Script generated by TTT

Title: Petter: Programmiersprachen (19.11.2014)

Date: Wed Nov 19 14:20:25 CET 2014

Duration: 96:19 min

Pages: 55

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Programming Languages

Dispatching Method Calls

Dr. Michael Petter
Winter term 2014

Dispatching - Outline

Dispatching

1. Motiviation
2. Formal Model
3. Quiz
4. Dispatching from the Inside

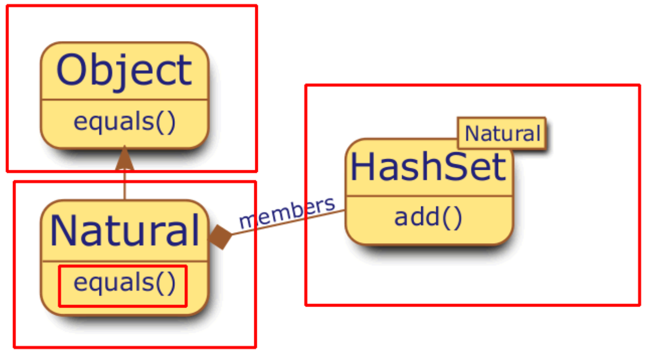
Solutions in Single-Dispatching

1. Type introspection
2. Generic interface

Multi-Dispatching

1. Formal Model
2. Multi-Java
3. Multi-dispatched compare in Java
4. Multi-dispatching in Clojure

Sets of Natural Numbers



Sets of Natural Numbers

```
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(0));
  System.out.println(set);
```


Sets of Natural Numbers

```
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(0));
  System.out.println(set);
```

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

⚠ Why? Is HashSet buggy?

Generalization

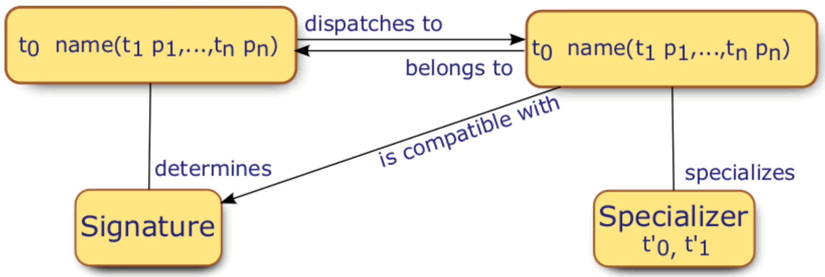
Let's think language independent!

Generalization

Let's think language independent!

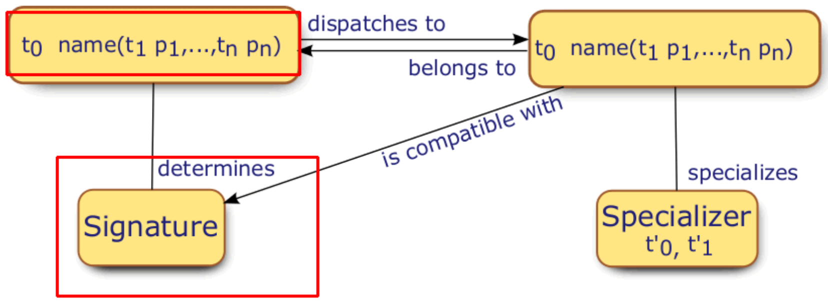
`n1.equals(n2);` $\Longrightarrow$ `equals(n1,n2);`

Methods are *dynamically dispatched*



Methods are *dynamically dispatched*

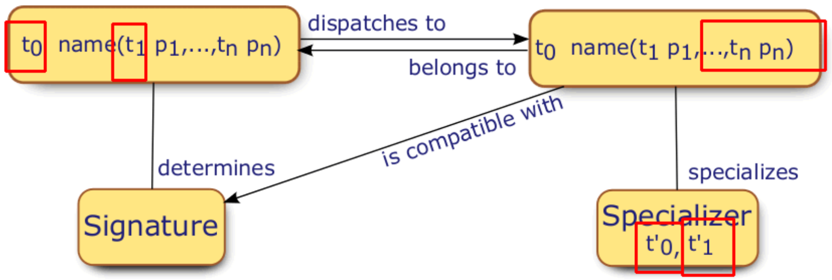
Generic Function
Dynamically dispatched function



