Resumen para el final de Paradigmas de Lenguajes de Programación

Pablo Heiber

14 de diciembre de 2005

Índice

1. Introducción

1.1. Sintaxis

BNF: var := constructor | ... (gramática libre de contexto)

1.2. Semántica

Denotacional: Estructura matremática (conjunto), reglas de transformación de sintaxis a estructura

Intérprete/compilador: Meta-programas

1.3. Paradigmas

Imperativo, funcional, lógico, orientado a objetos, concurrente, dataflow, alegebráico.

- **Imperativo:** Control por iteración, resultados intermedios en variables (memoria)
- **Funcional:** Control por recursión, resultados intermedios como parámetro de otra función
- **Lógico:** Control por recursión, resultados intermedios en la unificación
- **Objetos:** Control por iteración, resultados intermedios en parámetros de mensajes

2. Funcional

Valor: Entidades matemáticas con propiedades abstractas

Expresión: Cadena de caracteres, denota un valor

Transparencia referencial: El valor que denota una expresión solo depende de su contenido
- Se puede demostrar con lógica y modelo semántico
- Sólo se considera el comportamiento externo (abstracción de la ejecución)

Expresiones:
- Atómicas (valores, constantes)
- Compuestas

Bien o mal formadas:
- Reglas sintácticas
- Reglas de asignación de tipos

Funciones:
Visión denotacional: Valor matemático, relaciona conjunto de partida con conjunto de llegada
Visión operacional: Mecanismo que transforma
Operación básica: Aplicación (se denota por yuxtaposición)

Ecuaciones *orientadas*: se usan para *reducir*
$e1 = e2$
Denotacional: e1 y e2 expresan el mismo valor (sin orientación)
Operacional: Para calcular algo que contiene a e1 se lo puede reemplazar por e2 (no al revés, de ahí la orientación)

Programa funcional: Conjunto de ecuaciones
Uso: Reducción de una expresión

Las funciones son valores (resultados, parámetros, almacenables en estructuras de datos, usables como estructuras de datos)

Alto orden: Usar funciones como resultado o parámetro de otra función

Funcional puro: Expresiones con transparencia referencial y alto orden
Modelo de cómputo: Reemplazo de iguales por iguales (reducción)

2.1. Tipos

Tipo: Conjunto de valores con propiedades comunes

Expresión denota *Valor* que pertenece a *Conjunto (tipo)*

Tipado fuerte: Toda expresión válida debe tener tipo

Asignación de tipos:
- `e::A` (fuerza e de tipo A)
- Deducción por reglas (inferencia)

Deseable de un sistema de tipos:
- Automaticidad (algoritmo)
- De tipo a mayor cantidad de expresiones con sentido
- No de tipo a mayor cantidad de expresiones sin sentido
- Conservación por reducción
- Tipos descriptivos y sencillos de leer

Inferencia: Dado e, decidir si $\exists$ A tq e::A y encontrar ese A

Chequeo: Decidir si e::A

Tipos sirven para: Detectar errores, documentar, especificación rudimentaria, detectar oportunidades de optimización (para el compilador), es buena práctica saber el tipo del programa antes de comenzar (especificación)

Polimorfismo: `id x = x` tiene tipo?
Polimorfismo paramétrico: `id x = x` :: A → A es paramétrica
- Es característica del sistema de tipos
- Puede dar *infinitos* tipos a una expresión mediante una generalización

Tipo algebráico: Una forma + un mecanismo de inspección (pattern matching)
```
data sens = frio | calor
data lisint = empty | add int lisint
```

Tipo abstracto (TAD): Operaciones (NO la forma, ocultamiento)

Pattern matching: Mecanismo de acceso
Pattern: Expresión con constructores y variables sin repetir
Matching: Operación que dice si e es de la forma p (pattern)

Tipos algebráicos recursivos: Representan conjuntos inductivos, al menos un constructor debe tener almenos un parámetro de tipo si mismo

Currificación: $f : (a, b) \to c \quad \longrightarrow \quad f : a \to b \to c$
$((a \vee\ b) \Rightarrow c) \Longleftrightarrow (a \Rightarrow b \Rightarrow c)$

2.2. Evaluación

Redex: (sub)expresión que matchea el lado izquierdo de una ecuación

Evaluar: Buscar redex y luego reemplazar por lado derecho

Forma normal: Expresión sin redexes (no se puede reducir)

