Programming Languages (CS302 2007S)

Class 31: Types (1)

Held: Monday, April 16, 2007

Summary: Today we begin our exploration of types and type issues in programming language design.

Related Pages:

- EBoard.

Overview:

- An Introduction to Types.
- Simple Data Types.
- Type Constructors.

An Introduction to Types

- We've talked a lot about types this semester. This week, we consider them in a bit more depth.
- Types are fundamental to most programming languages, from imperative to declarative.
- Types provide a way to abstract away from the underlying architecture or representation for values. Recall that such *abstraction* is a key component of good language design.
- We all have an implicit understanding of what a type is, but how might we describe them to someone who was learning about them for the first time.
  - A type is a name for *a set of values*.
  - A type is a name for *a collection of restrictions on values*. That is, a type specifies things you cannot do (you can't find the nth element of an integer, you can't usually add two arrays).
  - A type is a name for *things that support certain operations*. For example, integers are things you can add, subtract, ...
  - A type is a *high-level description as to how data are used* .
  - ...
- We might also ask why we use types.
  - Types make a program easier to read, understand, modify, design.
  - Types can be used to help ensure correctness.
  - Types can be used to restrict how values are used
    - Don't use that `int` as a pointer!
    - Don't use that real as an integer!
    - Don't add that day-of-the-week to that temperature!
  - Some colleagues who use Haskell (a typeful functional language) claim that once you get the types right, the rest of the program often falls out naturally.
- There is often a gap between the abstract and implemented types.
  - For example, integers in many programming languages are restricted to a certain range.
  - Similarly, floating point numbers have varying degrees of precision (which may be hard for the

novice programmer to predict).

- How do we (language designers, programmers) describe types:
  - Limited to predefined types (some called “Simple Types”)
  - Permit description of new types by listing elements.
  - Permit description of new types by composing previous types (with a *type constructor*).
- Traditionally, we *name* types in addition to defining them.

Simple Data Types

- It’s clear that when defining types, you need to start somewhere.
- At the basis of most type systems are a collection of data types. These types are often called the *simple*, *basic*, or *primitive* data types.
- The primitive types are predefined by the language.
- They are often used to create more complex types.
- Note that not all predefined types are primitive/simple types.
- *How do we differentiate the primitive/simple and predefined types?* We say a data type is primitive/simple if it has not structure other than an inherent arithmetical or sequential structure.
- Some of the traditional simple types:
  - Integers
  - Floating-point numbers
  - Characters
  - Boolean Values
- *Is it possible for programmers to define primitive types?* In some languages, the answer is clearly yes.
  - Enumerated types: defined by listing valid values. This gives an implicit ordering.
  - Subrange types: defined as a subrange of a given enumerated type.
- *Why would we want to define these kinds of types?*
- *How do enumerated types differ from a series of constant declarations?*

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 x, 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 x B = { (a,b) s.t. a in A, b 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 -> B = { f s.t. a in A => f(a) 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* = { $(a_1,...,a_n)$ | n >= 0; for all 1 <= i <= n, a_i 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?

Arrays

- Let's think again about arrays. What set operation are arrays most like?
- We'll note that there are at least two "kinds" of arrays.
    - Pascal-like arrays, in which you must specify the indices when you declare the array.
    - Java-like arrays, in which you do not need to specify the indices/size.
- We may find that each corresponds to a different type constructor.