Callability Control

By Isaac Oscar Gariano[1] and Marco Servetto[2]

(Victoria University of Wellington)

[1] isaac@ecs.vuw.ac.nz [2] marco.servetto@ecs.vuw.ac.nz

Conventions

We will consider C♯ (or other .Net or JVM languages), since it

- is statically-typed,
- supports named/identifiable functions (e.g. static/instance methods and constructors),
- supports dynamic dispatch (with interfaces, virtual methods, etc.),
- supports dynamic code loading, and
- supports dynamic function lookup and invocation (with reflection).

For brevity I will omit accessibility modifiers and allow free standing static functions.

The Problem

1. What *could* this code do?

```
static void M1() { Sign(0); }
```

The Problem

1. What *could* this code do?

```
static void M1() { Sign(0); }
```

2. What about this, what *could* it do?

```
interface I { void Run(); }
static void M2(I x) { x.Run(); }
```

The Problem

1. What *could* this code do?

```
static void M1() { Sign(0); }
```

2. What about this, what *could* it do?

```
interface I { void Run(); }
static void M2(I x) { x.Run(); }
```

3. How about this?

```
static void M3(String url) {
  // Load code (possibly from the internet)
  Assembly code = Assembly.LoadFrom(url);
  code.GetMethod("M").Invoke(null, null); } // call M()
```

Callability

- *Callability* is the *ability* to *call* a function.
- A function's callability is the set of things it can call.

Callability

- *Callability* is the *ability* to *call* a function.
- A function's callability is the set of things it can call.

Restatement of the Problem:

1. What is the callability of `Sign`? (Where `Sign` is a static method.)
2. What is the callability of `x.Run`? (Where `x` is of an interface type `I`.)
3. What is the callability of `M`? (Where `M` was a dynamically loaded static-method.)

The Callability Rules

$f \rightsquigarrow g$, i.e. a function f *can-call* a function g, iff:

The Callability Rules

$f \rightsquigarrow g$, i.e. a function f *can-call* a function g, iff:

1. $g \in Calls(f) \Rightarrow f \rightsquigarrow g$, i.e. f is annotated with `calls[ ... ,g, ... ]`.

   Example:
   ```
   static void Write(String s) calls[WriteChar] {
       foreach (Char c in s) WriteChar(c); }
   ```

The Callability Rules

$f \rightsquigarrow g$, i.e. a function f *can-call* a function g, iff:

1. $g \in Calls(f) \Rightarrow f \rightsquigarrow g$, i.e. f is annotated with `calls[ ... ,g, ... ]`.

   Example:
   ```
   static void Write(String s) calls[WriteChar] {
       foreach (Char c in s) WriteChar(c); }
   ```

2. $(\forall h \in Calls(g) \bullet f \rightsquigarrow h) \Rightarrow f \rightsquigarrow g$, i.e. if f can call every function in the `calls[ ... ]` annotation of g.

   Example:
   ```
   static void WriteLine(String s) calls[WriteChar] {
       Write(s + "\n"); }
   ```

The Callability Rules

$f \rightsquigarrow g$, i.e. a function f *can-call* a function g, iff:

1. $g \in Calls(f) \Rightarrow f \rightsquigarrow g$, i.e. f is annotated with `calls[ ... ,g, ... ]`.

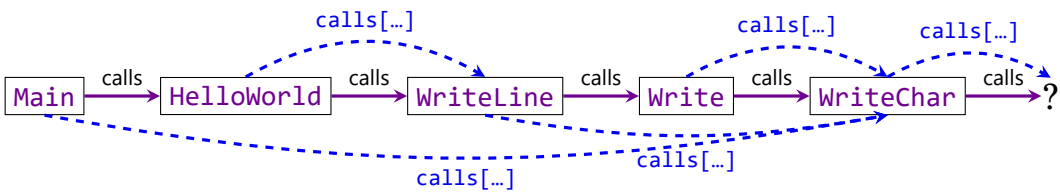
   Example:
   ```
   static void Write(String s) calls[WriteChar] {
       foreach (Char c in s) WriteChar(c); }
   ```

2. $(\forall h \in Calls(g) \bullet f \rightsquigarrow h) \Rightarrow f \rightsquigarrow g$, i.e. if f can call every function in the `calls[ ... ]` annotation of g.

   Example:
   ```
   static void WriteLine(String s) calls[WriteChar] {
       Write(s + "\n"); }
   ```

The previous rules apply transitively, and always allow for recursive calls.