Reducción:
Normalización: Termina siempre? (no! bottom ($\perp$))
Confluencia: Si termina, es único?
Estrategia: Hay mas de una forma? (si)

No toda expresión tiene forma normal (`infinito = infinito + 1`)

Sin forma normal:
Operacional: Cómputos que no terminan o no están definidos
Denotacional: bottom ($\perp$), valor teórico que lo representa

No se puede manejar bottom operacionalmente ($\perp == \perp$ da $\perp$)
$\perp$ es de tipo A

Orden de reducción:
Aplicativo: Primero parámetros, luego aplicación (primero redex interno)
Normal (lazy): Primero aplicación, luego parámetros (primero redex externo)

Listas: se construyen con `[]` y `:`

Esquemas de recursión: `map`, `foldr`, `filter`
sirven para modular, abstraer, demostrar propiedades comunes

2.3. SFL - simple functional language

2.3.1. Básico (luego se extiende)

Definición inductiva:
- Sintaxis correcta (concreta)
- Representación de las expresiones (abstracta)

Valores expresados: Subconjunto de expresiones a los que se puede llegar reduciendo

Entorno: Estructura de datos, map<variables, valores expresados>

Entorno inicial: Vacio

Interprete: Función recursiva sobre conjunto inductivo de expresiones (sintaxis abstracta)
Salida: Valores expresados

Sintaxis concreta:

```
P::= E
E::= N | id | PR ( E* ) | if E then E else E
PR::= + | - | * | add1 | sub1 | zero?
```

Sintaxis abstracta: Extensión a Haskell de la misma estructura definida con las mismas recursiones

Mapeo sintaxis concreta a abstracta:

Analisis léxico: Separa tokens (lexer)

Analisis sintáctico: Hace árbol de parsing y deja código Haskell (parser)

Valores:

Expresados: Lo que da la evaluación de una expresión (2, false)

Denotados: Se liga a ids (variables)

Valor expresado de `+(3,4)` = 7, valor expresado de `+(x,2)` depende de un Entorno.

¡SFL NO ES LAZY!

2.3.2. Declaraciones locales (let)

Sintaxis concreta:

```
E::= let (id = E)* in E
```

liga ids con una expresión, evalúa el cuerpo con el entorno extendido

Procedimientos, ejemplo (no hay letrec):

```
let faux = proc(next,x)
      if zero?(x) then 1 else next next sub1(x)
      in let fact(x) = (faux faux x) in fact(3)
```

Procedimientos, sintaxis concreta:

```
E::= proc ( id* ) E | E E*      (def | app)
```

Implementación:

let: Extender entorno y evaluar cuerpo (por eso no es lazy)

clausura: Estructura de cuerpo, parámetros y entorno actual (necesario por las vars libres) de un proc

proc: En una definición extender el entorno con la clausura, en una aplicación, buscar la clausura, extender el entorno contenido en ésta con los parámetros y evaluar cuerpo

Una clausura es un valor expresado y denotado

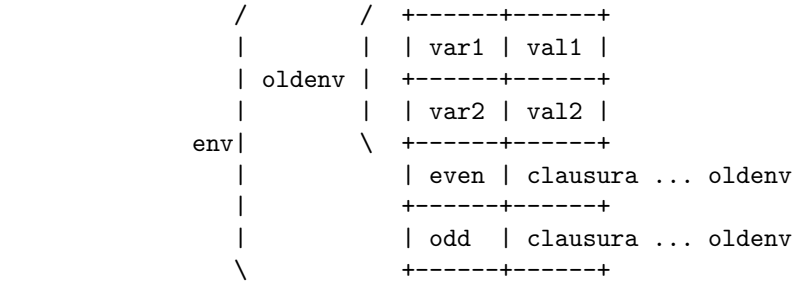
2.3.3. Recursión (letrec)

```
letrec even(x) = if zero?(x) then 1 else odd sub1(x)
letrec odd(x) = if zero?(x) then 0 else even sub1(x)
```

Sintaxis concreta:

```
E::= letrec (id ( id* ) = E)* in E
```

Implementación ingenua:



Las clausuras deben referenciar env y no oldenv, porque en oldenv odd y even no existen!

