Practical (and not so practical) Uses for the Mono Interpreter

RYAN DAVIS

Queensland C# Mobile Developers Meetup

2019 05 28

whoami

- Ryan Davis
- Professional ~~Mobile~~ LINQPad Developer

 ryandavis.io  rdavis_au

 rdavisau

- **essential-interfaces** – use DI/mocking with Xamarin.Essentials
- **dumpeditable-linqpad** – extensible inline object editor for LINQPad
- **jsondatacontext-linqpad** – json data context driver for LINQPad
- **sockets-for-pcl, sockethelpers** – socket comms in a PCL
  (today you should use netstandard sockets why are you all still installing this)

to cover

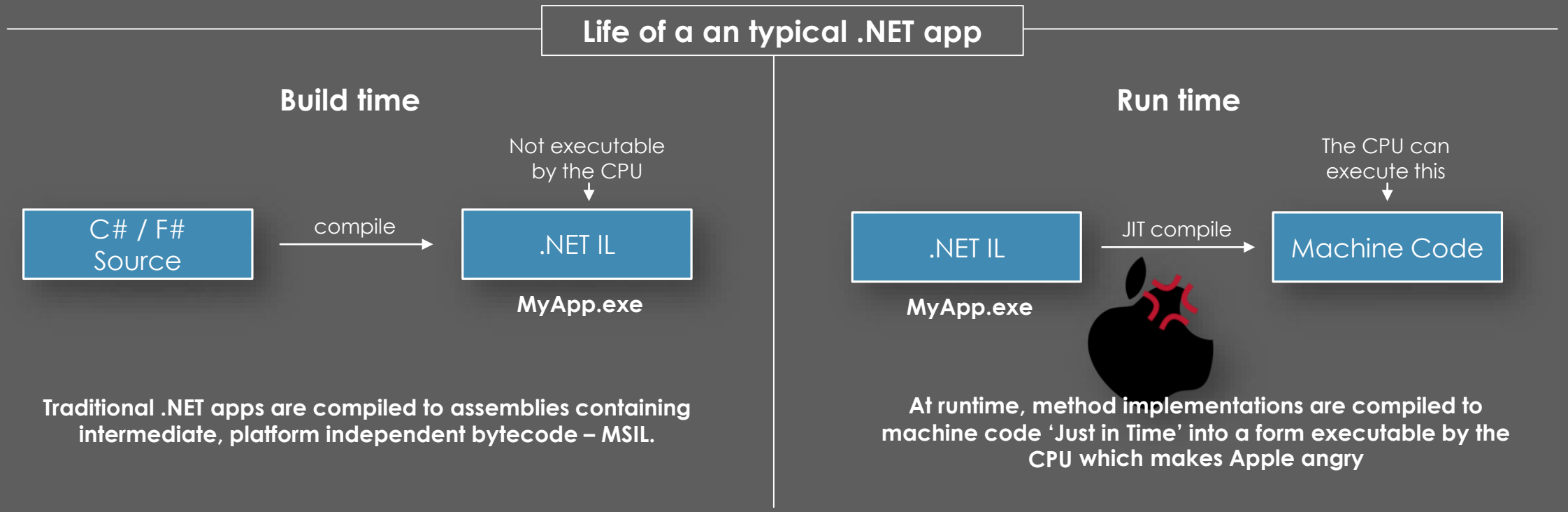
- what is the mono interpreter
- practical uses
- samples and demos
- resources

-= practical uses for the mono interpreter=-

what is it?

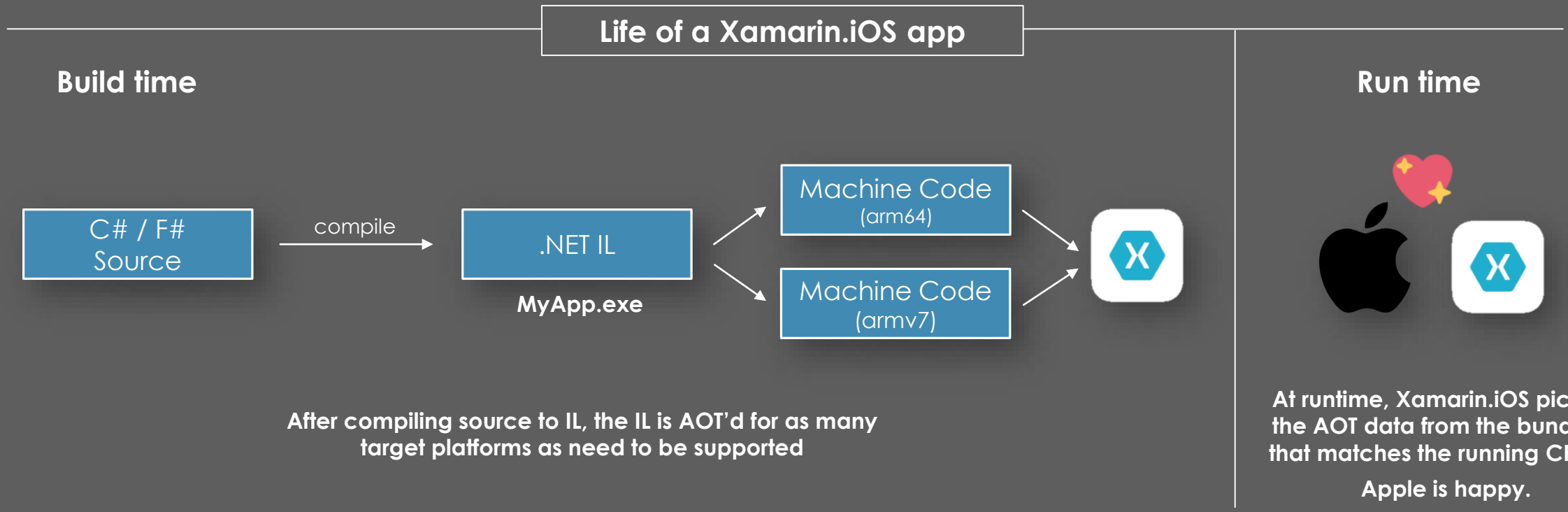
Putting .NET on iOS posed a major challenge…

Apple explicitly forbids the use of runtime code generation and execution.
NET apps are traditionally executed using JIT compilation – a form of codegen.



..but Xamarin had an answer!

Xamarin developed an AOT compiler to allow .NET applications to run on iOS.
The AOT compiler turns IL into architecture-specific machine code at build time.



AOT has many benefits, but also drawbacks

- ✓ **AOT'ing enables Xamarin.iOS**
- ✓ an AOT'd application (generally) outperforms the same app JIT'ed at runtime
- ✓ certain errors surface during compilation that would otherwise occur at JIT time

However:

- x AOT'ing produces larger binaries
- x AOT'ing involves longer build times
- x "AOT only" execution essentially prohibits dynamic execution, which causes challenges for various kinds of development and use cases in .NET

enter the revived Mono Interpreter

(and mixed mode execution)

A new runtime option that enables dynamic execution opportunities and size/performance tradeoffs for Xamarin.iOS, whilst remaining within the restrictions imposed by Apple and the iOS runtime.

