Function Pointers

Article • 03/24/2022 • 22 minutes to read

Summary

This proposal provides language constructs that expose IL opcodes that cannot currently be accessed efficiently, or at all, in C# today: `ldftn` and `calli`. These IL opcodes can be important in high performance code and developers need an efficient way to access them.

Motivation

The motivations and background for this feature are described in the following issue (as is a potential implementation of the feature):

https://github.com/dotnet/csharplang/issues/191

This is an alternate design proposal to compiler intrinsics

Detailed Design

Function pointers

The language will allow for the declaration of function pointers using the `delegate*` syntax. The full syntax is described in detail in the next section but it is meant to resemble the syntax used by `Func` and `Action` type declarations.

```C#
unsafe class Example {
    void Example(Action<int> a, delegate*<int, void> f) {
        a(42);
        f(42);
    }
}
```

These types are represented using the function pointer type as outlined in ECMA-335. This means invocation of a `delegate*` will use `calli` where invocation of a `delegate` will use

`callvirt` on the `Invoke` method. Syntactically though invocation is identical for both callvirt on the Invoke method. Syntactically though invocation is identical for both constructs.

The ECMA-335 definition of method pointers includes the calling convention as part of the type signature (section 7.1). The default calling convention will be `managed`. Unmanaged calling conventions can by specified by putting an `unmanaged` keyword afer the `delegate*` syntax, which will use the runtime platform default. Specific unmanaged conventions can then be specified in brackets to the `unmanaged` keyword by specifying any type starting with `CallConv` in the `System.Runtime.CompilerServices` namespace, leaving off the `CallConv` prefix. These types must come from the program's core library, and the set of valid combinations is platform-dependent.

```C#
//This method has a managed calling convention. This is the same as leaving the managed keyword off.
delegate* managed<int, int>;

// This method will be invoked using whatever the default unmanaged calling convention on the runtime
// platform is. This is platform and architecture dependent and is determined by the CLR at runtime.
delegate* unmanaged<int, int>;

// This method will be invoked using the cdecl calling convention
// Cdecl maps to System.Runtime.CompilerServices.CallConvCdecl
delegate* unmanaged[Cdecl] <int, int>;

// This method will be invoked using the stdcall calling convention, and suppresses GC transition
// Stdcall maps to System.Runtime.CompilerServices.CallConvStdcall
// SuppressGCTransition maps to System.Runtime.CompilerServices.CallConvSuppressGCTransition
delegate* unmanaged[Stdcall, SuppressGCTransition] <int, int>;
```

Conversions between `delegate*` types is done based on their signature including the calling convention.

```C#
unsafe class Example {
    void Conversions() {
        delegate*<int, int, int> p1 = ...;
        delegate* managed<int, int, int> p2 = ...;
        delegate* unmanaged<int, int, int> p3 = ...;
```

```
        p1 = p2; // okay p1 and p2 have compatible signatures
        Console.WriteLine(p2 == p1); // True
        p2 = p3; // error: calling conventions are incompatible
    }
}
```

A `delegate*` type is a pointer type which means it has all of the capabilities and restrictions of a standard pointer type:

- Only valid in an `unsafe` context.
- Methods which contain a `delegate*` parameter or return type can only be called from an `unsafe` context.
- Cannot be converted to `object`.
- Cannot be used as a generic argument.
- Can implicitly convert `delegate*` to `void*`.
- Can explicitly convert from `void*` to `delegate*`.

Restrictions:

- Custom attributes cannot be applied to a `delegate*` or any of its elements.
- A `delegate*` parameter cannot be marked as `params`
- A `delegate*` type has all of the restrictions of a normal pointer type.
- Pointer arithmetic cannot be performed directly on function pointer types.