Idea: Crear la clausura al hacer lookup

v1: Agregar nuevo constructor al entorno que linkea nombre con parámetros y cuerpo. Al hacer lookup tiene el env con las clausuras adentro y todos los datos, asi que construye la clausura con todo bien

v2: Es igual pero viendo al entorno como función de symbol en expval

Idea 2: Estructuras circulares, usar la recursion de Haskell (ver muypseudocodigo)

```
let env = extend names lst rec
     lst = map crearclausura a idss y bodies
```

2.4. PCF - language for programming computable functions

PCF: Lenguaje funcional basado en λ-calculo y teoría de punto fijo

2.4.1. Tipos

Tipos: `nat` | `bool` | $\sigma \to \tau$ | $\sigma \times \tau$

Pseudo-términos: Cosas sintácticamente correctas (pueden estar mal tipadas)

`M::= x` | `true` | `false` | $\underline{n}$ | `if M then P else Q`

`M::= eq? M N` | `M+N` | `<M,N>` | Π_1`(M)` | Π_2`(M)`

`M::=` λ`x:`σ`.M` | `M N` | $\texttt{fix}_\sigma$

Variables libres: Variables no alcanzadas por ningun λ

Sustitución: M{x←N} reemplaza ocurrencias libres de x en M por N (hay que renombrar variables para evitar ligamientos)

α-equivalencia: M $=_\alpha$ N si solo difieren en nombre de variables ligadas

La sustitución es no determinística porque en los renombres se puede elegir cualquier variable, pero bajo α-equivalencia si es determinística.

Sistema de tipado: Sistema formal de deducción o derivación que usa axiomas y reglas para caracterizar un sunconjunto de pseudo-términos llamados tipados

Contexto de tipado: $\Gamma = \{x_1 : \sigma_1, ..., x_n : \sigma_n\}$ $x_i \neq x_j$ $\forall i \neq j$

Juicio de tipado: $\Gamma \triangleright$ M:σ M tiene tipo σ en contexto Γ

Notas de apoyo tipado PCF

2.4.2. Semántica

Semántica axiomática:

Juicio de igualdad: $\Gamma \triangleright$ M = N:σ M = N de tipo σ bajo Γ

Notas de apoyo semántica axiomática PCF

Término: Pseudo-término tipable

Término cerrado: Término sin variables libres

Tipo observable: nat o bool

Funciones: No son observables por ser abstracciones, no se pueden comparar por igualdad

Programa: Término cerrado de un tipo observable

Resultado PCF: Un número, true o false

Semántica operacional:

small-step: Define $\to$ operación de reducción

big-step: Define $\Downarrow$ que reduce a resultado final en un paso. Se puede hacer directamente (natural semantics), o por medio de small-step

small-step:

Juicio de reducción: $\Gamma \triangleright M:\sigma \quad$ M reduce a N en un paso (M $\rightarrow$ N para simplificar)

Subject-reduction: Se prueba que $\Gamma \triangleright M:\sigma \wedge M \rightarrow N \Rightarrow \Gamma \triangleright N:\sigma$

Call-by-name: $\rightarrow$ se puede definir de muchas maneras, vamos a usar una llamada call-by-name para definir $\rightarrow_{cbn}$

Notas de apoyo semántica operacional y slides desde 16.2

$M \rightarrow_{cbn} N$ busca aplicar axiomas en el pseudo término más a la izquierda de M que no esté bajo un λ o un <,>

Forma normal: M está en forma normal si $\neg\exists$ N tq $M \rightarrow_{cbn} N$

Propiedades:
Determinismo: $M \rightarrow_{cbn} P \wedge M \rightarrow_{cbn} Q \Rightarrow P = Q$
Subject-reduction: $\Gamma \triangleright M:\sigma \wedge M \rightarrow N \Rightarrow \Gamma \triangleright N:\sigma$

$\Downarrow$: $\Downarrow M = P \Leftrightarrow ((\exists$ N tq $M \rightarrow_{cbn} N \wedge \Downarrow N = P) \vee M = P) \wedge P$ está en forma normal

$\rightarrow_{cbn}$ permite reducir paso a paso

Nota: En los slides se utiliza una fecha con doble punta en lugar de $\Rightarrow_{cbn}$ para indicar muchos pasos de $\rightarrow_{cbn}$, pero por cuestión de latex-comodidad aqui lo cambiaremos

EVAL: $EVAL(M) =_{def} N$ si $M \Rightarrow_{cbn} N \wedge N$ está en forma normal
- $M = N \Rightarrow M \Rightarrow_{cbn} N$
- $M \Rightarrow_{cbn} P \wedge P \Rightarrow_{cbn} N \Rightarrow M \Rightarrow_{cbn} N$

