Instituto de Computación
Facultad de Ingenieria
Universidad de la República
Montevideo - URUGUAY

Msc. Juan Jose Cabezas
Palmas y ombues 5764
Montevideo Uruguay
Telf. 50 29 84

IGUALDAD DEFINICIONAL DE EXPRESIONES MATEMATICAS EN LA LOGICA INTUITIVA

RESUMEN

En este trabajo es presentada una especificación formal de expresiones matematicas e igualdad definicional, basada en la Teoria de Expresiones de Per Martin-Löf.

Per Martin-Löf ha desarrollado una teoria de expresiones donde las definiciones son eliminables y la igualdad entre expresiones es decidible. Dicha teoria se basa en el concepto de aridad (funcionalidad) de expresiones como mecanismo para controlar la formación de expresiones bien-formadas.

En esta monografía, se propone un tipo abstracto de expresiones, se analiza la expansión de expresiones (eliminación de definiciones), y finalmente se expone un procedimiento para decidir igualdad definicional entre expresiones de la misma aridad.

INTRODUCCION

La Teoria de Tipos de Per Martin-Löf puede ser interpretada como una aplicación de la Lógica Intuitiva a la ciencia de la Computación.

Esta teoria presenta un cuerpo unificado para implementar la actividad de especificación, construcción y verificación de programas. Fue originalmente desarrollada como una formalización de la matemática constructiva, con la cualidad que su lenguaje consagra la ocurrencia de demostraciones en las proposiciones, y como consecuencia, las proposiciones pueden expresar propiedades de las demostraciones. Esta facilidad tiene importantes consecuencias para la Teoria de Tipos considerada como un lenguaje de programación.

El lenguaje de la Teoria de Tipos

En la Teoria de Tipos, las expresiones son contruidas a partir de variables y constantes por medio de la aplicación y la abstracción funcional. Aunque el lenguaje, en apariencia, puede presentar semejanzas con el Cálculo Lambda, no es libre de tipos en el sentido de dicho cálculo. Posee más restricciones con respecto a las formas que una expresión bien-formada (wfe) puede tener.

El uso de la noción de "aridad de una expresión" -la cual describe su funcionalidad-, para imponer las restricciones apropiadas sobre las formas de aplicación que son legítimas, ha sido sugerido por Per Martin-Löf.

Dos importantes consecuencias de esta sugerencia concentran la atención de este trabajo:

- las definiciones en una expresión son eliminables
- la igualdad definicional entre expresiones es decidible.

1. EXPRESIONES E IGUALDAD DEFINICIONAL

Este capítulo está basado en Nordström [1].

Como son construidas las expresiones ?

Una manera posible y natural es permitir que las expresiones en general, sean construidas a partir de variables y constantes primitivas por medio de la aplicación y la abstracción funcional. Este es, también, el análisis que han hecho Church, Curry y sus seguidores en Lógica Combinatoria.

Sin embargo, hay dos consecuencias no naturales de esta definición. Una es que se puede aplicar, por ejemplo, la expresión succ, que representa la función sucesor, a tantas expresiones como se desee y formar expresiones como succ(x)(x), aún cuando la función sucesor solo tiene un argumento. La otra es que la autoaplicación está permitida; se puede formar, por ej., succ(succ). La autoaplicación, conjuntamente con la definición

$$((x)b)(a) :=: b[a/x]$$

donde :=: denota igualdad definicional, conduce a expresiones en las que no se pueden eliminar definiciones.

Esto puede constatarse a través del conocido ejemplo

$$((x)x(x))((x)x(x)) :=: ((x)x(x))((x)x(x)) :=: \ldots$$

Según Church [3] sabemos, por lo tanto, que si analizamos la igualdad definicional entre dos expresiones en esta teoria, no será siempre decidible si las expresiones son iguales definicionalmente.

Para resolver estas limitaciones Per Martin-Löf ha sugerido, retornando a Frege [4], que con cada expresión debería asociarse una aridad, describiendo la funcionalidad de la expresión.

