M : \sigma$ es derivable y $\Gamma \cap \Gamma'$ contiene a todas las variables libres de M, entonces $\Gamma' \triangleright M : \sigma$

Sustitución

Si $\Gamma, x : \sigma \triangleright M : \tau$ y $\Gamma \triangleright N : \sigma$ son derivables, entonces $\Gamma \triangleright M\{x \leftarrow N\} : \tau$ es derivable

Semántica

- Habiendo definido la sintaxis de λ^b, nos interesa formular cómo se evalúan o ejecutan los términos
- Hay varias maneras de definir rigurosamente la semántica de un lenguaje de programación
  - Operacional
  - Denotacional
  - Axiomática
- Vamos a definir una semántica operacional para λ^b

¿Qué es semántica operacional?

- Consiste en
  - interpretar a los términos como estados de una máquina abstracta y
  - definir una función de transición que indica, dado un estado, cuál es el siguiente estado
- Significado de un término M: el estado final que alcanza la máquina al comenzar con M como estado inicial
- Formas de definir semántica operacional
  1. Small-step: la función de transición describe un paso de computación
  2. Big-step (o Natural Semantics): la función de transición, en un paso, evalúa el término a su resultado

Semántica operacional

- La formulación se hace a través de juicios de evaluación

$$M \rightarrow N$$

  que se leen: "*el término M reduce, en un paso, al término N*"
- El significado de un juicio de evaluación se establece a través de:
  - Axiomas de evaluación: establecen que ciertos juicios de evaluación son derivables.
  - Reglas de evaluación establecen que ciertos juicios de evaluación son derivables siempre y cuando ciertos otros lo sean.

Semántica operacional small-step de λ^b

- Vamos a presentar una semántica operacional small-step para el cálculo λ^b
- Lo haremos por partes
  - Primero abordamos las expresiones booleanas
  - Luego el resto de las expresiones
- Además de introducir la función de transición es conveniente introducir también los valores
  - Valores: Los posibles resultados de evaluación de términos bien-tipados (¿por qué?) y cerrados (¿por qué?)

Semántica Operacional - Expr. booleanas

Valores

$$V \quad ::= \quad \mathit{true} \mid \mathit{false}$$

Todo término bien-tipado y cerrado de tipo *Bool* evalúa, en cero o más pasos, a *true* o *false*

- Este resultado se demuestra formalmente

Semántica Operacional - Expr. booleanas

Juicio de evaluación en un paso

$$\frac{}{\textit{if true then } M_2 \textit{ else } M_3 \to M_2} \ (\text{E-IfTrue})$$

$$\frac{}{\textit{if false then } M_2 \textit{ else } M_3 \to M_3} \ (\text{E-IfFalse})$$

$$\frac{M_1 \to M_1'}{\textit{if } M_1 \textit{ then } M_2 \textit{ else } M_3 \to \textit{if } M_1' \textit{ then } M_2 \textit{ else } M_3} \ (\text{E-If})$$

Ejemplos

$$\dfrac{\dfrac{}{\textit{if false then false else true} \rightarrow \textit{true}}\ (\textsc{E-IfFalse})}{\begin{array}{c}\textit{if (if false then false else true) then false else true} \\ \rightarrow \\ \textit{if true then false else true}\end{array}}\ (\textsc{E-If})$$

Observar que

- No existe M tal que $\textit{true} \rightarrow M$ (idem con *false*).

Ejemplos

$$\begin{array}{rl} & \mathit{if\ true\ then\ } (\mathit{if\ false\ then\ false\ else\ true})\ \mathit{else\ true} \\ \not\to & \mathit{if\ true\ then\ true\ else\ true} \end{array}$$

La estrategia de evaluación corresponde con el orden habitual en lenguajes de programación.

1. Primero evaluar la guarda del condicional
2. Una vez que la guarda sea un valor, seguir con la expresión del then o del else, según corresponda

Propiedades

Lema (Determinismo del juicio de evaluación en un paso)

Si $M \rightarrow M'$ y $M \rightarrow M''$, entonces $M' = M''$

Propiedades

Una forma normal es un término que no puede evaluarse más (i.e. M tal que no existe N, $M \to N$)

Recordar que un valor es el resultado al que puede evaluar un término bien-tipado y cerrado

Lema

Todo valor está en forma normal

- No valdrá el recíproco en λ^b (pero sí vale en el cálculo de las expresiones booleanas cerradas):
  - $\mathit{if}\ x\ \mathit{then}\ \mathit{true}\ \mathit{else}\ \mathit{false}$
  - x
  - $\mathit{true}\ \mathit{false}$

Evaluación en muchos pasos

El juicio de evaluación en muchos pasos $\twoheadrightarrow$ es la clausura reflexiva, transitiva de $\rightarrow$. Es decir, la menor relación tal que

1. Si $M \rightarrow M'$, entonces $M \twoheadrightarrow M'$
2. $M \twoheadrightarrow M$ para todo M
3. Si $M \twoheadrightarrow M'$ y $M' \twoheadrightarrow M''$, entonces $M \twoheadrightarrow M''$

$$\begin{array}{ll} & \mathit{if}\ \mathit{true}\ \mathit{then}\ (\mathit{if}\ \mathit{false}\ \mathit{then}\ \mathit{false}\ \mathit{else}\ \mathit{true})\ \mathit{else}\ \mathit{true} \\ \twoheadrightarrow & \mathit{true} \end{array}$$

Evaluación en muchos pasos - Propiedades

Para el cálculo de expresiones booleanas valen:

Lema (Unicidad de formas normales)

Si $M \twoheadrightarrow U$ y $M \twoheadrightarrow V$ con U, V formas normales, entonces $U = V$

Lema (Terminación)

Para todo M existe una forma normal N tal que $M \twoheadrightarrow N$

Semántica operacional de λ^b

Valores

$$V \quad ::= \quad \mathit{true} \mid \mathit{false} \mid \lambda x : \sigma . M$$

Introduciremos una noción de evalución en λ^b tal que valgan los lemas previos y también el siguiente resultado:

Teorema

Todo término bien-tipado y cerrado de tipo

- *Bool* evalúa, en cero o más pasos, a *true*, *false*
- $\sigma \to \tau$ evalúa, en cero o más pasos, a $\lambda x : \sigma . M$, para alguna variable x y término M

Semántica operacional de λ^b

Juicio de evaluación en un paso

$$\frac{M_1 \to M_1'}{M_1\, M_2 \to M_1'\, M_2}\ (\text{E-App1} \ /\ \mu)$$

$$\frac{M_2 \to M_2'}{(\lambda x : \sigma.M)\, M_2 \to (\lambda x : \sigma.M)\, M_2'}\ (\text{E-App2} \ /\ \nu)$$

$$\frac{}{(\lambda x : \sigma.M)\, V \to M\{x \leftarrow V\}}\ (\text{E-AppAbs} \ /\ \beta)$$

Además de (E-IfTrue), (E-IfFalse), (E-If)

Ejemplos

- $(\lambda y : Bool.y)\, true \to true$
- $(\lambda x : Bool \to Bool.x\, true)\, (\lambda y : Bool.y) \to (\lambda y : Bool.y)\, true$
- $(\lambda z : Bool.z)\, ((\lambda y : Bool.y)\, true) \to (\lambda z : Bool.z)\, true$
- No existe M' tal que $x \to M'$
  - x está en forma normal pero no es un valor

Estado de error

- Estado (=término) que no es un valor pero en el que la evaluación está trabada
- Representa estado en el cual el sistema de run-time en una implementación real generaría una excepción

Ejemplos

- *if* x *then* M *else* N
  - Obs: no es cerrado
- *true* M
  - Obs: no es tipable

Objetivo de un sistema de tipos

Garantizar la ausencia de estados de error

- Decimos que un término termina o que es fuertemente normalizante si no hay cadenas de reducción infinitas a partir de él.

Teorema

- Todo término bien tipado termina (díficil)
- Si un término cerrado está bien tipado, entonces evalúa a un valor (consecuencia de lo anterior y de los reultados de correción que siguen)

Corrección

Corrección $=$ Progreso $+$ Preservación

Progreso

Si M es cerrado y bien tipado entonces

1. M es un valor
2. o bien existe M' tal que $M \to M'$

La evaluación no puede trabarse para términos cerrados, bien tipados que no son valores

Preservación

Si $\Gamma \rhd M : \sigma \quad \text{y} \quad M \to N$, entonces $\Gamma \rhd N : \sigma$

La evaluación preserva tipos

Tipos y términos de λ^{bn}

$$\sigma \quad ::= \quad Bool \mid Nat \mid \sigma \to \rho$$

$$M \quad ::= \quad \ldots \mid 0 \mid succ(M) \mid pred(M) \mid iszero(M)$$

Descripción informal:

- $succ(M)$: evaluar M hasta arrojar un número e incrementarlo
- $pred(M)$: evaluar M hasta arrojar un número y decrementar
- $iszero(M)$: evaluar M hasta arrojar un número, luego retornar $true/false$ según sea cero o no

Tipado de λ^{bn}

Agregamos a los axiomas y regla de tipado de λ^{b} los siguientes:

$$\frac{}{\Gamma \rhd 0 : \mathit{Nat}} \text{(T-Zero)}$$

$$\frac{\Gamma \rhd M : \mathit{Nat}}{\Gamma \rhd \mathit{succ}(M) : \mathit{Nat}} \text{(T-Succ)} \qquad \frac{\Gamma \rhd M : \mathit{Nat}}{\Gamma \rhd \mathit{pred}(M) : \mathit{Nat}} \text{(T-Pred)}$$

$$\frac{\Gamma \rhd M : \mathit{Nat}}{\Gamma \rhd \mathit{iszero}(M) : \mathit{Bool}} \text{(T-IsZero)}$$

Valores y evaluación en un paso de λ^{bn} (1/2)

Valores

$$V \quad ::= \quad \ldots \mid \underline{n} \quad \text{donde } \underline{n} \text{ abrevia } succ^{n}(0).$$

Juicio de evaluación en un paso (1/2)

$$\frac{M_1 \to M_1'}{succ(M_1) \to succ(M_1')} \ (\text{E-Succ})$$

$$\frac{}{pred(0) \to 0} \ (\text{E-PredZero}) \qquad \frac{}{pred(\underline{n+1}) \to \underline{n}} \ (\text{E-PredSucc})$$

$$\frac{M_1 \to M_1'}{pred(M_1) \to pred(M_1')} \ (\text{E-Pred})$$

Valores y evaluación en un paso de $\lambda^{bn}(2/2)$

Juicio de evaluación en un paso (2/2)

$$\frac{}{\mathit{iszero}(0) \to \mathit{true}} \; (\text{E-IsZeroZero})$$

$$\frac{}{\mathit{iszero}(\underline{n+1}) \to \mathit{false}} \; (\text{E-IsZeroSucc})$$

$$\frac{M_1 \to M_1'}{\mathit{iszero}(M_1) \to \mathit{iszero}(M_1')} \; (\text{E-IsZero})$$

Además de los juicios de evaluación en un paso de λ^b.

Tipos y términos de λ^{bnu}

$$\sigma \quad ::= \quad Bool \mid Nat \mid Unit \mid \sigma \to \rho$$

$$M \quad ::= \quad \ldots \mid unit$$

Descripción informal:

- *Unit* es un tipo unitario y el único valor posible de una expresión de ese tipo es *unit*.
- Cumple rol similar a *void* en C o Java

Tipado de λ^{bnu}

Agregamos el axioma de tipado:

$$\frac{}{\Gamma \triangleright unit : Unit} \text{ (T-Unit)}$$

NB:

- No hay reglas de evaluación nuevas
- Extendemos el conjunto de valores V con $unit$

$$V \quad ::= \quad \ldots \mid unit$$

Utilidad

- Su utilidad principal es en lenguajes con efectos laterales (próxima clase)
- En estos lenguajes es útil poder evaluar varias expresiones en secuencia

$$M_1; M_2 \stackrel{\text{def}}{=} (\lambda x : Unit.M_2)\, M_1 \qquad x \notin FV(M_2)$$

  - La evaluación de $M_1; M_2$ consiste en primero evaluar M_1 y luego M_2
  - Con la definición dada, este comportamiento se logra con las reglas de evaluación definidas previamente

Tipos y términos de $\lambda^{\ldots let}$

$$M \quad ::= \quad \ldots \mid let\ x : \sigma = M\ in\ N$$

Descripción informal:

- $let\ x : \sigma = M\ in\ N$: evaluar M a un valor V, ligar x a V y evaluar N
- Mejora la legibilidad
- La extensión con *let* no implica agregar nuevos tipos

Ejemplo

- $let\ x : Nat = \underline{2}\ in\ succ(x)$
- $pred\ (let\ x : Nat = \underline{2}\ in\ x)$
- $let\ x : Nat = \underline{2}\ in\ let\ x : Nat = \underline{3}\ in\ x$

Tipado de $\lambda^{\ldots let}$

$$\frac{\Gamma \rhd M : \sigma_1 \quad \Gamma, x : \sigma_1 \rhd N : \sigma_2}{\Gamma \rhd let\ x : \sigma_1 = M\ in\ N : \sigma_2} \text{(T-}\textsc{Let}\text{)}$$

Semántica operacional de $\lambda^{\ldots let}$

$$\frac{M_1 \to M_1'}{let\ x : \sigma = M_1\ in\ M_2 \to let\ x : \sigma = M_1'\ in\ M_2}\ (\text{E-Let})$$

$$\frac{}{let\ x : \sigma = V_1\ in\ M_2 \to M_2\{x \leftarrow V_1\}}\ (\text{E-LetV})$$

Tipos y términos de $\lambda^{\cdots r}$

Sea $\mathcal{L}$ un conjunto de etiquetas

$$\sigma \quad ::= \quad \ldots \mid \{l_i : \sigma_i \; ^{i \in 1..n}\}$$

- $\{nombre : String, edad : Nat\}$
- $\{persona : \{nombre : String, edad : Nat\}, cuil : Nat\}$

$$\{nombre : String, edad : Nat\} \neq \{edad : Nat, nombre : String\}$$

Tipos y términos de $\lambda^{\cdots r}$

$$M \quad ::= \quad \ldots \mid \{l_i = M_i \;^{i \in 1..n}\} \mid M.l$$

Descripción informal:

- El registro $\{l_i = M_i \;^{i \in 1..n}\}$ evalúa a $\{l_i = V_i \;^{i \in 1..n}\}$ donde V_i es el valor al que evalúa M_i, $i \in 1..n$
- $M.l$: evaluar M hasta que arroje $\{l_i = V_i \;^{i \in 1..n}\}$, luego proyectar el campo correspondiente

Ejemplos

- $\lambda x : Nat.\lambda y : Bool.\{edad = x, esMujer = y\}$
- $\lambda p : \{edad : Nat, esMujer : Bool\}.p.edad$
- $(\lambda p : \{edad : Nat, esMujer : Bool\}.p.edad)$
  $\{edad = 20, esMujer = false\}$

Tipado de $\lambda^{\cdots r}$

$$\frac{\Gamma \rhd M_i : \sigma_i \quad \text{para cada } i \in 1..n}{\Gamma \rhd \{l_i = M_i \ ^{i\in 1..n}\} : \{l_i : \sigma_i \ ^{i\in 1..n}\}} \ (\text{T-Rcd})$$

$$\frac{\Gamma \rhd M : \{l_i : \sigma_i \ ^{i\in 1..n}\} \quad j \in 1..n}{\Gamma \rhd M.l_j : \sigma_j} \ (\text{T-Proj})$$

Semántica operacional de $\lambda^{\cdots r}$

Valores

$$V \quad ::= \quad \ldots \mid \{l_i = V_i \ ^{i \in 1..n}\}$$

Semántica operacional de $\lambda^{\cdots r}$

$$\frac{j \in 1..n}{\{l_i = V_i \ {}^{i \in 1..n}\}.l_j \to V_j} \ (\textsc{E-ProjRcd})$$

$$\frac{M \to M'}{M.l \to M'.l} \ (\textsc{E-Proj})$$

$$\frac{M_j \to M_j'}{\begin{array}{c}\{l_i = V_i \ {}^{i \in 1..j-1}, l_j = M_j, l_i = M_i \ {}^{i \in j+1..n}\} \\ \to \\ \{l_i = V_i \ {}^{i \in 1..j-1}, l_j = M_j', l_i = M_i \ {}^{i \in j+1..n}\}\end{array}} \ (\textsc{E-Rcd})$$

Cálculo Lambda Tipado (2/3)

Alejandro Ríos

Departamento de Computación, FCEyN, UBA

"There may, indeed, be other applications of the system other than its use as a logic"

Alonzo Church, 1932

31 de agosto de 2017

La clase pasada

- Cálculo Lambda tipado y extensiones
  - Funciones, aplicación
  - Expresiones booleanas
  - Expresiones aritméticas
  - Unit
  - Declaraciones locales
  - Registros
- Para cada extensión
  - Expresiones de tipos
  - Términos
  - Tipado
  - Valores
  - Semántica operacional small-step

La clase pasada

Corrección = Progreso + Preservación

Progreso

Si M es cerrado y bien tipado entonces

1. M es un valor
2. o bien existe M' tal que $M \to M'$

La evaluación no puede trabarse para términos cerrados, bien tipados que no son valores.

Preservación

Si $\Gamma \rhd M : \sigma$ y $M \to N$, entonces $\Gamma \rhd N : \sigma$

La evaluación preserva tipos.

Terminación

La evaluación de todo término bien tipado termina.

La evaluación de los términos bien tipados no se cuelga.

Hoy - Dos extensiones más

- Referencias

  Programación Imperativa = Progr. Funcional + Efectos

- Recursión
  - Ninguna de las extensiones vistas permite definir funciones recursivas.
  - Todas las funciones definibles hasta el momento son totales.

Referencias - Motivación

- En una expresión como $let\ x : Nat = \underline{2}\ in\ M$
  - x es una variable declarada con valor 2.
  - El valor de x permanece inalterado a lo largo de la evaluación de M.
  - En este sentido x es inmutable: no existe una operación de asignación.
- En programación imperativa pasa todo lo contrario.
  - Todas las variables son mutables.
- Vamos a extender Cálculo Lambda Tipado con variables mutables.

Operaciones básicas

Alocación (Reserva de memoria)

$ref\ M$ genera una referencia fresca cuyo contenido es el valor de M.

Derreferenciación (Lectura)

$!x$ sigue la referencia x y retorna su contenido.

Asignación

$x := M$ almacena en la referencia x el valor de M.

Ejemplos

Nota: En los ejemplos de esta clase omitiremos los tipos de las let-expresiones para facilitar la lectura.

- $let\ x = ref\ \underline{2}\ in\ (\lambda_{\_} : unit.!x)\,(x := succ(!x))$ evalúa a $\underline{3}$.
- ¿$let\ x = ref\ \underline{2}\ in\ x$ a qué evalúa?
- $let\ x = \underline{2}\ in\ x$ evalúa a $\underline{2}$.
- $let\ x = ref\ \underline{2}\ in\ let\ y = x\ in\ (\lambda_{\_} : unit.!x)\,(y := succ(!y))$ evalúa a $\underline{3}$.
  - x e y son alias para la misma celda de memoria.

Comandos = Expresiones con efectos

- El término $let\ x = ref\ \underline{2}\ in\ x := succ(!x)$, ¿A qué evalúa?
- La asignación es una expresión que interesa por su efecto y no su valor.
  - No tiene interés preguntarse por el valor de una asignación.
  - ¡Sí tiene sentido preguntarse por el efecto!

Comando

Expresión que se evalúa para causar un efecto; definimos a *unit* como su valor.

- Un lenguaje funcional puro es uno en el que las expresiones son puras en el sentido de carecer de efectos.

Expresiones de tipos

Las expresiones de tipos se extienden del siguiente modo

$$\sigma \quad ::= \quad \mathit{Bool} \mid \mathit{Nat} \mid \sigma \to \tau \mid \mathit{Unit} \mid \mathit{Ref}\ \sigma$$

Descripción informal:

- $\mathit{Ref}\ \sigma$ es el tipo de las referencias a valores de tipo σ.
- Ej. $\mathit{Ref}\ (\mathit{Bool} \to \mathit{Nat})$ es el tipo de las referencias a funciones de Bool en Nat.

Términos

$$
\begin{array}{lcl}
M & ::= & x \\
 & | & \lambda x : \sigma . M \\
 & | & M\,N \\
 & | & \mathit{unit} \\
 & | & \mathit{ref}\;\; M \\
 & | & !M \\
 & | & M := N \\
 & | & \ldots
\end{array}
$$

El sistema de tipado excluirá términos "mal formados".

- $!\underline{2}$
- $\underline{2} := \underline{3}$

Reglas de tipado

- Las reglas de tipado serán presentadas en dos etapas.
- Primera presentación:
  - es de carácter preliminar;
  - se basa en la sintaxis de términos introducidas al momento.
- Segunda presentación:
  - es la definitiva;
  - al estudiar la semántica operacional surgirá la necesidad de ampliar la sintaxis por cuestiones técnicas;
  - se basa en la sintaxis de términos ampliada.