Un pseudo-término M tiene forma normal $\Leftrightarrow \exists$ N tq EVAL(M) = N

EVAL es parcial, restringida a términos que tengan forma normal

Formas normales cerradas: $\Gamma \triangleright M:\sigma$ y M cerrado y en forma normal $\Rightarrow$ M esta hecho con un constructor (true, false, $\underline{n}$, <M,N> ó $\lambda x:\sigma.M$)

Correctitud:
- $\triangleright$ M:nat y M tiene forma normal $\Rightarrow$ EVAL(M) = $\underline{n}$ para algún n
- $\triangleright$ M:bool y M tiene forma normal $\Rightarrow$ EVAL(M) = true ó false

Equivalencias: Cada forma de definir semántica da una noción de equivalencia
Axiomática ecuacional: $\Gamma \triangleright M:\sigma$ y $\Gamma \triangleright N:\sigma$ equivalentes si $\Gamma \triangleright M = N:\sigma$ es derivable
Operacional: EVAL(M) $\cong$ EVAL(N) ($\cong$: iguales o ambos no definidos)

La equivalencia en el caso axiomático sirve para cualquier término, pero como EVAL solo está definida para programas, necesitamos definir la noción de equivalencia entre términos en dicho caso

Contexto: Término con un agujero notado [_]
Ej. $C[] =_{def} \lambda x:nat.x+[\_]$
Relleno: El agujero se rellena con términos, ej. $C[2] = \lambda x:nat.x+2$, $C[x] = \lambda x:nat.x+x$
esto *puede* ligar variables

$\Gamma \triangleright M:\sigma$ y $\Gamma \triangleright N:\sigma$ equivalentes si $\forall$ C tq C[M] y C[N] son programas EVAL(C[M]) $\cong$ EVAL(C[N]), y lo notaremos $M =_{op} N$

Idea: M y N no tienen forma de diferenciarse, se pueden intercambiar siempre uno por el otro

Distinguish-context: C[] tq $\neg$ (EVAL(C[M]) $\cong$ EVAL(C[N])), ver $M \neq_{op} N$ es encontrar uno, en cambio ver $M =_{op} N$ es realmente difícil

3. Imperativo

Side-effect: Efecto externo causado por una ejecución, opuesto a la transparencia referencial

Recursos imperativos: Asignación, actualización de estructuras de datos, I/O
todas tienen side-effects

3.1. Extensión de SFL

SFL es funcional puro, se le agregará el recurso de asignación

Valores denotados: Referencias a números y procedimientos

Valores expresados: Números y procedimientos

Sintaxis concreta:
`E::= set id = E`

set x = v no devuelve nada, solo ocasiona un side-effect

Formas de pasaje de parámetros:
- Por valor
- Por referencia

Secuenciamiento: Orden explícito de evaluación (si hay side-effects es importante)

Sintaxis concreta para secuenciamiento:
`begin E*` (begin e_1 e_2 ... e_n retorna el valor de e_n luego de ejecutar en orden)

Sintaxis abstracta: Agregar mapeos para set y begin

Memoria: map<referencias, valores expresados>
Ahora el entorno mapea variables a referencias (direcciones de memoria)

Store: Función de referencias en valores expresados

Ahora las funciones toman sus parámetros y el store y devuelven su resultado y el store eventualmente modificado

set x = K con K una constante devolverá siempre 0, lo interesante es la memoria que devuelve

4. Lógica

Programa: Hechos + reglas inferencia + goal a probar (declarativo)

4.1. Lógica proposicional

Forma normal conyuntiva (FNC): $C_1 \wedge C_2 \wedge ... \wedge C_n \quad C_i = B_{i1} \vee B_{i2} \vee ... \vee B_{im} \quad B_{ij}$ literal
conjunción de disyunciones de literales

$\forall A \; \exists A'$ tq $A \Leftrightarrow A'$ es tautología y A' esta en FNC

sin pérdida de generalidad $C_i \neq C_j \; \forall i \neq j \wedge B_{ij} \neq B_{ik} \; \forall j \neq k$

Notación para FNC: Conjunto de conjuntos de literales
$(P \vee Q) \wedge (P \vee \neg Q) \longrightarrow \{\{P, Q\}, \{P, \neg Q\}\}$

