Expression Trees v2 Spec

Bill Chiles

This document specifies ETs v2 for CLR 4.0.

PRINT WARNING: Doc is almost 200 pages, but conceptual introduction is about 15 pages.

1 Introduction

In the .NET Framework 3.5 we created Expression Trees (ETs) to model code for LINQ expressions in C# and VB. They were limited in .NET 3.5 to focus on LINQ requirements; for example, a LambdaExpression could not contain control flow, only a simple expression as its body. Looking forward, there are several reasons we'd like to extend ETs:

- We'd like to further develop a single semantic code model as a common currency for compilation tools.
- We're adding the Dynamic Language Runtime (DLR) into the .NET Framework, and it uses semantic trees to represent code as a means for making it easier to port new dynamic languages to .NET.
- The DLR also uses semantic trees to represent how to perform abstract operations in its dynamic call site caching mechanism.
- We want to support meta-programming going forward where customers can more readily get programs as data, manipulate the data, and emit new code as data based on the input code.

Obviously, to support the above goals, we need more in the ET model than v1 provided. We need to model control flow, assignment, recursion, etc., in addition to simple expressions. There is no plan in .NET 4.0 to add modeling for types/declarations, but we'll consider these for V-next+1 (that is, the next major release after .NET 4.0).

Some quick terminology:

- The term "ET" indicates a tree structure of instances of Expression (direct or indirect).
- The term "ET node" indicates a single instance of Expression (direct or indirect). The ET node could be the root of a tree.
- The term "ET node type" means the specific class of which the ET node is an instance.
- The term "ET kind" refers to the Expression.NodeType property or indicates the value of the ExpressionType enum. This is a legacy naming issue.
- The term "sub ET" indicates the root of a tree where the root is an interior or leaf node of some other ET.

1.1 Relation to Other Models

We do not think we're creating Yet Another Code Model from Microsoft. We are extending LINQ ET v1. We understand that there is an issue with multiple code models already, given codeDOM and VS Code Model.

ETs have a distinct mission from Design Time Language Models (DTLMs). An example of a DTLM is the VS Code Model or the model used internally within VS for smart editor features. If you think about what they need to support, you can see they have very different requirements:

- DTLMs are necessarily highly language-specific at certain levels. We can build a good story for future DTLMs and ETs v2 interoperating smoothly (see Reducible Nodes in section 2.2).
- DTLMs must explicitly represent errors (that is, contain error nodes with syntax trees).

- DTLMs need ancillary information such as cached parser state, special nodes outside the semantically common ET nodes for representing partial knowledge of code, annotations on common ET nodes, links to editor text and view models, links to project system models, etc.
- DTLMs have many more unbound trees to represent what is clearly an identifier from syntax but still a name that cannot be resolved statically to a concrete reference.
- DTLMs have bound nodes referring to design time models or derivations of System.Type and MemberInfo for types partially defined in editor buffers.
- DTLMs need structural or syntactic fidelity to code, including line break info, whitespace info, etc.

1.2 Review from LINQ Expression Trees v1

ETs are a set of public classes that live in the System.Linq.Expressions namespace.

The Expression Tree API defines a set of classes which representation a number of node types. Combining nodes forms expression trees. The top level node of an expression tree represents an expression, and children trees represent sub expressions.

Each Expression has a NodeType property with a value from the ExpressionType enum. Each kind of ET node kind (not ET node *type*) is uniquely identified by an ExpressionType value in the node's NodeType property. For example, a BinaryExpression node type could have NodeType values indicating different kinds of binary expressions, Add vs. Multiply.

Expression trees represent semantics in that they represent specific language level expressions. However, expression trees are not tied to any specific language and constitute a language-independent representation of common expressions found in CLR languages. ETs are lossy in the sense that they do not exactly model the syntax of a language or how an expression was formed in a language.

1.3 Design Goals

There are three groups of design goals: compatibility, abstraction level, and some explicit non-goals worth mentioning.

1.3.1 Compatibility with V1

The underlying design goal is to stay true to the design of LINQ expression trees v1. We can add new node types and add members to exiting node types. We cannot break existing invariants, such as when properties may be null or what a particular value means.

Existing consumers should continue to work on currently supported ET subsets. There will be no new semantics for current combinations of properties. If an ET v2 node looks like a node your program recognized before, then it can continue to process it in the same way.

Existing consumers who coded defensively for unknown nodes will fall into the same "don't care or don't grok" branches of their code. If they encounter an ET v2 node type or an extended ET v1 type with new combinations of values for properties, they will naturally fall into their defensive code handling these unknown nodes.

ETs v2 keep the immutable property they had in v1.

ETs v2 stay tree-structured with the exception of the ParameterExpression nodes.

ETs v2 keep the "shape-based" design of ETs v1. For example, whenever a node would have the same count of children, we tried to unify the nodes. This is why BinaryExpression has so many node kinds, and furthermore, why assignment is modeled as a BinaryExpression. You can see this too with Throw and Unbox modeled as UnaryExpression. The node kind distinction helps the meta-programmer see the exact semantics of the node.

1.3.2 Model Abstraction Level

Going back to our motivation for ETs v2, we want to achieve a generally useful semantic code model that can serve as a common currency in compilation tools. We would also like to support end-user meta-programming where they can get programs as data, manipulate them, and return new programs. We clearly do not want a purely syntactic model that is necessarily very language-specific and not representative of semantics. We also do not want an MSIL model, which would be so far removed from source programs that meta-programming would be unbearable.

The ET v2 model should provide easy consumption so that they are able to preserve language-level concepts for meta-programming. For example, rather than simply have a bare LoopExpression with GotoExpressions, we need common iteration constructs modeled closer to languages. We should have ForExpression, ForEachExpression, WhileExpression, RepeatUntilExpression, etc., in addition to a fundamental LoopExpression.

The model should provide easy production support for new code. Tools will rewrite ETs. Given the number of programmers and their backgrounds who write these tools, they will not be comfortable with lower-level node types. They will likely produce bushier or expanded trees as more direct input to compilers. The end-user meta-programmers however will be most comfortable producing an ET closer to the languages they are familiar with. They will want higher-level ET modeling.

These constraints lead us to avoid nodes that are specific to a single language and that are specific to a single consumer. For example, we include MethodInfo or CallSiteBinder in nodes to specifically capture their intended semantics where they might be interpreted differently by different consumers. ~~We also include an annotations mechanism so that consumers can tailor common nodes to themselves when appropriate~~.

We resolve the tension of abstraction levels in two ways. One is to have some higher-level node types we find common across multiple languages, such as the iteration models mentioned above. The other is by providing an ET node reducing mechanism. This allows language-specific derived types from Expression to reduce to ETs that are more explicit in their semantics, comprised purely of common ET v2 nodes, and commonly understood by all consumers.

1.3.3 .NET 4.0 vs. V-next+1

We can only add so much to ETs in .NET 4.0 due to resources and time line. We know we need to add everything needed to support IronPython, IronRuby, DLR JScript, and ETs generated for dynamic CallSites (see sites-binders-dynobj-interop.doc spec). In some places we can avoid adding new node types to the common ET nodes because languages can add extension nodes

that reduce to a tree expressed in common nodes. We also aren't trying to add any modeling of declarations for types in .NET 4.0.

What we decide to add in .NET 4.0 is driven by needs of current languages so that they are not doing undue amounts of work to express certain language features. For example, we're adding an explicit scope expression (which we model with a BlockExpression that has an explicit list of variables). Languages do not have to jump through hoops to create scopes with runtime helpers, using LambdaExpression in weird ways to avoid forcing unneeded closures, or other techniques. Another example is adding a GlobalExpression node, which has the semantics of getting a variable value from a host-supplied scope. Hosted languages (IPy and IRuby) can work around not having a GlobalExpression, but in V-next+1 we'll want a general model of hosted execution as well as "natural" execution like C# and VB on .NET.

Sometimes we've added node types because we know in V-next+1 we'll try to achieve full coverage of IL semantics (not just CLS semantics). We aren't adding everything now of course due to resources and time. However, we will add some node types now that we might otherwise avoid in .NET 4.0 because we know we'll need them eventually. For example, we're adding the UnboxExpression in .NET 4.0 to model the IL of passing an IntPtr to a boxed value type's value. IronPython could have worked around this in .NET 4.0, but adding this now will make their programming model for .NET interop cleaner sooner.

1.3.4 Non-goal: Design Time Language Models

ETs have a mission that is distinct from Design Time Language Models (for example, VS Code Model). See section 1.1 for several reasons or ways in which these two kinds of models are the same model.

We can provide a smooth transition from DTLMs to ETs as well as some interoperability. Using the reducible node mechanism in ETs v2, the DTLM types could all derive from Expression. They could have all the special properties and hook they need into the design time environment of a tool (project models, text models in editors, etc.). However, when a DTLM tree was complete, or error free, those node types could reduce to common ET v2 nodes representing a correct program's semantics. The reduced tree could have a very similar shape to the DTLM tree.

2 Highlighted Concepts

There are several high-level ideas or characteristics about the ET v2 design that are worth calling out. Think of these as high-order bit calls to make or top ten questions language implementers will be thinking as they approach a semantic model of code.

2.1 Expression-based Model

One high-order bit to language design is whether to be expression-based. Do you have distinct notions of statements or control flow, or do you have a common concept of evaluating expressions where everything has a value. We decided to stay expression-based, so statements are modeled as having a result value and type.

There are several reasons for this design:

- Expression remains the base type for all ET nodes, and we avoid dual type hierarchies.

- Void is already allowed as a type, indicating there is no return value for an expression.
- Lambdas don't change at all from v1 to v2.
- Being expression-based matches many languages (Lisp, Scheme, Ruby, F#), and it does no harm when modeling other languages. They can easily make expressions be void returning.

Let's look at a couple of examples:

- BlockExpression has a value. By default its value is the last expression in the sequence, and its type is the same type as the last expression. This design also allows us to avoid another CommaExpression since BlockExpression now models this semantics as well.
- We model 'if' with ConditionalExpression, which returns the value of its consequent or alternative expression, whichever executes. The types of the branches must match the type of the ConditionalExpression. If there's no alternative expression, then we use a DefaultExpression with the matching type. Languages with distinct notions for statements often have a 'e1 ? e2 : e3' expression since their 'if' cannot return values, but they can model this with ConditionalExpression.

DefaultExpression serves two useful purposes in our expression-based model. First the Expression.Empty factory returns a DefaultExpression with Type void. This can be useful if you need an expression in a value-resulting position that matches the containing expressions result type. The second use of DefaultExpression is when you do have a non-void Expression in which you do need to sometimes return the "`default(T)`" value. Without this expression, you would have to generate a lot more ET to express "`default(T)`".

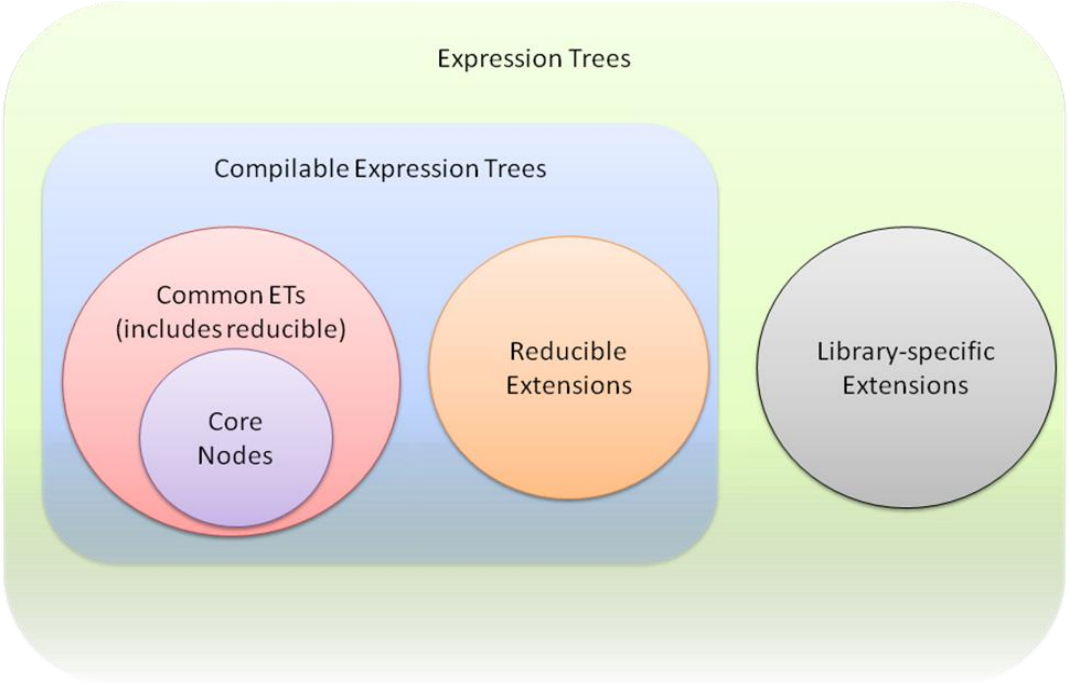
Often you do not need to use Expression.Empty to match a containing node's result type. There are expressions used in common patterns, typically control flow expression, where the result value is not used. For these common patterns, some nodes implicitly convert to void or squelch a result value. SwitchExpression, ConditionalExpression, TryExpression, BlockExpression, LambdaExpression, GotoExpression, and LabelExpression all automatically convert their result expression to void if they themselves have a void Type property (or delegate with result type in the case of lambdas).

2.2 Reducible Nodes

We have a tension for what level of abstraction to provide in our model (see section 1.3.2). While we think Design Time Language Models have a distinct mission from Expression Trees v2, we would like to allow for smooth interoperability between them. We enable higher-level models (even language-specific models) that can reduce to a common set of ET v2 node types that all consumers can process. Programs can query an ET node as to whether it reduces, and if so, a program can call on the node to reduce itself. When a node reduces, it returns a semantically equivalent ET with a root node than can replace the original ET node.

Reductions are allowed to be partial. The resulting ET may include nodes that need to be further reduced. Typically the immediate result of reducing one node comprises only children that are common ET v2 node types. Expression.Compile only compiles common node types and those that reduce to common nodes. If a node type does not reduce or does not reduce to only common nodes, then it may still be useful as part of a library-specific set of extensions that model code. An example might be a Design Time Language Model for code that either has too

many errors to reduce or that is part of a model too specific to tooling needs to bother reducing it to common nodes.



The ET v2 common set of nodes will include some reducible nodes. For example, for meta-programming goals, there will be higher-level iteration models. We'll include ForExpression for counter termination, ForEachExpression for member iterations, WhileExpression for test first iterations, and RepeatUntilExpression for test last iterations. We'll also include a fundamental LoopExpression to which the other iteration models reduce. Other examples include BinaryExpression with node kind AddAssign (compound assignment) or UnaryExpression with node kinds PreIncrementAssign and PostIncrementAssign.

Note, due to time constraints we cut the higher-level iteration node types for .NET 4.0, but they will likely show up soon in the DLR's codeplex open source project.

Common ET nodes with a given node kind are either reducible always, or never. That is, a node is not conditionally reducible based on other properties it has that may be different for different instantiations. For example, the GeneratorExpression nodes are always reducible. Regardless of the reducibility, the compiler may have direct support for the node kind, or the compiler may reduce the nodes. For example, when we add ForEachExpression, the compiler will likely directly compile it without reducing it.

2.3 Bound, Unbound, and Dynamic Nodes

There are three categories or states of being bound for modeling expressions. More commonly mathematicians or computer scientists think of only two, bound and unbound. For example, in the expression "for all x such that 0 < x + y < 10", 'x' is a bound variable while 'y' is a free reference or unbound variable. If 'y' were not present in the expression, the expression would be fully statically bound such that we could evaluate it. However, to evaluate the expression, we need to first bind 'y' to some value.

An unbound ET node:

- would need to be bound before executing it
- would represent syntax more than semantics
- would have a Type property that is null (see .NET 4.0 vs. V-next+1 note below)

Consider a language that supported LINQ-like expression and that also had late-bound member access (for example, if VB added late-bound LINQ). You would then need to model unbound trees for the lambda expression in the following pseudo-code:

```
o.Where( lambda (x) => x > 0 )   #o had late bound semantics
```

To be able to execute an ET modeling this code, you would need to inspect the runtime type of 'o', search its 'Where' overloads, and pattern match for one that can take a delegate. Furthermore, you would need to match lambda expression to the delegate. The delegate needs take an argument and returns a value with some type. The delegate's type for 'x' needs to make sense to bind '>' to an implementation taking the type of 'x', an operand assignable from integer, and returning the type of the delegate.

A key observation in this situation is that the late-bound node representing the call to 'Where' necessarily has language-specific binding information representing the lambda. The representation cannot be language-neutral semantically. It also can't even be just syntax in any common representation because you need the language that produced the ET to process the lambda representation in the presence of runtime type information while binding. Support for unbound ETs may not be a good solution or one worth trying to share across languages.

NOTE: Since no languages currently support late bound LINQ expressions, we won't actually allow Expression.Type to be null in .NET 4.0. We'll reconsider this in V-next+1 if we think ETs are useful to languages that need to represent unbound trees like the lambda expression in the example above.

In the ET v2 model, a bound ET node:

- has non-null Type property (that is, we statically know its type)
- could be dynamic expression

A dynamic expression often has a Type property that is Object, but its Type that is not null. It might not be Object as well. For example, in "`if x > y`" the ET node for '>' could be typed Boolean even if it is a dynamic node.

The ET v2 model includes dynamically bound nodes that:

- must be resolved at run time to determine how to perform the operation they model
- represented by DynamicExpression nodes

2.3.1 DynamicExpression Node

The DynamicExpression has binding information representing the metadata that further describes the expression beyond the ET node type and any children it has. For example, an ET representing the dynamic expression for comparing a variable to a string literal might have a flag in its binding information indicating whether the compilation unit had an option set to compare strings case-sensitively (which VB has). The binding information also encapsulates the language that created the ET node. The language determines the semantics of the dynamic expression at runtime when its binder searches for an implementation of the operation represented by the node.

One observation about the DynamicExpression node is that in some sense it marks a sub ET as being late-bound. These nodes need more information about what sort of operation they represent, unlike MethodCallExpression. Furthermore, they need to represent most of what all the other ET node types represent. This begs the question of having a dual representation for dynamic vs. fully bound static nodes. We went down the design path for quite a while using only the other node types with an optional BindingInfo property to mark the dynamic expressions.

It turned out in practice that while we thought we had a more economic design, the usage of the model with an optional BindingInfo property was too awkward:

- Hosted language scenarios need more arguments (often in the form of run-time context). For example, BinaryExpression Divide or Equal can require an extra context argument for the specific semantics in a module for Python or VB, meaning the operation does not fit the binary shape.
- A BinaryExpression with node kind Assign and a Left operand that is a MemberExpression is not intuitive as to where the extra dynamic metadata belongs. Does it go on the BinaryExpression or the MemberExpression, or both?
- Due to how we compile dynamic expression to use the DLR's fast dynamic dispatch caching, we need a delegate type to describe the dynamic call site. We don't generate the delegate types, and languages may need to control their creation such as pre-building some of them. This extra data also makes these expressions not fit the fully static nodes' shapes (number of operands or children), but all the dynamic expression do match shape (binding info, delegate type, arguments).
- The programming model for ET consumers always included a check for the BindingInfo Property to handle every node type. Since consumers want to handle dynamic nodes a specific way, using the ExpressionVisitor and other helper code, handling them now is a simple object-oriented method dispatch.

2.3.2 Binding Information in Expression Nodes

Expression nodes can have semantic hints or specifications that are more detailed than, for example, BinaryExpression Add on Int32s or MethodCallExpression with instance and name. These nodes can have MethodInfos attached to them to indicate the exact implementation of Add or resolution of method name to invoke. The MethodInfos serve two main purposes. The first is to be very exact when creating a node what the implementation is for the node's semantics. Language implementations should supply MethodInfos rather than leaving method resolution to the Expression factories because those factories may not resolve overloads in the same way the language would. The second job of the MethodInfos is to provide hints to LINQ providers that might interpret ETs. Those providers can have tables mapping to implementations of the node's semantics when they aren't actually compiling the ET and executing in a .NET run time.

