

Polymorphic type inference

Haskell's type-inference system works for polymorphic types.

Example:

```
> foldr f k [] = k
> foldr f k (x:xs) = f x (foldr f k xs)
```

Informally, we note how the parameters are used, and type them separately (using as many type variables as we need):

```
f      :: a->b->c -- a 2-argument function
k      :: d
[], (x:xs) :: [e] -- list patterns
x      :: e -- left argument of (:)
xs     :: [e] -- right argument of (:)
so
foldr :: (a->b->c) -> d -> [e] -> g
```

Then we look for equalities between type variables:

```
g = d because foldr f k [] = k
e = a because f is applied to x
g = b because f is applied to (foldr f k xs)
g = c because foldr f k (x:xs)
      is defined as an application of f
```

Eliminating variables, we obtain

```
foldr :: (a->b->b) -> b -> [a] -> b
```

Overloading

- = the use of a *single* name or symbol for *two* or more distinct items of different types (an item is selected according to the context in which the name or symbol occurs).

Common example of overloading:

Arithmetic operators in most programming languages:

```
(* :: int × int → int integer multiplication
(*) :: real × real → real floating-point multiplication

2 * 3 (*) is integer multiplication
2.0 * 3.0 (*) is floating-point multiplication
```

An overloaded name's type either

- contains no type variables (most languages) or
- restricts the type variables' range (Haskell)

An *overloaded* name refers to a finite number of different definitions; a *polymorphic* definition is the same for all types.

In an application of an overloaded function name or operator, which of the name's functions is invoked depends on the types of the argument(s).

In an application of a function to an overloaded argument, the selection of the argument's actual type (and hence its value) depends on the type of the function applied to it.

Conversion

A function that takes an argument of type α and produces a result of type β may be viewed as an $\alpha \rightarrow \beta$ type-conversion function.

Example:

```
toFloat :: int → real
fromInt :: Num a ⇒ Int -> a
```

Coercion

= *implicit* conversion

Consider the expression

```
2 + 3.14159265
```

Because $2 :: \text{integer}$ and $3.14159265 :: \text{real}$, neither

```
(+) :: int × int → int
```

nor

```
(+) :: real × real → real
```

is applicable.

One solution: overload (+) with more types —

```
(+) :: int × int → int      (+) :: int × real → real
(+) :: real × int → real    (+) :: real × real → real
```

A cheaper solution: automatically insert a tacit type-conversion function—

```
2 + 3.14159265 → toFloat 2 + 3.14159265
```

In C/C++ and Java, coercion is very common.

Pascal provides no coercion; all type conversions are explicit.

Haskell provides coercion *only* for integer literals.

Types in Pascal, C/C++/Java, and Haskell

Built-in Types (value sets and their operations)

Categories of operations

logical:

Pascal:	and	or	not
C:	&&	 	!
Haskell:	&&	 	not

arithmetic:

Pascal:	+	-	*	/	div	mod
C/C++/Java:	+	-	*	/		%
Haskell:	+	-	*	/	div	mod

comparison:

Pascal:	<	<=	=	<>	>=	>
C/C++/Java:	<	<=	==	!=	>=	>
Haskell:	<	<=	==	/=	>=	>

Enumerations

Pascal:

(*north, east, south, west*)

operations:

comparison
ord, succ, pred

C/C++ (not Java):

enum { north, east, south, west }

operations:

comparison, arithmetic, logical

Haskell:

> **data** Direction = North | East | South | West

Boolean

Pascal: **boolean**
(**false, true**)

operations: logical

C: (see Integer; if $(-1 \leq x \leq 1)$...)

