Data Types

(with examples in Haskell)

COMP 524: Programming Language Concepts
Björn B. Brandenburg

The University of North Carolina at Chapel Hill

Data Types

Hardware-level: only little (if any) data abstraction.

- **Computers operate on fixed-width words** (strings of bits).
  - 8 bits (micro controllers), 16 bit, 32 bits (x86), 64 bits (x86-64, ia64, POWER, SPARC V9).
- Often include ability to address smaller (but not larger) words
  - Intel x86 chips can also address bytes (8 bits) and half-words (16 bits)
- Number, letter, address: all just a sequence of bits.

Pragmatic view.

- Data types define how to **interpret bit strings** of various lengths.
- Allow compiler / runtime system to **detect misuse** (type checking).

Semantical view (greatly simplified; this is an advanced topic in itself)**.**

- A data type is a **set of possible values** (the domain).
- Together with a number of **pre-defined operations**.

Kinds of Data Types

Constructive View

Primitive types.

- A primitive value is **atomic**; the type is “structureless.”
- **Built into the language**.
- Special status in the language.
  - e.g., **literals**, special **syntax**, special **operators**
- Often correspond to elementary **processor capabilities**.
  - E.g., integers, floating point values.

Composite Types.

- Types **constructed from simpler types**.
- Can be defined by users.
- Basis for abstract data types.

Recursive Types.

- Composite types that are (partially) defined in terms of themselves.
- Lists, Trees, etc.

Primitive Types

logic — numbers — letters

Boolean.

- **Explicit type** in most languages.
- In C, booleans are just **integers with a convention**.
  - Zero: False; any other value: True.
- True&False: **literals** or pre-defined **constant symbol**.

In Haskell.

- Type: **`Bool`**.
- Values: **`True`** and **`False`**.
- Functions: **`not`**, **`&&`** (logical and), **`||`** (logical or), …

Primitive Types

logic — numbers — letters

Integers.

- ➡ Every language has them, but designs differ greatly.
- ➡ Size (in bits) and **max**/**min value**.
  - signed vs. unsigned.
- ➡ Use native word size or standardized word size?
  - **Java**: standardized, **portable**, **possibly inefficient**.
  - **C**: native, **portability errors easy to make**, **efficient**.

In Haskell.

- ➡ Type: `Int`.
  - Signed, based on native words, **fast**, size impl.-dependent.
- ➡ Type: `Integer`.
  - Signed, **unlimited size** (no overflow!), **slower**.
  - Sometimes known as **BigNums** in other languages.

Primitive Types

logic — numbers — letters

Intege

➡ Eve

➡ Size

‣ sig

➡ Use

‣ **Ja**

‣ **C**:

Ada Range Types:

(Pascal also has range types.)

```
type Month is range 1..12;
type Day is range 1..31;
type Year is range 1..10000;
```

In Has

➡ Type: `Int`.

‣ Signed, based on native words, **fast**, size impl.-dependent.

➡ Type: `Integer`.

‣ Signed, **unlimited size** (no overflow!), **slower**.

‣ Sometimes known as **BigNums** in other languages.

Primitive Types

logic — numbers — letters

Enumeration Types.

➡ (small) set of related **symbolic constants**.

➡ Compiled to ordinary integer constants.

‣ But much better in terms of readability readability.

➡ Can be **emulated with regular constants** (e.g., classic Java)

‣ But compiler can check for invalid assignments if explicitly declared as an enumeration.

➡ **`enum`** in C, C++.

In Haskell.

➡ Integral part of the language.

➡ Example: `data LetterGrade = A | B | C | D | F`.

Primitive Types

logic — numbers — letters

Floating point.

- ➡ IEEE 754 defines several standard floating point formats.
- ➡ Tradeoff between **size**, **precision**, and **range**.
- ➡ Subject to **rounding**.
- ➡ Not all computers support hardware floating point arithmetic.

In Haskell.

- ➡ Type: `Float`.
  - ‣ Signed, single-precision machine-dependent floating point.
- ➡ Type: `Double`.
  - ‣ Double-precision, double the size.

Primitive Types

logic — numbers — letters

Representing money.