1.1 La aridad de una expresión.

Utilizaremos las siguientes convenciones notacionales:

b(a1,..,an) en lugar de b(a1)(a2)... (an)
(x1,..,xn)d en lugar de (x1)(x2)... (xn)d
(w1,..,wn) en lugar de (w1)(w2)... (wn)

Si w1,..,wn (n>Ø) son aridades, entonces (w1,..,wn) es una aridad.
Para n=Ø se obtiene la aridad de una expresión saturada (completa) y se denota () o Ø.
Para n>Ø , (w1,..,wn) es la aridad de expresiones no saturadas

(incompletas, funcionales) las que aplicadas a n expresiones de aridad w1,..,wn , se saturan.
Cada variable y cada constante primitiva (predefinida) tienen una aridad asociada a ella.

A modo de ejemplo aclaratorio se presentan algunas expresiones y sus respectivas aridades.

expresión		aridad
x	-	Ø
1	-	Ø
succ	-	(Ø)
plus	-	(Ø,Ø)
succ(1)	-	Ø
plus(1)	-	(Ø)
plus(succ(1),x)	-	Ø

1.2 Que es una expresión de aridad w ?

Nordström [1] define lo que es una expresión de una cierta aridad; primero se refiere a expresiones saturadas, luego las no saturadas.

1. Si x es una variable de aridad (w1,..,wn) y a1,..,an son expresiones de aridadea w1,..,wn , respectivamente , entonces

x(a1,..,an) es una expresión saturada.

2. Si c es una constante de aridad (w1,..,wn) y a1,..,an son expresiones de aridades w1,..,wn . respectivamente , entonces

c(a1,..,an) es una expresión saturada.

3. Si, en una definición abreviatoria, el definiens (miembro derecho) es una expresión saturada, entonces también lo es el definiendum (miembro izquierdo).

4. Si b(x1,..,xn) es una expresión saturada, donde x1,..,xn son variables de aridades w1,..,wn , respectivamente, que no ocurren libres en b, entonces b es una expresión de aridad (w1,..,wn).

1.3 Reglas de formación de expresiones.

Nordtröm[1] define las reglas de formación de expresiones como sigue:

1. Variables - Una variable de aridad w es una expresión de aridad w.

2. Constantes Primitivas - Una constante primitiva de aridad w es una expresión de aridad w.

3. Constantes Definidas - Si b es una expresión saturada sin otras variables libres que x1,..,xn de aridades w1,..,wn respectivamente, y la constante c de aridad w1,..,wn es definida explicitamente por b, entonces c es una expresión de aridad w.

4. Abstracción - Si v es una expresión de aridad (v1,..,vn) y x1,..,xm son variables de aridad w1,..,wm respectivamente, entonces (x1,..,xm)b es una expresión de aridad (w1,..,wm,v1,..,vn)

5. Aplicación - Si c es una expresión de aridad (w1,..,wm,v1,..,vn) y x1,..,xm son variables de aridades w1,..,wm respectivamente, entonces c(x1,..,xm) es una expresión de aridad (v1,..,vn).

6. Substitución - Es posible substituir variables por expresiones: Si b es una expresión de aridad w, posiblemente conteniendo ocurrencias libres de las variables x1,..,xn de aridades v1,..,vm respectivamente, y a1,..,an son expresiones de estas aridades, entonces b[a1,..,am / x1,..,xm] es una expresión de aridad w.

---

2. UN TIPO ABSTRACTO DE EXPRESIONES

2.1 Sobre el diseño.

El primer paso en la especificación de expresiones es caracterizar un conjunto de funciones que definan la sintaxis abstracta de una expresión arbitraria.

Este conjunto de funciones sigen estrechamente las reglas de formación de expresiones descriptas en [1.3]. Sin embargo, no hay contraparte para la regla 6 -substitución- y la regla 5 se generaliza de forma tal que las expresiones son aceptadas como operandos.

2.1.1 Identificadores y listas · Se asume que los siguientes conjuntos y funciones están disponibles:

a. Id, el conjunto de los identificadores.

```
--  equal   :  Id x Id  --> Bool
```

b. Para cada conjunto A, List(A) es el conjunto de todas las listas con elementos de A. Se dispone, además, del siguiente grupo de funciones para construir y seleccionar listas.

