XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa
Buenos Aires, Setiembre 1988.

UN SISTEMA DE TIPOS PARA UN LENGUAJE FUNCIONAL

DAIQUI SYLVIA
GASPES VERONICA
KESNER DELIA
LAVATELLI CAROLINA

ESLAI
Escuela Superior Latinoamericana de Informática

1 - RESUMEN

Se define un lenguaje de programacion funcional basado en el calculo lambda con un sistema de tipos con polimorfismo. El lenguaje es fuertemente tipado y la asignacion de tipos a sus expresiones se resuelve por inferencia a partir de las declaraciones y el contexto.

Los fundamentos teoricos del trabajo se encuentran en "A theory of type polymorphism in programming ", Milner 77.

Se implemento un precompilador que consiste en un analizador sintactico, un modulo de chequeo e inferencia de tipos y un generador de codigo Lisp puro.

2 - INTRODUCCION

2.1 - ACERCA DE LOS SISTEMAS DE TIPOS EN PROGRAMACION

La nocion de tipo surge informalmente en cualquier dominio. Categorizar objetos de acuerdo a su uso y comportamiento, da lugar a sistemas de tipos mas o menos bien definidos, de modo que los universos no tipados pueden pensarse como tipados.

En matematica y computacion, los tipos imponen restricciones sobre la interaccion entre objetos que previenen que estos interactuen en forma inconsistente con otros objetos.

Las siguientes son algunas caracteristicas que pueden tener los sistemas de tipos de los lenguages de programacion.

Se dice que un lenguage con un sistema de tipos es **estatico**, cuando el tipo de todas las expresiones puede determinarse mediante un analisis del texto. Esto hace que no haya que realizar ningun chequeo de tipos durante la ejecucion del programa. El requerimiento de que los tipos de todas las variables y expresiones este determinado estaticamente es muy restrictivo, y limita el poder de expresion al restringir prematuramente el comportamiento de los objetos.

Estos sistemas excluyen procedimientos y funciones genericos.

Se dice que un lenguage con un sistema de tipos es **fuertemente tipado** si mediante un analisis del texto puede decirse si las expresiones son consistentes, a pesar de que el tipo de las expresiones puede no conocerse. Esta propiedad garantiza las mismas cosas que la anterior y es menos restrictiva.

Dos lenguages que adoptan esta filosofia son ML [ML 85] y Miranda [Tur 86].

A continuacion se intenta caracterizar la idea de polimorfismo y las distintas formas en que se puede presentar.

Una funcion es **monomorfica** si sus argumentos deben tener un unico tipo. Por el contrario, si se permite que una funcion se aplique a valores de distintos tipos esta es **polimorfica**. Por extension si un lenguaje soporta funciones polimorficas es polimorfico.

En principio se podria hablar de dos tipos de polimorfismo :

1- aparente : cuando una funcion se aplica (o parece que se aplica) a argumentos de distintos tipos puede comportarse en forma distinta para cada caso . En general puede pensarse que esa funcion es en realidad un conjunto finito de funciones monomorficas , que ejecuta un codigo diferente para cada tipo. Casi todos los lenguages presentan este tipo de polimorfismo en la forma de sobrecarga de operadores y coercion . Un ejemplo es el operador " + " de PASCAL.

2- universal : cuando una funcion se aplica normalmente sobre un numero infinito de tipos todos de estructura similar. Se puede afirmar que estas funciones realmente tienen muchos tipos , y que se ejecuta el mismo codigo para argumentos de cualquier tipo admisible. Puede resultar util pensar en este tipo de polimorfismo de alguna de las dos formas siguientes, sin que esto imponga una clasificacion

_ parametrico : toda funcion polimorfica tiene parametros (explicitos o implicitos), que determinan el tipo del argumento para cada aplicacion de la funcion (ML fue construido enteramente alrededor de este estilo).

_ por inclusion : un objeto puede verse como perteneciente a muchas clases diferentes no necesariamente disjuntas.

