CPSC320 Class - October 25/07

Robert Pringle

October 24, 2007

Type Construction

- Type constructors are used by programming languages to construct complex types out of **simple types** to create **user defined types**.
- Type construction can be represented mathematically through domain equations for the given type over the sets for the types that where used to create a given user-defined type.
- This can include such things as cartesian products, union, powersets, function mapping or subsets for subranges.
- It is possible (and likely) to have types whose domain equations require a combination of the above operations.

Union

- The union operation is usually used to represent types that use type discrimination, such as C unions or ADA variant records.
- The actual element associated with a particular instance of this type can only be one of the type options in the structure as result we use the union form a set of the possible values resulting from the usage of this type.

Union Domain Equation Example

The C union type specified below is an example of a data type whose domain equation would utilize unions. For the following we will define the set of all integers as **I**, characters as **C** and floats as **F**.

```
union Example {
     int   integer;
     char  character;
     float floatingpoint;
}
```

For the given union type we have a domain equation $Example = I \cup C \cup F$.

Subset

- The subset operation can be used to represent constructs that can only handle some of the elements of the type they where derived from.
- An example of this is subranges in ADA.

Subset Domain Equation Example

The ADA subrange given below is an example of a datatype whose domain equation can utilize types. For the following we will define the set of all integers as **I**.

```
subtype Example is integer range -10..10;
```

For the given subrange type we have a domain equation
$Example = \{i \epsilon I | -10 \leq i \leq 10\}$

Function Mappings

- Function mappings can be used to represent any construct that produces results/references from arguments.
- Actual functions that take arguments and produce a result or hash maps that take a key and use it to retrieve values are examples of constructs that can be represented by function mapping.
- Arrays can also be represented by domain equations using function mapping as they take an index (usually a integer) to retrieve a value stored at that index.
  - Arrays can be defined with a fixed size on the stack as in C or dynamic sizes using dynamic allocation on the heap as in JAVA or the stack as in ADA.

Mapping Domain Equation Example

The C floating point array specified below is an example of a data type whose domain equation would utilize mapping. For the following we will define the set of all integers as **I** and floats as **F**.

```
float Example[10];
```

For the above array we have the domain equation $Example = I \rightarrow F$. Note in this case we do not bound the range for the integers that are used to index the array as you can reference past the end of the array (though this is not recommended).

Pointers and Recursive Types

- For pointers and recursive types we are also dealing with the memory of the system in which the particular type is being used and as such there is direct mathematical operation to represent these.
- However it is possible to take a simplified view of the memory and the recursive and represent them using the function mapping we have previously seen.
- The full domain of a pointer would include the set of all addresses that refer to the types compatible with the pointer (those that can be dereferenced from the pointer).
- It is also possible to represent references, which are pointers under the control of system rather than the programmer (such as JAVA).
- Pointers can be used to define recursive types, types that use themselves in their declaration.

Mixed Domain Equation Example

In order to show a structure that utilizing a mixture of operations in its domain equation we will use a custom C structure used to represent a tree node with nodes allocated dynamically through pointers and leaves terminated with null children. For the following we will define the set of all integers as **I**, characters as **C** and floats as **F**.

```
struct Example {
     int I;
     union { char C; float F; } content;
     Example* lchild, rchild;
}
```

The domain equation for the above user-defined type is
$Example = I \times (C \cup F) \times (Example \times Example \cup Example \cup \{\epsilon\})$.

Type Equivalence

- Type equivalence is concerned with whether two types can be considered the same or not.
- The two primary methods used to determine type equivalence are structural equivalance, name equivalence and declaration equivalence.
- In structural equivalence two types are considered the same if they share the same structure.
  - For user-defined types that utilize other non-simple types structural equivalence can be determined by replacing a typename with its structure though this becomes problematic for recursive types.
- In name equivalence two types are considered the same only if they share the same name.
  - How name equivalence is applied to anonymous types is not always clear and can be implementation dependant.
- Declaration equivalence is where types constructed from another types (subrange, derived classes, etc.) are considered equivalent to the base type.

Type Equivalence Example

Consider the two C structs below where the structure of the undefined types **TypeThree** and **TypeFour** are considered equivalent. These would be considered equal by structural equivalence however they would not be considered equal by name or declaration equivalence.

```
struct TypeOne {
     int       I;
     double    D;
     TypeThree type;
}
```

```
struct TypeTwo {
      int      I;
      double   D;
      TypeFour type;
}
```

Type Equivalence Example

Consider the two C structs below where the structure of the undefined types **TypeThree** and **TypeFour** are considered equivalent. These would be considered equal by structural equivalence however they would not be considered equal by name or declaration equivalence.

```
struct TypeOne {
     int       I;
     double    D;
     TypeThree type;
}
```

```
struct TypeTwo {
     int      I;
     double   D;
     TypeFour type;
}
```

Type Equivalence Example

Consider the two C typedefs below. Both of these typedefs would be considered equivalent by structural equivalence however they would not be considered the equivalent by name or declaration equivalence. When comparing both types to float they both be considered equivalent to floats by strutural and declaration equivalence but not by name.

```
typedef float TypeOne;
typedef float TypeTwo;
```