This extra information is required with DynamicExpression nodes. They must have binding information that can be emitted when compiling. The binding information informs the run-time binder how to search for a correct implementation of the node's semantics, given the run-time operands that flow into the operation represented by the ET node. ETs use a CallSiteBinder as the binding information representation, not MethodInfo. In fact, the CallSiteBinder is also the run-time object used in the DLR's CallSites that manage the fast dynamic dispatch for dynamic operations. CallSiteBinder encapsulates both the binding information for the exact semantics of

the node and the language that created the node, which governs the binding of the operation at run time.

Two design issues arise immediately from the choice to use CallSiteBinders vs. MethodInfos. The first is serializability. The ET design supports fully serializable ETs but doesn't enforce that they are always serializable. One reason we use the CallSiteBinders in the DynamicExpression is that they naturally fit exactly what the binding information is that the ET needs and help in hosted execution scenarios. If a language produces an ET as part of a hosting scenario to immediately execute the ET, then the binder can tuck away pointers to live data structures used by the language's engine. Languages can still produce ETs with DynamicExpressions that are serializable if they need to do so.

The second design issue is re-using MethodInfos by deriving custom implementations for use with DynamicExpression nodes. There's a nice consistency in representing the binding information as a MethodInfo, looking on at ETs only. However, the roles played by the MethodInfo are different than the binding information on DynamicExpression. It is more important to have the dynamic binding information be consistent across ETs, the DLR fast dynamic CallSites, and the DLR interoperability MetaObject protocol. Not only does MethodInfo have many members that would throw exceptions if used for dynamic binding information, it would be awkward to require creating an LCG method so that the MethodInfo was invocable. CallSiteBinders are best for detailed semantics capture in DynamicExpression nodes, not MethodInfos.

2.4 Iteration, Goto's, and Exits

Representing iteration has a couple of interesting concepts and design points. How to represent iteration goes to the heart of what abstraction level to model for expressions (see section 1.3.2). Whether to include Goto goes back ages in language design. ETs v2 provides a nice combination of high-level modeling and lower-level support if needed. There are some higher level modeling nodes so that you almost never need Goto, but we provide Goto for reducing some node types to more primitive control flow.

We provide GotoExpression because it is needed for C#, VB, and other languages. The ET v2 GotoExpression started with simple, clean semantics designed into C#. Basically this meant that the target label of the Goto must be lexically within the same function body, and you could only exit inner basic blocks to outer basic blocks. However, we need to accommodate more of VB's older Goto semantics, and we need a richer Goto for some of the ET transformations we do internally (see section 2.4.3).

GotoExpression has an optional Value expression. This allows GotoExpression to enable other node types to truly be expressions. The value expression's type must be ref assignable with the type of the GotoExpression's label target. See the following sub sections for more details.

GotoExpression has a Kind property with a value from GotoExpressionKind (Goto, Break, Continue, Return). This is explained further in the following sub sections. This is convenience for meta-programming purposes.

2.4.1 LoopExpression and Higher-level Iteration Node Types

Given Goto, we provide a basic LoopExpression with explicit label targets for where to break to and where to continue to. Explict labels in LoopExpressions have multiple benefits. You can use

GotoExpression inside the LoopExpression's Body expression and verify the jumps are to the right locations. Explicit labels support languages that can return from or break out of outer loops or scopes, such as JScript or Lisp. Explicit label targets make transformations easier to get right and allow for better error reporting when transformations are not right.

For meta-programming goals, there will be higher-level iteration models. We'll include ForExpression for counter termination, ForExachExpression for member iterations, WhileExpression for test first iterations, and RepeatUntilExpression for test last iterations. These will be common nodes that reduce to combinations of LoopExpression, BlockExpression, GotoExpression, etc.

Due to time constraints we cut the higher-level iteration node types for .NET 4.0, but they will likely show up soon in the DLR's codeplex open source project.

2.4.2 Modeling Exits, LabelExpression, and GotoExpression with a Value

As many have observed, lexical exits are ultimately just a Goto, proceeding to the end of the function and then returning. Sometimes when you leave a function, you leave one or more values on the stack. The ET model only supports a single return value inherently. Once we had to have Goto, there is an economy of design obtained by merging the models of explicit function return and Goto.

There are a couple of very nice benefits from merging lexical exits and Goto. The biggest is that by having Goto optionally carry a value to its target, more of our nodes became truly expression-based. This makes the overall model more consistent by more fully embracing the benefits of being expression-based. For example, LoopExpression is truly an expression that can be used anywhere because you can exit the loop with a value at the break label target.

The second benefit of merging Goto with lexical exits is that we've added explicit label targets to some nodes. This enables more Goto checking in factories and better error messages when compiling. For example, when you have a Goto expression inside a LoopExpression with kind Break, the expression compiler can ensure the Goto's target is the containing Loop's break label target. Also, when you form a GotoExpression, you have to supply the label target which has a Type property. The factory can ensure the Goto and the label target match by type, and the compiler can check the types match the type of the containing expression. Having label targets explicit in the tree also enables tree transformations to be safer and more reliable.

One quirk in the design is how to handle LabelExpression which has a LabelTarget that marks a destination in code for Goto. If a LabelExpression has non-void type, and execution gets to the target via Goto with a value of that type, we're consistent. What happens if I encounter the target location by straight line sequence, and how do we keep the IL value stack consistent? We solve this by adding an optional default value expression to a LabelExpression and by specifying the semantics of LabelExpression to place its target AFTER the default expression. In practice this works very naturally. For example, a LambdaExpression's Body can be a LabelExpression whose default value expression is the actual body of the function. Then the actual body of the function, an expression, is the value left on the stack if you naturally flow through the expressions in the body. If you exit via a GotoExpression (with node kind Return), you have a verifiable target at the end of the function and a verifiable value type that matches the return type from the Type property of the LambdaExpression's.

2.4.3 GotoExpression Capabilities

As stated, we expanded Goto capabilities beyond C#'s. VB is not fully using the DLR yet, but when it does, we will need a more flexible Goto. If we do not allow more cases for GotoExpression, VB would need to produce a VBBlockExpression and their own VBGotoExpression that reduced to a complicated rewriting of the ET. It seems useful to provide the more general GotoExpression. However, VB would still need a special VBBlock to model their on_error_goto semantics, which seems too specific to a single language to generally model in common ET nodes.

ETs v2 limit Goto lexically within a function. ETs allow jumping into and out of the following:

- BlockExpressions
- ConditionalExpressions
- LoopExpressions
- SwitchExpressions
- TryExpressions (under certain situations)
- LabelExpressions

ETs allow some jumps relative to TryExpressions:

- jumping out of TryExpression's body
- jumping out of a CatchBlock's body
- jumping into TryExpression's body from one of its own CatchBlocks

The above constitutes what ETs v2 allow. Just by way of examples, we do not allow jumping into the middle of argument expressions, such as those in binary operands, method calls, invocations, indexing, instance creation, etc. We do not allow jumping into or out of GeneratorExpressions. You could however jump within a BlockExpression used as an argument expression.

2.5 Assignments and L-values

We model assignments with BinaryExpression nodes that have an Assign node kind. The factory methods restrict the Left expression of the BinaryExpression to a fixed set of common ET node types that the compiler recognizes. This permitted ET node types are ParameterExpressions, MemberExpressions, and IndexExpressions.

For ref and out parameters, we also restrict ET node types, but we do allow a few more node types. We additionally allow BinaryExpressions with node kind ArrayIndex, MethodCallExpressions with MethodInfo that is Array.Get, and the new UnboxExpression. The first two are legacy from LINQ ETs v1, and we will obsolete them eventually (see section 2.5.1). Expression.Compile handles write backs for these expressions passed to ref and out parameters. We explicitly do not support property accesses wrapped in conversion expressions due to ambiguities with how to convert back when writing back the out or ref value.

We considered CanWrite and CanRead flags on Expression, but cut them. This would allow assignment factories to check the Left expression. It turns out they weren't that useful. They felt more like form over function since you had to fill them in to call the assignment factory methods without error, but you'd get a nice error anyway from the compiler if you formed your tree incorrectly.

We considered supporting reducible Left expressions in the assignment factory methods, but cut them. The Left expression would have to reduce to the common recognized set of node types. If your assignment needed complicated logic with temps and a block of code, you would have to use a reducible node for the entire BinaryExpression (with Assign node kind). This design felt stilted, so we'll look at this again in v-next+1.

We believe in V-next+1 we can introduce a generalized l-value model. We would add pre and post expressions for temps set up and for write backs. We don't think this model would be overly complex. If added, then l-value positions could be any node type that was writable and had pre and post ETs for setting up temps and writing back values. We would still NOT support l-value positions with property accesses wrapped in conversion expressions.

2.5.1 Indexed Locations

In ETs v2 we have IndexExpression that handles array access, indexers, and indexed properties. You can use these as l-values for assignments and ref/out parameters.

We will obsolete the LINQ v1 support for BinaryExpressions with node kind ArrayIndex and MethodCallExpressions with methodinfo Array.Get. We only support those for ref/out parameters for LINQ v1 compatibility, and we do not support them for the new BinaryExpression with node kind Assign. The plan, which could change, is as follows:

- In .NET 4.0, add bold red text to MSDN docs around ArrayIndex factories. Add bold red text to MethodCallExpression factories if methodinfo has ref/out parameters, and you're using one of the two nodes (BinaryExpressions with node kind ArrayIndex or MethodCallExpressions with methodinfo Array.Get).
- In v-next+1, mark the factory methods as obsolete.
- In v-next+1, issue warnings when compiling ref/out parameters whose expressions are BinaryExpressions with node kind ArrayIndex or MethodCallExpressions with methodinfo Array.Get.
- In v-next+2, remove ArrayIndex factories.
- In v-next+2, throw exceptions where we issues warnings in v-next+1.

We also disallow ref and out arguments for IndexExpression. Neither VB nor C# support these. They are not part of .NET CLS.

When creating IndexExpressions, you must supply PropertyInfo. Another clean up to the LINQ v1 trees is that you cannot use random get/set method pairs.

2.5.2 Compound Assignment or In-place Binary and Unary Operations

We represent operations such as "+=" and "++" as BinaryExpression or UnaryExpression nodes with distinct node kinds (for example, AddAssign). These nodes are reducible common nodes. They reduce to the appropriate binary assignment expressions or blocks with temps to ensure sub-expressions are evaluated at most once. Regarding dynamic expressions at run time, some languages will need to inspect the l-value expression for .NET types to handle events, properties with delegates, etc., the right way when binding the expression.

2.5.3 User-defined Setting Syntax

Languages that have user-defined setting forms for storable locations should provide language-specific reducible nodes. These nodes can reduce, for example, to a MethodCallExpression that takes the arguments defining the location and an argument for the value. We don't provide extensibility for this sort of language feature.

2.5.4 Destructuring Assignment

Languages that support destructuring assignment should provide language-specific reducible nodes. Different languages handle destructuring bind in different ways (first-class carrier objects for the values, single vs. multiple kinds of carrier objects for values, runtime-internal carrier objects or calling conventions, etc.). A language-specific node will allow for better meta-programming consumers working with a given language. These nodes can reduce to an ET with exact semantics represented in common ET v2 nodes types.

2.6 Array Access

In ETs v2 we have IndexExpression that handles array access, indexers, and indexed properties. You can use these as l-values for assignments and ref/out parameters. See section 2.5.1.

We continue to support the LINQ v1 ArrayIndex factory methods that return BinaryExpression and MethodCallExpression for fetching array elements for backward compatibility. Eventually, we will obsolete them in lieu of IndexExpression.

2.7 Blocks, Scopes, Variables, ParameterExpression, and Explicit Lifting

ETs v2 model variable references with ParameterExpression nodes and a node kind value of Parameter. Initially for readability of code we chose to introduce a VariableExpression node type. It turns out in practice this meant much code that processed ETs had to be duplicated due to static typing in languages or had to use typeof() to dispatch to the right code if using Expression as a variable's type. Very little code actually needed to treat the declarations or references differently.

ETs v2 includes a BlockExpression node type that has an explicit list of variables. It creates a binding scope. Producers of ETs can use these to introduce new lexical variables, including temporaries. Blocks do not guarantee definite assignment. Languages need to do that when producing ETs. To initialize variables, BlockExpressions must explicitly include expressions in their body to set the variables. Definite assignment semantics is the concern of consumers of trees and compilers that enforce those semantics. Different languages have varying semantics here from definite assignment, to explicit unbound errors, to sentinel $Unassigned first class values. Lastly, recall ETs v2 is expression-based, so the value of a BlockExpression is the value of the last expression in its body.

Some languages have strong lexical semantics for unique binding of variables. For example, in Common Lisp or Scheme, each iteration around a loop or any basic block of code creates unique bindings for variables introduced in those basic blocks. Thus, returning a lambda would create unique closures environments for each iteration. Some languages, such as Python and F#, move all variables to the implicit scope of their containing function. ETs v2 supports both models, all depending on where you create the BlockExpression in the ET and list variables. For the

stronger lexical model, for example with the loop, place the BlockExpression inside the loop body and list variables there, instead of putting the Block Expression outside the loop or at the start of the function. See section 4.22.1.3 for an example.

ETs v2 also supports explicit lifting of variables to support languages that provide explicit meta-programming of local variables. For example, Python has a "locals" keyword that returns a dictionary of the lexical variables within a function so that you can manipulate them or 'eval' code against your function's environment. You can use the RuntimeVariablesExpression to list the ParameterExpressions that you need explicitly lifted.

~~Annotations on variable references are an issue. Each reference aliases the one ParameterExpression object representing the introduction of the variable to a lambda or a scope. Therefore, annotating each reference with source location to accurately report null deferences, unassigned references, etc., has to be managed outside the ParameterExpression node. This is design legacy from ETs v1~~.

2.8 Lambdas

Lambdas are modeled with LambdaExpression and Expression<T>. The latter derives from the former, and the T is a delegate type. LambdaExpression.Type holds the same T, and there is a ReturnType property that holds the type of value the T delegate would return. All lambdas created by the factory methods are actually Expression<T>. LambdaExpression provides the general base type for code that needs to process any lambda, or if you need to make a lambda with a computed delegate type at runtime. LambdaExpression supports two Compile methods that return a delegate of type LambdaExpression.Type, which can be invoked dynamically at run time.

Handling arbitrary returns or exits from lambdas has some interesting issues. Returns from a lambda can be arbitrarily deep in control constructs and blocks. Lexical exits represent a sort of non-local exit from nested control constructs. The expression that is the body of a lambda might have a particular type from the last sub expression it contains. Execution of the ET may never reach this last sub expression because of a Goto node. Even though we've added these sorts of control flow to ETs v2, they still keep the constraints ETs v1 had regarding LambdaExpression.Type and .Body.Type.

2.8.1 Ensuring Returns are Lexical Exits with Matching Types

We model returns with GotoExpression and LabelExpression. Due to how ETs v2 use these, we have the same verifiable properties discussed in section 2.4.2. We can verify exits are lexical. We can verify that the function's return type, its Body's type, and any lexical exits with result values all match in type. At one point in the ETs v2 design, we were not able to ensure the types invariant, and while the v1 behavior was not guaranteed, we didn't want to break that. Note, there is one case from v1 that counters this invariant property of matching types. When the lambda's delegate type has a void return, the body's type does not have to match since any resulting value is implicitly "converted to void" or squelched.

There is a factory method to create a LambdaExpression that only takes an Expression for the lambda's body and a parameters array. This factory method infers the lambda's return type from the body expression. For many ET node types, we only ensure a parent's Type property matches the sub expressions' Types that by default produce the result of the parent node. For

example, when constructing a BlockExpression, the factory only checks that the last sub expression has the same type as the BlockExpression. However, you can get proper checking on lexical exits from lambdas at creation time (without waiting to compile) using LabelExpression as the lambda's body.

The best pattern for creating lambdas with returns is to make the LambdaExpression's Body be a LabelExpression. The LabelExpression's default value expression is the actual body of the function. Then the actual body of the function, an expression, is the value left on the stack if you naturally flow through the expressions in the body. If you exit via a GotoExpression, you provide a value expression and a label target. This pattern enables ETs with a verifiable target at the end of the function and that any return value has the appropriate matching type.

You could use a BlockExpression as the LambdaExpression.Body. You would then have to put a LabelExpression as the last expression in the block. If the lambda returned non-void, you would need to make the BlockExpression.Type match the LambdaExpression.ReturnType. To do this, you would need to make the LabelExpression.Type match the block's type, and you would need to fill in the LabelExpression's default value expression with a DefaultExpression, supplying the lambda's return type. This is the normal way to think about creating a target to use as the exit location for the function, but it is much more work than using a LabelExpression as the lambda's body itself.

When rewriting reducible nodes in a lambda, if you need to create a return from the lambda, you'll need to ensure the LambdaExpression's Body is a LabelExpression. If there isn't one already, your rewriting visitor will need to create the label target and then as you unwind in the visitor, replace the Body of the LambdaExpression with a LabelExpression using the same label target.

Lastly, what about languages that allow return targets with dynamic extent? Some languages allow closing over function return or block labels (that is, true non-local returns). If these languages do not require full continuations semantics, they could easily map these non-local return label closures to throws and catches that implement the semantics.

2.8.2 Throw

We model Throw with a UnaryExpression. We had a ThrowExpression at one point. To be consistent with the "shape" aspect of re-using expression node types with the same kinds of children or properties, you can think of Throw as an operator with a single argument.

The Type property of the UnaryExpression with node kind Throw does not have to be void, which you may expect since the Throw never really returns. Using types other than void can be useful for ensuring a node has consistently typed children. For example, you might have a ConditionExpression with Type Foo, where the consequent returns a Foo, but the alternative is a Throw. Allowing the UnaryExpression with node kind Throw to have a non-Void Type, you do not have to artificially wrap your alternative in a BlockExpression and use a DefaultExpression in it.

CatchBlocks are helper objects that convey information to a TryExpression, like the MethodInfos or CallSiteBinder objects some Expression have. CatchBlock is not an expression as you might expect, primarily because they cannot appear anywhere any expression can appear. We could have thought of TryExpression as having a value from its Body, but sometimes its value comes from a CatchExpression. This would be analogous to ConditionalExpression. However, unlike a

language like Lisp where Catch is a first-class expression, we felt the model we settled on is both expression-based and more amenable to .NET programmers.

2.8.3 Recursion

ETs avoid needing a Y-Combinator expression by simply using assignment. An ET can have a BlockExpression with a variable assigned to a LambdaExpression. The LambdaExpression's body can have a MethodCallExpression that refers to a ParameterExpression that refers to the variable bound in the outer scope. The compile effectively closes over the variable to create the recursive function.

2.8.4 Tail Call

LambdaExpressions have a TailCall property that indicates whether compiling the lambda should attempt to use tail calls for any value returning expressions. If true, this is not a guarantee since some calls cannot be tailed called (for example, there may be write backs needed for some properties or ref args).

2.9 Generators (Codeplex only)

Cut from .NET 4.0, available on www.codeplex.com/dlr .

Generators are first-class concepts in the ET v2 model. However, they are only available in the DLR's Codeplex project. The code can be re-used readily and ships in IronPython and IronRuby. The basic model is that you create the LambdaExpression with your enumerable return type and then use a GeneratorExpression inside the lambda to get the yielding state machine. The outer lambda can do any argument validation needed, and the GeneratorExpression reduces to code that closes over the lambda's variables. The body of the GeneratorExpression can have YieldExpressions in it. The generator node reduces to an ET that open codes the state machine necessary for returning values and re-entering the state machine to re-establish any dynamic context (try-catch, etc.).

The main reason for not shipping generators in .NET 4.0 is they have a couple of features that are specific to Python, and we don't have time to abstract the design cleanly. Python allows yield from finally blocks, and while we could define what that means, we'd need offer some extensibility here; for example, what happens when the yield occurs via a call to Dispose. Python requires 'yield' to be able to pass a value from the iteration controller into the GeneratorExpression body, which the generator may use to change the iteration state.

2.10 Rich Static Call Site Modeling (v-next+1)

Cut from .NET 4.0, planned for v-next+1.

