Intro to Trimming and AOT

Andy Gocke

Trimming

- Tree shaking: hold on to everything reachable, and remove everything else
- Problem: what's "reachable"?
- IL not a problem
- Reflection may be a problem

```
var text = Console.ReadLine();
var type = Type.GetType(text);
var instance = Activator.CreateInstance(type);
```

Static Analysis

- Trimmer (linker/AOT compiler) produce warnings for all code which can’t be statically analyzed
  - Reflection
  - Assembly load
  - COM interop
  - `dynamic`
  - RequiresUnreferencedCodeAttribute
- Trim/AOT compatible = no warnings and minimal behavior changes (don’t just throw in AOT)

DynamicallyAccessedMembersAttribute

- Some reflection can be analyzed
- Applies to type parameters or System.Type
- [DynamicallyAccessedMembers(DynamicallyAccessedMemberTypes.All)]
- Equivalent to generic constraint, constraints must match on assignment or warning occurs
- Requirements settled at substitution or call to `typeof()`

```
class Type : MemberInfo, IReflect
{
 public ConstructorInfo? GetConstructors();
 public MethodInfo[] GetMethods()
 public FieldInfo[] GetFields();
 public PropertyInfo[] GetProperties()
}
```

```
enum DynamicallyAccessedMemberTypes
{
        PublicParameterlessConstructor,
        PublicConstructors,
        NonPublicConstructors,
        PublicMethods,
        PublicFields,
        PublicProperties,
    …
}
```

Limitations

- Runtime type info (object.GetType() vs typeof())
- Recursion
- Complex relationships, e.g. “keep properties on generic arguments”
- Serialization (all the above)
- Compiler-generated code

Compiler-generated code

```
void Method([DAM(DAMT.PublicMethods)] Type type)
{
    LocalFunc();
    void LocalFunc()
    {
        foreach (var method in type.GetMethods())
        {
            Console.WriteLine(method.Name);
        }
    }
}
```

```
struct Disp
{
    public Type type;
}

void Method([DAM(DAMT.PublicMethods)] Type type)
{
    Disp disp = new { type = type; };
    LocalFunc(ref disp);
}

static void LocalFunc(ref Disp P_0)
{
    foreach (var method in P_0.type.GetMethods())
    {
        Console.WriteLine(method.Name);
    }
}
```

Compiler-generated code

```
void Method<[DAM(DAMT.PublicMethods)]T>()
{
    Action action = () =>
    {
        foreach (var method in typeof(T).GetMethods())
        {
            Console.WriteLine(method.Name);
        }
    };
    action();
}
```

```
sealed class Env<T>
{
    public static readonly Env<T> _s = new Env<T>();

    public static Action a;

    internal void M()
    {
        foreach (var m in typeof(T).GetMethods())
        {
            Console.WriteLine(m.Name);
        }
    }
}
```

Aside: Lambda Calculus

- $x \;=>\; y(x)$
  - y: $x \;+\; 1$
  - $(x \;=>\; y(x))[y\text{: } x \;+\; 1]$
  - ✗ $(x \;=>\; (x+1)(x))$
  - Not the same x
- α-renaming
  - $(x \;=>\; y(x))[x := z]$
    - $z \;=>\; z(x)$
  - $(z => \; y(z))[y\text{: } x \;+\; 1]$
  - ☑ $z \;=>\; (x+1)(z)$

α-equivalence

```
void Method<T>()
 where T : IPublicMethods
{
   () => typeof(T)
}
```

```
class Env<T>
 where T : IPublicMethods
{
   internal void M()
   {
              typeof(T)
   }
}
```

Serialization

- Most common source of incompatibility
- Jackpot of all possible problems
  - Recursion
  - Reflection-based engines and object models
  - Complex dependencies (runtime and attribute-based converters)
  - Worst part: every serializer is different

