TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Script **generated by TTT**

Title:	Petter: Programmiersprachenh (28.11.2018)
Date:	Wed Nov 28 14:12:25 CET 2018
Duration:	93:01 min
Pages:	42

Programming Languages

Dispatching Method Calls

Dr. Michael Petter
Winter Term 2018

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-dispatching in Perl6
4. Multi-dispatching in Clojure

Function Dispatching (ANSI C89)

```
#include <stdio.h>

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

int main(){
  fun(1);
  bar(1,1.2);
  void (*foo)(int);
  foo = fun;
  return 0;
}
```

Function Dispatching (ANSI C89)

```
#include <stdio.h>

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

int main(){
  fun(1);
  bar(1,1.2);
  void (*foo)(int);
  foo = fun;
  return 0;
}
```

Section 2

Overloading Function Names

Function Dispatching (ANSI C89)

```
#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;
}
```

Generic Selection (C11)

generic-selection ↦ `_Generic(`*exp*`,` *generic-assoclist*`)`
generic-assoclist ↦ (*generic-assoc*`,`)* *generic-assoc*
generic-assoc ↦ `typename` : *exp* | `default` : *exp*

Overloading (Java/C++)

```
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();
  D.p(d.f(2)+"\n");
  D.p(d.f(2.3)+"\n");
}
```

Overloading with Inheritance (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();
  B.p(d.f(2)+"\n");
  B.p(d.f(2.3)+"\n");
}
```

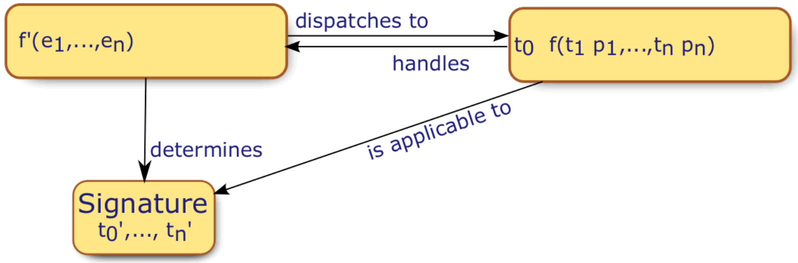
Overloading with Scopes (C++)

```
#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';
}
```

Overloading Hassles

```
class D {
  public static void p(Object o) { System.out.print(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*



Inside the Javac – Predicates

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 moreSpec, Type lessSpec) //... type based specialization
```

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

```
// true if "more" is in every argument at least as specific as "less"
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.getType().getArgumentTypes()); // parameter type based
}
```

Finding the Most Specific Concrete Method

```
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;
}
```

Section 3

Overriding Methods

Object Orientation

Emphasizing the *Receiver* of a Call

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

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

...
incBy(nat,42);
```

$\Longrightarrow$

```
class Natural {
  int value;
  void incBy(int i){
    this.value += Math.abs(i);
  }
}
...
nat.incBy(42);
```

- Associating the first parameter as *Receiver* of the method, and pulling it out of the parameters list
- Implicitely 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.

```
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);
  }
}
```

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

⚠ In OO, at runtime subtypes can inhabit statically more general typed variables

⇝ Implicitely call the specialized method!

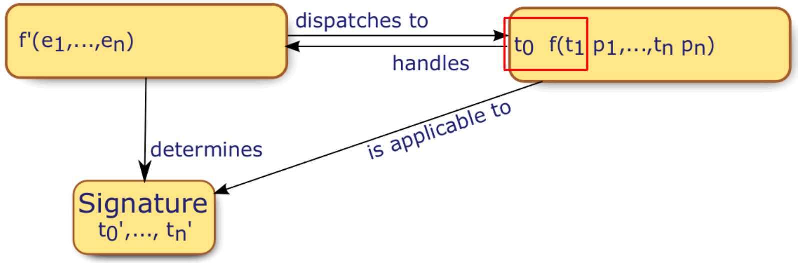
Methods are *dynamically dispatched*

Function Call Expression

Call expression to be dispatched.

Concrete Method

Provides calling target for a call signature



Signature

Static types of actual parameters.