We considered adding ComplexMethodCallExpression, ComplexIndexExpression, and ComplexNewExpression to model function calls with unsupplied arguments and named argument values. We need these eventually for call sites that do not simply supply required positional arguments. However, there are no language features in .NET 4.0 VS languages that demand we add this modeling in .NET 4.0, and there are no Codeplex languages we ship that would use the support if we added it. The reason no VS languages need this is that there are no

plans to enhance the lambda syntax we have to permit control flow, assignments, or new language features inside of lambdas.

When we do add support for rich static call sites, we cannot add an ArgumentsInfo property to MethodCallExpression. This would break backward compatibility. If we added this extra property that is sometimes null, then when it wasn't null, the node would have different semantics than it did in ETs v1. ET v1 consumer code would not check for the new property, and sometimes the argument processing would be incorrect. We have a first-cut design on what we will add in v-next+1. The rest of this section outlines what we think is a nice general model.

These nodes would have two arrays, .ArgumentEvaluationOrder and . ArgumentDescriptions. The would have parallel elements. The first just contains the supplied expression in the appropriate order to evaluate them. It contains only as many expression as the number that appeared at the call site. The descriptions array contains ArgumentDescriptions, and each element describes the argument in the corresponding element of the first array. The descriptions detail whether the argument was positional or named, as well as in what position it was supplied. For example, in the expression (however kooky) "foo(,,3,x=5)", the arrays would be of length two, and the descriptions would describe the arguments in positions 2 and 3.

We will NOT model 'ref' and 'out' arguments in the ArgumentDescription objects for ComplexMethodCallExpression. You do not need these for correctness in producing an ET or for compiling it. If you parse a 'ref' argument in source at a call site, you can bake that information into the delegate type that is the T in the resulting dynamic CallSite<T> (see spec for "Sites, Binders, and Dynamic Object Interop"). If you do not parse 'ref', but your language's binder on sees an appropriate method at runtime with a 'ref' parameter, then you know to throw an exception. You only need the 'ref' and 'out' modeling for meta-programming convenience. Since 'out' is a single language oddity, and adding 'ref' for convenience only would require MethodCallExpression to have an Argument Description array and sometimes to be reducible, we will not model 'ref' and 'out' in ArgumentDescriptions.

2.11 Expression "Tree" Where Possible

With all but two exceptions, Expression Trees truly are trees. However, there are technically two common ways in which they are a DAGs, which is still good, but they are not trees.

One case is ParameterExpression from ET v1 where the declaration site in the ET and the reference sites alias the ParameterExpression instances. This does not create any cycles, but it does mean we have pairs of vertices with more than one path joining them.

The other case is LabelExpression. We clearly want to avoid having BreakExpression or GotoExpression pointing to some other ET node such that traversing a tree would mean having to detect cycles. We do this by having LabelExpression nodes that point to LabelTargets. Then ET nodes, such as GotoExpression, can point to the LabelTarget of the LabelExpression marking the destination.

There are of course other ways to break the tree nature of the graph. You might alias the same MethodInfo in various places or ConstantExpressions. You might rewrite a tree and re-use sub ETs in different locations. None of this should be an issue since we don't have cycles, but purists should realize ETs are DAGs.

2.12 Serializability

The ability to serialize an ET is important. You may want to send the ET as a language-neutral semantic representation of code to a remote execution environment or to another app domain. You might store pre-parsed code snippets as ETs as a representation higher level than MSIL and more readily executable than source. Nothing should prevent the ability to save an ET to disk and reconstitute it.

It will always be possible to create an ET that does not serialize. In fact, in the hosted language scenarios, most produced ETs may not serialize because they are created for immediate execution. In this case it is fine to directly point at data needed for execution that may not serialize. For example, DynamicExpression nodes can hold onto rich binding information for use at runtime. They may have references to the ScriptRuntime in which they execute or other execution context information.

If a language does need to refer to a ScriptRuntime or scope objects for free references to variables, then the language can still create serializable ETs. The entry point to executable code produced from an ET is always a lambda (even if it is an outer most lambda wrapping the top-level forms of a file of script code). The language can create the LambdaExpression with parameters for the ScriptRuntime and/or variables scope chain through ScriptRuntime.Globals. Since the language is in control when it invokes the lambda at a later time, or in another context, it can pass in the execution context the code needs. Finally, if the language uses DynamicExpression, it needs to ensure its CallSiteBinders are serializable.

2.13 Shared Visitor Support

ETs v2 provides a tree walker class you can derive from and customize. Customers often asked about tree walkers, and the DLR uses them a lot too. With the advent of Extension node kinds and reducibility, providing a walker model is even more important for saving work and having well-behaved extensions going forward. Without providing a walking mechanism out of the box, everyone would have to fully reduce extension nodes to walk them. Reducing is lossy for meta-programming because usually you can't go back to the original ET, especially if you're rewriting parts of the tree.

As an example, without the visitor mechanism, Extension node kinds inside of Quote (UnaryExpressions with node kind Quote) would be problematic. The Extensions would be black boxes, and Quote would be unable to substitute for ParameterExpressions in the black box. The Quote mechanism would need to fully reduce all nodes to substitute the ParameterExpressions. Then the quoted expression would not have the shape or structure that is expected when using Quote. This would make meta-programming with such expression work poorly.

The ExpressionVisitor class is abstract with two main entry points and many methods for sub classes to override. The entry points visit an arbitrary Expression or collection of Expressions. The methods for sub classes to override correspond to the node types. For example, if you only care to inspect or act on BinaryExpressions and ParameterExpressions, you'd override VisitBinary and VisitParameter. The methods you override all have default implementations that just visit their children. If the result of visiting a child produces a new node, then the default implementations construct a new node of the same kind, filling in the new children. If the child comes back identity equal, then the default just returns it.

As an Extension node kind author, you should override Expression.VisitChildren to visit your sub expressions. Furthermore, if any come back as new objects, you should reconstruct your node type with the new children, returning the new node as your result. By default VisitChildren reduces the expression to a common node and then calls the visitor on the result. As discussed above, this is not the best result for meta-programming purposes, so it is important that Extension nodes override this behavior.

2.14 Design Impact from Performance Improvements

We increased the surface area slightly for performance gains. The properties on Expression for fetching the node kind and Type properties from sub classes are now virtual. In many cases these needlessly took up a slot in the Expression object. Removing the backing fields for these members where possible saved about 50% of ET working set on real code processing in the DLR.

There are also several factory overloads to avoid having to allocate read-only collections. In typical usage before this change, a language would have an original array (of, say, expressions) and then call a factory to create an Expression with a read-only collection of children. The factory would have to allocate a new array to guarantee it being read-only. Then when someone fetched the Expression children, the node would wrap the array in a ReadOnlyCollection. We now use factories that take up to N children as individual arguments, create some clever sub classes of the node type, and leverage those for smart ReadOnlyCollection implementations. This saved about 70% of the read-only collection impact throughout real usage of ETs for the DLR.

2.15 Annotations (Debug/Source Information Only)

We had a general annotation mechanism for a long time, but it kept presenting issues both in terms of the nature of the annotations and rewriting trees while correctly preserving annotations. We concluded that most annotations are short-lived for a single phase of processing, and they did not need node identity across tree rewrites. One common case of a persisted annotation, source locations, needed to span phases and rewrites, so we kept support for them specifically. We cut the general support for this release and will look at adding it back in a future release.

Note, all factory methods return new objects each time you call them. They return fresh objects so that you can associate them with unique annotations when that's needed. If you need caching for working set pressure or other performance turning (for example, re-using all Expression.Constant(1) nodes), then you need to provide that.

2.15.1 Source Location Information

We model source location information with a DebugInfoExpression that represents a point in the ET where there is debugging information (a la .NET sequence points). A later instance of this class with the IsClear property set to True clears the debugging information. This node type has properties for start and end location information. It also can point to SymbolDocumentInfo which contains file, document type, language, etc.

2.15.2 CUT General Annotations Support

~~ETs v2 provide an annotation mechanism. This is useful for debug information and source locations. Different tools, ET producers and consumers, can tag ET nodes with access information or other metadata.~~

~~The information is immediately available via an Annotations object to which the ET node refers. We considered having the mechanism be completely outside the ETs. This was a more complicated model. It involved hash identities, aside tables, callback convention on transformations, etc. ET nodes do not have strong object identify in the presence of transformations. Now when tools transform ETs, the annotations on nodes naturally travel with them. They can also be modified or moved when creating new nodes.~~

~~Annotations instances are immutable. Of course, users can add an annotation member that is an indirection point, from which they can maintain mutable information. You might do this in some processing pass where an annotation changes, but you do not want to incur the cost of copying the sub ET from the node to the root of its containing tree over and over.~~

~~The information element in the Annotations object is keyed by a type. An Annotations object can have more than one element of a given type.~~

2.16 Node Kind and Operator Enum Values

Each Expression has a NodeType property with a value from the ExpressionType enum. Each kind of ET node (kind, not node type) is uniquely identified by an ExpressionType value in the node's NodeType property. For example, a BinaryExpression node type could have NodeType values indicating different kinds of binary expressions, Add vs. Multiply.

For DynamicExpressions with the BinaryOperationBinder or UnaryOperationBinder, for example, the binder has an Operation property respresenting the abstract operation. We've re-used the ExpressionType enum for modeling the operation in the binder. This has the benefit of fewer types and consistency with operator modeling. The down side is that ExpressionType has elements that are only used either for binders or for ET node kinds.

2.17 ToString Method

Expression.ToString is for light weight debugging purposes only. We do not return strings with a helpful structural representation of the ET, as you may want when developing a language on the Dynamic Language Runtime or returning expressions from DLR MetaObjects. For that, there are helpers in the DLR code built on Codeplex.com, Expression.Dump and Expression.DumpExpression. These members that produce a form of pretty printed tree may move into v-next+1 or some other version of .NET when we have more time to bake them.

ToString does not try to return semantically accurate C# or VB code in particular. We try to return terse strings loosely suggesting what an ET node contains for quick inspection only.

3 ET Runtime

There are a few functions that are required by the code the ET compiler produces. When generating and compiling ETs for execution within the same process, there are no issues

whatsoever. The following functions or any changes to them will be consistent within the version of the .NET Framework that you are using:

```
RuntimeOps.MergeRuntimeVariables(
    IRuntimeVariables first, IRuntimeVariables second, int[]
indexes)
RuntimeOps.Quote(Expression expression, object hoistedLocals,
                 object[] locals)
```

If you use LambdaExpression.CompileToMethod to generate a DLL with your code, then you cannot use UnaryExpression with node kind Quote, which requires the above functions. However, CompileToMethod may emit calls to these methods:

```
IRuntimeVariables RuntimeOps.CreateRuntimeVariables(
    object[] data, long[] indexes)
IRuntimeVariables RuntimeOps.CreateRuntimeVariables()
```

We need to keep these around and working if the DLL the customer writes out is otherwise working in future version of the CLR.

4 API Reference

This section contains sub sections for each type of ET node, and they have sub sections for each member. While all factories are on the Expression type, this document describes them in a sub section for the type of ET node that the factories create. All factory methods return new objects each time you call them. They return fresh objects so that you can associate them with unique annotations when that's needed. If you need caching for working set pressure or other performance turning (for example, re-using all Expression.Constant(1) nodes), then you need to provide that.

For economy of definition, this spec often refers to C# operators or behaviors. The ET model is explicitly NOT slanted towards C#. There are times when the nodes do not have equivalent C# semantics for generality of modeling, for example, the TypeIs node kind or BinaryExpressions with comparison node kinds. It is often MUCH easier for the reader to site a C# behavior with equivalent semantics than to spec fundamental operators, CLR behaviors, numeric types, etc., to capture the semantics of an ET node.

4.1 Terminology for Lifted V1 Spec Text (marked by boxed red text)

The Expression class contains many static factory methods. These are the only public means of creating ETs. The factory methods are described in the sub sections for the classes identified as the factory method return types.

Integral type -- one of byte, sbyte, short, ushort, int, uint, long, ulong, or the corresponding nullable types

Numeric type -- an integral type; one of float, double, decimal, or char; or the corresponding nullable types

Predefined type -- a numeric or boolean type (usually written as "numeric or boolean" instead of "predefined" to avoid ambiguity with BCL types)

Predefined operator -- one that is specified to exist by the C# Language Specification (usually written with "C#" in front to avoid ambiguity with IL instructions).

A type T1 is assignable to another T2 only if one of the following criteria holds:

- T1 and T2 are the same
- T1 and T2 are classes and T1 directly or indirectly inherits from T2
- T1 and T2 are interfaces and T1 directly or indirectly inherits from T2
- T1 is an array type T[] and T2 is IList<T> or one of the interfaces it derives from

Note that a value type is only assignable to itself. Assignability does not include notions of boxing, conversions, etc., so while a value in some languages may be allowed to be assigned to type Object, in an ET, you would need an explicit conversion to Object.

4.2 Quirks Mode for Silverlight

Silverlight has a high bar for compatibility between its releases. Due to code shipped in SL3, for which we fixed bugs, we had to keep some behaviors when programmers throw the Quirks Mode switch for SL4 and later SL versions. Here's a list of those changes when you use ETs in Quirks Mode (these are NOT the behaviors in .NET 4.0 or desktop builds of DLR from Codeplex.com):

- Improvements to ToString results are reverted in Quirks Mode.
- The Quote factory method allows calls in Quirks Mode that resulted in trees that caused errors as explained in section 4.4.42.
- The Call factory method does not throw an error when you pass a 'this' argument to a static method in Quirks Mode.
- When passing get_... and set_... MemberInfos to factory methods, we do not map them in some cases to PropertyInfos in Quirks Mode.
- If you pass a ReadOnlyCollection<T> to some factory method in Quirks Mode, you will be able to mutate the underlying data and effectively undermind the immutability of ET nodes.

4.3 Expression Abstract Class

This abstract class is the base class from which all ET node types derive. This class contains many static factory methods (listed in the class summary) for sub types. See the node type sub section for factory methods semantics.

If we add annotations back in v-next+1, we need to make sure we capture that passing null Annotations is the same as supplying Annotations.Empty.

