Function Dispatching (ANSI C89)

```c
#include <stdio.h>

void fun(int i) { }
void foo(int i) { }
void bar(int i, double j) { }

int main(){
  fun(i);
  bar(i,1.2);
  foo = fun;
  return 0;
}
```

Function Dispatching (ANSI C89)

```c
#include <stdio.h>

void println(int i) { printf("%d\n",i); }
void println(float f) { printf("%f\n",f); }

int main(){
  println(1.2);
  println(1.2);
  return 0;
}
```
Function Dispatching (ANSI C89)

```c
#include <stdio.h>

void println(int i) { print("%d\n",i); }
void println(float f) { print("%f\n",f); }

int main(){
    println(1.2);
    println(1);
    return 0;
}
```

⚠️ Functions with same names but different parameters not legal

---

Section 2

Overloading Function Names

---

Generic Selection (C11)

```c
generic-selection  →  .Generic(exp, generic-assoclist)
generic-assoclist  →  (generic-assoc,)\*generic-assoc
generic-assoc     →  typename : exp | default : exp
```

Example:

```c
#include <stdio.h>
#define printf_dec_format(x) .Generic(x, \n    signed int "%%d" \n    float "%%f"
#define println(x) printf(printf_dec_format(x), x), printf("\n");

int main(){
    println(1.2);
    println(1);
    return 0;
}
```
Overloading (Java/C++)

```java
class D {
    public static void p(Object o) { System.out.print(o); }
    public int f(int i) { p("f(int): "); return i+1; }
    public double f(double d) { p("f(double): "); return d+1.3; }
}

public static void main() {
    D d = new D();
    B.p(d.f(2)+"\n");
    B.p(d.f(2.3)+"\n");
}

Overloading (Java/C++)

```java
```java
public static void main() {
    D d = new D();
    B.p(d.f(2)+"\n");
    B.p(d.f(2.3)+"\n");
}
```

>` javac Overloading.java; java Overloading
f(int): 3
f(double): 3.6

Overloading with Inheritance (Java)

```java
class B {
    public static void p(Object o) { System.out.print(o); }
    public int f(int i) { p("f(int): "); return i+1; }
}
class D extends B {
    public double f(double d) { p("f(double): "); return d+1.3; }
}

public static void main() {
    D d = new D();
    D.p(d.f(2)+"\n");
    D.p(d.f(2.3)+"\n");
}

Overloading with Inheritance (Java)

```java
```java
class B {
    public static void p(Object o) { System.out.print(o); }
    public int f(int i) { p("f(int): "); return i+1; }
}
class D extends B {
    public double f(double d) { p("f(double): "); return d+1.3; }
}
```

>` javac Overloading.java; java Overloading
f(int): 3
f(double): 3.6
Overloading Hazards

```java
class D {
    public static void p(Object o) { System.out.println(o); }
    public int f(int i, double j) { p("f(i,d): "); return i; }
    public int f(double i, int j) { p("f(d,i): "); return j; }
}

public static void main() {
    D d = new D();
    D.p(d.f(2, 2) + "\n");
}
```

Static Methods are *Statically Dispatched*

```java
class D {
    public static void p(Object o) { System.out.println(o); }
    public int f(int i, double j) { p("f(i,d): "); return i; }
    public int f(double i, int j) { p("f(d,i): "); return j; }
}

public static void main() {
    D d = new D();
    D.p(d.f(2, 2) + "\n");
}
```

Overloading Hazards

```bash
gcc Overloading.java
Overloading.java:10: error: reference to f is ambiguous
```

Overloading Hazards

```java
class D {
    public static void p(Object o) { System.out.println(o); }
    public int f(int i, double j) { p("f(i,d): "); return i; }
    public int f(double i, int j) { p("f(d,i): "); return j; }
}

public static void main() {
    D d = new D();
    D.p(d.f(2, 2) + "\n");
}
```

Overloading Hazards

```bash
gcc Overloading.java
Overloading.java:10: error: reference to f is ambiguous
```
Static Methods are \textit{Statically Dispatched}

Function Call Expression
Function to be dispatched

\[ f(e_1, \ldots, e_n) \]
\[ \text{dispatches to} \]
\[ t_0 \text{ \(f\)t}_1 p_1, \ldots, t_n p_n \]

\text{Signature}
\[ t_0', \ldots, t_n' \]

\text{Signature}
\[ t_0, \ldots, t_n \]

\text{Function Name}
\text{Static Types of Parameters}
\text{Return Type}

Concrete Method
Provides calling target for a call signature

\[ f(e_1, \ldots, e_n) \]
\[ \text{dispatches to} \]
\[ t_0 \text{ \(f\)t}_1 p_1, \ldots, t_n p_n \]