- ➡ Uncontrolled rounding is catastrophic error in the financial industry (small errors add up quickly).
- ➡ **Fixed-point arithmetic**.
- ➡ **Binary-coded decimal** (BCD).
  - Hardware support in some machines.
- ➡ New 128 bit IEE754 floating point formats with **exponent 10 instead of 2**.
  - Allows decimal fractions to be stored without rounding.

In Haskell.

- ➡ Not in the language standard.
- ➡ But you can build your own types (next lecture).
- ➡ Also, can do rounding-free **rational** arithmetic…

Primitive Types

logic — numbers — letters

Rational numbers.

- Store fractions as numerator / denominator pairs.
- Primitive type in some languages (e.g., Scheme).

In Haskell.

- Not primitive.
- Type: `(Integral a) => Rational a`.
  - **Type class** that can be instantiated for either `Int` (native words) or `Integer` (no overflow).
- With a `Rational Integer`, you **never** (!) have to worry about lack of precision or over/underflow.
- (We'll discuss type classes soon…)

Primitive Types

logic — numbers — letters

Characters.

- ➡ Every language has them, but some only implicitly.
- ➡ In legacy C, a character is just an **8-bit** integer.
  - Only **256** letters can be represented (ASCII + formatting).
  - Chinese alone has over **40000** characters…
- ➡ To be relevant, modern languages **must support Unicode**.
  - Full Unicode **codepoint** support **requires 32bit characters**.
  - Java (16bit **char** type) was designed for Unicode, but the Unicode standard was revised and extended…
  - Modern C and C++ support **wide characters**.

In Haskell.

- ➡ Type: `Char`
  - Unicode characters.

Digression: Phaistos Disk

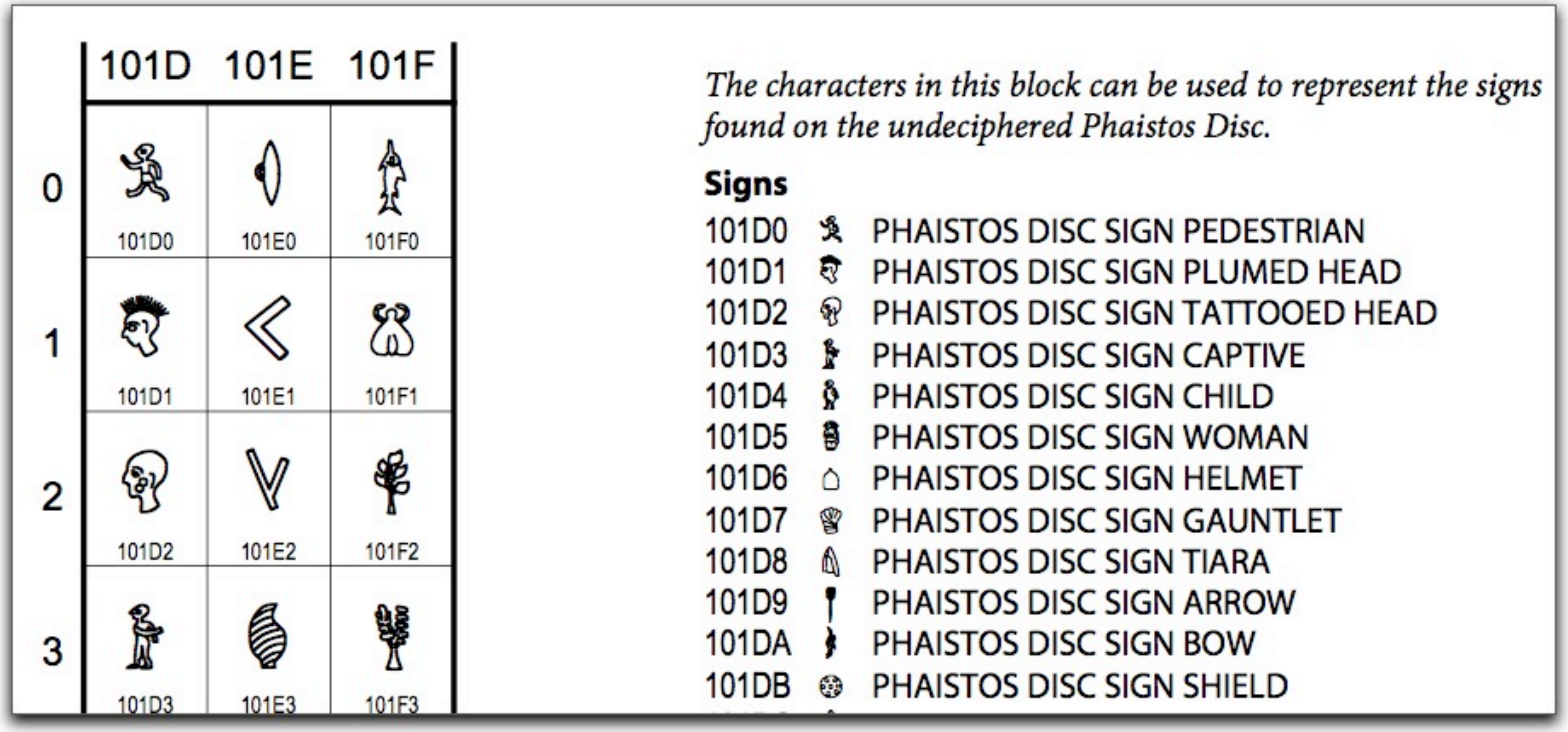
Source: Wikimedia Commons, subject to the GFDL