Example: System.Text.Json

```
public class WeatherForecast
{
    [JsonConverter(typeof(DateTimeOffsetJsonConverter))]
    public DateTimeOffset Date { get; set; }
    public List<Guid> Guids { get; set; }
    public IEnumerable<string>? Summaries { get; set; }
}

JsonSerializer.Serialize(weatherForecast, new JsonSerializerOptions
{ Converters = { new GuidConverter() } });

public class GuidConverter : JsonConverterFactory { … }
```

Aside: “Polymorphic deserialization”

- C# users are reinventing discriminated unions, but worse

```
[JsonDerivedType(typeof(WeatherForecastBase), typeDiscriminator: "base")]
[JsonDerivedType(typeof(WeatherForecastWithCity), typeDiscriminator: "withCity")]
public class WeatherForecastBase
{ … }

public class WeatherForecastWithCity : WeatherForecastBase
{
   public string? City { get; set; }
}
// Sample output: {
// "$type" : "withCity",
// "City": "Milwaukee",
// … }
```

Everything Is Serialization

- JSON/XML/YAML/INI/CSV/TOML/etc
- ORMs—Entity Framework, Dapper, etc
- Complex logging
- Complex tracing (e.g. EventSource self-describing events)

Compatible Solutions

- Support a finite list of types (e.g., primitives)
- Manual serialization (convert to and from types manually)
- Source generation
  - Generate from external data (protobuf)
  - Generate from type definitions (source generator)
- Most existing libraries can only be handled with a source generator

Source Generators

- Very high difficulty, usually needs custom knowledge for every library
- Anonymous types are a dead end
- Two problems to solve:
  - Generate the implementation
  - Discover/invoke the implementation

Other Languages

- Other AOT languages: Go, Swift, Rust, Kotlin/Java (Graal Native Image)
- Go: limited automatic serialization for JSON, manual in advanced scenarios
- Java: manual “reflection preservation” files. No static analysis.
- Swift, Rust, Kotlin: all different flavors of the same thing

Swift, Rust, Kotlin

- *Serialization* is not *parsing*.
- Parsers understand data formats, serializers understand types.
  - Serializer: “Write a property named, ‘P’ with type ‘int’, and value ‘5’”
  - JSON Parser/Formatter: “For a type write, ‘{‘, a property name write ‘”P”’, a value write ‘,’, and an int write ‘5’.”
- Understanding types can be separated from understanding data formats

Interfaces

- *Recall:* two problems for source generation. Generate the implementation and invoke the implementation
- Interfaces: connect code to input
- Types and parsers: one set of interfaces (/traits/protocols) for understanding types, one for understanding data formats
- Data formats can be prewritten, type logic must be generated

	Swift	Rust	Kotlin
Interface	Encodable/Decodable	Serde Serialize/Deserialize	KotlinX SerializationStrategy<T>, DeserializationStrategy<T>
Codegen Strategy	Compiler-generated	Macros	Compiler plugin (source generator)

Problem: Generics

- List<T>
  - Serializable only if T is serializable
  - Need serialization implementation for T
  - Kotlin: construct separate serializers, pass in requirements manually

```
fun main() {
   val stringListSerializer: KSerializer<List<String>> =
             ListSerializer(String.serializer())
   println(stringListSerializer.descriptor)
}
```

Problem: Generics

- Swift + Rust: *Conditional implementation*
- Impossible to represent on definition, must be extension
- Extension is itself generic

```
extension<T> SomeType<T>: Codable
   where T : Codable {
   ...
}
```

Swift, Rust, Kotlin: End Result

- *One* source generation solution for *all* serialization problems
- Some limitations in data format support or performance based on choice of interface structure
- Varying ergonomics for different problems
- Complex attributes/configuration make non-source-generated options almost impossible

C# Lessons

- Source generator per library doesn't scale
- Trimming/AOT work with all statically-typed language features out-of-the-box
- Interface-based design is fast (when generic-constrained) and inherently trim-friendly
- Extensions are more powerful than interface definition, *even if you control the type definition*
- Big source generator weaknesses at type "connection" or "discovery"

Q & A