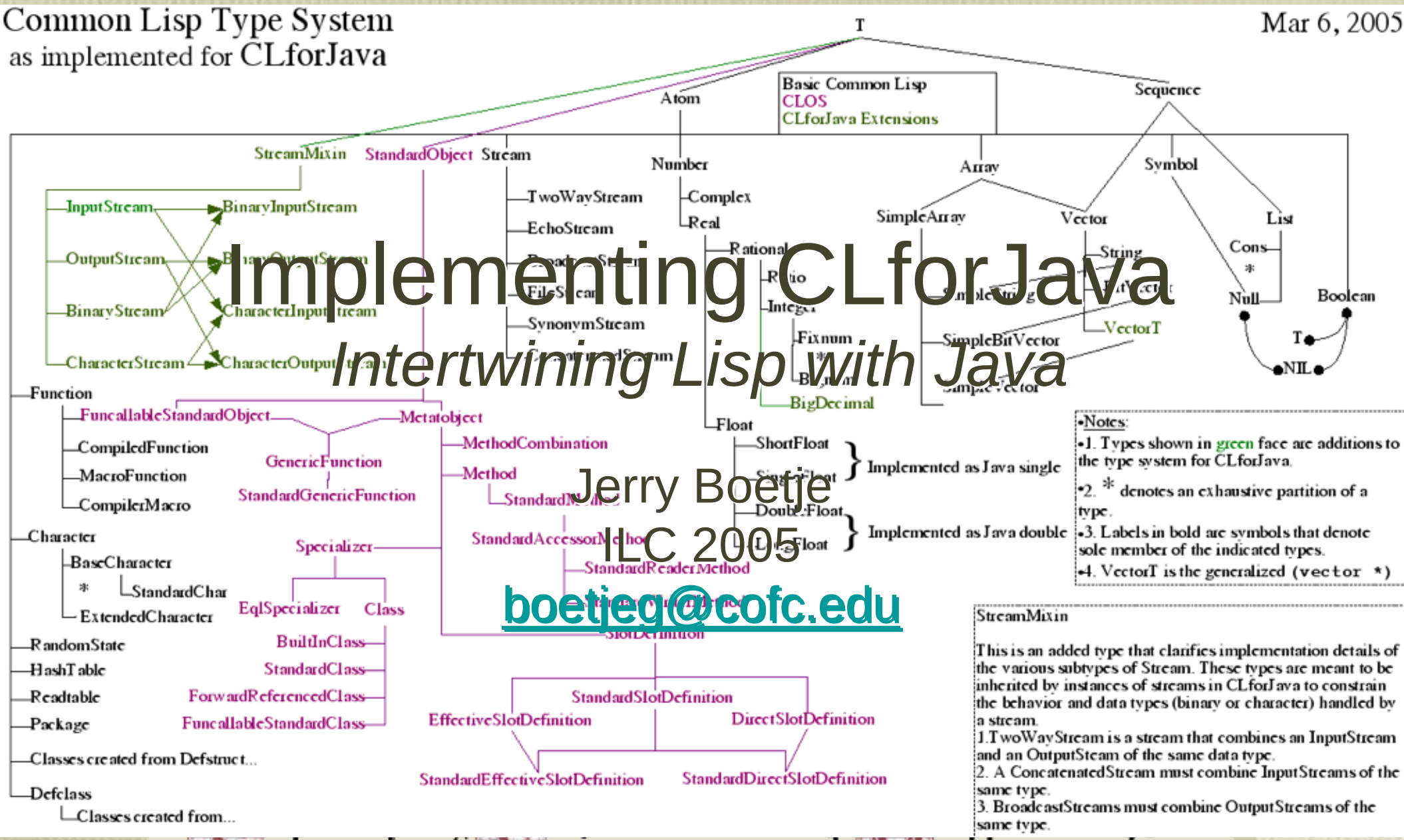


Implementing CLforJava

Intertwining Lisp with Java

Jerry Boetje
ILC 2005
boetjeg@cofc.edu

School of Sciences and Mathematics

Overview

CLforJava Project

- Multi-semester undergraduate project
- Capstone software engineering course
- Gives students a “real world” experience
  - Develop a complex product
  - Develop teamwork skills
  - Use industrial tools and methods
  - Graded by industry standards

The Product

- New, original implementation of Common Lisp
- Runs on the Java Virtual Machine
- Written in Java and Lisp
- So, what's new?