Function pointer syntax

The full function pointer syntax is represented by the following grammar:

```antlr
pointer_type
    : ...
    | funcptr_type
    ;

funcptr_type
    : 'delegate' '*' calling_convention_specifier? '<' funcptr_parameter_list
funcptr_return_type '>'
    ;

calling_convention_specifier
    : 'managed'
    | 'unmanaged' ('[' unmanaged_calling_convention ']')?
```

```
    | 'unmanaged' ('[' unmanaged_calling_convention ']')?
    ;

unmanaged_calling_convention
    : 'Cdecl'
    | 'Stdcall'
    | 'Thiscall'
    | 'Fastcall'
    | identifier (',' identifier)*
    ;

funptr_parameter_list
    : (funcptr_parameter ',')*
    ;

funcptr_parameter
    : funcptr_parameter_modifier? type
    ;

funcptr_return_type
    : funcptr_return_modifier? return_type
    ;

funcptr_parameter_modifier
    : 'ref'
    | 'out'
    | 'in'
    ;

funcptr_return_modifier
    : 'ref'
    | 'ref readonly'
    ;
```

If no `calling_convention_specifier` is provided, the default is `managed`. The precise metadata encoding of the `calling_convention_specifier` and what `identifier`s are valid in the `unmanaged_calling_convention` is covered in Metadata Representation of Calling Conventions.

C#

```csharp
delegate int Func1(string s);
delegate Func1 Func2(Func1 f);

// Function pointer equivalent without calling convention
delegate*<string, int>;
delegate*<delegate*<string, int>, delegate*<string, int>>;


// Function pointer equivalent with calling convention
```

```
// Function pointer equivalent with calling convention
delegate* managed<string, int>;
delegate*<delegate* managed<string, int>, delegate*<string, int>>;
```

Function pointer conversions

In an unsafe context, the set of available implicit conversions (Implicit conversions) is extended to include the following implicit pointer conversions:

- *Existing conversions* - (§22.5)
- From *funcptr_type* `F0` to another *funcptr_type* `F1`, provided all of the following are true:
  - `F0` and `F1` have the same number of parameters, and each parameter `D0n` in `F0` has the same `ref`, `out`, or `in` modifiers as the corresponding parameter `D1n` in `F1`.
  - For each value parameter (a parameter with no `ref`, `out`, or `in` modifier), an identity conversion, implicit reference conversion, or implicit pointer conversion exists from the parameter type in `F0` to the corresponding parameter type in `F1`.
  - For each `ref`, `out`, or `in` parameter, the parameter type in `F0` is the same as the corresponding parameter type in `F1`.
  - If the return type is by value (no `ref` or `ref readonly`), an identity, implicit reference, or implicit pointer conversion exists from the return type of `F1` to the return type of `F0`.
  - If the return type is by reference (`ref` or `ref readonly`), the return type and `ref` modifiers of `F1` are the same as the return type and `ref` modifiers of `F0`.
  - The calling convention of `F0` is the same as the calling convention of `F1`.

Allow address-of to target methods

Method groups will now be allowed as arguments to an address-of expression. The type of such an expression will be a `delegate*` which has the equivalent signature of the target method and a managed calling convention:

C#

```
unsafe class Util {
    public static void Log() { }


    void Use() {
```

```
    void Use() {
        delegate*<void> ptr1 = &Util.Log;

        // Error: type "delegate*<void>" not compatible with "delegate*<int>";
        delegate*<int> ptr2 = &Util.Log;
    }
}
```

In an unsafe context, a method `M` is compatible with a function pointer type `F` if all of the following are true:

- `M` and `F` have the same number of parameters, and each parameter in `M` has the same `ref`, `out`, or `in` modifiers as the corresponding parameter in `F`.
- For each value parameter (a parameter with no `ref`, `out`, or `in` modifier), an identity conversion, implicit reference conversion, or implicit pointer conversion exists from the parameter type in `M` to the corresponding parameter type in `F`.
- For each `ref`, `out`, or `in` parameter, the parameter type in `M` is the same as the corresponding parameter type in `F`.
- If the return type is by value (no `ref` or `ref readonly`), an identity, implicit reference, or implicit pointer conversion exists from the return type of `F` to the return type of `M`.
- If the return type is by reference (`ref` or `ref readonly`), the return type and `ref` modifiers of `F` are the same as the return type and `ref` modifiers of `M`.
- The calling convention of `M` is the same as the calling convention of `F`. This includes both the calling convention bit, as well as any calling convention flags specified in the unmanaged identifier.
- `M` is a static method.