4.1.1. Método de resolución

Método de resolución: Refutación, ve si ¬P en FNC es insatisfactible

Se basa en la tautología: $(A \vee P) \wedge (B \vee \neg P) \Leftrightarrow (A \vee P) \wedge (B \vee \neg P) \wedge (A \vee B)$

Regla de resolución:
$\{C_1, C_2, ..., C_n, \{A, P\}, \{B, \neg P\}\} \longrightarrow \{C_1, C_2, ..., C_n, \{A, P\}, \{B, \neg P\}, \{A, B\}\}$

Resolvente: Resolvente de C_1 y C_2 es $C = (C_1 - \{L\}) \cup (C_2 - \{\bar{L}\})$

Regla general: (n y m pueden ser 0)

$$\frac{\{A_1, A_2, ..., A_n, Q\} \quad \{B_1, B_2, ..., B_m, \neg Q\}}{\{A_1, A_2, ..., A_n, B_1, B_2, ..., B_m\}}$$

Método:
- Paso de resolución $S := S \cup \{$ resolvente de cosas de S $\}$
- Si S contiene la cláusula vacía, S es refutación (no se puede satisfacer)

Termina porque hay cantidad finita de literales, o sea, cláusulas finitas (partesDe)

S insatisfactible $\Leftrightarrow$ tiene refutación (método correcto y completo)

4.2. Lógica de primer orden

Lenguaje: Constantes y funciones

Término: Variable, constante o aplicación de función a términos

Fórmula atómica: $P(t_1, t_2, ..., t_n)$, $\doteq(t_1, t_2)$, $\perp$

Fórmula: Fórmulas atómicas conectadas con conectivos lógicos y/o cuantificadores

Fórmula rectificada: $FV(A) \neq BV(A)$ (variables libres y ligadas disjuntas) y cada cuantificador de A tiene una variable distinta

Toda fórmula se puede rectificar renombrando variables y dejar otra α-equivalente

Sentencia: Fórmula cerrada ($FV(A) = \emptyset$)

Toda fórmula es equivalente a su clausura universal (que es una sentencia)

Modelos: Ver en lógica y computabilidad modelos de primer orden, se trata de interpretar cada constante, predicado y función

Teorema de Church: La lógica de primer orden no es decidible (i.e. computable)

Veremos procedimientos de semi-decisión, encuentran refutación si la hay, pero de no haberla pueden colgarse

4.2.1. Resolución ground

Forma normal negada (NNF): No usa $\Rightarrow$ y todos los $\neg$ se encuentran al lado de un término
Toda fórmula es equivalente a una en NNF (mediante reglas se pasan para adentro los $\neg$)

Skolemización: Transformar $\exists x P(x)$ en $P(c)$ donde c es una constante nueva
No preserva validez pero si satistactibilidad

Para skolemizar una NNF, para cada $\exists x B$ reemplazarlo por $B\{x \leftarrow f(x_1, x_2, ..., x_n)\}$
f: nueva función y $x_1, x_2, ..., x_n = FV(B) - \{x\}$ (si es $\emptyset$, $f() = c$ constante)

Alcance universal: $AU(A)$ = conjunto de pares de las subfórmulas de A con sus variables libres cuantificadas universalmente fuera de la subfórmula en cuestión

Forma normal de Skolem: $SK(A)$ es la skolemización de A, no tiene cuantificadores existenciales
$A \rightarrow$ rectificar $\rightarrow$ NNF $\rightarrow$ skolemizar

A satisfactible $\Leftrightarrow$ $SK(A)$ lo es

Instancias compuestas: IC, se aplica sobre una forma de skolem
- $IC(l) = l$ para todo literal l
- $IC(A \star B) = IC(A) \star IC(B)$ para $\star \in \{\wedge, \vee\}$
- $IC(\forall x C) = H_1 \wedge H_2 \wedge ... \wedge H_n$ $\;IC(C\{x \leftarrow t_i\}) = H_i$ para términos arbitrarios t_i

La idea es generar todas las combinaciones para cada $\forall$ dado que el universo es enumerable y la cantidad de $\forall$ es finita. Notar qur las IC son libres de cuantificadores.