2.2 - INFERENCIA DE TIPOS EN LENGUAJES FUNCIONALES

En este apartado se introduce la metodologia de Milner, mediante algunos ejemplos. El objetivo es conseguir un sistema de tipos flexible y seguro.

Por seguro se entiende que en un programa bien tipado es imposible por ejemplo tratar un entero como una lista, o tratar a true como una funcion. De cualquier forma, el sistema de tipos no puede proteger de todos los errores de tiempo de ejecucion; por ejemplo tomar el primer elemento de una lista vacia o dividir por cero.

La flexibilidad se consigue usando polimorfismo.

El fundamento teorico que garantiza la existencia de un algoritmo capaz de asignar tipos de forma que se consiga la seguridad mencionada incluye los siguientes teoremas:

1-Solidez semantica: las expresiones bien tipadas no pueden andar mal. Esto es, si se las evalua con una asignacion de valores que respeta el contexto entonces el resultado tiene el tipo previsto.

2-Solidez sintactica: si el algoritmo propuesto termina asignando un tipo a una expresion, esta es bien tipada.

Algunas caracteristicas importantes del trabajo de Milner son:

- Todo lo relacionado con tipos es resuelto estaticamente, de modo que pueda ejecutarse el programa sin controlar los tipos.

- El tipo de los terminos puede ser inferido del contexto. No es obligacion del programador declarar el tipo de todos los terminos de su programa. En caso que lo haga se verifica que sea correcto.

- El polimorfismo juega un rol muy importante. El polimorfismo presente en un programa se considera una extension natural de los operadores polimorficos primitivos presentes en todos los lenguages de programacion (aplicacion funcional,formacion de pares ordenados, operadores de procesamiento de listas,etc.)

Los ejemplos se describen en ML ya que este es un lenguaje que sigue la metodologia de tipos propuesta.

Los tipos seran denotados con las letras R, S, T, ...

Se hara distincion entre:

- MONOTIPOS: tipos construidos a partir de un conjunto de tipos basicos (integer, boolean, etc.) por los operadores

  \# (producto cartesiano)

  \+ (union de tipos)

  ---> (tipo de una funcion)

  LIST (lista de objetos de un mismo tipo)

- POLITIPOS: tipos obtenidos admitiendo variables que denotan tipos (VT) en su definicion. A las VT se las representara por A, B,C, ...

EJEMPLO 1: Inferencia del tipo de la funcion map.

```
letrec MAP(f,m) = if null(m) then nil
                  else cons(f(hd(m)),MAP(f,tl(m)))
```

El contexto esta dado por el tipo de las primitivas de manejo de listas

- null : A LIST ---> boolean (dice si una lista es vacia)
- nil : A LIST (lista vacia)
- hd : A LIST ---> A (selector del primer elemento de la lista)
- tl : A LIST ---> A LIST (selector de la cola de la lista)
- cons : (A # A LIST) ---> A LIST (agrega un elemento al comienzo de la lista)

Estos operadores son polimorficos , ya que en sus tipos aparece por lo menos una VT.

Intuitivamente se vera que tipo asignar a MAP.

La definicion de MAP menciona hd(m) y tl(m) de lo que se deduce que m debe ser una lista de algo, es decir,

m : B LIST

y ademas menciona que f se aplica a un elemento de m, de lo que se deduce que f debe ser una funcion, es decir,

f : C ---> D

El resultado de MAP es nil, o se obtiene usando el operador cons, de lo que se deduce que es una lista.

Ahora bien, f se aplica a elementos de m, por lo tanto el dominio de f debe ser del mismo tipo que los elementos de m, con esto se puede decir que el dominio de MAP es

(B ---> D) # B LIST

-------- --------

^el tipo de f ^el tipo de m

Los elementos de la lista que MAP da como resultado, se obtienen aplicando f a elementos de m, por lo tanto tendran el tipo del rango de f, y entonces el tipo de MAP seria:

MAP : ((B ---> D) # B LIST) ---> D LIST

Todo esto se podria formalizar un poco. Se denota con T(id) al tipo del identificador id. De la definicion de Map y del contexto se pueden plantear las siguientes ecuaciones:

