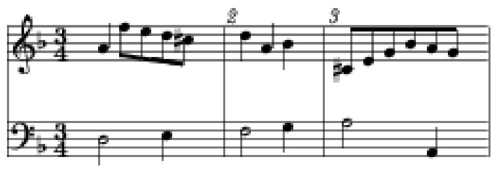
1

Behavioral refinement



Ultimately, types are meant to specify behavior in an abstract way. To capture behavioral properties, we will generalize our notion of *types as constraints* to include behavioral descriptions in the form of logical assertions.

Behavioral refinement $\boxed{10}$ *1-1*

- types as behavior
- verification
- abstraction and representation
- behavioral compositions

Additional keywords and phrases: *behavioral subtypes, state transformers, correctness formulae, assertion logic, transition systems, invariants, formal specification*

Slide 1-1: Behavioral refinement

In this chapter we will explore the notion of behavioral (sub)types. First we characterize the trade-offs between statically imposed (typing) constraints and dynamic constraints resulting from the specification of behavioral properties. We will provide a brief introduction to the assertion logic underlying the verification of behavioral constraints. Also, we look at how we may characterize the behavior of object-based systems in a mathematical way. Then we will describe the duality between abstraction and representation in defining behavioral subtypes that define concrete realizations of abstract specifications. In particular, we specify the correspondence requirements for behavioral subtypes. We will conclude this chapter by discussing the problems involved in specifying behavioral compositions, and explore what specification techniques are available to model the behavior of object-based systems.

1.1 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.

1-2

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 1-2: 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 1-2.

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 1-3.

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