

sistemas de tipos

Paradigmas de la Programación
FaMAF-UNC 2015
capítulo 6. Mitchell
filminas de Vitaly Shmatikov

http://en.wikipedia.org/wiki/Type_system

<http://en.wikibooks.org/wiki/>

Category:Introduction to Programming Languages

tipos

- qué son y para qué sirve un sistema de tipos
- tipado
- sistemas de tipos y tipos
- sobrecarga y polimorfismo
- inferencia de tipos

qué es un tipo?

un tipo es una **colección de valores** computables que comparten alguna propiedad estructural

ejemplos

enteros

strings

$\text{int} \rightarrow \text{bool}$

$(\text{int} \rightarrow \text{int}) \rightarrow \text{bool}$

“no ejemplos”

$\{3, \text{true}, \lambda x.x\}$

Even integers

$\{f:\text{int} \rightarrow \text{int} \mid \text{if } x>3$
then $f(x) > x*(x+1)\}$

la distinción entre conjuntos que son tipos y conjuntos que no lo son depende del lenguaje

para qué sirven los tipos

- organización y documentación de programas
 - tipos distintos para usos distintos
 - representan los conceptos del dominio del problema
 - indican el uso de los identificadores de variable
 - se pueden chequear, a diferencia de los comentarios
- identificar y prevenir errores
 - en tiempo de compilación o de ejecución, se pueden prevenir cálculos sin sentido como `3 + true - "Bill"`
- optimización del manejo de memoria (porque se puede calcular el tamaño de los datos)

sistema de tipos

- **jerarquía** de tipos: los tipos se pueden organizar con relación de subtipo
- tipos básicos (o primitivos): indivisibles (atómicos)
- tipos compuestos: tienen diferentes partes
- tipos **definidos por usuarios** (vs. built-in)
 - en tiempo de compilación o de ejecución
 - problemas para detectar errores por los métodos tradicionales
 - si se definen en tiempo de ejecución, no se puede hacer el árbol sintáctico, porque el análisis sintáctico pasa a ser Turing completo

operaciones con valores tipados

- muchas operaciones se definen en función de los tipos
 - enteros: $+$ $-$ $/$ $*$ $<$ $>$... booleanos: \wedge \vee \neg ...
- el conjunto de valores suele ser finito por su representación binaria en la computadora
 - enteros de 32-bits en C: -2147483648 a 2147483647
 - la suma y la resta pueden sobrepasar el rango finito, así que a veces $a + (b + c) \neq (a + b) + c$
 - excepciones: fracciones ilimitadas en Smalltalk, tipo entero ilimitado en Haskell

errores de tipo

- los datos en la memoria de la máquina no tienen información de tipos
 - 010000000101100000000000000000 significa....
- un error de tipo es el que surge porque se intenta realizar una operación en un valor de un tipo para el que la operación no ha sido definida
 - en Fortran y Algol, todos los tipos eran built in, no se podían construir tipos nuevos, así que si se necesitaba un tipo, por ejemplo, “color,” se usaban enteros. Pero entonces se podían hacer cosas como multiplicar colores!!! qué sentido tiene eso?

semántica de tipos

- en algunos lenguajes los tipos son parte de la especificación del lenguaje, por lo tanto su semántica es la misma en cualquier contexto
- en algunos lenguajes la semántica depende del compilador, por ejemplo, en C el tamaño de los enteros depende de la arquitectura

tipos

- qué son y para qué sirve un sistema de tipos
- **tipado**
- sistemas de tipos y tipos
- sobrecarga y polimorfismo
- inferencia de tipos

tipado estático vs. dinámico

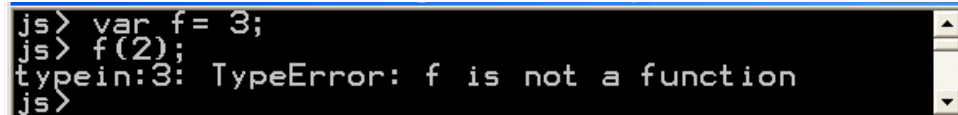
- el sistema de tipos impone restricciones en el uso de valores
 - ej.: sólo valores numéricos en la suma
 - no se puede expresar sintácticamente en una EBNF
- tipado estático: los tipos de las variables se fijan en tiempo de compilación
- tipado dinámico: el tipo de una variable depende de su valor, y eso sólo se puede determinar en tiempo de ejecución

tipado fuerte vs. débil

- tipado fuerte: se detectan todos los errores de tipo, ya sea en tiempo de compilación o de ejecución
- las conversiones de tipos tienen que ser explícitas
- Pascal se considera “demasiado fuerte” porque el tipo de un arreglo o string incluye su longitud
- es difícil detectar todos los errores posibles en los tipos “unión” (una variable es de tipo A o de tipo B)

comprobación en tiempo de compilación o ejecución

- comprobación de tipos (*type-checking*) en tiempo de compilación
 - C, ML
- ... o ejecución
 - Perl, JavaScript
- Java usa ambos
- ventajas y desventajas
 - ambos previenen errores de tipos
 - en tiempo de ejecución hace la ejecución más lenta
 - en tiempo de compilación restringe la flexibilidad del programa
 - JavaScript array: elements can have different types
 - ML list: all elements must have same type



```
js> var f= 3;  
js> f(2);  
typein:3: TypeError: f is not a function  
js>
```

expresividad vs. seguridad

- en JavaScript podemos escribir una función como

```
function f(x) { return x < 10 ? x : x(); }
```

en algunos casos producirá un error de tipos, en otros no

- el tipado estático es siempre conservador

```
if (big-hairy-boolean-expression)  
    then f(5);  
    else f(10);
```

no se puede decidir en tiempo de compilación si habrá un error en ejecución, así que no se puede definir la función de arriba

seguridad de tipos (relativa)

- inseguros: familia BCPL, incluyendo C y C++
 - casteos, aritmética de punteros
- casi seguros: familia Algol, Pascal, Ada
 - punteros colgantes: se asigna un puntero p a un entero, se desasigna la memoria a la que se referenciaba mediante p, luego se usa el valor al que apuntaba p. Los lenguajes con desalojo de memoria no son seguros.
- seguros: Lisp, ML, Smalltalk, JavaScript, Java
 - Lisp, Smalltalk, JavaScript: tipado dinámico
 - ML, Java: tipado estático