Intertwining with Java

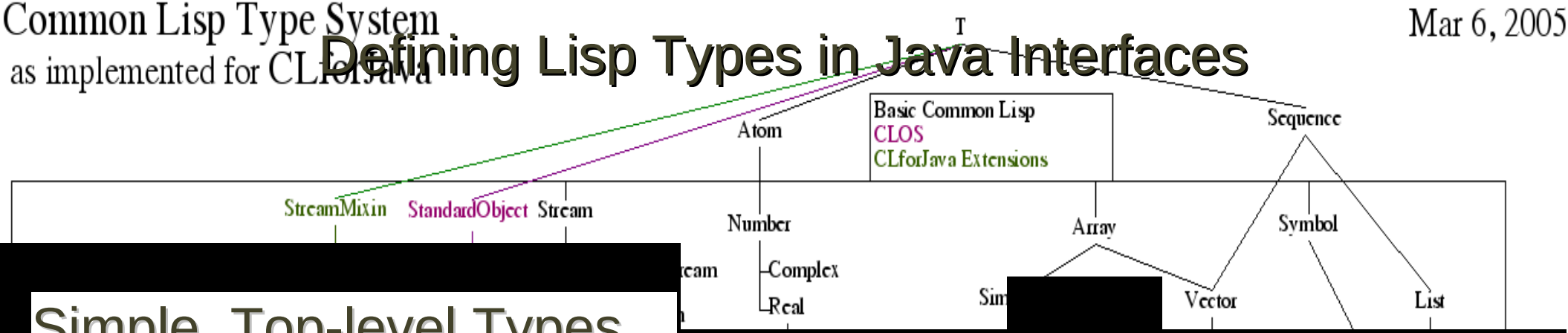
- Make it simple and “natural” to
  - Call Java routines from Lisp
  - Call Lisp routines from Java
- Complete, documented Java API
- No foreign function interface
  - CLOS classes, generic functions, and methods

Where To Start?

- Key is to mesh the type systems
  - Java is strict - class/interface based
  - Lisp is “tangled” and dynamic
- Use Java interfaces for multiple inheritence
  - Create root interface **`lisp.common.type.T`**
  - Define primitive methods for the type
  - Define nested Factory classes