Updated interpreter: enables the 'execution' of .NET IL without code generation

Mixed-AOT mode: enables combined execution of AOT'd code and interpreted IL

In practice:

- Enables use of APIs like `Assembly.Load` and advanced techniques using the `dynamic` keyword
- Enables code generation using methods like `Reflection.Emit`

recently announced preview for Xamarin.iOS

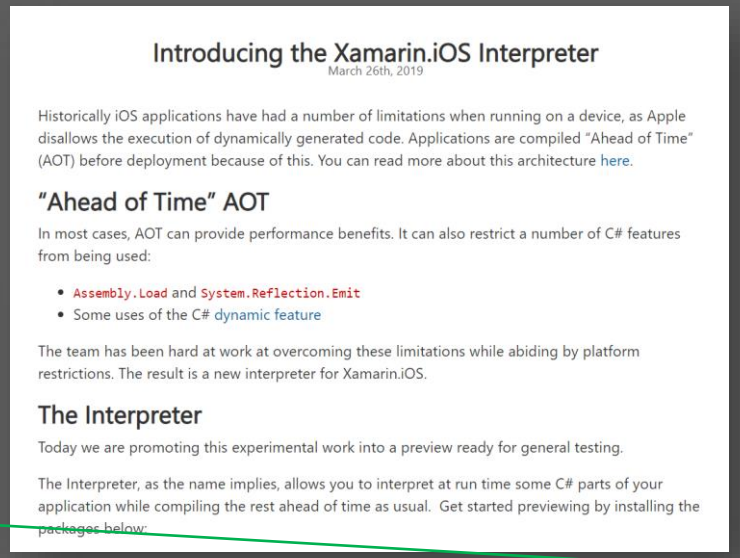
Mono's New .NET Interpreter

Miguel de Icaza November 13, 2017 runtime

Mono is complementing its Just-in-Time compiler and its static compiler with a .NET interpreter allowing a few new ways of running your code.

In 2001 when the Mono project started, we wrote an interpreter for the .NET instruction set and we used this to bootstrap a self-hosted .NET development environment on Linux.

Interpreter updated late 2017



Introducing the Xamarin.iOS Interpreter

March 26th, 2019

Historically iOS applications have had a number of limitations when running on a device, as Apple disallows the execution of dynamically generated code. Applications are compiled "Ahead of Time" (AOT) before deployment because of this. You can read more about this architecture here.

"Ahead of Time" AOT

In most cases, AOT can provide performance benefits. It can also restrict a number of C# features from being used:

- `Assembly.Load` and `System.Reflection.Emit`
- Some uses of the C# dynamic feature

The team has been hard at work at overcoming these limitations while abiding by platform restrictions. The result is a new interpreter for Xamarin.iOS.

The Interpreter

Today we are promoting this experimental work into a preview ready for general testing.

The Interpreter, as the name implies, allows you to interpret at run time some C# parts of your application while compiling the rest ahead of time as usual. Get started previewing by installing the packages below:

Mixed-mode + interp announced for Xamarin.iOS in March 2019, hiding in builds for months prior

Branch: master ▾ mono / mono / mini / interp / transform.c

CoffeeFlux [coop] Transition various public APIs into an external/internal form ...

19 contributors

6167 lines (5674 sloc) | 186 KB

```
1  /**
2   * \file
3   * transform CIL into different opcodes for more
4   * efficient interpretation
5   *
6   * Written by Bernie Solomon (bernard@ugsolutions.com)
7   * Copyright (c) 2004.
8   */
```

Has its roots in a 2001 relic!

major effort

▲ migueldeicaza on Nov 13, 2017 [-]

It is worth pointing out that when we dropped the interpreter, we only had two or three engineers working on the VM and they had to both develop the JIT and maintain the interpreter, plus work on the GC, io-layer and other VM features.

Without a reason to keep the interpreter (the world was a JIT-friendly place back then), it made no sense to maintain it.

But times change, statically compiled environments are more common nowadays (iOS, PlayStation, Xbox, tvOS, watchOS) and with it the need to have dynamic capabilities.

To put things in perspective, adding generics to the revived interpreter probably took an engineer that was not familiar with .NET about 4-6 weeks of work.

Miguel's comment on some of the rationale behind the revival

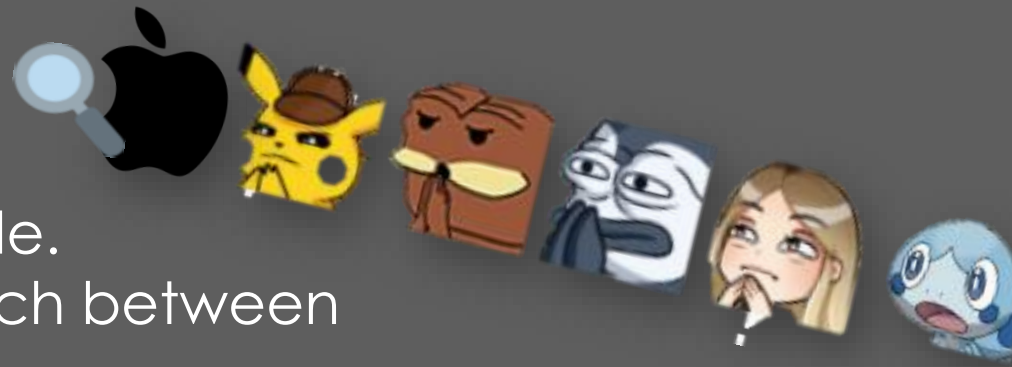
There are probably more contributors than this

the interpreter affords us dynamic execution

In general, an interpreter produces an execution-like result from non-machine executable input eg:

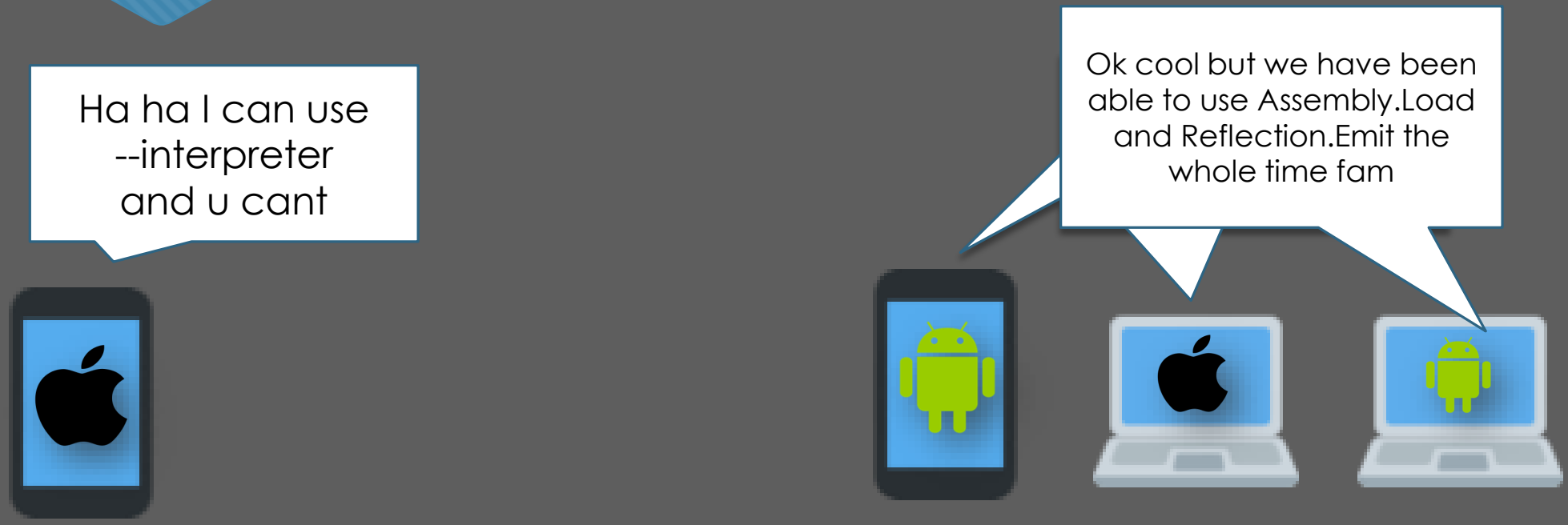
- Source code of a programming language (javascript, python)
- Machine code of a different architecture (emulating a Gameboy cpu)

So, we can use the mono interpreter to process IL instead of JIT'ing it - giving us 'execution' of dynamic code without executing it.