4.3.1 Class Summary

```
public abstract class Expression {
    protected Expression();

    public virtual ExpressionType NodeType { get; }
    public virtual Type Type { get; }

    public virtual Boolean CanReduce { get; }
    public virtual Expression Reduce();
```

```
public Expression ReduceAndCheck();
public Expression ReduceExtensions();

public static Type GetActionType(params Type[] typeArgs);
public static bool TryGetActionType(Type[] typeArgs,
                                    out Type actionType)
public static Type GetFuncType(params Type[] typeArgs);
public static bool TryGetFuncType(Type[] typeArgs,
                                  out Type funcType);
public static Type GetDelegateType(params Type[] typeArgs)

protected virtual Expression VisitChildren
    (ExpressionVisitor visitor);
protected internal virtual Expression Accept
   (ExpressionVisitor visitor)


// These build only on Codeplex.com for debugging aids (may
// be added in v-next+1).  DebugView is build into CLR 4.0, but it
// it is private for use with VS datatips and debugging tools only.
public string DebugView {get {}}
public void DumpExpression(string descr, TextWriter writer) {

public static BinaryExpression Add
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression Add(Expression left,
                                   Expression right);
public static BinaryExpression AddAssign
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression AddAssign(Expression left,
                                         Expression right);
public static BinaryExpression AddAssign
    (Expression left, Expression right, MethodInfo method,
     LambdaExpression conversion)
public static BinaryExpression AddAssignChecked(Expression left,
                                                Expression right);
public static BinaryExpression AddAssignChecked
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression AddAssignChecked
    (Expression left, Expression right, MethodInfo method,
     LambdaExpression conversion)
public static BinaryExpression AddChecked
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression AddChecked(Expression left,
                                          Expression right);

public static BinaryExpression And(Expression left,
                                   Expression right);
public static BinaryExpression And
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression AndAlso(Expression left,
                                       Expression right);
public static BinaryExpression AndAlso
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression AndAssign(Expression left,
                                         Expression right);
public static BinaryExpression AndAssign
```

```
    (Expression left, Expression right, MethodInfo method);


public static IndexExpression ArrayAccess
    (Expression array, IEnumerable<Expression> indexes);
public static IndexExpression ArrayAccess
    (Expression array, params Expression[] indexes);

// The ArrayIndex factories will be obsolete in lieu of more
// general IndexExpression factories above.
public static BinaryExpression ArrayIndex(Expression array,
                                          Expression index);
public static MethodCallExpression ArrayIndex
    (Expression array, params Expression[] indexes);
public static MethodCallExpression ArrayIndex
    (Expression array, IEnumerable<Expression> indexes);

public static UnaryExpression ArrayLength(Expression array);

public static BinaryExpression Assign(Expression left,
                                      Expression right);

public static MemberAssignment Bind(MethodInfo propertyAccessor,
                                    Expression expression);
public static MemberAssignment Bind(MemberInfo member,
                                    Expression expression);

public static BlockExpression Block
    (IEnumerable<ParameterExpression> variables,
     params Expression[] expressions);
public static BlockExpression Block
    (Type type,
     IEnumerable<ParameterExpression> variables,
     params Expression[] expressions);
public static BlockExpression Block
    (IEnumerable<ParameterExpression> variables,
     IEnumerable<Expression> expressions);
public static BlockExpression Block
    (Type type,
     IEnumerable<ParameterExpression> variables,
     IEnumerable<Expression> expressions);
public static BlockExpression Block
    (IEnumerable<Expression> expressions);
public static BlockExpression Block
    (Type type, IEnumerable<Expression> expressions);
public static BlockExpression Block
    (Expression arg0, Expression arg1, Expression arg2,
     Expression arg3, Expression arg4);
public static BlockExpression Block
    (params Expression[] expressions);
public static BlockExpression Block
    (Type type, params Expression[] expressions);
public static BlockExpression Block(Expression arg0,
                                    Expression arg1);
public static BlockExpression Block
    (Expression arg0, Expression arg1, Expression arg2);
public static BlockExpression Block
```

```
    (Expression arg0, Expression arg1, Expression arg2,
     Expression arg3);

public static GotoExpression Break(LabelTarget target,
                                   Expression value);
public static GotoExpression Break(LabelTarget target);


public static MethodCallExpression Call
    (MethodInfo method, params Expression[] arguments);
public static MethodCallExpression Call
    (MethodInfo method, Expression arg0, Expression arg1,
     Expression arg2, Expression arg3, Expression arg4);
public static MethodCallExpression Call
    (MethodInfo method, Expression arg0, Expression arg1);
public static MethodCallExpression Call
    (Expression instance, MethodInfo method,
     params Expression[] arguments);
public static MethodCallExpression Call
    (Expression instance, MethodInfo method, Expression arg0,
     Expression arg1, Expression arg2);
public static MethodCallExpression Call
    (Expression instance, MethodInfo method, Expression arg0,
     Expression arg1);
public static MethodCallExpression Call(MethodInfo method,
                                        Expression arg0);
public static MethodCallExpression Call
    (Expression instance, String methodName, Type[] typeArguments,
     params Expression[] arguments);
public static MethodCallExpression Call
    (Type type, String methodName, Type[] typeArguments,
     params Expression[] arguments);
public static MethodCallExpression Call
    (Expression instance, MethodInfo method,
     IEnumerable<Expression> arguments);
public static MethodCallExpression Call
    (MethodInfo method,
     IEnumerable<Expression> arguments);
public static MethodCallExpression Call
    (MethodInfo method, Expression arg0, Expression arg1,
     Expression arg2, Expression arg3);
public static MethodCallExpression Call
    (MethodInfo method, Expression arg0, Expression arg1,
     Expression arg2);
public static MethodCallExpression Call(Expression instance,
                                        MethodInfo method);

public static CatchBlock Catch(ParameterExpression variable,
                               Expression body);
public static CatchBlock Catch(Type type, Expression body);
public static CatchBlock Catch
    (ParameterExpression variable, Expression body,
     Expression filter);
public static CatchBlock Catch(Type type, Expression body,
                               Expression filter);

public static BinaryExpression Coalesce(Expression left,
```

```
                                          Expression right);
public static BinaryExpression Coalesce
    (Expression left, Expression right,
     LambdaExpression conversion);

public static ConditionalExpression Condition
    (Expression test, Expression ifTrue, Expression ifFalse);
public static ConditionalExpression Condition
    (Expression test, Expression ifTrue, Expression ifFalse,
     Type type);
public static ConditionalExpression IfThen
    (Expression test, Expression ifTrue)
public static ConditionalExpression IfThenElse
    (Expression test, Expression ifTrue, Expression ifFalse);

public static ConstantExpression Constant(Object value);
public static ConstantExpression Constant(Object value,
                                          Type type);

public static GotoExpression Continue(LabelTarget target);

public static UnaryExpression Convert
    (Expression expression, Type type, MethodInfo method);
public static UnaryExpression Convert(Expression expression,
                                      Type type);
public static UnaryExpression ConvertChecked
    (Expression expression, Type type, MethodInfo method);
public static UnaryExpression ConvertChecked
    (Expression expression, Type type);

public static CatchBlock CreateCatchBlock
    (Type type, ParameterExpression variable, Expression body,
     Expression filter);

public static DebugInfoExpression DebugInfo
    (SymbolDocumentInfo document, Int32 startLine,
     Int32 startColumn, Int32 endLine, Int32 endColumn);
public static DebugInfoExpression ClearDebugInfo
    (SymbolDocumentInfo document)

public static SymbolDocumentInfo SymbolDocument(String fileName,
                                                Guid language);
public static SymbolDocumentInfo SymbolDocument(String fileName);
public static SymbolDocumentInfo SymbolDocument
    (String fileName, Guid language, Guid languageVendor);
public static SymbolDocumentInfo SymbolDocument
    (String fileName, Guid language, Guid languageVendor,
     Guid documentType);

public static UnaryExpression Decrement(Expression expression,
                                        MethodInfo method);
public static UnaryExpression Decrement(Expression expression);

public static DefaultExpression Default(Type type);
public static DefaultExpression Empty();

public static BinaryExpression Divide(Expression left,
```

```
                                        Expression right);
public static BinaryExpression Divide
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression DivideAssign(Expression left,
                                            Expression right);
public static BinaryExpression DivideAssign
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression DivideAssign
    (Expression left, Expression right, MethodInfo method,
     LambdaExpression conversion)

public static ElementInit ElementInit
    (MethodInfo addMethod,
     IEnumerable<Expression> arguments);
public static ElementInit ElementInit
    (MethodInfo addMethod, params Expression[] arguments);

public static BinaryExpression Equal(Expression left,
                                     Expression right);
public static BinaryExpression Equal
    (Expression left, Expression right, Boolean liftToNull,
     MethodInfo method);
public static BinaryExpression ReferenceEqual(Expression left,
                                              Expression right)

public static BinaryExpression NotEqual(Expression left,
                                        Expression right);
public static BinaryExpression NotEqual
    (Expression left, Expression right, Boolean liftToNull,
     MethodInfo method);
public static BinaryExpression ReferenceNotEqual(Expression left,
                                                 Expression right)


public static BinaryExpression ExclusiveOr
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression ExclusiveOr(Expression left,
                                           Expression right);
public static BinaryExpression ExclusiveOrAssign
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression ExclusiveOrAssign
    (Expression left, Expression right);
public static BinaryExpression ExclusiveOrAssign
    (Expression left, Expression right, MethodInfo method,
     LambdaExpression conversion)

public static MemberExpression Field(Expression expression,
                                     String fieldName);
public static MemberExpression Field
    (Expression expression, Type type, String fieldName);
public static MemberExpression Field(Expression expression,
                                     FieldInfo field);

public static GotoExpression Goto(LabelTarget target);
public static GotoExpression Goto(LabelTarget target,
                                  Expression value);
```

```
public static BinaryExpression GreaterThan
    (Expression left, Expression right, Boolean liftToNull,
     MethodInfo method);
public static BinaryExpression GreaterThan(Expression left,
                                           Expression right);
public static BinaryExpression GreaterThanOrEqual
    (Expression left, Expression right, Boolean liftToNull,
     MethodInfo method);
public static BinaryExpression GreaterThanOrEqual
    (Expression left, Expression right);

public static UnaryExpression Increment(Expression expression,
                                        MethodInfo method);
public static UnaryExpression Increment(Expression expression);

public static InvocationExpression Invoke
    (Expression expression, params Expression[] arguments);
public static InvocationExpression Invoke
    (Expression expression, IEnumerable<Expression> arguments);

public static UnaryExpression IsFalse(Expression expression) {
public static UnaryExpression IsFalse(Expression expression,
                                      MethodInfo method) {
public static UnaryExpression IsTrue(Expression expression) {
public static UnaryExpression IsTrue(Expression expression,
                                     MethodInfo method) {

public static LabelTarget Label(Type type, String name);
public static LabelTarget Label(Type type);
public static LabelTarget Label();
public static LabelTarget Label(String name);
public static LabelExpression Label(LabelTarget target);
public static LabelExpression Label(LabelTarget target,
                                    Expression defaultValue);

public static LambdaExpression Lambda
    (Expression body, params ParameterExpression[] parameters);
public static LambdaExpression Lambda
    (Expression body,
     IEnumerable<ParameterExpression> parameters);
public static Expression<TDelegate> Lambda<TDelegate>
    (Expression body, String name,
     IEnumerable<ParameterExpression> parameters);
public static Expression<TDelegate> Lambda<TDelegate>
    (Expression body, params ParameterExpression[] parameters);
public static Expression<TDelegate> Lambda<TDelegate>
    (Expression body,
     IEnumerable<ParameterExpression> parameters);
public static LambdaExpression Lambda
    (Expression body, String name,
     IEnumerable<ParameterExpression> parameters);
public static LambdaExpression Lambda
    (Type delegateType, Expression body, String name,
     IEnumerable<ParameterExpression> parameters);
public static LambdaExpression Lambda
    (Type delegateType, Expression body,
     params ParameterExpression[] parameters);
```

```
public static LambdaExpression Lambda
    (Type delegateType, Expression body,
     IEnumerable<ParameterExpression> parameters);

public static BinaryExpression LeftShift
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression LeftShift(Expression left,
                                         Expression right);
public static BinaryExpression LeftShiftAssign(Expression left,
                                               Expression right);
public static BinaryExpression LeftShiftAssign
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression LeftShiftAssign
    (Expression left, Expression right, MethodInfo method,
     LambdaExpression conversion)
public static BinaryExpression LessThan
    (Expression left, Expression right, Boolean liftToNull,
     MethodInfo method);
public static BinaryExpression LessThan(Expression left,
                                        Expression right);
public static BinaryExpression LessThanOrEqual
    (Expression left, Expression right, Boolean liftToNull,
     MethodInfo method);
public static BinaryExpression LessThanOrEqual(Expression left,
                                               Expression right);

public static MemberListBinding ListBind
    (MemberInfo member,
     IEnumerable<ElementInit>
     initializers);
public static MemberListBinding ListBind
    (MemberInfo member, params ElementInit[] initializers);
public static MemberListBinding ListBind
    (MethodInfo propertyAccessor,
     IEnumerable<ElementInit>
     initializers);
public static MemberListBinding ListBind
    (MethodInfo propertyAccessor,
     params ElementInit[] initializers);

public static ListInitExpression ListInit
    (NewExpression newExpression,
     params ElementInit[] initializers);
public static ListInitExpression ListInit
    (NewExpression newExpression, MethodInfo addMethod,
     params Expression[] initializers);
public static ListInitExpression ListInit
    (NewExpression newExpression,
     IEnumerable<ElementInit> initializers);
public static ListInitExpression ListInit
    (NewExpression newExpression,
     IEnumerable<Expression> initializers);
public static ListInitExpression ListInit
    (NewExpression newExpression,
     params Expression[] initializers);
public static ListInitExpression ListInit
    (NewExpression newExpression, MethodInfo addMethod,
```

```
     IEnumerable<Expression> initializers);

public static LoopExpression Loop(Expression body);
public static LoopExpression Loop
    (Expression body, LabelTarget break, LabelTarget continue);
public static LoopExpression Loop(Expression body,
                                  LabelTarget break);

public static BinaryExpression MakeBinary
    (ExpressionType binaryType, Expression left,
     Expression right);
public static BinaryExpression MakeBinary
    (ExpressionType binaryType, Expression left, Expression right,
     Boolean liftToNull, MethodInfo method);
public static BinaryExpression MakeBinary
    (ExpressionType binaryType, Expression left, Expression right,
     Boolean liftToNull, MethodInfo method,
     LambdaExpression conversion);

public static GotoExpression MakeGoto
    (GotoExpressionKind kind, LabelTarget target,
     Expression value);

public static IndexExpression MakeIndex
    (Expression instance, PropertyInfo indexer,
     IEnumerable<Expression> arguments);

public static MemberExpression MakeMemberAccess
    (Expression expression, MemberInfo member);

public static UnaryExpression MakeUnary
    (ExpressionType unaryType, Expression operand, Type type);
public static UnaryExpression MakeUnary
    (ExpressionType unaryType, Expression operand, Type type,
     MethodInfo method);

public static MemberMemberBinding MemberBind
    (MemberInfo member,
     IEnumerable<MemberBinding> bindings);
public static MemberMemberBinding MemberBind
    (MemberInfo member, params MemberBinding[] bindings);
public static MemberMemberBinding MemberBind
    (MethodInfo propertyAccessor,
     IEnumerable<MemberBinding> bindings);
public static MemberMemberBinding MemberBind
    (MethodInfo propertyAccessor,
     params MemberBinding[] bindings);

public static MemberInitExpression MemberInit
    (NewExpression newExpression,
     params MemberBinding[] bindings);
public static MemberInitExpression MemberInit
    (NewExpression newExpression,
     IEnumerable<MemberBinding> bindings);

public static BinaryExpression Modulo
    (Expression left, Expression right, MethodInfo method);
```

```
public static BinaryExpression Modulo(Expression left,
                                      Expression right);
public static BinaryExpression ModuloAssign(Expression left,
                                            Expression right);
public static BinaryExpression ModuloAssign
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression ModuloAssign
    (Expression left, Expression right, MethodInfo method,
     LambdaExpression conversion)

public static BinaryExpression Multiply(Expression left,
                                        Expression right);
public static BinaryExpression Multiply
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression MultiplyAssign
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression MultiplyAssign
    (Expression left, Expression right, MethodInfo method,
     LambdaExpression conversion)
public static BinaryExpression MultiplyAssign
    (Expression left, Expression right);
public static BinaryExpression MultiplyAssignChecked
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression MultiplyAssignChecked
    (Expression left, Expression right);
public static BinaryExpression MultiplyChecked
    (Expression left, Expression right);
public static BinaryExpression MultiplyChecked
    (Expression left, Expression right, MethodInfo method);

public static NamedArgumentInfo NamedArg(String name);

public static UnaryExpression Negate(Expression expression);
public static UnaryExpression Negate(Expression expression,
                                     MethodInfo method);
public static UnaryExpression NegateChecked(Expression expression,
                                            MethodInfo method);
public static UnaryExpression NegateChecked
    (Expression expression);

public static NewExpression New
    (ConstructorInfo constructor,
     IEnumerable<Expression> arguments,
     params MemberInfo[] members);
public static NewExpression New(ConstructorInfo constructor);
public static NewExpression New(Type type);
public static NewExpression New(ConstructorInfo constructor,
                                params Expression[] arguments);
public static NewExpression New
    (ConstructorInfo constructor,
     IEnumerable<Expression> arguments);
public static NewExpression New
    (ConstructorInfo constructor,
     IEnumerable<Expression> arguments,
     IEnumerable<System.Reflection.MemberInfo> members);

public static NewArrayExpression NewArrayBounds
```

```
    (Type type,
     IEnumerable<Expression> bounds);
public static NewArrayExpression NewArrayBounds
    (Type type, params Expression[] bounds);
public static NewArrayExpression NewArrayInit
    (Type type,
     IEnumerable<Expression> initializers);
public static NewArrayExpression NewArrayInit
    (Type type, params Expression[] initializers);

public static UnaryExpression Not(Expression expression);
public static UnaryExpression Not(Expression expression,
                                  MethodInfo method);

public static UnaryExpression OnesComplement
    (Expression expression)
public static UnaryExpression OnesComplement
    (Expression expression, MethodInfo method)

public static BinaryExpression Or
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression Or(Expression left,
                                  Expression right);
public static BinaryExpression OrAssign(Expression left,
                                        Expression right);
public static BinaryExpression OrAssign
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression OrAssign
    (Expression left, Expression right, MethodInfo method,
     LambdaExpression conversion)
public static BinaryExpression OrElse(Expression left,
                                      Expression right);
public static BinaryExpression OrElse
    (Expression left, Expression right, MethodInfo method);

public static ParameterExpression Parameter
    (Type type, String name);
public static ParameterExpression Parameter(Type type)
public static ParameterExpression Variable
    (Type type, String name);
public static ParameterExpression Variable(Type type)

public static PositionalArgumentInfo PositionalArg
    (Int32 position);

public static UnaryExpression PostDecrementAssign
    (Expression expression);
public static UnaryExpression PostDecrementAssign
    (Expression expression, MethodInfo method);
public static UnaryExpression PostIncrementAssign
    (Expression expression);
public static UnaryExpression PostIncrementAssign
    (Expression expression, MethodInfo method);

public static BinaryExpression Power
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression Power(Expression left,
```

```
                                        Expression right);
public static BinaryExpression PowerAssign(Expression left,
                                           Expression right);
public static BinaryExpression PowerAssign
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression PowerAssign
    (Expression left, Expression right, MethodInfo method,
     LambdaExpression conversion)
public static UnaryExpression PreDecrementAssign
    (Expression expression);
public static UnaryExpression PreDecrementAssign
    (Expression expression, MethodInfo method);
public static UnaryExpression PreIncrementAssign
    (Expression expression, MethodInfo method);
public static UnaryExpression PreIncrementAssign
    (Expression expression);

public static MemberExpression Property(Expression expression,
                                        PropertyInfo property);
public static MemberExpression Property
    (Expression expression, MethodInfo propertyAccessor);
public static IndexExpression Property
    (Expression instance, PropertyInfo indexer,
     IEnumerable<Expression> arguments);
public static IndexExpression Property
    (Expression instance, PropertyInfo indexer,
     params Expression[] arguments);
public static IndexExpression Property
    (Expression instance, String propertyName,
     params Expression[] arguments);
public static MemberExpression Property(Expression expression,
                                        String propertyName);
public static MemberExpression Property
    (Expression expression, Type type, String propertyName);
public static MemberExpression PropertyOrField
    (Expression expression, String propertyOrFieldName);

public static UnaryExpression Quote(Expression expression);

public static UnaryExpression Rethrow();
public static UnaryExpression Rethrow(Type type);

public static GotoExpression Return(LabelTarget target,
                                    Expression value);
public static GotoExpression Return(LabelTarget target);

public static BinaryExpression RightShift
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression RightShift(Expression left,
                                          Expression right);
public static BinaryExpression RightShiftAssign(Expression left,
                                                Expression right);
public static BinaryExpression RightShiftAssign
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression RightShiftAssign
    (Expression left, Expression right, MethodInfo method,
     LambdaExpression conversion)
```

```
public static RuntimeVariablesExpression RuntimeVariables
    (IEnumerable<ParameterExpression>
     variables);
public static RuntimeVariablesExpression RuntimeVariables
    (params ParameterExpression[] variables);

public static BinaryExpression Subtract
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression Subtract(Expression left,
     Expression right);
public static BinaryExpression SubtractAssign(Expression left,
                                              Expression right);
public static BinaryExpression SubtractAssign
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression SubtractAssign
    (Expression left, Expression right, MethodInfo method,
     LambdaExpression conversion)

public static BinaryExpression SubtractAssignChecked
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression SubtractAssignChecked
    (Expression left, Expression right);
public static BinaryExpression SubtractChecked
    (Expression left, Expression right);
public static BinaryExpression SubtractChecked
    (Expression left, Expression right, MethodInfo method);

public static SwitchExpression Switch(Expression value,
                                      params SwitchCase[] cases);
public static SwitchExpression Switch
    (Expression switchValue, Expression defaultBody,
     params SwitchCase[] cases)
public static SwitchExpression Switch
    (Expression switchValue, Expression defaultBody,
     MethodInfo comparison, params SwitchCase[] cases)
public static SwitchExpression Switch
    (Type type, Expression switchValue, Expression defaultBody,
     MethodInfo comparison, params SwitchCase[] cases)
public static SwitchExpression Switch
    (Expression switchValue, Expression defaultBody,
     MethodInfo comparison, IEnumerable<SwitchCase> cases)
public static SwitchExpression Switch
    (Type type, Expression switchValue, Expression defaultBody,
     MethodInfo comparison, IEnumerable<SwitchCase> cases)

public static SwitchCase SwitchCase
    (Expression body, params Expression[] testValues)
public static SwitchCase SwitchCase
    (Expression body, IEnumerable<Expression> testValues)

public static UnaryExpression Throw(Expression value, Type type);
public static UnaryExpression Throw(Expression value);

public static TryExpression TryCatch
    (Expression body, params CatchBlock[] handlers);
public static TryExpression TryCatchFinally
```

```
    (Expression body, Expression finally,
     params CatchBlock[] handlers);
public static TryExpression TryFault(Expression body,
                                     Expression fault);
public static TryExpression TryFinally(Expression body,
                                       Expression finally);
public static TryExpression MakeTry
    (Type type, Expression body, Expression finally,
     Expression fault,
     IEnumerable<CatchBlock> handlers);

public static UnaryExpression TypeAs(Expression expression,
                                     Type type);
public static TypeBinaryExpression TypeEqual
    (Expression expression, Type type);
public static TypeBinaryExpression TypeIs(Expression expression,
                                          Type type);

public static UnaryExpression UnaryPlus(Expression expression);
public static UnaryExpression UnaryPlus(Expression expression,
                                        MethodInfo method);

public static UnaryExpression Unbox(Expression expression,
                                    Type type);
```

4.3.2 NodeType Property

This property returns the kind of expression. The Expression node's type may be identified by a single ExpressionType value; for example, ConstantExpression always has the value Constant. Some Expression node types have many possible sub kinds; for example, BinaryExpression may have the Add or Multiply.

This is virtual for subclasses of Expression that can return their node kind without having to store it in a field. For example, BlockExpression can save a word of memory, but BinaryExpression needs a field to store one of many applicable node kinds. Derivations of Expression that are not in the common set of nodes in .NET should return node kind Extension.

The value returned by this property should never change for a given object.

Signature:

```
Public virtual ExpressionType NodeType { get; }
```

4.3.3 Type Property

With the call that unbound trees are truly lang-specific and can't be shared, IsBound is cut, and Type must never be null again.

This property returns the System.Type object representing the result type of the expression this Expression object represents. Type can be null when the node is unbound. When the node is dynamic, the type is not necessarily System.Object; it may have a known expected result type.

This is virtual for subclasses of Expression that can (or need to) return their node's Type by computing it. For example, a non-void BlockExpression can return the Type of its last expression. Not requiring a Type backing field saves a lot of memory usage in ETs.

The value returned by this property should never change for a given object.

Signature:

```
public Type Type { get; }
```

4.3.4 CanReduce Property

This property returns whether the Reduce method returns a different but semantically equivalent ET. By default, this property returns false.

In the typical case, the resulting ET contains all common ET nodes suitable for passing to any common compilation or ET processing code. Sometimes the result is only partially reduced, and when walking the resulting ET, you'll need to further reduce some nodes.

The value returned by this property should never change for a given object.

Signature:

```
public virtual Boolean CanReduce { get; }
```

4.3.5 Reduce Method

This method returns a semantically equivalent ET representing the same expression. By default, this method returns the object on which it was invoked.

Typically the result comprises only common ET types, ET nodes suitable for passing to any compilation or ET processing code. Usually the result is only partially reduced (that is, only the root node). You'll probably need to further reduce some nodes.

Signature:

```
public virtual Expression Reduce();
```

4.3.6 ReduceAndCheck Method

This method returns a semantically equivalent ET representing the same expression by calling Reduce and then checking the result. The result is guaranteed to not be the same object and to have the same (or a reference assignable) Type property. If the result does not check positively, this method throws an exception.

Signature:

```
public virtual Expression ReduceAndCheck();
```

4.3.7 ReduceExtensions Method

This method returns a semantically equivalent ET representing the same expression by calling Reduce repeatedly until the result is a common ET node. The result may be reducible (for example, if it is a BinaryExpression with node kind AddAssign or a ForEachExpression).