Methods are *dynamically dispatched*

Generic Function
Dynamically dispatched function

Concrete Method
Provides code body for a generic function



Signature
Permissible arguments for calls to generic functions

Example: Java [4]

Java determines *generic function signatures* implicitly at each call site from the static types of the arguments.

```
Object o1 = new Natural(1);
Object o2 = new Natural(2);
equals(o1,o2);
```

Signature for call to generic function:

```
equals(Object, Object)
```

Concrete methods within Natural:

```
boolean equals(Natural n1, Natural n2)
boolean equals(Object o1, Object o2)
```

Example: Java [4]

Java determines *generic function signatures* implicitly at each call site from the static types of the arguments.

```
Object o1 = new Natural(1);
Object o2 = new Natural(2);
equals(o1,o2);
```

Signature for call to generic function:

`equals(Object, Object)`

⚠ Specializer in Java only for return type and first argument

Concrete methods within Natural:

```
boolean equals(Natural n1, Natural n2)
boolean equals(Object o1, Object o2)
boolean equals(Natural o1, Object o2)
```


Example: Java [4]

Java determines *generic function signatures* implicitly at each call site from the static types of the arguments.

```
Object o1 = new Natural(1);
Object o2 = new Natural(2);
equals(o1,o2);
```

Signature for call to generic function:

`equals(Object, Object)`

⚠ Specializer in Java only for return type and first argument

⚠ and static methods are not specialized at all

Concrete methods within Natural:

```
boolean equals(Natural n1, Natural n2)
boolean equals(Object o1, Object o2)
boolean equals(Natural o1, Object o2)
```


Mini-Quiz: Java Method Dispatching

```
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());           }
   private static 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);
```


Mini-Quiz: Java Method Dispatching

```
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());           }
   private static 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);
```

m1(A) in A

Mini-Quiz: Java Method Dispatching

```
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());            }
   private static 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);                 m1(A) in A
B b = new B(); B a = b; b.m1(a);
```


Mini-Quiz: Java Method Dispatching


```
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());            }
   private static 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);                 m1(A) in A
B b = new B(); B a = b; b.m1(a);                 m1(B) in B
B b = new B(); b.m1();
```


Mini-Quiz: Java Method Dispatching


```
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());            }
   private static 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);         }
}
```


this

```
B b = new B(); A a = b; a.m1(b);                 m1(A) in A
B b = new B(); B a = b; b.m1(a);                 m1(B) in B
B b = new B(); b.m1();                           m1(A) in A
```


Mini-Quiz: Java Method Dispatching


```
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());            }
   private static 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);                 m1(A) in A
B b = new B(); B a = b; b.m1(a);                 m1(B) in B
B b = new B(); b.m1();                           m1(A) in A
B b = new B(); b.m2();
```

Mini-Quiz: Java Method Dispatching

```
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());            }
   private static 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);`	`m1(A) in A`
`B b = new B(); B a = b; b.m1(a);`	`m1(B) in B`
`B b = new B(); b.m1();`	`m1(A) in A`
`B b = new B(); b.m2();`	`m2(A) in A`

Mini-Quiz: Java Method Dispatching

```
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());            }
   private static 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);`	`m1(A) in A`
`B b = new B(); B a = b; b.m1(a);`	`m1(B) in B`
`B b = new B(); b.m1();`	`m1(A) in A`
`B b = new B(); b.m2();`	`m2(A) in A`
`B b = new B(); b.m3();`	

Mini-Quiz: Java Method Dispatching

```
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());            }
   private static 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);`	`m1(A) in A`
`B b = new B(); B a = b; b.m1(a);`	`m1(B) in B`
`B b = new B(); b.m1();`	`m1(A) in A`
`B b = new B(); b.m2();`	`m2(A) in A`
`B b = new B(); b.m3();`	`m1(A) in A`

So what is happening here?

Let's look at what Java does!

The Java platform as example for state of the art OO systems:

- Static Javac-based compiler
- Dynamic Hotspot JIT-Compiler/Interpreter

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

```
public static void main(String[] args){
    Object o1 = new Natural(1);
    Object o2 = new Natural(2);
    o1.equals(o2);
}
```

Mini-Quiz: Java Method Dispatching

