0.1 Types as behavior

In the previous chapter we have developed a formal definition of types and the subtyping relation. However, we have restricted ourselves to (syntactic) signatures only, omitting (semantic) behavioral properties associated with function and object types.

0-1

Subtype requirements – ***signature and behavior***

- preservation of behavioral properties

Safety properties – ***nothing bad***

- invariant properties – *true of all states*
- history properties – *true of all execution sequences*

Slide 0-1: Subtyping and behavior

From a behavioral perspective, the subtype requirements (implied by the substitutability property) may be stated abstractly as *the preservation of behavioral properties*. According to [Liskov93], behavioral properties encompass *safety properties* (which express that nothing bad will happen) and *liveness properties* (which express that eventually something good will happen). For safety properties we may further make a distinction between *invariant properties* (which must be satisfied in all possible states) and *history properties* (which hold for all possible execution sequences). See slide 0-1.

Behavioral properties (which are generally not captured by the signature only) may be important for the correct execution of a program. For example, when we replace a *stack* by a *queue* (which both have the same signature if we rename *push* and *insert* into *put*, and *pop* and *retrieve* into *get*) then we will get incorrect results when our program depends upon the *LIFO* (*last-in first-out*) behavior of the stack.

As another example, consider the relation between a type *FatSet* (which supports the methods *insert*, *select* and *size*) and a type *IntSet* (which supports the methods *insert*, *delete*, *select* and *size*). See slide 0-2.

With respect to its signature, *IntSet* merely extends *FatSet* with a *delete* method and hence could be regarded as a subtype of *FatSet*. However, consider the history property stated above, which says that for any (*FatSet*) s, when an integer x is an element of s in state ϕ then x will also be an element of s in any state ψ that comes after ϕ. This property holds since instances of *FatSet* do not have a method *delete* by which elements can be removed. Now if we take this property into account, *IntSet* may not be regarded as a subtype of *FatSet*, since instances of *IntSet* may grow and shrink and hence do not respect the *FatSet* history property.

This observation raises two questions. Firstly, how can we characterize the behavior of an object or function and, more importantly, how can we extend our