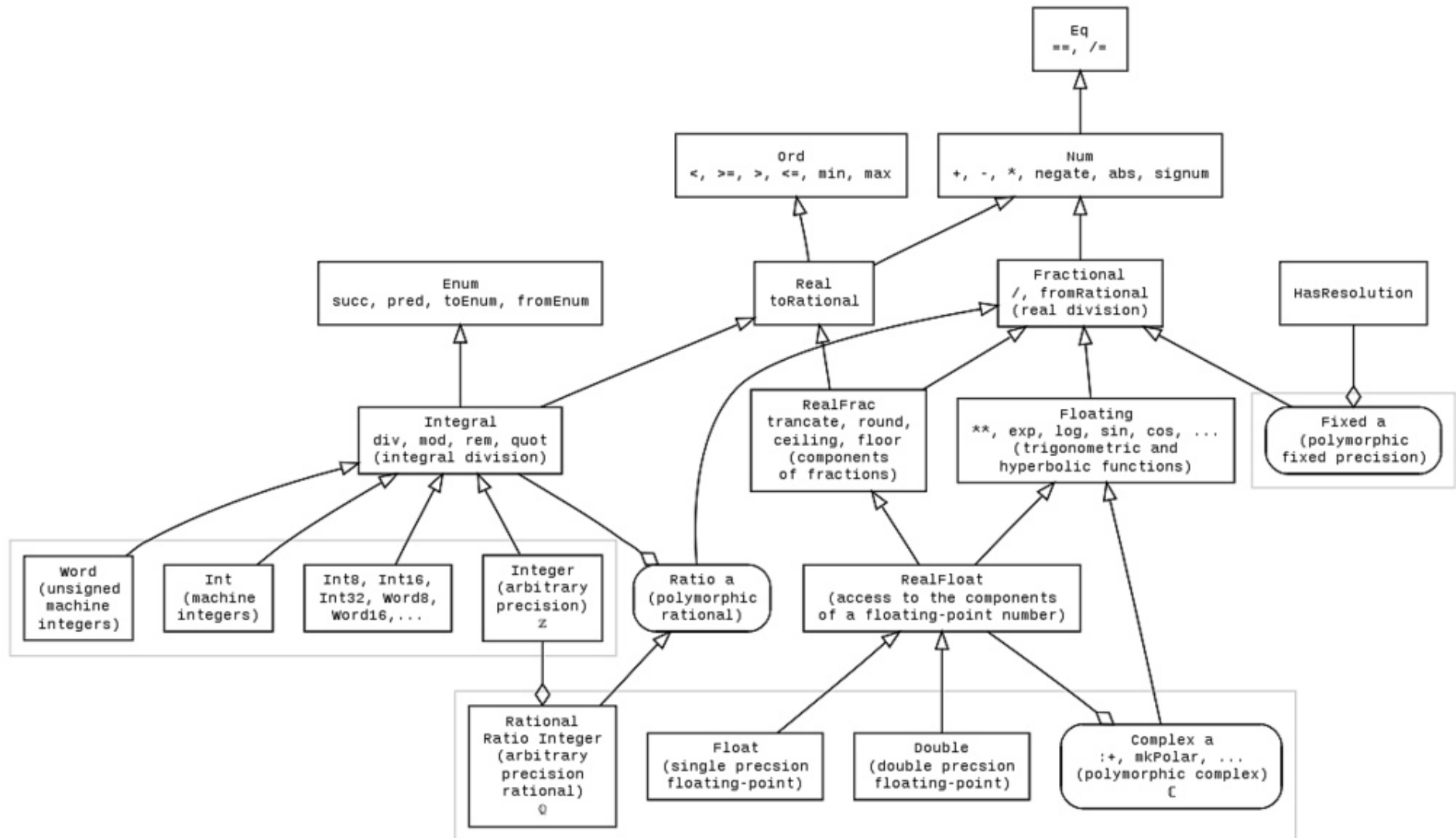
tipos

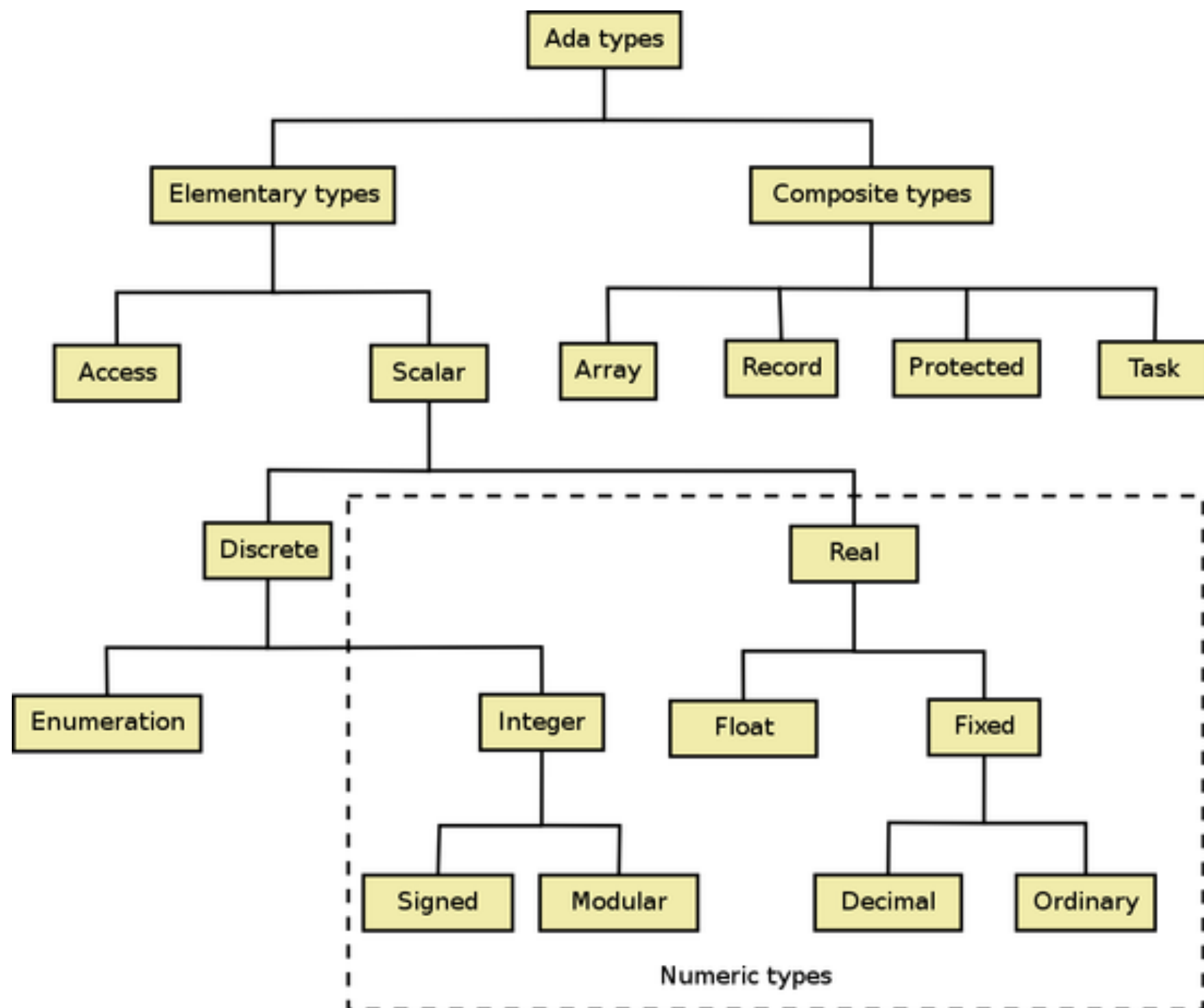
- qué son y para qué sirve un sistema de tipos
- tipado
- **sistemas de tipos y tipos**
- sobrecarga y polimorfismo
- inferencia de tipos

comparación de sistemas de tipos en los diferentes lenguajes

- [http://en.wikipedia.org/wiki/Comparison_of_programming_languages#Type systems](http://en.wikipedia.org/wiki/Comparison_of_programming_languages#Type_systems)

jerarquía de tipos de Haskell





tipos primitivos

- booleano
- entero
- real
- caracter
- string? o es un arreglo de caracteres?