\text{Signature}
\[ t_0', \ldots, t_n' \]

\text{Signature}
\[ t_0, \ldots, t_n \]

\text{Function Name}
\text{Static Types of Parameters}
\text{Return Type}

\textit{f is applicable to} \textit{f' } \iff \textit{f} \leq \textit{f'}:
\[ \leq \text{ is the subtype relation:} \]
\[ R f(T_1, \ldots, T_n) \leq R' f'(T_1', \ldots, T_n') \]
\[ \implies R \leq R' \quad T_i' \leq T_i \]
Inside the Javac – Predicates

Concept of methods being applicable for arguments:

```java
boolean isApplicable(MemberDefinition m, Type arg[]) {
  // Sanity checks:
  Type mtype = m.getType();
  if (!mtype.isType(TC_METHOD)) return false;
  Type margs[] = mtype.getArgumentTypes();
  if (arg.length != margs.length) return false;
  for (int i = 0; i < arg.length; ++i) {
    if (isMoreSpecific(arg[i], margs[i])) return false;
  }
  return true;
}
```

boolean isMoreSpecific(Type moreSpec, Type lessSpec) //... type based specialization

Concept of method signatures being more specific than others:

```java
boolean isMoreSpecific(MemberDefinition more, MemberDefinition less) {
  Type moreType = more.getClassDeclaration().getType();
  Type lessType = less.getClassDeclaration().getType();
  return isMoreSpecific(moreType, lessType) // return type based comparison
  && isApplicable(less, more).getArgumentTypes()); // parameter type based
}
```

Overloading with Scopes – C++

```cpp
#include<iostream>
using namespace std;

class B {
public:
  int f(int i) { cout << "f(int): "; return i+1; }
};

class D : public B {
public:
  double f(double d) { cout << "f(double): "; return d+1.3; }
};

int main() {
  D* pd = new D;
  cout << pd->f(2) << "\n";
  cout << pd->f(2.3) << "\n";
}
```

Finding the Most Specific Concrete Method

```java
MemberDefinition matchMethod(Environment env, ClassDefinition accessor,
  Identifier methodName, Type[] argumentTypes) throws ...
{
  // A tentative maximally specific method.
  MemberDefinition tentative = null;
  // A list of other methods which may be maximally specific too.
  List candidateList = null;
  for (MemberDefinition method : allMethodsLookup(methodeName)) {
    // Select if this method is applicable.
    if (env.isApplicable(method, argumentTypes)) continue;
    // See if this method is accessible.
    if (accessor == null || (accessor.canAccess(env, method))) continue;
    if ((tentative == null) || (env.isMoreSpecific(method, tentative)))
      // 'method' becomes our tentative maximally specific match.
      tentative = method;
    else if (env.isMatch(method, tentative))
      // Find out if our 'tentative' match is a uniquely maximally specific.
      for (MemberDefinition method : candidateList)
        if (env.isMoreSpecific(tentative, method))
          throw new AmbiguousMethod(tentative, method);
      return tentative;
  }
```

Overloading with Scopes – C++

```cpp
#include<iostream>
using namespace std;

class B {
public:
  int f(int i) { cout << "f(int): "; return i+1; }
};

class D : public B {
public:
  double f(double d) { cout << "f(double): "; return d+1.3; }
};

int main() {
  D* pd = new D;
  cout << pd->f(2) << "\n";
  cout << pd->f(2.3) << "\n";
  cout << pd->f(2.3) << "\n";
}
```
Overloading with Scopes – C++

```cpp
#include<iostream>
using namespace std;
class B { public:
    int f(int i) { cout << "f(int): "; return i+1; }
};
class D : public B { public:
    using B::f;
    double f(double d) { cout << "f(double): "; return d+1.3; }
};

int main() {
    D* pd = new D;
    cout << pd->f(2) << "\n";
    cout << pd->f(2.3) << "\n";
} $ ./overloading
f(int): 3
f(double): 3.6
```

Overloading with Scopes – C++

```cpp
#include<iostream>
using namespace std;
class B { public:
    int f(int i) { cout << "f(int): "; return i+1; }
};
class D : public B { public:
    using B::f;
    double f(double d) { cout << "f(double): "; return d+1.3; }
};

int main() {
    D* pd = new D;
    cout << pd->f(2) << "\n";
    cout << pd->f(2.3) << "\n";
} $ ./overloading
f(int): 3
f(double): 3.6
```

