Typing in OOPLs

9-11-2001

Typing in General

- Strong vs. Weak
  - In a strongly typed language, every statement in the language is checked for type either at compile time or at run time. The types of assignments, parameters, etc. will ALWAYS agree in a strongly typed language or an error occurs.
- Static vs. Dynamic
  - Static checking is done at compile time, dynamic checking at runtime.

Two Uses of Inheritance

- Inheritance for reuse: This is probably one of the main things you learned about inheritance here. However, I personally feel that it hasn't worked all that well.
- Inheritance for subtyping: This is largely what we are concerned with in this class. Java expanded on this by having interfaces.

Assignment Rules

- You can make an assignment of u=t (with types U and T) when
  - T<:U - always with no check
  - U<:T - this is a narrowing operation and requires a runtime check. In Java or C++ it would require an explicit cast.
- Their rule for ordinal types isn't used much in modern languages.

Arrays

- What you can do with arrays depends significantly on how the language deals with arrays.
- In Java arrays are objects and all objects are references so it is easy to put elements of a subtype in an array of the supertype.
- In C++ the array actually allocates a block of memory to store things.

Name Equivalence vs. Structural Equivalence

- The languages you are likely to work with use name equivalence. Types are compared in relation to their declarations.
- Structural equivalence seems to have generally lost the battle here. Part of that is because we don't want unintended overlaps.
- Java can deal with this somewhat because of the compact size of bytecodes.

Inclusion Polymorphism

- One feature of a subtype is that an object of any subtype can used in place of an object of a supertype.
- This leads to a form of universal polymorphism called inclusion polymorphism. Code can be written once to work with a supertype and the same code can be executed with any subtype.

Virtual Functions

- In order to fully take advantage of inclusion polymorphism, objects need to be able to perform specialized behaviors depending on type for a given function call. That is to say that when a method foo of an object is invoked it should be able to call a method specialized the the specific suptype and not just the function of the supertype. This is what virtual functions do.

Virtual Function Tables

- The simplest method of implementing virtual functions is through virtual function tables. This only works with a simple implementation when the language has only single inheritance.
- When you have single inheritance, the functions and data members of a class can be numbered where the new members and methods of subclasses have higher numbers.

Single vs. Multiple Inheritance

- In a language with multiple inheritance the implementation is more difficult (we might read a paper by Stroustrup on how C++ implements this). However, the general idea remains the same.
- Not all languages use this type of implementation to achieve Universal polymorphism. Smalltalk and JavaScript resolve function calls at runtime. Templates and Java generics (should be present in 1.4) enable resolution without subtyping at compile time.

Minute Essay

- Are there things we talked about today that you think we need to discuss more? What other topics do you think need to be discussed before we launch into pointer analysis? Would you like to have a discussion of hardware to help motivate some of the optimization techniques? Would you like me to make the Stroustrup paper the paper for next class?