T(MAP) = T(f) # T(m) ---> R1

T(null) = T(m) ---> boolean

T(hd) = T(m) ---> R2

T(tl) = T(m) ---> R3

T(f) = R2 ---> R4

T(MAP) = T(f) # R3 ---> R5

T(cons) = R4 # R5 ---> R6

T(nil) = R1 = R6

Cada una de estas ecuaciones surge de algun subtermino de la definicion de MAP, salvo la ultima que surge de que las dos subexpresiones de un condicional tienen el mismo tipo y de que definiens y definiendum deben tener el mismo tipo . El tipo que se asigne a MAP debera satisfacer estas ecuaciones y las del contexto , y puede obtenerse usando el algoritmo de unificacion de Robinson [ROB 65] .

La aparicion de variables en el tipo de un identificador hace que cada ocurrencia del identificador sea tipada con una instancia de substitucion de este tipo generico (sustituyendo las VT por tipos). El tipo asignado al identificador en sus distintas ocurrencias no tiene por que ser el mismo. Un ejemplo de esto podria ser el siguiente:

Si se supone que string es un tipo basico, estructurado como lista de caracteres y que se dispone de la funcion

long : string ---> int

Para el tipo int se dispone de la funcion

sqrt : int ---> real

Entonces, si para tipar el termino

MAP(sqrt,MAP(long,cad:string LIST))

las dos ocurrencias de MAP tendran los tipos

((string --->int) # string LIST) ---> int LIST (la ocurrencia interna)
((int ---> real) # int LIST) ---> real LIST (la ocurrencia externa)

Esto tiene sentido siempre que no se trate de las distintas ocurrencias de un parametro formal o las del identificador que se esta definiendo recursivamente, ya que en estos casos se desea que todas tengan el mismo tipo.

EJEMPLO 2: Interaccion entre el let local y el lambda (una definicion util: variables genericas).

Se desea construir una funcion que produzca cosas de la forma

(b , c) |---> ((a , b) , (a , c))

Esto se puede describir en terminos de las dos funciones siguientes (a pesar de que puede ser descripta mucho mas facilmente de otro modo , pero que no mostraria lo que se pretende)

SUSTI tal que

SUSTI (f , g) (a , b) = (f(a) , g(b))

PAR tal que

PAR(a) (b) = (a , b)

con estas dos, la funcion que se pretende definir puede describirse :

GRAN_PAR ::= lmb x . lmb y z . let tag = PAR(x) in SUSTI(tag , tag)

Los tipos de SUSTI y PAR y el tipo esperado para GRAN_PAR son :

T(SUSTI) = (A ---> B) # (C ---> D) ---> ((A # C) ---> (B # D))

T(PAR) = A ---> (B ---> (A # B))

T(GRAN_PAR) = A' ---> [(B' # C') ---> ((A' # B') # (A' # C'))]

Si en GRAN_PAR se le da a x el tipo A', el tipo local de PAR(x) es D' ---> A' # D' y si se permite que tanto A' como D' sean instanciados en forma diferente en las dos apariciones ligadas de tag, se tendra para GRAN_PAR un tipo mas general que el esperado. El problema se debe a que tanto tag como su tipo generico dependen de la variable x ligada por un lambda (decimos que x esta lmb_ligada) y de su tipo , y no se permitira que distintas ocurrencias de una variable lmb_ligada tengan tipos diferentes. La forma de hacer esto es sencilla ya que se esta hablando del argumento de la funcion; se decreta que al instanciar el tipo de las variables ligadas por let solamente se podra instanciar las variables tipo que no aparezcan en el tipo de variables lmb_ligadas por lambdas que encierren al let . A estas variables instanciables se las llama genericas , y son las que dan el polimorfismo local del identificador. En el ejemplo anterior D' es generica , pero A' no .