Example:
```
static void HelloWorld() calls[WriteLine] {
    WriteLine("Hello World!"); }
static void Main(String[] args) calls[WriteChar] {
    HelloWorld(); }
```



Primitive Operations

To simplify things we will assume that the language provides only two intrinsic functions, `Unrestricted` and `Restricted`.

Primitive Operations

To simplify things we will assume that the language provides only two intrinsic functions, `Unrestricted` and `Restricted`.

1. `Unrestricted` can be called by any function:

```
static Object Unrestricted(String op, params Object[] args)
  calls[];
```

Example: `Unrestricted("Add", 1, 2); // Returns 3`

Primitive Operations

To simplify things we will assume that the language provides only two intrinsic functions, `Unrestricted` and `Restricted`.

1. `Unrestricted` can be called by any function:

```
static Object Unrestricted(String op, params Object[] args)
  calls[];
```

   Example:
```
Unrestricted("Add", 1, 2); // Returns 3
```

2. `Restricted` can only be directly called by functions who name it in their `calls`:

```
static Object Restricted(String op, params Object[] args)
  calls[Restricted];
```

   Example:
```
static void WriteChar(Char c) calls[Restricted] {
          Restricted("CCall", "putchar", c); }
```

How to Solve Problem 1 (Static Dispatch)

What can `Sign(0)` do?

How to Solve Problem 1 (Static Dispatch)

What can `Sign(0)` do?

1. (indirectly) perform only `Unrestricted` operations:
   ```
   static Int32 Sign(Int32 x) calls[] {…}
   ```

How to Solve Problem 1 (Static Dispatch)

What can `Sign(0)` do?

1. (indirectly) perform only `Unrestricted` operations:
   ```
   static Int32 Sign(Int32 x) calls[] {…}
   ```
2. also (indirectly) perform *some* `Restricted` operations:
   ```
   static Int32 Sign(Int32 x) calls[WriteLine] {…}
   ```

How to Solve Problem 1 (Static Dispatch)

What can `Sign(0)` do?

1. (indirectly) perform only `Unrestricted` operations:
   ```
   static Int32 Sign(Int32 x) calls[] {…}
   ```
2. also (indirectly) perform *some* `Restricted` operations:
   ```
   static Int32 Sign(Int32 x) calls[WriteLine] {…}
   ```
3. also (indirectly) perform *any* `Restricted` operation:
   ```
   static Int32 Sign(Int32 x) calls[Restricted] {…}
   ```

How to Solve Problem 2 (Dynamic Dispatch)

What can `x.Run()` do?

How to Solve Problem 2 (Dynamic Dispatch)

What can x.Run() do?

```
interface I { void Run() calls[]; }
```

How to Solve Problem 2 (Dynamic Dispatch)

What can x.Run() do?

```
interface I { void Run() calls[]; }
```

Callability Generics

Consider this:

```
interface I<'a> { void Run() calls['a]; }
```

Example:

```
class HelloWorld: I<[WriteLine]> {
    void I.Run() calls[WriteLine] { WriteLine("Hello World!"); }}
```

How to Solve Problem 2 (Dynamic Dispatch)

What can x.Run() do?

```
interface I { void Run() calls[]; }
```

Callability Generics

Consider this:

```
interface I<'a> { void Run() calls['a]; }
```

Example:

```
class HelloWorld: I<[WriteLine]> {
    void I.Run() calls[WriteLine] { WriteLine("Hello World!"); }}
```

Now to answer the question: what can x.Run() do?

How to Solve Problem 2 (Dynamic Dispatch)

What can `x.Run()` do?

```
interface I { void Run() calls[]; }
```

Callability Generics

Consider this:

```
interface I<'a> { void Run() calls['a]; }
```

Example:

```
class HelloWorld: I<[WriteLine]> {
        void I.Run() calls[WriteLine] { WriteLine("Hello World!"); }}
```

Now to answer the question: what can `x.Run()` do?

1. Only perform `Unrestricted` operations:

```
static void M2(I<[]> x) calls[] { x.Run(); }
```

How to Solve Problem 2 (Dynamic Dispatch)

What can `x.Run()` do?

```
interface I { void Run() calls[]; }
```

Callability Generics

Consider this:

```
interface I<'a> { void Run() calls['a]; }
```

Example:

```
class HelloWorld: I<[WriteLine]> {
    void I.Run() calls[WriteLine] { WriteLine("Hello World!"); }}
```

Now to answer the question: what can `x.Run()` do?