This is the standard way for compilers like the expression compiler to reduce nodes to the core set in .NET.

Signature:

```
public virtual Expression ReduceExtensions();
```

4.3.8 DebugView Property

Only available privately for debugging support in CLR 4.0, may be productized in v-next+1, but this is public in the codeplex sources.

This property walks the expression tree and renders "pretty printing" of the tree for debugging purposes. When there's more experience with what is maximally useful, and in what format customers would like to see the pretty printing, we could bake these into a future version of .NET.

Signatures:

```
private string DebugView {get {}}
```

4.3.9 DumpExpression Method (Codeplex only)

Only available on Codeplex.com builds, may be productized in v-next+1.

This method walks the expression tree and renders "pretty printing" of the tree for debugging purposes. When there's more experience with what is maximally useful, and in what format customers would like to see the pretty printing, we could bake these into a future version of .NET.

Signatures:

```
public void DumpExpression(string descr, TextWriter writer) {
```

4.3.10 GetFuncType Method

This helper method creates Type objects for delegate types with non-void return values. This method constructs the Func<...> types from the generic System.Linq.Func delegates using the supplied types.

Signature:

```
public static Type GetFuncType(params Type[] typeArgs);
```

TypeArgs must contain at least one argument and at most 17 elements. If the elements of typeArgs represent the types T1…Tn, the resulting Type object represents the constructed delegate type System.Linq.Func<T1,…,Tn>. The last argument must be the return type.

4.3.11 TryGetFuncType Method

This helper method is just like GetFuncType, but it does not throw if given more than 17 type arguments, returning False instead. It also returns false if there are any ByRef parameters.

Signature:

```
public static bool TryGetFuncType(Type[] typeArgs,
                                  out Type funcType);
```

4.3.12 GetActionType Method

This helper method creates Type objects for delegate types with void return type. This method constructs the Action<...> types from the generic System.Linq.Action delegates using the supplied types.

Signature:

```
public static Type GetActionType(params Type[] typeArgs);
```

TypeArgs must contain at least one argument and at most 16 elements. If the elements of typeArgs represent the types T1…Tn, the resulting Type object represents the constructed delegate type System.Linq.Action<T1,…,Tn>.

4.3.13 TryGetActionType Method

This helper method is just like GetActionType, but it does not throw if given more than 16 type arguments, returning False instead. It also returns false if there are any ByRef parameters.

Signature:

```
public static bool TryGetActionType(Type[] typeArgs,
                                    out Type actionType)
```

4.3.14 GetDelegateType Method

This helper method creates Type objects for delegate types. It determines whether to return a Func or Action based on whether the last argument, the return type, is void. If possible, this returns an instantiated Func or Action type, as GetFuncType or GetActionType would. If necessary, this generates a new delegate type.

If invoked on the same type sequence as seen previously in an App Domain, this may return the same delegate object returned the first time, but there is no guarantee on this behavior. In general, if all of your parameter and return types are built into mscorlib or system.core (and therefore known not to be collectible), then this method caches the returned delegate type. If caching is important to the performance of your compiler, you should implement your own caching (re-using the code on codeplex.com and removing our limitations if you like).

This method is useful when calling the Lambda factories, and it is what the Lambda factories call when you do not supply the delegate type.

Signature:

```
public static Type GetDelegateType(params Type[] typeArgs)
```

4.3.15 VisitChildren Method

This virtual method is for sub classes of Expression that are not common node kinds. These nodes should have node kind Extension, and they should override this method. If you derive from Expression and do not override this method, then your extension will not work well with

visitors. The default VisitChildren will fully reduce your extension node which is sub optimal in meta-programming scenarios.

Signature:

```
protected virtual Expression VisitChildren
    (ExpressionVisitor visitor);
```

4.3.16 Accept Method

This method invokes the visitor's specific method for visiting nodes of this node's type; for example, MethodCallExpression overrides this to invoke ExpressionVisitor.VisitMethodCall. By default, this method calls ExpressionVisitor.VisitExtension.

It is rare to need to override this method, and it is available for subclasses only for marginal completeness. For example, if you had several customer node types and your own customer visitor, you could more directly dispatch to your visitor's VisitMumble methods for each Mumble extension node type.

Signature:

```
protected internal virtual Expression Accept
   (ExpressionVisitor visitor)
```

4.4 ExpressionType Enum

This enum has value to mark the kind of node an ET node is. Some of the ET node types are re-used for several expressions due to their "shape". For example, BinaryExpression has certain properties common to expressions such as addition, multiplication, and even assignment. When re-using BinaryExpression, the factory methods use unique values from this enum to distinguish the operation.

The following sub sections' text that describes the static node semantics for the enum members comes from the v1 spec ... it may have been edited for clarification.

4.4.1 Type Summary

```
public enum ExpressionType {
    Add,
    AddChecked,
    And,
    AndAlso,
    ArrayLength,
    ArrayIndex,
    Call,
    Coalesce,
    Conditional,
    Constant,
    Convert,
    ConvertChecked,
    Divide,
    Equal,
    ExclusiveOr,
    GreaterThan,
    GreaterThanOrEqual,
```

```
Invoke,
Lambda,
LeftShift,
LessThan,
LessThanOrEqual,
ListInit,
MemberAccess,
MemberInit,
Modulo,
Multiply,
MultiplyChecked,
Negate,
UnaryPlus,
NegateChecked,
New,
NewArrayInit,
NewArrayBounds,
Not,
NotEqual,
Or,
OrElse,
Parameter,
Power,
Quote,
RightShift,
Subtract,
SubtractChecked,
TypeAs,
TypeIs,
Assign,
Block,
DebugInfo,
Decrement,
Dynamic,
Default,
Extension,
Goto,
Increment,
Index,
Label,
RuntimeVariables,
Loop,
Switch,
Throw,
Try,
Unbox,
AddAssign,
AndAssign,
DivideAssign,
ExclusiveOrAssign,
LeftShiftAssign,
ModuloAssign,
MultiplyAssign,
OrAssign,
PowerAssign,
RightShiftAssign,
SubtractAssign,
```

```
AddAssignChecked,
MultiplyAssignChecked,
SubtractAssignChecked,
PreIncrementAssign,
PreDecrementAssign,
PostIncrementAssign,
PostDecrementAssign,
TypeEqual,
OnesComplement,
IsTrue,
IsFalse
```

4.4.2 Add

Use Add in BinaryExpression to represent arithmetic addition without overflow checking. Given an Add node, exp, let e1 and e2 be the C# equivalent of exp.Left and exp.Right. Then the C# equivalent of exp is (e1) + (e2), including order of evaluation.

Use this in a DynamicExpression to represent a binary operator for asking the first object to add the second object to itself. Neither object is modified.

4.4.3 AddChecked

Use AddChecked in BinaryExpresion to represent arithmetic addition with overflow checking. Given an AddChecked node, exp, let e1 and e2 be the C# equivalent of exp.Left and exp.Right. Then the C# equivalent of exp is checked(unchecked(e1) + unchecked(e2)), including order of evaluation.

There is no use for this node kind in DynamicExpressions.

4.4.4 And

Use And in BinaryExpression to represent a bitwise or logical And operation. The semantics depends on whether your Left and Right expression are ints or bool. In the case where the operands are of type bool?, three-valued logic applies:

AND	False	null	True		**OR**	False	null	True
False	False	False	False		False	False	null	True
null	False	null	null		null	null	null	True
True	False	null	True		True	True	True	True

Order of evaluation is Left then Right.

Use this in a DynamicExpression to represent a binary operator for asking the first object to perform a bitwise AND of its contents with the second object. Neither object is modified. Logical AND is "open coded" in ETs for operations on IDynamicMetaObjectProviders for more control over what a "true" or "false" value is.

4.4.5 AndAlso

Use AndAlso in BinaryExpression to represent a conditional And operator, evaluating the right operand only if necessary. Given an AndAlso node, exp, let e1 and e2 be the C# equivalent of exp.Left and exp.Right. Then the C# equivalent of exp is

((e1) == false) ? false : (e1) & (e2)

except that e1 is evaluated only once, and e2 may not be evaluated at all. Three valued logic applies to the AndAlso operator over nullable Booleans (see And enum member). In the case where both operands are non-nullable, the semantics is equivalent to (e1) && (e2).

There are some additional complexities with AndAlso/OrElse, in particular combining overloaded operators and nullables. Here is the comment from BinaryExpression.cs:

```
// For a userdefined type T which has op_False defined and L, R are
// nullable, (L AndAlso R) is computed as:
//
// L.HasValue
//     ? T.op_False(L.GetValueOrDefault())
//         ? L
//         : R.HasValue
//             ? (T?)(T.op_BitwiseAnd(L.GetValueOrDefault(),
//                                    R.GetValueOrDefault()))
//             : null
//     : null
```

There is no use for this node kind in DynamicExpressions since both operands would always be evaluated before passing them to the dynamic call site. However, you can produce a dynamic AndAlso by combining IsFalse and And with pseudo-code like the following (ignore mulitiple evaluations of sub expressions):

DynamicExpr(IsFalse, e1) ? false : Dynamic(And, e1, e2)

4.4.6 ArrayLength

Use ArrayLength with UnaryExpression to represent taking the length of a one-dimensional array.

There is no use for this node kind in DynamicExpressions.

4.4.7 ArrayIndex

The ArrayIndex node kind and Expression factories will be obsolete in lieu the more general IndexExpression node.

Use ArrayIndex in BinaryExpression to represent indexing into a one-diemnsional array. Given an ArrayIndex node, exp, let e1 and e2 be the C# equivalent of exp.Left and exp.Right. Then the C# equivalent of exp is (e1)[e2].

There is no use for this node kind in DynamicExpressions since a GetIndexBinder represents this operation.

4.4.8 Call

MethodCallExpressions use the Call node kind. It represents calling a method. Given a Call node, exp, let e0 be the C# equivalent of exp.object, let e1…en be the comma-separated list of C# expressions equivalent to the corresponding nodes in exp.Arguments and let m be the C# identifier denoting exp.Method. Furthermore let r1…rn be ref, out, or empty in accordance with possible modifiers on the corresponding parameter in exp.Method. Then the C# equivalent of exp is

(e0).m(r1 e1…rn en)

If Method denotes a static method, with T being the C# equivalent of the type it belongs to, the C# equivalent of exp is

T.m(r1 e1…rn en)

In the case of ref or out parameters on non-addressable expressions, the semantics is to create a local variable holding the value of the expression, passing that local instead of the expression itself as an argument to the call.

There is no use for this node kind in DynamicExpressions because they use InvokeMemberBinder objects to indicate the semantics, and they don't use node kinds.

4.4.9 Coalesce

Use Coalesce in BinaryExpression to represent a null coalescing expression. This operation is one that evaluates its first operand and conditionally evaluates any successive operands, returning the value of the first operand that has a non-null result. Given a Coalesce node, exp, let e1 and e2 be the C# equivalent of exp.Left and exp.Right. Then exp is similar to the C# expression, (e1) ?? (e2), including order of evaluation.

The difference in semantics is that a Coalesce node kind has a conversion lambda you can specify that executes last and converts either e1 or e2 to the node's Type. This conversion aspect is required in the node's semantics so that languages or ET producers can specify the exact overloaded conversion method or a custom method.

There is no use for this node kind in DynamicExpressions since dynamic languages typically perform these semantics naturally with their 'or' operator. If they don’t, then they can trivially open code the semantics with dynamic objects to appropriately avoid evaluating some operands.

4.4.10 Conditional

ConditionalExpressions use the Conditional node kind. It represents an if-then-else for value. Given a Conditional node created without a specified Type property (call the node "exp"), let e1, e2, and e3 be the C# equivalent of exp.Test, exp.IfTrue, and exp.IfFalse, respectively. Then the C# equivalent of exp is (e1) ? (e2) : (e3), including order of evaluation.

If you do supply the Type property when constructing the node, and it is void, then the sub expression types do not have to match, and any resulting value is "converted to void" or squelched. If you specify Type explicitly, then e2 and e3 must be reference-assignable to the type represented by the node's Type property.

There is no use for this node kind in DynamicExpressions since dynamic languages typically perform these semantics naturally with their 'if-then-else' or 'and'/'or' operators. If they don't, then they can trivially open code the semantics with dynamic objects to appropriately avoid evaluating some operands.

4.4.11 Constant

Constant is the node kind for a ConstantExpression node. It represents an expression that has a constant value. A Constant node, exp, may have any exp.Value, and the value may not have any syntactic representation in any programming language. The result of evaluating exp is exp.Value.

There is no use for this node kind in DynamicExpression.

4.4.12 Convert

Use Convert in a UnaryExpression node to represent an explicit numeric or enumeration conversion. The semantics is to silently overflow if the converted value does not fit the target type. Given a Convert node, exp, let e be the C# equivalent of exp.Operand, and let T be a type expression in C# for the type represented by exp.Type. The C# equivalent of exp is "(T)(e)". T is static compile-time information, so it is not evaluated at run time.

There is no use for this node kind in DynamicExpressions since a ConvertBinder represents this operation.

4.4.13 ConvertChecked

Use ConvertChecked in a UnaryExpression node to represent an explicit numeric or enumeration conversion that throws an exception if the converted value does not fit the target type. Given a ConvertChecked node, exp, let e be the C# equivalent of exp.Operand, and let T be a type expression in C# for the type represented by exp.Type. The C# equivalent of exp is "checked ((T)unchecked(e))". T is static compile-time information, so it is not evaluated at run time.

There is no use for this node kind in DynamicExpressions since a ConvertBinder represents this operation.

4.4.14 Divide

Use Divide in a BinaryExpression node to represent arithmetic division. Given a Divide node, exp, let e1 and e2 be the C# equivalent of exp.Left and exp.Right. Then the C# equivalent of exp is (e1) / (e2), including evaluation order. This is equivalent to the IL 'div' instruction (truncating integer division), so for example, -10/3 is -3.

Use this in a DynamicExpression to represent a binary operator for asking the first object to divide itself by the second object. Neither object is modified. If the objects are numbers, the expectation is that the result is an integer, rational, or floating point number.

4.4.15 Equal

Use Equal in BinaryExpression nodes to represent a comparison for equality. Given an Equal node, exp, let e1 and e2 be the C# equivalent of exp.Left and exp.Right. Then the C# expression,

(e1) == (e2), including order of evaluation is similar to the Equal node kind. The only difference is that reference types always compare a pointer equality.

Use this in a DynamicExpression to ask the first object to return whether it is equal to the second object. The equality test is a deep structural equality, not object identity or first level aggregation equality. The expectation is that this operator is a comparison returning a Boolean value, but the DLR does not enforce that.

4.4.16 ExclusiveOr

Use ExclusiveOr in BinaryExpression nodes to represent a bitwise xor operation. Given an ExclusiveOr node, exp, let e1 and e2 be the C# equivalent of exp.Left and exp.Right. Then the C# equivalent of exp is (e1) ^ (e2), including evaluation order.

Use this in a DynamicExpression to represent a binary operator for asking the first object to perform a bitwise XOR of its contents with the second object. Neither object is modified.

4.4.17 GreaterThan

Use GreaterThan in BinaryExpression nodes to represent a numeric comparison. Given a GreaterThan node, exp, let e1 and e2 be the C# equivalent of exp.Left and exp.Right. Then the C# equivalent of exp is (e1) > (e2), including evaluation order.

Use this in DynamicExpression to ask the first object to return whether it is greater than the second object. The expectation is that this operator is a comparison returning a Boolean value, but it could be a composition, I/O, or any kind of operator returning any type of value.

4.4.18 GreaterThanOrEqual

Use GreaterThanOrEqual in BinaryExpression nodes to represent a numeric comparison. Given a GreaterThanOrEqual node, exp, let e1 and e2 be the C# equivalent of exp.Left and exp.Right. Then the C# equivalent of exp is (e1) >= (e2), including evaluation order.

Use this in DynamicExpression to ask the first object to return whether it is greater than or equal to the second object. The equality test is a deep structural equality, not object identity or first level aggregation equality. The expectation is that this operator is a comparison returning a Boolean value, but it could be a composition, I/O, or any kind of operator returning any type of value.

4.4.19 Invoke

InvocationExpression nodes use this node kind. It represents invoking a delegate or lambda expression on a list of argument expressions. Given an Invoke node, exp, let e0 be the C# equivalent of exp.Expression, and let e1…en be the comma-separated list of C# expressions equivalent to the corresponding nodes in exp.Arguments. Then the C# equivalent of exp is (e0)(e1…en), including evaluation order.

If e0 evaluates to a value of type Expression<T>, the DLR compiles the lambda and then invokes it. C# does not allow this, but ETs do.

There is no use for this node kind in DynamicExpressions because they use InvokeBinder objects to indicate the semantics, and they don't use node kinds.

4.4.20 Lambda

LambdaExpressions use the Lambda node kind. They represent a lambda expression with a delegate type. Given a Lambda node, exp, let e be the C# equivalent of exp.Body, and let p1…pn be the comma separated list of C# parameters corresponding to each of the elements in exp.Parameters. Finally, let T be a type expression in C# for the type represented by exp.Type. Then the C# equivalent of exp is primarily "((T)((p1…pn) => e))", but there are some flags and features ET Lambda nodes support that C# does not.

There is no use for this node kind in DynamicExpressions.

4.4.21 LeftShift

Use LeftShift in BinaryExpression nodes to represent a bitwise left shift operation. Given a LeftShift node, exp, let e1 and e2 be the C# equivalent of exp.Left and exp.Right. Then the C# equivalent of exp is "(e1) << (e2)", including evaluation order.

Use this node kind in DynamicExpressions to ask the first object to shift its contents left by the number of positions indicated by the second object. Neither object is modified. Any vacant locations created on the right side of object one are filled by a default value appropriate to the first object and language that owns the object and shift semantics.

4.4.22 LessThan

Use LessThan in BinaryExpression nodes to represent a numeric comparison. Given a LessThan node, exp, let e1 and e2 be the C# equivalent of exp.Left and exp.Right. Then the C# equivalent of exp is "(e1) < (e2)", including evaluation order.

Use this in DynamicExpression to ask the first object to return whether it is less than the second object. The expectation is that this operator is a comparison returning a Boolean value, but it could be a composition, I/O, or any kind of operator returning any type of value.

4.4.23 LessThanOrEqual

Use LessThanOrEqual in BinaryExpression nodes to represent a numeric comparison. Given a LessThanOrEqual node exp, let e1 and e2 be the C# equivalent of exp.Left and exp.Right. Then the C# equivalent of exp is "(e1) <= (e2)", including evaluation order.

Use this in DynamicExpression to ask the first object to return whether it is less than or equal to the second object. The expectation is that this operator is a comparison returning a Boolean value, but it could be a composition, I/O, or any kind of operator returning any type of value.

4.4.24 ListInit

The ListInitExpression uses the ListInit node kind to represent creating a new collection object and initializing it from a list of elements. Given a ListInit node, exp, let c be the C# equivalent of exp.NewExpression, and let e1…en be the comma-separated list of C# expressions equivalent to the corresponding nodes in exp.Expressions. Then the C# equivalent of exp is "c{e1…en}"

There is no use for this node kind in DynamicExpressions since a CreateInstanceBinder represents this operation, combined with other expressions.

4.4.25 MemberAccess

A MemberExpression node uses MemberAccess node kind. It represents reading from a field or property, but as the Left expression of a BinaryExpression node with node kind Assign, this node kind represents a storable location or l-value. Given a MemberAccess node, exp, let e be the C# equivalent of exp.expression, and m be the C# identifier denoting exp.Member. Then the C# equivalent of exp is "(e).m". If Member denotes a static member, with T being the C# equivalent of the type it belongs to, the C# equivalent of exp is "T.m".

4.4.26 MemberInit

MemberInitExpression uses the MemberInit node kind. It represents creating a new object and initializing some of its members. Given a MemberInit node, exp, let c be the C# equivalent of exp.NewExpression, and let b1…bn be the comma-separated list of C# bindings equivalent to the corresponding nodes in exp.Bindings. Then the C# equivalent of exp is c{b1…bn}, including evaluation order.

There is no use for this node kind in DynamicExpressions since a CreateInstanceBinder represents this operation, combined with other expressions.