La regla que se acaba de introducir hace que dos esquemas de expresiones lambda cuya semantica es la misma esten sometidas a reglas de tipado diferentes . El caso es el siguiente :

- los esquemas

(lmb x.e') e

let x = e in e'

tienen ambas la misma semantica: e' [x /e] . Sin embargo cuando se intenta tipar los siguientes casos particulares de estos esquemas, pueden resultar tipos diferentes:

- las expresiones

let PEPE = lmb x.x in PEPE(PEPE)

(lmb PEPE . PEPE(PEPE)) (lmb x.x)

son casos de los esquemas anteriores y la semantica de ambas es (lmb x.x) (lmb x.x) . En la primera expresion, PEPE no es argumento de una funcion ni es un identificador definido recursivamente, por lo tanto sus dos ocurrencias pueden tiparse con instancias distintas de algun tipo mas general.

En la segunda expresion, PEPE es el argumento de la definicion de una funcion por lo cual las dos ocurrencias deben instanciarse igual. Es por esto que la expresion no puede tiparse pues unifican los tipos de las dos ocurrencias de PEPE.

Poder resolver la interaccion entre variables ligadas por lmb y por let estaticamente es de fundamental importancia : permite tratar la declaracion no global de una funcion que contiene como variable libre un parametro formal de una funcion que la engloba, sin postergar los chequeos .

3 - DEFINICION DEL LENGUAJE

Teniendo en cuenta el objetivo del trabajo, no era esencial dar un conjunto muy amplio de primitivas, sino uno suficiente para poder especificar funcionalmente y permitir declaraciones de tipos.

Una primera aproximacion fue el lenguaje EXP [Mil 77] .

<EXP> ::=X (X denota un identificador valido)
(<EXP> <EXP>)
if <EXP> then <EXP> else <EXP>
lambda X . <EXP>
let X = <EXP> in <EXP>

Esto bastaba para incluir el calculo lambda, pero era muy limitado en su poder de expresion, no tenia ninguna primitiva para el manejo de listas y no permitia DECLARAR tipos explicitamente.

Por ello se definio un lenguaje con sintaxis similar a la de LISP [LSP 84] (que incluye al lenguaje EXP) y con una estructura de tipos que es un subconjunto de la de ML [ML 85]

Este nuevo lenguaje provee las siguientes primitivas:

- operadores aritmeticos:

+, *, /, -

- operadores logicos:

and, not, or

- operadores relacionales:

less, greater, eqnum (operador numerico), eqchar (operador sobre caracteres)

- operadores sobre listas:

cons, null, car, cdr, atom

- operador condicional :

if

todos con la semantica usual.

- operadores sobre producto cartesiano prim, seg, par, donde

prim ([a , b]) = a

seg ([a , b]) = b

par (a , b) = [a , b]

con a y b expresiones validas del lenguaje.

- let con evaluacion simultanea.

- lmb permite definir una funcion anonima, con cantidad arbitraria de parametros. La instanciacion de unode ellos define una funcion con un argumento menos y el argumento es sustituido por la instancia en toda la definicion de la funcion. Por ejemplo, lmb (x y) x+y (no es la sintaxis correcta) instanciado en 8 es la funcion lmb (y) 8+y Se puede instanciar mas de un argumento por vez.

Nota: Solo se proveen primitivas para comparar enteros y caracteres La funcion para comparar objetos de cualquier otro tipo debera ser implementada por el programador.

Con respecto a los tipos, estos podran ser monotipos o politipos donde:

- number, boolean y char son monotipos.
- si A y B son monotipos entonces A > B, A # B y list A tambien son monotipos.
- $x , con x identificador valido, es un politipo (una variable tipo).
- si A es un politipo, list A lo es.
- si A o B son politipos, A > B y A # B lo son.
- si A es tipo y @iden = A entonces @iden es tipo. (abreviatura de un tipo)

Un programa en este lenguaje constara de una declaracion de tipos opcional y de un cuerpo en que se definen funciones. Se da la opcion al programador de que asigne tipo a los argumentos de las funciones, a los de las expresiones lambda y let , y al resultado de las funciones.