Nobody knows what it means, but it's in Unicode.

http://unicode.org/charts/PDF/U101D0.pdf

Digression: Phaistos Disk

Source: Wikimedia Commons, subject to the GFDL



	101D	101E	101F
0	101D0	101E0	101F0
1	101D1	101E1	101F1
2	101D2	101E2	101F2
3	101D3	101E3	101F3

The characters in this block can be used to represent the signs found on the undeciphered Phaistos Disc.

Signs

101D0 PHAISTOS DISC SIGN PEDESTRIAN
101D1 PHAISTOS DISC SIGN PLUMED HEAD
101D2 PHAISTOS DISC SIGN TATTOOED HEAD
101D3 PHAISTOS DISC SIGN CAPTIVE
101D4 PHAISTOS DISC SIGN CHILD
101D5 PHAISTOS DISC SIGN WOMAN
101D6 PHAISTOS DISC SIGN HELMET
101D7 PHAISTOS DISC SIGN GAUNTLET
101D8 PHAISTOS DISC SIGN TIARA
101D9 PHAISTOS DISC SIGN ARROW
101DA PHAISTOS DISC SIGN BOW
101DB PHAISTOS DISC SIGN SHIELD

Nobody knows what it means, but it's in Unicode.

http://unicode.org/charts/PDF/U101D0.pdf

Thursday, March 25, 2010

Mapping Types

An relation between two sets.

$$\mathrm{m} : I \mapsto V$$

Mathematical function.

- Maps values from a **domain** to values in a **codomain**.

In programming languages.

- **Array**: maps a set of **integer indices** to values.
  - In practice, integer indices must be consecutive (and often start at 0).
  - This enables **efficient implementations** using offsets.
- **Associative Array**: maps “arbitrary” indices to values.
  - Called **dictionary** in some scripting languages.
  - Usually based on hashing + arrays.
- **Subroutines / functions**: implement **arbitrary** mappings.
  - Each **function signature** defines a type.

Functions in Haskell

$$\mathrm{m} : I \mapsto V$$

```
square :: Integer -> Integer
square x = x * x
```

Named mappings.

➡Type declaration (optional).

➡Defined by **equation**.

Functions in Haskell

$$\mathrm{m} : I \mapsto V$$

```
square :: Integer -> Integer
square x = x * x
```

Named mappings.

➡Type declaration (optional).

➡Defined by **equation**.

Type declaration: type of a symbol defined with **::** “keyword.”
Example: a mapping from Integers to Integers.

Functions in Haskell

$$\mathrm{m} : I \mapsto V$$

```
square :: Integer -> Integer
square x = x * x
```

Named mappings.

➡Type declaration (optional).

➡Defined by **equation**.