Sea a,b:A l:List(A)

```
--   *      :      --> List(A)
--   .      :  A x  List(A)   --> List(A)

-- empty    :  List(A)  --> Bool
-- first    :  List(A)  --> A
-- rest     :  List(A)  --> List(A)

   empty(*)=true
   empty(a.l)=false

   first(a.l)=a

   rest(*)=*
   rest(a.l)=l
```

2.2 Constructores

De acuerdo con las reglas de formación de expresiones [1.3], podemos definir las siguientes funciones para construie expresiones a partir de variable y constantes.

Sea e:Exp, lx:List(Var), le:List(Exp).

```
-- cr_var        :  Id        -->    Var

-- cr_cont       :  Id        -->   Const

-- cr_abs        :  List(var)  x Exp  --> Exp

   cr_abs(lx,e) construye (lx)e

-- cr_appl       :  Exp  x  List(Exp)  -->  Exp

   cr_appl(e,le) construye e(le)
```

donde Var y Const son subconjuntos del conjunto Exp:

Exp - El conjunto de las expresiones.
Var - El subconjunto de las expresiones que son variables
Const - El subconjunto de las expresiones que son constantes

2.3 Selectores

Sea i:Id, e,a:Exp, le:List(Exp), lx:List(Var).

```
-- is_var        :   Exp   -->   Bool
   is_var(cr_var(i))=true
   para los demás casos se cumple is_var(a)=false

-- is_const      :   Exp   -->   Bool
   is_const(cr_const(i))=true
   para los demás casos se cumple is_const(a)=false

-- is_abs        :   Exp   -->   Bool
   is_abs(cr_abs(lx,e))=true
   para los demás casos se cumple is_abs(a)=false


-- is_appl       :   Exp   -->   Bool
   is_appl(cr_appl(e,le)=true
   para los demás casos se cumple is_appl(a)=false
```

Los selectores is_var, is_const, is_abs e is_appl cumplen con la propiedad de que dada una expresión arbitraria, uno y solo uno de ellos será verdadero:

(is var(e) XOR is const(e)) XOR (is abs(e) XOR is appl(e))=true

```
   indent        :   Exp  -->  Id
   indent(cr_var(i))=i
   indent(cr_const(i))=i
```

```
operator      :   Exp  --->  Exp
operator(cr_appl(e,le))=e

ops           :   Exp  --->  List(Exp)
ops(cr_appl(e,le))=le

expr          :   Exp  --->  Exp
expr(cr_abs(lx,e))=e

vars          :   Exp  --->  List(Var)
vars(cr_abs(lx,e))=lx
```

Se aclara que las funciones indent, operator, ops, expr y vars están indefinidas para ciertos argumentos.

---

3. EXPANSION DE EXPRESIONES

Nosotros hemos visto en [1.3] que si c es una constante definida de aridad (w1,..,wn), ella está definida explicitamente por una expresión saturada b sin otras ocurrencias de variables libres que no sean x1,..,xn de aridades w1,..,wn respectivamente.

Este capítulo está dirigido a demostrar que si e es una expresión saturada de la forma c(e1,..,en) y c es una constante definida, entonces c puede siempre ser eliminada de e. El proceso de eliminar una constante definida de una expresión es llamado expansión de expresiones.