Interpreted IL is significantly slower than AOT'd code.
Thanks to mixed-mode execution, we can use switch between AOT'ing and interpreting where it makes sense.

but --interpreter is currently ios device only



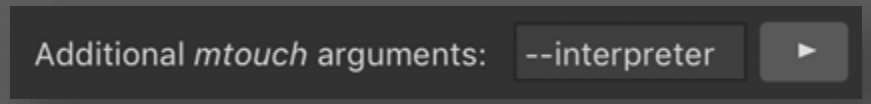
The iOS device target has the most to gain from an interpreter, given the iOS simulator and android devices/emulators all support JIT compilation.

Our practical uses therefore fall into two categories:

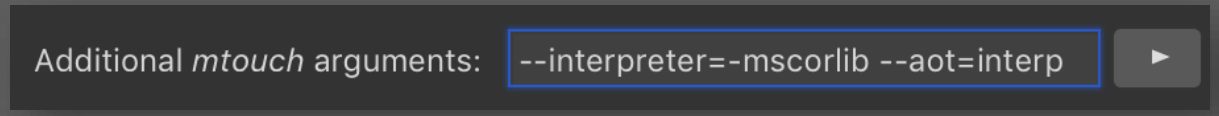
- Improvements to development time experience for device specific features
- Fundamentally new opportunities for release capabilities

how 2

Add `--interpreter` to your `mtouch` args



Without arguments, `--interpreter` actually expands to something like this:



Meaning, "interpret everything except **mscorlib**, and enable mixed execution"

With this set of flags, any time an assembly with IL and no AOT data is encountered, the runtime will fall back to the interpreter to execute it.

how 2 actually

```
}

static void
interp_delegate_ctor (MonoObjectHandle this_obj, MonoObjectHandle target, gpointer addr, MonoError *error)
{
        /*
                 ad  i    result of an LDFTN opcode, i.e. an InterpMethod

                      met      (I     Method*)addr;

                                                    METHOD_ATTRIBUTE_STATIC)) {
                                                    no_get_delegate_invoke_internal (mono_handle_class (this_obj));
                                                    ates must not have null check */
                                                    re_internal (imethod->method)->param_count == mono_method_signature_internal (invoke)
                                                    _HANDLE_IS_NULL (target)) {
                                                    argument (error, "this", "Delegate to an instance method cannot have null 'this'");
```

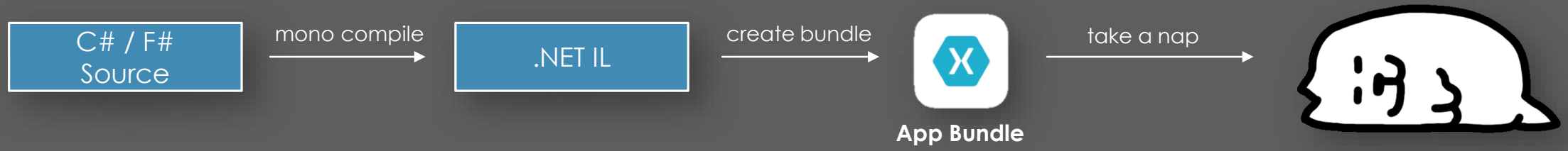
-= practical uses for the mono interpreter=-

#1 Inner-loop development speed

default interpreter options disable AOT

Ordinary AOT debug build

--interpreter debug build

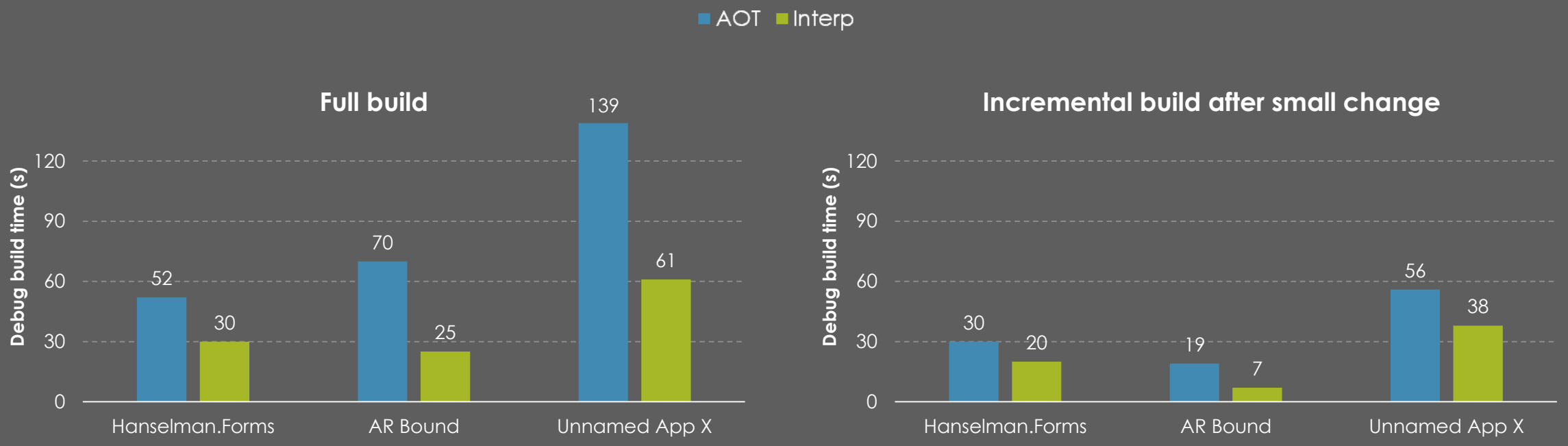


Add `--interpreter` to your debug configuration to save time and energy!

skipping the AOT step improves compile times

Highly Unscientific But Possibly Real World Representative* Comparison of Build Times between AOT and non-AOT (--interpreter) iOS Debug Device Builds

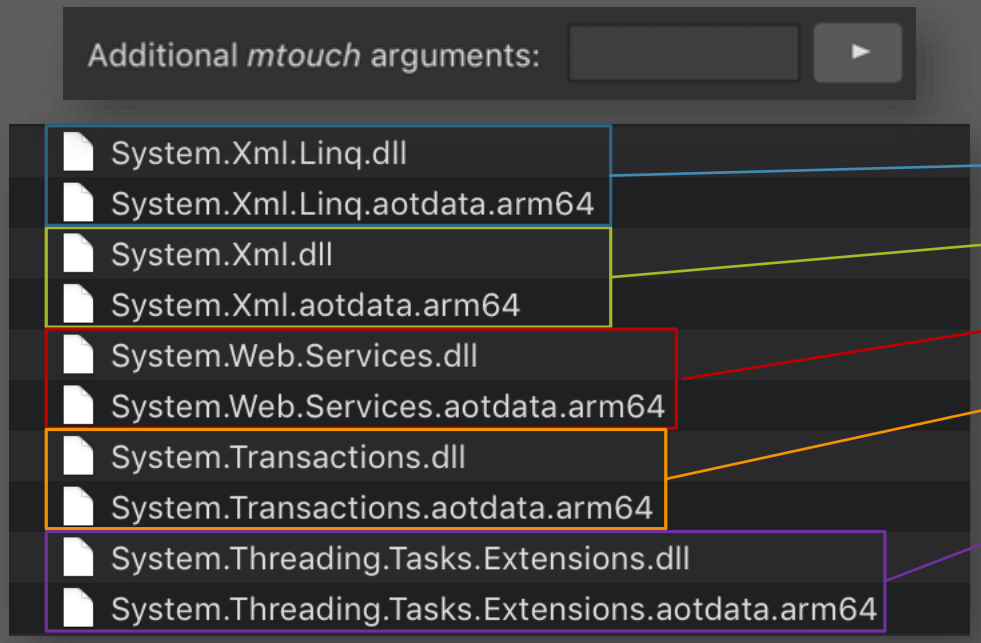
(lower is better)