```
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());            }
    private static 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);          m1(A) in A
B b = new B(); B a = b; b.m1(a);          m1(B) in B
B b = new B(); b.m1();                    m1(A) in A
B b = new B(); b.m2();                    m2(A) in A
B b = new B(); b.m3();                    m1(A) in A
```


So what is happening here?

Let's look at what Java does!

The Java platform as example for state of the art OO systems:

- Static Javac-based compiler
- Dynamic Hotspot JIT-Compiler/Interpreter

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

```
public static void main(String[] args){
    Object o1 = new Natural(1);
    Object o2 = new Natural(2);
    o1.equals(o2);
}
```


Inside the Javac – Prerequisites

Concept of methods being *applicable* for arguments:

```
// true if the given method is applicable to the given arguments
boolean isApplicable(MemberDefinition m, Type args[]) {
    // Sanity checks:
    Type mType = m.getType();
    if (!mType.isType(TC_METHOD))                return false;

    Type mArgs[] = mType.getArgumentTypes();
    if (args.length != mArgs.length)             return false;

    for (int i = args.length ; --i >= 0 ;)
        if (!isMoreSpecific(args[i], mArgs[i])) return false;
    return true;
}
boolean isMoreSpecific(Type from, Type to) //... type based specialization
```

Concept of method signatures being *more specific* then others:

```
// true if "best" is in every argument at least as good as "other"
boolean isMoreSpecific(MemberDefinition best, MemberDefinition other) {
    Type bestType = best.getClassDeclaration().getType();
    Type otherType = other.getClassDeclaration().getType();
    return isMoreSpecific(bestType, otherType) // type based relation
           && isApplicable(other, best.getType().getArgumentTypes());
}
```


Inside the Javac

```
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;
    // Get all the methods inherited by this class which have the name 'methodName'
    for (MemberDefinition method : allMethods.lookupName(methodName)) {
        // See 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 this method could possibly be another maximally specific
               // method, add it to our list of other candidates.
               if (!env.isMoreSpecific(tentative,method)) {
                   if (candidateList == null) candidateList = new ArrayList();
                   candidateList.add(method);
    } }     }
    if (tentative != null && candidateList != null)
        // Find out if our 'tentative' match is a uniquely maximally specific.
        for (MemberDefinition method : candidateList )
            if (!env.isMoreSpecific(tentative, method))
               throw new AmbiguousMember(tentative, method);
    return tentative;
}
```

Bytecode

- matchMethod returns the statically most specific signature
- Codegeneration hardcodes invokevirtual with this signature

```
Code:
 0:   new             #4; //class Natural
 3:   dup
 4:   iconst_1
 5:   invokespecial   #5; //Method "<init>":(I)V
 8:   astore_1
 9:   new             #4; //class Natural
 12:  dup
 13:  iconst_2
 14:  invokespecial   #5; //Method "<init>":(I)V
 17:  astore_2
 18:  aload_1
 19:  aload_2
 20:  invokevirtual   #6; //Method java/lang/Object.equals:(Ljava/lang/Object;)Z
 23:  pop
 24:  return
```

? What is the semantics of invokevirtual?

Bytecode

- matchMethod returns the statically most specific signature
- Codegeneration hardcodes invokevirtual with this signature

```
Code:
 0:   new             #4; //class Natural
 3:   dup
 4:   iconst_1
 5:   invokespecial   #5; //Method "<init>":(I)V
 8:   astore_1
 9:   new             #4; //class Natural
 12:  dup
 13:  iconst_2
 14:  invokespecial   #5; //Method "<init>":(I)V
 17:  astore_2
 18:  aload_1
 19:  aload_2
 20:  invokevirtual   #6; //Method java/lang/Object.equals:(Ljava/lang/Object;)Z
 23:  pop
 24:  return
```

? What is the semantics of invokevirtual?

- Check the runtime interpreter: Hotspot VM calls resolve_method!

Inside the Hotspot

```
void LinkResolver::resolve_method(methodHandle& resolved_method, KlassHandle resolved_klass,
                                  Symbol* method_name, Symbol* method_signature,
                                  KlassHandle 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 and its super klasses
  lookup_method_in_klasses(resolved_method, resolved_klass, method_name, method_signature);
      // calls klass::lookup_method() -> next slide

  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

    if (resolved_method.is_null()) {
      // JSR 292:  see if this is an implicitly generated method MethodHandle.invoke(*...)
      lookup_implicit_method(resolved_method, resolved_klass, method_name, method_signature, c
    }

    if (resolved_method.is_null()) { // 4. method lookup failed
      // ... throw  java_lang_NoSuchMethodError()
  }  }