Teorema de Skolem-Herbrand-Gödel: A instatisfactible $\Leftrightarrow$ $IC(SK(A))$ insatisfactible (alguna de ellas)

Forma prenexa: Todos los cuantificadores al principio (si esta rectificada no afecta moverlos)

Forma clausal: Conjunción de disyunciones prenexas
$\forall x_1, ..., \forall x_n\ C_1 \wedge \forall y_1, ..., \forall y_m\ C_2 \wedge ... \wedge \forall z_1, ..., \forall z_k\ C_t$

Cláusula ground: No tiene cuantificadores ni variables (se le puede aplicar el método de resolución de lógica proposicional)

$\forall A\ \exists A'$ en forma clausal tal que $A \Leftrightarrow A'$

Cada IC se testea con resolución ground, como es necesario tenerla en forma clausal para esto, mejor pasarla al principio dado que la construcción de IC mantiene esta forma (ver reglas).

$A \rightarrow$ rectificar $\rightarrow$ NNF $\rightarrow$ skolemizar $\rightarrow$ prenexa $\rightarrow$ FNC $\rightarrow$ distribuir $\forall$ en cada cláusula

Las IC son enumerables, asi que en caso de haber refutación termina seguro, pero de todas maneras es muy costoso.

4.2.2. Resolución general

Tenemos una fórmula en forma clausal del mismo modo que en ground.

Sustitución: σ: variables $\rightarrow$ términos $(\#\{x \mid \sigma(x) \neq x\} < \infty)$
Se extiende a $\overline{\sigma}$: términos $\rightarrow$ términos inductivamente (escribimos siempre σ como abuso de notación)

General: σ es mas general que ρ si $\exists \tau$ tq $\rho = \tau \circ \sigma$ (puede no hacer ciertos cambios o hacerlos mas levemente)

Unificador: Un unificador de $t_1, t_2, ..., t_n$ es una sustitución tq $\sigma(t_1) = \sigma(t_2) = ... = \sigma(t_n)$

Unificador mas general (MGU): Un unificador que es mas general que cualquier otro unificador cuando existe, es único el MGU, salvo renombres de variables

Algoritmo de Martelli-Montanari: Para unificar dos términos
- **Descomposición:** $\{\langle f(s_1, .., s_n), f(t_1, ..., t_n)\rangle\} \cup G \mapsto \{\langle s_1, t_1\rangle, ..., \langle s_n, t_n\rangle\} \cup G$
- **Eliminación par trivial:** $\{\langle x, x\rangle\} \cup G \mapsto G$
- **Swap:** $\{\langle t, x\rangle\} \cup G \mapsto \{\langle x, t\rangle\} \cup G$ (para t término y no variable)
- **Eliminación de variable:** $\{\langle x, t\rangle\} \cup G \mapsto_{x/t} G\{x \leftarrow t\}$ si $x \notin FV(t)$

- **Falla:** $\{\langle f(s_1,..,s_n), g(t_1,...,t_m)\rangle\} \cup G \mapsto$ falla si $f \neq g$
- **Occur-check:** $\{\langle x,t\rangle\} \cup G \mapsto$ falla si $x \neq t$ y $x \in FV(t)$

Éxito: Si se llega a vacio, el resultado es componer todos los x/t en eliminación de variable
Falla: Si llega a una falla ($\neg\exists$ MGU)
Es correcto (da un MGU) y completo

Método de resolución general: $S :=$ paso de resolución general(S)

$$\frac{\{B_1, B_2, ..., B_k, A_1, A_2, ..., A_n\} \quad \{\neg D_1, \neg D_2, ..., \neg D_l, C_1, C_2, ..., C_m\}}{\sigma(\{A_1, A_2, ..., A_n, C_1, C_2, ..., C_m\})} \qquad \sigma = MGU(\{B_1, B_2, ..., B_k, D_1, D_2, ..., D_l\})$$

Hay que renombrar las variables de cada conjunto para que sean disjuntas.

El método de reoslución general es correcto y completo. Es mas eficiente que el ground por qué no pierde tiempo generando todas las IC sino que las genera on-demand (al unificar).

Regla binaria:

$$\frac{\{B, A_1, A_2, ..., A_n\} \quad \{\neg D, C_1, C_2, ..., C_m\}}{\sigma(\{A_1, A_2, ..., A_n, C_1, C_2, ..., C_m\})} \qquad \sigma = MGU(\{B, D\})$$

Pierde completitud ($\{\{P(x), P(y)\}, \{\neg P(w), \neg P(z)\}, \{P(u), \neg P(v)\}\}$).
Recupera completud agregando regla de factoreo

$$\frac{\{B_1, B_2, B_m, A_1, A_2, ..., A_n\}}{\sigma(\{B_1, B_2, ..., B_m, A_1, A_2, ..., A_n\})} \qquad \sigma = MGU(\{B_1, B_2, ..., B_m\})$$

De todas maneras, resolución general sigue siendo demasiado caro.