Object Orientation

**Emphasizing the Receiver of a Call**

In Object Orientation, we see objects associating strongly with particular procedures, a.k.a. **Methods.**

```cpp
class Natural {
    int value;
    void incBy(Natural n, int i){
        value += Math.abs(i);
    }
}
```

```cpp
class Natural {
    int value;
    void incBy(int i){
        this.value += Math.abs(i);
    }
}
```

- Associating the first parameter as **Receiver** of the method, and pulling it out of the parameters list
- Implicitly binding the first parameter to the fixed name **this**

Subtyping in Object Orientation

**Emphasizing the Receiver’s Responsibility**

An Object Oriented Subtype is supposed to take responsibility for calls to Methods that are associated with the type, that it specializes.

```cpp
class Integral {
    int i;
    void incBy(int delta){
        i += delta;
    }
}
class Natural extends Integral {
    int value;
    void incBy(int i){
        this.value += Math.abs(i);
    }
}
```

```cpp
Integral i = new Integral(-5);
i.incBy(42);
```

- In OO, at runtime subtypes can inhabit statically more general typed variables
- Implicitly call the specialized method!
Methods are dynamically dispatched

Function Call Expression
Call expression to be dispatched.

Concrete Method
Provides calling target for a call signature

\[ f(e_1, \ldots, e_n) \]
\[ \text{dispatches to} \]
\[ t_0 \text{ fit } p_1, \ldots, p_n \text{ } \]
\[ \text{handles} \]
\[ \text{is applicable to} \]
\[ t_0' \ldots, t_n' \]

Signature
Static types of actual parameters.

Specializer
Specialized types to be matched at the call

How can we implement that?

Let's look at what Java does!
The Java platform as example for state of the art OO systems:
- Static Java-based compiler
- Dynamic Hotspot JIT-Compiler/Interpreter

Let's watch the following code on its way to the CPU:

```java
public static void main(String[] args){
    Integral i = new Natural(1);
    i.isBy(42);
}
```

Bytecode

\[ \Rightarrow \text{matchMethod returns the statically most specific signature} \]
\[ \Rightarrow \text{Codegeneration hardcodes invokevirtual with this signature} \]

```java
0: new #1 // class Natural
3: dup
4: iconst_1
5: invokevirtual #5 // Method *<init>*:(I)V
8: astore_1
9: aload_1
10: invokevirtual #6 // Method Integral.isBy:(I)V
12: return
```

? What is the semantics of invokevirtual?
Bytecode

Code:
0: new #1 // class Natural
3: dup
4: invokevirtual #5 // Method \texttt{<init>}:()V
8: astore_1
9: aload_1
10: bipush 42
12: invokevirtual #6 // Method \texttt{Integral incBy}:()I
14: return

? What is the semantics of invokevirtual?

Check the runtime interpreter: Hotspot VM calls \texttt{resolve method}.

Inside the Hotspot VM

void LinkResolver::resolve_method(StringHash& resolved_method, ClassHandle resolved_klass, Symbol* method_name, Symbol* method_signature, ClassHandle current_klass) {
  // 1. check if klass is not interface
  if (resolved_klass->is_interface()) ... throw “Found interface, but class was expected”

  // 2. lookup method in resolved klass not first super klass
  lookup_method_in_klass(resolved_method, resolved_klass, resolved_method, method_name, method_signature);

  // calls klass::lookup_method
  if (resolved_method is null) ... not found in the class hierarchy

  // 3. lookup method in all the interfaces implemented by the resolved klass
  lookup_method_in_interfaces(resolved_method, resolved_klass, method_name, method_signature);

  // 4. method lookup failed
  if (resolved_method is null) ... threw java_lang_RuntimeException()

  // 5. check if method is concrete
  if (resolved_method is abstract() & resolved_klass->is_abstract())
    ... threw java_lang_RuntimeException()

  // 6. access checks, etc.

The method lookup recursively traverses the super class chain:

```
MethodDesc* klass::lookup_method(Symbol* name, Symbol* signature) {
  for (ClassDesc* klas = as_class(klass); klas != NULL; klas = as_class(klas)->super()) {
    MethodDesc* method = klas::cast(klas)->find_method(name, signature);
    if (method == NULL) return method;
  }
  return NULL;
}
```
Single-Dispatching: Summary

Compile Time

Runtime

JavaC
Matches a method call expression 
statically to the most specific 
method signature via 
methoedMatch(...)

Hotspot VM
Interprets invokevirtual via 
resolveMethod(...), scanning 
the superclass chain with 
findMethod(...) for the statically 
fixed signature

Inside the Hotspot VM

```java
MethodDesc* n = (MethodDesc*)methods->obj_at(i);
if (n->name() != name) break;
if (n->signature() == signature) return n;
```

Example: Sets of Natural Numbers

JavaC
Matches a method call expression 
statically to the most specific 
method signature via 
methoedMatch(...)