  // 5. check if method is concrete
  if (resolved_method->is_abstract() && !resolved_klass->is_abstract()) {
    // ... throw java_lang_AbstractMethodError()
  }

  // 6. access checks, etc.
```


Inside the Hotspot VM

The method lookup recursively traverses the super class chain:

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

Inside the Hotspot

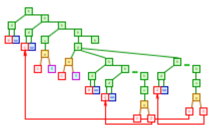
```
void LinkResolver::resolve_method(methodHandle& resolved_method, KlassHandle resolved_klass,
                                  Symbol* method_name, Symbol* method_signature,
                                  KlassHandle current_klass) {

  // 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 and its super klasses
  lookup_method_in_klasses(resolved_method, resolved_klass, method_name, method_signature);
      // calls klass::lookup_method() -> next slide

  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)

    if (resolved_method.is_null()) {
      // JSR 292:  see if this is an implicitly generated method MethodHandle.invoke(*...)
      lookup_implicit_method(resolved_method, resolved_klass, method_name, method_signature, cu
    }

    if (resolved_method.is_null()) { // 4. method lookup failed
      // ... throw  java_lang_NoSuchMethodError()
  } }

  // 5. check if method is concrete
  if (resolved_method->is_abstract() && !resolved_klass->is_abstract()) {
     // ... throw java_lang_AbstractMethodError()
  }

  // 6. access checks, etc.
```


Inside the Hotspot

```
MethodDesc* klass::find_method(ObjArrayDesc* methods, Symbol* name, Symbol* signature) {
  int len = methods->length();
  // methods are sorted, so do binary search
  int i, l = 0 , h = len - 1;
  while (l <= h) {
    int mid = (l + h) >> 1;
    MethodDesc* m = (MethodDesc*)methods->obj_at(mid);
    int res = m->name()->fast_compare(name);
    if (res == 0) {
      // found matching name; do linear search to find matching signature
      // first, quick check for common case
      if (m->signature() == signature) return m;
      // search downwards through overloaded methods
      for (i = mid - 1; i >= l; i--) {
        MethodDesc* m = (MethodDesc*)methods->obj_at(i);
        if (m->name() != name) break;
        if (m->signature() == signature) return m;
      }
      // search upwards
      for (i = mid + 1; i <= h; i++) {
        MethodDesc* m = (MethodDesc*)methods->obj_at(i);
        if (m->name() != name) break;
        if (m->signature() == signature) return m;
      }
      return NULL; // not found
    } else if (res < 0) l = mid + 1;
           else         h = mid - 1;
  }
  return NULL;
}
```


Summary

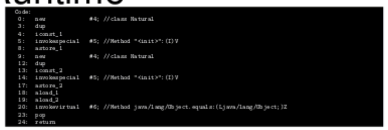
Compile Time



Javac

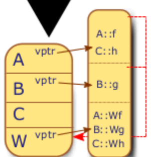
Matches a method call expression *statically* to the *most specific* method signature via `matchMethod( ... )`

Runtime



Hotspot VM

Interprets `invokevirtual` via `resolve_method(...)`, scanning the superclass chain with `find_method(...)` for the statically fixed `signature`



So what to do with Single-Dispatching?

Mainstream languages support specialization of first parameter:

C++, Java, C#, Smalltalk, Lisp

So how do we solve the `equals()` problem?

1. introspection?
2. generic programming?
3. cheating?

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);
```


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

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
⚠ and is only available for languages with type introspection

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);
```

Generic Programming

```
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...

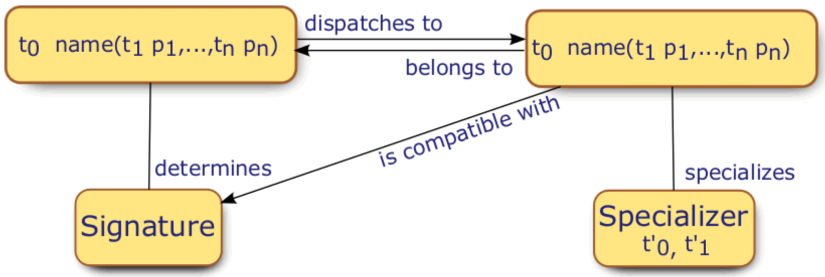
Generic Programming

```
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