Definition: simple **equation** defines the mapping.
"The square of x is given by `x * x`."

Composite Types

types consisting of multiple components

Mathematical foundation.

➡ Recall that each **type is a set of values**.

➡ composite: “one value of each component type”

➡ **Cartesian product**:

$$S \times T = \{(x, y) \mid x \in S \wedge y \in T\}$$

The set of all tuples in which the first element is in S and the second element is in T.

Composite Types

types consisting of multiple components

Mathematical foundation.

➡ Recall that each **type is a set of values**.

➡ composite: “one value of each component type”

➡ **Cartesian product**:

$$S \times T = \{(x, y) \mid x \in S \wedge y \in T\}$$

The set of all tuples in which the first element is in S and the second element is in T.

Example:
Given a 1024x768 pixel display,
each coordinate of the form **(x, y)** is element of the set:

$$\{1, \ldots, 1024\} \times \{1, \ldots, 768\}$$

Composite Types in Programming Languages

History.

- ➡ **Cobol** was the first language to formally use records.
  - ‣ Adopted and generalized by **Algol**.
- ➡ Fortran and LISP historically do not use record definitions.
  - ‣ Classic LISP structures everything using **cons cells** (linked lists).
- ➡ Virtually all modern languages have some means to express structured data.
  - ‣ Basis for **abstract data types** (ADTs)!

Composite types go by many names.

- ➡ C/C++: **struct**
- ➡ Pascal/Ada: **record**
- ➡ Prolog: **structures** (= named tuples)
- ➡ Python: **tuples**
- ➡ Object-orientation: from a data point of view, **classes** also define composite types.
  - ‣ We'll look at OO in depth later.

Composite Types in Haskell (1)

Explicit type declaration.

- ➡Named **type**.
- ➡Named **tuple**.
- ➡**Components** optionally named.

```
-- Implicit fields: only types are given, no explicit names
-- These can be accessed using pattern matching
-- (de-structuring bind).
data Coordinate = Coord2D Int Int

-- Explicit field names.
data Color = RGB { red   :: Int
                 , green :: Int
                 , blue  :: Int
                 }

-- Composite type of composite types.
-- Again, implicit fields.
data Pixel = Pixel Coordinate Color
```

Composite Types in Haskell (1)

data declaration: introduces a type name.

➡ **Components** optionally named.

```
-- Implicit fields: only types are given, no explicit names
-- These can be accessed using pattern matching
-- (de-structuring bind).
data Coordinate = Coord2D Int Int

-- Explicit field names.
data Color = RGB { red   :: Int
                 , green :: Int
                 , blue  :: Int
                 }

-- Composite type of composite types.
-- Again, implicit fields.
data Pixel = Pixel Coordinate Color
```


Thursday, March 25, 2010

Composite Types in Haskell (1)

Ex

➡ **M**

➡ **M**

➡ **Components** optionally named.

named tuple: introduces a **constructor** name.

```
-- Implicit fields: only types are given, no explicit names
-- These can be accessed using pattern matching
-- (de-structuring bind).
data Coordinate = Coord2D Int Int

-- Explicit field names.
data Color = RGB { red   :: Int
                 , green :: Int
                 , blue  :: Int
                 }

-- Composite type of composite types.
-- Again, implicit fields.
data Pixel = Pixel Coordinate Color
```


Thursday, March 25, 2010

Composite Types in Haskell (1)

Explicit type declaration

component names: give each field a meaningful name.

➡ **Components** optionally named.

```
-- Implicit fields: only types are given, no explicit names
-- These can be accessed using pattern matching
-- (de-structuring bind).
data Coordinate = Coord2D Int Int

-- Explicit field names.
data Color = RGB { red   :: Int
                 , green :: Int
                 , blue  :: Int
                 }

-- Composite type of composite types.
-- Again, implicit fields.
data Pixel = Pixel Coordinate Color
```

Composite Types in Haskell (1)

Explicit type declaration

Digression: this would be a good use case for a proper **sub-range type**.

```
-- Implicit fields: only types are given, no explicit names
-- These can be accessed using pattern matching
-- (de-structuring bind).
data Coordinate = Coord2D Int Int

-- Explicit field names.
data Color = RGB { red   :: Int
                 , green :: Int
                 , blue  :: Int
                 }

-- Composite type of composite types.
-- Again, implicit fields.
data Pixel = Pixel Coordinate Color
```

Composite Types in Haskell (2)

Tuples.