In an unsafe context, an implicit conversion exists from an address-of expression whose target is a method group `E` to a compatible function pointer type `F` if `E` contains at least one method that is applicable in its normal form to an argument list constructed by use of the parameter types and modifiers of `F`, as described in the following.

- A single method `M` is selected corresponding to a method invocation of the form `E(A)` with the following modifications:
  - The arguments list `A` is a list of expressions, each classified as a variable and with the type and modifier (`ref`, `out`, or `in`) of the corresponding *funcptr_parameter_list* of `F`.
  - The candidate methods are only those methods that are applicable in their normal form, not those applicable in their expanded form.
  - The candidate methods are only those methods that are static.

- The candidate methods are only those methods that are static.
- If the algorithm of overload resolution produces an error, then a compile-time error occurs. Otherwise, the algorithm produces a single best method `M` having the same number of parameters as `F` and the conversion is considered to exist.
- The selected method `M` must be compatible (as defined above) with the function pointer type `F`. Otherwise, a compile-time error occurs.
- The result of the conversion is a function pointer of type `F`.

This means developers can depend on overload resolution rules to work in conjunction with the address-of operator:

C#

```
unsafe class Util {
    public static void Log() { }
    public static void Log(string p1) { }
    public static void Log(int i) { };

    void Use() {
        delegate*<void> a1 = &Log; // Log()
        delegate*<int, void> a2 = &Log; // Log(int i)

        // Error: ambiguous conversion from method group Log to "void*"
        void* v = &Log;
    }
```

The address-of operator will be implemented using the `ldftn` instruction.

Restrictions of this feature:

- Only applies to methods marked as `static`.
- Non-`static` local functions cannot be used in `&`. The implementation details of these methods are deliberately not specified by the language. This includes whether they are static vs. instance or exactly what signature they are emitted with.

Operators on Function Pointer Types

The section in unsafe code on operators is modified as such:

> In an unsafe context, several constructs are available for operating on all _pointer_type_s that are not _funcptr_type_s:
>
> - The `*` operator may be used to perform pointer indirection (§23.6.2...)

> - The `*` operator may be used to perform pointer indirection (§22.6.2).
> - The `->` operator may be used to access a member of a struct through a pointer (§22.6.3).
> - The `[]` operator may be used to index a pointer (§22.6.4).
> - The `&` operator may be used to obtain the address of a variable (§22.6.5).
> - The `++` and `--` operators may be used to increment and decrement pointers (§22.6.6).
> - The `+` and `-` operators may be used to perform pointer arithmetic (§22.6.7).
> - The `==`, `!=`, `<`, `>`, `<=`, and `=>` operators may be used to compare pointers (§22.6.8).
> - The `stackalloc` operator may be used to allocate memory from the call stack (§22.8).
> - The `fixed` statement may be used to temporarily fix a variable so its address can be obtained (§22.7).
>
> In an unsafe context, several constructs are available for operating on all _funcptr_type_s:
>
> - The `&` operator may be used to obtain the address of static methods (Allow address-of to target methods)
> - The `==`, `!=`, `<`, `>`, `<=`, and `=>` operators may be used to compare pointers (§22.6.8).

Additionally, we modify all the sections in `Pointers in expressions` to forbid function pointer types, except `Pointer comparison` and `The sizeof operator`.

Better function member

The better function member specification will be changed to include the following line:

> A `delegate*` is more specific than `void*`

This means that it is possible to overload on `void*` and a `delegate*` and still sensibly use the address-of operator.