3.1 - GRAMATICA

<PROGRAMA> ::= <DECLARACION> <ESPECIFICACION>

<DECLARACION> ::= μ |
type <LISTA_DECLARACION> ;

<LISTA_DECLARACION> ::= <IDEN_TIPO>= <TIPO> |
<IDEN_TIPO>= <TIPO> , <LISTA_DECLARACION>

<TIPO> ::= <TIPO_BASICO> |
<IDEN_TIPO>|
(<TIPO> > <TIPO>)|
(<TIPO> # <TIPO>)|
(list (<TIPO>))

<ESPECIFICACION> ::= <DEF>|
<DEF> <ESPECIFICACION>

<DEF> ::= (defun iden (<ARG_LISTA>) <EXP>) <TIPO_FUN>

<TIPO_FUN> ::= μ |
: <TIPO>

<ARG_LISTA> ::= μ|
iden |
iden : <TIPO> |
iden <ARG_LISTA> |
iden : <TIPO> <ARG_LISTA>

```
<EXP> ::= (if <EXP> <EXP> <EXP>) |          condicional

          (+ <EXP> <EXP> ) |                operadores aritmeticos
          (- <EXP> <EXP> ) |
          (* <EXP> <EXP> ) |
          (/ <EXP> <EXP> ) |

          (eqnum <EXP> <EXP>) |             operadores relacionales
          (eqchar <EXP> <EXP>) |
          (less <EXP> <EXP>) |
          (greater <EXP> <EXP>) |

          (and <EXP> <EXP> ) |              operadores logicos
          (or <EXP> <EXP> ) |
          (not <EXP> ) |

          (atom <EXP>) |                    operadores sobre listas
          (cdr <EXP> ) |
          (car <EXP> ) |
          (null <EXP> ) |
          (cons <EXP> <EXP> ) |

          (prim <EXP> ) |                   operadores sobre producto cartesiano
          (seg <EXP> ) |
          (par <EXP> <EXP> )|

          (let (<LETLIS>) <EXP> ) |

          (iden <LISTA_EXP>) |

          (lambda (<ARG_LAMBDA>) <EXP>)

          '<EXP> |
```

```
        true |                                  constantes
        false|
        num |
        nil |
        iden

<LISTA_EXP>::= <EXP>
               <EXP> <LISTA_EXP>

<ARG_LAMBDA> ::= iden  <COLA_ARG_LISTA>

<LETLIS> ::=  <PAR> |
              <PAR> <LETLIS>

<PAR> ::= ( iden  <EXP> )
```

4 - CHEQUEO E INFERENCIA DE TIPOS

El chequeo e inferencia de tipos debe hacerse sobre un programa sintacticamente correcto. El cuerpo de un programa en este lenguaje puede verse como una lista de definiciones de funciones. De esta forma lo trata el algoritmo de chequeo e inferencia.
Lo que hace dicho algoritmo es determinar el tipo de cada una de las funciones de la lista, ya sea , mediante inferencia y/o respetando la asignacion de tipo del programador y chequeando los tipos para las funciones propias del lenguaje.

El tipo de una expresion se obtiene a partir del tipo de sus subexpresiones. El tipo que se obtiene es el llamado " tipo mas general ", T, tal que para cualquier tipo H que pueda tener esa expresion existe una sustitucion S de variables tipo que cumple S.T=H (H=<T) [Hindley] [Milner].
Finalmente, lo que calcula el algoritmo es una lista de la forma

((nombre-funcion-1 tipo1) . . . (nombre-funcion-k tipok))

Para tipar cada una de las funciones de la lista , introduciremos primero los siguientes conceptos:

Un prefijo tipado P es una secuencia finita cuyos miembros son de la forma 'let x :m' o 'lmb x :m ', donde x es una variable y m es un tipo.
Una expresion prefijada tipada tiene la forma P | E , **donde toda variable libre de la** expresion E aparece en al menos un miembro del prefijo tipado P.

Para la gramatica dada anteriormente las subexpresiones prefijadas tipadas (sub-pes) quedan definidas del siguiente modo:

1- P | x [: tipo] no tiene mas sub-pes que ella misma.

2- P | (e e') tiene sub-pes P | e y P | e'.