tipos contruidos

- enumeraciones
- tuplas
- registros (records)
- listas
- arreglos
- arreglos asociativos
- clases
- funciones

enumeraciones

un conjunto de valores definidos por el usuario

```
enum day {Lunes, Martes, Miércoles,  
          Jueves, Viernes, Sábado, Domingo};
```

```
enum day myDay = Miércoles;
```

– en C/C++, los valores de los tipos enumeración se representan como enteros: 0, ..., 6

- más poderoso en Java:

```
for (day d : day.values())  
    System.out.println(d);
```

punteros

- C, C++, Ada, Pascal
- el valor es una dirección de memoria
- permite referencia directa
- punteros en C/C++
 - Si T es un tipo y la referencia T es un puntero:
 $\& : T \rightarrow \text{ref } T$ $* : \text{ref } T \rightarrow T$ $*(&x) = x$
- el acceso a memoria explícito mediante punteros puede resultar en código erróneo y vulnerabilidades de seguridad

arreglos

- ejemplo: `float x[3][5];`
- indexar []
 - signature de tipos: $T[] \times \text{int} \rightarrow T$
 - en el ejemplo de arriba, el tipo de `x`: `float[] []`, el tipo de `x[1]`: `float[]`, el tipo de `x[1][2]`: `float`
- equivalencia entre arreglos y punteros
 - `a = &a[0]`
 - If either e1 or e2 is type: ref T,
then $e1[e2] = *((e1) + (e2))$
 - Example: a is `float[]` and i int, so $a[i] = *(a + i)$

strings

- actualmente es muy básico y los lenguajes proveen soporte nativo para este tipo
- en C: un string es un arreglo de caracteres unidimensional terminado en un caracter NULL
- en Java, Perl, Python: una variable string puede tener un número ilimitado de caracteres
- hay muchas librerías de operaciones y funciones sobre strings
 - las librerías de Standard C para strings son inseguras!

estructuras

- colección de elementos de distintos tipos
 - no existen en Fortran ni Algol 60 se usaron por primera vez en Cobol, PL/I
 - frecuentes en lenguajes como Pascal o C
 - redundantes con los objetos de Java

```
struct employeeType {  
    char name[25];  
    int age;  
    float salary;  
};  
struct employeeType employee;  
employee.age = 45;
```

uniones

se llaman uniones en C, case-variant records en Pascal

- Idea: multiple views of same storage

```
type union =  
  record  
    case b : boolean of  
      true : (i : integer);  
      false : (r : real);  
    end;  
var tagged : union;  
begin tagged := (b => false, r => 3.375);  
  put(tagged.i); -- error
```

uniones

- unión de varios tipos (=conjuntos de valores)
- algunos lenguajes fuerzan al programador a distinguir cada tipo base de una unión en su uso, para prevenir errores de tipo

uniones: C

```
union element {  
    int i;  
    float f;  
};  
  
int main() {  
    union element e;  
    e.f = 177689982.02993F;  
    printf("Int = %d\n", e.i);  
    printf("Float = %f\n", e.f);  
    e.f = 0.0F;  
}
```

uniones: SML

```
datatype element =  
  I of int | F of real;  
fun getReal (F x) = x  
  | getReal (I x) = real x;
```

- las etiquetas permiten saber el tipo de dato que se guarda en la variable en tiempo de ejecución
- C no usa etiquetas, por lo tanto no puede hacer chequeo de tipos en las uniones

tipos de datos recursivos

- `data Value = IntValue Integer | FloatValue Float | BoolValue Bool | CharValue Char deriving (Eq, Ord, Show)`
- `data Expression = Var Variable | Lit Value | Binary Op Expression Expression | Unary Op Expression deriving (Eq, Ord, Show)`
- `type Variable = String`
- `type Op = String`
- `type State = [(Variable, Value)]`

funciones como tipos

- Pascal:

```
function newton(a, b: real; function f:  
    real): real;
```

- Declara que f devuelve un valor real, pero los argumentos de f están infraespecificados

- Java:

```
public interface RootSolvable {double  
    valueAt(double x);}
public double Newton(double a, double  
    b, RootSolvable f);
```


equivalencia de tipos

- Pascal Report:
“The assignment statement serves to replace the current value of a variable with a new value specified as an expression ... The variable (or the function) and the expression must be of identical type”
- pero qué tipos son equivalentes?

```
struct complex { float re, im; };  
struct polar { float x, y; };  
struct { float re, im; } a, b;  
struct complex c,d;    struct polar e;    int f[5],  
    g[10];
```

subtipos

- un subtipo es un tipo con algunas restricciones en sus valores u operaciones
- algunos lenguajes permiten especificarlos directamente

Ada:

```
subtype one_to_ten is Integer range 1 .. 10;
```

- concepto básico para programación orientada a objetos

tipos definidos por el usuario

- definiendo tipos en Haskell

```
data Shape = Circle Float Float Float |  
Rectangle Float Float Float Float
```

```
surface :: Shape -> Float
```

```
surface (Circle _ _ r) = pi * r ^ 2
```

```
surface (Rectangle x1 y1 x2 y2) = (abs $ x2 -  
x1) * (abs $ y2 - y1)
```

tipos

- qué son y para qué sirve un sistema de tipos
- tipado
- sistemas de tipos y tipos
- **sobrecarga y polimorfismo**
- inferencia de tipos

sobrecarga

- sobrecargamos un operador o función cuando su significado varía dependiendo de los tipos de sus operandos o argumentos o resultado
- ejemplos:
 - suma: enteros y decimales, también concatenación de strings en Java
 - Class PrintStream en Java:
`print` y `println` definidos para boolean, char, int, long, float, double, char[], String, Object

sobrecarga de funciones en C++

- funciones con el mismo nombre pero que pueden tomar argumentos de distinto tipo

```
inline void swap(int& a, int& b) { int temp = a; a = b; b = temp; }  
inline void swap(char& a, char& b) { char temp = a; a = b; b = temp; }  
inline void swap(float& a, float& b) { float temp = a; a = b; b = temp; }
```

