Advertisement

There will be no lectures next week. However, you are strongly encouraged to attend the following talk:

Synergistic Language Design and Implementation Research
Craig Chambers, University of Washington
Monday, 4/15/96, in 405 Soda

and at the end of the semester, especially for those interested in parallelism.

ZPL: A High Performance Scientific Programming Language
Larry Snyder, University of Washington
Monday, 5/6/96, in 405 Soda

Subtyping

Recall that subtypes are used to allow broaden assignment statements and parameter passing:

```
x: T := E
```

is legal if the type of `E` is a subtype of `T`.

Similarly, if function `f` has given signature, then the function call is legal if the type of `E` is a subtype of `T`.

```
f: T -> T'
f(E)
```

One expects that within `f` or after assignment to `x`, the program will be correct if the object behaves like an object of type `T`.

What's Wrong with Syntactic Subtyping?

Recall the contra/covariance rules:

$$\frac{\forall i\ (T_i \sqsubseteq S_i)\ ;\ S_{body} \sqsubseteq T_{body}}{(\texttt{->}\ \ (S_1\ S_n)\ S_{body}) \sqsubseteq (\texttt{->}\ \ (T_1\ T_n)\ T_{body})}\ [\texttt{->}\ - \sqsubseteq]$$

In particular, this says that two functions with identical type signatures can be used in place of one another.

```
class Stack {
   put (T e)
   T get()
}
```

```
class Queue {
   put (T e)
   T get()
}
```

`Stack` is a subtype of `Queue` and *vice-versa*. But it is unlikely that code written for a `Stack` will work correctly when used on a `Queue`.

Behavioral Subtype Requirement

Requirement: Let $\phi(x)$ be a property provable about all objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.

- Within a particular program, calls made to methods assuming an object is a member of the supertype must continue to work as expected if the object is a member of a subtype.
- Properties independent of any particular program must be preserved.

Properties

What kinds of properties are we thinking of preserving?

- Safety properties: nothing "bad" happens.
- Liveness properties: something "good" eventually happens.

A example of a safety property is $x > 0$. An example of a liveness property is p *terminates*. More generally, within concurrent programs a liveness property may be: if x happens then eventually y happens.

More formally, given a program execution, $S = s_1, s_2, \ldots,$ in which each s_i is a state [Lamport and Abadi]:

- If a safety property *phi* does not hold on S, then there exists some finite prefix $P = s_1, s_2, \ldots s_i$ such that ϕ does not hold on P.
- If a liveness property *theta* does not hold on an infinite sequent S, it may still hold on every finite prefix of S.

For now, we will focus on safety properties.

Safety Properties

Within the domain of safety properties, we can distinguish between two classes:

- An *invariant* is a property that is true of all states.
- A *history* property is a property true of all sequences of states.

An invariant is formulated as a predicate on individual states and a history property as a predicate on pairs of states.
Example on a bounded queue type:

- Invariant: the size of the queue is always less than or equal to its bound.
- History: the bound is constant.

Specifications

Overview of [Liskov,Wing]:

- Give two definitions of a subtyping relation that satisfies the subtype requirement.
- Apply even to concurrent environments.
- Constructive approach: the subtype properties can be proven with a small set of lemmas.
- Provides a way of specifying types.
- Identifies the notion of *type families.*

Example: Bounded Bag

A `BoundedBag` has two operations:

- `put (T e)`
  - precondition: the size of the bag is less than the bound.
  - postcondition: inserts the element `e` into the bag.
- `T get ()`
  - precondition: the bag is not empty.
  - postcondition: removes an arbitrary element from the bag and returns it.

Example: Bounded Stack

A `BoundedStack` has three operations:

- `push (T e)`
  - precondition: the size of the stack is less than the bound.
  - postcondition: pushes `e` onto the stack.
- `T pop()`
  - precondition: the stack is not empty.
  - postcondition: removes the top element from the stack and returns it.
- `T swap_top(T e)`
  - precondition: the stack is not empty.
  - postcondition: replaces the top of the stack with `e`.

Subtyping Between Stack and Bag

Intuitively, a `BoundedStack` is a subtype of a `BoundedBag`.

To prove this, we would define an abstraction function that maps any `BoundedStack` object to a `BoundedBag` object.

```
Bag:
  []  --put(1)--> [1]   --put(2)--> [2,1] --get()--> [1]
   ^               ^                  ^   -> 2        ^
   |               |                  |        Abstraction
Stack:                                                |
  [] --push(1)--> [1]  --push(2)--> [1,2] --pop()--> [1]
                                          -> 2
```

- The only visible effects are the return results.
- Because the specification of `get` allows any element to be chosen, a legal execution is one in which the stack top is chosen.
- Because the preconditions on Stack match those of the Bag, there are no legal executions of Stack that cannot map to an execution of Bag.