3- P | (if b then a else c) tiene sub-pes P | b , P | a , P | c .

4- P | (lmb x1 [:t1] ...xk[:tk] . e) tiene sub-pe P.lmb x1:m1 ... lmb xk :mk | e .

5- P | (let x1 [:t1] = e1 ... xk[:tk] = ek . e) tiene sub-pes P | (e1, ... , ek) y P :-let x1:m1 ...-let xk:mk | e.

6- P | (e1, ... ,ek) tiene sub-pes P | e1 a P | ek.

7- P | (defun F (x1 [:t1], ... , xk[:tk]) . e) [: tipo] tiene sub-pe P | (lmb x1[: t1]... xk[:tk] . e) .

donde m es t si x esta declarada de la forma x : t , y una variable generica si no.
Por ejemplo,

(1) lmb y : number | let f= lmb x . (x y) in (f y)

tiene como sub-pes tipadas

(2) lmb y : number | lmb x . (x y)
(3) lmb y : number . let f : $A | (f y)

donde (2) tiene como sub-pes tipadas a

(4) lmb y : number . lmb x : $B | (x y) ,que a su vez tiene como sub-pes tipadas a

(5) lmb y :number . lmb x : $B | x

(6) lmb y :number . lmb x : $B | y

y donde (3) tiene como sub-pes tipadas a

(7) lmb y : number . let f : $A | f

(8) lmb y : number . let f : $A | y

4.1 - ALGORITMO DE CHEQUEO E INFERENCIA

La forma general de determinar el tipo de una expresion es:
Dada una expresion E a tipar y un prefijo tipado P, inicialmente vacio, lo que se hace es:

* determinar las sub-pes de P|E
* tiparlas
* segun la forma de E y con los tipos de las subexpresiones construir el tipo de E.

Lo que hace el algoritmo que asigna tipos a las funciones de un programa es:
Tipar cada una de las funciones (defun ...), y mantener una lista que contiene el nombre de cada funcion y el tipo correspondiente (el mas general).

Ya que un programa es sintacticamente correcto aunque haya aplicaciones que involucren funciones aun no definidas (el analizador sintactico garantiza que su correspondiente definicion este mas adelante en el texto del programa), con el siguiente ejemplo se puede ver lo que el algoritmo debe hacer.
Sea F la funcion a tipar, cuya definicion es

(defun F (x) (H x))

El calculo del tipo de esta expresion, dara el tipo mas general de F.
Durante el calculo deben considerarse dos casos,

1- Que la funcion H ya este definida , es decir que se conozca su tipo mas general.
Sea r >= q el tipo mas general de H, con r y q politipos o monotipos .
En este caso debe chequearse que el tipo de x sea "menor o igual" que r .
Si es asi, la aplicacion (H x) es correcta. Si no, se levanta un error y el programa de inferencia termina .

2- Que no se conozca aun el tipo mas general de H.
En cuyo caso se infiere el tipo del argumento x, t_x, y se mantiene $t_x > B$ (con B variable generica nueva) junto con los tipos (provisorios) con que fue usada la funcion H . En la determinacion del tipo de F se usa B como tipo de la aplicacion (H x).

Despues de obtener el tipo de F hay dos nuevos casos a considerar:

1- Que F haya aparecido antes en el programa (es decir en alguna otra funcion ocurrio la aplicacion de F a una expresion).
En este caso, cada vez que se la uso se le dio un tipo provisorio que no necesariamente era el mas general, y ahora es el momento de chequear que cada uno de esos sea menor o igual que el mas general (ya obtenido). Si esto no ocurrio en algun caso, es decir, en ese caso se aplico F a argumentos con tipos incompatibles con los inferidos de su definicion, el programa tal como esta no admite asignacion de tipo. Se levanta un error y el programa de inferencia termina.
Si por el contrario se pudo unificar cada tipo provisorio con el mas general, entonces la funcion F se uso correctamente en todos los casos hasta el momento tratados y el programa de inferencia continua con las demas funciones.