Types

Defining Lisp Types in Java Interfaces

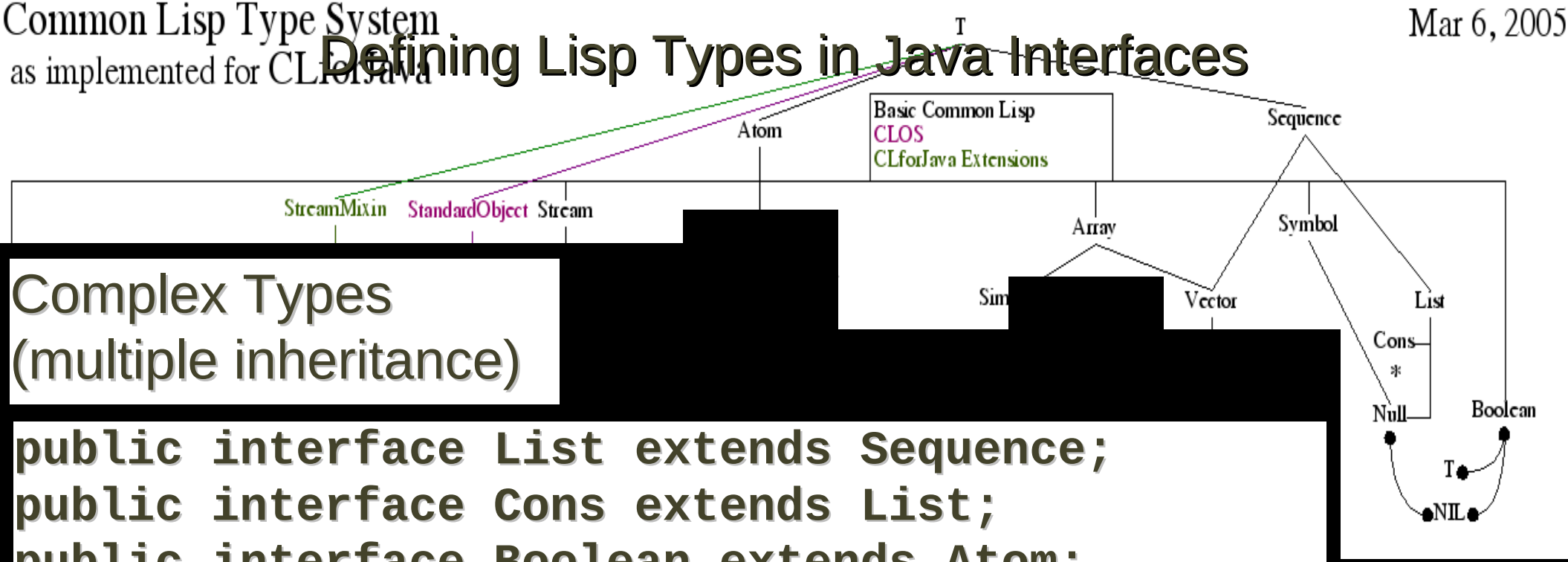


Simple, Top-level Types

```
package lisp.common.type;

public interface T;
public interface Atom extends T;
public interface Sequence extends T;
public interface Number extends Atom;
public interface Symbol extends Atom;
```

Defining Lisp Types in Java Interfaces



Complex Types (multiple inheritance)

```
public interface List extends Sequence;
public interface Cons extends List;
public interface Boolean extends Atom;
public interface Null extends List, Symbol;
```

Add Primitive Methods

- Define a base set for the type
- Use Java conventions for method naming

```
interface Number
    extends Atom, Comparable {
 Number plus(Integer arg);
 Number minus(Integer arg);
 Number mult(Integer arg);
 Number div(Integer arg);
 ...
 Number plus(SingleFloat arg);
 ...
 Number plus(DoubleFloat arg);
 ...
 Number plus(Ratio arg);
 ...
 Number plus(Complex arg);
 ... }
```

```
interface List<CarType, CdrType>
  extends Sequence, Collection {
 List copy();
 CarType getCar();
 CdrType getCdr();
 boolean isCircular();
 CarType last();
 void setCar(CarType car);
 void setCdr(CdrType cdr);
}
// Generic syntax abbreviated
// Common type defaults
//  CarType - Object
//  CdrType - List
```

Atomic Types

- Translates directly to simple Java Interfaces
- Examples
- `base-char => interface BaseChar extends Character`
- `fixnum => interface Fixnum extends Integer`
- `function => interface Function extends T`
- `null => interface Null extends Symbol, List`

Structural Types

- Like atomic types, just dynamically created
  - Javafied type name and slot names
- Example

```
(defstruct foo slot-a slot-b)
=>
public interface Foo extends T {
 Object getSlotA();
 void setSlotA(Object arg);
 Object getSlotB();
 void setSlotB(Object arg);
 public static class Factory {
  public static final Foo newInstance(a, b){
    return new internal-foo-name(a, b);
 }}
 protected class internal-foo-name implements Foo {
   implementations of the methods
}}
```

Supports subclassing when extending `foo`

Including a Structure

- Just like a simple **`defstruct`** but different

```
(defstruct (bar :include foo) slot-c)
=>
public interface Bar extends Foo {
 Object getSlotC();
 void setSlotC(Object arg);
 public static class Factory {
  public static final Bar newInstance(a, b, c){
    return new internal-bar-name(a, b, c);
 }}
 protected class internal-bar-name {
   public internal-bar-name(a, b, c) {
     super(a, b);
     ... }
   implementations of the methods
}}
```

Constuctor calls the superclass 2-arg constructor

Compound Types

- Define sub-interfaces of an existing type
- Add instance of a **`TypeConstraint`** to the type interface
- **`TypeConstraint`** is an interface specifying
  - **`boolean checkConstraints(Object[] args)`**;
  - **`Object[] getConstraints()`**;
- All atomic types specifications (Java interfaces) have a **`Factory`** method to build a **`TypeConstraint`**
- A type constraint may be any Java class or interface

Compound-Only Types

- Arbitrary test for membership
  - All are variations on **`satisfies`**
- **`CompoundOnlyTypeFactory`** abstract class
  - Static method returns a Factory class for each of the compound-only types
  - Use the Factory to create a **`CompoundOnlyType`** interface
  - Interface contains method to check for a member of the type

Function Architecture

Basic Function Pattern

- A class implementing **`lisp.common.type.Function`**
  - An interface that defines the **`apply`** method
  - **`public Object apply(List args);`**
- **`(lambda (x) (1+ X)) =>`**

```
public class Lambda21 implements Function {
   public Object apply(List args) {
     code for 1+ x } ...}
```

- **`#'(lambda (x) (1+ X)) => new Lambda21();`**

Basic Function Pattern

- Additional methods depending on the number of arguments - **`funcall`**
- Defined in interfaces
  - No args => **`lisp.extensions.type.Function0`**
  - 1 arg => **`lisp.extensions.type.Function1`**, etc
- Current limit is 11 - most needed for CL