4.4.27 Modulo

Use Modulo in BinaryExpression nodes to represent computing an arithmetic remainder. Given a Modulo node, exp, let e1 and e2 be the C# equivalent of exp.Left and exp.Right. Then the C# equivalent of exp is (e1) % (e2). This is equivalent to the IL 'rem' instruction, so for example, -10 mod 3 is -1, and 10 mod -3 is 1.

Use this node kind in Dynamic Expression to ask the first object to divide itself by the second object. If the objects are numbers, the expectation is that the result is an integer remainder resulting from TruncateDivide'ing the first object by the second. Neither object is modified.

4.4.28 Multiply

Use Multiply in BinaryExpression nodes to represent arithmetic multiplication without overflow checking. Given a Multiply node, exp, let e1 and e2 be the C# equivalent of exp.Left and exp.Right. Then the C# equivalent of exp is "(e1) * (e2)".

Use this in a DynamicExpression to represent a binary operator for asking the first object to multiply itself by the second object. Neither object is modified. If the objects are numbers, the expectation is that the result is a number.

4.4.29 MulitplyChecked

Use MultiplyChecked in BinaryExpression nodes to represent arithmetic multiplication with overflow checking. Given a MultiplyChecked node, exp, let e1 and e2 be the C# equivalent of exp.Left and exp.Right. Then the C# equivalent of exp is "checked(unchecked(e1) * unchecked(e2))".

There is no use for this node kind in DynamicExpressions.

4.4.30 Negate

Use Negate in UnaryExpression nodes to represent arithmetic negation. Given a Negate node, exp, let e be the C# equivalent of exp.Operand. Then the C# equivalent of exp is "-(e)".

Use this in DynamicExpressions to request an object to produce its negated value.

4.4.31 UnaryPlus

Use UnaryPlus in a UnaryExpression node to represent keeping the sign for a numeric argument, which has no effect. In general this node kind is a no-op, but .NET does allow user-defined implementations for user types, which can do anything. Given a UnaryPlus node, exp, let e be the C# equivalent of exp.Operand. Then the C# equivalent of exp is "+(e)".

Use this in DynamicExpressions to request an object to produce its value with the same sign or whatever the object defines that it does for this operation.

4.4.32 NegateChecked

Use NegateChecked in a UnaryExpression node to represent arithmetic negation that has overflow checking.

There is no use for this in DynamicExpressions.

4.4.33 New

The NewExpression uses the New node kind. It represents calling a constructor to create a new object. Given a New node, exp, let T be the C# name of the declaring type of exp.Constructor, and let e1…en be the comma-separated list of C# expressions equivalent to the corresponding nodes in exp.Arguments. Then the C# equivalent of exp is "new T(e1…en)".

There is no use for this node kind in DynamicExpressions since a CreateInstanceBinder represents this operation, combined with other expressions possibly.

4.4.34 NewArrayInit

The NewArrayExpression node uses the NewArrayInit node kind. It represents creating a new one-dimensional array by specifying a list of elements. Given a NewArrayInit node, exp, let T denote the C# name for the element type of the array type represented by exp.Type, and let e1…en be the comma-separated list of C# expressions equivalent to the corresponding nodes in exp.Expressions. Then the C# equivalent of exp is "new T[]{e1…en}".

There is no use for this with DynamicExpression .

4.4.35 NewArrayBounds

Use NewArrayBounds in a NewArrayExpression node to represent creating a new array by specifying its bounds for each dimension. Given a NewArrayBounds node, exp, let T denote the C# name for the element type of the array type represented by exp.Type, and let e1…en be the comma-separated list of C# expressions equivalent to the corresponding nodes in exp.Expressions. Then the C# equivalent of exp is "new T[e1…en]".

There is no use for this node kind in DynamicExpression.

4.4.36 Not

Use the Not node kind in UnaryExpression nodes to represent bitwise complement or logical negation. Given a Not node, exp, let e be the C# equivalent of exp.Operand. If e has integral type, the C# equivalent of exp is "~(e)". If e has type bool, the C# equivalent of exp is "!(e)".

Use this in DynamicExpressions to ask an object to return its logical negation. Use OnesComplement in a DynamicExpression to ask an object to returns its bitwise negation.

4.4.37 NotEqual

Use NotEqual in BinaryExpressions to represent a comparison for operands to not be equal. Given a NotEqual node, exp, let e1 and e2 be the C# equivalent of exp.Left and exp.Right. Then the C# equivalent of exp is "(e1) != (e2)".

Use this in DynamicExpressions to ask an object to return whether it is not equal to the second object.

4.4.38 Or

Use Or in BinaryExpression nodes to represent a bitwise or logical Or operation. Given an Or node, exp, let e1 and e2 be the C# equivalent of exp.Left and exp.Right. Then the C# equivalent of exp is "(e1) | (e2)".

In the case where the operands are of type bool?, three-valued logic applies:

AND	False	null	True		**OR**	False	null	True
False	False	False	False		False	False	null	True
null	False	null	null		null	null	null	True
True	False	null	True		True	True	True	True

Use this in a DynamicExpression to represent a binary operator for asking the first object to perform a bitwise OR of its contents with the second object. Neither object is modified. Logical OR is "open coded" in ETs for operations on IDynamicMetaObjectProviders for more control over what a "true" or "false" value is.

4.4.39 OrElse

Use OrElse in a BinaryExpression node to represent a conditional or operator, evaluating the right operand only if necessary. Given an OrElse node, exp, let e1 and e2 be the C# equivalent of exp.Left and exp.Right. Then the C# equivalent of exp is "((e1) == true) ? true : (e1) | (e2)", except that e1 evaluates only once, and e2 may not evaluate at all.

Three valued logic applies to the OrElse operator over nullable booleans. In the case where both operands are non-nullable, the semantics is equivalent to "(e1) || (e2)".

There are some additional complexities with AndAlso/OrElse, in particular combining overloaded operators and nullables. Here is the comment from BinaryExpression.cs:

```
// For a userdefined type T which has op_True defined and L, R are
// nullable, (L OrElse R)  is computed as:
```

```
//
// L.HasValue
//     ? T.op_True(L.GetValueOrDefault())
//         ? L
//         : R.HasValue
//             ? (T?)(T.op_BitwiseOr(L.GetValueOrDefault(),
//                                   R.GetValueOrDefault()))
//             : null
//     : null
//
// This is the same behavior as VB. It's combining the normal pattern
// for short-circuiting operators, with the normal pattern for lifted
// operations: if either of the operands is null, the result is also
// null.
```

There is no use for this node kind in DynamicExpressions since both operands would always be evaluated before passing them to the dynamic call site.

4.4.40 Parameter

The ParameterExpression uses the Parameter node kind. It represents a reference to a variable via an identifier defined in the containing context.

Variables must be listed using ParameterExpressions as parameters for LambdaExpression or as lexical variables in a BlockExpression to (in effect) define them in some sub tree. To reference a variable, you alias the ParameterExpression object used to define the variable. Note, while Parameter node references are what determine variable binding, you can declare the same Parameter object in nested BlockExpressions. The ET compiler resovles the references to the tightest containing Block that declares the Parameter.

The name on Parameter is purely for debugging or pretty printing purposes and has no semantics to it.

There is no use for this node kind in DynamicExpressions themselves. Of course, when binders produce implementing expressions for dynamic operations, they may use ParameterExpressions.

4.4.41 Power

Use the Power node kind in BinaryExpression nodes to represent an exponentiation operation. The semantics is to invoke the System.Math.Pow function on the operands.

Use this in DynamicExpressions to request that the first object raise itself to the power of the second object. For numeric objects, you should expect a numeric result, ignoring NaN and overflow issues. Otherwise, the semantics is whatever the object implements. Neither object should be modified.

4.4.42 Quote

Use Quote in UnaryExpressions to represents an expression that has a "constant" value of type Expression. Unlike a Constant node, the Quote node specially handles contained ParameterExpression nodes. If a contained ParameterExpression node declares a local that

would be closed over in the resulting expression, then Quote replaces the ParameterExpression in its reference locations. At run time when the Quote node is evaluated, it substitutes the closure variable references for the ParameterExpression reference nodes, and then returns the quoted expression.

It is rare that an ET producer would need to use this factory or node kind. It exists with special semantics for use in LINQ v1 language features (see below). In v-next+1 we'll consider a complete and explicit quasi-quoting model.

There is no use for Quote in DynamicExpressions.

Here's an example that works:

```
ParameterExpression x = Expression.Parameter(typeof(int), "x");
ParameterExpression y = Expression.Parameter(typeof(int), "y");
Func<int, Expression> f = Expression.Lambda<Func<int, Expression>>(
    Expression.Quote(Expression.Lambda<Func<int, int>>(
        // This x is a reference from the inner lambda to the declared x
in
        // the outer lambda, but y is declared and referenced inside
Quote.
        Expression.Add(x, y), y)),
    // This use of the ParameterExpression declares the outer lambda's
x.
    x).Compile();
Func<int, int> z = ((Expression<Func<int, int>>)f(123)).Compile();
int a = z(111);
int b = z(222);
int c = z(333);
Console.WriteLine("a: {0}, b: {1}, c: {2}", a, b, c);
// a: 234, b: 345, c: 456
```

Note, the next two examples throw errors in ETs v2 in the Quote factory since it only accepts nodes with Type Expession<T> (that is, lambdas).

Here's an example from v1 that does NOT work as you might expect:

```
ParameterExpression x = Expression.Parameter(typeof(int), "x");
Func<int, Expression> f
   = Expression.Lambda<Func<int, Expression>>(Expression.Quote(x), x)
        .Compile();
Expression a = f(123);
Debug.Assert(a == x);   // This assert is true but unexpected.
```

In this case, Quote returned the ParameterExpression object (hence, a == x) since the variable didn't appear to come from a closed over reference.

Here's another example from v1 that does NOT work as you might expect:

```
ParameterExpression x = Expression.Parameter(typeof(int), "x");
Func<int, Expression> f = Expression.Lambda<Func<int, Expression>>(
    Expression.Invoke(Expression.Lambda<Func<Expression>>(Expression

.Quote(x))),
    // Here x is declared in an outer lambda (and closed over), BUT
    // Quote tries to cast its result to the node type (not its Type
    // property), assuming it is a lambda/delegate type.
    x
).Compile();
```

```
// Throws InvalidCastException: Unable to cast object of type
// 'MemberExpression' to type
// 'ParameterExpression'.
Expression a = f(123);
```

LINQ tries to convert the quoted expression as its actual type in case it gets passed to a function, for example, but the actual type becomes the closure variable reference, which is a MemberExpression.

4.4.43 RightShift

Use RightShift in BianryExpression nodes to represent a bitwise right shift operation. Given a RightShift node, exp, let e1 and e2 be the C# equivalent of exp.Left and exp.Right. Then the C# equivalent of exp is "(e1) >> (e2)".

Use this node kind in DynamicExpressions to ask the first object to shift its contents right by the number of positions indicated by the second object. Neither object is modified. Any vacant locations created on the left side of object one are filled by a default value appropriate to the first object and language that owns the object and shift semantics.

4.4.44 Subtract

Use Subtract in BinaryExpression nodes to represent arithmetic subtraction without overflow checking. Given a Subtract node, exp, let e1 and e2 be the C# equivalent of exp.Left and exp.Right. Then the C# equivalent of exp is "(e1) - (e2)".

Use this in a DynamicExpression to represent a binary operator for asking the first object to add the second object to itself. Neither object is modified.

4.4.45 SubtrtactChecked

Use SubtractChecked in a BinaryExpression node to represent arithmetic subtraction with overflow checking. Given a SubtractChecked node, exp, let e1 and e2 be the C# equivalent of exp.Left and exp.Right. Then the C# equivalent of exp is "checked(unchecked(e1) - unchecked(e2))".

There is no use for this node kind in DynamicExpressions.

4.4.46 TypeAs

Use TypeAs in a UnaryExpression node to represent an explicit reference or boxing conversion supplying the null value if the conversion fails. Given a TypeAs node, exp, let e be the C# equivalent of exp.Operand, and let T be a type expression in C# for the type represented by exp.Type. If T is a reference type or nullable type, the C# equivalent of exp is "(e) as T".

There is no use for this in DynamicExpressions. They have a Convert protocol, but due to how dynamic operations work, the invoking language has control over whether a null or an exception comes from the conversation in exceptional situations.

4.4.47 TypeIs

TypeBinaryExpressions usesTypeIs to represent a sub type test. The semantics is the same as the CLR's IsInst instruction, which is very close to the C# 'is' operator.

4.4.48 TypeEqual

TypeBinaryExpressions uses TypeEqual to represent testing whether an object exactly has the specified type. The C# similar expression would be "x.GetType() == T".

4.4.49 Assign

Use Assign in BianryExpression nodes to represent evaluating the right expression and assigning it to a location indicated by the left expression. The left expression must be one of the node types ParameterExpression, MemberExpression, or IndexExpression. The result is the value of the right expression, but it has the type of the left expression.

The order of evaluation is first any sub expressions in the left expression followed by the right expression. For example, if the left expression is “e1.foo” or “e1[e2, e3]”, then e1 is evaluated first in both cases, then e2 and e3 in the second case. The root semantics of the left expression (member fetch, index, variable fetch) are not evaluated but instead used to emit instructions for storing into the specified location.

There is no use for Assign with DynamicExpressions since the left would always be evaluated to produce an object. The DynamicMetaObject protocol has operations for setting members and indexed locations of dynamic objects.

4.4.50 Block

The BlockExpression uses the Block node kind. It represents a sequence of expressions. The semantics are to execute each expression in order, squelching the result of each except the last expression. The value of the Block node is the result of the last expression in the body.

Each block also has a collection, possibly empty, of variables whose scope is the body of the block. Ignoring closures, the variables' lifetimes are limited to the block. There is no initialization of the variables beyond any guarantees of .NET. If you care about definite assignment or order of initialization, then the first expressions in the block should assign the variables. Your compiler should emit any errors in your language should the program not adhere to your language's semantics in these regards.

Some languages have strong lexical semantics for unique binding of variables. For example, in Common Lisp or Scheme, each iteration around a loop or any basic block of code creates unique bindings for variables introduced in those basic blocks. Thus, returning a lambda would create unique closures environments for each iteration. Some languages, such as Python and F#, move all variables to the scope of their containing function. ETs v2 supports both models, depending on where you create the BlockExpression in the ET and list variables.

For the stronger lexical model, for example with the loop, place the BlockExpression inside the loop body and list variables there. Putting the Block Expression outside the loop or at the start of the function ensures the variables have one binding across iterations of the loop. These effects are usually only observable when closing over lexical environments, and ETs v2 guarantees the unique binding semantics per loop iteration in this case. If you do not close over

any of the Block's variables, and you do not explicitly assign to them before referencing them, then the behavior or observability of unique bindings is undefined. See section 4.22.1.3 for an example.

Post CLR 4.0 (remove IsByRef error check): If any of the ParameterExpressions representing the local variables has IsByRef set to True, then you must initialize them with an AssignRef BinaryExpression node. Otherwise, if the Body of the Block references the variable, or the variable appears as the Left expression of an Assign BinaryExpression node, there will be compile time error (or possibly an error when the code executes).

There is no use for this node kind in DynamicExpressions themselves.

4.4.51 DebugInfo

DebugInfoExpression use this node kind. It represents a point in an ET where there is debugging information (a la .NET sequence points). To clear the sequence point information, use a DebugInfo node with IsClear set to True.

4.4.52 Decrement

Use Decrement in UnaryExpression nodes to represent functional decrement of the operand expression by one unit. The operand should not be modified by the operation. The methodinfo represents the implementing method for subtracting one unit from the operand. If the methodinfo is null, and the operand is numeric, the semantics is to subtract one.

In DynamicExpressions the expected semantics are to ask the object to return a unit decremented value of itself. The object is not modified.

4.4.53 Dynamic

The DynamicExpression node uses this node kind. It represents an operation that must be bound at runtime of the expression tree. The semantics is to create a DLR CallSite for caching of implementations of the operation given the different kinds of objects passed to the CallSite during the program's execution. The Dynamic node has a DLR CallSiteBinder that determines the exact semantics of the operation given the runtime operands. The binder encapsulates the language semantics for the creator of the node, as well as any payload meta data that informs the binder how to compute an implementation of the operation at runtime.

For more on expected semantics, see other node kinds' statements as to their semantics in DynamicExpression nodes. You might also see the sub classes of DynamicMetaDataBinder and language documentation on their dynamic semantics. See the documents at on the DLR Codeplex site, sites-binders-dynobj-interop.doc and library-authors-introduction.doc.

4.4.54 Default

DefaultExpressions use this node kind. If DefaultExpression.Type represents the void type, the node represents a no-op empty expression. If Type is some other type, then node represents a constant returning default(T) for the type.

4.4.55 Extension

Use the Extension node kind for any node type that is a user-derived type of Expression. You should not use the common node kinds for

4.4.56 Goto

GotoExpressions use this node kind. It represents an unstructured flow of control to a labeled location in the ET. The Goto refers to a LabelTarget that a LabelExpression must refer to somewhere in the ET, and it is the LabelExpression that sets the target location for the flow of control. The LabelExpression must be lexically in the same LambdaExpression.Body as the GotoExpression. There is no non-local exit support in ETs v2.

The semantics of the LabelTarget chosen as the destination is lexically scoped in a sense. If all LabelTargets in the LambdaExpression are unique, then the LabelTarget in the GotoExpression simply must be found within the LambdaExpression containing the Goto. If the same LabelTarget is used multiple times within a LambdaExpression, then the GotoExpression targets the first matching LabelTarget found while searching up the ET to the Lambda root. This is a convenience for re-writers or tree builders that re-use sub trees that contain LabelExpressions and GotoExpressions so that the sub trees behave as expected unto themselves.

The Goto can optionally deliver a value to the location, as expressed by a non-null Expression property. If this property is non-null, then the expression Type property must represent a type that is reference assignable to the type represented by Target.Type. However, if Target.Type is void, the GotoExpression.Expression.Type can represent anything since the ET compiler will automatically convert the result to void or squelch the value.

We limit Goto lexically within a function. In addition to the simple ability to jump from inner basic blocks to outer basic blocks, we allow jumping into the following:

- BlockExpressions
- ConditionalExpressions
- LoopExpressions
- SwitchExpressions
- TryExpression body from a contained CatchBlock

We don't support jumping from the outside into the following possibly surprising basic blocks (you can't jump into various other expressions such as arguments, which probably aren't surprising):

- TryExpressions
- CatchBlocks
- Finally blocks

We don't support jumping from (that is, leaving) the following:

- finally blocks
- LambdaExpressions
- GeneratorExpressions

~~CUT Beta2 BUG SEMANTICS: ETs v2 limit Goto lexically within a function. ETs allow jumping into and out of the following:~~

- ~~BlockExpressions~~

- ~~ConditionalExpressions~~
- ~~LoopExpressions~~
- ~~SwitchExpressions~~
- ~~TryExpressions (under certain situations)~~
- ~~LabelExpressions~~

~~ETs allow some jumps relative to TryExpressions:~~

- ~~jumping out of TryExpression's body~~
- ~~jumping out of a CatchBlock's body~~
- ~~jumping into TryExpression's body from one of its own CatchBlocks~~

~~The above constitutes what ETs v2 allow. Just by way of examples, we do not allow jumping into the middle of argument expressions, such as those in binary operands, method calls, invocations, indexing, instance creation, etc. We do not allow jumping into or out of GeneratorExpressions. You could however jump within a BlockExpression used as an argument expression.~~

4.4.57 Increment

Use Increment in UnaryExpression nodes to represent functional increment of the operand expression by one unit. The operand should not be modified by the operation. The methodinfo represents the implementing method for adding one unit to the operand. If the methodinfo is null, and the operand is numeric, the semantics is to add one.

In DynamicExpressions the expected semantics are to ask the object to return a unit incremented value of itself. The object is not modified.

4.4.58 Index

IndexExpressions use this node kind. It represents an indexing operation or a property invocation that takes arguments.

4.4.59 Label

LabelExpressions use this node kind. It represents an unstructured flow of control target location.

A LabelTarget object identifies the Label's location. GotoExpressions refer to the same target object to designate that they jump to this LabelExpression location in the ET.