C++: `bool = {false, true}`

Java: `boolean = {false, true}`

Haskell: `data Bool = False | True`

Character

Pascal: **char** (ascii or ebcdic or Kanji or ...)
operations: comparison, *ord*

C: (see Integer)

C++: `char` (implementation-dependent)

Java: `char` (16-bit Unicode)

Haskell: `Char` (8-bit ASCII)

Integer

Pascal: **integer** (value set is processor-dependent)
arithmetic, comparison, *chr*

Modula-2: `INTEGER, CARDINAL`

C ("integral" types):

<code>int</code>	<code>INTEGER</code>
<code>[signed unsigned] [long short] int</code>	
<code>unsigned int</code>	<code>CARDINAL</code>
<code>char {0..255} or {-128..127}</code>	
<code>signed char {-128..127}</code>	
<code>unsigned char {0..255}</code>	
<code>long (= long int)</code>	
<code>etc.</code>	

The compile-time C function `sizeof(T)` gives the storage size of T in bytes, where T is a type name or a variable name.

Standard relationships among C's integral types:

$1 \leq \text{sizeof}(\text{char}) \leq \text{sizeof}(\text{short})$

$\leq \text{sizeof}(\text{int}) \leq \text{sizeof}(\text{long})$

$1 \leq \text{sizeof}(\text{bool}) \leq \text{sizeof}(\text{long})$

$\text{sizeof}(T) = \text{sizeof}(\text{signed } T)$

$= \text{sizeof}(\text{unsigned } T)$

for $T \in \{ \text{char}, \text{short}, \text{int}, \text{long} \}$

Java: return to simplicity

`byte`: 8-bit signed 2's-complement integer

`short`: 16-bit signed 2's-complement integer

`int`: 32-bit signed 2's-complement integer

`long`: 64-bit signed 2's-complement integer

Haskell

`Int` processor-supported integers (32 bits)

`Integer` unbounded-magnitude integers
("bignums")

Subrange types

Pascal: `constant0 .. constant1`
for types

integer, char, enumerations

examples: `3..17, 'a'..'z'`

Note: **integer** subrange types are **not closed** under arithmetic operations.

Modula-2: as in Pascal

C/C++/Java: no subrange types

Haskell: no subrange types

Floating-point numbers

Pascal: **real**
(machine-dependent floating-point)

C/C++: `float, double`
(machine-dependent floating-point)

$\text{sizeof}(\text{float}) \leq \text{sizeof}(\text{double})$
 $\leq \text{sizeof}(\text{long double})$

Java:

`float`: 32-bit IEEE 754-1985 floating-point

`double`: 64-bit IEEE 754-1985 floating-point

Haskell: `Float, Double` (also `Rational`, ...)

Sets

Pascal: *type constructor*: **set of** $\langle T \rangle$
where T is the set's *base type*

$T \in \{ \text{integer subrange, enumeration, char} \}$

T 's cardinality \leq some small number
(e.g., 256)

examples of set types:

set of char

set of (*red, yellow, green, blue, violet*)

set of 1..10

set-value constructor: `[]`

examples of set values:

`[1 .. 10]` :: (**set of** 0 .. 100)

`['a', 'e', 'i', 'o', 'u']` :: (**set of char**)

`['3' .. '7']` :: (**set of** '0' .. '9')

set selector: **in** :: $T \times \text{set of } T \rightarrow \text{boolean}$

`3 in [1 .. 10]`

other set operations:

$(+, -, *)$:: **set of** $T \times \text{set of } T \rightarrow \text{set of } T$

$(\leq, =, <, >, \geq)$:: **set of** $T \times \text{set of } T \rightarrow \text{boolean}$

Implementation of sets: boolean vectors
(i.e., *characteristic vectors*)

A, B : **set of** 0..9

$\approx A, B$: **array** [0..9] **of boolean**

Example: `[2, 3, 7, 8]` :: (**set of** 0..9)

0 1 2 3 4 5 6 7 8 9

F	F	T	T	F	F	F	T	T	F
---	---	---	---	---	---	---	---	---	---

$x \text{ in } A \approx A[x]$

$A + B \approx R$ **where**

$(\forall i | 0 \leq i \leq 9: R[i] = A[i] \vee B[i])$

$A \leq B \approx (\forall i | 0 \leq i \leq 9: A[i] \Rightarrow B[i])$

For storage economy and speed, sets' boolean vectors are usually stored as densely as possible: one element per bit.

C/C++/Java: (no built-in *set* type)

Haskell: (no built-in *set* type)