May Subtypes Have Additional Methods

What about `swap_top`?

Most definitions of subtyping allow the subtype to have methods that are not represented in the supertype.

- In isolation, any program that uses a subtype in place of a supertype will not use these "extra" methods.
- In an environment with aliasing, some other parts of the program (which recognize the subtype as itself) may use them.

```
T2 is a subtype of T1
f (x T1, y T2) { ...}
z: T2
f (z, z)
```

- In a concurrent environment, similar problem arise.

```
user1 inserts <"b1", abag>
user2 inserts <"b2", astack>
user2 uses astack as a stack, but user1 assumes a bag
```

Modifying Specifications for Subtyping

For each type, there is an **invariant** clause that states what invariants must be satisfied on all subtypes of the type.

E.g., 0 <= size < bound

There are two different notions of subtype that deal differently with history properties:

- Add a **constraint** clause to the supertype that captures the history properties that must be preserved by any subtype. May use two different constraints, as long as subtype's implies supertype's.

  E.g., bound is constant

- For each extra method in a subtype, given an **explanation** of how its behavior could be "simulated" using methods defined on the supertype.

  E.g., `s.swap_top(i) = s.pop(); s.push(i)`

Model of Computation

Obj = set of all possible objects, partitioned by types

$State = Env \times Store$
$Env = Var \to Obj$
$Store = Obj \to Val$

Given a variable x and state ρ, x_ρ is the value of the object named by x.

A type is modeled as: $\langle O, V, M \rangle$, where:

- $O \subseteq Obj$
- $V \subseteq Val$
- M is a set of methods
- Each method is either a *constructor*, *observer* or *mutator*

A *computation* is a sequence of alternating states and transitions:

$$\rho_0 \; Tr_1 \; \rho_1 \ldots \; \rho_{n-1} \; Tr_n \; \rho_n$$

Specifications

A type specification includes:

- The type's name
- Description of the value space
- for each method, a name, signature, and pre/post conditions.

Note: *creators* are omitted, to allow multiple implementations of a type. Because they are not included, type induction cannot be used.

Example Specification

bag = **type**
uses BBag (bag **for** B)
for all b : bag

`put` = **proc** (i: int)
requires $|b_{pre}.elems| < b_{pre}.bound$
modifies b
ensures $b_{post}.elems = b_{pre}.elems \cup \{i\}$
$\wedge\ b_{post}.bound = b_{pre}.bound$

`get` = **proc** () **returns** res: int
requires $b_{pre}.elems \neq \{\}$
modifies b
ensures $b_{post}.elems = b_{pre}.elems - \{res\}$
$\wedge\ res \in b_{pre}.elems$
$\wedge\ b_{post}.bound = b_{pre}.bound$

`card` = **proc** () returns (res: int)
ensures $res = |bpre.elems|$

`equal` = **proc** (a: bag) returns (res: bool)
ensures $res = (a = b)$

Adding Invariants

To the *bag* specification, we add:

invariant $|b_p.elems| \leq b_p.bound$

This is shorthand for the predicate: For all computations c and all states p in c:

$$\forall x \; : \; \tau x \in dom(p) \Rightarrow \phi(x_p)$$

All creators for a type τ must establish τ's invariant.

For each creator for type τ show that for all

$$x \; : \; \tau, \; Invariant_{\tau}[result_{post}/x_p]$$

History: Constraint Rule (approach 1)

Add to `bag` specification:

constraint $b_\rho.bound = b\psi.bound$

The specification for `Stack` explicitly names its supertype:

stack = **type**
uses BStack (stack for S)
for all s: stack

invariant $length(s_\rho.items) \leq s_\rho.limit$

$\ldots$

subtype of bag (`push` for `put`, `pop` for `get`, `height` for `card`)

$\forall st \ : \ S.A(st) = \langle mk\_elems(st.items), stl.limit\rangle$

where $mk\_elems \ : \ Seq \rightarrow M$

$\forall i \ : \ Int, sq : Seq$

$mk\_elems([]) = \{\}$

$mk\_elems(sq||[i]) = mk\_elems(sq) \cup \{i\}$

History Rule

History Rule: For each of the i mutators m of τ, for all $x : \tau$:

$$\frac{m_i.pre \wedge m_i.post \Rightarrow \phi[x_{pre}/x_\rho, x_{post}/x_\psi]}{\phi}$$

Definition of the subtype relation:

1. The abstraction function respect invariants

2. Subtype methods presever teh supertypes methods' behavior.

3. Subtype constraints ensure supertype constraints

Extension Map (2nd Approach)

With the constraint approach, users cannot use the history rule to deduce history properties.
Definition of the subtype relation:

1. The abstraction function respect invariants

2. Subtype methods presever teh supertypes methods' behavior.

3. The extension map, E, must e define for each mthod m.