Reglas de tipado - Preliminares

$$\frac{\Gamma \rhd M_1 : \sigma}{\Gamma \rhd \mathit{ref}\ \ M_1 : \mathit{Ref}\ \ \sigma} \text{(T-Ref)}$$

$$\frac{\Gamma \rhd M_1 : \mathit{Ref}\ \ \sigma}{\Gamma \rhd !M_1 : \sigma} \text{(T-DeRef)}$$

$$\frac{\Gamma \rhd M_1 : \mathit{Ref}\ \ \sigma_1 \quad \Gamma \rhd M_2 : \sigma_1}{\Gamma \rhd M_1 := M_2 : \mathit{Unit}} \text{(T-Assign)}$$

Ejemplos

- $let\ x = ref\ \underline{2}\ in\ (\lambda\_ : unit.!x)\,(x := succ(!x))$
- $let\ x = ref\ \underline{2}\ in\ x$
- $let\ x = \underline{2}\ in\ x$
- $let\ x = ref\ \underline{2}\ in\ let\ y = x\ in\ (\lambda\_ : unit.!x)\,(y := succ(!y))$

Nota: el ítem del primer punto puede escribirse también:

$$let\ x = ref\ \underline{2}\ in\ (x := succ(!x)); !x$$

Recordemos que:

$$M_1; M_2 \stackrel{\text{def}}{=} (\lambda x : Unit.M_2)\, M_1 \qquad x \notin FV(M_2)$$

Motivación

Al intentar formalizar la semántica operacional surgen las preguntas:

- ¿Cuáles son los valores de tipo $Ref\ \sigma$?
- ¿Cómo modelizar la evaluación del término $ref\ M$?

Las respuestas dependen de otra pregunta.

¿Qué es una referencia?

Rta. Es una abstracción de una porción de memoria que se encuentra en uso.

Memoria o "store"

- Usamos direcciones (simbólicas) o "locations" $l, l_i \in \mathcal{L}$ para representar referencias.

  Memoria (o "store"): función parcial de direcciones a valores.

- Usamos letras μ, μ' para referirnos a stores.
- Notación:
  - $\mu[l \mapsto V]$ es el store resultante de pisar $\mu(l)$ con V.
  - $\mu \oplus (l \mapsto V)$ es el store extendido resultante de ampliar μ con una nueva asociación $l \mapsto V$ (asumimos $l \notin Dom(\mu)$).

Los juicios de evaluación toman la forma:

$$M \mid \mu \to M' \mid \mu'$$

Valores

Intuición:

$$\frac{l \notin Dom(\mu)}{ref\ \ V \mid \mu \to l \mid \mu \oplus (l \mapsto V)}\ (\text{E-RefV})$$

Los valores posibles ahora incluyen las direcciones.

$$V \quad ::= \quad unit \mid \lambda x : \sigma.M \mid l$$

Dado que los valores son un subconjunto de los términos,

- debemos ampliar los términos con direcciones;
- éstas son producto de la formalización y no se pretende que sean utilizadas por el programador.

Términos extendidos

$$
\begin{array}{rcl}
M & ::= & x \\
& \mid & \lambda x : \sigma.M \\
& \mid & M\,N \\
& \mid & \mathit{unit} \\
& \mid & \mathit{ref}\ \ M \\
& \mid & !M \\
& \mid & M := N \\
& \mid & l \\
& \mid & \ldots
\end{array}
$$

Juicios de tipado

$$\Gamma \rhd l : ?$$

- Depende de los valores que se almacenen en la dirección l.
- Situación parecida a las variables libres.
- Precisamos un "contexto de tipado" para direcciones:
  - Σ función parcial de direcciones en tipos.

Nuevo juicio de tipado

$$\Gamma | \Sigma \rhd M : \sigma$$

Reglas de tipado - Definitivas

$$\frac{\Gamma|\Sigma \rhd M_1 : \sigma}{\Gamma|\Sigma \rhd \mathit{ref}\ \ M_1 : \mathit{Ref}\ \ \sigma}\ (\text{T-Ref})$$

$$\frac{\Gamma|\Sigma \rhd M_1 : \mathit{Ref}\ \ \sigma}{\Gamma|\Sigma \rhd !M_1 : \sigma}\ (\text{T-DeRef})$$

$$\frac{\Gamma|\Sigma \rhd M_1 : \mathit{Ref}\ \ \sigma_1 \quad \Gamma|\Sigma \rhd M_2 : \sigma_1}{\Gamma|\Sigma \rhd M_1 := M_2 : \mathit{Unit}}\ (\text{T-Assign})$$

$$\frac{\Sigma(l) = \sigma}{\Gamma|\Sigma \rhd l : \mathit{Ref}\ \ \sigma}\ (\text{T-Loc})$$

Juicios de evaluación en un paso

- Retomamos la semántica operacional.
- Vamos a introducir axiomas y reglas que permiten darle significado al juicio de evaluación en un paso.

$$M \mid \mu \to M' \mid \mu'$$

- Recordar el conjunto de valores (expresiones resultantes de evaluar por completo a términos cerrados y bien tipados).

$$V \quad ::= \quad true \mid false \mid 0 \mid \underline{n} \mid unit \mid \lambda x : \sigma.M \mid l$$

Juicios de evaluación en un paso (1/4)

$$\frac{M_1 \mid \mu \to M_1' \mid \mu'}{M_1\, M_2 \mid \mu \to M_1'\, M_2 \mid \mu'} \text{ (E-App1)}$$

$$\frac{M_2 \mid \mu \to M_2' \mid \mu'}{(\lambda x : \sigma.M)\, M_2 \mid \mu \to (\lambda x : \sigma.M)\, M_2' \mid \mu'} \text{ (E-App2)}$$

$$\frac{}{(\lambda x : \sigma.M)\, V \mid \mu \to M\{x \leftarrow V\} \mid \mu} \text{ (E-AppAbs)}$$

Nota: Estas reglas no modifican el store.

Juicios de evaluación en un paso (2/4)

$$\frac{M_1 \mid \mu \to M_1' \mid \mu'}{!M_1 \mid \mu \to !M_1' \mid \mu'} \text{ (E-Deref)}$$

$$\frac{\mu(l) = V}{!l \mid \mu \to V \mid \mu} \text{ (E-DerefLoc)}$$

Juicios de evaluación en un paso (3/4)

$$\frac{M_1 \mid \mu \to M_1' \mid \mu'}{M_1 := M_2 \mid \mu \to M_1' := M_2 \mid \mu'} \text{(E-Assign1)}$$

$$\frac{M_2 \mid \mu \to M_2' \mid \mu'}{V := M_2 \mid \mu \to V := M_2' \mid \mu'} \text{(E-Assign2)}$$

$$\frac{}{l := V \mid \mu \to unit \mid \mu[l \mapsto V]} \text{(E-Assign)}$$

Juicios de evaluación en un paso (4/4)

$$\frac{M_1 \mid \mu \to M_1' \mid \mu'}{\mathit{ref}\;\; M_1 \mid \mu \to \mathit{ref}\;\; M_1' \mid \mu'}\;(\text{E-}\textsc{Ref})$$

$$\frac{l \notin \mathit{Dom}(\mu)}{\mathit{ref}\;\; {\color{red}V} \mid \mu \to l \mid \mu \oplus (l \mapsto {\color{red}V})}\;(\text{E-}\textsc{RefV})$$

Ejemplo

$$
\begin{array}{ll}
& let\ x = ref\ \underline{2}\ in\ (\lambda\_ : Unit.!x)\,(x := succ(!x)) \mid \mu \\
\to & let\ x = l_1\ in\ (\lambda\_ : Unit.!x)\,(x := succ(!x)) \mid \mu \oplus (l_1 \mapsto \underline{2}) \\
\to & (\lambda\_ : Unit.!l_1)\,(l_1 := succ(!l_1)) \mid \mu \oplus (l_1 \mapsto \underline{2}) \\
\to & (\lambda\_ : Unit.!l_1)\,(l_1 := succ(\underline{2})) \mid \mu \oplus (l_1 \mapsto \underline{2}) \\
\to & (\lambda\_ : Unit.!l_1)\,unit \mid (\mu \oplus (l_1 \mapsto \underline{2}))[l_1 \mapsto \underline{3}] \\
\to & !l_1 \mid \mu \oplus (l_1 \mapsto \underline{3}) \\
\to & \underline{3} \mid \mu \oplus (l_1 \mapsto \underline{3})
\end{array}
$$

Ejemplo

Sea

$$\begin{array}{rcl} M & = & \lambda r : Ref(Unit \to Unit). \\ & & \quad let\ f = !r \\ & & \quad\ in\ (r := \lambda x : Unit.f\ x); (!r)\ unit \end{array}$$

$$\begin{array}{ll} & M\,(ref\ (\lambda x : Unit.x)) \mid \mu \\ \to & M\ l_1 \mid \mu \oplus (l_1 \mapsto \lambda x : Unit.x) \\ \to & let\ f = !l_1\ in\ (l_1 := \lambda x : Unit.f\ x); (!l_1)\ unit \mid \ldots \\ \to & let\ f = \lambda x : Unit.x\ in\ (l_1 := \lambda x : Unit.f\ x); (!l_1)\ unit \mid \ldots \\ \to & (l_1 := \lambda x : Unit.(\lambda x : Unit.x)\ x); (!l_1)\ unit \mid \ldots \\ \to & unit; (!l_1)\ unit \mid \mu \oplus (l_1 \mapsto \lambda x : Unit.(\lambda x : Unit.x)\ x) \\ \to & (!l_1)\ unit \mid \mu \oplus (l_1 \mapsto \lambda x : Unit.(\lambda x : Unit.x)\ x) \\ \to & (\lambda x : Unit.(\lambda x : Unit.x)\ x)\ unit \mid \ldots \\ \to & (\lambda x : Unit.x)\ unit \mid \ldots \\ \to & unit \mid \ldots \end{array}$$

Ejemplo

Sea

$$\begin{array}{rcl} M & = & \lambda r : Ref(Unit \to Unit). \\ & & \quad let\ f = !r \\ & & \quad\ in\ (r := \lambda x : Unit.f\ x); (!r)\, unit \end{array}$$

Reemplazamos f por $!r$ y nos queda

$$\begin{array}{rcl} M' & = & \lambda r : Ref\ (Unit \to Unit). \\ & & \quad (r := \lambda x : Unit.(!r)\, x); (!r)\, unit \end{array}$$

Vamos a evaluar este nuevo M' aplicado al mismo término que en el slide anterior y ver qué pasa...

Ejemplo

$$
\begin{array}{rcl}
M' & = & \lambda r : Ref\ (Unit \to Unit). \\
 & & \quad (r := \lambda x : Unit.(!r)\, x); (!r)\, unit
\end{array}
$$

$$
\begin{array}{ll}
 & M'\, (ref\ (\lambda x : Unit.x)) \mid \mu \\
\to & M'\, l_1 \mid \mu \oplus (l_1 \mapsto \lambda x : Unit.x) \\
\to & (l_1 := \lambda x : Unit.(!l_1)\, x); (!l_1)\, unit \mid \ldots \\
\to & unit; (!l_1)\, unit \mid \mu \oplus (l_1 \mapsto \lambda x : Unit.(!l_1)\, x) \\
\to & \boxed{(!l_1)\, unit} \mid \ldots \\
\to & (\lambda x : Unit.(!l_1)\, x)\, unit \mid \ldots \\
\to & \boxed{(!l_1)\, unit} \mid \ldots \\
\to & \ldots
\end{array}
$$

Nota: no todo término cerrado y bien tipado termina en λ^{bnr} (λ-cálculo con booleanos, naturales y referencias).

La clase pasada - Corrección de sistema de tipos

Corrección = Progreso + Preservación

Progreso

Si M es cerrado y bien tipado entonces

1. M es un valor
2. o bien existe M' tal que $M \to M'$

Preservación

Si $\Gamma \triangleright M : \sigma$ y $M \to N$, entonces $\Gamma \triangleright N : \sigma$

Debemos reformular estos resultados en el marco de referencias.

Preservación - Formulación ingenua

La formulación ingenua siguiente es errónea:

$$\Gamma|\Sigma \rhd M : \sigma \quad \text{y} \quad M \,|\, \mu \to M' \,|\, \mu' \quad \text{implica} \quad \Gamma|\Sigma \rhd M' : \sigma$$

- El problema: puede que la semántica no respete los tipos asumidos por el sistema de tipos para las direcciones (i.e. Σ).
- Vamos a ver un ejemplo concreto.

Preservación - Formulación ingenua

$$\Gamma|\Sigma \triangleright M : \sigma \quad \text{y} \quad M \,|\, \mu \to M' \,|\, \mu' \quad \text{implica} \quad \Gamma|\Sigma \triangleright M' : \sigma$$

Supongamos que

- $M = !l$
- $\Gamma = \emptyset$
- $\Sigma(l) = Nat$
- $\mu(l) = true$

Observar que

- $\Gamma|\Sigma \triangleright M : Nat$ y
- $M \,|\, \mu \to true \,|\, \mu$
- pero $\Gamma|\Sigma \triangleright true : Nat$ no vale.

Preservación - Formulación ingenua

$$\Gamma|\Sigma \triangleright M : \sigma \quad \text{y} \quad M \,|\, \mu \to M' \,|\, \mu' \quad \text{implica} \quad \Gamma|\Sigma \triangleright M' : \sigma$$

Supongamos que

- $M = !l$
- $\Gamma = \emptyset$
- $\Sigma(l) = \boxed{Nat}$
- $\mu(l) = \boxed{true}$

Observar que

- $\Gamma|\Sigma \triangleright M : Nat$ y
- $M \,|\, \mu \to true \,|\, \mu$
- pero $\Gamma|\Sigma \triangleright true : Nat$ no vale.

Preservación - Reformulada

- Precisamos una noción de compatibilidad entre el store y el contexto de tipado para stores.
  - Debemos tipar los stores.
- Introducimos un nuevo "juicio de tipado":

$$\Gamma|\Sigma \rhd \mu$$

- Este juicio se define del siguiente modo:

$$\Gamma|\Sigma \rhd \mu \text{ sii}$$

1. $Dom(\Sigma) = Dom(\mu)$ y
2. $\Gamma|\Sigma \rhd \mu(l) : \Sigma(l)$ para todo $l \in Dom(\mu)$.

Preservación - Reformulada

Reformulamos preservación del siguiente modo.

$$\text{Si} \quad \Gamma|\Sigma \rhd M : \sigma \quad \text{y} \quad M \,|\, \mu \to N \,|\, \mu' \quad \text{y} \quad \Gamma|\Sigma \rhd \mu,$$

$$\text{entonces} \quad \Gamma|\boxed{\Sigma} \rhd N : \sigma.$$

- Esto es casi correcto.
- No contempla la posibilidad de que el Σ encuadrado haya crecido en dominio respecto a Σ.
  - Por posibles reservas de memoria.

Preservación - Definitiva

Si

- $\Gamma|\Sigma \triangleright M : \sigma$
- $M \mid \mu \to N \mid \mu'$
- $\Gamma|\Sigma \triangleright \mu$

implica que existe $\Sigma' \supseteq \Sigma$ tal que

- $\Gamma|\Sigma' \triangleright N : \sigma$
- $\Gamma|\Sigma' \triangleright \mu'$

Progreso - Reformulado

Si M es cerrado y bien tipado (i.e. $\emptyset|\Sigma \triangleright M : \sigma$ para algún Σ, σ) entonces:

1. M es un valor
2. o bien para cualquier store μ tal que $\emptyset|\Sigma \triangleright \mu$, existe M' y μ' tal que $M \,|\, \mu \to M' \,|\mu'$.

Recursión

Ecuación recursiva

$$f = \ldots f \ldots f \ldots$$

Dos explicaciones de la función denotada (cuando existe).

- Denotacional
  - Límite de una cadena de aproximaciones
- Operacional
  - El "desdoblador" y puntos fijos

Nota: a desarrollar en el pizarrón.

Términos y tipado

$$M \quad ::= \quad \ldots \mid \mathsf{fix}\ M$$

- No se precisan nuevos tipos pero sí una regla de tipado.

$$\frac{\Gamma \rhd M : \sigma_1 \to \sigma_1}{\Gamma \rhd \mathsf{fix}\ M : \sigma_1}\,(\text{T-}\textsc{Fix})$$

Semántica operacional small-step

No hay valores nuevos pero sí reglas de evaluación en un paso nuevas.

$$\frac{M_1 \to M_1'}{\mathsf{fix}\ \ M_1 \to \mathsf{fix}\ \ M_1'}\ (\text{E-Fix})$$

$$\frac{}{\mathsf{fix}\ (\lambda x : \sigma.M) \to M\{x \leftarrow \mathsf{fix}\ (\lambda x : \sigma.M)\}}\ (\text{E-FixBeta})$$

Ejemplos

Sea M el término

$$\begin{array}{l} \lambda f : Nat \to Nat. \\ \ \lambda x : Nat. \\ \ \ \mathit{if}\ \ \mathrm{iszero}(x)\ \mathit{then}\ \underline{1}\ \mathit{else}\ x * f(\mathrm{pred}(x)) \end{array}$$

en

$$\mathit{let}\ \ \mathit{fact} = \mathrm{fix}\ \ M\ \ \mathit{in}\ \ \mathit{fact}\,\underline{3}$$

Ejemplos

Ahora podemos definir funciones parciales:

$\text{fix}\,(\lambda x : Nat.succ\; x)$

Ejemplos

Sea M el término

$$\begin{array}{l}\lambda s : Nat \to Nat \to Nat. \\ \;\lambda x : Nat. \\ \;\;\lambda y : Nat. \\ \;\;\; \mathit{if}\ \ \mathrm{iszero}(x)\ \mathit{then}\ y\ \mathit{else}\ \ \mathrm{succ}(s\ \mathrm{pred}(x)\, y)\end{array}$$

en

$$\mathit{let}\ \mathit{suma} = \mathrm{fix}\ M\ \mathit{in}\ \mathit{suma}\, \underline{2}\, \underline{3}$$

Letrec

Una construcción alternativa para definir funciones recursivas es

$$letrec\ f : \sigma \to \sigma = \lambda x : \sigma.M\ in\ N$$

Por ejemplo,

$$\begin{array}{l} letrec \\ \ fact : Nat \to Nat = \\ \lambda x : Nat.if\ x = 0\ then\ \underline{1}\ else\ x * fact(pred(x)) \\ \ \ in\ fact\,\underline{3} \end{array}$$

letrec puede escribirse en términos de fix del siguiente modo:

$$let\ f = \mathrm{fix}\,(\lambda f : \sigma \to \sigma.\lambda x : \sigma.M)\ in\ N$$

Fin de la clase

La clase que viene...

...inferencia de tipos

...y algoritmo de unificación

Lambda Cálculo Tipado (3/3)

Eduardo Bonelli / Alejandro Ríos

Departamento de Computación, FCEyN, UBA

"There may, indeed, be other applications of the system other than its use as a logic"

Alonzo Church, 1932

19 de abril de 2012

Estructura de la clase

Inferencia de tipos

- Problema que consiste en transformar términos sin información de tipos o con información de tipos parcial en términos tipables
- Para ello debe inferirse la información de tipos faltante
- Beneficio para lenguajes con tipos
  - el programador puede obviar algunas declaraciones de tipos
  - en general, evita la sobrecarga de tener que declarar y manipular todos los tipos
  - todo ello sin desmejorar la performance del programa: la inferencia de tipos se realiza en tiempo de compilación

Inferencia de tipos

- Inferencia de tipos es especialmente útil en lenguajes polimórficos
- Nosotros vamos a restringir nuestro estudio a inferencia en Lambda Cálculo Tipado (LC)
- Si bien LC no es polimórfico, basta para presentar los conceptos básicos detrás de la inferencia de tipos
- Diremos más sobre polimorfismo à la ML o Haskell al final de la clase
- Algunos nombres importantes en la historia de la inferencia de tipos: Curry, Feys, Hindley, Milner

El problema de la inferencia de tipos

Primero modificamos la sintaxis de los términos de LC eliminando toda anotación de tipos

$$
\begin{array}{rcl}
M & ::= & x \\
& \mid & \mathit{true} \mid \mathit{false} \mid \mathit{if}\;\; M \;\; \mathit{then} \;\; P \;\; \mathit{else} \;\; Q \\
& \mid & 0 \mid \mathit{succ}(M) \mid \mathit{pred}(M) \mid \mathit{iszero}(M) \\
& \mid & \lambda x : \sigma . M \mid M\, N \mid \\
& \mid & \mathit{fix}\, M
\end{array}
$$

Denotamos este conjunto de términos con $\Lambda_{\mathcal{T}}$

El problema de la inferencia de tipos

Primero modificamos la sintaxis de los términos de LC eliminando toda anotación de tipos

$$
\begin{array}{rcl}
M & ::= & x \\
& | & \mathit{true} \mid \mathit{false} \mid \mathit{if}\ \ M\ \ \mathit{then}\ \ P\ \ \mathit{else}\ \ Q \\
& | & \mathit{0} \mid \mathit{succ}(M) \mid \mathit{pred}(M) \mid \mathit{iszero}(M) \\
& | & \lambda x.M \mid M\,N \mid \\
& | & \mathit{fix}\ M
\end{array}
$$

Denotamos este conjunto de términos con Λ

Función de borrado

Llamaremos $\textsc{Erase}(\cdot)$ a la función que dado un término de LC elimina las anotaciones de tipos de las abstracciones

$\textsc{Erase} : \Lambda_{\mathcal{T}} \longrightarrow \Lambda$ se define de la manera esperada.