- el compilador sustituye el código de la función en el momento de invocarla

Overloading Infix Operators in C++

```
class Complex {
private:
    long double r; // real part
    long double i; // imaginary part
public:
    /* "Complex object constructor function" */
    Complex () { r = 0.0; i = 0.0; }
    Complex (double real, double imag) { r = real; i = imag; }
    ...
    /* "friend" functions can access the private data of a Complex object */
    friend Complex operator+ (Complex a, Complex b) { return Complex(a.r+b.r, a.i+b.i); }
    friend Complex operator- (Complex a, Complex b) { return Complex(a.r-b.r, a.i-b.i); }
    friend Complex operator* (Complex a, Complex b) { return ...; }
    friend Complex operator/ (Complex a, Complex b) { return ...; }
};

Complex x; // same as Complex x(0.0,0.0);
Complex a(1.0, 0.0);
Complex b(2.5, 3.0);
Complex c(2.0, 2.0);
...
Complex r = a + b * c; // a + (b * c) --- you can't change associativity in C++
```

Cannot change position, associativity or precedence

sobrecarga de operadores en ML

- ML infiere qué función usar a partir del tipo de los operandos

```
- 3 + 5;  
val it = 8 : int  
- 3.14 + 2.0;  
val it = 5.14 : real  
- 3.14 + 2;  
stdIn:1.1-2.4 Error: operator and operand don't agree [literal]  
  operator domain: real * real  
  operand:          real * int  
  in expression:  
    + : overloaded (3.14, (2 : int))
```


operadores infijos definidos por el usuario en ML

```
- infix xor;  
infix xor  
  
- fun p xor q = (p orelse q) andalso not (p andalso q);  
val xor = fn : bool * bool -> bool  
  
- true xor false xor true;  
val it = false : bool
```

– la precedencia se especifica con enteros 0-9

```
- infix 6 plus;  
infix 6 plus  
- fun a plus b = "(" ^ a ^ "+" ^ b ^ " )";  
val plus = fn : string * string -> string  
  
- infix 7 times;  
infix 7 times  
- fun a times b = "(" ^ a ^ "*" ^ b ^ " )";  
val times = fn : string * string -> string
```

polimorfismo y genéricos

- un operador o función es polimórfico si se puede aplicar a cualquier tipo relacionado
 - permite reuso de código
- ejemplo: funciones genéricas en C++
 - la función hace exactamente lo mismo independientemente del tipo de los argumentos

```
template<class type> void swap(type& a, type& b) { type temp = a; a = b; b = temp; }
```

- en cada uso, el compilador sustituye el tipo de los argumentos por los parámetros de tipo de la plantilla

```
void swap(int& a, int& b) { int temp = a; a = b; b = temp; }
```

- ejemplo de **polimorfismo paramétrico**

polimorfismo vs. sobrecarga

- polimorfismo paramétrico
 - un solo algoritmo puede tener diferentes tipos
 - la variable de tipo se reemplaza por cualquier tipo
 - $f : t \rightarrow t \Rightarrow f : \text{int} \rightarrow \text{int}, f : \text{bool} \rightarrow \text{bool}, \dots$
- sobrecarga
 - un solo símbolo se refiere a más de un algoritmo
 - cada algoritmo tiene diferentes tipos
 - se selecciona el algoritmo dependiendo del contexto de tipos
 - los tipos pueden ser arbitrariamente distintos

tipos

- qué son y para qué sirve un sistema de tipos
- tipado
- sistemas de tipos y tipos
- sobrecarga y polimorfismo
- **inferencia de tipos**

comprobación vs. inferencia de tipos

- comprobación de tipos estándar

```
int f(int x) { return x+1; };  
int g(int y) { return f(y+1)*2; };
```

mirar el cuerpo de la función y usar los **tipos declarados** de los identificadores para comprobar

- inferencia de tipos

```
int f(int x) { return x+1; };  
int g(int y) { return f(y+1)*2; };
```

mirar el código sin información de tipo y **figurarse** qué tipos se podrían haber declarado

Motivation

- Types and type checking
 - Type systems have improved steadily since Algol 60
 - Important for modularity, compilation, reliability
- Type inference
 - Widely regarded as important language innovation
 - ML type inference is an illustrative example of a **flow-insensitive static analysis algorithm**
 - What does this mean?

inferencia de tipos en ML

- ejemplo

- `fun f(x) = 2+x;`
 - `> val it = fn : int → int`

- cómo funciona?

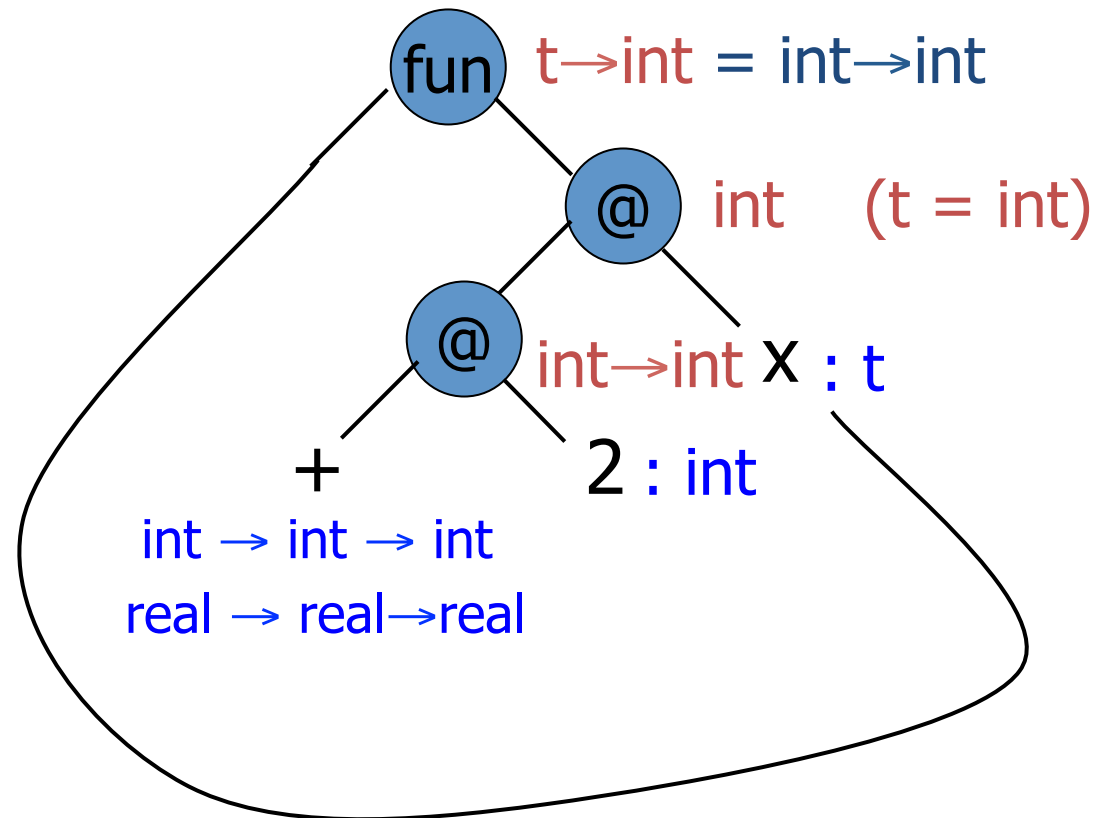
- `+` tiene dos tipos: `int*int → int`,
`real*real→real`
 - `2 : int` tiene un solo tipo
 - esto implica que `+` : `int*int → int`
 - por el contexto, es necesario que `x: int`
 - por lo tanto, `f(x:int) = 2+x` tiene tipo `int
→ int`