- ➡ **Not explicitly introduced** as a type declaration.
- ➡ **Can be used directly** as a type.
- ➡ Can be named using **type synonyms**.

```
stats :: [Double] -> (Double, Double, Double)
stats lst = (maximum lst, average lst, minimum lst)
    where
      average lst = sum lst / fromIntegral (length lst)

type Statistics = (Double, Double, Double)

stats2 :: [Double] -> Statistics
stats2 = stats
```

Composite Types in Haskell (2)

Tuples.

- ➡ **Not explicitly introduced** as a type declaration.
- ➡ **Can be used directly** as a type.
- ➡ Can be named using **type synonyms**.

```
stats :: [Double] -> (Double, Double, Double)
stats lst = (maximum lst, average lst, minimum lst)
    where
      average lst = sum lst / fromIntegral (length lst)

type Statistics = (Double, Double, Double)
```

Type of function:

`stats` **maps** lists of doubles to **3-tuples** of doubles.

$$\text{stats} : \text{ListsOfDoubles} \mapsto \text{Double} \times \text{Double} \times \text{Double}$$

Composite Types in Haskell (2)

Tuples used directly without declaration.

Pragmatic view: **multiple return values**.

```
stats :: [Double] -> (Double, Double, Double)
stats lst = (maximum lst, average lst, minimum lst)
    where
      average lst = sum lst / fromIntegral (length lst)

type Statistics = (Double, Double, Double)

stats2 :: [Double] -> Statistics
stats2 = stats
```

Composite Types in Haskell (2)

Tuples.

➡ **Not explicitly introduced** as a type declaration

Type synonym: optionally named.

```
stats :: [Double] -> (Double, Double, Double)
stats lst = (maximum lst, average lst, minimum lst)
    where
      average lst = sum lst / fromIntegral (length lst)

type Statistics = (Double, Double, Double)

stats2 :: [Double] -> Statistics
stats2 = stats
```

Composite Types in Haskell (2)

Tuples.

- ➡ **Not explicitly introduced** as a type declaration.
- ➡ **Can be used directly** as a type.
- ➡ Can be named using **type synonyms**.

```
stats :: [Double] -> (Double, Double, Double)
stats lst = (maximum lst, average lst, minimum lst)
    where
      average lst = sum lst / fromIntegral (length lst)

type Statistics = (Double, Double, Double)

stats2 :: [Double] -> Statistics
stats2 = stats
```

Type synonym: **equivalent**, but nicer to read.

Disjoint Union

One value, chosen from multiple (disjoint domains).

Mathematical view.

- ➡ Simply a union of all possible types (= sets of values).
- ➡ Each value is **tagged** to tell to which domain it belongs.
  - Tag can be used for checks at runtime.

$$(\{1\} \times S) \cup (\{2\} \times T) = \{(t, x) | \ (t = 1 \wedge x \in S) \vee (t = 2 \wedge y \in T)\}$$

Disjoint Union

One value, chosen from multiple (disjoint domains).

Mathematical view.

- ➡ Simply a union of all possible types (= sets of values).
- ➡ Each value is **tagged** to tell to which domain it belongs.
  - ‣ Tag can be used for checks at runtime.

$$(\{1\} \times S) \cup (\{2\} \times T) = \{(t, x) | \ (t = 1 \wedge x \in S) \vee (t = 2 \wedge y \in T)\}$$

Example:
A **pixel color** can be defined using **RGB** (red, green, blue color channels) or **HSB** (hue, saturation, brightness). Both are simply **three-tuples**, but values must be **distinguished at runtime** in order to be rendered correctly.

Disjoint Union in Haskell

enumeration of named tuples

Algebraic data type.