Type Inference

In unsafe code, the following changes are made to the type inference algorithms:

In unsafe code, the following changes are made to the type inference algorithms:

Input types

§11.6.3.4

The following is added:

> If `E` is an address-of method group and `T` is a function pointer type then all the parameter types of `T` are input types of `E` with type `T`.

Output types

§11.6.3.5

The following is added:

> If `E` is an address-of method group and `T` is a function pointer type then the return type of `T` is an output type of `E` with type `T`.

Output type inferences

§11.6.3.7

The following bullet is added between bullets 2 and 3:

> - If `E` is an address-of method group and `T` is a function pointer type with parameter types `T1...Tk` and return type `Tb`, and overload resolution of `E` with the types `T1..Tk` yields a single method with return type `U`, then a *lower-bound inference* is made from `U` to `Tb`.

Better conversion from expression

§11.6.4.4

The following sub-bullet is added as a case to bullet 2:

> - `V` is a function pointer type `delegate*<V2..Vk, V1>` and `U` is a function pointer type `delegate*<U2..Uk, U1>`, and the calling convention of `V` is identical to `U`,
>
> and the refness of `Vi` is identical to `Ui`.

> and the refness of `V1` is identical to `U1`.

Lower-bound inferences

§11.6.3.10

The following case is added to bullet 3:

> - `V` is a function pointer type `delegate*<V2..Vk, V1>` and there is a function pointer type `delegate*<U2..Uk, U1>` such that `U` is identical to `delegate*<U2..Uk, U1>`, and the calling convention of `V` is identical to `U`, and the refness of `Vi` is identical to `Ui`.

The first bullet of inference from `Ui` to `Vi` is modified to:

> - If `U` is not a function pointer type and `Ui` is not known to be a reference type, or if `U` is a function pointer type and `Ui` is not known to be a function pointer type or a reference type, then an *exact inference* is made

Then, added after the 3rd bullet of inference from `Ui` to `Vi`:

> - Otherwise, if `V` is `delegate*<V2..Vk, V1>` then inference depends on the i-th parameter of `delegate*<V2..Vk, V1>`:
>   - If V1:
>   - If the return is by value, then a *lower-bound inference* is made.
>   - If the return is by reference, then an *exact inference* is made.
>   - If V2..Vk:
>   - If the parameter is by value, then an *upper-bound inference* is made.
>   - If the parameter is by reference, then an *exact inference* is made.

Upper-bound inferences

§11.6.3.11

The following case is added to bullet 2:

> - `U` is a function pointer type `delegate*<U2..Uk, U1>` and `V` is a function pointer type which is identical to `delegate*<V2..Vk, V1>`, and the calling convention of `U` is identical to `V`, and the refness of `Ui` is identical to `Vi`.

> is identical to `V`, and the refness of `Ui` is identical to `Vi`.

The first bullet of inference from `Ui` to `Vi` is modified to:

> - If `U` is not a function pointer type and `Ui` is not known to be a reference type, or if `U` is a function pointer type and `Ui` is not known to be a function pointer type or a reference type, then an *exact inference* is made

Then added after the 3rd bullet of inference from `Ui` to `Vi`:

> - Otherwise, if `U` is `delegate*<U2..Uk, U1>` then inference depends on the i-th parameter of `delegate*<U2..Uk, U1>`:
>   - If U1:
>   - If the return is by value, then an *upper-bound inference* is made.
>   - If the return is by reference, then an *exact inference* is made.
>   - If U2..Uk:
>   - If the parameter is by value, then a *lower-bound inference* is made.
>   - If the parameter is by reference, then an *exact inference* is made.

Metadata representation of `in`, `out`, and `ref readonly` parameters and return types

Function pointer signatures have no parameter flags location, so we must encode whether parameters and the return type are `in`, `out`, or `ref readonly` by using modreqs.