4.2.3. Resolución SLD

La resolución SLD sirve para conjunciones de cláusulas de Horn.

Cláusula de Horn: Disyunción de literalos con a lo sumo un literal positivo
- $\{B, \neg A_1, \neg A_2, \neg A_n\}$
- $\{B\}$
- $\{\neg A_1, \neg A_2, \neg A_n\}$

Las dos primeras se llaman cláusulas de definición y la tercera cláusula goal o negativa.

Prolog: `B:-A1,...,An.` $\longleftrightarrow \{B, \neg A_1, \neg A_2, \neg A_n\}$
`B.` $\longleftrightarrow \{B\}$

Derivación SLD: $\langle N_0, ..., N_p\rangle$ $N_0 := G_i$ (goal)
- $N_i = A_1, ..., A_{k-1}, A_k, A_{k+1}, ..., A_n$
- $N_{i+1} = \sigma(A_1, ..., A_{k-1}, B_1, ..., B_m, A_{k+1}, ..., A_n)$
  donde $\sigma = MGU(\{A_k, A\})$ y $\exists i$ tq $C_i = A :- B_1, ..., B_m$ (m puede ser 0)

A y A_k se llaman átomos

Refutación SLD: Derivación donde N_p es la cláusula vacia
La respuesta SLD es la composición de todos los σ

SLD es correcto (insatisfactible $\Leftrightarrow$ tiene refutación SLD) y completo.

Árbol SLD: Depende de estrategias para elegir A_k y C_i

Programa lógico: $\langle\{C_1, C_2, ..., C_n\}, G\rangle$ C_i cláusulas de definición, G goal

Las cláusulas de definición se separan en hechos (solo un literal positivo) y reglas (literal positivo y al menos uno negativo).

Prolog: Elige átomo mas a la izquierda, hace DFS

Para deducir negativos: CWA (close world assumption)
- **Problema:** El programa puede no terminar, como saber si nunca falla?
- **Rta:** Negation as failure. Solo sobre árboles finitos, menos poderoso pero computable

5. Orientado a objetos

Objeto: Entidad que tiene un estado y puede recibir mensajes

Clase: Molde para objetos que comparten comportamiento
- Nombre
- Padre
- Lista de campos
- Lista de métodos
    - Nombre (`initialize` se llama por defecto cuando es creado)
    - Parámetros
    - Cuerpo

Interacción: Pasaje de mensajes

Method dispatch: Búsqueda y llamado del código a ejecutar
Estático: En tiempo de compilación
Dinámico: En tiempo de ejecución (muchas veces no se puede hacer en otro momento)

Herencia: Promueve reutilización de código
Transitiva: Genera relaciones de orden: ancestro y descendiente
Simple: Cada clase tiene exactamente un padre (excepto Object)
Múltiple: Se puede tener cualquier cantidad de clases padre (trae problemas a la hora de decidir el dispatch si es dinámico)

Jerarquia de clases: Árbol que ilustra las relaciones de herencia entre las clases

Hijos: Agregan o cambian comportamiento y eventualmente campos

C_2 es subclase de $C_1 \Leftrightarrow C_1$ es padre de $C_2 \Leftrightarrow C_2$ subclasea $C_1 \Leftrightarrow C_2$ es hija de C_1

Polimorfismo de subclase: Se puede usar cualquier descendiente, requiere dispatch dinámico

Super: Llamada al método de la clase padre, precisa dispatch estático

Self: Referencia al objeto dentro del cual se esta ejecutando el código actual

5.1. SOOL (extensión de SFL)

Sintaxis concreta:

```
P::= CD* E
CD::= class id extendes id (field id)* MD*
MD::= method id id* E
E::= new id E* | send E id E* | super id E* | begin E ; E* end
PR::= list | cons | nil | car | cdr | null?
```

Sintaxis abstracta: Extrapolar de la concreta a Haskell

Valores expresados: Resultados de evaluar expresiones

Valores denotados: Lo que se puede asignar a una variable

En SOOL:
Valores expresados: Números, ProcVal, Objetos, List(ValExpr)
Valores denotados: Referencias a valores expresados

