

## Class 32: Types (2)

**Held:** Wednesday, April 18, 2007

**Summary:** Today we continue our exploration of types, grounding our exploration in the consideration of sets and in some of the issues in the design of CLU.

### Related Pages:

- EBoard.
- Reading: Barbara Liskov et al. - Abstraction Mechanisms in CLU.

### Overview:

- Types, Revisited.
- Types as Sets.
- Lessons from CLU.
- References, Pointers, and Variables.

## Types, Revisited

- Recall that we've been talking about types and type systems.
- We've found that there are (at least) four ways to think about types:
  - A type describes a set of values.
  - A type describes a collection of operations that can be performed on the values of the type.
  - A type describes restrictions on the parameters to a function.
  - A type provides a translation of the underlying bits.
- We've also talked about the kinds of primitive types available.
- Today, we'll reflect on a variety of extensions of these concepts.
  - How might you combine types?
  - How might users define their own types?
  - What does the CLU reading do to inform our understanding of types?

## Type Constructors

- When building new data types from previously defined data types, we need “things” that join the other types together. That is, we want ways to *construct* new data types.
- In other words, *what constructors can be used to create new types?*
  - As language designers, we might also ask *what subset of these constructors do we permit our programmers to use?*
- Surprisingly, different language theorists appear present different “principal” type constructors.
- There are, however, three basics, which derive from the notion of type as set.
  - Product

- Function
- Sequence.
- The *product* constructor, often represented with an  $\times$ , takes two types and creates a type which represents a set of ordered pairs
  - The first element of each pair belongs to the first type.
  - The second element of each pair belongs to the second type.
  - Formally,  $A \times B = \{ (a,b) \text{ s.t. } a \text{ in } A, b \text{ in } B \}$
- The *function* constructor, often represented with an arrow, takes two types and creates a type which represents a map.
  - When an element of a function type is applied to an element of its domain, the resulting object is an element of the range.
  - Formally,  $A \rightarrow B = \{ f \text{ s.t. } a \text{ in } A \Rightarrow f(a) \text{ in } B \}$
- The sequence constructor takes one type and creates a set of all tuples that can be formed from elements of the base type.
  - Like the Kleene star, it is usually written with a  $*$ .
  - Formally,  $A^* = \{ (a_1, \dots, a_n) \mid n \geq 0; \text{ for all } 1 \leq i \leq n, a_i \text{ in } A \}$
- Let's consider the type constructors in Pascal/Java/whatever. What are the primary type constructors these languages provide? Which of the previous constructors do they correspond to?
  - Records
  - Arrays
  - Functions
  - Lists (are these a type?) ...
- Are there other constructors? Certainly. Other constructors include
  - Subset
  - Power set
  - Pointer (!?!?)
  - Union
- How might we formally define these?

## CLU

- What does the year in which the article was published (1977) tell you about the context in which the article was written?
- What is the thesis of the CLU reading?
- What seem to be the more interesting type features of CLU?
- Are there type features that you are surprised have not been adopted (or adopted as quickly) in more modern languages?

## References, Pointers, and Variables

- In many programming languages, we have another kind of “type constructor” used to build types: the pointer/reference.
- For example, elements of the “pointer to an integer” type support:
  - *Dereference*: get the integer.

- *Reference*: point to a different integer.
- Note that we might want to make a similar distinction between “integer” and “integer variable”.
  - The latter supports *assign* while the former does not.
  - This is a somewhat subtle distinction that many have trouble making. However, failure to make this distinction can lead to situations in which you treat integer constants as variables, leading to very strange results. (Yes, there were languages in which you could change the value of 2.)