Ejemplo

$\textsc{Erase}(\lambda x : Nat.\lambda f : Nat \to Nat.f\ x) = \lambda x.\lambda f.f\ x$

El problema de la inferencia - Definición

Dado un término U sin anotaciones de tipo, hallar un término estándar (i.e. con anotaciones de tipos) M tal que

1. $\Gamma \triangleright M : \sigma$, para algún Γ y σ, y
2. $\textsc{Erase}(M) = U$

Ejemplos

- Para $U = \lambda x.x + 5$ tomamos $M = \lambda x : Nat.x + 5$ (observar que no hay otra posibilidad)

El problema de la inferencia - Definición

Dado un término U sin anotaciones de tipo, hallar un término estándar (i.e. con anotaciones de tipos) M tal que

1. $\Gamma \triangleright M : \sigma$, para algún Γ y σ, y
2. $\textsc{Erase}(M) = U$

Ejemplos

- Para $U = \lambda x.x + 5$ tomamos $M = \lambda x : Nat.x + 5$ (observar que no hay otra posibilidad)
- Para $U = \lambda x.\lambda f.f\ x$ tomamos $M_{\sigma,\tau} = \lambda x : \sigma.\lambda f : \sigma \to \tau.f\ x$ (hay un $M_{\sigma,\tau}$ por cada σ, τ)

El problema de la inferencia - Definición

Dado un término U sin anotaciones de tipo, hallar un término estándar (i.e. con anotaciones de tipos) M tal que

1. $\Gamma \triangleright M : \sigma$, para algún Γ y σ, y
2. $\textsc{Erase}(M) = U$

Ejemplos

- Para $U = \lambda x.x + 5$ tomamos $M = \lambda x : Nat.x + 5$ (observar que no hay otra posibilidad)
- Para $U = \lambda x.\lambda f.f\ x$ tomamos $M_{\sigma,\tau} = \lambda x : \sigma.\lambda f : \sigma \to \tau.f\ x$ (hay un $M_{\sigma,\tau}$ por cada σ, τ)
- Para $U = \lambda x.\lambda f.f\ (f\ x)$ tomamos $M_\sigma = \lambda x : \sigma.\lambda f : \sigma \to \sigma.f\ (f\ x)$ (hay un M_σ por cada σ)

El problema de la inferencia - Definición

Dado un término U sin anotaciones de tipo, hallar un término estándar (i.e. con anotaciones de tipos) M tal que

1. $\Gamma \triangleright M : \sigma$, para algún Γ y σ, y
2. $\textsc{Erase}(M) = U$

Ejemplos

- Para $U = \lambda x.x + 5$ tomamos $M = \lambda x : Nat.x + 5$ (observar que no hay otra posibilidad)
- Para $U = \lambda x.\lambda f.f\ x$ tomamos $M_{\sigma,\tau} = \lambda x : \sigma.\lambda f : \sigma \to \tau.f\ x$ (hay un $M_{\sigma,\tau}$ por cada σ, τ)
- Para $U = \lambda x.\lambda f.f\ (f\ x)$ tomamos $M_\sigma = \lambda x : \sigma.\lambda f : \sigma \to \sigma.f\ (f\ x)$ (hay un M_σ por cada σ)
- Para $U = xx$ no existe ningún M con la propiedad deseada

El problema del chequeo de tipos

chequeo de tipos $\neq$ inferencia de tipos

Chequeo de tipos

Dado un término estándar M determinar si existe Γ y σ tales que $\Gamma \rhd M : \sigma$ es derivable.

- Es mucho más fácil que el problema de la inferencia
- Consiste simplemente en seguir la estructura sintáctica de M para reconstruir una derivación del juicio
- Es esencialmente equivalente a determinar, dados Γ y σ, si $\Gamma \rhd M : \sigma$ es derivable.

Variables de tipo

- Dado $\lambda x.\lambda f.f\,(f\,x)$, para cada σ,
  $M_\sigma = \lambda x : \sigma.\lambda f : \sigma \to \sigma.f\,(f\,x)$ es un solución posible
- ¿De qué manera podemos escribir una única expresión que englobe a todas ellas? Usando variables de tipo
  - Todas las soluciones se pueden representar con
    $$\lambda x : s.\lambda f : s \to s.f\,(f\,x)$$
  - "s" es una variable de tipos que representa una expresión de tipos arbitraria
  - Si bien esta expresión no es una solución en sí misma, la sustitución de s por cualquier expresión de tipos sí arroja una solución válida

Variables de tipo

- Extendemos las expresiones de tipo de LC con variables de tipo $s, t, u, \ldots$

$$\sigma \quad ::= \quad s \,|\, Nat \,|\, Bool \,|\, \sigma \to \tau$$

- Denotamos con $\mathcal{V}$ al conjunto de variables de tipo
- Denotamos con $\mathcal{T}$ al conjunto de tipos así definidos

Ejemplos

- $s \to t$
- $Nat \to Nat \to t$
- $Bool \to t$

Sustitución de tipos (o simplemente sustitución)

- Función que mapea variables de tipo en expresiones de tipo.

  Usamos S, T, etc. para sustituciones.

  Formalmente, $S : \mathcal{V} \longrightarrow \mathcal{T}$

  Sólo nos interesan las S tales que $\{t \in \mathcal{V} \mid St \neq t\}$ es finito.

Sustitución de tipos (o simplemente sustitución)

- Función que mapea variables de tipo en expresiones de tipo.

  Usamos S, T, etc. para sustituciones.

  Formalmente, $S : \mathcal{V} \longrightarrow \mathcal{T}$

  Sólo nos interesan las S tales que $\{t \in \mathcal{V} \mid St \neq t\}$ es finito.

- Una sustitución S puede aplicarse (de manera natural) a

Sustitución de tipos (o simplemente sustitución)

- Función que mapea variables de tipo en expresiones de tipo.

  Usamos S, T, etc. para sustituciones.

  Formalmente, $S : \mathcal{V} \longrightarrow \mathcal{T}$

  Sólo nos interesan las S tales que $\{t \in \mathcal{V} \mid St \neq t\}$ es finito.

- Una sustitución S puede aplicarse (de manera natural) a
  1. una expresión de tipos σ (escribimos $S\sigma$)

Sustitución de tipos (o simplemente sustitución)

- Función que mapea variables de tipo en expresiones de tipo.

  Usamos S, T, etc. para sustituciones.

  Formalmente, $S : \mathcal{V} \longrightarrow \mathcal{T}$

  Sólo nos interesan las S tales que $\{t \in \mathcal{V} \mid St \neq t\}$ es finito.

- Una sustitución S puede aplicarse (de manera natural) a
  1. una expresión de tipos σ (escribimos $S\sigma$)
  2. un término M (escribimos SM)

Sustitución de tipos (o simplemente sustitución)

- Función que mapea variables de tipo en expresiones de tipo.

  Usamos S, T, etc. para sustituciones.

  Formalmente, $S : \mathcal{V} \longrightarrow \mathcal{T}$

  Sólo nos interesan las S tales que $\{t \in \mathcal{V} \mid St \neq t\}$ es finito.

- Una sustitución S puede aplicarse (de manera natural) a
  1. una expresión de tipos σ (escribimos $S\sigma$)
  2. un término M (escribimos SM)
  3. un contexto de tipado $\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ (escribimos $S\Gamma$ y lo definimos como sigue)

$$S\Gamma \stackrel{\text{def}}{=} \{x_1 : S\sigma_1, \ldots, x_n : S\sigma_n\}$$

Sustitución - Nociones adicionales

- El conjunto $\{t \mid St \neq t\}$ se llama soporte de S
- El soporte representa las variables que S "afecta"
- Usamos la notación $\{\sigma_1/t_1, \ldots, \sigma_n/t_n\}$ para la sustitución con soporte $\{t_1, \ldots, t_n\}$ definida de la manera obvia
- La sustitución cuyo soporte es $\emptyset$ es la sustitución identidad y la notamos Id

Instancia de un juicio de tipado

Un juicio de tipado $\Gamma' \rhd M' : \sigma'$ es una instancia de $\Gamma \rhd M : \sigma$ si existe una sustitución de tipos S tal que

$$\Gamma' = S\Gamma,\ M' = SM \text{ y } \sigma' = S\sigma$$

Propiedad

Si $\Gamma \rhd M : \sigma$ es derivable, entonces cualquier instancia del mismo también lo es

Función de Inferencia $\mathbb{W}(\cdot)$

Definir una función $\mathbb{W}(\cdot)$ que dado un término U sin anotaciones verifica

Corrección $\mathbb{W}(U) \; = \; \Gamma \triangleright M : \sigma$ implica

- $\textsc{Erase}(M) = U$ y
- $\Gamma \triangleright M : \sigma$ es derivable

Completitud Si $\Gamma \triangleright M : \sigma$ es derivable y $\textsc{Erase}(M) = U$, entonces

- $\mathbb{W}(U)$ tiene éxito y
- produce un juicio $\Gamma' \triangleright M' : \sigma'$ tal que $\Gamma \triangleright M : \sigma$ es instancia del mismo (se dice que $\mathbb{W}(\cdot)$ computa un tipo principal)

Unificación

- El algoritmo de inferencia analiza un término (sin anotaciones de tipo) a partir de sus subtérminos
- Una vez obtenida la información inferida para cada uno de los subtérminos debe
  1. (Consistencia) Determinar si la información de cada subtérmino es consistente
  2. (Síntesis) Sintetizar la información del término original a partir de la información de sus subtérminos

Ejemplo

Consideremos el término $x\,y + x(y+1)$

- Del análisis de $x\,y$ surge que $x :: s \to t$ e $y :: s$
- Del análisis de $x\,(y+1)$ surge que $x :: Nat \to u$ e $y :: Nat$
- Dado que una variable puede tener un sólo tipo debemos compatibilizar la información de tipos
  - El tipo $s \to t$ debe ser compatible o unificable con $Nat \to u$ dado que ambos se refieren a x
  - El tipo s debe ser compatible o unificable con Nat dado que ambos se refieren a y

Unificación

- ¿El tipo $s \to t$ es compatible o unificable con $Nat \to u$? Sí
  - Basta tomar la sustitución $S \stackrel{\text{def}}{=} \{Nat/s, u/t\}$
  - Y observar que $S(s \to t) = Nat \to u = S(Nat \to u)$
- ¿El tipo s es compatible o unificable con Nat? Sí
  - La sustitución antedicha es tal que $Ss = S\,Nat$

El proceso de determinar si existe una sustitución S tal que dos expresiones de tipos σ, τ son unificables (ie. $S\sigma = S\tau$) se llama unificación

- Vamos a estudiar con precisión el proceso de unificación, repasando antes algunos conceptos básicos sobre sustituciones

Composición de sustituciones

La composición de S y T, denotada $S \circ T$, es la sustitución que se comporta como sigue:

$$(S \circ T)(\sigma) = S(T\sigma)$$

Ejemplo

Sea $S = \{u \to Bool/t, Nat/s\}$ y $T = \{v \times Nat/u, Nat/s\}$, entonces $T \circ S = \{(v \times Nat) \to Bool/t, v \times Nat/u, Nat/s\}$

- Decimos que $S = T$ si tienen el mismo soporte y $St = Tt$ para todo t en el soporte de S
- $S \circ Id = Id \circ S = S$
- $S \circ (T \circ U) = (S \circ T) \circ U$

Preorden sobre sustituciones

Una sustitución S es más general que T si existe U tal que $T = U \circ S$.

- La idea es que S es más general que T porque T se obtiene instanciando S

Unificador

Una ecuación de unificación es una expresión de la forma $\sigma_1 \doteq \sigma_2$.
Una sustitución S es una solución de un conjunto de ecuaciones de unificación $\{\sigma_1 \doteq \sigma'_1, \ldots, \sigma_n \doteq \sigma'_n\}$ si $S\sigma_1 = S\sigma'_1, \ldots, S\sigma_n = S\sigma'_n$

Ejemplos

- La sustitución $\{Bool/v, Bool \times Nat/u\}$ es solución de $\{v \times Nat \to Nat \doteq u \to Nat\}$
- $\{Bool \times Bool/v, (Bool \times Bool) \times Nat/u\}$ también!
- $\{v \times Nat/u\}$ también!
- $\{Nat \to s \doteq t \times u\}$ no tiene solución
- $\{u \to Nat \doteq u\}$ no tiene solución

Unificador más general (MGU)

Una sustitución S es un MGU de $\{\sigma_1 \doteq \sigma'_1, \ldots, \sigma_n \doteq \sigma'_n\}$ si

1. es solución de $\{\sigma_1 \doteq \sigma'_1, \ldots, \sigma_n \doteq \sigma'_n\}$
2. es más general que cualquier otra solución de $\{\sigma_1 \doteq \sigma'_1, \ldots, \sigma_n \doteq \sigma'_n\}$

Ejemplos

- La sustitución $\{Bool/v, Bool \times Nat/u\}$ es solución de $\{v \times Nat \to Nat \doteq u \to Nat\}$ pero no es un MGU pues es instancia de la solución $\{v \times Nat/u\}$
- $\{v \times Nat/u\}$ es un MGU del conjunto

Algoritmo de unificación

Teorema

Si $\{\sigma_1 \doteq \sigma'_1, \ldots, \sigma_n \doteq \sigma'_n\}$ tiene solución, existe un MGU y además es único salvo renombre de variables

- Entrada:
  - Conjunto de ecuaciones de unificación $\{\sigma_1 \doteq \sigma'_1, \ldots, \sigma_n \doteq \sigma'_n\}$
- Salida:
  - MGU S de $\{\sigma_1 \doteq \sigma'_1, \ldots, \sigma_n \doteq \sigma'_n\}$, si tiene solución
  - `falla`, en caso contrario

Algoritmo de Martelli-Montanari

- Vamos a presentar un algoritmo no-determinístico
- Consiste en reglas de simplificación que simplifican conjuntos de pares de tipos a unificar (*goals*)

$$G_0 \mapsto G_1 \mapsto \ldots \mapsto G_n$$

- Las secuencias que terminan en el goal vacío son exitosas; aquellas que terminan en `falla` son fallidas
- Algunos pasos de simplificación llevan una sustitución que representa una solución parcial al problema

$$G_0 \mapsto G_1 \mapsto_{S_1} G_2 \mapsto \ldots \mapsto_{S_k} G_n$$

- Si la secuencia es exitosa el MGU es $S_k \circ \ldots \circ S_1$

Reglas del algoritmo de Martelli-Montanari

1. **Descomposición**
   $\{\sigma_1 \to \sigma_2 \doteq \tau_1 \to \tau_2\} \cup G \mapsto \{\sigma_1 \doteq \tau_1, \sigma_2 \doteq \tau_2\} \cup G$
   $\{Nat \doteq Nat\} \cup G \mapsto G$
   $\{Bool \doteq Bool\} \cup G \mapsto G$
2. **Eliminación de par trivial**
   $\{s \doteq s\} \cup G \mapsto G$
3. **Swap**: si σ no es una variable
   $\{\sigma \doteq s\} \cup G \mapsto \{s \doteq \sigma\} \cup G$
4. **Eliminación de variable**: si $s \notin FV(\sigma)$
   $\{s \doteq \sigma\} \cup G \mapsto_{\{\sigma/s\}} \{\sigma/s\}G$
5. **Falla**
   $\{\sigma \doteq \tau\} \cup G \mapsto$ `falla`, con $(\sigma, \tau) \in T \cup T^{-1}$ y
   $T = \{(Bool, Nat), (Nat, \sigma_1 \to \sigma_2), (Bool, \sigma_1 \to \sigma_1)\}$
6. **Occur check**: si $s \neq \sigma$ y $s \in FV(\sigma)$
   $\{s \doteq \sigma\} \cup G \mapsto$ `falla`

Ejemplo de secuencia exitosa

$$
\begin{array}{ll}
 & \{(Nat \to r) \to (r \to u) \doteq t \to (s \to s) \to t\} \\
\mapsto^{1} & \{Nat \to r \doteq t, r \to u \doteq (s \to s) \to t\} \\
\mapsto^{3} & \{t \doteq Nat \to r, r \to u \doteq (s \to s) \to t\} \\
\mapsto^{4}_{Nat \to r/t} & \{r \to u \doteq (s \to s) \to (Nat \to r)\} \\
\mapsto^{1} & \{r \doteq s \to s, u \doteq Nat \to r\} \\
\mapsto^{4}_{s \to s/r} & \{u \doteq Nat \to (s \to s)\} \\
\mapsto^{4}_{Nat \to (s \to s)/u} & \emptyset
\end{array}
$$

- El MGU es
  $\{Nat \to (s \to s)/u\} \circ \{s \to s/r\} \circ \{Nat \to r/t\} =$
  $\{Nat \to (s \to s)/t, s \to s/r, Nat \to (s \to s)/u\}$

Ejemplo de secuencia fallida

$$
\begin{array}{ll}
 & \{r \to (s \to r) \doteq s \to ((r \to Nat) \to r)\} \\
\mapsto^1 & \{r \doteq s, s \to r \doteq (r \to Nat) \to r\} \\
\mapsto^4_{s/r} & \{s \to s \doteq (s \to Nat) \to s\} \\
\mapsto^1 & \{s \doteq s \to Nat, s \doteq s\} \\
\mapsto^6 & \texttt{falla}
\end{array}
$$

Propiedades del algoritmo

Teorema

- El algoritmo de Martelli-Montanari siempre termina
- Sea G un conjunto de pares
  - si G tiene un unificador, el algoritmo termina exitosamente y retorna un MGU
  - si G no tiene unificador, el algoritmo termina con `falla`

Algoritmo de inferencia

- Vamos a presentar un algoritmo de inferencia para LC
- El objetivo es definir $\mathbb{W}(U)$ por recursión sobre la estructura de U
- Primero presentamos la cláusulas que definen $\mathbb{W}(.)$ sobre las constantes y las variables, luego pasamos a las demás construcciones
- Utilizaremos el algoritmo de unificación

Algoritmo de inferencia (caso constantes y variables)

$$
\begin{array}{rcl}
\mathbb{W}(\mathit{0}) & \stackrel{\text{def}}{=} & \emptyset \triangleright \mathit{0} : \mathit{Nat} \\
\mathbb{W}(\mathit{true}) & \stackrel{\text{def}}{=} & \emptyset \triangleright \mathit{true} : \mathit{Bool} \\
\mathbb{W}(\mathit{false}) & \stackrel{\text{def}}{=} & \emptyset \triangleright \mathit{false} : \mathit{Bool} \\
\mathbb{W}(x) & \stackrel{\text{def}}{=} & \{x : s\} \triangleright x : s, \quad s \text{ variable fresca}
\end{array}
$$

Algoritmo de inferencia (caso *succ*)

- Sea $\mathbb{W}(U) \; = \; \Gamma \rhd M : \tau$
- Sea $S = MGU\{\tau \doteq Nat\}$
- Entonces

$$\mathbb{W}(succ(U)) \quad \stackrel{\mathrm{def}}{=} \quad S\Gamma \rhd S\, succ(M) : Nat$$

- Nota: Caso *pred* es similar

Algoritmo de inferencia (caso *iszero*)

- Sea $\mathbb{W}(U) = \Gamma \triangleright M : \tau$
- Sea $S = MGU\{\tau \doteq Nat\}$
- Entonces

$$\mathbb{W}(iszero(U)) \stackrel{\text{def}}{=} S\Gamma \triangleright S\; iszero(M) : Bool$$

Algoritmo de inferencia (caso ifThenElse)

- Sea
  - $\mathbb{W}(U) \;=\; \Gamma_1 \rhd M : \rho$
  - $\mathbb{W}(V) \;=\; \Gamma_2 \rhd P : \sigma$
  - $\mathbb{W}(W) \;=\; \Gamma_3 \rhd Q : \tau$
- Sea

$$S = MGU(\{\sigma_1 \doteq \sigma_2 \mid x : \sigma_1 \in \Gamma_i \wedge x : \sigma_2 \in \Gamma_j, i \neq j\} \cup \{\sigma \doteq \tau,\ \rho \doteq Bool\})$$

- Entonces

$$\mathbb{W}(if\ \ U\ \ then\ \ V\ \ else\ \ W) \stackrel{\text{def}}{=} S\Gamma_1 \cup S\Gamma_2 \cup S\Gamma_3 \rhd S(if\ \ M\ \ then\ \ P\ \ else\ \ Q) : S\sigma$$

Algoritmo de inferencia (caso aplicación)

- Sea
  - $\mathbb{W}(U) \quad = \quad \Gamma_1 \rhd M : \tau$
  - $\mathbb{W}(V) \quad = \quad \Gamma_2 \rhd N : \rho$
- Sea

$$S = MGU(\{\sigma_1 \doteq \sigma_2 \mid x : \sigma_1 \in \Gamma_1 \wedge x : \sigma_2 \in \Gamma_2\} \\ \cup \\ \{\tau \doteq \rho \to t\}) \quad \text{con } t \text{ una variable fresca}$$

- Entonces

$$\mathbb{W}(U\ V) \quad \stackrel{\text{def}}{=} \quad S\Gamma_1 \cup S\Gamma_2 \rhd S(M\ N) : St$$

Algoritmo de inferencia (caso abstracción)

- Sea $\mathbb{W}(U) \quad = \quad \Gamma \triangleright M : \rho$
- Si el contexto tiene información de tipos para x (i.e. $x : \tau \in \Gamma$ para algún τ), entonces

$$\mathbb{W}(\lambda x.U) \quad \stackrel{\text{def}}{=} \quad \Gamma \setminus \{x : \tau\} \triangleright \lambda x : \tau.M : \tau \to \rho$$

- Si el contexto no tiene información de tipos para x (i.e. $x \notin \mathrm{Dom}(\Gamma)$) elegimos una variable fresca s y entonces

$$\mathbb{W}(\lambda x.U) \quad \stackrel{\text{def}}{=} \quad \Gamma \triangleright \lambda x : s.M : s \to \rho$$

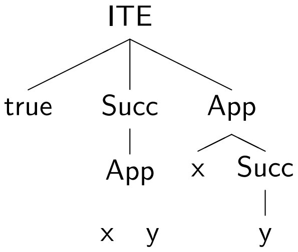
Algoritmo de inferencia (caso *fix*)

- Sea $\mathbb{W}(U) \; = \; \Gamma \rhd M : \tau$
- Sea $S = MGU\{\tau \doteq t \to t\}$, t variable fresca

$$\mathbb{W}(\mathit{fix}(U)) \quad \stackrel{\mathrm{def}}{=} \quad S\Gamma \rhd S\,\mathit{fix}(M) : St$$