Named Functions

- Like other functions, but have 2 static fields
  - **`FUNCTION`** - an instance of the function class
  - **`SYMBOL`** - the symbol that names the function
    - **`symbol-function`** returns function instance
- They are singleton instances
  - Private constructor

For the Java Programmer

- All of the CL functions are available directly
  - Static fields in class **`CommonLispFunctions`**
- Examples
  - **`public static final Function Car;`**
  - **`public static final MacroFunction Do;`**

```
package lisp.common.function;
// NOTE: imports removed for clarity

public class Car extends FunctionBaseClass implements Function1 {
    public static final Function FUNCTION = new Car();
    public static final Symbol SYMBOL = (Symbol)Package.CommonLisp.intern("CAR").get(0);
    static { SYMBOL.setFunction(FUNCTION); }

    /** Creates a new instance of Car */
    private Car() {
    }

    /**
     * @param args an array of objects. Valid only for one element
     * @param args is a lisp list. Valid only for one element
     * @return the first element in the Cons
     */
    public Object apply(Object[] args) {
        return funcall(args[0]);
    }
    public Object apply(lisp.common.type.List args) {
        return funcall(args.getCar());
    }

    /**
     * @param arg1 a List (Cons or NIL)
     * @return the car of the argument
     */
    public Object funcall(Object arg1) {
        if (arg1 instanceof List) {
            List list = (List)arg1;
            return list.getCar();
        } else {
            throw new FunctionException("Argument must be of type LIST", new IllegalArgumentException());
        }
    }
}
```

Multiple Values

- Java stack discipline is too strong
- Create a MultipleValue class to hold values
- Have to check function return type
  - Later version of the compiler can improve

Compilation

Bootstrap Compiler

- Basic Function Translator
  - Entirely in memory
  - Produces Oolong assembler code
  - In-memory Java class loader
- Handles a dozen special forms
- Variables always looked-up in dynamic environment (slow but works)

Compiler V2

- Improved Binding Analysis
  - Differential handling of locals v closures
- Explicit closure allocation
- Use of Java locals
- Compiler data structures all lists
  - Support moving to Lisp

File Compilation

- Reads forms and wraps in a lambda
  - In-line code compiled in outer lambda
  - Nested lambdas treated as nested classes
- Compiler honors **`eval-when`** forms
- Written to a standard Jar file
  - Outer lambda referenced in the Manifest
  - Loader locates the outer lambda class, creates an instance, and calls the **`apply`** method

System Documentation

JavaDoc

- Full JavaDoc generated for the Java code
- Planning an extension to provide JavaDoc for Lisp code

LispDoc

Documentation Strings

- Gather documentation strings
- Augment with context-dependent information
  - Function args and types
  - Source file
  - Other function references
- May add ‘;;;’ comment information

XML Encoding

- All of the text is encoded in XML
- Supports runtime transforms using XML
  - Simple text
  - DocBook
  - PDF, etc
- Bachelor's Thesis this year