cómo funciona?

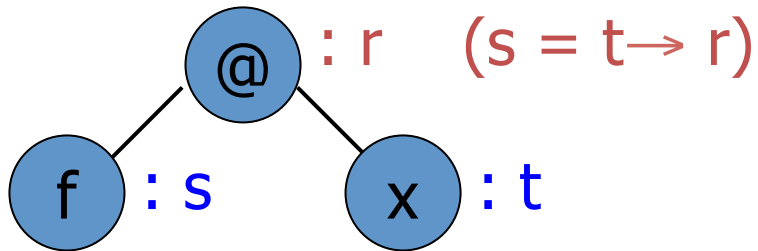
```
fun f(x) = 2+x;  
> val it = fn :  
    int → int
```

grafo para $f(x) = 2+x$

1. asignar tipo a las hojas
2. propagar a nodos internos y asignar restricciones
3. resolver por sustitución

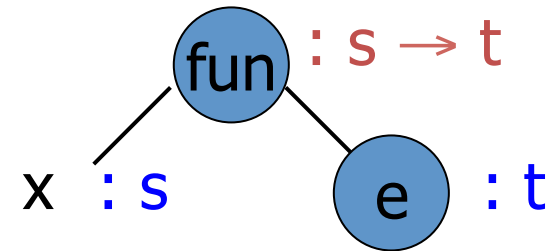


restricciones expresadas en el grafo



aplicación de función

- f tiene que tener el tipo función **dominio→rango**
- el dominio de f tiene que ser el tipo del argumento x
- el tipo del resultado es el rango de f



expresión (definición) de función

- el tipo es el tipo de función **dominio→rango**
- el dominio es el tipo de la variable x
- el rango es el tipo del cuerpo de la función e

1. asignación de tipo a los nodos

Subexpresión	Tipo
$\lambda x. ((+ 5) x)$	r
$((+ 5) x)$	s
$(+ 5)$	t
+	$\text{int} \rightarrow (\text{int} \rightarrow \text{int})$
5	int
x	u

1. asignación de tipo a los nodos
2. generar restricciones, usando el árbol sintáctico de la expresión. Las restricciones son ecuaciones entre los tipos que deben resolverse, y dependen de la forma de cada subexpresión.
 - restricciones (ecuaciones) para aplicación de función: si el tipo de la función es a , el tipo del cuerpo de la función es e y el tipo de $f(e)$ es c , entonces debemos tener $a = b \rightarrow c$.
 - restricciones para expresión (definición) de función: si el tipo de x es a y el tipo de e es b , entonces el tipo de $\lambda x.e$ debe ser $a \rightarrow b$.

1. asignación de tipo a los nodos
2. generar restricciones, usando el árbol sintáctico de la expresión. Las restricciones son ecuaciones entre los tipos que deben resolverse, y dependen de la forma de cada subexpresión

$$\text{int} \rightarrow (\text{int} \rightarrow \text{int}) = \text{int} \rightarrow t,$$

$$t = u \rightarrow s,$$

$$r = u \rightarrow s.$$

1. asignación de tipo a los nodos
2. generar restricciones, usando el árbol sintáctico de la expresión. Las restricciones son ecuaciones entre los tipos que deben resolverse, y dependen de la forma de cada subexpresión
3. resolver por sustitución

$$\text{int} \rightarrow (\text{int} \rightarrow \text{int}) = \text{int} \rightarrow t,$$

$$t = u \rightarrow s,$$

$$r = u \rightarrow s.$$

1. asignación de tipo a los nodos
2. generar restricciones, usando el árbol sintáctico de la expresión. Las restricciones son ecuaciones entre los tipos que deben resolverse, y dependen de la forma de cada subexpresión
3. resolver por sustitución

$$\begin{aligned} \text{int} \rightarrow (\text{int} \rightarrow \text{int}) &= \text{int} \rightarrow t, \\ t &= u \rightarrow s, \\ r &= u \rightarrow s. \end{aligned}$$

$$\begin{aligned} \text{int} \rightarrow \text{int} &= u \rightarrow s, \\ r &= u \rightarrow s. \end{aligned}$$

1. asignación de tipo a los nodos
2. generar restricciones, usando el árbol sintáctico de la expresión. Las restricciones son ecuaciones entre los tipos que deben resolverse, y dependen de la forma de cada subexpresión
3. resolver por sustitución

$$\text{int} \rightarrow (\text{int} \rightarrow \text{int}) = \text{int} \rightarrow t,$$

$$t = u \rightarrow s,$$

$$r = u \rightarrow s.$$

$$\text{int} \rightarrow \text{int} = u \rightarrow s,$$

$$r = u \rightarrow s.$$

$$r = \text{int} \rightarrow \text{int}.$$

1. asignación de tipo a los nodos
2. generar restricciones, usando el árbol sintáctico de la expresión. Las restricciones son ecuaciones entre los tipos que deben resolverse, y dependen de la forma de cada subexpresión
3. resolver por sustitución

$$\text{int} \rightarrow (\text{int} \rightarrow \text{int}) = \text{int} \rightarrow t,$$