The target has a Type property because Goto's can transfer control to a location with a value. The type allows factory methods and the ET compiler to verify static typing intent within the ET. See section 2.4 for more information.

A Goto can optionally deliver a value to the LabelExpression's location. In case execution flows through the label in a structured way (not via a jump), it has a DefaltValue expression that provides the result of the LabelExpression. The label's location is AFTER the DefaultValue expression. If an unstructured flow of control lands at this LabelExpression's location, the GotoExpression provides the result for the LabelExpression.

The LabelExpression.Type property is the same value as its Target.Type. Any GotoExpression that references the same Target must have a Type property that represents a type that is

reference assignable to the type represented by Target.Type. There are two exceptions, see the factory method documentation for an explanation.

4.4.60 RuntimeVariables

RuntimeVariablesExpression uses this node kind. It represents an expression that returns an IRuntimeVariables for a list of provided ParameterExpressions. It effectively lifts these variables to a closure for getting/setting them. Languages with constructs for accessing lexical variables can use this expression to access these closure variables.

4.4.61 Loop

LoopExpression uses this node kind. It represents an expression that executes its body expression infinitely until a sub expression of the body exits the loop via a GotoExpression. Loop nodes have a Break property with a LabelTarget. When control transfers to this label with a value, the value becomes the result of the LoopExpression. The Break label can be null, in which case the Loop's Type property represents the void type.

4.4.62 Switch

SwitchExpression uses this node kind. At a high level, this node's semantics is to evaluate the SwitchValue expression, then to evaluate each SwitchCase's TestValues in order. For each test value, if the SwitchCase.Comparison (invoked on the SwitchValue and TestValue) return True, then the corresponding SwitchCAse.Body executes. Each case is considered in the order it appears in the node. If no case fires, then the DefaultBody executes. The value resulting from the SwitchExpression is the last expression executed, which is typically the last expression of the selected case body.

If you want the effect of case fall through, then you can use GotoExpression and construct the target case as follows. The case's body can be a BlockExpression, and the first expression in it can be a LabelExpression with a null Expression property. The expression compiler detects patterns for eliminating the goto's.

If DefaultBody is null, the Type property must represent void type.

The Comparison MethodInfo must result in a Boolean value. If Comparison is null, then the test is performed as it would be with an Equal node. The comparison function gets invoked with the switch value as the first argument and each case's test value in turn as the second argument. Each test value is considered in the order it appears in the case.

The TestValue must have an integer type for the compilation of a Switch node to use .NET's IL level switch. If the SwitchValue type is a string, and all cases are string constants, the node compiles into a dictionary lookup followed by an IL level switch, as an optimization.

All test values in all cases must have exactly the same type.

All the case body expressions must have the same Type property as the Switch's Type property. There is one exception. If the Switch has a void type, then the case bodies can be any type, and the semantics is to automatically convert to void or squelch any result value.

As a sidenote, you can construct an if-then-else or Lisp-style 'cond' construct. If SwitchValue is True, then the case test values can be any expression that returns True. It is not an exact match

semantically given various semantics regarding what is a non-false value, but the effect is very close.

The Type property must match every case's body's type, unless the Type property is void. If it is void, then the case bodies can be of any type, and any results are automatically "converted to void" or squelched.

4.4.63 Throw

Use this node kind in UnaryExpression nodes to represent a dynamic flow of control to a matching catch block. These semantics are equivalent to C#'s throw with the following exception: you can use Throw expressions in Filter block since IL supports this.

4.4.64 Try

TryExpression uses this node kind. It represents try-catch control flow, including finally and fault blocks. First the Body executes. Under normal execution, control flows from the body to the Finally expression, if any. The value of the TryExpresson is the Body expression's value. If an object is thrown (directly or indirectly) from within the Body, and there's no dynamically intervening appropriate catch handler, the CatchBlock' s are considered in order to find one with a Test type that is assignable from the thrown object. If there is such a CatchBlock, execution flows to its Body and then to the Finally expression, if any. The CatchBlock's body produces the value for the TryExpression in this case.

If non-null, the Finally expression always executes regardless of dynamic flow of control to or through the TryExpression.

If Fault is non-null, then Finally is null, and Handlers is empty. The Fault expresson executes if an object is thrown from the Body, directly or indirectly. Control flows from the Fault expression to some appropriate catch handler, or possibly an process unhandled throw error block. If execution flows from the Body expression to the Finally expression, if any, and no objects are thrown, then the Fault expression does NOT execute.

Try is legal in some places where it would not be allowed in IL, such as when the stack is non-empty. For example: Call(e0, e1, try { … } catch { …}, e3)

4.4.65 Unbox

Use the Unbox node kind in UnaryExpression nodes to represent an explicit unboxing operation. The semantics is to create an interior pointer to the boxed value that is tagged with the type represented by the Unbox node's Type property. The Unbox node's Type property represents the boxed value's value type. The operand expression's Type property must represent an interface type or type Object; a value type would not be boxed for any reason other than to be passed or assigned to an interface type or type Object. If the operand's Type property does not represent the type Object, then the Unbox node's Type property must represent a type that implements the operand's interface type.

This node maps to the IL unbox and unbox.any instructions.

Design Rationale ...

We will need this in v-next+1 if we look toward IL coverage in the ET model, but we added Unbox in .NET 4.0 to support DLR languages, such as IronPython. These languages want to present a programming model as exemplified here (without the commented out line):

```
r = Rect()
## s = StrongBox[Rect](r)
r.Intersect(r2)
```

The problem is that a dynamic language like IronPython has to box r to have actual pointers to objects. When the Intersect call happens, the CLR box gets unpacked, and the destructive Intersect call modifies a copy of r's contents. If IronPython programmers explicitly StrongBox the rect, as in the commented out line, then the IronPython programmer gets the expected behavior of modifying r's contents. This is not how dynamic language programmers want to think about using values like rects, and for the language to manage this boxing and unboxing correctly everywhere would be a lot of work to get right.

Note, the C# code does the right thing:

```
Rect r = new Rect()
//Expr.Call(Expr.Convert(r, typeof(Rect)),
//                       "Intersect", new[] {r2})
Expr<Funct<>> e = () => ((Rect) r).Intersect(r2)
```

C# emits IL to pass the address to the r value in the CLR box object, so it works as expected. We've simply enabled dynamic languages using the DLR to do the same implementation.

4.4.66 AddAssign

Use this AddAssign node kind in BinaryExpression nodes to represent an Add compound assignment operation, without overflow checking. The Left expression must be one of the node types: ParameterExpression, MemberExpression, or IndexExpression. The result of the AddAssign node is the result of performing the Add opreration on the operands.

The node's methodinfo performs only the basic binary operation (Add, Subtract, etc.). The resulting value is stored in the location represented by Left. If methodinfo is null, and the Left.Type and Right.Type properties represent the same numeric type, the node has the semantics of IL addition. Otherwise, the node searches for and applies a user-defined op_Addition method.

If the node's conversion lambda is non-null, then the semantics is to pass the result of the basic binary operation to the lambda. The result of the conversion lambda is then stored in the Left location.

Use this in a DynamicExpression to represent a binary operator for asking the first object to add the second object to itself, modifying the first object's value in place.

4.4.67 AddAssignChecked

Use the AddAssignChecked node kind in BinaryExpression nodes to represent an Add compound assignment operation, with overflow checking. The Left expression must be one of the node types: ParameterExpression, MemberExpression, or IndexExpression.

The node's methodinfo performs only the basic binary operation (Add, Subtract, etc.). The resulting value is stored in the location represented by Left. If methodinfo is null, and the Left.Type and Right.Type properties represent the same numeric type, the node has the

semantics of IL addition. Otherwise, the node searches for and applies a user-defined op_Addition method.

If the node's conversion lambda is non-null, then the semantics is to pass the result of the basic binary operation to the lambda. The result of the conversion lambda is then stored in the Left location.

There is no use for this in a DynamicExpression.

4.4.68 DivideAssign

Use the DivideAssign node kind in BinaryExpression nodes to represent a Divide compound assignment operation, with overflow checking. The Left expression must be one of the node types: ParameterExpression, MemberExpression, or IndexExpression. The result of the DivideAssign node is the result of performing the Divide opreration on the operands.

The node's methodinfo performs only the basic binary operation (Add, Subtract, etc.). The resulting value is stored in the location represented by Left. If methodinfo is null, and the Left.Type and Right.Type properties represent the same numeric type, the node has the semantics of IL div. Otherwise, the node searches for and applies a user-defined op_Division method.

If the node's conversion lambda is non-null, then the semantics is to pass the result of the basic binary operation to the lambda. The result of the conversion lambda is then stored in the Left location.

Use this in a DynamicExpression to represent a binary operator for asking the first object to divide itself by the second object, modifying the first object's value in place.

4.4.69 ExclusiveOrAssign

Use the ExclusiveOrAssign node kind in BinaryExpression nodes to represent an ExclusiveOr compound assignment operation. The Left expression must be one of the node types: ParameterExpression, MemberExpression, or IndexExpression. The result of the ExclusiveOrAssign node is the result of performing the ExclusiveOr opreration on the operands.

The node's methodinfo performs only the basic binary operation (Add, Subtract, etc.). The resulting value is stored in the location represented by Left. If methodinfo is null, and the Left.Type and Right.Type properties represent the same integer or boolean type, the node has the semantics of IL xor. Otherwise, the node searches for and applies a user-defined op_ExclusiveOr method.

If the node's conversion lambda is non-null, then the semantics is to pass the result of the basic binary operation to the lambda. The result of the conversion lambda is then stored in the Left location.

Use this in a DynamicExpression to represent a binary operator for asking the first object to ExlusiveOr the second object to itself, modifying the first object's value in place.

4.4.70 LeftShiftAssign

Use the LeftShiftAssign node kind in BinaryExpression nodes to represent a LeftShift compound assignment operation. The Left expression must be one of the node types:

ParameterExpression, MemberExpression, or IndexExpression. The result of the LeftShiftAssign node is the result of performing the LeftShift opreration on the operands.

The node's methodinfo performs only the basic binary operation (Add, Subtract, etc.). The resulting value is stored in the location represented by Left. If methodinfo is null, and the Left.Type and Right.Type properties represent the type integer, the node has the semantics of IL LeftShift. Otherwise, the node searches for and applies a user-defined op_LeftShift method.

If the node's conversion lambda is non-null, then the semantics is to pass the result of the basic binary operation to the lambda. The result of the conversion lambda is then stored in the Left location.

Use this in a DynamicExpression to represent a binary operator for asking the first object to shift contents (most likely bits) left the number of times representd by the second object, modifying the first object's value in place.

4.4.71 ModuloAssign

Use the ModuloAssign node kind in BinaryExpression nodes to represent an Modulo compound assignment operation. The Left expression must be one of the node types: ParameterExpression, MemberExpression, or IndexExpression. The result of the ModuloAssign node is the result of performing the Modulo opreration on the operands.

The node's methodinfo performs only the basic binary operation (Add, Subtract, etc.). The resulting value is stored in the location represented by Left. If methodinfo is null, and the Left.Type and Right.Type properties represent the same numeric type, the node has the semantics of IL rem. For example, -10 mod 3 is -1, and 10 mod -3 is 1. Otherwise, the node searches for and applies a user-defined op_Modulus method.

If the node's conversion lambda is non-null, then the semantics is to pass the result of the basic binary operation to the lambda. The result of the conversion lambda is then stored in the Left location.

Use this node kind in Dynamic Expression to ask the first object to divide itself by the second object. If the objects are numbers, the expectation is that the result is an integer remainder resulting from TruncateDivide'ing the first object by the second. The first object is modified in place.

4.4.72 MultiplyAssign

Use the MultipleAssign node kind in BinaryExpression nodes to represent an Multiply compound assignment operation, without overflow checking. The Left expression must be one of the node types: ParameterExpression, MemberExpression, or IndexExpression. The result of the MultiplyAssign node is the result of performing the Multiply opreration on the operands.

The node's methodinfo performs only the basic binary operation (Add, Subtract, etc.). The resulting value is stored in the location represented by Left. If methodinfo is null, and the Left.Type and Right.Type properties represent the same numeric type, the node has the semantics of IL multiplication. Otherwise, the node searches for and applies a user-defined op_Multiply method.

If the node's conversion lambda is non-null, then the semantics is to pass the result of the basic binary operation to the lambda. The result of the conversion lambda is then stored in the Left location.

Use this in a DynamicExpression to represent a binary operator for asking the first object to multiply the second object to itself, modifying the first object's value in place. If the objects are numbers, the expectation is that the result is a number.

4.4.73 MulitplyAssignChecked

Use the MultiplyAssignChecked node kind in BinaryExpression nodes to represent an Multiply compound assignment operation, with overflow checking. The Left expression must be one of the node types: ParameterExpression, MemberExpression, or IndexExpression. The result of the MultiplyAssign node is the result of performing the Multiply opreration on the operands.

The node's methodinfo performs only the basic binary operation (Add, Subtract, etc.). The resulting value is stored in the location represented by Left. If methodinfo is null, and the Left.Type and Right.Type properties represent the same numeric type, the node has the semantics of IL multiplication. Otherwise, the node searches for and applies a user-defined op_Multiply method.

If the node's conversion lambda is non-null, then the semantics is to pass the result of the basic binary operation to the lambda. The result of the conversion lambda is then stored in the Left location.

There is no use for this node kind a DynamicExpression.

4.4.74 OrAssign

Use the OrAssign node kind in BinaryExpression nodes to represent an Or compound assignment operation. The Left expression must be one of the node types: ParameterExpression, MemberExpression, or IndexExpression. The result of the OrAssign node is the result of performing the Or opreration on the operands.

The node's methodinfo performs only the basic binary operation (Add, Subtract, etc.). The resulting value is stored in the location represented by Left. If methodinfo is null, and the Left.Type and Right.Type properties represent the same integer or boolean type, the node has the semantics of IL or. Otherwise, the node searches for and applies a user-defined op_BitwiseOr method.

If the node's conversion lambda is non-null, then the semantics is to pass the result of the basic binary operation to the lambda. The result of the conversion lambda is then stored in the Left location.

Use this in a DynamicExpression to represent a binary operator for asking the first object to Or the second object to itself, modifying the first object's value in place.

4.4.75 PowerAssign

Use the PowerAssign node kind in BinaryExpression nodes to represent an exponentiation compound assignment operation. The Left expression must be one of the node types: ParameterExpression, MemberExpression, or IndexExpression. The result of the PowerAssign node is the result of performing the exponentiation opreration on the operands.

The node's methodinfo performs only the basic binary operation (Add, Subtract, etc.). The resulting value is stored in the location represented by Left. If methodinfo is null, the semantics is to invoke the System.Math.Pow function on the operands.

If the node's conversion lambda is non-null, then the semantics is to pass the result of the basic binary operation to the lambda. The result of the conversion lambda is then stored in the Left location.

Use this in DynamicExpressions to request that the first object raise itself to the power of the second object. For numeric objects, you should expect a numeric result, ignoring NaN and overflow issues. Otherwise, the semantics is whatever the object implements. Neither object should be modified.

4.4.76 RightShiftAssign

Use the RightShfitAssign node kind in BinaryExpression nodes to represent a RightShift compound assignment operation. The Left expression must be one of the node types: ParameterExpression, MemberExpression, or IndexExpression. The result of the RightShiftAssign node is the result of performing the RightShift opreration on the operands.

The node's methodinfo performs only the basic binary operation (Add, Subtract, etc.). The resulting value is stored in the location represented by Left. If methodinfo is null, and the Left.Type and Right.Type properties represent the type integer, the node has the semantics of IL RightShift. Otherwise, the node searches for and applies a user-defined op_RightShift method.

If the node's conversion lambda is non-null, then the semantics is to pass the result of the basic binary operation to the lambda. The result of the conversion lambda is then stored in the Left location.

Use this in a DynamicExpression to represent a binary operator for asking the first object to shift contents (most likely bits) right the number of times representd by the second object, modifying the first object's value in place.

4.4.77 SubtractAssign

Use SubtractAssign node kind in BinaryExpression nodes to represent an Subtract compound assignment operation, without overflow checking. The Left expression must be one of the node types: ParameterExpression, MemberExpression, or IndexExpression. The result of the SubtractAssign node is the result of performing the Subtract opreration on the operands.

The node's methodinfo performs only the basic binary operation (Add, Subtract, etc.). The resulting value is stored in the location represented by Left. If methodinfo is null, and the Left.Type and Right.Type properties represent the same numeric type, the node has the semantics of IL subtraction. Otherwise, the node searches for and applies a user-defined op_Subtraction method.

If the node's conversion lambda is non-null, then the semantics is to pass the result of the basic binary operation to the lambda. The result of the conversion lambda is then stored in the Left location.

Use this in a DynamicExpression to represent a binary operator for asking the first object to subtract the second object from itself, modifying the first object's value in place.

4.4.78 SubtractAssignChecked

Use SubtractAssignCheced node kind in BinaryExpression nodes to represent an Subtract compound assignment operation, with overflow checking. The Left expression must be one of the node types: ParameterExpression, MemberExpression, or IndexExpression. The result of the SubtractAssign node is the result of performing the Subtract opreration on the operands.

The node's methodinfo performs only the basic binary operation (Add, Subtract, etc.). The resulting value is stored in the location represented by Left. If methodinfo is null, and the Left.Type and Right.Type properties represent the same numeric type, the node has the semantics of IL subtraction. Otherwise, the node searches for and applies a user-defined op_Subtraction method.

If the node's conversion lambda is non-null, then the semantics is to pass the result of the basic binary operation to the lambda. The result of the conversion lambda is then stored in the Left location.

There is no use for this node kind a DynamicExpression.

4.4.79 PreIncrementAssign

Use the PreIncrementAssign node kind in UnaryExpression nodes to represent incrementing a value in-place by one unit. The result is stored to the operand's location. The operand expression must be one of the node types: ParameterExpression, MemberExpression, or IndexExpression. The result of the PreIncrementAssign node is the result of performing the increment opreration on the operand.

The node's methodinfo performs only the basic unary operation (Add, Subtract, etc.). If methodinfo is null, and the operand's Type property represents a numeric type, the node has the semantics of IL addition of one, with the result stored in the location indicated by the operand. Otherwise, the node searches for and applies a user-defined op_Increment method.

There is no use for this in a DynamicExpression.

4.4.80 PreDecrementAssign

Use the PreDecrementAssign node kind in UnaryExpression nodes to represent decrementing a value in-place by one unit. The result is stored to the operand's location. The operand expression must be one of the node types: ParameterExpression, MemberExpression, or IndexExpression. The result of the PreDecrementAssign node is the result of performing the decrement opreration on the operand.

The node's methodinfo performs only the basic unary operation (Add, Subtract, etc.). If methodinfo is null, and the operand's Type property represents a numeric type, the node has the semantics of IL subtraction of one, with the result stored in the location indicated by the operand. Otherwise, the node searches for and applies a user-defined op_Decrement method.

There is no use for this in a DynamicExpression.

4.4.81 PostIncrementAssign

Use the PostIncrementAssign node kind in UnaryExpression nodes to represent incrementing a value in-place by one unit. The result is stored to the operand's location. The operand

expression must be one of the node types: ParameterExpression, MemberExpression, or IndexExpression. The result of the PostIncrementAssign node is the the operand's value before performing the operation.

The node's methodinfo performs only the basic unary operation (Add, Subtract, etc.). If methodinfo is null, and the operand's Type property represents a numeric type, the node has the semantics of IL addition of one, with the result stored in the location indicated by the operand. Otherwise, the node searches for and applies a user-defined op_Increment method.

There is no use for this in a DynamicExpression.

4.4.82 PostDecrementAssign

Use the PreDecrementAssign node kind in UnaryExpression nodes to represent decrementing a value in-place by one unit. The result is stored to the operand's location. The operand expression must be one of the node types: ParameterExpression, MemberExpression, or IndexExpression. The result of the PreDecrementAssign node is the result of the operand before performing the operation.

The node's methodinfo performs only the basic unary operation (Add, Subtract, etc.). If methodinfo is null, and the operand's Type property represents a numeric type, the node has the semantics of IL subtraction of one, with the result stored in the location indicated by the operand. Otherwise, the node searches for and applies a user-defined op_Decrement method.