3.1 Reglas de igualdad definicional

1. La igualdad definicional es una relación de equivalencia:

$$a :=: a \quad ; \quad \frac{a :=: b}{b :=: a} \quad ; \quad \frac{a :=: b \quad b :=: c}{a :=: c}$$

$$\frac{a :=: b}{(x)a :=: (x)b}$$

2. Si a y c son expresiones de aridad w y b una expresión de aridad (w) entonces

$$\frac{a :=: c}{b(a) :=: b(c)}$$

3. Beta-conversion

$$((x)b)(a) :=: b[a/x]$$

Las variables libres en a no pueden aparecer ligadas en b[a/x]

5.

$$(x)(b(x)) :=: b$$

x no debe ocurrir libre en b

6. Constantes definidas - Si c es una constante de aridad (w1,..,wn) y b es una expresión saturada (aridad=Ø) sin otras ocurrencias de variables libres que no sean x1,..,xn de aridades w1,..,wn respectivamente, entonces se puede plantear la definición explícita

c(x1,..,xn) :=: b

3.2 Expansión de una expresión

Supongamos que tenemos una expresión de la forma

(i) c(e1,..,en) · Ø

donde c es una constante definida de aridad (w1,..,wn), c(x1,..,xn) es un definiendum con definiens b, y e1,..,en son expresiones de aridades w1,..,wn, respectivamente. Con el soporte de [3.1] las siguientes iguadades son válidas

c(x1,..,xn) :=: b Reglas 2 y 6 [3.1]

(x1,..,xn)(c(x1,..,xn)) :=: (x1,..,xn)b Regla 5 [3.1]

(ii) c :=: (x1,..,xn)b

Ahora es posible eliminar c de (i) con ayuda de la igualdad (ii)

c(e1,..,en) :=: ((x1,..,xn)b)(e1,..,en) Regla 4 [3.1]

y por lo tanto

((x1,..,xn)b)(e1,..,en) :=: b[e1,..,en / x1,..,xn]

concluyendo

c(e1,..,en) :=: b[e1,..,en / x1,..,xn]

Se dice, entonces, que la expresión c(e1,..,en) ha sido expandida a una equivalente de la forma b[e1,..,en / x1,..,xn].

3.3 La lista de definiciones.

Cuando se ha de expandir una expresión debe ser posible el acceso a todas la constantes definidas y sus definiciones. Por esta razón, se supone disponible una lista de definiciones.

Se define el tipo de las definiciones, Def :

Sea i:Id, e:Exp, lx:List(Var).

```
--      cr_def      :  Id  x  List(Var)  x  Exp  ->  Def

---     def_id      :  Def  ->  Id
                    - def_id(cr_def(i,lx,e))= i

--      def_vars    :  Def  ->  List(var)
                    - def_vars(cr_def(i,lx,e))= lx

--      def_exp     :  Def  ->  Exp
                    - def_exp(cr_def(i,lx,e))= e
```

La lista de definiciones será accedida por las siguientes operaciones:

Sea c:Const, b:Exp, x1,..,xn:Var, d:Def, defs:List(Def).

```
is_def_const        :  Const  x  List(Def)  ->  Bool
                    -- is_def_const(c,defs) = true
                       si existe una definicion d en defs tal que
                       equal(ident(c),def_id(d)) = true


fetch_def           :  Const  x  List(Def)  ->  Def
                    - Si is_def_const(c,defs) = true
                       entonces fetch_def(c,defs) = d
                       y se cumple que
                       equal(ident(c),def_id(d)) = true
```

3.4 Eliminación de una constante definida en una expresión.

Si tenemos una expresión de la forma

c(e1,..,en)

donde c es una constante definida, c(x1,..,xn) es un definiendum con definiens b, y e1,..,en son expresiones, entonces se puede expandir dicha expresión de la manera expuesta en [3.2]:

c(e1,..,en) :=: b[e1,..,en / x1,..,xn]

Esta igualdad indica que la regla 6 de formación de expresiones -substitución- debe ser aplicada cuando se ha de expandir una expresión.

3.4.1 Substitución - Con la intención de definir una función que represente la regla 6 [1.3] capaz de evitar colisiones entre los nombres de variables libres y ligadas, se ha de seguir la estrategia

de avanzar paso a paso, comenzando por definir una función para un caso simple y restringido de substitución.