* Performed while at least one twitch stream was playing, a zillion chrome tabs were open, Parallels VM was on and I was also running Slack

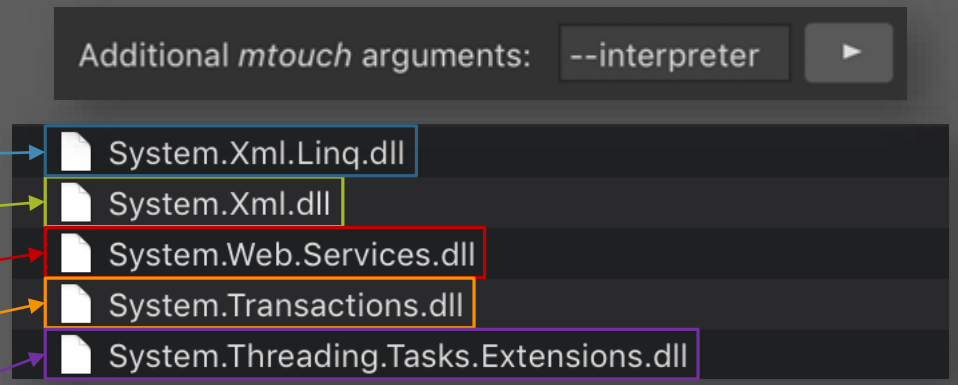
Look ma, no aotdata!

Ordinary AOT build




```
.method public final hidebysig virtual newslot instance bool
  Equals(
    valuetype System.Threading.Tasks.ValueTask other
  ) cil managed noinlining
{
  .maxstack 8

  IL_0000: ret

}
```

--interpreter build

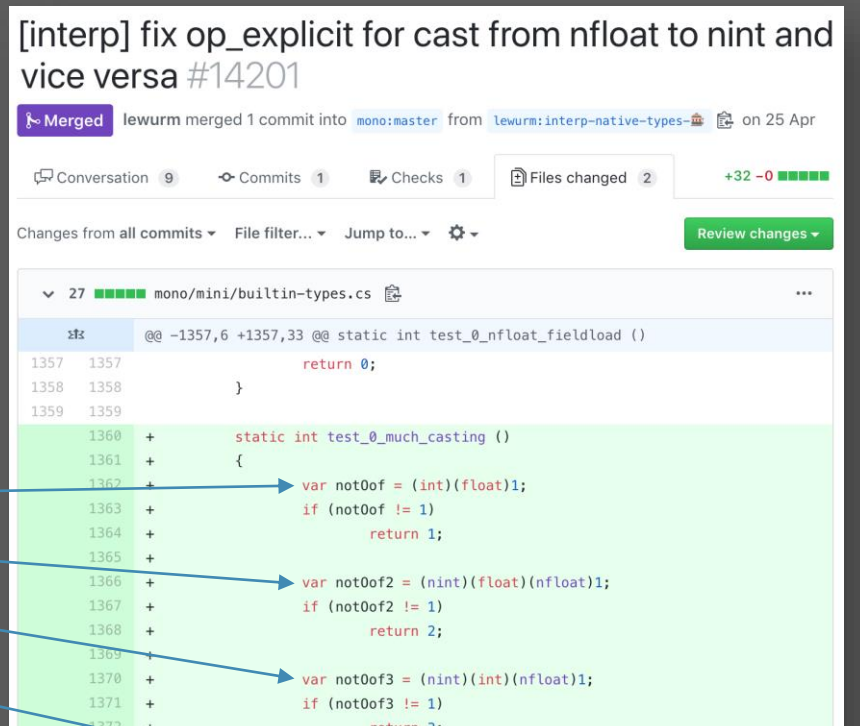



```
.method public final hidebysig virtual newslot instance bool
  Equals(
    valuetype System.Threading.Tasks.ValueTask other
  ) cil managed
{
  .maxstack 8

  // [70 7 - 70 35]
  IL_0000: ldarg.0      // this
  IL_0001: ldfld        object System.Threading.Tasks.ValueTask::_obj
  IL_0006: ldarg.1      // other
  IL_0007: ldfld        object System.Threading.Tasks.ValueTask::_obj
  IL_000c: bne.un.s     IL_001d

  // [71 9 - 71 56]
  IL_000e: ldarg.0      // this
  IL_000f: ldfld        int16 System.Threading.Tasks.ValueTask::_token
  IL_0014: ldarg.1      // other
  IL_0015: ldfld        int16 System.Threading.Tasks.ValueTask::_token
  IL_001a: ceq
  IL_001c: ret

  // [72 7 - 72 20]
```

Inner loop development – practical use? ✔

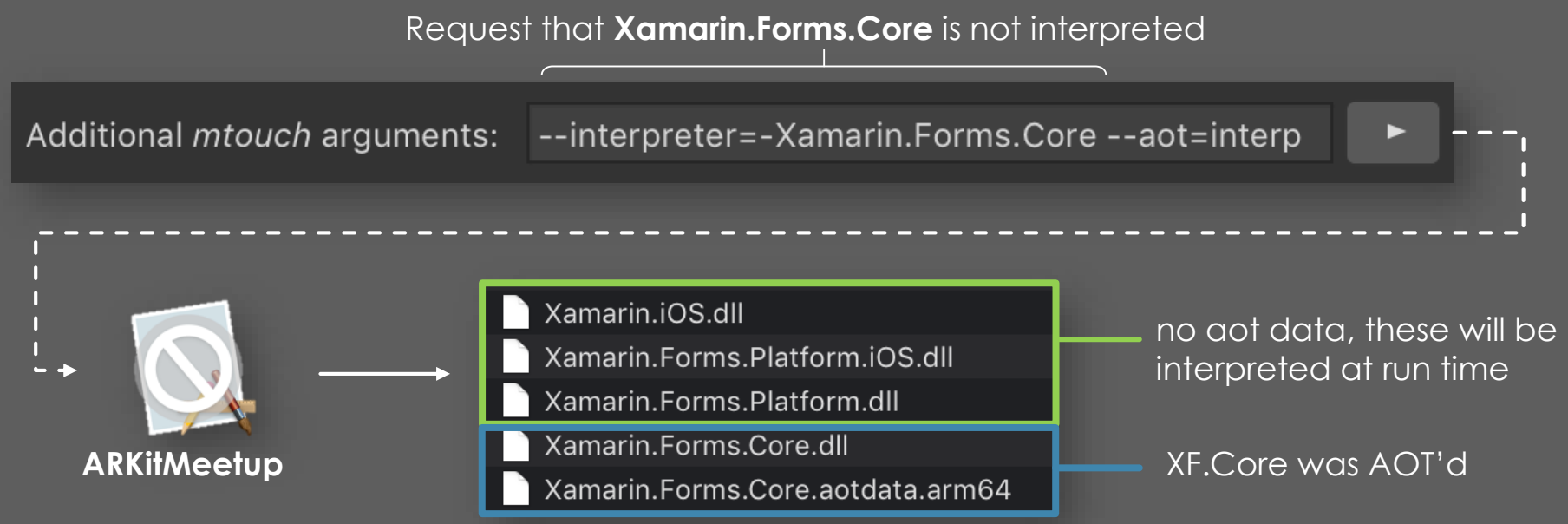
Lots of benefits, only minor drawbacks:

- Lower performance than AOT'd *(debug builds don't represent real performance anyway)*
- May encounter a bug in the interpreter *(but then you'll report it and be helping the world)*



inner loop development – tips

- Handle bugs or performance sensitive code by selectively AOT'ing assemblies:
  - `--interpreter=-AssemblyToAOT` will cause the assembly to be AOT'd, not interpreted
- To verify that the right parts are/aren't being interpreted, inspect the app bundle:



-= practical uses for the mono interpreter=-

#2

Hot reload (of device-only features)

device features are the most painful to debug

- alternating between typing on the pc and working with the device
- work that requires movement, being away from the pc etc (e.g. ARKit)
- work that requires fiddling and lacks tooling (e.g. ARKit)
- longer deploy times (even with --interpreter)
- hot reload is the hero we need

device features are things like

ARKit

Metal

Camera

Barcode

SceneKit*

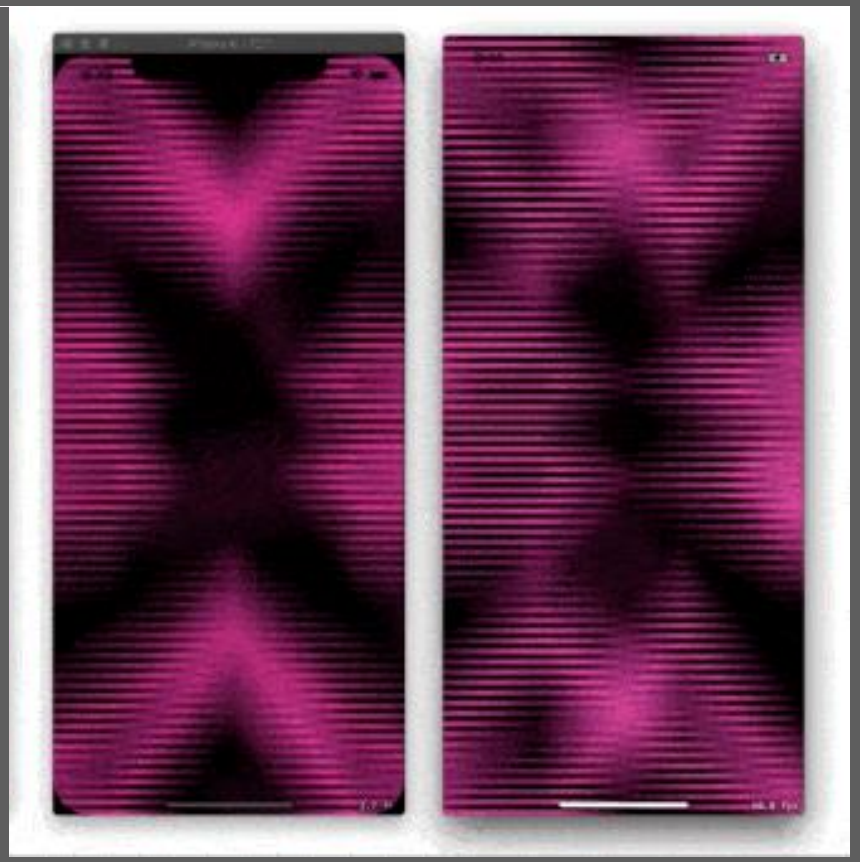
SpriteKit*

Push Notifications

* these do work on the simulator but with unusable performance

hot reload

~~damn right u can use continuous~~ demo



Simulator performance vs device performance
OpenGLES

```
// now we can create a geometry
var box = new SCNBox
{
    Width = size.Width,
    Length = size.Height,
    Height = 0.001f, // because cards are flat,
    ChamferRadius = 0
};

// make it reddish
box.FirstMaterial.Diffuse.Contents = UIColor.Red.ColorWithAlpha(.5f);

// create a node with this geometry
var cardNode = new SCNNode
{
    Geometry = box
};

// add it to the node we were given
node.Add(cardNode);

// cool!
// but red is too angry

ublic override void ViewWillAppear(bool animated)
{
    base.ViewWillAppear(animated);
```

Merge
Call Stack | Continuous Coding | Breakpoints | Immediate | Application Output - ARKitMeetup
Clear Edits

Hot reloading ARKit

hot reload – practical use? ✔

Lots of benefits, some drawbacks:

- no endorsed hot reload solutions
- hot reload + interpreter is an additional level of complication over interpreter alone – some bugs exist in this combination that don't exist in normal use

hot reload - tips

- tailor your hot reload setup to the task at hand
- consider what state should survive between changes e.g.:
  - UI – none or viewmodel state
  - 2D AR – AR view but not AR state
  - 3D AR – AR view and AR state

-= practical uses for the mono interpreter=-

#3

Hot patching

releasing on ios can be scary

- Apple review basically guarantees at least 8 hours of lead time for any release/fix
- Apple scrutiny is very inconsistent
- Maybe it would be nice to patch our app outside of the normal release process
- ~~Maybe it wouldn't?~~

i execute, therefore i patch

Transparent hot patching would need lots of runtime magic that doesn't (yet?) exist

We can roll our own w/**`Assembly.Load`**, but our app must 'expect' to be patched

Fortunately, .NET tends towards abstraction and loose-coupling:

```
public AboutViewModel(
    INavigationService navigationService,
    IFeatureService featureService,
    IUserService userService)
```

```
DependencyService.Get<IUserService>(); // good

DependencyService.Get<UserService>(); // not good
```

Code not tied to specific implementations, easy to replace with hot patch

```
await _navigationService.Navigate("myapp://home");

await _navigationService.PushAsync<LoginViewModel>();
```

Navigator calls not coupled to view or viewmodel implementations

```
Scenes =
    AppDomain.CurrentDomain
        .GetAssemblies()
        .SelectMany(x => x.GetTypes())
        .Where(x => typeof(BaseARViewController).IsAssignableFrom(x))
        .Where(x => !x.IsAbstract)
        .OrderBy(x => x.Namespace)
        .Select(x =>
```

Dynamic menu contents, easy to augment with hot patch

roll your own hotpatch in 3 easy steps

1. detect and download hot patch if available
   - simplest case: .dll, complicated case: bundle with dlls, assets, etc.
   - can do in the background to keep checks off the startup path

2. load patch contents at every startup (volatile patching)

3. integrate patch content at appropriate points, for example:
   - add/override or intercept service registration
   - add/replace navigator references
   - any other hard coded patch handling