1. Only perform `Unrestricted` operations:
   ```
   static void M2(I<[]> x) calls[] { x.Run(); }
   ```
2. Also print lines to standard-output:
   ```
   static void M2(I<[WriteLine]> x) calls[WriteLine] { x.Run(); }
   ```

How to Solve Problem 2 (Dynamic Dispatch)

What can `x.Run()` do?

```
interface I { void Run() calls[]; }
```

Callability Generics

Consider this:

```
interface I<'a> { void Run() calls['a]; }
```

Example:

```
class HelloWorld: I<[WriteLine]> {
    void I.Run() calls[WriteLine] { WriteLine("Hello World!"); }}
```

Now to answer the question: what can `x.Run()` do?

1. Only perform `Unrestricted` operations:
   ```
   static void M2(I<[]> x) calls[] { x.Run(); }
   ```
2. Also print lines to standard-output:
   ```
   static void M2(I<[WriteLine]> x) calls[WriteLine] { x.Run(); }
   ```
3. Perform any `Restricted` operation:
   ```
   static void M2(I<[Restricted]> x) calls[Restricted] { x.Run(); }
   ```

How to Solve Problem 2 (Dynamic Dispatch)

What can `x.Run()` do?

```
interface I { void Run() calls[]; }
```

Callability Generics

Consider this:

```
interface I<'a> { void Run() calls['a]; }
```

Example:

```
class HelloWorld: I<[WriteLine]> {
    void I.Run() calls[WriteLine] { WriteLine("Hello World!"); }}
```

Now to answer the question: what can `x.Run()` do?

1. Only perform `Unrestricted` operations:
   ```
   static void M2(I<[]> x) calls[] { x.Run(); }
   ```
2. Also print lines to standard-output:
   ```
   static void M2(I<[WriteLine]> x) calls[WriteLine] { x.Run(); }
   ```
3. Perform any `Restricted` operation:
   ```
   static void M2(I<[Restricted]> x) calls[Restricted] { x.Run(); }
   ```
4. Defer the decision to the caller of `M2`:
   ```
   static void M2<'a>(I<['a]> x) calls['a] { x.Run(); }
   ```

How to Solve Problem 3 (Dynamic Code Loading & Invocation)

In C♯ to dynamically invoke a static or instance method, you simply write:

```
methodInfo.Invoke(receiver, args)
```

How to Solve Problem 3 (Dynamic Code Loading & Invocation)

In C♯ to dynamically invoke a static or instance method, you simply write:

```
methodInfo.Invoke(receiver, args)
```

For our system to be sound, we could declare `Invoke` like this:

```
/// Represents a method m
class MethodInfo {
  …
  /// Throws an exception if Invoke<'a> ↛ m,
  /// otherwise calls receiver.m(args)
  Object Invoke<'a>(Object receiver, Object[] args) calls['a] { … }
  … }
```

How to Solve Problem 3 (Dynamic Code Loading & Invocation)

In C♯ to dynamically invoke a static or instance method, you simply write:

```
methodInfo.Invoke(receiver, args)
```

For our system to be sound, we could declare `Invoke` like this:

```
/// Represents a method m
class MethodInfo {
  …
  /// Throws an exception if Invoke<'a> ↛ m,
  /// otherwise calls receiver.m(args)
  Object Invoke<'a>(Object receiver, Object[] args) calls['a] { … }
  … }
```

```
static void M3(String url) {
  // Load code (possibly from the internet)
  Assembly code = Assembly.LoadFrom(url);
  // call M(), but only if it can only perform Unrestricted operations
  code.GetMethod("M").Invoke<[]>(null, null); }
```

Conclusion

1. No need to look at the body of methods to determine what they can do.
2. No need to look at *every* piece of code we are compiling with.
3. Our reasoning is static and consistently sound.

Conclusion

1. No need to look at the body of methods to determine what they can do.
2. No need to look at *every* piece of code we are compiling with.
3. Our reasoning is static and consistently sound.

Future Work

- Make our annotations less verbose:
  - Infer `calls` annotations?
  - Have a `cant-call` annotation?
  - Support wild-cards?
  - Allow named groups of functions?
- Support sound reasoning about unsafe operations (like executing arbitrary machine code).
- Improve support for dynamic code loading:
  - Allow calling newly loaded functions, even if they have themselves in their `calls` annotation.
- Formalise the reasoning properties we want from our system:
  - Prove them!