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 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){
    Integral i = new Natural(1);
    i.incBy(42);
}
```

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: aload_1
  10: bipush        42
  12: invokevirtual #6                  // Method Integral.incBy:(I)V
  15: return
```

? What is the semantics of `invokevirtual`?

Inside the Hotspot VM

```
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, current_klass);
    }

    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:

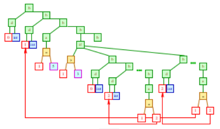
```
MethodDesc* klass::lookup_method(Symbol* name, Symbol* signature) {
  for (KlassDesc* klas = as_klassOop(); klas != NULL; klas = klass::cast(klas)->super()) {
    MethodDesc* method = klass::cast(klass)->find_method(name, signature);
    if (method != NULL) return method;
  }
  return NULL;
}
```

Inside the Hotspot VM

```
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;
}
```

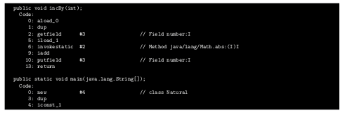
Single-Dispatching: Summary

Compile Time

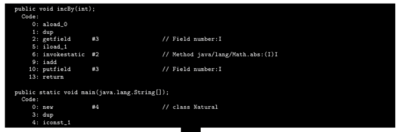


Javac

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

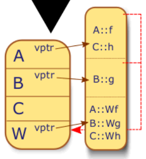


Runtime



Hotspot VM

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



Example: 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);
```

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());            }
   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);
```

Section 4

Multi-Dispatching

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?

1. introspection?
2. generic programming?

Introspection

```
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);
```

Generic Programming

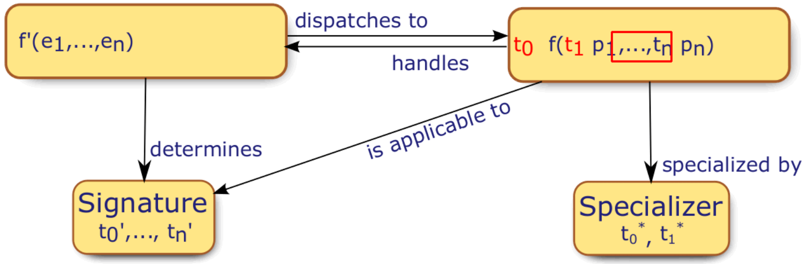
```
interface Equalizable<T>{
  boolean equals(T other);
}
class Natural implements Equalizable<Natural> {
  Natural(int n){ number=Math.abs(n); }
  int number;
  public boolean equals(Natural n){
    return n.number == number;
  }
}
...
  EqualizableAwareSet<Natural> set = new HashSet<>();
  set.add(new Natural(0));
  set.add(new Natural(0));
  System.out.println(set);
```

Double Dispatching

```
abstract class EqualsDispatcher{
  boolean dispatch(Natural) { return false };
  boolean dispatch(Object)  { return false };
}
class Natural {
  Natural(int n){ number=Math.abs(n); }
  int number;
  public boolean doubleDispatch(EqualsDispatcher ed) {
    return ed.dispatch(this);
  }
  public boolean equals(Object n){
    return n.doubleDispatch(
       new EqualsDispatcher(){
         boolean dispatch(Natural nat) {
           return nat.number=number;
       }; }
); }  }
```

✓ Works

Formal Model of Multi-Dispatching [?]



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 determines the most specialized method

Performance penalty

The runtime-penalty for multi-dispatching is related to the number of parameters of a multi-method many `instanceof` tests.

Natural Numbers in Multi-Java [?]

```
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
```

Perl6

```
my Cool $foo;
my Cool $bar;
multi fun(Cool $one, Cool $two){
    say "Dispatch base"
}
multi fun(Int $one,Str $two){
    say "Dispatch 1"
}
multi fun(Str $one,Int $two){
    say "Dispatch 2"
}
$foo=1;
$bar="blabla";
fun($foo,$bar);
```

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))
```

More Creative dispatching in Clojure

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

(defrecord UniPerson [name wage])

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

(pr (print (UniPerson. "Petter"  2000)))
(pr (print (UniPerson. "Stefan"   200)))
(pr (print (UniPerson. "Seidl" 16000)))
```

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 Perl6
6. Multi Dispatching Clojure

Section 6

Further materials