Ejemplo

- Vamos a mostrar cómo inferir el tipo de $\textit{if true then succ}(x\,y)\ \textit{else}\ x\,(\textit{succ}(y))$
- Aplicaremos el algoritmo, paso por paso



Ejemplo (1/4)

$$\textit{if } \textit{true} \textit{ then } \textit{succ}(x\,y) \textit{ else } x\,(\textit{succ}(y))$$

$$\mathbb{W}(\textit{true}) \quad = \quad \emptyset \triangleright \textit{true} : \textit{Bool}$$

Ejemplo (2/4)

$$\textit{if true then } succ(x\,y) \textit{ else } x\,(succ(y))$$

$$\begin{array}{rcl}
\mathbb{W}(x) & = & \{x : s\} \triangleright x : s \\
\mathbb{W}(y) & = & \{y : t\} \triangleright y : t \\
\mathbb{W}(x\,y) & = & \{x : t \to r, y : t\} \triangleright x\,y : r \\
 & & \text{donde } S = MGU(\{s \doteq t \to r\}) = \{t \to r/s\} \\
\mathbb{W}(succ(x\,y)) & = & \{x : t \to Nat, y : t\} \triangleright succ(x\,y) : Nat \\
 & & \text{donde } S = MGU(\{r \doteq Nat\}) = \{Nat/r\}
\end{array}$$

Ejemplo (3/4)

$$\mathit{if}\ \ \mathit{true}\ \ \mathit{then}\ \mathit{succ}(x\,y)\ \mathit{else}\ x\,(\mathit{succ}(y))$$

$$\begin{array}{rcl}
\mathbb{W}(y) & = & \{y : v\} \rhd y : v \\
\mathbb{W}(\mathit{succ}(y)) & = & \{y : \mathit{Nat}\} \rhd \mathit{succ}(y) : \mathit{Nat} \\
 & & \text{donde } S = \mathit{MGU}(\{v \doteq \mathit{Nat}\}) = \{\mathit{Nat}/v\} \\
\mathbb{W}(x) & = & \{x : u\} \rhd x : u \\
\mathbb{W}(x\,\mathit{succ}(y)) & = & \{x : \mathit{Nat} \to w, y : \mathit{Nat}\} \rhd x\,\mathit{succ}(y) : w \\
 & & \text{donde } \mathit{MGU}(\{u \doteq \mathit{Nat} \to w\}) = \{\mathit{Nat} \to w/u\}
\end{array}$$

Ejemplo (4/4)

$$M = \mathit{if}\ \ \mathit{true}\ \ \mathit{then}\ \mathit{succ}(x\,y)\ \mathit{else}\ x\,(\mathit{succ}(y))$$

- $\mathbb{W}(\mathit{true}) = \emptyset \triangleright \mathit{true} : \mathit{Bool}$
- $\mathbb{W}(\mathit{succ}(x\,y)) = \{x : t \to \mathit{Nat}, y : t\} \triangleright \mathit{succ}(x\,y) : \mathit{Nat}$
- $\mathbb{W}(x\,\mathit{succ}(y)) = \{x : \mathit{Nat} \to w, y : \mathit{Nat}\} \triangleright x\,\mathit{succ}(y) : w$

$$\mathbb{W}(M) \quad = \quad \{x : \mathit{Nat} \to \mathit{Nat}, y : \mathit{Nat}\} \triangleright M : \mathit{Nat}$$

donde $S = \mathit{MGU}(\{t \to \mathit{Nat} \doteq \mathit{Nat} \to w,\ t \doteq \mathit{Nat}, \mathit{Nat} \doteq w\}) =$
$= \{\mathit{Nat}/t, \mathit{Nat}/w\}$

Un ejemplo de falla

$$M = \mathit{if}\ \ \mathit{true}\ \ \mathit{then}\ x\,\underline{2}\ \mathit{else}\ x\ \mathit{true}$$

$$\begin{array}{rcl}
\mathbb{W}(x) & = & \{x : s\} \triangleright x : s \\
\mathbb{W}(\underline{2}) & = & \emptyset \triangleright \underline{2} : \mathit{Nat} \\
\mathbb{W}(x\,\underline{2}) & = & \{x : \mathit{Nat} \to t\} \triangleright x\,\underline{2} : t \\
\\
\mathbb{W}(x) & = & \{x : u\} \triangleright x : u \\
\mathbb{W}(\mathit{true}) & = & \emptyset \triangleright \mathit{true} : \mathit{Bool} \\
\mathbb{W}(x\,\mathit{true}) & = & \{x : \mathit{Bool} \to v\} \triangleright x\,\mathit{true} : v \\
\\
\mathbb{W}(M) & = & \mathit{falla}
\end{array}$$

no existe el $\mathit{MGU}(\{\mathit{Nat} \to t \doteq \mathit{Bool} \to v\})$

Complejidad

- Tanto la unificación como la inferencia para LC se puede hacer en tiempo lineal
- El tipo principal asociado a un término sin anotaciones puede ser exponencial en el tamaño del término

Considerar inferir el tipo de $P^n\,M$ con $P : s \to s \times s$ y $M : \sigma$

- ¿Esto no contradice lo antedicho?
- No. Se pueden representar usando dags en cuyo caso el tamaño del tipo principal de U será $O(n)$
- NB: En la presencia de polimorfismo la inferencia es exponencial

Let-Polymorphism

- Los lenguajes funcionales como ML, Haskell, etc. permiten tipos polimórficos de la forma

$$\forall s_1 \dots s_n.\sigma \text{ } (\sigma \text{ sin cuantificadores})$$

- Este tipo de polimorfismo restringido se llama predicativo
- En particular no se pueden definir funciones que tomen a otras funciones polimórficas como argumento

Let-Polymorphism

```
Prelude> (\f-> (f True, f 3)) (\x -> 5)
ERROR - Illegal Haskell 98 class constraint in inferred type
*** Expression : (\f -> (f True,f 3)) (\x -> 5)
*** Type       : Num Bool => (Integer,Integer)

Prelude> (\f-> (f True, f 3)) id
ERROR - Illegal Haskell 98 class constraint in inferred type
*** Expression : (\f -> (f True,f 3)) id
*** Type       : Num Bool => (Bool,Bool)
```

Let-Polymorphism

- Para poder declarar y usar funciones polimórficas se introduce la construcción let

```
Prelude> let g = \x->5 in (g True, g 3)
(5,5)
```

- Polimorfismo predicativo con declaraciones let polimórficas forman el núcleo (básico) del sistema de tipos de ML y Haskell
- La inferencia de tipos para este sistema es muy similar a aquella vista hoy
- Para más detalles consultar capítulo 11 del texto de Mitchell o capítulo 22 del texto de Pierce

Fundamentos de Programación Lógica

Paradigmas de Lenguajes de Programación

Departamento de Computación, FCEyN, UBA

28 de septiembre de 2017

Paradigma lógico

- Se basa en el uso de la lógica como un lenguaje de programación
- Se especifican
  - ciertos hechos y reglas de inferencia
  - un objetivo ("goal") a probar
- Un motor de inferencia trata de probar que el objetivo es consecuencia de los hechos y reglas
- Es declarativo: se especifican hechos, reglas y objetivo sin indicar cómo se obtiene éste último a partir de los primeros

Prolog

- Lenguaje de programación basado en este esquema que fue introducido a fines de 1971 (cf. "The Birth of Prolog", A. Colmerauer y P. Roussel, `www.lif-sud.univ-mrs.fr/~colmer/`)
- Los programas se escriben en un subconjunto de la lógica de primer orden
- El mecanismo teórico en el que se basa es el método de resolución

Prolog

Ejemplo de programa

```
habla(ale,ruso).
habla(juan,ingles).
habla(maria,ruso).
habla(maria,ingles).

seComunicaCon(X,Y):-habla(X,L),habla(Y,L),X\=Y
```

Ejemplo de goal

```
seComunicaCon(X,ale)
```

Nuestro enfoque

1. Lógica proposicional
2. Método de resolución para lógica proposicional
3. Repaso de lógica de primer orden
4. Método de resolución para lógica de primer orden
5. Cláusulas de Horn y resolución SLD, programación lógica

Sintaxis de la lógica proposicional

Dado un conjunto $\mathcal{V}$ de variables proposicionales, podemos definir inductivamente al conjunto de fórmulas proposicionales (o proposiciones) **Prop** de la siguiente manera:

1. Una variable proposicional $P_0, P_1, \ldots$ es una proposición
2. Si A, B son proposiciones, entonces:
   - $\neg A$ es una proposición
   - $A \wedge B$ es una proposición
   - $A \vee B$ es una proposición
   - $A \supset B$ es una proposición
   - $A \iff B$ es una proposición

Ejemplos: $A \vee \neg B$, $(A \wedge B) \supset (A \vee A)$

Semántica

- Una valuación es una función $v : \mathcal{V} \to \{\mathbf{T}, \mathbf{F}\}$ que asigna valores de verdad a las variables proposicionales
- Una valuación satisface una proposición A si $v \models A$ donde:

$$
\begin{array}{rcl}
v \models P & \textit{sii} & v(P) = \mathbf{T} \\
v \models \neg A & \textit{sii} & v \not\models A \ (i.e. \text{ no } v \models A) \\
v \models A \vee B & \textit{sii} & v \models A \text{ o } v \models B \\
v \models A \wedge B & \textit{sii} & v \models A \text{ y } v \models B \\
v \models A \supset B & \textit{sii} & v \not\models A \text{ o } v \models B \\
v \models A \iff B & \textit{sii} & (v \models A \text{ sii } v \models B)
\end{array}
$$

Tautologías y satisfactibilidad

Una proposición A es

- una tautología si $v \models A$ para toda valuación v
- satisfactible si existe una valuación v tal que $v \models A$
- insatisfactible si no es satisfactible

Un conjunto de proposiciones S es

- satisfactible si existe una valuación v tal que para todo $A \in S$, se tiene $v \models A$
- insatisfactible si no es satisfactible

Ejemplos

Tautologías

- $A \supset A$
- $\neg\neg A \supset A$
- $(A \supset B) \iff (\neg B \supset \neg A)$

Proposiciones insatisfactibles

- $(\neg A \vee B) \wedge (\neg A \vee \neg B) \wedge A$
- $(A \supset B) \wedge A \wedge \neg B$

Tautologías e insatisfactibilidad

Teorema

Una proposición A es una tautología sii $\neg A$ es insatisfactible

Dem.

$\Rightarrow$. Si A es tautología, para toda valuación v, $v \models A$. Entonces, $v \not\models \neg A$ (i.e. v no satisface $\neg A$).

$\Leftarrow$. Si $\neg A$ es insatisfactible, para toda valuación v, $v \not\models \neg A$. Luego $v \models A$.

Notar

Este resultado sugiere un método indirecto para probar que una proposición A es una tautología, a saber probar que $\neg A$ es insatisfactible

Forma normal conjuntiva (FNC)

- Un Literal es una variable proposicional P o su negación $\neg P$
- Una proposición A está en FNC si es una conjunción

$$C_1 \wedge \ldots \wedge C_n$$

  donde cada C_i (llamado cláusula) es una disyunción

$$B_{i1} \vee \ldots \vee B_{in_i}$$

  y cada B_{ij} es un literal
- Una FNC es una "conjunción de disyunciones de literales"

Forma normal conjuntiva

Ejemplos

- $(P \vee Q) \wedge (P \vee \neg Q)$ está en FNC
- $(P \vee Q) \wedge (P \vee \neg\neg Q)$ no está en FNC
- $(P \wedge Q) \vee P$ no está en FNC

Teorema

Para toda proposición A puede hallarse una proposición A' en FNC que es lógicamente equivalente a A.

Nota

A es lógicamente equivalente a B sii $A \iff B$ es una tautología

Notación conjuntista para FNC

- Dado que tanto $\vee$ como $\wedge$

  1. son conmutativos (i.e. $(A \vee B) \iff (B \vee A)$)
  2. son asociativos (i.e. $((A \vee B) \vee C) \iff (A \vee (B \vee C))$)
  3. son idempotentes (i.e. $(A \vee A) \iff A$)

  Podemos asumir que

  1. Cada cláusula C_i es distinta
  2. Cada cláusula puede verse como un conjunto de literales distintos

Notación conjuntista para FNC

Consecuentemente para una FNC podemos usar la notación

$$\{C_1, \ldots, C_n\}$$

donde cada C_i es un conjunto de literales

$$\{B_{i1}, \ldots, B_{in_i}\}$$

Por ejemplo, la FNC $(P \vee Q) \wedge (P \vee \neg Q)$ se anota

$$\{\{P, Q\}, \{P, \neg Q\}\}$$

Validez por refutación

Principio de demostración por refutación:

Probar que A es válido mostrando que $\neg A$ es insatisfactible

- Hay varias técnicas de demostración por refutación
  - Tableaux semántico (1960)
  - Procedimiento de Davis-Putnam (1960)
  - Resolución (1965)
- Nos vamos a concentrar en Resolución

Resolución

- Introducido por Alan Robinson en 1965

  A machine-oriented logic based on the resolution principle, J. of the ACM (12).
- Es simple de implementar
- Popular en el ámbito de demostración automática de teoremas
- Tiene una única regla de inferencia: la regla de resolución
- Si bien no es imprescindible, es conveniente asumir que las proposiciones están en forma normal conjuntiva

Principio fundamental del método de resolución

- Se basa en el hecho de que la siguiente proposición es una tautología
$$(A \vee P) \wedge (B \vee \neg P) \iff (A \vee P) \wedge (B \vee \neg P) \wedge (A \vee B)$$
- En efecto, el conjunto de cláusulas
$$\{C_1, \ldots, C_m, \{A, P\}, \{B, \neg P\}\}$$
es lógicamente equivalente a
$$\{C_1, \ldots, C_m, \{A, P\}, \{B, \neg P\}, \{A, B\}\}$$

Resolución

- En consecuencia, el conjunto de cláusulas

$$\{C_1, \ldots, C_m, \{A, P\}, \{B, \neg P\}\}$$

es insatisfactible sii

$$\{C_1, \ldots, C_m, \{A, P\}, \{B, \neg P\}, \{A, B\}\}$$

es insatisfactible

- La cláusula $\{A, B\}$ se llama resolvente de las cláusulas $\{A, P\}$ y $\{B, \neg P\}$
- El resolvente de las cláusulas $\{P\}$ y $\{\neg P\}$ es la cláusula vacía y se anota $\square$

Regla de resolución

- Dado un literal L, el opuesto de L (escrito $\overline{L}$) se define como:
  - $\neg P$ si $L = P$
  - P si $L = \neg P$
- Dadas dos cláusulas C_1, C_2, una cláusula C se dice resolvente de C_1 y C_2 sii, para algún literal L, $L \in C_1$, $\overline{L} \in C_2$, y

$$C = (C_1 - \{L\}) \cup (C_2 - \{\overline{L}\})$$

Ejemplos

Las cláusulas $\{A, B\}$ y $\{\neg A, \neg B\}$ tienen dos resolventes: $\{A, \neg A\}$ y $\{B, \neg B\}$.
Las cláusulas $\{P\}$ y $\{\neg P\}$ tienen a la cláusula vacía como resolvente

Regla de resolución

$$\frac{\{A_1, \ldots, A_m, Q\} \quad \{B_1, \ldots, B_n, \neg Q\}}{\{A_1, \ldots, A_m, B_1, \ldots, B_n\}}$$

El método de resolución

El proceso de agregar a un conjunto S la resolvente C de dos cláusulas C_1, C_2 que pertenecen a S (i.e. de aplicar la regla de resolución a S) se llama un paso de resolución.

Nota:

- Asumiremos que el resolvente C que se agrega a S no pertenecía ya a S
- Pasos de resolución preservan insatisfactibilidad

$$S \text{ es insatisfactible sii } S \cup \{C\} \text{ es insatisfactible}$$

El método de resolución

- Un conjunto de cláusulas se llama una refutación si contiene a la cláusula vacía (i.e. a $\square$).
- El método de resolución trata de construir una secuencia de conjuntos de cláusulas, obtenidas usando pasos de resolución hasta llegar a una refutación.

$$S_1 \Rightarrow S_2 \Rightarrow \ldots \Rightarrow S_{n-1} \Rightarrow S_n \ni \square$$

- En ese caso se sabe que el conjunto inicial de cláusulas es insatisfactible dado que
  1. cada paso de resolución preserva insatisfactibilidad
  2. el último conjunto de cláusulas es insatisfactible (contiene la cláusula vacía)

Ejemplo

Objetivo: mostrar que el conjunto de cláusulas $\{\{P, Q\}, \{P, \neg Q\}, \{\neg P, Q\}, \{\neg P, \neg Q\}\}$ es insatisfactible.

1. $\{\{P, Q\}, \{P, \neg Q\}, \{\neg P, Q\}, \{\neg P, \neg Q\}\}$
2. $\{\{P, Q\}, \{P, \neg Q\}, \{\neg P, Q\}, \{\neg P, \neg Q\}, \{P\}\}$
3. $\{\{P, Q\}, \{P, \neg Q\}, \{\neg P, Q\}, \{\neg P, \neg Q\}, \{P\}, \{Q\}\}$
4. $\{\{P, Q\}, \{P, \neg Q\}, \{\neg P, Q\}, \{\neg P, \neg Q\}, \{P\}, \{Q\}, \{\neg P\}\}$
5. $\{\{P, Q\}, \{P, \neg Q\}, \{\neg P, Q\}, \{\neg P, \neg Q\}, \{P\}, \{Q\}, \{\neg P\}, \square\}$

Ejemplo

Objetivo: mostrar que el conjunto de cláusulas $\{\{A, B, \neg C\}, \{A, B, C\}, \{A, \neg B\}, \{\neg A\}\}$ es insatisfactible.

1. $\{\{A, B, \neg C\}, \{A, B, C\}, \{A, \neg B\}, \{\neg A\}\}$
2. $\{\{A, B, \neg C\}, \{A, B, C\}, \{A, \neg B\}, \{\neg A\}, \{A, B\}\}$
3. $\{\{A, B, \neg C\}, \{A, B, C\}, \{A, \neg B\}, \{\neg A\}, \{A, B\}, \{A\}\}$
4. $\{\{A, B, \neg C\}, \{A, B, C\}, \{A, \neg B\}, \{\neg A\}, \{A, B\}, \{A\}, \square\}$

Ejemplo

Objetivo: mostrar que el conjunto de cláusulas
$S = \{\{A, B, C\}, \{A\}, \{B\}\}$ es insatisfactible.

- No podemos aplicar ningún paso de resolución a S
- Por lo tanto, no puede llegarse a una refutación a partir S
- S debe ser satisfactible
- En efecto, tomar por ejemplo $v(A) = v(B) = \mathbf{T}$

Ejemplo

Objetivo: probar que R es consecuencia de $P \vee Q$, $P \supset R$, $Q \supset R$.

- Queremos probar que $((P \vee Q) \wedge (P \supset R) \wedge (Q \supset R)) \supset R$ es tautología.
- Esto es equivalente a probar que $\neg[((P \vee Q) \wedge (P \supset R) \wedge (Q \supset R)) \supset R]$ es insatisfactible.
- Para ello usamos resolución sobre su FNC.

Terminación de la regla de resolución

- La aplicación reiterada de la regla de resolución siempre termina (suponiendo que el resolvente que se agrega es nuevo)
- En efecto, notar que
  1. El resolvente (i.e. la cláusula nueva que se agrega) se forma con los literales distintos que aparecen en el conjunto de cláusulas de partida S
  2. Hay una cantidad finita de literales en el conjunto de cláusulas de partida S
- En el peor de los casos, la regla de resolución podrá generar una nueva cláusula por cada combinación diferente de literales distintos de S

Corrección y completitud

- El siguiente resultado establece la corrección y completitud del método de resolución

Teorema

Dado un conjunto finito S de cláusulas,

$$S \text{ es insatisfactible sii tiene una refutación}$$

Recapitulando

Para probar que A es una tautología hacemos lo siguiente:

1. Calculamos la forma normal conjuntiva de $\neg A$
2. Aplicamos el método de resolución
3. Si hallamos una refutación:
   - $\neg A$ es insatisfactible,
   - Y, por lo tanto, A es una tautología
4. Si no hallamos ninguna refutación:
   - $\neg A$ es satisfactible,
   - Y, por lo tanto, A no es una tautología

Lenguaje de primer orden

Un lenguaje de primer orden (LPO) $\mathcal{L}$ consiste en:

1. Un conjunto numerable de constantes $c_0, c_1, \ldots$
2. Un conjunto numerable de símbolos de función con aridad $n > 0$ (indica el número de argumentos) $f_0, f_1, \ldots$
3. Un conjunto numerable de símbolos de predicado con aridad $n \geq 0$, $P_0, P_1, \ldots$. La aridad indica el número de argumentos que toma (si $n = 0$, es una variable proposicional)

Ejemplo: Lenguaje de primer orden para la aritmética

Constantes: 0; Símbolos de función: $S, +, *$; Símbolos de predicado: $<$, $=$.

Términos de primer orden

Sea $\mathcal{V} = \{x_0, x_1, \ldots\}$ un conjunto numerable de variables y $\mathcal{L}$ un LPO. El conjunto de $\mathcal{L}$-términos se define inductivamente como:

1. Toda constante de $\mathcal{L}$ y toda variable es un $\mathcal{L}$-término
2. Si $t_1, \ldots, t_n \in \mathcal{L}$-términos y f es un símbolo de función de aridad n, entonces $f(t_1, \ldots, t_n) \in \mathcal{L}$-términos

Ejemplo: Aritmética (cont.)