➡Generalizes enumeration types and composite types.

```
-- Implicit fields: only types are given, no explicit names
-- These can be accessed using pattern matching
-- (de-structuring bind).
data Coordinate = Coord2D Int Int
                | Coord3D Int Int Int

-- Enumeration type.
data ColorName = White | Black | Green | Red | Blue | CarolinaBlue

-- Explicit field names.
data Color = RGB  { red :: Int, green :: Int, blue  :: Int}
           | Named ColorName
           | HSB  { hue :: Double, sat :: Double, bright  :: Double}

-- Composite type of composite types.
-- Again, implicit fields.
data Pixel = Pixel Coordinate Color
```

Disjoint Union in Haskell

enumeration of named tuples

Disjoint Union: enumeration of constructors.

Al

➡Generalizes enumeration types and composite types.

```
-- Implicit fields: only types are given, no explicit names
-- These can be accessed using pattern matching
-- (de-structuring bind).
data Coordinate = Coord2D Int Int
                | Coord3D Int Int Int

-- Enumeration type.
data ColorName = White | Black | Green | Red | Blue | CarolinaBlue

-- Explicit field names.
data Color = RGB  { red :: Int, green :: Int, blue  :: Int}
           | Named ColorName
           | HSB  { hue :: Double, sat :: Double, bright  :: Double}

-- Composite type of composite types.
-- Again, implicit fields.
data Pixel = Pixel Coordinate Color
```

Disjoint Union in Haskell

enumeration of named tuples

Algebraic data type.

➡Generalizes enumeration types and composite types.

Data types of sub-domains can be **heterogenous**.

```
-- (de-structuring bind).
data Coordinate = Coord2D Int Int
                | Coord3D Int Int Int

-- Enumeration type.
data ColorName = White | Black | Green | Red | Blue | CarolinaBlue

-- Explicit field names.
data Color = RGB  { red :: Int, green :: Int, blue  :: Int}
           | Named ColorName
           | HSB  { hue :: Double, sat :: Double, bright  :: Double}

-- Composite type of composite types.
-- Again, implicit fields.
data Pixel = Pixel Coordinate Color
```

Recursive Types

types defined in terms of themselves

Classic example: List.

- ➡ defined as a **head** (some value) and a **tail** (which is a list).
- ➡ Semantical view: infinite set of values.
  - ‣ **Rigorous** treatment of the **semantics** of recursive types is **non-trivial**.

Implementation.

- ➡ Requires **pointers** (abstraction of addresses) or **references** (abstraction of object location).
  - ‣ Pointer arithmetic: calculate new addresses based on new ones.
  - ‣ No arithmetic on references.
- ➡ References **not necessarily exposed** in programming language.
  - ‣ e.g., Haskell does not have a reference type!
- ➡ However, references must be exposed to construct **cyclical data structures**.

Recursive Types in Haskell

```
data IntList = EndOfList
             | Link { elem :: Int, tail :: IntList }
```

Algebraic type with self-reference.

- ➡Can use name of type in definition of type.
- ➡However, no explicit references.
  - ‣No doubly-linked lists!
- ➡Haskell has generic built-in lists…

Recursive Types in Haskell

```
data IntList = EndOfList
             | Link { elem :: Int, tail :: IntList }
```

Algebraic type with self-reference.

- ➡Can use name of type in definition of type.
- ➡However, no explicit references.
  - ‣No doubly-linked lists!
- ➡Haskell has generic built-in lists…

Type that is being defined is used in definition.

What are Strings?

Character sequences.

- Is it a **primitive type**?
  - Most languages support string **literals**.
- Is it a **composite type**?
  - Array of characters (e.g., C).
  - Object type?
- Is it a **recursive type**?
  - sequence = list (e.g., Prolog).

In Haskell.

- `type  String  =  [Char]`
- Strings are simply lists of characters.
  - A **type synonym**, both ways of referring to the type can be used interchangeably.

What are Strings?

Character sequences.

- Is it a **primitive type**?
  - Most languages support string **literals**.
- Is it a **composite type**?
  - Array of characters (e.g., C).
  - Object type?
- Is it a **recursive**
  - sequence = list

In Haskell.

- `type String`
- Strings are simpl
  - A **type synony**
    used interchang

Bottom line: **No Consensus**

No approach to treating strings has been universally accepted; each approach has certain advantages.