Paso I .- Si e es una expresión de aridad w, que posiblemente contiene ocurrencias libres de la variable x de aridad v, y es una variable de aridad v que no ocurre ligada en e, y si e', x', y' son representaciones de e, x e y, entonces simple subs(x',e',y') puede ser considerada como una representación de e[x/y].

simple_subs : Var x Exp x Var -> Exp

Paso II .- Si e es una expresión de aridad w, que posiblemente contiene ocurrencias libres de las variables x1,..,xn de aridad v1,..,vn, y1,..,yn son variables de estas aridades que no ocurren ligadas en e y ident(xi)<>ident(yk) para todo Ø<i<n+1, y si x1, e',y1 son representaciones de (x1,..,xn), e y (y1,..,yn), entonces vars_subs(x1,e',y1) puede ser considerada como una representación de e[y1,..,yn / x1,..,xn].

vars_subs : List(Var) x Exp x List(var) -> Exp

Paso III .- Si e es una expresión de aridad w, que posiblemente contiene ocurrencias libres de las variables x1,..,xn de aridades v1,..,vn, a1,..,an son expresiones de dichas aridades tales que ningún miembro de (x1,..,xn) ocurre libre en algún miembro de (a1,..,an), entonces exps_subs(x1,e',a1) puede ser considerada como una representación de e[a1,..,an / x1,..,xn].

exps_subs : List(Var) x Exp x List(Exp) -> Exp

Con el fin de prevenir colisiones entre los nombres de las variables, exps_subs puede substituir los nombres de variables ligadas por otros no utilizados previamente. Para crear estos nombres se supone disponible la operación new_names:

new_names : Integer -> List(Var)
- new_names(n) produce una lista de
n variables cuyos nombres no han sido
previamente utilizados.

Paso IV .- Finalmente, la función subst, que representa la regla ó [1.3], puede ser definida. Si e es una expresión de aridad w, que posiblemente contiene ocurrencias libres de las variables x1,..,xn de aridades v1,..,vn y a1,..,an son expresiones de estas aridades entonces

Sea e:Exp, x1,n1:List(Var), e1:List(Exp).

subst : List(var) x Exp x List(Exp) -> Exp

- subst(x1,e,e1) = aux_subst(x1,e,e1,new_names(length(x1)))

```
where
aux_subst(x1,e,e1,n1)=
      exps_subs(n1,vars_subs(x1,e,n1),e1)
```

3.4.2 Expand .- Con la función subst definida es posible presentar una definición de la función expand, que representa el proceso de expandir una expresión.

Sea c:Const una constante definida, e1:List(Exp), defs:List(Def).

```
    expand     :  Exp  x  List(Def)  -> Exp

  - expand(cr_appl(c,e1),defs) = subst( def_vars(fetch_def(c,defs)),
                                        def_exp(fetch_def(c,defs)),
                                        e1)
```

4. IGUALDAD DEFINICIONAL

4.1 Cuando son iguales dos expresiones de aridad w ?

Las siguientes reglas, según Nordström[1], hacen posible decidir cuando dos expresiones arbitrarias de la misma aridad son iguales. Utilizaremos la notación b - w por "b es una expresion de aridad w", y a :=: b - w por "a y b son expresiones iguales de aridad w".

1. Si x es una variable de aridad (w1,..,wn) y
   a1 :=: b1 - w1
   a2 :=: b2 - w2
   .. .. ..
   an :=: bn - wn
   entonces x(a1,..,an) :=: x(b1,..,bn) - Ø

2. Si x es una constante de aridad (w1,..,wn) y
   a1 :=: b1 - w1
   a2 :=: b2 - w2
   .. .. ..
   an :=: bn - wn
   entonces x(a1,..,an) :=: x(b1,..,bn) - Ø

3. Si a es un definiendum (miembro izquierdo) con definiens (miembro derecho) b, y b :=: c - Ø entonces a :=: c - Ø. Del mismo modo si a :=: b - Ø y b es el definiens de un definiendum c, entonces a :=: c - Ø.

4. Si b(x1,..,xn) :=: d(x1,..,xn) - Ø donde x1,..,xn son variables de aridades w1,..,wn, respectivamente, las cuales no ocurren libres en b o d entonces b :=: d - Ø.