$S(0), +(S(0), S(S(0))), *(S(x_1), +(x_2, S(x_3)))$

Fórmulas atómicas

Sea $\mathcal{V}$ un conjunto numerable de variables y $\mathcal{L}$ un LPO. El conjunto de $\mathcal{L}$-fórmulas atómicas se define inductivamente como:

1. Todo símbolo de predicado de aridad 0 es una $\mathcal{L}$-fórmula atómica
2. Si $t_1, \ldots, t_n \in \mathcal{L}$-términos y P es un símbolo de predicado de aridad n, entonces $P(t_1, \ldots, t_n)$ es una $\mathcal{L}$-fórmula atómica

Ejemplo: Aritmética (cont.)

$< (0, S(0)), < (x_1, +(S(0), x_2))$

Fórmulas de primer orden

Sea $\mathcal{V}$ un conjunto numerable de variables y $\mathcal{L}$ un LPO. El conjunto de $\mathcal{L}$-fórmulas se define inductivamente como:

1. Toda $\mathcal{L}$-fórmula atómica es una $\mathcal{L}$-fórmula
2. Si $A, B \in \mathcal{L}$-fórmulas, entonces $(A \wedge B), (A \vee B), (A \supset B), (A \iff B)$ y $\neg A$ son $\mathcal{L}$-fórmulas
3. Para toda variable x_i y cualquier $\mathcal{L}$-fórmula A, $\forall x_i.A$ y $\exists x_i.A$ son $\mathcal{L}$-fórmulas

Ejemplo: Aritmética (cont.)

- $\forall x.\forall y.(x < y \supset \exists z.y = x + z)$
- $\forall x.\forall y.((x < y \vee y < x) \vee x = y)$

Variables libres y ligadas

Las variables pueden ocurrir libres o ligadas.

- Los cuantificadores ligan variables
- Usamos $FV(A)$ y $BV(A)$ para referirnos a las variables libres y ligadas, resp., de A
- $FV(A)$ y $BV(A)$ se pueden definir por inducción estructural en A

Ejemplo

Si $A = \forall x.(R(x, y) \supset P(x))$, entonces $FV(A) = \{y\}$ y $BV(A) = \{x\}$

Variables libres y ligadas

- Una fórmula A se dice rectificada si
  - $FV(A)$ y $BV(A)$ son disjuntos y
  - Cuantificadores distintos de A ligan variables distintas
- Toda fórmula se puede rectificar (renombrando variables ligadas) a una fórmula lógicamente equivalente
- Una sentencia es una fórmula cerrada (i.e. sin variables libres).

Estructura de primer orden

Dado un lenguaje de primer orden $\mathcal{L}$, una estructura para $\mathcal{L}$, $\mathbf{M}$, es un par $\mathbf{M} = (M, I)$ donde

- M (dominio) es un conjunto no vacío
- I (función de interpretación) asigna funciones y predicados sobre M a símbolos de $\mathcal{L}$ de la siguiente manera:
  1. Para toda constante c, $I(c) \in M$
  2. Para todo f de aridad $n > 0$, $I(f) : M^n \to M$
  3. Para todo predicado P de aridad $n \geq 0$, $I(P) : M^n \to \{\mathbf{T}, \mathbf{F}\}$

Satisfactibilidad

Asignación

Sea **M** una estructura para $\mathcal{L}$. Una asignación es una función $s : \mathcal{V} \to M$

- Si s es una asignación y $a \in M$, usamos la notación $s[x \leftarrow a]$ para denotar la asignación que se comporta igual que s salvo en el elemento x, en cuyo caso retorna a

Satisfactibilidad

La relación $s \models_{\mathbf{M}} A$ establece que la asignación s satisface la fórmula A en la estructura **M**

- Vamos a definir la relación $s \models_{\mathbf{M}} A$ usando inducción estructural en A

Satisfactibilidad

La relación $s \models_{\mathbf{M}} A$ se define inductivamente como:

$$
\begin{array}{rcl}
s \models_{\mathbf{M}} P(t_1, \ldots, t_n) & \textit{sii} & P_{\mathbf{M}}(s(t_1), \ldots, s(t_n)) \\
s \models_{\mathbf{M}} \neg A & \textit{sii} & s \not\models_{\mathbf{M}} A \\
s \models_{\mathbf{M}} (A \land B) & \textit{sii} & s \models_{\mathbf{M}} A \text{ y } s \models_{\mathbf{M}} B \\
s \models_{\mathbf{M}} (A \lor B) & \textit{sii} & s \models_{\mathbf{M}} A \text{ o } s \models_{\mathbf{M}} B \\
s \models_{\mathbf{M}} (A \supset B) & \textit{sii} & s \not\models_{\mathbf{M}} A \text{ o } s \models_{\mathbf{M}} B \\
s \models_{\mathbf{M}} (A \iff B) & \textit{sii} & (s \models_{\mathbf{M}} A \text{ sii } s \models_{\mathbf{M}} B) \\
s \models_{\mathbf{M}} \forall x_i.A & \textit{sii} & s[x_i \leftarrow a] \models_{\mathbf{M}} A \text{ para todo } a \in M \\
s \models_{\mathbf{M}} \exists x_i.A & \textit{sii} & s[x_i \leftarrow a] \models_{\mathbf{M}} A \text{ para algún } a \in M
\end{array}
$$

Validez

- Una fórmula A es satisfactible en **M** sii existe una asignación s tal que
$$s \models_{\mathbf{M}} A$$
- Una fórmula A es satisfactible sii existe un **M** tal que A es satisfactible en **M**. En caso contrario se dice que A es insatisfactible.
- Una fórmula A es válida en **M** sii
$$s \models_{\mathbf{M}} A, \text{ para toda asignación } s$$
- Una fórmula A es válida sii es válida en toda estructura **M**.
- **Nota:** A es válida sii $\neg A$ es insatisfactible.

Teorema de Church

No existe un algoritmo que pueda determinar si una fórmula de primer orden es válida

- Como consecuencia el método de resolución que veremos para la lógica de primer orden no es un procedimiento efectivo (i.e. un algoritmo)
- Es un procedimiento de semi-decisión:
  - si una sentencia es insatisfactible hallará una refutación,
  - pero si es satisfactible puede que no se detenga

Resolución en lógica de primer orden

Paradigmas de Lenguajes de Programación

Departamento de Computación, FCEyN, UBA

28 de septiembre de 2017

Forma clausal

- Es una forma normal conjuntiva, en notación de conjuntos.
- Análogo a la forma clausal del marco proposicional.
- Pero requiere tener en cuenta los cuantificadores.
- El pasaje a forma clausal consiste en seis pasos de conversión.
  1. Escribir la fórmula en términos de $\wedge, \vee, \neg, \forall, \exists$ (i.e. eliminar implicación).
  2. Pasar a forma normal negada.
  3. Pasar a forma normal prenexa (opcional).
  4. Pasar a forma normal de Skolem.
  5. Pasar matriz a forma normal conjuntiva.
  6. Distribuir cuantificadores universales.

Forma normal negada

El conjunto de fórmulas en forma normal negada (FNN) se define inductivamente como:

1. Para cada fórmula atómica A, A y $\neg A$ están en FNN.
2. Si $A, B \in \mathrm{FNN}$, entonces $(A \vee B), (A \wedge B) \in \mathrm{FNN}$.
3. Si $A \in \mathrm{FNN}$, entonces $\forall x.A, \exists x.A \in \mathrm{FNN}$.

Ejemplos

- $\neg\exists x.((P(x) \vee \exists y.R(x,y)) \supset (\exists z.R(x,z) \vee P(a)))$ no está en FNN.
- $\forall x.((P(x) \vee \exists y.R(x,y)) \wedge (\forall z.\neg R(x,z) \wedge \neg P(a)))$ está en FNN.

Forma normal negada

Toda fórmula es lógicamente equivalente a otra en FNN.

Dem.
Por inducción estructural usando:

$$\begin{array}{rcl} \neg(A \wedge B) & \iff & \neg A \vee \neg B \\ \neg(A \vee B) & \iff & \neg A \wedge \neg B \\ \neg\neg A & \iff & A \end{array} \qquad \begin{array}{rcl} \neg\forall x.A & \iff & \exists x.\neg A \\ \neg\exists x.A & \iff & \forall x.\neg A \end{array}$$

Ejemplos

- $\neg\exists x.(\neg(P(x) \vee \exists y.R(x,y)) \vee (\exists z.R(x,z) \vee P(a)))$ se transforma en
- $\forall x.((P(x) \vee \exists y.R(x,y)) \wedge (\forall z.\neg R(x,z) \wedge \neg P(a)))$

Forma normal prenexa

Fórmula de la forma $Q_1 x_1 \dots Q_n x_n . B$, $n \geq 0$, donde

- B sin cuantificadores (llamada matriz)
- $x_1, \dots, x_n$ son variables
- $Q_i \in \{\forall, \exists\}$

Forma prenexa

Toda fórmula A es lógicamente equivalente a una fórmula B en forma prenexa.

Demostración

Por inducción estructural usando (las fórmulas se asumen rectificadas):

$$\begin{array}{rclcrcl}
(\forall x.A) \wedge B & \iff & \forall x.(A \wedge B) & \quad & (\forall x.A) \vee B & \iff & \forall x.(A \vee B) \\
(A \wedge \forall x.B) & \iff & \forall x.(A \wedge B) & & (A \vee \forall x.B) & \iff & \forall x.(A \vee B) \\
(\exists x.A) \wedge B & \iff & \exists x.(A \wedge B) & & (\exists x.A) \vee B & \iff & \exists x.(A \vee B) \\
(A \wedge \exists x.B) & \iff & \exists x.(A \wedge B) & & (A \vee \exists x.B) & \iff & \exists x.(A \vee B)
\end{array}$$

- Nota: Con estas equivalencias basta, si asumimos que A está en FNN.

Ejemplo

1. $\forall x.\neg P(x) \wedge (\exists y.Q(y) \vee \forall z.P(z))$
2. $\forall x.\neg P(x) \wedge (\exists y.(Q(y) \vee \forall z.P(z)))$
3. $\exists y.(\forall x.\neg P(x) \wedge (Q(y) \vee \forall z.P(z)))$
4. $\exists y.(\forall x.\neg P(x) \wedge \forall z.(Q(y) \vee P(z)))$
5. $\exists y.\forall z.(\forall x.\neg P(x) \wedge (Q(y) \vee P(z)))$
6. $\exists y.\forall z.\forall x.(\neg P(x) \wedge (Q(y) \vee P(z)))$

Forma normal de Skolem

- Hasta ahora tenemos una fórmula que:
  1. está escrita en términos de $\wedge, \vee, \neg, \forall, \exists$,
  2. si tiene negaciones, solamente se aplican a átomos (forma normal negada),
  3. (opcionalmente) si tiene cuantificadores, se encuentran todos en el prefijo (forma normal prenexa).
- El proceso de pasar una fórmula a forma normal de Skolem se llama skolemización.
- El objetivo de la skolemización es
  1. eliminar los cuantificadores existenciales
  2. sin alterar la satisfactibilidad.

Eliminación de cuantificadores existenciales

- ¿Cómo eliminamos los $\exists$ sin cambiar la satisfactibilidad?
- Introducimos "testigos" para los mismos.
  - Todo cuantificador existencial se instancia en una constante o función de skolem.
  - Ejemplo: $\exists x.P(x)$ se skolemiza a $P(c)$ donde c es una nueva constante que se agrega al lenguaje de primer orden.
  - Estas funciones y constantes se suelen conocer como parámetros.

¿Cómo se altera el significado de la fórmula?

Prop.

Si A' es el resultado de skolemizar A, entonces A es satisfactible sii A' es satisfactible.

- Consecuencia: La skolemización preserva insatisfactibilidad.
- Esto es suficiente para poder aplicar el método de resolución, tal como veremos.

¿Preservación de validez?

- ¿Podremos eliminar los cuantificadores existenciales, usando Skolemización, sin alterar la validez?
- Esto es mucho más fuerte que preservar satisfactibilidad...
- Respuesta: No.
- Ejemplo: $\exists x.(P(a) \supset P(x))$ es válida pero $P(a) \supset P(b)$ no lo es.
- Tal como se mencionó, la skolemización sí preserva satisfactibilidad y ello es suficiente para el método de resolución.

Skolemización

Cada ocurrencia de una subfórmula

$$\exists x.B$$

en A se reemplaza por

$$B\{x \leftarrow f(x_1, \ldots, x_n)\}$$

donde

- $\bullet\{\bullet \leftarrow \bullet\}$ es la operación usual de sustitución (sustituir todas las ocurrencias libres de una variable en una expresión - fórmula o término - por otra expresión).
- Si $\exists x.B$ forma parte de una fórmula mayor, decimos que x depende de las variables libres de B, y sólo de ellas (por ejemplo, en $\forall z.\forall y.\exists x.P(y, x)$ la x depende de y).
- f es un símbolo de función nuevo y las $x_1, \ldots, x_n$ son las variables de las que depende x en B.

Definición de forma normal de Skolem (1/2)

- Sea A una sentencia rectificada en FNN.
  - No es necesario que esté en forma prenexa.
- Una forma normal de Skolem de A (escrito $\mathbf{SK}(A)$) es una fórmula sin existenciales que se obtiene recursivamente como sigue.
- Sea A' cualquier subfórmula de A.
  - Si A' es una fórmula atómica o su negación, $\mathbf{SK}(A') = A'$.
  - Si A' es de la forma $(B \star C)$ con $\star \in \{\vee, \wedge\}$, entonces $\mathbf{SK}(A') = (\mathbf{SK}(B) \star \mathbf{SK}(C))$.
  - Si A' es de la forma $\forall x.B$, entonces $\mathbf{SK}(A') = \forall x.\mathbf{SK}(B)$.
  - *Sigue en siguiente diapositiva.*

Definición de forma normal de Skolem (2/2)

- Si A' es de la forma $\exists x.B$ y $\{x, y_1, \ldots, y_m\}$ son las variables libres de B[1], entonces
  1. Si $m > 0$, crear un nuevo símbolo de función de Skolem, f_x de aridad m y definir
  $$\mathbf{SK}(A') = \mathbf{SK}(B\{x \leftarrow f_x(y_1, \ldots, y_m)\})$$
  2. Si $m = 0$, crear una nueva constante de Skolem c_x y
  $$\mathbf{SK}(A') = \mathbf{SK}(B\{x \leftarrow c_x\})$$

Nota: dado que A está rectificada, cada f_x y c_x es única.

[1] Notar que se ligan en A dado que A es sentencia

Ejemplos

Considere la fórmula

$$\forall x.\left(P(a) \vee \exists y.(Q(y) \wedge \forall z.(P(y,z) \vee \exists u.Q(x,u)))\right) \vee \exists w.Q(w)$$

La forma normal de Skolem es:

$$\forall x.(P(a) \vee (Q(g(x)) \wedge \forall z.(P(g(x),z) \vee Q(x,f(x))))) \vee Q(c)$$

Ejemplos

- Considere la sentencia:

$$\forall x.\exists y.\exists z.R(x,y,z)$$

1. Alternativa 1 (rojo, azul)
    1.1 $\forall x.\exists y.\exists z.R(x,y,z)$
    1.2 $\forall x.\exists z.R(x,f(x),z)$
    1.3 $\forall x.R(x,f(x),g(x))$
2. Alternativa 2 (azul, rojo)
    2.1 $\forall x.\exists y.\exists z.R(x,y,z)$
    2.2 $\forall x.\exists y.R(x,y,h(x,y))$
    2.3 $\forall x.R(x,k(x),h(x,k(x)))$
3. La skolemización no es determinística.

- Es mejor skolemizar de afuera hacia adentro.

Forma clausal

Hasta ahora tenemos una fórmula que:

1. está escrita en términos de $\wedge, \vee, \neg, \forall$;
2. si tiene negaciones, solamente se aplican a átomos (forma normal negada);
3. si tiene cuantificadores, son universales (forma normal de Skolem);
4. si está en forma normal prenexa y tiene cuantificadores, éstos se encuentran todos en el prefijo.

$$\forall x_1 \ldots \forall x_n . B$$

Forma clausal

$$\forall x_1 \ldots \forall x_n . B$$

1. Pasar B a forma normal conjuntiva B' como si fuera una fórmula proposicional arrojando

$$\forall x_1 \ldots \forall x_n . B'$$

2. Distribuir los cuantificadores sobre cada conjunción usando la fórmula válida $\forall x.(A \wedge B) \iff \forall x.A \wedge \forall x.B$ arrojando una conjunción de cláusulas

$$\forall x_1 \ldots \forall x_n . C_1 \wedge \ldots \wedge \forall x_1 \ldots \forall x_n . C_m$$

   donde cada C_i es una disyunción de literales

3. Se simplifica escribiendo $\{C_1, \ldots, C_m\}$.

Ejemplo

$$\forall x.\forall z.(P(a) \vee (Q(g(x)) \wedge (P(g(x), z) \vee Q(x, f(x))))) \vee Q(c)$$

1. Pasamos la matriz a forma normal conjuntiva
$$\forall x.\forall z.\big([P(a) \vee Q(g(x)) \vee Q(c)] \wedge [P(a) \vee P(g(x), z) \vee Q(x, f(x)) \vee Q(c)]\big)$$
2. Distribuimos los cuantificadores
$$\forall x.\forall z.\big[P(a) \vee Q(g(x)) \vee Q(c)\big] \wedge \forall x.\forall z.\big[P(a) \vee P(g(x), z) \vee Q(x, f(x)) \vee Q(c)]$$
3. Pasamos a notación de conjuntos
$$\Big\{\{P(a), Q(g(x)), Q(c)\}, \{P(a), P(g(x), z), Q(x, f(x)), Q(c)\}\Big\}$$

Forma clausal - Resumen

1. Escribir la fórmula en términos de $\wedge, \vee, \neg, \forall, \exists$ (i.e. eliminar implicación)
2. Pasar a forma normal negada
3. Pasar a forma normal prenexa
4. Pasar a forma normal de Skolem (puede hacerse antes de 3)
5. Pasar matriz a forma normal conjuntiva
6. Distribuir cuantificadores universales

Nota: todos los pasos preservan validez lógica, salvo la skolemización (que preserva la satisfactibilidad).

Resolución en lógica proposicional

Consideremos la siguiente fórmula:

$$(\forall x.P(x)) \land \neg P(a)$$

- Es satisfactible? NO.
- Su forma clausal es

$$\{\{P(x)\}, \{\neg P(a)\}\}$$

- ¿Podemos aplicar la regla de resolución?

$$\frac{\{A_1, \ldots, A_m, Q\} \quad \{B_1, \ldots, B_n, \neg Q\}}{\{A_1, \ldots, A_m, B_1, \ldots, B_n\}}$$

- No. $P(x)$ y $P(a)$ no son idénticos, son ... unificables.

Ahora sí, la Regla de resolución

$$\frac{\{B_1, \ldots, B_k, A_1, \ldots, A_m\} \quad \{\neg D_1, \ldots, \neg D_j, C_1, \ldots, C_n\}}{\sigma(\{A_1, \ldots, A_m, C_1, \ldots, C_n\})}$$

donde σ es el MGU de $\{B_1, \ldots, B_k, D_1, \ldots, D_j\}$.

- Asumimos que las cláusulas $\{B_1, \ldots, B_k, A_1, \ldots, A_m\}$ y $\{\neg D_1, \ldots, \neg D_j, C_1, \ldots, C_n\}$ no tienen variables en común; en caso contrario se renombran las variables.
- Observar que $\sigma(B_1) = \ldots = \sigma(B_k) = \sigma(D_1) = \ldots = \sigma(D_j)$.
- La cláusula $\sigma(\{A_1, \ldots, A_m, C_1, \ldots, C_n\})$ se llama resolvente (de $\{B_1, \ldots, B_k, A_1, \ldots, A_m\}$ y $\{\neg D_1, \ldots, \neg D_j, C_1, \ldots, C_n\}$).

Método de resolución

- Las siguientes nociones son análogas al caso proposicional.
  - Cláusula vacía
  - Paso de resolución
  - Refutación
- Al igual que en el caso proposicional contamos con el siguiente resultado.

Teorema de Herbrand-Skolem-Gödel

Cada paso de resolución preserva satisfactibilidad.

Ejemplo

Supongamos que dado A, obtenemos $\neg A$, lo convertimos a forma clausal y nos queda: $C_1 \wedge C_2 \wedge C_3$ donde

- $C_1 = \forall z_1.\forall x.(\neg P(z_1, a) \vee \neg P(z_1, x) \vee \neg P(x, z_1))$
- $C_2 = \forall z_2.(P(z_2, f(z_2)) \vee P(z_2, a))$
- $C_3 = \forall z_3.(P(f(z_3), z_3) \vee P(z_3, a))$

Abreviado (sin cuantificadores + notación de conjuntos):

$$\Big\{\{\neg P(z_1, a), \neg P(z_1, x), \neg P(x, z_1)\},\\ \{P(z_2, f(z_2)), P(z_2, a)\},\\ \{P(f(z_3), z_3), P(z_3, a)\}\Big\}$$

Ejemplo

$$\begin{aligned} C_1 &= \{\neg P(z_1, a), \neg P(z_1, x), \neg P(x, z_1)\}, \\ C_2 &= \{P(z_2, f(z_2)), P(z_2, a)\}, \\ C_3 &= \{P(f(z_3), z_3), P(z_3, a)\}. \end{aligned}$$

1. De C_1 y C_2 con $\{z_1 \leftarrow a, x \leftarrow a, z_2 \leftarrow a\}$:
   $C_4 = \{P(a, f(a))\}$
2. De C_1 y C_3 con $\{z_1 \leftarrow a, x \leftarrow a, z_3 \leftarrow a\}$:
   $C_5 = \{P(f(a), a)\}$
3. De C_1 y C_5 con $\{z_1 \leftarrow f(a), x \leftarrow a\}$:
   $C_6 = \{\neg P(a, f(a))\}$
4. De C_4 y C_6: $\square$

Otro ejemplo

- Consideremos las siguientes fórmulas:
  - $F_1 : \forall x.\forall y.(P(x, y) \supset P(y, x))$
  - $F_2 : \forall x.\forall y.\forall z.((P(x, y) \wedge P(y, z)) \supset P(x, z))$
  - $F_3 : \forall x.\exists y.P(x, y)$
- Establecen que P es simétrica, transitiva y total

Apelar al método de resolución para probar que P es reflexiva

- Tenemos que probar
$$F_1 \wedge F_2 \wedge F_3 \supset \forall x.P(x, x)$$
- Para ello, basta con probar que
$$\neg(F_1 \wedge F_2 \wedge F_3 \supset \forall x.P(x, x))$$
es insatisfactible.