$$t = u \rightarrow s,$$

$$r = u \rightarrow s.$$

$$\text{int} \rightarrow \text{int} = u \rightarrow s,$$

$$r = u \rightarrow s.$$

$$r = \text{int} \rightarrow \text{int}.$$

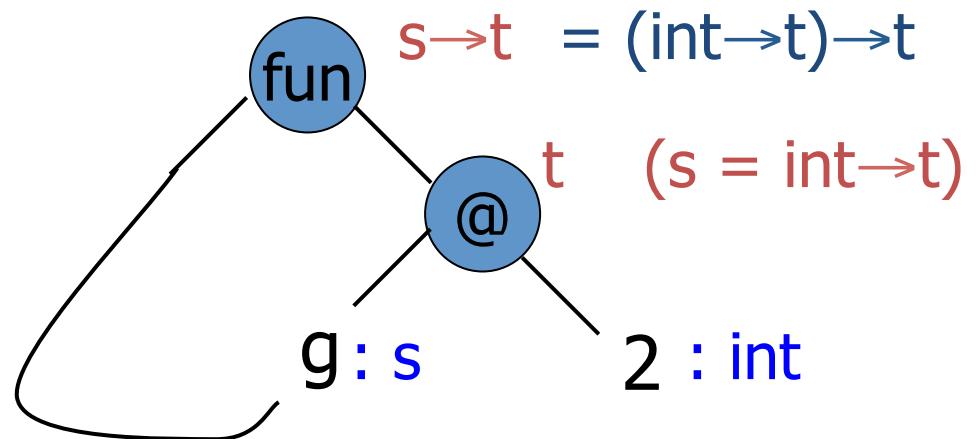
$$u = \text{int}, s = \text{int}.$$

tipos con variables de tipo

```
fun f(g) = g(2);  
val it = fn :  
  (int → t) → t
```

1. asignar tipos a las hojas
2. propagar a nodos internos y generar restricciones
3. resolver por sustitución

grafo para $f(g) = g(2)$



con una función polimórfica

```
fun f(g) = g(2);  
> val it = fn : (int  
  → t) → t
```

aplicaciones posibles

```
- fun add(x) = 2+x;  
> val it = fn : int  
  → int  
  
- f(add);  
> val it = 4 : int
```

reconocer errores de tipos con inferencia

```
fun f(g) = g(2);  
> val it = fn : (int → t) → t
```

uso incorrecto

```
fun not(x) = if x then false else  
  true;  
> val it = fn : bool → bool  
f(not);
```

error de tipo: no se puede hacer $\text{bool} \rightarrow$

$\text{bool} = \text{int} \rightarrow t$

otro ejemplo de inferencia

```
fun f(g,x) = g(g(x));
```

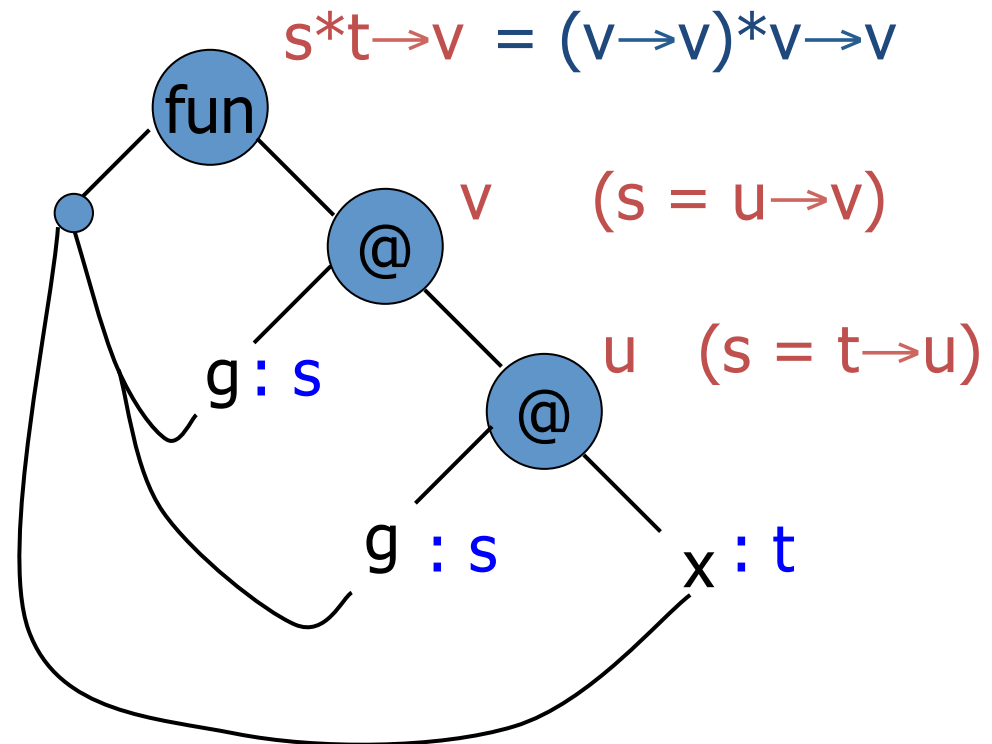
```
> val it = fn : (t → t)*t → t
```

grafo para $f(g,x) = g(g(x))$

1. Asignar tipos a las hojas

2. Propagar a los nodos internos y generar restricciones

3. resolver por sustitución



tipos de datos polimórficos

tipo de dato con variable de tipo
(llamamos 'a a las variables de tipo)

```
datatype 'a list = nil | cons of 'a*( 'a list)
> nil : 'a list
> cons : 'a*( 'a list) → 'a list
```

- función polimórfica

```
fun length nil = 0
    | length (cons(x,rest)) = 1 + length(rest)
> length : 'a list → int
```

- inferencia de tipos

- se infiere un tipo separado para cada cláusula
- si es necesario combinar, se hacen los dos tipos iguales (equivalentes)