Las reglas 1, 2 y 3 son definidas para expresiones saturadas (de aridad Ø). Como solamente variables, constantes primitivas y aplicaciones pueden ser expresiones saturadas, la regla 4 hace posible la decidibilidad de la igualdad definicional de expresiones no saturadas (de aridad (w1,..,wn)) aplicando las reglas 1, 2 y 3 de la siguiente manera:

Para decidir si las expresiones b y d de aridad (w1,..,wn) n>Ø son iguales, se aplican las reglas 1,2 y 3 a b(x1,..,xn) y d(x1,..,xn). Si b(x1,..,xn) :=: d(x1,..,xn) entonces b :=: d.

4.2 Forma reducida de expresiones.

Una expresión está en forma reducida si:

- es una variable o una constante primitiva
- es una aplicación a(e1,..,en) y el operador a es una variable o una constante primitiva
- es una abstracción (x1,..,xn)b y b no es una abstracción.

Las expresiones son preferidas en esta forma ya que, aplicando las reglas presentadas en [4.1], puede decidirse la igualdad definicional de dos expresiones, en cualquiera de los seis casos en que se presentan:

Caso 1.- Ambas expresiones son variables o contantes primitivas.

Caso 2.- Una expresión es una variable o una constante primitiva y la otra es una aplicación.

Caso 3.- Una expresión es una variable o una constante primitiva y lae otra es una abstracción.

Caso 4.- Ambas expresiones son aplicaciones.

Caso 5.- Una expresión es una aplicación y la otra es una abstracción.

Caso 6.- Ambas expresiones son abstracciones.

4.3 Un método para reducir una expresión.

Teorema : Existe un método para tranformar una expresión e de aridad w a una igual definicionalmente en forma reducida.

Definición I, Lema I y Lema II serán utilizados para obtener la forma reducida de expresiones:

Definicion I.

(a(e1,..,en))(en+1,..,em) :=: a(e1,..,en,en+1,..,em)

La función red_appl representa la definicion I:

Sea e,a:Exp, el1,el2:List(Exp).

```
red_appl     :     Exp    ->   Exp
             - Si e= cr_appl(cr_appl((a,el1),el2)
               entonces red_appl((e) = cr_appl(a,concat(el1,el2))
```

Lema I. - Regla 5 [3.1] y la Definición I sustentan el siguiente lema:

(x1,..,xn)((xn+1,..,xm)b) :=: (x1,..,xn,xn+1,..,xm)b

Prueba: las siguientes expresiones son definicionalmente iguales.

(x1,..,xn)((xn+1,..,xm)b)
Regla 5 [3.1]
(x1,..,xm)(((x1,..,xn)((xn+1,..,xm)b)) (x1,..,xm))
Definición I
(x1,..,xm)((((((x1,..,xn)((xn+1,..,xm)b))(x1,..,xn)))(xn+1,..,xm)))
Regla 5 [3.1]
(x1,..,xm)(((xn+1,..,xm)b)(xn+1,..,xm))
Regla 5 [3.1]
(x1,..,xm)b

La función red_abs representa el Lema I:

Sea e,a:Exp, xl1,xl2:List(Exp).

red_abs : Exp -> Exp
- Si e= cr_abs(xl1,cr abs(xl2,a))
entonces red_abs((e) = cr_abs(concat(xl1,xl2),a)

Lema II.- Regla 4 [3.1] (Beta-conversion), Definición I y Lema I soportan el siguiente lema:

n=k, ((x1,..,xn)b)(e1,..,ek) :=: b[e1,..,ek/x1,..,xn]
n<k, ((x1,..,xn)b)(e1,..,ek) :=: (b[e1,..,en/x1,..,xn])(en+1,..,ek)
n>k, ((x1,..,xn)b)(e1,..,ek) :=: ((xk+1,..,xn)b)[e1,..,ek/x1,..,xk]

La función beta representa el Lema II:

Sea b,e:Exp, xl,xl1,xl2:List(Var), el,el1,el2:List(Exp).

beta : Exp -> Exp
- Si e = cr_appl(cr_abs(xl,b),el)
entonces
si length(xl)=length(el)
entonces beta(e) = subst(xl,b,el)
si length(xl) < length(el) y existen
el1,el2 tales que el=concat(el1,el2) y
length(xl) = length(el1)
entonces beta(e) = cr_appl(subst(xl,b,el1),el2)
si length(xl) > length(el) y existen
xl1,xl2 tales que xl=concat(xl1,xl2) y
length(xl1) = length(el)
entonces beta(e) = (subst(xl1,cr_abs(xl2,b),el)

El siguiente procedimiento (Tabla 4.1) describe un método para

transformar una expresión e de aridad w en una definicionalmente igual en forma reducida.