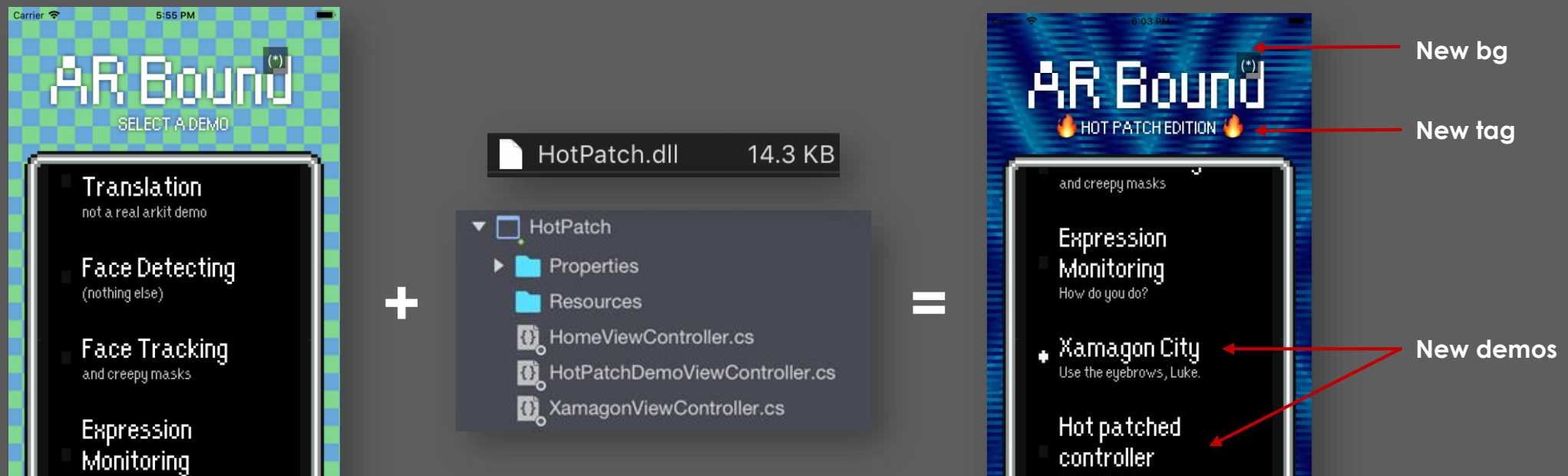
home grown hot patching – demo (ar bound)

```
Scenes =
    AppDomain.CurrentDomain
        .GetAssemblies()
        .SelectMany(x => x.GetTypes())
        .Where(x => typeof(BaseARViewController).IsAssignableFrom(x))
        .Where(x => !x.IsAbstract)
        .OrderBy(x => x.Namespace)
```

Since menu contents are generated dynamically, just loading the hot patch is enough to add new demos to it

```
private UIViewController ProcessHotPatch(byte[] patchData)
{
    var asm = Assembly.Load(patchData);
    var patchedHomeViewControllerType =
        asm.GetTypes().FirstOrDefault(x => x.Name.EndsWith("HomeViewController"));

    return patchedHomeViewControllerType != null
        ? Activator.CreateInstance(patchedHomeViewControllerType) as UIViewController
        : null;
}
```

Convention in patch loader – “Prefer a patched HomeViewController over the compiled one”



home grown hot patching – demo (prism)

```
private void ProcessHotPatch(byte[] patchData)
{
    var asm = Assembly.Load(patchData);
    var serviceTypes =
        asm.GetTypes()
           .Where(x => x.IsSubclassOf(typeof(ServiceBase)));

    var pageTypes =
        asm.GetTypes()
           .Where(x => x.IsSubclassOf(typeof(Page)));

    PatchServices(serviceTypes);
    PatchPages(asm, pageTypes);
}
```

Create dedicated patching implementations for different types of patch content

```
private void PatchServices(IEnumerable<Type> serviceTypes)
{
    foreach (var st in serviceTypes)
        foreach (var @if in st.GetInterfaces())
            ContainerRegistry.Register(@if, st);
}
```

```
private void PatchPages(Assembly asm, IEnumerable<Type> pageTypes)
{
    foreach (var p in pageTypes)
    {
        // use bad code to determine expected vm name
        var pageName = p.Name.Split('.').Last().Replace("Page", "");
        var vmName = $"{pageName}ViewModel";

        // check for patched vm
        var vmType =
            asm.GetTypes()
               .FirstOrDefault(t => t.Name.EndsWith(vmName));

        // register vm for page
        if (vmType != null)
            ViewModelLocationProvider.Register(p.Name, vmType);

        // register page
        ContainerRegistry.RegisterForNavigation(p, p.Name);
    }
}
```

Register new services, pages and viewmodels in the standard Prism manner

hot patching – practical use? the good

- Changes can be deployed and integrated extremely quickly, various options available to keep startup impact low
- Using mixed-AOT allows everything originally shipped to be AOT'd and only the incoming patch contents to be interpreted, minimal performance impact
- Hot patching as a concept is blessed by Apple, and "proven" by React Native

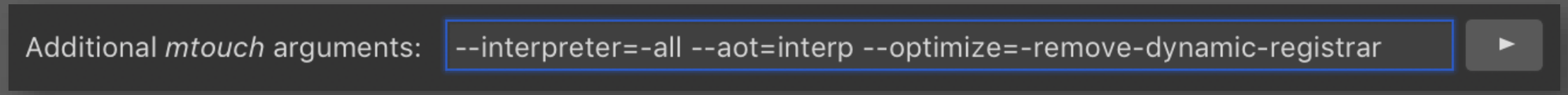
> **3.3.2** Except as set forth in the next paragraph, an Application may not download or install executable code. Interpreted code may be downloaded to an Application but only so long as such code: (a) does not change the primary purpose of the Application by providing features or functionality that are inconsistent with the intended and advertised purpose of the Application as submitted to the App Store, (b) does not create a store or storefront for other code or applications, and (c) does not bypass signing, sandbox, or other security features of the OS.

hot patching – practical use? the bad

- Increases versioning complications
  - Can fragment userbase – clients who do/don't have hot patches
  - If patches cause side effects, user state is no longer easy to reason about
- Patching significant changes is a great way to see how effective the linker is 
- Certain classes of errors might be uncatchable and unrecoverable, or present in sections of the app without error handling
- Allowing execution of code from a remote source has many security concerns.

hot patching – tips

- Use `--interpreter=-all` to ensure all original code is AOT'd, and disable removal of the dynamic registrar if your patch will include types deriving from native types



A reasonable set of hot-patch friendly mtouch arguments

- Try this at home, or maybe with QA builds, not in production

- Feature flag it, include a rollback/unpatch allowance, don't @ me

-= practical uses for the mono interpreter=-

#4 Embedded repl

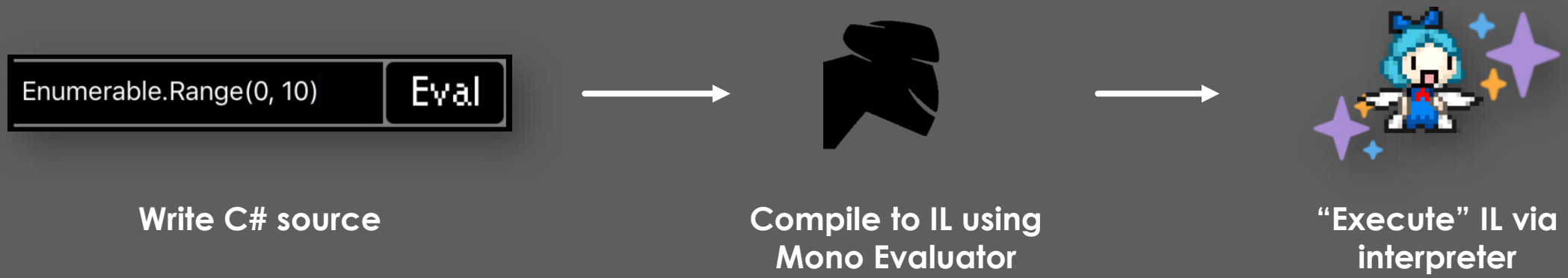
sometimes you want to code inside your app*

- device related features like AR can be fiddly and highly state-dependant
- you can persist state when hot reloading, but complicated preservation usually pollutes code
- sometimes you're not at your PC when you want to fiddle programmatically with your app?
- dynamically executing code within the context of the running app has its uses, probably

*citation needed

a repl is possible w/the evaluator + interpreter

- the mono interpreter is an IL interpreter, but we'd prefer not to write IL
- we can approximate a c# repl by using the mono evaluator to generate IL from c#, which the interpreter then executes