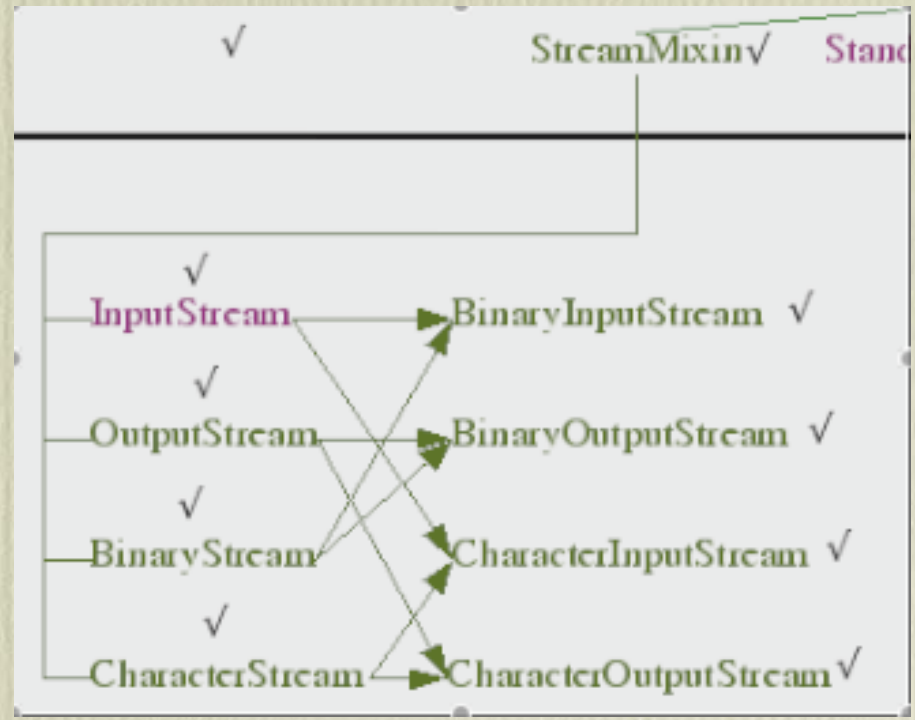
File System

Meshing with Java I/O

- Use the Java I/O system
- Character streams
  - BufferedReader and BufferWriter
- Binary I/O
  - RandomFile
  - Arbitrary byte size

Typing the Lisp I/O

- Lisp I/O type is dynamic
- Java is statically typed
- Create a set of mixins



CLOS
The Best for Last

Java Packages and Naming

- Java packages are a specialization of Package
- Package name is the Java package name
  - **`java.io`**, **`java.util.logging`**
- Symbols have the form
  - JavaClassName[.MemberName]
  - **`System`**, **`System.out`**
- Ex: **`java.lang:System.out`**

Integrating with Java Types

- So, what do these symbols represent?

	Class Name	*Java Field*	*Method*
Value	Instance of `java.lang.Class`	Value of Java field	Instance of `java.lang.reflect.method`
Function	Instance of `java.lang.reflect.Constructor`	--	CLOS method

CLOS Types

- CLOS type defined by a Java interface
- Interface extends the superclass Java interfaces
  - Java stores them in the same order as CLOS
  - Topo sort algorithm works just fine
- Implementing class is a nested static class
  - As usual, implements Function interface

Generic Functions

- **`StandardGenericFunction`** is an abstract class implementing the **`GenericFunction`** interface
- Implementing classes (nested in type interface) subclass **`StandardGenericFunction`**
- Contains a static method to compute the discriminating function

Method Combination

- There's an interface - of course!
  - **`MethodCombination`**
- Several supplied classes
  - **`StandardMethodCombination`**
  - other common ones
- Instances of each available in **`MethodCombination`**

Calling Java Methods

- NO FOREIGN FUNCTION INTERFACE
- Uses CLOS generic functions
-
```
(defgeneric java.io:PrintStream.println
    (stream object))
```
-
```
(defmethod java.io:PrintStream.println
    ((java.io:PrintStream.println stream)
object)
  (call-next-method))
```
-
```
;; now call home
(java.io:PrintStream.println
   java.lang:System.out “Hello World”)
```

Calling Java Methods

- Can use multi-methods
- Define a method to count times we write to **`System.out`**
- ```
  (defmethod java.io:PrintStream.println
       (((eql java.io:System.out) stream) object)
     (incf *system-out-counter*)
     (call-next-method))
  ```
- That's all there is to it!
- Can use any of the method combinations
- Last **`call-next-method`** calls the Java method
  ```

MOP

- Goal is to implement the entire MOP
- Master’s candidate’s problem!

Engineering

The Process

- Classic spiral method
- Each semester is one turn around the spiral
  - New team each semester
  - 4 weeks orientation, 8 weeks development, 3 weeks clean-up
- Occasionally a standout
  - Bachelor or Master's thesis work for a year
  - Boostrap compiler, XML doc, CLOS

The Tool Set

- IDE - Netbeans 4.1 (works also with XCode)
- Source control - Perforce
- Bug tracking - Bugzilla
- Testing - JUnit
- Build system - ANT
- Documentation - TWiki
- Status reporting - MoveableType
- Discussion Forum - Simple Machines Forum

Benchmarks

- Not running the Gabriel benchmarks yet
- Roughly 50-100% slower than CLisp and LispWorks
- Except in Tak and factorial - about even
  - Surprised us too
  - Java implementation of BigInteger

Futures

- Support for the Java debugger architecture
- CLIM in Swing (using the generic functions)

Summary

- A new CL version intertwined with Java
- Done by undergraduates
- 2 years done, about 5-6 more to version 1
- Engineering education, not market development
- But it has some interesting features!
- http://clforjava.cs.cofc.edu/CLforJava.htm
- http://clforjava.cs.cofc.edu/forum/

Q & A