  Notar que $\neg(F_1 \wedge F_2 \wedge F_3 \supset \forall x.P(x, x))$ es equivalente a $F_1 \wedge F_2 \wedge F_3 \wedge \neg\forall x.P(x, x)$.

Otro ejemplo (cont.)

Pasamos la siguiente fórmula a forma clausal.

$$\neg(F_1 \wedge F_2 \wedge F_3 \supset \forall x.P(x,x))$$

- Donde
  - $F_1 : \forall x.\forall y.(P(x,y) \supset P(y,x))$
  - $F_2 : \forall x.\forall y.\forall z.((P(x,y) \wedge P(y,z)) \supset P(x,z))$
  - $F_3 : \forall x.\exists y.P(x,y)$
- Sus formas clausales son:
  - $C_1 : \{\neg P(x,y), P(y,x)\}$
  - $C_2 : \{\neg P(x,y), \neg P(y,z), P(x,z)\}$
  - $C_3 : \{P(x,f(x))\}$
- Finalmente, nos queda hallar una refutación a partir del conjunto de cláusulas

$$\{C_1, C_2, C_3, \{\neg P(a,a)\}\}$$

Otro ejemplo (cont.)

$$\begin{aligned} C_1 &= \{\neg P(x,y), P(y,x)\}, \\ C_2 &= \{\neg P(x,y), \neg P(y,z), P(x,z)\}, \\ C_3 &= \{P(x,f(x))\}, \\ C_4 &= \{\neg P(a,a)\}. \end{aligned}$$

1. De C_2 y C_4 con $\{x \leftarrow a, z \leftarrow a\}$:
   $C_5 = \{\neg P(a,y), \neg P(y,a)\}$
2. De C_3 y C_5 con $\{x \leftarrow a, y \leftarrow f(a)\}$:
   $C_6 = \{\neg P(f(a),a)\}$
3. De C_1 y C_6 con $\{x \leftarrow a, y \leftarrow f(a)\}$:
   $C_7 = \{\neg P(a,f(a))\}$
4. De C_3 y C_7 con $\{x \leftarrow a\}$: $\square$

Diferencias con proposicional

1. En proposicional

$$\frac{\{Q, A_1, \ldots, A_m\} \quad \{\neg Q, B_1, \ldots, B_n\}}{\{A_1, \ldots, A_m, B_1, \ldots, B_n\}}$$

2. En primer orden

$$\frac{\{B_1, \ldots, B_k, A_1, \ldots, A_m\} \quad \{\neg D_1, \ldots, \neg D_j, C_1, \ldots, C_n\}}{\sigma(\{A_1, \ldots, A_m, C_1, \ldots, C_n\})}$$

donde σ es el MGU de $\{B_1, \ldots, B_k, D_1, \ldots, D_j\}$.

Regla de resolución binaria

$$\frac{\{B, A_1, \ldots, A_m\} \quad \{\neg D, C_1, \ldots, C_n\}}{\sigma(\{A_1, \ldots, A_m, C_1, \ldots, C_n\})}$$

donde σ es el MGU de $\{B, D\}$.

- Es incompleta.
- Ejemplo: intentar refutar $\{\{P(x), P(y)\}, \{\neg P(v), \neg P(w)\}\}$

Regla de resolución binaria

- Se puede recuperar la completitud incorporando una regla adicional: factorización.

$$\frac{\{B_1, \dots, B_k, A_1, \dots, A_m\}}{\sigma(\{B_1, A_1, \dots, A_m\})}$$

  donde σ es el MGU de $\{B_1, \dots, B_k\}$.

- En el ejemplo anterior

  1. $\{\{P(x), P(y)\}, \{\neg P(v), \neg P(w)\}\}$
  2. $\{\{P(x), P(y)\}, \{\neg P(v), \neg P(w)\}, \{P(z)\}\}$ (fact.)
  3. $\{\{P(x), P(y)\}, \{\neg P(v), \neg P(w)\}, \{P(z)\}, \{\neg P(u)\}\}$ (fact.)
  4. $\{\{P(x), P(y)\}, \{\neg P(v), \neg P(w)\}, \{P(z)\}, \{\neg P(u)\}, \square\}$ (resolución (binaria))

Resolución SLD y Prolog

Departamento de Computación, FCEyN, UBA

5 de octubre de 2017

Método de resolución (Repaso)

- Dado un conjunto de cláusulas S
  1. Aplicar repetidamente la regla de resolución dando lugar a pasos de resolución
  2. Hasta producir una refutación (i.e. conjunto de cláusulas conteniendo la cláusula vacía o contradicción)
- En este caso decimos que "*S tiene una refutación por resolución*"
- Sustento del método

  Cada paso de resolución preserva insatisfactibilidad

  $+$ las refutaciones son trivialmente insatisfactibles

  $=$ refutación por resolución general de S implica que S es insatisfactible

Regla de resolución (Repaso)

$$\frac{\{B_1, \ldots, B_k, A_1, \ldots, A_m\} \quad \{\neg D_1, \ldots, \neg D_l, C_1, \ldots, C_n\}}{\sigma(\{A_1, \ldots, A_m, C_1, \ldots, C_n\})}$$

donde σ es el MGU de $\{B_1, \ldots, B_k, D_1, \ldots, D_l\}$.

- Asumimos que las cláusulas $\{B_1, \ldots, B_k, A_1, \ldots, A_m\}$ y $\{\neg D_1, \ldots, \neg D_l, C_1, \ldots, C_n\}$ no tienen variables en común; en caso contrario se renombran las variables

- Observar que $\sigma(B_1) = \ldots = \sigma(B_k) = \sigma(D_1) = \ldots = \sigma(D_l)$

- La cláusula $\sigma(\{A_1, \ldots, A_m, B_1, \ldots, B_n\})$ se llama resolvente (de $\{B_1, \ldots, B_k, A_1, \ldots, A_m\}$ y $\{\neg D_1, \ldots, \neg D_l, C_1, \ldots, C_n\}$)

Resolución lineal

- Si bien el método de resolución general es completo, hallar refutaciones es un proceso muy caro en el caso general
- El espacio de búsqueda producido puede ser enorme
- Hay un alto grado de no-determinismo
  - ¿Qué cláusulas elegimos? Regla de búsqueda
  - ¿Qué literales eliminamos? Regla de selección
- Se precisan restricciones (regla de búsqueda y selección) para reducir el espacio de búsqueda (utilidad práctica)
- Es deseable que dichas restricciones no renuncien a la completitud del método

Resolución lineal

Una secuencia de pasos de resolución a partir de S es lineal si es de la forma:

$$
\begin{tikzcd}
B_1 \arrow[dr, dash] & C_0 \arrow[d, dash] & \\
B_2 \arrow[dr, dash] & C_1 \arrow[d, dash] & \sigma_1 \\
B_3 \arrow[dr, dash] & C_2 \arrow[d, dash] & \sigma_2 \\
 & C_3 & \sigma_3 \\
B_{p-1} \arrow[dr, dash] & \ldots \arrow[d, dash] & \\
B_p \arrow[dr, dash] & C_{p-1} \arrow[d, dash] & \sigma_{p-1} \\
 & C_p = \square & \sigma_p
\end{tikzcd}
$$

donde C_0 y cada B_i es un elemento de S (o algún C_j con $j < i$)

Resolución lineal

- En general, reduce el espacio de búsqueda considerablemente
- Preserva completitud
- Sin embargo sigue siendo altamente no-determinístico
  - El criterio de búsqueda deja espacio para refinamientos
  - No se especificó ningún criterio de selección

Cláusulas de Horn

- Mayor eficiencia en el proceso de producir refutaciones sin perder completitud puede lograrse para subclases de fórmulas
- Una de estas clases es la de Cláusulas de Horn
- Una cláusula de Horn es una disyunción de literales que tiene a lo sumo un literal positivo
- Para conjuntos de cláusulas de Horn puede usarse una variante de resolución lineal, llamada resolución SLD, que goza de buenas propiedades

Forma clausal (Repaso)

Conjunción de sentencias prenexas de la forma $\forall x_1 \ldots \forall x_m C$ donde

1. C es una disyunción de literales y
2. los conjuntos $\{x_1, \ldots, x_m\}$ de variables ligadas son disjuntos para todo par distinto de cláusulas.

Nota:

- Cada $\forall x_1 \ldots \forall x_m C$ se llama cláusula
- La forma clausal

$$\forall x_{11} \ldots \forall x_{1m_1} C_1 \wedge \ldots \wedge \forall x_{k1} \ldots \forall x_{km_k} C_k$$

se escribe $\{C'_1, \ldots, C'_k\}$ donde C'_i resulta de reemplazar la disyunción de literales C_i por el conjunto de literales asociado

Cláusulas de Horn

Una cláusula de Horn es de la forma $\forall x_1 \ldots \forall x_m C$ tal que la disyunción de literales C tiene a lo sumo un literal positivo

Nota: C puede tomar una de las formas:

- $\{B, \neg A_1, \ldots, \neg A_n\}$
- $\{B\}$
- $\{\neg A_1, \ldots, \neg A_n\}$ (llamada cláusula "goal" o negativa)

Relación con fórmulas de primer orden

- No toda fórmula de primer orden puede expresarse como una cláusula de Horn
- Ejemplos
  - $\forall x.(P(x) \vee Q(x))$
  - $(\forall x.P(x) \vee \forall x.Q(x)) \supset \forall x.(P(x) \vee Q(x))$
- Sin embargo, el conjunto de cláusulas de Horn es suficientemente expresivo para representar programas, en la visión de resolución como computación

Resolución SLD

- Cláusula de definición ("Definite Clause")
  - Cláusula de la forma $\forall x_1 \ldots \forall x_m C$ tal que la disyunción de literales C tiene exactamente un literal positivo
- Sea $S = P \cup \{G\}$ un conjunto de cláusulas de Horn (con nombre de variables disjuntos) tal que
  - P conjunto de cláusulas de definición y
  - G un cláusula negativa
- $S = P \cup \{G\}$ son las cláusulas de entrada
  - P se conoce como el programa o base de conocimientos y
  - G el goal o meta

Resolución SLD

Un secuencia de pasos de resolución SLD para S es una secuencia

$$< N_0, N_1, \ldots, N_p >$$

de cláusulas negativas que satisfacen las siguientes dos condiciones.

1. N_0 es el goal G
2. *sigue en transparencia siguiente*

Resolución SLD

2. para todo N_i en la secuencia, $0 < i < p$, si N_i es

$$\{\neg A_1, \ldots, \neg A_{k-1}, \neg A_k, \neg A_{k+1}, \ldots, \neg A_n\}$$

entonces hay alguna cláusula de definición C_i de la forma $\{A, \neg B_1, \ldots, \neg B_m\}$ en P tal que A_k y A son unificables con MGU σ, y si

- $m = 0$, entonces N_{i+1} es $\{\sigma(\neg A_1, \ldots, \neg A_{k-1}, \neg A_{k+1}, \ldots, \neg A_n)\}$
- $m > 0$, entonces N_{i+1} es $\{\sigma(\neg A_1, \ldots, \neg A_{k-1}, \neg B_1, \ldots, \neg B_m, \neg A_{k+1}, \ldots, \neg A_n)\}$

Refutación SLD

Una refutación SLD es una secuencia de pasos de resolución SLD $< N_0, \ldots, N_p >$ tal que $N_p = \square$

$$
\begin{array}{ccc}
C_1 & N_0 = G & \\
C_2 & N_1 & \sigma_1 \\
C_3 & N_2 & \sigma_2 \\
 & N_3 & \sigma_3 \\
C_{p-1} & \ldots & \\
C_p & N_{p-1} & \sigma_{p-1} \\
 & N_p = \square & \sigma_p
\end{array}
$$

Sustitución respuesta

- En cada paso, las cláusulas $\{\neg A_1, \ldots, \neg A_{k-1}, \neg A_k, \neg A_{k+1}, \ldots, \neg A_n\}$ y $\{A, \neg B_1, \ldots, \neg B_m\}$ son resueltas
  - No se especifica regla de búsqueda alguna
- Los átomos A_k y A son unificados con MGU σ_i
- El literal A_k se llama átomo seleccionado de N_i
  - No se especifica regla de selección alguna
- Sustitución respuesta es la sustitución

$$\sigma_p \circ \ldots \circ \sigma_1$$

  - se usa en Prolog para extraer la salida del programa

Ejemplo

Consideremos las siguientes cláusulas de definición

- $C_1 = \{add(U, 0, U)\}$
- $C_2 = \{add(X, succ(Y), succ(Z)), \neg add(X, Y, Z)\}$

y la cláusula goal G

$$\{\neg add(succ(0), V, succ(succ(0)))\}$$

- Deseamos mostrar que el conjunto de estas cláusulas (i.e. $\{C_1, C_2, G\}$) es insatisfactible
- Contamos con la siguiente refutación SLD

Ejemplo

$$C_1 = \{add(U, 0, U)\}$$
$$C_2 = \{add(X, succ(Y), succ(Z)), \neg add(X, Y, Z)\}$$

Cláusula goal	**Cláusula de entrada**	**Sust.**
$\{\neg add(succ(0), V, succ(succ(0)))\}$	C_2	
$\{\neg add(succ(0), Y, succ(0))\}$	C_1	σ_1
$\square$		σ_2

donde

$$\sigma_1 = \{X \leftarrow succ(0), V \leftarrow succ(Y), Z \leftarrow succ(0), \}$$
$$\sigma_2 = \{U \leftarrow succ(0), Y \leftarrow 0\}$$

- La sustitución resultado es $\sigma_2 \circ \sigma_1 =$

$$\{X \leftarrow succ(0), V \leftarrow succ(0), Z \leftarrow succ(0), U \leftarrow succ(0), Y \leftarrow 0\}$$

Corrección y completitud

Corrección

Si un conjunto de cláusulas de Horn tiene una refutación SLD, entonces es insatisfactible

Completitud

Dado un conjunto de cláusulas de Horn $P \cup \{G\}$ tal como se describió, si $P \cup \{G\}$ es insatisfactible, existe una refutación SLD cuya primera cláusula es G.

Resolución SLD en Prolog

- Prolog utiliza resolución SLD con las siguientes restricciones
  - Regla de búsqueda: se seleccionan las cláusulas de programa de arriba hacia abajo, en el orden en que fueron introducidas
  - Regla de selección: seleccionar el átomo de más a la izquierda
- La suma de regla de búsqueda y regla de selección se llama estrategia
- Cada estrategia determina un árbol de búsqueda o árbol SLD

Ejemplo

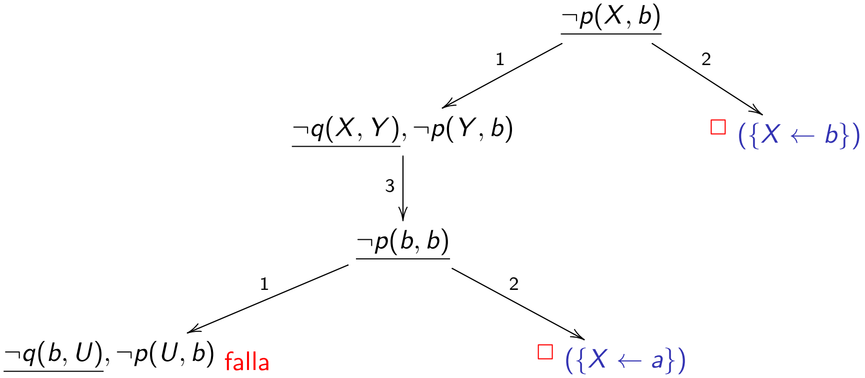
Cláusulas de Def.

1. $\{p(X,Z), \neg q(X,Y), \neg p(Y,Z)\}$
2. $\{p(X,X)\}$
3. $\{q(a,b)\}$

Goal

$\{\neg p(X,b)\}$

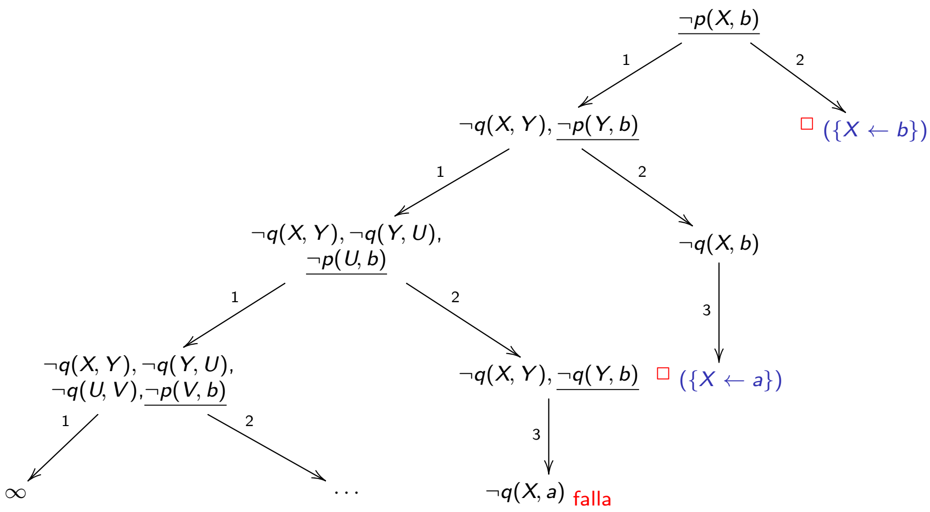
Ejemplo - árbol SLD



Variando la regla de selección

- Si variamos la regla de selección, varía el árbol SLD asociado
- Vamos a asumir ahora que la regla de selección es "seleccionar el de más a la derecha"

Ejemplo - átomo de más la derecha



Variando la regla de búsqueda

- Si variamos la regla de búsqueda, también varía el árbol SLD asociado
- Por ejemplo: "seleccionar las cláusulas de programa de abajo hacia arriba"

Motivación

- Ahora exploramos de qué manera resolución SLD puede usarse para computar
- Asimismo, vamos a enfatizar el rol de la sustitución respuesta como “resultado de cálculo”

Motivación

- Recordar el ejemplo de la suma
  - $C_1 = \{add(U, 0, U)\}$
  - $C_2 = \{add(X, succ(Y), succ(Z)), \neg add(X, Y, Z)\}$
- Estas cláusulas pueden verse como una definición recursiva de la suma
- Supongamos que queremos saber si, dada esa definición,

$$\exists V.add(succ(0), V, succ(succ(0)))$$

- En otras palabras

“¿Existe V tal que $1 + V = 2$?”

Motivación

Podemos plantearlo como la validez de la fórmula

$$C_1 \wedge C_2 \supset \exists V.add(succ(0), V, succ(succ(0)))$$

Es decir,

$$\underbrace{(\forall U.C_1 \wedge \forall X.\forall Y.\forall Z.C_2)}_{\text{Define la suma}} \supset$$

$$\underbrace{\exists V.add(succ(0), V, succ(succ(0)))}_{\text{Pide calcular } V} \text{ es válida}$$

Motivación

Esto es lo mismo que preguntarse por la insatisfactilidad de

$$C_1 \wedge C_2 \wedge \neg \exists V. add(succ(0), V, succ(succ(0)))$$

O lo que es lo mismo

$$C_1 \wedge C_2 \wedge \forall V. \neg add(succ(0), V, succ(succ(0)))$$

Motivación

Resolución SLD se dispara a partir del conjunto de cláusulas

$$\Big\{\{add(U,0,U),\{add(X,succ(Y),succ(Z)),\neg add(X,Y,Z)\}$$
$$\{\neg add(succ(0),V,succ(succ(0)))\}\Big\}$$

- En caso de tener éxito, va a hallar el V buscado
- Importante observar que
  - No solamente interesa saber que existe tal V (i.e. que existe la refutación)
  - Además, queremos una instancia del mismo
- La sustitución respuesta σ proveerá dichas instancias; las mismas pueden interpretarse como el resultado del cómputo

Programas lógicos - Notación Prolog

Recordar que la resolución SLD parte de un conjunto de cláusulas $S = P \cup \{G\}$ donde

1. P es un conjunto de cláusulas de definición
   - Cláusulas con exactamente un literal positivo

$$\{B, \neg A_1, \ldots, \neg A_n\}$$
$$\{B\}$$

2. G es un goal
   - Cláusula negativa

$$\{\neg A_1, \ldots, \neg A_n\}$$

Notación Prolog para programas lógicos

$$\begin{aligned} B \vee \neg A_1 \vee \ldots \vee \neg A_n \quad & \Longleftrightarrow \quad \neg(A_1 \wedge \ldots \wedge A_n) \vee B \\ & \Longleftrightarrow \quad (A_1 \wedge \ldots \wedge A_n) \supset B \end{aligned}$$

Como consecuencia, las cláusulas en P se escriben

- $B :- A_1, \ldots, A_n.$ para $\{B, \neg A_1, \ldots, \neg A_n\}$ (reglas)
- $B.$ para B (hechos)

Ejemplo de la suma en notación Prolog

Volviendo al ejemplo de la suma, el programa Prolog es

```
add(U,0,U).
add(X,succ(Y),succ(Z)):-add(X,Y,Z).
```

Ingresamos el goal

```
?- add(succ(0),V,succ(succ(0))).
```

La respuesta es:

```
V=succ(0)
```

Búsqueda de refutaciones SLD en Prolog

- Recorre el árbol SLD en profundidad ("depth-first search")
- La ventaja del recorrido en profundidad es que puede ser implementado de manera muy eficiente
  - Se usa una pila para representar los átomos del goal
  - Se hace un push del resolvente del átomo del tope de la pila con la cláusula de definición
  - Se hace un pop cuando el átomo del tope de la pila no unifica con ninguna cláusula de definición más (luego, el átomo que queda en el tope se unifica con la siguiente cláusula de definición)
- Desventaja: ¡puede que no encuentre una refutación SLD aún si existe!