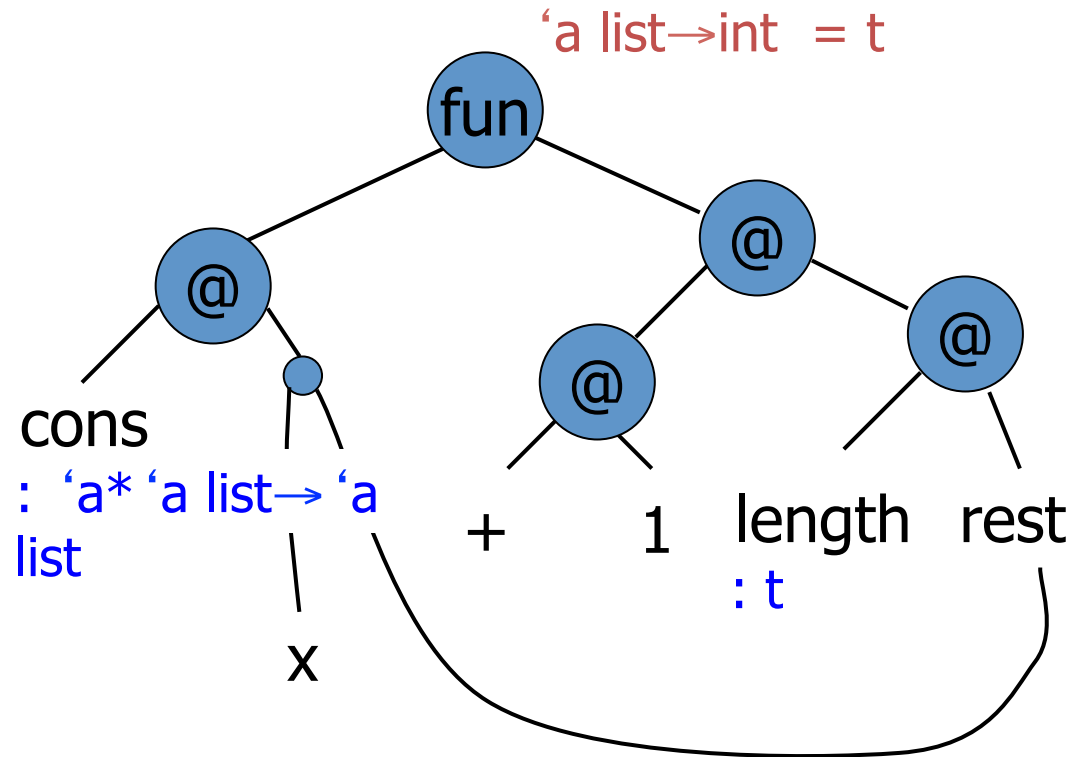
inferencia con recursión

- segunda cláusula

```
length(cons  
  (x,rest)) = 1 +  
length(rest)
```

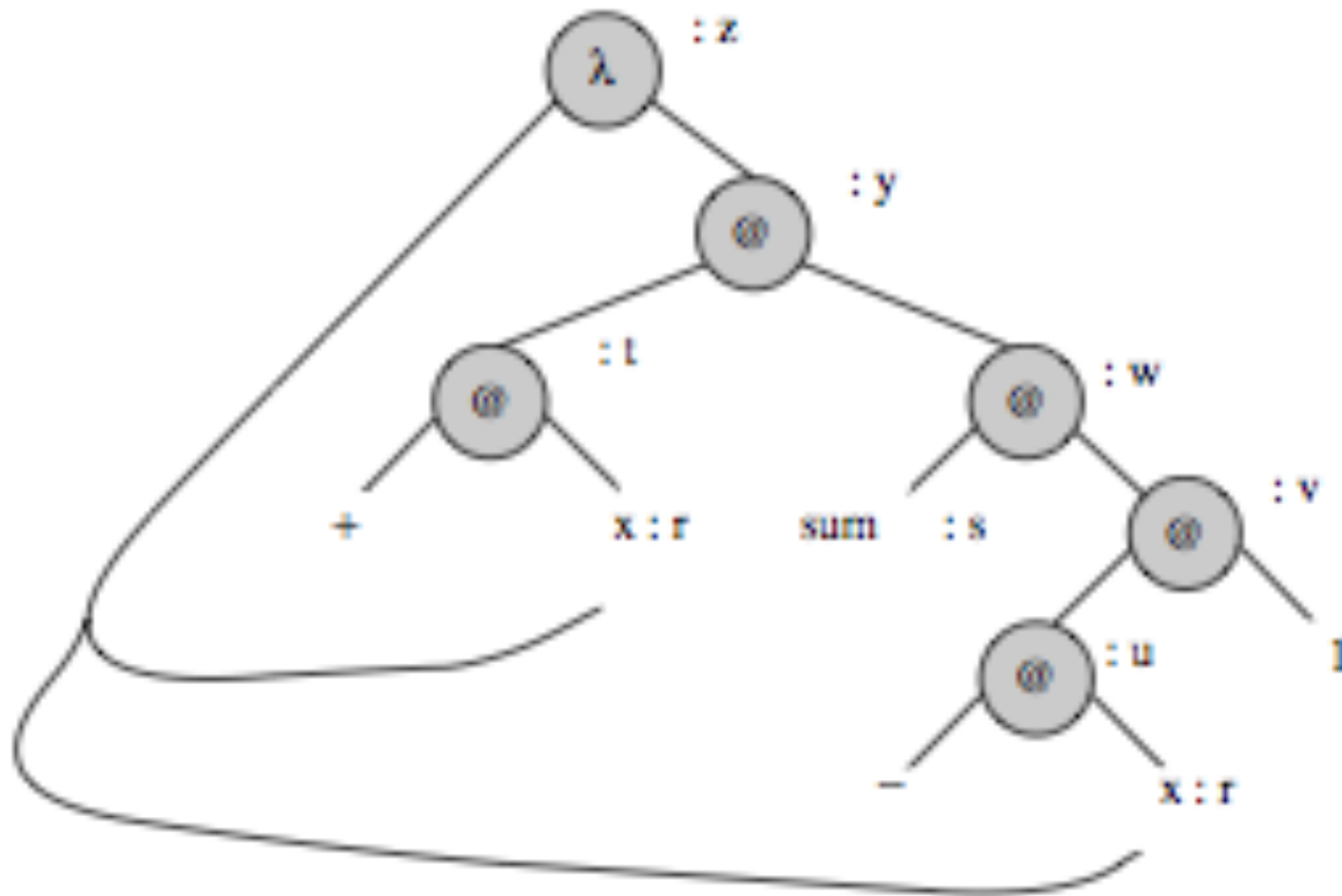
- inferencia de tipos

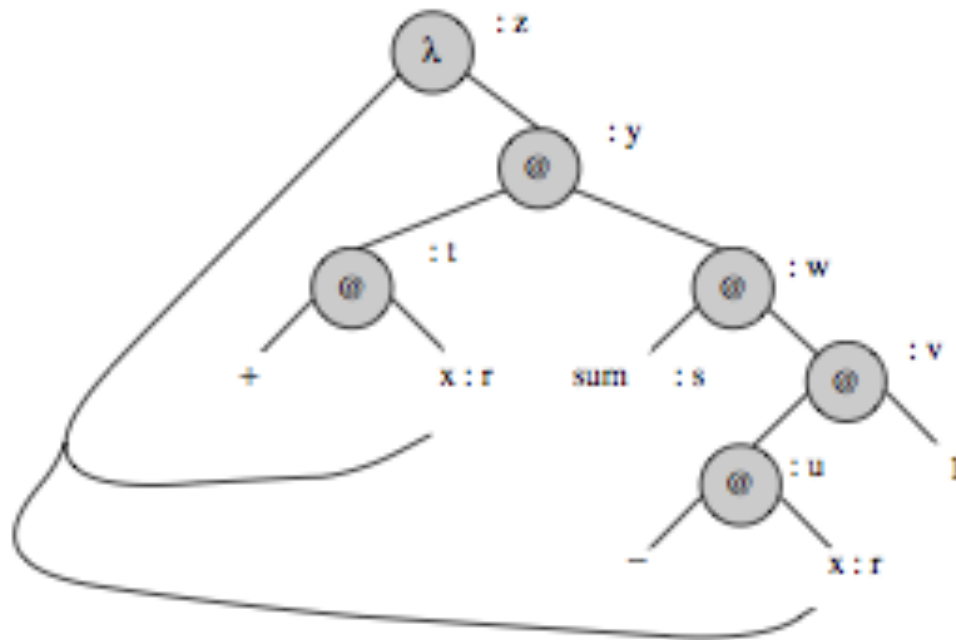
- se asignan tipos a las hojas, incluyendo el nombre de la función
- seguimos como siempre
- añadir la restricción de que el tipo del cuerpo de la función es igual al tipo del nombre de la función



```
fun sum(x) = x+sum(x-1);  
val sum = fn : int -> int
```

fun sum(x) = x+sum(x-1)