`in`

We reuse `System.Runtime.InteropServices.InAttribute`, applied as a `modreq` to the ref specifier on a parameter or return type, to mean the following:

- If applied to a parameter ref specifier, this parameter is treated as `in`.
- If applied to the return type ref specifier, the return type is treated as `ref readonly`.

`out`

We use `System.Runtime.InteropServices.OutAttribute`, applied as a `modreq` to the ref

We use `System.Runtime.InteropServices.OutAttribute`, applied as a `modreq` to the ref specifier on a parameter type, to mean that the parameter is an `out` parameter.

Errors

- It is an error to apply `OutAttribute` as a modreq to a return type.
- It is an error to apply both `InAttribute` and `OutAttribute` as a modreq to a parameter type.
- If either are specified via modopt, they are ignored.

Metadata Representation of Calling Conventions

Calling conventions are encoded in a method signature in metadata by a combination of the `CallKind` flag in the signature and zero or more `modopt`s at the start of the signature. ECMA-335 currently declares the following elements in the `CallKind` flag:

antlr

```
CallKind
    : default
    | unmanaged cdecl
    | unmanaged fastcall
    | unmanaged thiscall
    | unmanaged stdcall
    | varargs
    ;
```

Of these, function pointers in C# will support all but `varargs`.

In addition, the runtime (and eventually 335) will be updated to include a new `CallKind` on new platforms. This does not have a formal name currently, but this document will use `unmanaged ext` as a placeholder to stand for the new extensible calling convention format. With no `modopt`s, `unmanaged ext` is the platform default calling convention, `unmanaged` without the square brackets.

Mapping the `calling_convention_specifier` to a `CallKind`

A `calling_convention_specifier` that is omitted, or specified as `managed`, maps to the `default` `CallKind`. This is default `CallKind` of any method not attributed with

`UnmanagedCallersOnly`.

C# recognizes 4 special identifiers that map to specific existing unmanaged `CallKind`s from ECMA 335. In order for this mapping to occur, these identifiers must be specified on their own, with no other identifiers, and this requirement is encoded into the spec for `unmanaged_calling_convention`s. These identifiers are `Cdecl`, `Thiscall`, `Stdcall`, and `Fastcall`, which correspond to `unmanaged cdecl`, `unmanaged thiscall`, `unmanaged stdcall`, and `unmanaged fastcall`, respectively. If more than one `identifer` is specified, or the single `identifier` is not of the specially recognized identifiers, we perform special name lookup on the identifier with the following rules:

- We prepend the `identifier` with the string `CallConv`
- We look only at types defined in the `System.Runtime.CompilerServices` namespace.
- We look only at types defined in the core library of the application, which is the library that defines `System.Object` and has no dependencies.
- We look only at public types.

If lookup succeeds on all of the `identifier`s specified in an `unmanaged_calling_convention`, we encode the `CallKind` as `unmanaged ext`, and encode each of the resolved types in the set of `modopt`s at the beginning of the function pointer signature. As a note, these rules mean that users cannot prefix these `identifier`s with `CallConv`, as that will result in looking up `CallConvCallConvVectorCall`.

When interpreting metadata, we first look at the `CallKind`. If it is anything other than `unmanaged ext`, we ignore all `modopt`s on the return type for the purposes of determining the calling convention, and use only the `CallKind`. If the `CallKind` is `unmanaged ext`, we look at the modopts at the start of the function pointer type, taking the union of all types that meet the following requirements:

- The is defined in the core library, which is the library that references no other libraries and defines `System.Object`.
- The type is defined in the `System.Runtime.CompilerServices` namespace.
- The type starts with the prefix `CallConv`.
- The type is public.

These represent the types that must be found when performing lookup on the `identifier`s in an `unmanaged_calling_convention` when defining a function pointer type in source.

It is an error to attempt to use a function pointer with a `CallKind` of `unmanaged ext` if the

It is an error to attempt to use a function pointer with a `CallKind` of `unmanaged ext` if the target runtime does not support the feature. This will be determined by looking for the presence of the `System.Runtime.CompilerServices.RuntimeFeature.UnmanagedCallKind` constant. If this constant is present, the runtime is considered to support the feature.