Más sobre Prolog

Veremos dos temas de Prolog que trascienden la lógica subyacente:

1. Cut
2. Negation as failure (negación por falla)

Cut

- Es un precadicado 0-ario, notado !
- Sólo tiene éxito la primera vez que se lo invoca.
- Brinda un mecanismo de control que permite podar el árbol SLD
- Es de carácter extra-lógico (i.e. no se corresponde con un predicado estándar de la lógica)
- Se encuentra presente por cuestiones de eficiencia
- Debe usarse con cuidado dado que puede podarse una rama de interés

Ejemplo

```
?-member(Y,[[1,2],[3,4]]),member(X,Y).

   % devuelve X=1,Y=[1,2]; X=2,Y=[1,2];
              X=3,Y=[3,4]; X=4,Y=[3,4]

?-member(Y,[[1,2],[3,4]]),member(X,Y),!.

   % devuelve X=1,Y=[1,2] solamente

?-member(Y,[[1,2],[3,4]]),!,member(X,Y).

   % devuelve X=1,Y=[1,2]; X=2,Y=[1,2]

?-!,member(Y,[[1,2],[3,4]]),member(X,Y).

   % devuelve lo mismo que el primer ejemplo
```

Ejemplo

```
1.  p(a).
2.  p(X) :- q(X), r(X).
3.  p(X) :- u(X).
4.  q(X) :- s(X).
5.  r(a).
6.  r(b).
7.  s(a).
8.  s(b).
9.  s(c).
10. u(d).
11. t(X) :- p(X).
12. t(e).
```

```
?- p(X).
X=a ; X=a ; X=b ; X=d ;
no

?- p(X),!.
X=a ;
no

?- r(X),!,s(Y).
X=a Y=a ;
X=a Y=b ;
X=a Y=c ;
no

?- r(X), s(Y), !.
X=a Y=a ;
no
```

Ejemplo

```
1.  p(a).                        ?- p(X).
2.  p(X) :- q(X),!,r(X).         X=a ;
3.  p(X) :- u(X).                X=a ;
4.  q(X) :- s(X).                no
5.  r(a).
6.  r(b).                        ?- t(X).
7.  s(a).                        X=a ;
8.  s(b).                        X=a ;
9.  s(c).                        X=e ;
10. u(d).                        no
11. t(X) :- p(X).
12. t(e).
```

Ejemplo

- Definición de `max`

```
max1(X,Y,Y) :- X =< Y.
max1(X,Y,X) :- X>Y.
```

- Es ineficiente. ¿Por qué? Pensar en `max(3,4,Z)`...
- Mejor así

```
max2(X,Y,Y) :- X =< Y, !.
max2(X,Y,X) :- X>Y.
```

- ¿Y esto?

```
max3(X,Y,Y) :- X =< Y, !.
max3(X,Y,X).
```

  - Cambia la semántica de max
  - Probar `max(2,3,2)`

Cut - En general

- Cuando se selecciona un cut, tiene éxito inmediatamente
- Si, debido a backtracking, se vuelve a este cut, su efecto es el de hacer fallar el goal que le dio origen
  - El goal que unificó con la cabeza de la cláusula que contiene al corte y que hizo que esa cláusula se "activara"
- El efecto obtenido es el de descartar soluciones (i.e. no dar más soluciones) de
  1. el goal padre
  2. cualquier goal que ocurre a la izquierda del corte en la cláusula que contiene el corte
  3. todos los objetivos intermedios que se ejecutaron durante la ejecución de los goals precedentes

Negación por falla

- Se dice que un árbol SLD falla finitamente si es finito y no tiene ramas de éxito
- Dado un programa P el conjunto de falla finita de P es $\{B \mid$ B átomo cerrado ('ground') y existe un árbol SLD que falla finitamente con B como raíz$\}$

Negation as failure

$$\frac{B \text{ átomo cerrado} \quad B \text{ en conjunto de falla finita de } P}{\neg B}$$

Predicado not

Negación por falla en Prolog

```
not(G) :- call(G), !, fail.
not(G).
```

Ejemplo

Puede deducirse `not(student(mary))` a partir de

```
student(joe).
student(bill).
student(jim).
teacher(mary).
```

Negación por falla no es negación lógica

```
hombre(juan).
hombre(pedro).
mujer(X) :- not(hombre(X)).
```

- El query `mujer(juan)` da "no", tal como se espera
- El query `mujer(julia)` da "sí", tal como se espera
- ¿Resultado del query `mujer(X)`? (i.e. ¿$\exists X.mujer(X)$?) da "no", contrariamente a lo esperado
- Es importante que el predicado del not se haya instanciado previamente

Negación por falla no es negación lógica

```
hombre(juan).
hombre(pedro).
mujer(X) :- not(hombre(X)).
```

```
not(G) :- call(G), !, fail.
not(G).
```

- Observar: el goal `not(G)` nunca instancia variables de G
  - Si G tiene éxito, `fail` falla y descarta la sustitución
  - Caso contrario, `not(G)` tiene éxito inmediatamente
- En consecuencia, `not(not(hombre(X)))` no es equivalente a `hombre(X)`

Negación por falla no es negación lógica

```
firefighter_candidate(X) :-
                not( pyromaniac(X) ),
                punctual(X).
pyromaniac(attila).
punctual(jeanne_d_arc).
```

- ¿Resultado del query `firefighter_candidate(W)`?
- ¿Por qué `jeanne_d_arc` no es solución?
- Después de todo: ¡Si se intercambian las dos cláusulas en la definición de `firefighter_candidate` sí da a `jeanne_d_arc` como solución!

Programación Orientada a Objetos

Departamento de Computación, FCEyN, UBA

6 de junio de 2017

Enfoque Conceptual/Aplicado

- Vamos a introducir los conceptos fundamentales del paradigma
  - Objetos y modelo de cómputo.
  - Clasificación y herencia
  - Super y static method dispatch
- Ilustraremos estos conceptos con el lenguaje Smalltalk

Objetos - La visión mística (o metáfora)

- Todo programa es una simulación. Cada entidad del sistema que se está simulando se representa en el programa a través de una entidad u objeto.
    - Personificar los objetos físicos o conceptuales de un dominio del mundo real en objetos en el dominio del programa
    - los objetos en el programa tienen las características y capacidades del mundo real que nos interese modelar.
- Todas las componentes de un sistema son objetos

Modelo de Computación - La visión mística (o metáfora)

- El modelo de computación consiste en el envío de mensajes: Un sistema está formado por *objetos* que comunican a través del intercambio de mensajes.
- Mensajes es una solicitud para que un objeto lleve a cabo una de sus operaciones
- El receptor, es decir, el objeto que recibe el mensaje, determina cómo llevar a cabo la operación.

Ejemplo: `"unCirculo radio"`

- `unCirculo` es el objeto receptor
- `radio` es el mensaje que se le envía.

Objetos

- El conjunto de mensajes a los que un objeto responde se denomina interfaz o protocolo
- La forma en que un objeto lleva a cabo una operación está descripta por un método (describe la implementación de las operaciones)
- La forma en que un objeto lleva a cabo una operación puede depender de un estado interno
  - El estado se representa a través de un conjunto de colaboradores internos (también llamados atributos o variables de instancia)

unRectangulo

- interfaz: `area`
- atributos: `alto` y `ancho`
- método: `area`

```
          ^alto*ancho
```

Objetos

La única manera de interactuar con un objeto es a través del envío de mensajes

- la implementación de un objeto no puede depender de los detalles de implementación de otros objetos
- principio básico heredado de los Tipos Abstractos de Datos, antecesores de los Objetos:

Principio de ocultamiento de la información

El estado de un objeto es privado y solamente puede ser consultado o modificado por sus métodos

Mensajes - Tipos y Sintaxis en Smalltalk

Tipos

- Unarios:
  - anArray **size**
  - 1 **factorial**
- Binarios:
  - 1 **+** 2
  - December**,** 2014
- Keyword
  - anArray **at:** 1 **put:** 2
  - aList **add:** 2
  - Rectangulo deBase: 10 cm yAltura: 20 cm

Prioridad

unarios > binarios > keywords (de izquierda a derecha)

anArray at: 2 * 10 factorial * 3 put: name size

== (anArray at: ((2 * (10 factorial)) *3) put: (name size))

Detrás del modelo de cómputo: Method dispatch

- La interacción entre objetos se lleva a cabo a través de envío de mensajes
- Al recibir un mensaje se activa el método correspondiente
- Para poder procesar este mensaje es necesario hallar la declaración del método que se pretende ejecutar
- El proceso de establecer la asociación entre el mensaje y el método a ejecutar se llama method dispatch
- Si el method dispatch se hace en tiempo de
  - compilación (i.e. el método a ejecutar se puede determinar a partir del código fuente): se habla de method dispatch estático
  - ejecución: se habla de method dispatch dinámico

Corrientes

¿Quién es responsable de conocer los métodos de los objetos?

2 Alternativas conocidas:

- Clasificación
- Prototipado

Clasificación

Clases

- Modelan conceptos abstractos del dominio del problema a resolver
- Se utilizan principios de diseño para decidir cuando crearlas
- Definen el comportamiento y la forma de un conjunto de objetos (sus instancias)
- Todo objeto es instancia de alguna clase

Ejemplo

clase Point

```
Variables de instancia 'xCoord' e 'yCoord'
Métodos
x
   ^xCoord
y
   ^yCoord

dist: aPoint
"Answer the distance between aPoint and the receiver."
   | dx dy |
   dx := aPoint x - xCoord.
   dy := aPoint y - yCoord.
   ^ (dx * dx + (dy * dy)) sqrt
```

Componentes de una clase

Componentes de una clase

- un nombre
- definición de variables de instancia
- métodos de instancia
- por cada método se especifica
  - su nombre
  - parámetros formales
  - cuerpo

Además en Smalltalk

- Las clases son objetos
- definición de variables de clase
- métodos de clase

Ejemplo

clase INode

```
Métodos de clase
l: leftchild r:rightchild
    "Creates an interior node"
    ...

Vars. de instancia 'left right'
Métodos de instancia
sum
    ^ left sum + right sum
```

clase Leaf

```
Métodos de clase
new: anInteger
    "Creates a leaf"
    ...

Vars. de instancia 'value'
Métodos de instancia
sum
    ^value
```

Ejemplos

```
1) Leaf new: 5
2) (INode l: (Leaf new: 3) r: (Leaf new: 4)) sum
```

Self

Pseudo variable que, durante la evaluación de un método, referencia al receptor del mensaje que activó dicha evaluación.

- no puede ser modificada por medio de una asignación.
- se liga automáticamente al receptor cuando comienza la evaluación del método.

clase INode

```
Métodos de clase
l: leftchild r:rightchild
    "Creates an interior node"
    ...

Var. de instancia 'left right'
Métodos de instancia
l
    ^left
r
    ^right
sum
    ^ (self l) sum + (self r) sum
```

Uso común de self

clase Point

```
Métodos de clase
x: xInteger y: yInteger
    "Creates an instance of Point"
    ^self new setX: xInteger setY: yInteger

Variables de instancia 'x' e 'y'
Métodos de instancia
x
    ^x
y
    ^y

setX: xValue setY: yValue
   x :=  xValue.
   y :=  yValue.

dotProduct: aPoint
  ^ (self x * aPoint x) + (self y * aPoint y)
```

¿Quién es self?

```
(Point x: 10 y: 20) dotProduct: (Point x: 1 y: 1)
```

Jerarquía de clases

- Es común que nuevas clases aparezcan como resultado de la extensión de otras existentes incluyendo
  - adición o cambio del comportamiento de uno o varios métodos
  - adición de nuevas variables de instancia o clase
- Una clase puede heredar de o extender una clase pre-existente (la superclase)
- La transitividad de la herencia da origen a las nociones de ancestros y descendientes

Ejemplo

```
Object subclass: #Point
instanceVariableNames: 'x y'
```

Métodos de clase

```
x: p1 y: p2
    ^self new setX: p1 setY: p2
```

Métodos de instancia

```
x
    ^x
y
    ^y

setX: xValue setY:yValue
   x :=  xValue.
   y :=  yValue.
```

```
Point subclass: #ColorPoint
instanceVariableNames: 'color'
```

Métodos de clase

```
x: p1 y: p2 color: aColor
    |instance|
    instance := self x: p1 y: p2.
    instance color: aColor.
    ^instance
```

Métodos de instancia

```
color: aColor
    color := aColor

color
    ^color
```

uso

```
ColorPoint x: 10 y: 20 color: red.
```

Herencia

- Hay dos tipos de herencia
  - Simple: una clase tiene una única clase padre (salvo la clase raíz `object`)
  - Múltiple: una clase puede tener más de una clase padre
- La gran mayoría de los lenguajes OO utilizan herencia simple
- La herencia múltiple complica el proceso de method dispatch

Inconveniente con herencia múltiple

- Supongamos que
  - clases A y B son incomparables y C es subclase de A y B
  - A y B definen (o heredan) dos métodos diferentes para m
  - se envía el mensaje m a una instancia C
- ¿Qué método debe ejecutarse?
- Dos soluciones posibles:
  - Establecer un orden de búsqueda sobre las superclases de una clase
  - Si se heredan dos métodos diferentes para el mismo mensaje debe ser redefinidos en la clase nueva

Method Dispatch Estático

- Method dispatch dinámico es uno de los pilares de la POO (junto con la noción de clase y de herencia)
- Por cuestiones de eficiencia (o diseño, como el caso de C++) muchos lenguajes también cuentan con method dispatch estático
- Sin embargo, hay algunas situaciones donde method dispatch estático es requerido, más allá de cuestiones de eficiencia
- Un ejemplo es el `super`

Method Dispatch Estático

Supongamos que queremos extender la clase `point` del siguiente modo:

```
Object subclass: #Point
Métodos de instancia
setX: xValue setY: yValue
  x:= xValue.
  y:= yValue.

Point subclass: #ColorPoint
Métodos de instancia
setX: xValue setY: yValue setColor: aColor
  x:= xValue.
  y:= yValue.
  color:= aColor.
```

¡`setX: setY: setColor:` duplica código innecesariamente!

Method Dispatch Estático

- Esto es un ejemplo claro de mala práctica de programación en presencia de herencia
- Deberíamos recurrir al código ya existente del método `initialize` de `point` para que se encargue de la inicialización de `x` e `y`

```
Object subclass: #Point
Métodos de instancia
setX: xValue setY: yValue
  x:= xValue.
  y:= yValue.

Point subclass: #ColorPoint
Métodos de instancia
setX: xValue setY:yValue setColor: aColor
   self setX: xValue setY: yValue.
   color:= aColor.
```

Method Dispatch Estático

- ¿La siguiente variante funciona?

```
Object subclass: #Point
Métodos de instancia
setX: xValue setY: yValue
  x:= xValue.
  y:= yValue.

Point subclass: #BluePoint
Métodos de instancia
setX: xValue setY: yValue
   self setX: xValue setY: yValue.
   color:= 'azul'.
```

Method Dispatch Estático

Super

- Pseudovariable que referencia al objeto que recibe el mensaje
- Cambia el proceso de activación al momento del envío de un mensaje.
- Una expresión de la forma "`super msg`" que aparece en el cuerpo de un método `m` provoca que el method lookup se haga desde el padre de la clase anfitriona de `m`

Código corregido

```
Object subclass: #Point
Métodos de instancia
setX: xValue setY: yValue
  x:= xValue.
  y:= yValue.
```

```
Point subclass: #BluePoint
Métodos de instancia
setX: xValue setY: yValue
   super setX: xValue setY: yValue.
   color:= 'azul'.
```

Ejemplo - Super/Self

```
Object subclass: #C1
Métodos de instancia
m1
  ^self m2
m2
  ^13
```

```
C1 subclass: #C2
Métodos de instancia
m1
  ^22
m2
  ^23
m3
  ^super m1
```

```
C2 subclass: #C3
Métodos de instancia
m1
  ^32
m2
  ^33
```

Sigamos la ejecución de

```
(c2 new) m3. ----> Qué valor devuelve?
```

Variante de super en algunos lenguajes

```
super[A] n(...)
```

- Similar super ya visto
- Salvo que la búsqueda comienza desde la clase A
- A debe ser una superclase de la clase anfitriona de m, el método que contiene la expresión super

Lenguajes basados en Objetos

- Caracterizados por la ausencia de clases
- Constructores para la creación de objetos particulares

```
object cell is
   var contents := 0;
   method get() is return self.contents end;
   method set(n) is self.contentes:= n end;
end
```

- Procedimientos para generar objetos

```
procedure newCell(m) is
   object cell is
      var contents := 0;
      method get() is return self.contents end;
      method set(n) is self.contentes:= n end;
   end;
   return cell;
end;
```

Prototipado

- Construye instancias concretas que se interpretan como representantes canónicos de instancias (llamados prototipos)
- Otras instancias se generan por clonación (copia shallow)

```
clonedCell := clone cell
```

- clone opera sobre instancias en lugar de sobre clases
- los clones se pueden cambiar

```
clonedCell.contents := 3

clonedCell.get := method () return self.contents + 1 end;
```

- Clonado y modificación de métodos es una forma limitada de herencia
- Hay otros mecanismos (embedding y delegación)

Cálculo de Objetos (intuición) [Abadi&Cardelli,98]

Syntaxis

$$a ::= x \mid [l_i : \varsigma(x_i)b_i{}^{i\in 1..n}] \mid a.l \mid a.l \leftarrow \sigma(x_i)b_i$$

Semántica operacional

$$\frac{a \to v' \quad b_j[v'/x_j] \to v \quad j \in 1..n}{a.l_j \to v}\,[Sel] \quad con\ v' = [l_i : \varsigma(x_i)b_i{}^{i\in 1..n}]$$

$$\frac{a \to [l_i : \varsigma(x_i)b_i{}^{i\in 1..n}] \quad j \in 1..n}{a.l_j \leftarrow \varsigma(x)b \to [l_j : \varsigma(x)b, l_i : \varsigma(x_i)b_i{}^{i\in 1..n-\{j\}}]}\,[Upd]$$

Ejemplo

$$
\begin{aligned}
\mathit{zero} = [\quad & \\
& \mathit{iszero} = \varsigma(x)\mathit{true}, \\
& \mathit{pred} = \varsigma(x)x, \\
& \mathit{succ} = \varsigma(x)(x.\mathit{iszero} \leftarrow \mathit{false}).\mathit{pred} \leftarrow x]
\end{aligned}
$$

Notas finales

- En este cálculo pueden codificarse las funciones (abstracciones lambda)
- Con las abstracciones lambda pueden escribirse métodos que reciben parámetros
- Se pueden representar clases (usando traits) y herencia.
- Existe la versión tipada

Tipos (y Subtipado) para Lenguajes Orientados a Objetos

9 de noviembre de 2017

Introducción

- Desde mediados de los 80 ha habido numerosos esfuerzos por realizar estudios rigurosos que analizan tipos para LOO
- Dos alternativas han sido exploradas
  - Codificar objetos en términos de lenguajes funcionales
    - Trabajo pionero de Cardelli en 1984
    - Utiliza funciones, registros, recursión y subtipado
  - Formulación de cálculos fundacionales (del estilo de Lambda Cálculo) para el paradigma orientado a objetos como por ejemplo
    - [Abadi y Cardelli, 1996] A Theory of Objects
    - [Castagna, 1997] Object-Oriented Programming: A Unified Foundation
    - [Bruce, 2002] Foundations of Object Oriented Languages

¿Qué es un error de tipos en un LOO?

- Además de los errores de tipos habituales como ser
  - métodos que reciben número o tipo de parámetros incorrectos
  - asignaciones que no respetan el tipo declarado de las variables
- Hay un error de tipos característico que todo sistema de tipos para un LOO debe detectar
  - La invocación a métodos inexistentes
- Los sistemas de tipos actuales para LOO imponen restricciones severas para poder detectar estos errores de tipos

Tipos para LOO

- La noción de clase y de tipo se mantienen separadas
  - Clase: inherentemente de implementación (ej. variables de instancia privadas, código fuente de los métodos, etc.)
  - Tipo de un objeto: la interface pública del mismo
    - nombres de todos los métodos
    - tipo de los argumentos de cada método y tipo del resultado

Especificación de un objeto (tipo) $\neq$ Implementación (clase)

- Esta separación beneficia el desarrollo modular de sistemas: varias clases pueden instanciar objetos con el mismo tipo
- El tipo de un objeto a veces se conoce como interface type

Ejemplo - Clase y su tipo

```
Object subclass: #Point
instanceVariableNames: 'x y'
x
 ...
y
 ...
dist: aPoint
 ...
```

Un objeto instancia de la clase `Point` tendría el siguiente tipo

```
PointType = {
  x: Unit -> Int;
  y: Unit -> Int;
  dist: PointType -> Int;
}
```

OBS: `x`,`y` y `dist` son las componentes del tipo

Tipos a partir de clases

- Como lo muestra el ejemplo podemos extraer de manera mecánica la información necesaria para construir los tipos de objetos a través de las declaraciones de clases
  - Esto aplica a lenguajes tipados estáticamente
- Notación: Si `C` es una clase usaremos `CType` para hacer referencia al tipo de los objetos extraídos de esa clase

Juicio de subtipado

$$\sigma <: \tau$$

- Lectura: "En todo contexto donde se espera una expresión de tipo τ, puede utilizarse una de tipo σ en su lugar sin que ello genere un error"
  - Ej. si `D` es subclase de `C`, entonces se espera que

    $$\texttt{DType} <: \texttt{CType}$$
- ¿Qué relación hay entre $\Gamma \triangleright M : \sigma$ y $\sigma <: \tau$?

