Subtyping and the Liskov Substitution Principle

March 19, 2008

Syntactic and Semantic Properties

Syntactic Properties

Program properties program defined by “simple” formal rules, and automatically checked.

- Are all parentheses matched?
- Does the program contain type errors?
- Are there are undefined variables?

Semantic Properties

Program properties defined with arbitrary reasoning.

- Do these methods calculate the same function?
- What is the asymptotic complexity of this method?
- Does this loop satisfy an invariant?

Syntactic and Semantic Properties

Syntactic Properties

Program properties program defined by “simple” formal rules, and automatically checked.

- Are all parentheses matched?
- Does the program contain type errors?
- Are there are undefined variables?
- Is class A derived from class B?

Semantic Properties

Program properties defined with arbitrary reasoning.

- Do these methods calculate the same function?
- What is the asymptotic complexity of this method?
- Does this loop satisfy an invariant?
- Is it safe to treat an instance of A like a B?

Subtyping

Say A is a *subtype* of B when an instance of A can be used in place of an instance B.

Example

```
class Math {
  virtual int Add(int x,int y){ return x + y; } }

// More math is a subtype of Math
class MoreMath : Math {
  virtual int Divide(int x, int y){ return (x/y); }

// What is EvenMoreMath a subtype of?
class EvenMoreMath : MoreMath {
  override int Add(int x,int y){ return 399; } }
```

Two views of subtyping

- Syntactic view: A `is_a` B when A extends or implements B.
  - Ensures type safety: Anyone expecting a B will find A has the appropriate members.
  - Relatively easy to check.
  - But does not ensure programs will work correctly.
- Semantic view: A `<:` B when instances of A exhibit behavior equivalent to instances of B in places where Bs are expected.
  - A `is_a` B necessary condition for A `<:` B
  - Rules out many errors possible with only syntactic subtyping.
  - But impossible to enforce automatically.

Back to the even more EvenMoreMath example

- EvenMoreMath is a syntactic subtype of Math and MoreMath.
- EvenMoreMath is a semantic subtype of Math.
- Is EvenMoreMath a semantic subtype of MoreMath?.
  - Depends on the specification of MoreMath and EvenMoreMath. . .

Method specification subtyping

Before working out class subtyping, need to figure out when method specifications can be considered subtypes.

Use functional specifications (Feb. 20 Lecture) to help determine subtyping relation.

Key Idea: m $<:$ n when m

- accepts more inputs than n
- produces fewer potential outputs than n
- otherwise obeys the specification of n.

Subtyping and method results

When is m <: n?

```
//returns:  An assumption about the returned S
S m(X in)

//returns:  An assumption about the returned T
T n(X in)
```

If some program expects an n, but is given an m, things will be ok when both:

Subtyping and method results

When is m <: n?

```
//returns:  An assumption about the returned S
S m(X in)

//returns:  An assumption about the returned T
T n(X in)
```

If some program expects an n, but is given an m, things will be ok when both:

- S <: T, and
- assumptions about the returned S are *stronger* (i.e. more specific, or more restrictive) than assumptions about the returned T.

Example

Assume W <: U

```
// a() returns some W
W a(X in)

// b() returns some W != null
W b(X in)

// c() returns some U
U c(X in)

// d() returns some U != null
U d(X in)
```

What subtypes are here?

Example

Assume W <: U

```
// a() returns some W
W a(X in)

// b() returns some W != null
W b(X in)

// c() returns some U
U c(X in)

// d() returns some U != null
U d(X in)
```

What subtypes are here? b <: a, a <: c, b <: d, d <: c...
but a and d are not subtypes of each other.

Subtyping and method inputs

When is m <: n?

```
//requires:  An assumption about the X input
T m(X in)

//requires:  An assumption about the Y input
T n(Y in)
```

If some program expects an n, but is given an m, things will be ok when:

Subtyping and method inputs

When is m <: n?