$\text{int} \rightarrow (\text{int} \rightarrow \text{int}) = r \rightarrow t,$

$\text{int} \rightarrow (\text{int} \rightarrow \text{int}) = r \rightarrow u,$

$u = \text{int} \rightarrow v,$

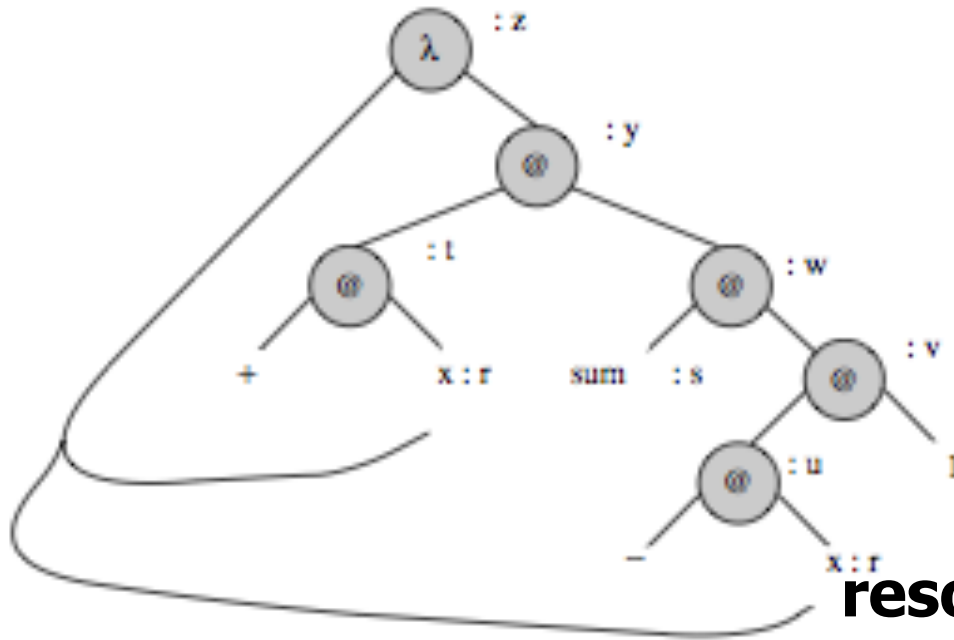
$s = v \rightarrow w,$

$t = w \rightarrow y,$

$z = r \rightarrow y.$

y además:

$s = z.$



resolvemos paso a paso:

$\text{int} \rightarrow (\text{int} \rightarrow \text{int}) = r \rightarrow t,$

$\text{int} \rightarrow (\text{int} \rightarrow \text{int}) = r \rightarrow u,$

$u = \text{int} \rightarrow v,$

$s = v \rightarrow w,$

$t = w \rightarrow y,$

$z = r \rightarrow y.$

y además:

$s = z.$

$r = \text{int}, t = \text{int} \rightarrow \text{int},$

$u = \text{int} \rightarrow \text{int},$

$v = \text{int},$

$s = \text{int} \rightarrow w,$

$t = w \rightarrow y,$

$z = r \rightarrow y,$

$w = \text{int}, y = \text{int},$

$z = \text{int} \rightarrow \text{int}, s = \text{int} \rightarrow \text{int}.$

resumen de inferencia de tipos

- se computa el tipo de las expresiones, no se declara
 - no requiere declaración de tipo para variables
 - se encuentra el tipo más general resolviendo restricciones
 - lleva al polimorfismo
- chequeo de tipos estático sin especificaciones de tipo
- a veces es mejor que chequeo de tipos, porque puede detectar más errores

coste de la inferencia de tipos

- más difícil identificar la línea del programa que causa el error
- diferentes tipos pueden requerir diferentes sintaxis
- Complicaciones con la asignación de valor

información que puede proveer la inferencia de tipos

ejemplo:

```
fun reverse (nil) = nil  
  | reverse (x::lst) = reverse(lst);
```

- se infiere

```
reverse : 'a list → 'b list
```

- qué significa esto? como darle la vuelta a una lista no cambia su tipo, tiene que haber un error en la definición de `reverse`

polimorfismo paramétrico: ML vs. C++

- ML polymorphic function
 - Declaration has no type information
 - Type inference: type expression with variables, then substitute for variables as needed
 - C++ function template
 - Declaration gives type of function argument, result
 - Place inside template to define type variables
 - Function application: type checker does instantiation
- ML also has module system with explicit type parameters

Example: Swap Two Values

- ML

```
- fun swap(x,y) =  
    let val z = !x in x := !y; y := z end;  
val swap = fn : 'a ref * 'a ref -> unit
```

- C++

```
template <typename T>  
void swap(T& x, T& y){  
    T tmp = x; x=y; y=tmp;  
}
```

Declarations look similar, but compiled very differently

Implementation

- ML
 - Swap is compiled into one function
 - Typechecker determines how function can be used
- C++
 - Swap is compiled into linkable format
 - Linker duplicates code for each type of use
- Why the difference?
 - ML reference cell is passed by pointer, local x is a pointer to value on heap
 - C++ arguments passed by reference (pointer), but local x is on stack, size depends on type

Another Example

- C++ polymorphic sort function

```
template <typename T>
void sort( int count, T * A[count] ) {
    for (int i=0; i<count-1; i++)
        for (int j=i+1; j<count-1; j++)
            if (A[j] < A[i]) swap(A[i],A[j]);
}
```

- What parts of implementation depend on type?
 - Indexing into array
 - Meaning and implementation of <

ML Overloading and Type Inference

- Some predefined operators are overloaded
- User-defined functions must have unique type
 - `fun plus(x,y) = x+y;`
This is compiled to int or real function, not both
- Why is a unique type needed?
 - Need to compile code \Rightarrow need to know which +
 - Efficiency of type inference
 - Aside: general overloading is NP-complete

Resumen

- los tipos son una parte importante del lenguaje
 - organizan y documentan el programa, previenen errores, proveen información para el compilador
- inferencia de tipos
 - determina automáticamente el mejor tipo para una expresión, según los tipos de los símbolos de la expresión
- polimorfismo
 - un único algoritmo puede tener distintos tipos
- sobrecarga
 - un símbolo con muchos sentidos (distintos algoritmos), se resuelve cuando se compila el programa