Principio de sustitutividad

$$\sigma <: \tau$$

- Lectura: "En todo contexto donde se espera una expresión de tipo τ, puede utilizarse una de tipo σ en su lugar sin que ello genere un error"
- Lectura reflejada en la teoría como una nueva regla de tipado llamada Subsumption:

$$\frac{\Gamma \rhd M : \sigma \quad \sigma <: \tau}{\Gamma \rhd M : \tau} \text{(T-Subs)}$$

- Vamos a recordar el sistema de tipos para el lambda cálculo con registros

Tipado para LC con registros – Repaso

$$\frac{x : \sigma \in \Gamma}{\Gamma \triangleright x : \sigma} \text{(T-Var)}$$

$$\frac{\Gamma, x : \sigma \triangleright M : \tau}{\Gamma \triangleright \lambda x : \sigma . M : \sigma \to \tau} \text{(T-Abs)} \quad \frac{\Gamma \triangleright M : \sigma \to \tau \quad \Gamma \triangleright N : \sigma}{\Gamma \triangleright M\, N : \tau} \text{(T-App)}$$

$$\frac{\Gamma \triangleright M_i : \sigma_i \quad \forall i \in I = \{1..n\}}{\Gamma \triangleright \{l_i = M_i\}_{i \in I} : \{l_i : \sigma_i\}_{i \in I}} \text{(T-Rcd)}$$

$$\frac{\Gamma \triangleright M : \{l_i : \sigma_i \ ^{i \in 1..n}\} \quad j \in 1..n}{\Gamma \triangleright M.l_j : \sigma_j} \text{(T-Proj)}$$

$$\frac{\Gamma \triangleright M : \sigma \quad \sigma <: \tau}{\Gamma \triangleright M : \tau} \text{(T-Subs)}$$

Subtipado de tipos base

- Para los tipos base asumimos que nos informan de qué manera están relacionados; por ejemplo

$$\begin{array}{rcl} Nat & <: & Float \\ Int & <: & Float \\ Bool & <: & Nat \end{array}$$

Subtipado como preorden

$$\frac{}{\sigma <: \sigma}\ \text{(S-Refl)} \qquad \frac{\sigma <: \tau \quad \tau <: \rho}{\sigma <: \rho}\ \text{(S-Trans)}$$

Nota:

- Sin antisimetría

Subtipado de registros a lo "ancho"

`{nombre: String, edad:Int} <: {nombre:String}`

La regla general es

$$\frac{}{\{l_i : \sigma_i | i \in 1..n+k\} <: \{l_i : \sigma_i | i \in 1..n\}} \text{(S-RcdWidth)}$$

Nota:

- $\sigma <: \{\}$, para todo tipo registro σ
- ¿hay algún tipo registro τ tal que $\tau <: \sigma$, para todo tipo registro σ?

Otro ejemplo

```
Object subclass: #Point
instanceVariableNames: 'x y'
x
 ...
y
 ...
dist: aPoint
 ...
```

```
Point subclass: #ColorPoint
instanceVariableNames: 'color'
color
 ...
```

Vemos que `ColorPointType<:PointType` donde

```
PointType = {
  x: Unit -> Int;
  y: Unit -> Int;
  dist: PointType -> Int;
}
```

```
ColorPointType = {
  x: Unit -> Int;
  y: Unit -> Int;
  color: Unit -> Int;
  dist: PointType -> Int;
}
```

Digresión: Tipado Nominal à la Java

- Nuestro enfoque (subtipado estructural):
  - Asociar a cada clase C un registro CType
  - Determinar si CType<:DType en base a la estructura de los registros
- Enfoque de Java (subtipado nominal):
  - Asocia a cada clase C un símbolo #C
  - Declarar como nuevos axiomas de subtipado:

    $$\#C <: \#D$$

    siempre que `class C extends D` aparece en nuestro programa

Limitaciones de subtipado a lo ancho – Shallow clone

- Operación que permite hacer una copia o clon de un objeto
- Especialmente en lenguajes en los que todos los objetos se representan como referencias y la operación de asignación no hace más que copiar referencias
- Shallow cloning (la otra es deep cloning; no será usada en nuestro ejemplo):
  - Copiar los valores de las variables de instancia y tomar el mismo conjunto de métodos que el original
  - Si las variables de instancia tiene referencias a otros objetos sólo las referencias se copian (y no los objetos referenciados)
- Llamaremos `clone` (clase `Object`) a esta operación

Ejemplo de limitación - Shallow clone

```
Object
 clone
  ...
```

```
Cell subclass: #Object
 clone
  ...
```

- ¿Cuál es el tipo de `clone`?
  - En la clase `Object` debe retornar un valor de tipo `ObjectType`
  - Si `Cell` es una subclase de `Object`, uno querría que el método `clone` heredado por `Cell` sea `CellType`
  - ¡En sistemas de tipos invariantes, `clone` debe tener tipo `Object`, aún si el método retorna un valor de tipo `CellType`!

```
ObjectType = {
  clone: Unit -> ObjectType;
}
```

```
CellType = {
  clone: Unit -> ObjectType;
  ...
}
```

- El programador se verá forzado a hacer un type cast para permitir al sistema tratar el valor como si tuviera el tipo debería tener

Ejemplo de limitación - Shallow clone

```
ObjectType = {
  clone: Unit -> ObjectType;
}
```

```
CellType = {
  clone: Unit -> ObjectType;
  m: Unit -> Int;
  ...
}
```

- Si `m` es un método de la clase `Cell` y `o` es una variable de tipo `CellType`, la expresión

  `(o clone()) m()`

  genera un error de tipos
- El programador debe insertar un type cast como en

  `[CellType](o clone()) m()`

  para que funcione como se espera

Typecasts

- Los type casts son una manera de ayudar al sistema de tipos
- Hay dos tipos de typecast
  - "up cast": `[CType]e` donde `e` tiene tipo `DType` y `D` es una subclase de `C`
  - "down cast": `[DType]e` donde `e` tiene tipo `CType` y `D` es una subclase de `C`
- El up cast casi no se usa, mientras que el down cast se usa mucho (para permitir recuperar el tipo "real" de un objeto)

Nota: La necesidad de recurrir a typecasts son una señal de las limitaciones del sistema de tipos

¿Podemos evitar el cast? Subtipado en profundidad

```
Object
clone
 ...
```

```
Cell subclass: #Object
clone
 ...
```

Sería deseable que `CellType<:ObjectType` donde

```
ObjectType = {
  clone: Unit -> ObjectType;
}
```

```
CellType = {
  clone: Unit -> CellType;
}
```

Subtipado de registros en "profundidad"

Si fuese

```
Alumno <: Persona
```

es de esperarse

```
{a: Alumno, d:Int} <: {a:Persona, d:Int}
```

La regla general es

$$\frac{\sigma_i <: \tau_i \quad i \in I = \{1..n\}}{\{l_i : \sigma_i\}_{i \in I} <: \{l_i : \tau_i\}_{i \in I}} \text{(S-RcdDepth)}$$

Ejemplos

$$\{x : \{a : Nat, b : Nat\}, y : \{m : Nat\}\} <: \{x : \{a : Nat\}, y : \{\}\}$$

En efecto:

$$\dfrac{\dfrac{}{\{a : Nat, b : Nat\} <: \{a : Nat\}}\text{(S-RcdWidth)} \qquad \dfrac{}{\{m : Nat\} <: \{\}}\text{(S-RcdWidth)}}{\{x : \{a : Nat, b : Nat\}, y : \{m : Nat\}\} <: \{x : \{a : Nat\}, y : \{\}\}}\text{(S-RcdDepth)}$$

Ejemplos

Utilizando (S-Refl) se puede subtipar sólo un campo:

$$\{x : \{a : Nat, b : Nat\}, y : \{m : Nat\}\} <: \{x : \{a : Nat\}, y : \{m : Nat\}\}$$

En efecto:

$$\dfrac{\dfrac{}{\{a : Nat, b : Nat\} <: \{a : Nat\}}\;\text{(S-RcdWidth)} \qquad \dfrac{}{\{m : Nat\} <: \{m : Nat\}}\;\text{(S-Refl)}}{\{x : \{a : Nat, b : Nat\}, y : \{m : Nat\}\} <: \{x : \{a : Nat\}, y : \{m : Nat\}\}}\;\text{(S-RcdDepth)}$$

Permutaciones de campos

- Los registros son descripciones de campos y no deberían depender del orden dado

$$\frac{\{k_j : \sigma_j | j \in 1..n\} \text{ es permutación de } \{l_i : \tau_i | i \in 1..n\}}{\{k_j : \sigma_j | j \in 1..n\} <: \{l_i : \tau_i | i \in 1..n\}} \text{ (S-RcdPerm)}$$

Nota:

- (S-RcdPerm) puede usarse en combinación con (S-RcdWidth) y (S-Trans) para eliminar campos en cualquier parte del registro

Combinando width, depth y permutation subtyping

$$\frac{\{l_i |\ i \in 1..n\} \subseteq \{k_j |\ j \in 1..m\} \qquad k_j = l_i \Rightarrow \sigma_j <: \tau_i}{\{k_j : \sigma_j |\ j \in 1..m\} <: \{l_i : \tau_i |\ i \in 1..n\}} \text{ (S-Rcd)}$$

Subtipado de tipos función

$$\frac{\sigma' <: \sigma \quad \tau <: \tau'}{\sigma \to \tau <: \sigma' \to \tau'} \text{ (S-Func)}$$

- Observar que el sentido de $<:$ se da "vuelta" para el tipo del argumento de la función pero no para el tipo del resultado
- Se dice que el constructor de tipos función es contravariante en su primer argumento y covariante en el segundo.

Por ejemplo:

$$\mathit{Unit} \to \mathit{CellType} <: \mathit{Unit} \to \mathit{ObjectType}$$

Subtipado de tipos función

$$\frac{\sigma' <: \sigma \quad \tau <: \tau'}{\sigma \to \tau <: \sigma' \to \tau'} \text{ (S-Func)}$$

Si un contexto/programa P espera una expresión f de tipo $\sigma' \to \tau'$ puede recibir otra de tipo $\sigma \to \tau$ si dan las condiciones indicadas

- Toda aplicación de f será a argumentos de tipo σ'
- Los mismos se coercionan a argumentos de tipo σ
- Luego se aplica la función, cuyo tipo real es $\sigma \to \tau$
- Finalmente se coerciona el resultado a τ', el tipo del resultado que P está esperando

Por ejemplo:

$$\mathit{Unit} \to \mathit{CellType} <: \mathit{Unit} \to \mathit{ObjectType}$$

El tipo *Top* – Tipo máximo

Puede verse como representando la clase Object en Smalltalk

$$\frac{}{\sigma <: Top} \text{ (S-Top)}$$

- Notar que $Top \to Top <: Top$

Subtipando colecciones

List ¿es covariante? ¿Es contravariante?

$$\frac{\sigma <: \tau}{\mathit{List}\ \sigma <: \mathit{List}\ \tau}$$

Es covariante (en la mayoría de los lenguajes)

Subtipado de referencias

¿Covariante? Imaginemos esta regla:

$$\frac{\sigma <: \tau}{\mathit{Ref}\ \sigma <: \mathit{Ref}\ \tau}$$

¿Qué ocurre?

Ref no es covariante

```
letval r = ref 3 (*r:Ref int*)
 in
  r := 2.1; (*usando Ref Int <: Ref Float =>T-sub r: Ref Float*)
  !r
end::int
```

¡Pero 2.1 no es int!

$$\frac{\sigma <: \tau}{\mathit{Ref}\,\sigma <: \mathit{Ref}\,\tau} \qquad \frac{\mathit{int} <: \mathit{float}}{\mathit{Ref}\ \ \mathit{int} <: \mathit{Ref}\ \ \mathit{float}}$$

¿Ref contravariante?

¿Contravariante? Imaginemos esta regla:

$$\frac{\sigma <: \tau}{\mathit{Ref}\,\tau <: \mathit{Ref}\,\sigma}$$

Otra vez, ¿qué ocurre?

Ref no es contravariante

```
letval r = ref 2.1 (*r:Ref Float*)
 in
   !r (* por Ref float <: Ref int =>T-sub r: Ref int *)
 end :: int
```

pero 2.1 no es int!!!

$$\frac{\sigma <: \tau}{\mathit{Ref}\,\tau <: \mathit{Ref}\,\sigma} \qquad \frac{\mathit{int} <: \mathit{float}}{\mathit{Ref}\ \ \mathit{float} <: \mathit{Ref}\ \ \mathit{int}}$$

Ref es invariante

$$\frac{\sigma <: \tau \quad \tau <: \sigma}{\mathit{Ref}\,\sigma <: \mathit{Ref}\,\tau}$$

"Sólo se comparan referencias de tipos equivalentes."

Reglas de tipado como especificación de un algoritmo

- Las reglas de tipado sin subtipado son dirigidas por sintaxis.
- Ello hace que sea inmediato implementar un algoritmo de chequeo de tipos a partir de ellas.

$$\frac{x : \sigma \in \Gamma}{\Gamma \rhd x : \sigma}\ (\text{T-Var})$$

$$\frac{\Gamma, x : \sigma \rhd M : \tau}{\Gamma \rhd \lambda x : \sigma . M : \sigma \to \tau}\ (\text{T-Abs}) \quad \frac{\Gamma \rhd M : \sigma \to \tau \quad \Gamma \rhd N : \sigma}{\Gamma \rhd M\, N : \tau}\ (\text{T-App})$$

$$\frac{\Gamma \rhd M_i : \sigma_i \quad \forall i \in I = \{1..n\}}{\Gamma \rhd \{l_i = M_i\}_{i \in I} : \{l_i : \sigma_i\}_{i \in I}}\ (\text{T-Rcd})$$

$$\frac{\Gamma \rhd M : \{l_i : \sigma_i \ ^{i \in 1..n}\} \quad j \in 1..n}{\Gamma \rhd M.l_j : \sigma_j}\ (\text{T-Proj})$$

Agregando subsumption

- Con subsumption ya no son dirigidas por sintaxis.
- No es evidente cómo implementar un algoritmo de chequeo de tipos a partir de las reglas.

$$\frac{x : \sigma \in \Gamma}{\Gamma \rhd x : \sigma}\ (\text{T-Var}) \qquad \frac{\Gamma \rhd M : \sigma \quad \sigma <: \tau}{\Gamma \rhd M : \tau}\ (\text{T-Subs})$$

$$\frac{\Gamma, x : \sigma \rhd M : \tau}{\Gamma \rhd \lambda x : \sigma . M : \sigma \to \tau}\ (\text{T-Abs}) \quad \frac{\Gamma \rhd M : \sigma \to \tau \quad \Gamma \rhd N : \sigma}{\Gamma \rhd M\, N : \tau}\ (\text{T-App})$$

$$\frac{\Gamma \rhd M_i : \sigma_i \quad \forall i \in I = \{1..n\}}{\Gamma \rhd \{l_i = M_i\}_{i \in I} : \{l_i : \sigma_i\}_{i \in I}}\ (\text{T-Rcd})$$

$$\frac{\Gamma \rhd M : \{l_i : \sigma_i\ ^{i \in 1..n}\} \quad j \in 1..n}{\Gamma \rhd M.l_j : \sigma_j}\ (\text{T-Proj})$$

“Cableando” subsumption dentro de las demás reglas

- Un análisis cuidadoso determina que el único lugar donde se precisa subtipar es al aplicar una función a un argumento
- Esto sugiere la siguiente formulación:

$$\frac{x : \sigma \in \Gamma}{\Gamma \mapsto x : \sigma}\ (\text{T-Var}) \qquad \frac{\Gamma, x : \sigma \mapsto M : \tau}{\Gamma \mapsto \lambda x : \sigma.M : \sigma \to \tau}\ (\text{T-Abs})$$

$$\frac{\Gamma \mapsto M : \sigma \to \tau \quad \Gamma \mapsto N : \rho \quad \rho <: \sigma}{\Gamma \mapsto M\, N : \tau}\ (\text{T-App})$$

$$\frac{\Gamma \mapsto M_i : \sigma_i \quad \forall i \in I = \{1..n\}}{\Gamma \mapsto \{l_i = M_i\}_{i \in I} : \{l_i : \sigma_i\}_{i \in I}}\ (\text{T-Rcd})$$

$$\frac{\Gamma \mapsto M : \{l_i : \sigma_i\ ^{i \in 1..n}\} \quad j \in 1..n}{\Gamma \mapsto M.l_j : \sigma_j}\ (\text{T-Proj})$$

Variante dirigida por sintaxis

- Vamos a posponer la discusión de hasta qué punto es más fácil de implementar esta variante
- Antes: ¿Qué relación tiene con la formulación original?

Proposición:

1. $\Gamma \mapsto M : \sigma$ implica que $\Gamma \triangleright M : \sigma$
2. $\Gamma \triangleright M : \sigma$ implica que existe τ tal que $\Gamma \mapsto M : \tau$ con $\tau <: \sigma$

Hacia una implementación de chequeo de tipos

- Lo único que faltaría cubrir es de qué manera se implementa la relación $\sigma <: \tau$

$$\frac{x : \sigma \in \Gamma}{\Gamma \mapsto x : \sigma}\text{(T-Var)} \qquad \frac{\Gamma, x : \sigma \mapsto M : \tau}{\Gamma \mapsto \lambda x : \sigma . M : \sigma \to \tau}\text{(T-Abs)}$$

$$\frac{\Gamma \mapsto M : \sigma \to \tau \quad \Gamma \mapsto N : \rho \quad \rho <: \sigma}{\Gamma \mapsto M\, N : \tau}\text{(T-App)}$$

$$\frac{\Gamma \mapsto M_i : \sigma_i \quad \forall i \in I = \{1..n\}}{\Gamma \mapsto \{l_i = M_i\}_{i \in I} : \{l_i : \sigma_i\}_{i \in I}}\text{(T-Rcd)}$$

$$\frac{\Gamma \mapsto M : \{l_i : \sigma_i{}^{\,i \in 1..n}\} \quad j \in 1..n}{\Gamma \mapsto M.l_j : \sigma_j}\text{(T-Proj)}$$

Reglas de subtipado – Recordatorio

$$\frac{}{\sigma <: \sigma}\ (\text{S-Refl}) \qquad \frac{}{\sigma <: Top}\ (\text{S-Top})$$

$$\frac{}{Nat <: Float}\ (\text{S-NatFloat}) \frac{}{Int <: Float}\ (\text{S-IntFloat}) \frac{}{Bool <: Nat}\ (\text{S-BoolNat})$$

$$\frac{\sigma <: \tau \quad \tau <: \rho}{\sigma <: \rho}\ (\text{S-Trans}) \qquad \frac{\sigma' <: \sigma \quad \tau <: \tau'}{\sigma \to \tau <: \sigma' \to \tau'}\ (\text{S-Func})$$

$$\frac{\{l_i \mid i \in 1..n\} \subseteq \{k_j \mid j \in 1..m\} \qquad k_j = l_i \Rightarrow \sigma_j <: \tau_i}{\{k_j : \sigma_j \mid j \in 1..m\} <: \{l_i : \tau_i \mid i \in 1..n\}}\ (\text{S-Rcd})$$

- No son dirigidas por sintaxis...
- El problema es (S-Refl) y (S-Trans)

Deshaciéndonos de (S-Refl) y (S-Trans)

- Observando que se puede probar $\sigma <: \sigma$ y la transitividad, siempre que se tenga reflexividad para los tipos escalares:
  - $Nat <: Nat$
  - $Int <: Int$
  - $Bool <: Bool$
  - $Float <: Float$
- Agregamos estas cuatro y no consideramos explícitamente a las reglas (S-Refl) y (S-Trans).

El algoritmo de chequeo de subtipos (obviando los axiomas de Nat, Bool, Float)

subtype(S, T) =
  if T==Top
    then true
    else
     if S==$S1 \rightarrow\ S2$ and T==$T1 \rightarrow\ T2$
        then subtype($T1, S1$) and subtype($S2, T2$)
        else
         if S==$\{kj : Sj, j \in\ 1..m\}$ and T==$\{li : Ti, i\ \in\ 1..n\}$
            then $\{li,\ i\ \in\ 1..n\}\ \subseteq\ \{kj,\ j\ \in\ 1..m\}$ and
               $\forall\ i\ \exists\ j\ kj = li$ and subtype(Sj, Ti)
            else false

Lectura adicional

- A Theory of Objects, Martín Abadi, Luca Cardelli, Monographs in Computer Science, Springer-Verlag, 1996.
- Foundations of Object Oriented Languages, Kim Bruce, MIT Press, 2002.
- Some Challenging Typing Issues in Object-Oriented Languages, Kim Bruce. Electronic Notes in Theoretical Computer Science 82, no. 8 (2003). (disponible en su página web).
- On binary methods, Kim Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Objects Group, Gary T. Leavens, and Benjamin Pierce. Theory and Practice of Object Systems, 1(1995).
- Types and Programming Languages, Benjamin C. Pierce, The MIT Press, 2002.