Basado en este procedimiento, se define la función red_form que transforma una expresión en una equivalente en forma reducida, luego de un número finito de pasos.

Si e es	red_form(e,defs) =
1. - una variable o una constante primitiva.	e
Una aplicación cuyo operador puede ser:	
2.a- una variable o una constante primitiva.	e
2.b- una constante definida.	red_form(expand(e,defs),defs)
2.c- una aplicación.	red_form(red_appl(e),defs)
2.d- una abstracción.	red_form(beta(e),defs)
Una abstracción cuya expresión puede ser:	
3.a- una abstracción.	red_form(red_abs(e),defs)
3.b- una variable o una constante o una aplicación.	e

TABLA 4.1 - Una descripción de red_form.

4.4 Un algoritmo para igualdad definicional.

Sea e1 y e2 dos expresiones arbitrarias de la misma aridad. Para decidir si e1 y e2 son definicionalmente iguales, se transforman e1 y e2 a la forma reducida, y luego se aplican las reglas de igualdad [4.1] en los seis casos posibles que dos expresiones en forma reducida presentan:

Caso 1 - Si e1 y e2 son variables o constantes primitivas de aridad (w1,..,wn), entonces e1 y e2 son definicionalmente iguales si

$$e1(x1,..,xn) :=: e2(x1,..,xn) - \emptyset \qquad \text{Regla 4 [4.1]}$$

y esta igualdad se cumple si Regla 1 2 [4.1]

e1 y e2 son ambas variables y e1 = e2

o
e1 y e2 son ambas constantes primitivas y e1 = e2

donde x1,..,xn son variables de aridad w1,..,wn respectivamente, que no ocurren libres en e1 y e2.

Caso 2 - Si una de las expresiones es una aplicación b(a1,..,an) de aridad (w1,..,wk), y la otra es una variable o una constante primitiva d, entonces e1 y e2 son definicionalmente iguales si

e1(x1,..,xk) :=: e2(x1,..,xk) - Ø Regla 4 [4.1]

donde x1,..,xk son variables de aridad w1,..,wk respectivamente, que no ocurren libres en e1 y e2.

Pero
(b(a1,..,an))(x1,..,xk) :=: b(a1,..,an,x1,..,xk) Definición I
y
b(a1,..,an,x1,..,xk) y d(x1,..,xk) son definicionalmente iguales si y solo si n+k=k (n=Ø) y b y d son definicionalmente iguales (Reglas 1 y 2 [4.1]). Pero n=Ø no es válido ya que una aplicación tiene al menos un operando, n>Ø.
Conclusión: e1 y e2 no son definicionalmente iguales.

Caso 3 - Si una de las expresiones es una abstracción (x1,..,xn)b de aridad (w1,..,wk), y la otra es una variable o una constante primitiva de la misma aridad, entonces e1 y e2 son definicionalmente iguales si

e1(y1,..,yk) :=: e2(y1,..,yk) - Ø Regla 4 [4.1]
donde y1,..,yk son variables de aridad w1,..,wk respectivamente, que no ocurren libres en e1 y e2.