2- Que F no haya aparecido antes en el programa . En tal caso el tipo mas general inferido se asocia a F en la lista resultado, para luego comparar con el los tipos que se obtengan al tipar las aplicaciones de F.

El tipado de las expresiones y los chequeos de consistencia mencionados se resuelven basicamente con un algoritmo de unificacion llamado Unify . Este algoritmo tiene como argumentos dos expresiones y calcula la sustitucion que debera aplicarse para unificar dichas expresiones.

Una especificacion del algoritmo seria :

A(P , E) = T, donde

1- si E es x entonces

si lmb x:m pertenece a P , T=m

si let x :m pertenece a P , T=[Bi \ Ci] S(m) , donde Ci son las variables genericas de S(m), Bi variables genericas nuevas y S la sustitucion del ambiente.

si no T= B (variable generica nueva)

2- si E es (d e) , entonces

a1=A (P , d) ;

a2 = A(sust(P) , e), sust es la sustitucion del ambiente.

Si d tiene tipo definido t , K=Unify(((a2 > a1), t)

T= tipo que resulta de aplicar la sustitucion del ambiente a a1

Si no, agregar (a2 > a1) a los tipos posibles de d , y T=a1 .

3- si E es (if b then e1 else e2) , entonces

a1=A (P , b) ; z=Unify(a1,bool) ;

a2=A (P , e1) ; a3=A (P, e2) ;

T=Unify(a2,a3) ;

4- si E es (lmb x1[:t1] ...xk[:tk] . e) , entonces

a1=A(P : lmb x1:m1...lmb xk:mk , e) ;

n = sustitucion aplicada a (m1 # m2 # .... #mk)

T = n > a1

5- si E es (let x1[:t1]=e1 ... xk[:tk]=ek . e) , entonces

a1=A(P , (e1,...,ek)) ;

(n1,...,nk)=Unify(a1,(m1,...,mk)) , donde mi es ti si esta declarado y variable generica si no

T=A(P. let x1:n1...let xk:nk , e)

6- si E es (e1,..,ek) entonces

T=(A(P,e1) , A(P, (e2,...,ek)))

7- si E es (defun F (x1[:t1] ... xk[:tk]) . e) [:tipo-F] entonces

a1=A(P, (lmb x1[:t1] ... xk[:tk] . e));

T= Unify(a1, (B > m)) , donde m es tipo-F si esta declarado, y C (variable generica nueva) si no.

5 - CONCLUSIONES Y TEMAS ABIERTOS

Uno de los objetivos del trabajo era definir un lenguaje funcional con un sistema de tipos. Nuestro lenguaje es bastante sencillo y de limitado poder de expresion, pero esta es una primera aproximacion que constituye el nucleo de futuras extensiones, tanto del sistema de tipos como del conjunto de primitivas, que podrian resultar en lenguajes aptos para programar.

Con una extension del sistema de tipos se podria permitir que se definan nuevos tipos mediante constructores introducidos por el programador. Esta extension requiere de teoria adicional que permita decir en que condiciones se pueden garantizar programas bien tipados

Un tema interesante (abierto) es analizar que parte de esta teoria puede ser util para desarrollar un sistema de tipos con inferencia para algun otro paradigma de programacion.

Ambas propuestas estan siendo desarrolladas en la ESLAI como trabajos de pasantia.

6 - BIBLIOGRAFIA

[Hin 86] Hindley, J. Roger, Seldin, J.P. "An introduction to Combinators and the Lambda Calculus". Cambridge University Press. 1986.

[Lsp 84] Mc Carthy, Abrahamd, Edwards, Hart Levin. Lisp 1.5 Programer's manual. MIT Press.

[Ml 85] Instituto National de Recherche en Informatique et en Automatique. ML Handbook.

[Mil 77] Milner, R. "A theory of Type Polimorfism in Programing " Departament of computer Sciense. University of Edimburg.

[Rob 65] J.A.Robinson, "A machine-oriented logic based en the resolution principle".JACM 12, 1965.

[Tur 86] Turner, D. "An Overview of Miranda". Sigplan Notices, Dic 86.