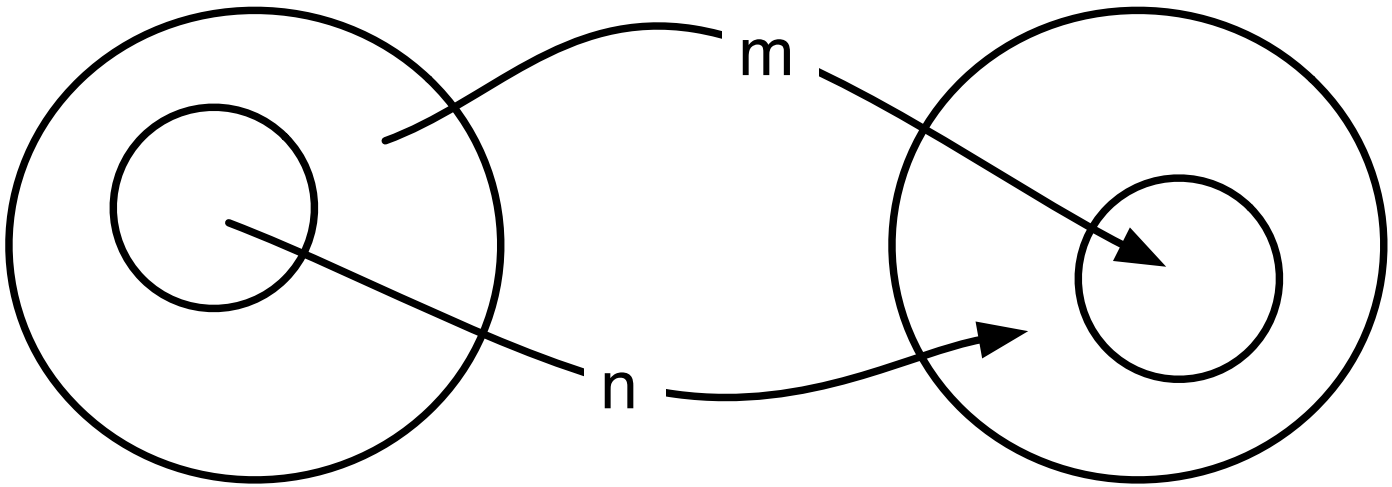
```
//requires:  An assumption about the X input
T m(X in)

//requires:  An assumption about the Y input
T n(Y in)
```

If some program expects an n, but is given an m, things will be ok when:

- Y <: X, and
- assumptions about the input Y are *stronger* than assumptions about the returned X.
- *Opposite of returns*
  - Returns—"covariant"
  - Requires—"contravariant"

Method Specification Subtyping Pictorially

m <: n



Checks vs. Requires

- //requires—anything can occur given bad input
- //checks—an exception must be thrown on bad inputs
- checks is more specific

```
//checks x > 0
void m(int x)

//requires x > 0
void n(int x)
```

m <: n

Effects must be “invariant”.

```
//A counter is part of some abstract state.

//Effects: increment the counter
void m()

//Effects: decrements the counter
void n()
```

Because m and n have different effects, we cannot replace one by the other:

$$m \not<: n \quad \text{and} \quad n \not<: m$$

Class/Interface Semantic Subtyping

$$A <: B \text{ for classes/interfaces A and B}$$

- For all accessible members, m,

$$A.m <: B.m$$

- Each abstract state of A is part of an abstract state of B.
  - Otherwise code trying to use an A in place of a B will be confused.

Example

```
// State: amount is EMPTY or FULL
class Glass{ ... }

// State: amount is EMPTY or FULL;
//        kind is ORANGE or APPLE
class JuiceGlass{ ... }

// State: amount is EMPTY, HALF or FULL
class PreciseGlass{ ... }

// State: amount is EMPTY, HALF or FULL
//        kind is ORANGE or APPLE
class PreciseJuiceGlass{ ... }
```

Subtypes:

Example

```
// State: amount is EMPTY or FULL
class Glass{ ... }

// State: amount is EMPTY or FULL;
//        kind is ORANGE or APPLE
class JuiceGlass{ ... }

// State: amount is EMPTY, HALF or FULL
class PreciseGlass{ ... }

// State: amount is EMPTY, HALF or FULL
//        kind is ORANGE or APPLE
class PreciseJuiceGlass{ ... }
```

Subtypes: JuiceGlass <: Glass, Glass <: PreciseGlass,
JuiceGlass <: PreciseJuiceGlass,
PreciseJuiceGlass <: PreciseGlass

(Adding methods may reduce number of subtypes)

The Liskov Substitution Principle

A `is_a` B only makes sense when A `<:` B

Alternatively: Let q(x) be a property provable about objects x of type T. Then q(y) should be true for objects y of type S where S is a subtype of T.

Thoughts on the Liskov Substitution Principle

- Hard to design an maintain classes that respect the $<:$ relation.
- Classes hierarchies that don't respect $<:$ are likely to
  - contain subtle bugs
  - requires lots of type tests using is or as
  - both
- In practice we see lots of shallow class hierarchies—is this due to the difficulty of building Liskov substitutable classes?

Projects

- Write an exciting program.
- Examples from C++ course:
  - Stock analysis tool
  - Games
- Groups of 3–4.
- Worth 50% of course grade.
- You'll propose projects by Friday and can start coding once I approve them.
- Write-up posted last week.