Hotspot VM
Interprets invokevirtual via 
resolveMethod(...), scanning 
the superclass chain with 
findMethod(...) for the statically 
fixed signature

```java
class Natural {
    Natural(int n) { number = Math.abs(n); }
    int number;
    public boolean equals(Natural n) {
        return n.number == number;
    }
    ...
    Set<Natural> set = new HashSet<>();
    set.add(new Natural(0));
    set.add(new Natural(1));
    System.out.println(set);
```
Mini-Quiz: Java Method Dispatching

```java
class A {
    public static void p (Object o) { System.out.println(o); }
    public void m1 (A a) { p("m1(A) in A"); }
    public void m1 () { m1(new B()); }
    public void m2 (A a) { p("m2(A) in A"); }
    public void m2 () { m2(this); }
}
class B extends A {
    public void m1 (B b) { p("m1(B) in B"); }
    public void m2 (A a) { p("m2(A) in B"); }
    public void m3 () { super.m1(this); }
}
B b = new B(); A a = b; a.m1(b);
B b = new B(); B a = b; b.m1(a);
B b = new B(); b.m2();
B b = new B(); b.m1();
B b = new B(); b.m3();
```

Can we expect more than Single-Dispatching?

Mainstream languages support specialization of first parameter:
C++, Java, C#, Smalltalk, Lisp

So how do we solve the `equals()` problem?
- introspection?
- generic programming?

Introspection

```java
class Natural {
    Natural(int n){ number=Math.abs(n); }
    int number;
    public boolean equals(Object n){
        if (!(n instanceof Natural)) return false;
        return ((Natural)n).number == number;
    }
    ...
    Set<Natural> set = new HashSet<>();
    set.add(new Natural(0));
    set.add(new Natural(0));
    System.out.println(set);
    }
```

```
> $ java Natural
[0]
```

Works⚠️ but bothers programmer with type safety
Generic Programming

```java
interface Equalizable<T>{
    boolean equals(T other);
}
class Natural implements Equalizable<Natural> {
    Natural(int n){ number=Math.abs(n); }
    int number;
    public boolean equals(Natural n){
        return n.number == number;
    }
}
...

EqualizableAwareSet<Natural> set = new MyHashSet<>();
set.add(new Natural(0));
set.add(new Natural(0));
System.out.println(set);
```

⚠️ but needs another Set implementation and...

---

Generic Programming

```java
interface Equalizable<T>{
    boolean equals(T other);
}
class Natural implements Equalizable<Natural> {
    Natural(int n){ number=Math.abs(n); }
    int number;
    public boolean equals(Natural n){
        return n.number == number;
    }
}
...

EqualizableAwareSet<Natural> set = new MyHashSet<>();
set.add(new Natural(0));
set.add(new Natural(0));
System.out.println(set);
```

⚠️ but needs another Set implementation and... ⚠️ does not compile

---

Formal Model of Multi-Dispatching [7]

- `f(e_1,...,e_n)`
- Dispatches to `t_q` handles `f(t_1, p_1,...,t_n, p_n)`
- Determines `t_q`'s, `t_i`'s
- Is applicable to `t_q`'s
- Specializes by `t_i`'s
Formal Model of Multi-Dispatching [7]

**Idea**
Introduce Specializers for all parameters

\[ f(e_1, \ldots, e_n) \text{ dispatches to } t_0(f_1 p_1, \ldots, t_n p_n) \]

- **Signature**
  - \( t_0', \ldots, t_n' \)
- **Specializer**
  - \( f_0, t_1' \ldots, t_n' \)

**How it works**

1. Specializers as subtype annotations to parameter types
2. Dispatcher selects *Most Specific Concrete Method*

Implications of the implementation

**Type-Checking**
1. Type checking of concrete methods introduces checking the existence of unique most specific methods for all valid visible type tuples.
2. Multiple-Inheritance or interfaces as specializers introduce ambiguities, and thus induce runtime ambiguity exceptions

**Code-Generation**
1. Specialized methods generated separately
2. Dispatcher method calls specialized methods
3. Order of the dispatch tests ensures to find the most specialized method

**Performance penalty**
The runtime penalty for multi-dispatching is number of parameters of a multi-method many instances of tests.