`System.Runtime.InteropServices.UnmanagedCallersOnlyAttribute`

`System.Runtime.InteropServices.UnmanagedCallersOnlyAttribute` is an attribute used by the CLR to indicate that a method should be called with a specific calling convention. Because of this, we introduce the following support for working with the attribute:

- It is an error to directly call a method annotated with this attribute from C#. Users must obtain a function pointer to the method and then invoke that pointer.
- It is an error to apply the attribute to anything other than an ordinary static method or ordinary static local function. The C# compiler will mark any non-static or static non-ordinary methods imported from metadata with this attribute as unsupported by the language.
- It is an error for a method marked with the attribute to have a parameter or return type that is not an `unmanaged_type`.
- It is an error for a method marked with the attribute to have type parameters, even if those type parameters are constrained to `unmanaged`.
- It is an error for a method in a generic type to be marked with the attribute.
- It is an error to convert a method marked with the attribute to a delegate type.
- It is an error to specify any types for `UnmanagedCallersOnly.CallConvs` that do not meet the requirements for calling convention `modopt`s in metadata.

When determining the calling convention of a method marked with a valid `UnmanagedCallersOnly` attribute, the compiler performs the following checks on the types specified in the `CallConvs` property to determine the effective `CallKind` and `modopt`s that should be used to determine the calling convention:

- If no types are specified, the `CallKind` is treated as `unmanaged ext`, with no calling convention `modopt`s at the start of the function pointer type.
- If there is one type specified, and that type is named `CallConvCdecl`, `CallConvThiscall`, `CallConvStdcall`, or `CallConvFastcall`, the `CallKind` is treated as `unmanaged cdecl`, `unmanaged thiscall`, `unmanaged stdcall`, or `unmanaged fastcall`, respectively, with no calling convention `modopt`s at the start of the function pointer

respectively, with no calling convention `modopt`s at the start of the function pointer type.
- If multiple types are specified or the single type is not named one of the specially called out types above, the `CallKind` is treated as `unmanaged ext`, with the union of the types specified treated as `modopt`s at the start of the function pointer type.

The compiler then looks at this effective `CallKind` and `modopt` collection and uses normal metadata rules to determine the final calling convention of the function pointer type.

Open Questions

Detecting runtime support for `unmanaged ext`

https://github.com/dotnet/runtime/issues/38135 tracks adding this flag. Depending on the feedback from review, we will either use the property specified in the issue, or use the presence of `UnmanagedCallersOnlyAttribute` as the flag that determines whether the runtimes supports `unmanaged ext`.

Considerations

Allow instance methods

The proposal could be extended to support instance methods by taking advantage of the `EXPLICITTHIS` CLI calling convention (named `instance` in C# code). This form of CLI function pointers puts the `this` parameter as an explicit first parameter of the function pointer syntax.

C#

```csharp
unsafe class Instance {
    void Use() {
        delegate* instance<Instance, string> f = &ToString;
        f(this);
    }
}
```

This is sound but adds some complication to the proposal. Particularly because function pointers which differed by the calling convention `instance` and `managed` would be incompatible even though both cases are used to invoke managed methods with the same incompatible even though both cases are used to invoke managed methods with the same C# signature. Also in every case considered where this would be valuable to have there was a simple work around: use a `static` local function.

```C#
unsafe class Instance {
    void Use() {
        static string toString(Instance i) => i.ToString();
        delegate*<Instance, string> f = &toString;
        f(this);
    }
}
```

Don't require unsafe at declaration

Instead of requiring `unsafe` at every use of a `delegate*`, only require it at the point where a method group is converted to a `delegate*`. This is where the core safety issues come into play (knowing that the containing assembly cannot be unloaded while the value is alive). Requiring `unsafe` on the other locations can be seen as excessive.