Intérprete: Entorno, memoria y *declaraciones de clase*
El lookup cuando hay un mensaje debe hacerlo primero en el entorno para hallar el objeto y luego buscar el código a ejecutar en las declaraciones de clase

Objetos: Se representan con lista de partes
Parte: Una clase en la cadena de herencia (nombre, clase, entorno que mapea campos - como referencias al store)

Aplicar método: Hay que bindear super, self, los parámetros y las variables de campo al entorno contenido en la parte correspondiente (como antes con la clausura)

Llamada Super: Igual, solo que empezando lookup en la clase padre a la actual

5.2. Tipos

Clase $\neq$ Tipo: Tipo sólo representa interfaz pública, clase también una implementación

Subtipado: Recordar polimorfismo
Notación: $A <: B \Leftrightarrow A$ es subtipo de B

Subsumption: Sustitutividad, siempre que se puede usar B, se puede usar A

$$\frac{\Gamma \triangleright M : A \quad A <: B}{\Gamma \triangleright M : B}$$

Si C_1 hereda de C_2, C1Type <: C2Type
Para esto hay que restringir la definición de métodos en el hijo, para que no cambie el tipo

Sistemas de tipos invariantes: (Java, C++, Pascal)
- **Método redefinido:** tiene el mismo tipo que el padre (misma visibilidad)
- **Variables de instancia:** No pueden cambiar el tipo
- $\Rightarrow$ Herencia $\subset$ Subtipado

Limitación: Molestas restricciones
- clone() requiere casting
- métodos <, +, etc. molestos de reimplementar (casting también)
- herencia (`Circle.setCenter(Point), ColorCircle.setCenter(ColorPoint)`)

Debilitamiento: Permitir argumentos mas generales y resultado mas específico
Sirve, pero no soluciona (ColorPoint es mas específico, no mas general)

Subtipado covariante: Arrays en Java, Integer[] es subtipo de Object[]
No es correcto, requiere chequeo de las escrituras en runtime (para no poner algo que no es un Integer)

6. Lista de palabras clave

Funcional	Valor	Expresión
Transparencia referencial	Ecuaciones orientadas	Programa funcional
Alto orden	Funcional puro	Tipo
Tipado fuerte	Inferencia de tipos	Chequeo de tipos
Polimorfismo paramétrico	Tipo alegebráico	Tipo abstracto
Pattern matching	Currificación	Redex
Forma normal	Reducción	Normalización
Confluencia	Orden de reducción	Esquemas de recursión
SFL	Entorno	Sintaxis concreta
Sintaxis abstracta	Valores expresados	Valores denotados
Declaraciones locales	Procedimientos	Clausura
Letrec	Estructura circular	λ-cálculo
PCF	Semántica axiomática	Semántica operacional
Pseudo-términos	Variables libres	Sustitución
α-equivalencia	Contexto de tipado	Juicio de tipado
Juicio de igualdad	Término	Término cerrado
Valor observable	Programa	Tipo observable
small-step	big-step	Juicio de reducción
Subject-reduction	Call-by-name	Forma normal
EVAL(M)	Forma normal cerrada	Equivalencia axiomática
Equivalencia operacional	Distinguish context	Side-effects
Pasaje por valor	Pasaje por referencia	Secuenciamiento
Memoria	Programa lógico	Hecho
Regla de inferencia	Cláusula de definición	Goal
FNC	Método de resolución proposicional	Fórmula rectificada
Teorema de Church	NNF	Skolmización
Alcance universal	Instancia compuesta	Teorema Skolem-Herbrand-Gödel
Forma prenexa	Forma clausal	Cláusula ground
Resolución ground	Resolución general	Unificador
MGU	Algortimo Martelli-Montanari	Regla binaria
Regla de factoreo	Resolución SLD	Derivación SLD
Refutación SLD	Cláusula de Horn	Árbol SLD
CWA	Negation-as-failure	Objeto
Clase	Herencia	Campo
Metodo	Pasaje de mensajes	Jerarquía
Herencia simple	Herencia múltiple	Polimorfismo de subclase
Dispatch estático	Dispatch dinámico	Super
Self	SOOL	Declaraciones de clase
Lista de partes	Tipo	Subtipado
Subsumption	Sistemas de tipos invariantes	Subtipado covariante