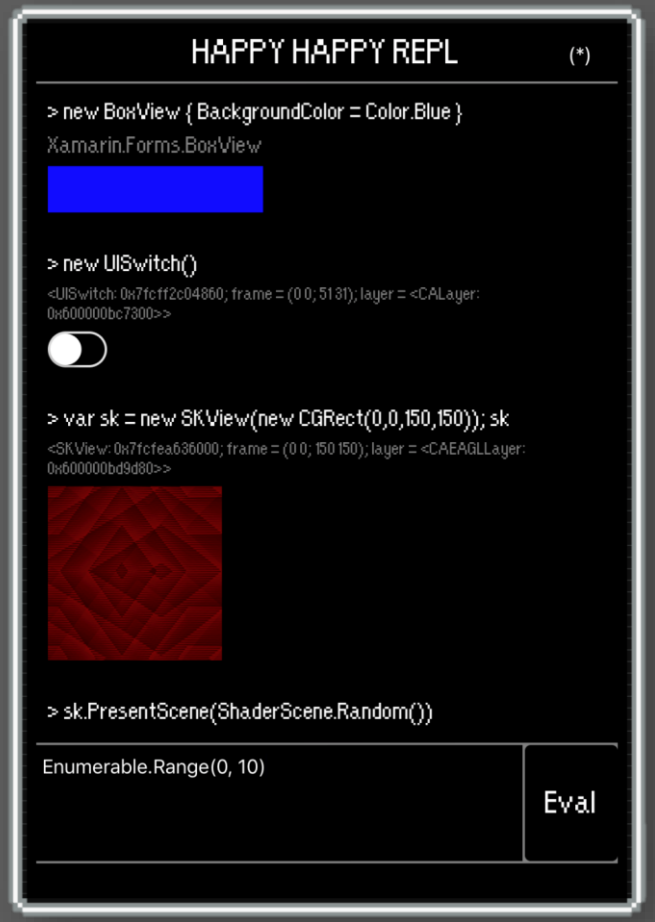


Write C# source

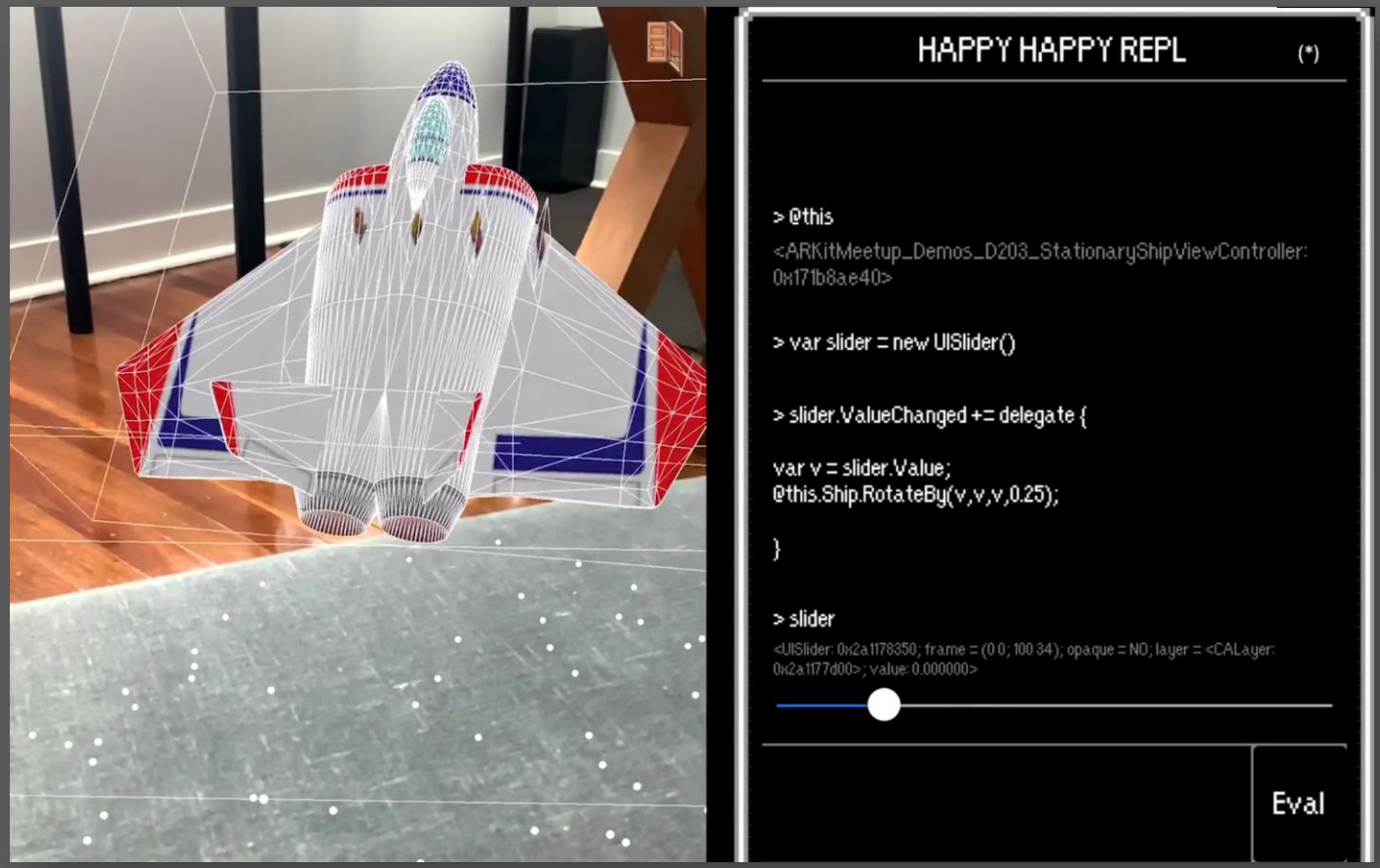
Compile to IL using Mono Evaluator

"Execute" IL via interpreter

embedded repl – demo

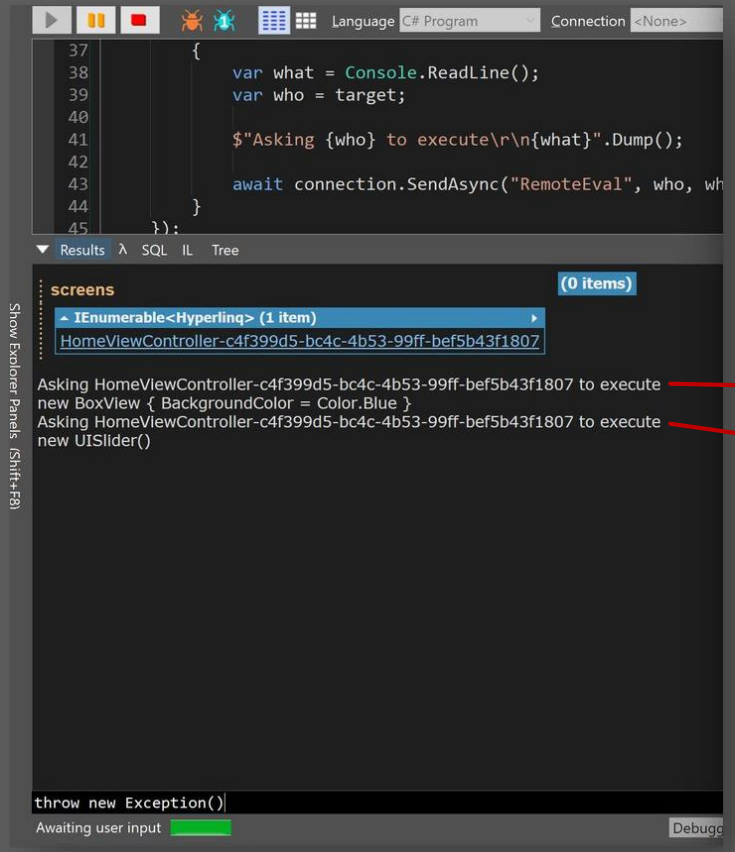


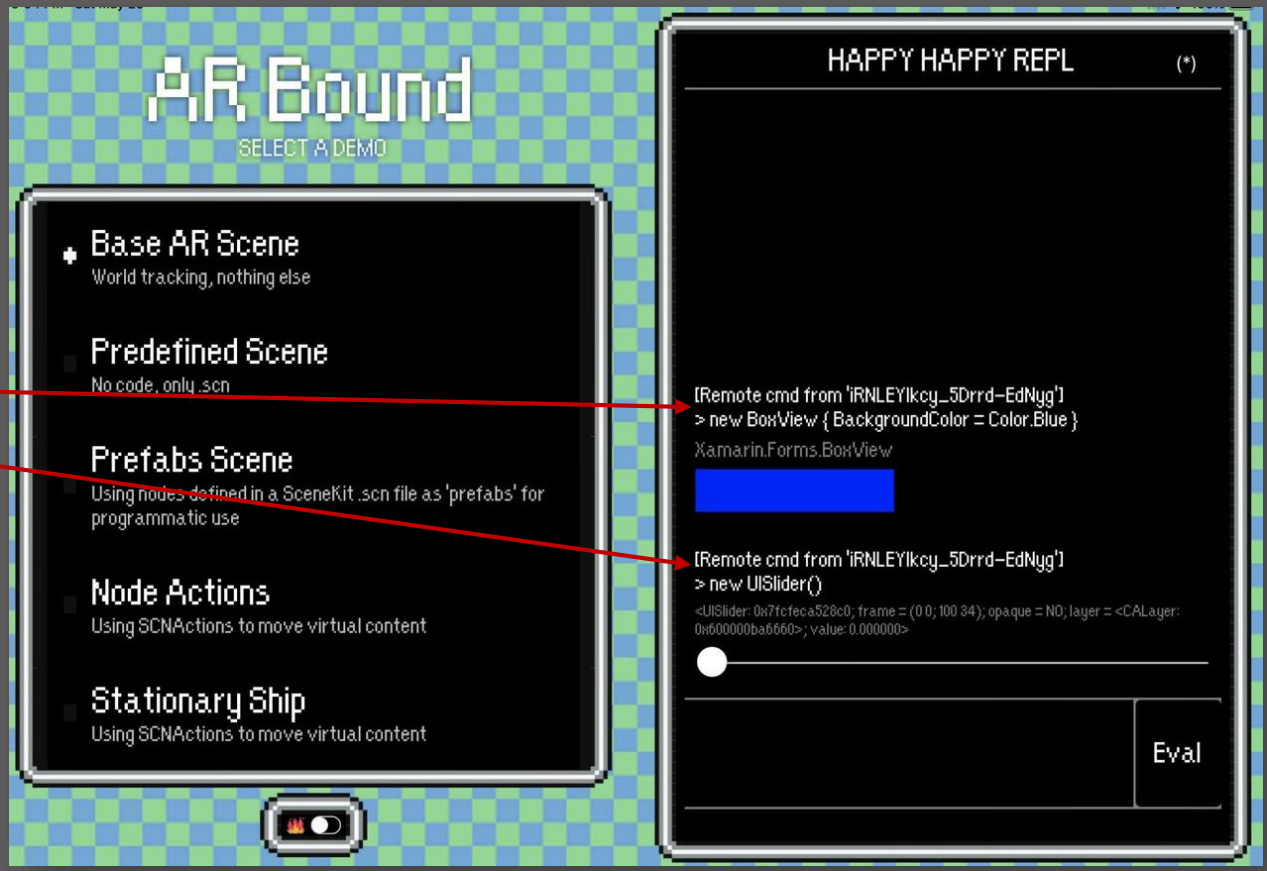
Evaluate C# at runtime on the device



Interact with running application from REPL

embedded repl – demo





Send code to be remotely evaluated

embedded repl, remote-eval – practical use?

- this was meant to be the meme use for the interpreter but it was actually kind of cool
- generalising to the ideas of arbitrary and remote execution there are a lot of practical uses
- the same security considerations that apply to hot patching apply here if you want to use it in production

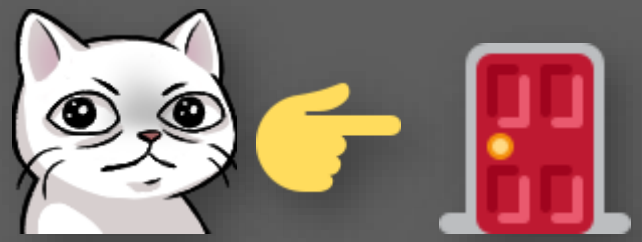
embedded repl – tips

- use an updated version of **Mono.CSharp.dll** from your Xamarin install, not the one on NuGet. It has all the MCS features and fixes that have been implemented since 2015.

Mono.CSharp
Mono C# Compiler

Id	Mono.CSharp
Author	Mono Development Team
Published	8/05/2015
Downloads	145,356

Old busted



Name	Date Modified	Size
Mono.CSharp.dll	21 Jan 2019 at 11:10 am	1.3 MB

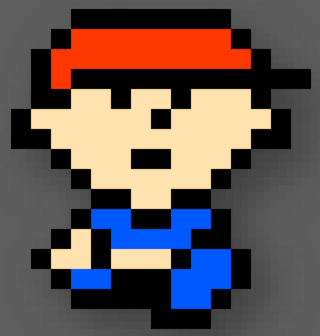
New shiny!