Pero
e1(y1,..,yk) :=: (e1(y1,..,yn))(yn+1,..,yk) Definición I
e1(y1,..,yk) :=: (e1(y1,..,yn))(yn+1,..,yk) Definición I
y si
e1(y1,..,yn) :=: e2(y1,..,yn) - (wn+1,..,wk) Regla 3 [3.1]
entonces
(e1(y1,..,yn))(yn+1,..,yk) :=: (e2(y1,..,yn))(yn+1,..,yk)
e1(y1,..,yk) :=: e2(y1,..,yk) - Ø
e1 :=: e2 (w1,..,wk)

Conclusión: e1 y e2 son definicionalmente iguales si e1(y1,..,yk) y e2(y1,..,yk) son definicionalmente iguales.

Caso 4 - Si e1:=:d(a1,..,am) y e2:=:b(g1,..,gn) son ambas aplicaciones de aridad (w1,..,wk), entonces e1 y e2 son definicionalmente iguales si m=n y si

e1(y1,..,yk) :=: e2(y1,..,yk) - Ø Regla 4 [4.1]
donde y1,..,yk son variables de aridad w1,..,wk respectivamente, que

no ocurren libres en e1 o e2.

Pero

e1(y1,..,yk) :=: d(a1,..,an,y1,..,yk) Definición I
e2(y1,..,yk) :=: b(a1,..,an,y1,..,yk) Definición I

y

si d :=: b Regla 1 o 2 [4.1]
    a1 :=: g1
    .. ..
    an :=: gn
    y1 :=: y1
    .. ..
    yk :=: yk
entonces

d(a1,..,an,y1,..,yk) :=: b(g1,..,gn,y1,..,yk) - Ø Regla 3 [4.1]

y

e1(y1,..,yk) :=: e2(y1,..,yk) - Ø

y

e1 :=: e2 - (w1,..,wk)

Conclusión: e1 y e2 son definicionalmente iguales si m=n y si
a1 y b1 son definicionalmente iguales y
a2 y b2 son definicionalmente iguales y
.. ..
am y bn son definicionalmente iguales.

Caso 5 - Si una de las expresiones es una abstracción (x1,..,xn)d de aridad (w1,..,wk), y la otra es una aplicación, entonces e1 y e2 son definicionalmente iguales si e1(y1,..,yk) e2(y1,..,yk) son definicionalmente iguales (procedimiento similar al caso 3).
y1,..,yk son variables de aridad w1,..,wk respectivamente, que no ocurren libres en e1 o e2.

Caso 6 - si e1:=:(x1,..,xn)d y e2:=:(y1,..,yr)b son ambas abstracciones de aridad (w1,..,wk) entonces e1 y e2 son definicionalmente iguales si e1(z1,..,zm) y e2(z1,..,zm) lo son (procedimiento similar al caso 3).
k>=n,k>=r y m=max(n,r)
z1,..,zm son variables de aridad w1,..,wm respectivamente, que no ocurren libres en e1 o e2.

Observe que no es necesario conocer la aridad de las expresiones para decidir la igualdad definicional. Solamente se supone que las expresiones son de la misma aridad.

La función def equal.

La función def_equal, basada en este algoritmo, es definida para decidir si dos expresiones de la misma aridad son definicionalmente iguales:

Sea e1,e2:Exp, defs:List(Def)

```
def_equal   :   Exp  x  Exp  x  List(Def)  ->  Bool
            -  def_equal(red_form(e1), red_form(e2),defs)=true
                    si e1 y e2 son definicionalmente iguales
               def_equal(red_form(e1), red_form(e2),defs)=false
                    si e1 y e2 no son definicionalmente iguales
```

==========

REFERENCIAS

[1] B. Nordström, K. Petersson and J. Smith
An Introduction to Type Theory.
Programming Methodology Group. Chalmers University of Technology, Gothemburg-Sweden. (1983)

[2] B. Nordström and J. Smith
Propositions, Types and Specifications in Martin-Löf's Type Theory.
Dept. of Computer Sciences. Chalmers University of Technology, Gothemburg-Sweden. (1983)

[3] A. Church
An unsolvable problem of elementary number theory.
American Journal of Mathematics. Vol 58 pp. 345-363 (1936)

[4] G. Frege
From Frege to Goedel
A Source Book in Mathematical Logic 1879-1931
ed. J. van Heijenoort, Harvard University Press, Cambridge, Mass. (1967)