More Composite Types and the Subtype Relation

CS 329 Lecture 4

August 2, 2007

Rushikesh K Joshi

Recursive Types: Lists

Some types are defined recursively in order to express the types in terms of closed expressions even if there are infinitely many possible values for them.

For example,
a list L of elements of type T:
L = either NULL or T X L
or in other words,
L = NULL + (T X L), where + defines a disjoint union

The set defining the list type contains all possible lists of type T, but we have a closed recursive expression for the list type L.

An example list value

L = abcdec

T = {a,b,c,d,e,f,....}

The above value can be shown to be a valid value of list type by constructing a terminating recursive expansion for the value as given below:

L = a X L
X b X L
X c X L
X d X L
X e X L
X c X L
X NULL

Disjoint Union Type

```
Union U {
        int i;
        char c;
}
```

U = int + char

i.e.
U = *either* int *or* char

A value of type U is either a value of type int or a value of type char.

The union type (either/or) was used in the definition of the list type.

example: A union type defined in C

When can a value of type T1 be safely treated as a value of type T2?

Firstly, if T1 and T2 are the same types, there is no problem. For example as in the below program:: int i; int j; ... i = j;

Further, if T1 and T2 are not the same types, we may still be able to treat ALL values of T1 as values of T2 provided that there is some relation between the two types. What's that relation?

Subtype Relation

S <: T

we say that type S is a subtype of type T

For primitive types, a subset can be considered as a subtype.

Exmples:
R1 = {1,2,3,4}
R1 <: Int
R2 = {a,b,c,D,E}
R2 <: Char

What can we do with subtypes?

We can use a value of a subtype wherever a value of the (super)type is expected. This is stated by the below rule of subsumption.

Subsumption Rule

$$\frac{t:S, \quad S<:T}{t:T}$$

The rule states that:
if value t is of type S and S is given as a subtype of type T,
then value t is also a value of type T.

Subtype relation for primitive types

For primitive types, subset is subtype.

e.g. S={1,2,3}, T={1,2,3,4}, S<:T

wherever value of a type is expected, a value from the subtype will work safely.

i.e. a call to function

f(T val) {.....}

will work correctly with any value of type S sent as a parameter, since all values of type S happen to be valid values of type T.

However, subtype relation is not symmetric.
For example, the below function will not work correctly for all values of type T when sent as input parameter to f().

f(S val) { A[3]; return A[val];} For which case does it not work?

Subtype Relation for Product types: The width rule

R1 = T1 X T2
R2 = T1 X T2 X T3

R2 can be considered as a subtype of R1

why? because a value of type R2 can be easily considered as a value of R1 by ignoring the T3 component in it.
Example:

R1 = RollNo X Name
R2 = RollNo X Name X Age

Subtype Relation for Product types: The depth rule

R1 = T1 X T2
R2 = S1 X S2

R2 can be considered as a subtype of R1, when S1 <: T1 and S2 <: T2

R1 = String X String
R2 = RollNo X Name

Subtype Relation for Product types: The combined rule

R1 = T1 X T2
R2 = S1 X S2 X S3

R2 can be considered as a subtype of R1, when S1 <: T1 and S2 <: T2

R1 = String X String
R2 = RollNo X Name X Age

Subtype Relation for Product types: Record Permutation Rule

R1 = T1 X T2
R2 = T2 X T1

R2 can be considered as a subtype of R1, and vice versa by the record permutation rule.

R1 = Name X RollNo
R2 = RollNo X Name

R1 <: R2, and R2<:R1

The rule is at conceptual level, and it's implementation in a programming language may requires manipulating with the memory layouts for correct implementation of the rule.

Properties of subtype relation

Reflexive 

Symmetric 

Anti-symmetric 

transitive