

Types and Sets

Membership notation

sets: $a \in A$

types: $a :: \text{Int}$

Atomic Types

built-in: *bool, int, real, char, ...*

enumerations: *color = red | white | blue*

Composite Types

type constructor

constructs a type from component types

value constructor

constructs a value from component values

value selector

selects a component value from composite values

Product types (sets of ordered n -tuples or *records*):

$A \times B = \{(a,b) \mid a :: A \wedge b :: B\}$

typical constructors: $\dots \times \dots$ (\dots, \dots) RECORD...END

examples: *int* \times *bool*, (*Int, Bool*) (3, **false**)

selectors: *first, second, ...* (or patterns or field names)

example: *first* (3, **false**) \rightsquigarrow 3

second (3, **false**) \rightsquigarrow **false**

A product type can be

- homogeneous: all elements have the same type
- heterogeneous: elements' types are mixed

Computed selection (i.e., indexing) on **heterogeneous** product types is **incompatible** with compile-time (i.e., *static*) type checking.

For example, would this expression be valid?

$2 + ((\text{True}, 2, \text{"form"})!!k)$

Statically, there's no way to decide.

Function types

type constructor: $(\alpha \rightarrow \beta)$ is the type "functions from α to β ".

$\alpha \rightarrow \beta = \{f \mid (\forall a \mid a :: \alpha : f a :: \beta)\}$

value constructor: function definition

$f x = E$

$x :: \alpha \wedge E :: \beta \Rightarrow f :: \alpha \rightarrow \beta.$

"constructs" a (possibly infinite) set of function pairs

selector: function application $f(a)$

$f :: \alpha \rightarrow \beta \wedge a :: \alpha \Rightarrow f a :: \beta.$

"selects" a function pair's second member (given the first member)

Generalizes to n -argument functions:

$g :: A_0 \times A_1 \times \dots \times A_{n-1} \rightarrow A_n$

Sequence types (lists, arrays)

A *sequence type* over a type α is the set of all ordered tuples whose elements' type is α .

$\text{boolSequence} = \{(), (\text{true}), (\text{false}), (\text{true}, \text{true}), (\text{true}, \text{false}), (\text{false}, \text{false}), \dots\}$

In statically typed languages, all elements of a sequence are of the *same type*; hence

$(\text{true}, \text{false}, 3)$

is a valid *tuple* but **not** a valid *sequence*;

\therefore sequence constructor \neq tuple constructor.

Typical constructors:

cons $:: \text{int} \times \text{int-list} \rightarrow \text{int-list}$

$(:)$ $:: a \rightarrow [a] \rightarrow [a]$

ARRAY [*length*] **OF** *bool*

Typical selectors:

head $:: \text{int-list} \rightarrow \text{int}$; **tail** $:: \text{int-list} \rightarrow \text{int-list}$

$([])$ $:: \text{ARRAY OF } \text{bool} \times \text{int} \rightarrow \text{bool}$

Because sequences are homogeneous, computed indexes applied to sequences are compatible with static type checking:

$[3,4,2,1,9]!!k :: \text{Int}$

Note: A sequence of α can be viewed as a function of type *index-type* $\rightarrow \alpha$:

$\text{ARRAY OF String} :: \text{Int} \rightarrow \text{String}$

Type checking prevents or detects **type errors**, i.e., type inconsistencies between operators and their operands.

Type checking comes in two flavors:

- static
- dynamic

Static type checks

(eager)

- are performed **before execution**
- prevent programs containing potential type errors from running
- necessarily reject some safe programs

Dynamic type checks

(lazy)

- are performed **during execution**
- prevent type errors from occurring (but may halt the program)
- reject operations, not programs (and hence are less restrictive than static checks)

Example— does

if $3 > 2$ **then** 47 **else** 6/**true** (*)

contain a type error?

Type systems are designed

- to **reject unsafe** programs, i.e., programs containing type errors
- to **accept safe** programs, i.e., programs not containing type errors.

Strength and Power

Type systems' **strength**

A type system is

- **strong** if it accepts **no** unsafe programs
- **weak** if it accepts **any** unsafe programs

Although program (*) is safe, most static type systems would reject it.

Strength in a type system is not enough (consider a system that accepts no programs at all).

Type systems' **power**

The more safe programs a type system accepts, the more it is **powerful**.

Is power alone sufficient?

Type checking: Static vs. Dynamic

Static type systems are generally less powerful than dynamic ones (they have less information to work with).

Nevertheless, static type systems are generally preferred; why?

- Once a program has been accepted by a *strong static* type checker, a large and important class of run-time errors has been ruled out *a priori*; an equally strong dynamic system lets them lie in wait indefinitely.
- Performing type checks at compile time improves run-time performance.

Static Type Checking

If the types of variables and functions are all declared, types are checked by applying an **application rule**:

```
Given   f :: t1 -> t2
and     e :: t1
then    f e :: t2
```

Example:

```
Given   not :: Bool -> Bool
        x < y :: Bool
        7 :: Int

conclude not (x < y) :: Bool
        not 7 is a type error (because Bool ≠ Int)
```

Static Type Inference

Although it's good practice, declaring Haskell functions' types is (mostly) optional.

How can a function's type be inferred from its definition?

Informally, ...

```
Suppose   f x = g ... (h x) ...
and       h :: t0 -> t1
          g :: t2 -> t3

conclude  f :: t0 -> t3
```

But this assumes that the types of *g* and *h* are known. These types must either be declared or be inferred from their definitions.

If types are not declared, what keeps the type-inference process from being infinitely recursive?

- Built-in functions have known types, so applying a built-in function to an argument *x* determines *x*'s type.

Example:

`implies p q = if p then q else True`

- `if`'s first argument is `Bool`, so `p :: Bool`
- `if`'s second and third arguments have the same type and `True :: Bool`, so `q :: Bool`
- The type of an `if`-expression is the type of its last two arguments.

Conclusion: `implies :: Bool -> Bool -> Bool`

- If *f* has a parameter to which no functions of known type are applied, then *f* is polymorphic (see below).

Type-system issues

- polymorphism (a.k.a. *parametric* polymorphism)
- overloading (a.k.a. *ad-hoc* polymorphism)
- coercion, conversion, and casting

Polymorphism

A type whose expression contains type variables is called **polymorphic** (Greek: "many-shaped").

Examples:

- *reverse*, *concatenate*, and *length* should be applicable to lists of *any type*
- indexing (`[]`) should be applicable to arrays of *any type*

How do we express these functions' types?

Letting α and β be **type variables**,

`reverse :: α -list \rightarrow α -list`

`length :: α -list \rightarrow integer`

`concatenate :: α -list \times α -list \rightarrow α -list`

`[] :: ARRAY OF β \times int \rightarrow β`

In Haskell,

- type-variable names are lower-case *a, b, c, ...*;
- *α -list* is spelled `[a]`.

[This is **parametric** polymorphism: α and β are *parameters* of the type expressions in which they occur, and can be bound to **any** specific type (*bool*, *int*, *char* \rightarrow *int*, ...).