This is how the design was originally intended. But the resulting language rules felt very awkward. It's impossible to hide the fact that this is a pointer value and it kept peeking through even without the `unsafe` keyword. For example the conversion to `object` can't be allowed, it can't be a member of a `class`, etc ... The C# design is to require `unsafe` for all pointer uses and hence this design follows that.

Developers will still be capable of presenting a *safe* wrapper on top of `delegate*` values the same way that they do for normal pointer types today. Consider:

```C#
unsafe struct Action {
    delegate*<void> _ptr;

    Action(delegate*<void> ptr) => _ptr = ptr;
    public void Invoke() => _ptr();
}
```

Using delegates

Instead of using a new syntax element, `delegate*`, simply use existing `delegate` types with

Instead of using a new syntax element, `delegate*`, simply use existing `delegate` types with a `*` following the type:

C#

```
Func<object, object, bool>* ptr = &object.ReferenceEquals;
```

Handling calling convention can be done by annotating the `delegate` types with an attribute that specifies a `CallingConvention` value. The lack of an attribute would signify the managed calling convention.

Encoding this in IL is problematic. The underlying value needs to be represented as a pointer yet it also must:

1. Have a unique type to allow for overloads with different function pointer types.
2. Be equivalent for OHI purposes across assembly boundaries.

The last point is particularly problematic. This mean that every assembly which uses `Func<int>*` must encode an equivalent type in metadata even though `Func<int>*` is defined in an assembly though don't control. Additionally any other type which is defined with the name `System.Func<T>` in an assembly that is not mscorlib must be different than the version defined in mscorlib.

One option that was explored was emitting such a pointer as `mod_req(Func<int>) void*`. This doesn't work though as a `mod_req` cannot bind to a `TypeSpec` and hence cannot target generic instantiations.

Named function pointers

The function pointer syntax can be cumbersome, particularly in complex cases like nested function pointers. Rather than have developers type out the signature every time the language could allow for named declarations of function pointers as is done with `delegate`.

C#

```
func* void Action();

unsafe class NamedExample {
    void M(Action a) {
        a();


    }
```

```
}
```

Part of the problem here is the underlying CLI primitive doesn't have names hence this would be purely a C# invention and require a bit of metadata work to enable. That is doable but is a significant about of work. It essentially requires C# to have a companion to the type def table purely for these names.

Also when the arguments for named function pointers were examined we found they could apply equally well to a number of other scenarios. For example it would be just as convenient to declare named tuples to reduce the need to type out the full signature in all cases.

```C#
(int x, int y) Point;

class NamedTupleExample {
    void M(Point p) {
        Console.WriteLine(p.x);
    }
}
```

After discussion we decided to not allow named declaration of `delegate*` types. If we find there is significant need for this based on customer usage feedback then we will investigate a naming solution that works for function pointers, tuples, generics, etc ... This is likely to be similar in form to other suggestions like full `typedef` support in the language.

Future Considerations

static delegates

This refers to the proposal to allow for the declaration of `delegate` types which can only refer to `static` members. The advantage being that such `delegate` instances can be allocation free and better in performance sensitive scenarios.

If the function pointer feature is implemented the `static delegate` proposal will likely be closed out. The proposed advantage of that feature is the allocation free nature. However recent investigations have found that is not possible to achieve due to assembly unloading.

There must be a strong handle from the `static delegate` to the method it refers to in order to keep the assembly from being unloaded out from under it.

To maintain every `static delegate` instance would be required to allocate a new handle which runs counter to the goals of the proposal. There were some designs where the allocation could be amortized to a single allocation per call-site but that was a bit complex and didn't seem worth the trade off.

That means developers essentially have to decide between the following trade offs:

1. Safety in the face of assembly unloading: this requires allocations and hence `delegate` is already a sufficient option.
2. No safety in face of assembly unloading: use a `delegate*`. This can be wrapped in a `struct` to allow usage outside an `unsafe` context in the rest of the code.