Natural Numbers in Multi-Java [3]

```java
class Natural {
    public Natural(int n) {
        number = Math.abs(n);
    }
    public int number;
    public boolean equals(Object@Natural n) {
        return n.number == number;
    }
}
...
Set<Natural> set = new HashSet<>();
set.add(new Natural(0));
set.add(new Natural(0));
System.out.println(set);
```
Natural Numbers Behind the Scenes

```java
public boolean equals(java.lang.Object);
```

Code:

```java
aload_1
instanceof #2; //class Natural
ifeq 16
aload_0
aload_1
checkcast #2; //class Natural
invokespecial #28; //Method equals$body3$0 (LNatural;)Z
ireturn
aload_0
aload_1
invokespecial #31; //Method equals$body3$1 (LObject;)Z
ireturn
```

Section 5
Natively multidispatching Languages

Clojure

... is a lisp dialect for the JVM with:

- Prefix notation
- () – Brackets for lists
- :: – Userdefined keyword constructor ::keyword
- [] – Vector constructor
- fn – Creates a lambda expression
  (fn [x y] (+ x y))
- derive – Generates hierarchical relationships
  (derive ::child ::parent)
- defmulti – Creates new generic method
  (defmulti name dispatch–fn)
- defmethod – Creates new concrete method
  (defmethod name dispatch–val &fn–tail)

Principle of Multidispatching in Clojure

```clojure
(derive ::child ::parent)
(defmulti fun ([a b] [a b])
  (defmethod fun [:child ::child] [a b] "child equals")
  (defmethod fun [:parent ::parent] [a b] "parent equals")

(pr (fun ::child ::child))
```
Natural Numbers in Clojure

Instantiating the dispatching function as follows approximates the desired behaviour in the Java Natural Numbers case:

```
(defmulti equ (fn [a b] [(::Class a) (::Class b)]))
(defmethod equ [::Natural iNatural] [c1 c2]
  (= (::number c1) (::number c2)))
(defmethod equ :default [x y] false)
(defn natural [i] (::Class ::Natural ::number i))
(defn object [] (::Class ::Object))
(def n1 (natural 42))
(def n2 (natural 42))
(def o (object))
(pr (equ n1 n2))
(pr (equ o n1))
true
false
```

More Creative dispatching in Clojure

```
(defn salary [amount]
  (cond (< amount 600) ::poor
        (>= amount 5000) ::boss
        :else ::assi))

(defrecord UniPerson [name wage])

(defmulti print (fn [person] (salary (:wage person))))
(defmethod print ::poor [person] (str "HiWi " (:name person)))
(defmethod print ::assi [person] (str "Dr. " (:name person)))
(defmethod print ::boss [person] (str "Prof. " (:name person)))

(pr (print (UniPerson. "Simon" 4000)))
(pr (print (UniPerson. "Stefan" 5000)))
(pr (print (UniPerson. "Seidl" 6000)))
```

Natural Numbers in Clojure

Instantiating the dispatching function as follows approximates the desired behaviour in the Java Natural Numbers case:

```
(defmulti equ (fn [a b] [(::Class a) (::Class b)]))
(defmethod equ [::Natural iNatural] [c1 c2]
  (= (::number c1) (::number c2)))
(defmethod equ :default [x y] false)
(defn natural [i] (::Class ::Natural ::number i))
(defn object [] (::Class ::Object))
(def n1 (natural 42))
(def n2 (natural 42))
(def o (object))
(pr (equ n1 n2))
(pr (equ o n1))
true
false
```

More Creative dispatching in Clojure

```
(defn salary [amount]
  (cond (< amount 600) ::poor
        (>= amount 5000) ::boss
        :else ::assi))

(defrecord UniPerson [name wage])

(defmulti print (fn [person] (salary (:wage person))))
(defmethod print ::poor [person] (str "HiWi " (:name person)))
(defmethod print ::assi [person] (str "Dr. " (:name person)))
(defmethod print ::boss [person] (str "Prof. " (:name person)))

(pr (print (UniPerson. "Simon" 4000)))
(pr (print (UniPerson. "Stefan" 5000)))
(pr (print (UniPerson. "Seidl" 6000)))
```

Dr. Simon
HiWi Stefan
Prof. Seidl
**Multidispatching**

**Pro**
- Generalization of an established technique
- Directly solves problem
- Eliminates boilerplate code
- Compatible with modular compilation/type checking

**Con**
- Counters privileged 1st parameter
- Runtime overhead
- New exceptions when used with multi-inheritance
- *Most Specific Method* ambiguous

**Other Solutions (extract)**
- Dylan
- Scala

**Lessons Learned**

**Lessons Learned**
- Dynamically dispatched methods are complex interaction of static and dynamic techniques
- Single Dispatching as in major OO-Languages
- Making use of Open Source Compilers
- Multi Dispatching generalizes single dispatching
- Multi Dispatching Java
- Multi Dispatching Clojure