```
>$ javac Natural.java
Natural.java:2: error: name clash: equals(T) in Equalizable and equals(Object)
in Object have the same erasure, yet neither overrides the other
```

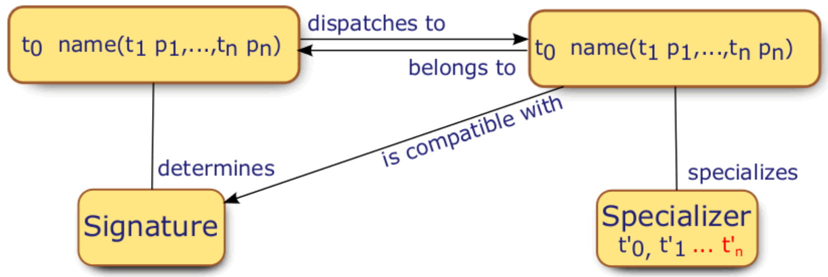

Formal Model of Multi-Dispatching [7]



Formal Model of Multi-Dispatching [7]

Idea

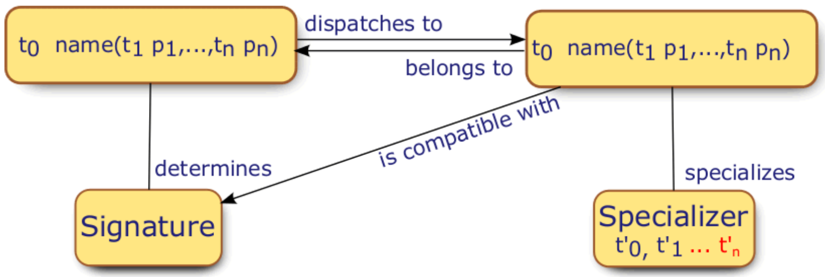
Introduce Specializers for all parameters



Formal Model of Multi-Dispatching [7]

Idea

Introduce Specializers for all parameters



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. Typechecking families of concrete methods introduces checking the existance 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 `instanceof` tests.

Natural Numbers in Multi-Java [3]

```
class Natural {
  public Natural(int n){ number=Math.abs(n); }
  private 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

```
>$ javap -c Natural
```

```
public boolean equals(java.lang.Object);
 Code:
  0:   aload_1
  1:   instanceof      #2; //class Natural
  4:   ifeq     16
  7:   aload_0
  8:   aload_1
  9:   checkcast       #2; //class Natural
  12:  invokespecial   #28; //Method equals$body3$0:(LNatural;)Z
  15:  ireturn
  16:  aload_0
  17:  aload_1
  18:  invokespecial   #31; //Method equals$body3$1:(LObject;)Z
  21:  ireturn
```

How about a natively dynamically dispatched language?

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

```
(derive ::child ::parent)

(defmulti fun (fn [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

Instanciating 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 ::Natural] [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))
```

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" 500)))
(pr (print (UniPerson. "Seidl" 6000)))
```


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

1. Dynamically dispatched methods are complex interaction of static and dynamic techniques
2. Single Dispatching as in major OO-Languages
3. Making use of Open Source Compilers
4. Multi Dispatching generalizes single dispatching
5. Multi Dispatching Java
6. Multi Dispatching Clojure

Further reading...

[1] hotspot/src/share/vm/interpreter/linkResolver.cpp.
OpenJDK 7 Hotspot JIT VM.
http://hg.openjdk.java.net/jdk7/jdk7.

[2] jdk/src/share/classes/sun/tools/java/ClassDefinition.java.
OpenJDK 7 Javac.
http://hg.openjdk.java.net/jdk7/jdk7.

[3] C. Clifton, T. Millstein, G. T. Leavens, and C. Chambers.
Multijava: Design rationale, compiler implementation, and applications.
ACM Transactions on Programming Languages and Systems (TOPLAS), May 2006.

[4] J. Gosling, B. Joy, G. Steele, and G. Bracha.
The Java Language Specification, Third Edition.
Addison-Wesley Longman, Amsterdam, 3 edition, June 2005.

[5] S. Halloway.
Programming Clojure.
Pragmatic Bookshelf, 1st edition, 2009.

[6] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley.
The Java Virtual Machine Specification.
Addison-Wesley Professional, Java SE7 edition, 2013.

[7] R. Muschevici, A. Potanin, E. Tempero, and J. Noble.
Multiple dispatch in practice.
23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications (OOPSLA), September 2008.