-= practical uses for the mono interpreter=-

wrapping up

how to start your interpreter adventures

Easy Mode – Interpreter only

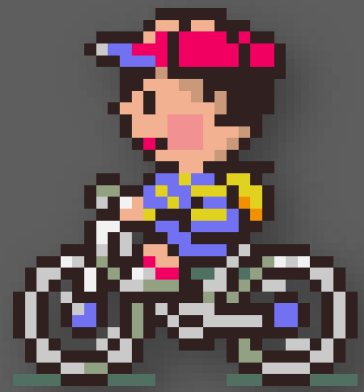


#1 Inner loop dev speed

#3 Hot patching

Although the feature itself is in preview, any recent stable Xamarin.iOS build supports the `--interpreter` flag

Hard Mode – Interpreter + Code Gen



#2 Hot Reload

#4 Embedded REPL

For code generation (`System.Reflection.Emit`) you need a Xamarin.iOS build on top of a mono runtime that doesn't cut Emit out:

- download one from Xamarin here
- or bake your own

useful resources

- **Interpreter blog posts**
  https://devblogs.microsoft.com/xamarin/introducing-xamarin-ios-interpreter/
  https://www.mono-project.com/news/2017/11/13/mono-interpreter/
- **iOS App Architecture**
  https://docs.microsoft.com/en-us/xamarin/ios/internals/architecture
- **Hot Reloading iOS "Device-Only" features with the new Mono Interpreter**
  https://ryandavis.io/hot-reloading-device-only-features-with-the-new-mono-interpreter/
- **Interpreter source (for the brave, or if you want to follow the history)**
  https://github.com/mono/mono/tree/master/mono/mini
- **Xamarin iOS/macOS gitter**
  https://gitter.im/xamarin/xamarin-macios

questions