There is no use for this in a DynamicExpression.

4.4.83 OnesComplement

Use OnesComplement in UnaryExpression nodes to represent bitwise negation.

For legacy purposes, the Not node kind represents bitwise and logical negation. The OneComplement node kind makes DLR interoperability intent more clear in the DynamicMetaObject protocol, and for meta-programming scenarios with the static nodes.

Use OnesComplement in a DynamicExpression to ask an object to returns its bitwise negation. Use Not in DynamicExpressions to ask an object to return its logical negation.

4.4.84 IsTrue

Use the IsTrue node kind in UnaryExpression nodes to represent an expression whose value is true if the argument expression represents a true value. The semantics is the same as an expression usable in C# statements such as 'if', 'while', 'for', etc.: if the argument value is not implicitly convertible to bool, then it must implement the op_true operator method.

Use this operator in DynamicExpressions (actually need two) to create ETs for conditional-or. For example, you could have a DynamicExpression using IsTrue to test if a left operand is true. If it is, return the operand value, but if it is not IsTrue, use another DynamicExpression to Or the left with the right operand to produce the result.

4.4.85 IsFalse

Use the IsFalse node kind in UnaryExpression nodes to represent an expression whose value is true if the argument expression represents a false value. The semantics is the same as an

expression usable in C# statements such as 'if', 'while', 'for', etc.: if the argument value is not implicitly convertible to bool, then it must implement the op_false operator method.

Use this operator in DynamicExpressions (actually need two) to create ETs for conditional-and. For example, you could have a DynamicExpression using IsFalse to test if a left operand is false. If it is, return the operand value, but if it is not IsFalse, use another DynamicExpression to And the left with the right operand to produce the result.

4.4.86 AssignRef (POST CLR 4.0)

Use AssignRef in BinaryExpressions to represent assigning an indirect reference to the location represented by the Right Expression. The Right expression must be one of the node types ParameterExpression, MemberExpression, or IndexExpression. The Left Expression must be a ParameterExpression with IsByRef set to True.

The order of evaluation is first any sub expressions in the Right expression. For example, if the Right expression is “e1.foo” or “e1[e2, e3]”, then e1 is evaluated first in both cases, then e2 and e3 in the second case. The root semantics of the Right expression (member fetch, index, variable fetch) are not evaluated but instead used to emit instructions for creating a ByRef local variable that refers to the storage location.

After an AssignRef initializes a variable represented by ParameterExpression, then the ParameterExpression can be used as the Left Expression in an Assign node. The semantics is to store the value resulting from the Assign node's Right expression to the location referenced by the IsByRef ParameterExpression.

There is no use for AssignRef with DynamicExpressions.

Design Rationale ...
We considered this in ETs v2 because we know we would need it post CLR 4.0 for IL completeness, and we could make use of it in rewriting ETs for compilation purposes. For example, to allow nesting of gotos/try-catch's, you could rewrite this:

```
class Point {
    public double X, Y;
}
Point Bar() { … }
void Foo(ref double x, …) { … }
void Main() {
    // NOTE: not currently legal in expression trees
    // because it combines “ref” with a nested try statement
    Foo(ref Bar().X, try { … } finally { … }, …);
```

into this:

```
// Using AssignRef this is legal, and preserves argument evaluation
order
double& arg0 = Bar().X;
arg2 = try { … } finally { … };
…
argN = …
Foo(arg0, arg1, …, argN);
```

Having AssignRef is not required to make this work, but it makes the ET transformation simpler. You need the temporary variables though since IL won't allow entering the try block with values on the IL stack.

4.5 DefaultExpression Class

This class represents empty expressions and default values of types. It has two factories. One takes no arguments and returns an empty or no-op Expression whose type is Void. The other factory takes a type, and the expression compiles to a constant returning default(T) for the type.

These are useful for filling in required expressions that simply complete other expressions, such as the alternative expression of an 'if' or the last expression of a block whose type is Void.

4.5.1 Class Summary

```
public sealed class DefaultExpression : Expression {}
```

4.5.2 Factory Methods

Expression has the following factory methods for DefaultExpressions:

```
    public static DefaultExpression Default(Type type);
    public static DefaultExpression Empty();
```

If type is void, Default returns the same node that calling Empty returns.

4.6 BinaryExpression Class

This class represents a many kinds of operations such as basic arithmetic, bit manipulations, logical comparisons, assignment, and so on. With only a couple of exceptions, all the operations fit the shape of having left and right operands as their primary inputs.

The exceptions to the shape design are Coalesce and ...Assign node kinds. They take an optional conversion lambda you can specify that executes last and converts the node's intermediate result value to the node's Type or the Left.Type. This conversion aspect is required in the node's semantics so that languages or ET producers can specify the exact overloaded conversion method or a custom method.

Operations on numeric types do not implicitly expand to 32bit integers.

These nodes have many node kinds for the various operations they can support. See section 4.4 for more details on the semantics of various node kinds used with BinaryExpressions.

4.6.1 Class Summary

```
public class BinaryExpression : Expression {
    public LambdaExpression Conversion { get; }
    public Boolean IsLifted { get; }
    public Boolean IsLiftedToNull { get; }
    public Expression Left { get; }
    public MethodInfo Method { get; }
    public Expression Right { get; }
    public BinaryExpression Update
```

```
(Expression left, LambdaExpression conversion, Expression right)
```

4.6.2 Conversion Property

This property returns the expression that models the type conversion function used by a Coalesce node kind, and by OPAssign nodes where OP is Add, Multiply, etc. If the node kind is not Coalese or one of the OPAssign, this returns null.

Signature:

```
public LambdaExpression Conversion { get; }
```

4.6.3 IsLifted Property

This property returns true if the node represents an operator call that takes non-nullable parameters, but the calls passes nullable arguments.

Signature:

```
public Boolean IsLifted { get; }
```

4.6.4 IsLiftedToNull Property

This property returns true if the node represents a call to an operator that returns a nullable type. If a nullable argument evaluates to null (Nothing in Visual Basic), the operator returns a null reference (Nothing in Visual Basic).

Signature:

```
public Boolean IsLiftedToNull { get; }
```

4.6.5 Method Property

This property returns the MethodInfo associated the operation to further specify its semantics. The value may be null if the operation represents a CLI predefined operator.

Signature:

```
public MethodInfo Method { get; }
```

4.6.6 Left Property

This property returns the expression for the first argument to the operation.

Signature:

```
public Expression Left { get; }
```

4.6.7 Right Property

This property returns the expression for the second argument to the operation.

Signature:

```
public Expression Right { get; }
```

4.6.8 Update Method

This method creates a new expression that is like this one, but using the supplied children. If all of the children are the same, this method returns this expression instead of creating a new one. This is useful when implementing ExpressionVisitors to rewrite the current node only when you need to do so.

Signature:

```
public BinaryExpression Update
   (Expression left, LambdaExpression conversion, Expression right)
```

4.6.9 Arithmetic, Shift, and Bit Operations Factory Methods

Expression has the following factory methods for BinaryExpressions representing arithemetic, shift, and bit operations:

```
public static BinaryExpression Add
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression Add(Expression left,
                                   Expression right);
public static BinaryExpression AddAssign
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression AddAssign(Expression left,
                                         Expression right);
public static BinaryExpression AddAssign
    (Expression left, Expression right, MethodInfo method,
     LambdaExpression conversion)
public static BinaryExpression AddAssignChecked(Expression left,
                                                Expression right);
public static BinaryExpression AddAssignChecked
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression AddAssignChecked
    (Expression left, Expression right, MethodInfo method,
     LambdaExpression conversion)
public static BinaryExpression AddChecked
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression AddChecked(Expression left,
                                          Expression right);

public static BinaryExpression And(Expression left,
                                   Expression right);
public static BinaryExpression And
    (Expression left, Expression right, MethodInfo method);

public static BinaryExpression AndAssign(Expression left,
                                         Expression right);
public static BinaryExpression AndAssign
    (Expression left, Expression right, MethodInfo method);

public static BinaryExpression Divide(Expression left,
                                      Expression right);
public static BinaryExpression Divide
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression DivideAssign(Expression left,
                                            Expression right);
public static BinaryExpression DivideAssign
```

```
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression DivideAssign
    (Expression left, Expression right, MethodInfo method,
     LambdaExpression conversion)

public static BinaryExpression ExclusiveOr
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression ExclusiveOr(Expression left,
                                           Expression right);
public static BinaryExpression ExclusiveOrAssign
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression ExclusiveOrAssign
    (Expression left, Expression right);
public static BinaryExpression ExclusiveOrAssign
    (Expression left, Expression right, MethodInfo method,
     LambdaExpression conversion)

public static BinaryExpression LeftShift
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression LeftShift(Expression left,
                                         Expression right);
public static BinaryExpression LeftShiftAssign(Expression left,
                                               Expression right);
public static BinaryExpression LeftShiftAssign
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression LeftShiftAssign
    (Expression left, Expression right, MethodInfo method,
     LambdaExpression conversion)

public static BinaryExpression Modulo
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression Modulo(Expression left,
                                      Expression right);
public static BinaryExpression ModuloAssign(Expression left,
                                            Expression right);
public static BinaryExpression ModuloAssign
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression ModuloAssign
    (Expression left, Expression right, MethodInfo method,
     LambdaExpression conversion)

public static BinaryExpression Multiply(Expression left,
                                        Expression right);
public static BinaryExpression Multiply
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression MultiplyAssign
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression MultiplyAssign
    (Expression left, Expression right, MethodInfo method,
     LambdaExpression conversion)
public static BinaryExpression MultiplyAssign
    (Expression left, Expression right);
public static BinaryExpression MultiplyAssignChecked
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression MultiplyAssignChecked
    (Expression left, Expression right);
public static BinaryExpression MultiplyChecked
```

```
    (Expression left, Expression right);
public static BinaryExpression MultiplyChecked
    (Expression left, Expression right, MethodInfo method);

public static BinaryExpression Or
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression Or(Expression left,
                                  Expression right);
public static BinaryExpression OrAssign(Expression left,
                                        Expression right);
public static BinaryExpression OrAssign
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression OrAssign
    (Expression left, Expression right, MethodInfo method,
     LambdaExpression conversion)

public static BinaryExpression Power
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression Power(Expression left,
                                     Expression right);
public static BinaryExpression `ign(Expression left,
                                    Expression right);
public static BinaryExpression PowerAssign
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression PowerAssign
    (Expression left, Expression right, MethodInfo method,
     LambdaExpression conversion)

public static BinaryExpression RightShift
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression RightShift(Expression left,
                                          Expression right);
public static BinaryExpression RightShiftAssign(Expression left,
                                                Expression right);
public static BinaryExpression RightShiftAssign
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression RightShiftAssign
    (Expression left, Expression right, MethodInfo method,
     LambdaExpression conversion)

public static BinaryExpression Subtract
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression Subtract(Expression left,
     Expression right);
public static BinaryExpression SubtractAssign(Expression left,
                                              Expression right);
public static BinaryExpression SubtractAssign
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression SubtractAssign
    (Expression left, Expression right, MethodInfo method,
     LambdaExpression conversion)
public static BinaryExpression SubtractAssignChecked
    (Expression left, Expression right, MethodInfo method);
public static BinaryExpression SubtractAssignChecked
    (Expression left, Expression right);
public static BinaryExpression SubtractChecked
    (Expression left, Expression right);
```

```
public static BinaryExpression SubtractChecked
    (Expression left, Expression right, MethodInfo method);
```

The following is from the v1 spec ... except liftToNull semantics which now describes the code that shipped ... and comments regarding ...Assign factory methods

Left and right parameters must be non-null.

To determine an implementation of the node, if a non-null method is supplied, that becomes the implementing method for this node. It must represent a non-void static method with two arguments, or an exception occurs. Otherwise, if left.Type and right.Type are both numeric or both boolean types for which the corresponding operator is predefined in CLI, the factories set the implementing method is null. In the case of Power, if left.Type and right.Type both represent double, the implementing method is System.Math.Pow. Otherwise, if either of left.Type and right.Type contains a user definition of the corresponding binary operator (in the form of a static non-void op_... method with two arguments), the MethodInfo representing that becomes the implementing method. Otherwise, an exception occurs.

If the implementing method is non-null, then:

- if left.Type and right.Type are assignable to the corresponding argument types of the implementing method, the node is non-lifted, and the node type is the return type of the implementing method.
- If the following is true, then the BinaryExpression's Type is lifted to the corresponding nullable type:
    - left.Type and right.Type are both value types and both nullable
    - the corresponding non-nullable types are equal to the corresponding argument types of the implementing method
    - the return type of the implementing method is non-nullable value type
    - you supply liftToNull as true

  If the method return type is not bool, and both arguments are nullable, then the result will always be a nullable type regardless of the liftToNull argument.
- Otherwise an exception occurs.

If the implementing method is null, then left.Type and right.Type are numeric or boolean types for which the corresponding operator is predefined in C#. Furthermore

- If both left.Type and right.Type are non-nullable, the node is non-lifted, and the node type is the result type of the C# predefined operator.
- If both left.Type and right.Type are nullable, the node is lifted, and the node type is the nullable type corresponding to the result type of the C# predefined operator.
- Otherwise an exception occurs.

The ...Assign factories require the left argument expression to be one of the node types ParameterExpression, MemberExpression, or IndexExpression. They use the method argument only for the basic binary operation (Add, Subtract, etc.) and then assign the result to the location specified by the left argument. If methodinfo is null, and the Left.Type and Right.Type properties represent the same numeric type, the node has the semantics of IL addition. Otherwise, the node searches for and applies a user-defined op_Addition method. If the node's conversion lambda is non-null, then the semantics is to pass the result of the basic binary operation to the lambda. The result of the conversion lambda is then stored in the Left location.

If the conversion lambda is non-null, and the Left.Type and Right.Type properties represent numeric types, then the factories throw and exception.

The resulting BinaryExpression has:

- Node kind set to the ExpressionType member with the same name as the factory method
- Left and Right set to left and right, respectively
- Type set to the node type as described above
- Method set to the implementing method
- If the node is lifted, IsLifted and IsLiftedToNull are true, otherwise they are false.
- Conversion set to null for all node kinds except the ...Assign node kinds, in which case Conversion is set to the supplied MethodInfo.

4.6.10 Obsolete Array Index (Single-dimension) Factory

The ArrayIndex factories will be obsolete in lieu of the more general IndexExpression factory methods.

Expression has the following factory methods for BinaryExpressions representing single-dimension array element fetching:

```
public static BinaryExpression ArrayIndex(Expression array,
                                          Expression index);
```

The following is derived from the v1 spec ...

Array and index must be non-null. array.Type must represent an array type with rank 1, and index.Type must represent the int type. The resulting BinaryExpression has:

- NodeType ArrayIndex.
- Left and Right properties equal to array and index, respectively.
- Type representing the element type of array.Type.
- Method and Conversion are null.
- Both IsLifted and IsLiftedToNull are false.

4.6.11 Assignment Factory Method

Expression has the following factory methods for BinaryExpressions representing assignment operations:

```
public static BinaryExpression Assign(Expression left,
                                      Expression right);
```

Left must be one of the node types ParameterExpression, MemberExpression, or IndexExpression. Right.Type must be reference assignable to Left.Type. The resulting node has node kind Assign. If the left type represents a property or indexed property, it must have a setter.

4.6.12 Coalesce Operator Factory Methods

Expression has the following factory methods for BinaryExpressions representing coalesce operations (that is, what 'or' returns in a dynamic language):

```
public static BinaryExpression Coalesce(Expression left,
                                        Expression right);
public static BinaryExpression Coalesce
    (Expression left, Expression right,
     LambdaExpression conversion);
```

The following is from the v1 spec ...

Left and right must be non-null. left.Type must represent a reference type or a nullable value type. If left.Type is nullable, and right.Type is implicitly convertible to the non-nullable version of left.Type then the result type is the non-nullable version of left.Type. Otherwise if right.Type is implicitly convertible to left.Type then the result type is left.Type. Otherwise if the non-nullable version of left.Type is implicitly convertible to right.Type then the result type is right.Type. Otherwise an exception is thrown.

If conversion is non-null, it must have conversion.Type equal to a delegate type. The return type of the delegate type (conversion.Type) must NOT be void, and must be equal to right.Type. The delegate type must have exactly one parameter, and the type of this parameter must be assignable from the erased or unerased version of left.Type.

The resulting BinaryExpression has Left and Right properties equal to left and right, respectively, Conversion equal to conversion, and Type equal to the result type. Method is null, and both IsLifted and IsLiftedToNull are false.

4.6.13 Conditional And and Or Operator Factory Methods

Expression has the following factory methods for BinaryExpressions representing conditional 'and' and 'or' operations (referred to sometimes as "short-circuiting"):

```
public static BinaryExpression AndAlso(Expression left,
                                       Expression right);
public static BinaryExpression AndAlso
    (Expression left, Expression right, MethodInfo method);

public static BinaryExpression OrElse(Expression left,
                                      Expression right);
public static BinaryExpression OrElse
    (Expression left, Expression right, MethodInfo method);
```

The following is from the v1 spec ... except liftToNull semantics which now describes the code that shipped ...

Left and right must be non-null. If a non-null method is supplied, that becomes the implementing method for this node. It must represent a non-void static method with two arguments, or an exception occurs.

Otherwise, if either of left.Type and right.Type contains a user definition of the corresponding binary operator (in the form of a static non-void op_... method with two arguments), the

MethodInfo representing that becomes the implementing method. Otherwise, if left.Type and right.Type are both numeric or boolean types for which the corresponding operator is predefined in C#, the implementing method is null. Otherwise, an exception occurs.

If the implementing method is non-null, then:

- if left.Type and right.Type are assignable to the corresponding argument types of the implementing method, the node is non-lifted, and the node type is the return type of the implementing method.
- If the following is true, then the BinaryExpression's Type is lifted to the corresponding nullable type:
  - left.Type and right.Type are both value types and both nullable
  - the corresponding non-nullable types are equal to the corresponding argument types of the implementing method
  - the return type of the implementing method is non-nullable value type
  - you supply liftToNull as true

  If the method return type is not bool, and both arguments are nullable, then the result will always be a nullable regardless of the "liftToNull" flag.
- Otherwise an exception occurs.

If the implementing method is null, then left.Type and right.Type are the same boolean type. Furthermore

- If both left.Type and right.Type are non-nullable, the node is non-lifted, and the node type is the result type of the C# predefined operator.
- If both left.Type and right.Type are nullable, the node is lifted, and the node type is the nullable type corresponding to the result type of the C# predefined operator.
- Otherwise an exception occurs.

The resulting BinaryExpression has Left and Right properties equal to left and right, respectively, Type equal to the node type and Method equal to the implementing method. If the node is lifted, IsLifted and IsLiftedToNull are true, otherwise they are false. Conversion is equal to null.

4.6.14 Comparison Operators Factory Methods

Expression has the following factory methods for BinaryExpressions representing comparison operations:

```
public static BinaryExpression Equal(Expression left,
                                     Expression right);
public static BinaryExpression Equal
    (Expression left, Expression right, Boolean liftToNull,
     MethodInfo method);
public static BinaryExpression ReferenceEqual(Expression left,
                                               Expression right)

public static BinaryExpression NotEqual(Expression left,
                                        Expression right);
public static BinaryExpression NotEqual
    (Expression left, Expression right, Boolean liftToNull,
     MethodInfo method);
public static BinaryExpression ReferenceNotEqual(Expression left,
                                                 Expression right)
```

```
    public static BinaryExpression GreaterThan
        (Expression left, Expression right, Boolean liftToNull,
         MethodInfo method);
    public static BinaryExpression GreaterThan(Expression left,
                                               Expression right);
    public static BinaryExpression GreaterThanOrEqual
        (Expression left, Expression right, Boolean liftToNull,
         MethodInfo method);
    public static BinaryExpression GreaterThanOrEqual(Expression left,
                                                      Expression
right);

    public static BinaryExpression LessThan
        (Expression left, Expression right, Boolean liftToNull,
         MethodInfo method);
    public static BinaryExpression LessThan(Expression left,
                                            Expression right);
    public static BinaryExpression LessThanOrEqual
        (Expression left, Expression right, Boolean liftToNull,
         MethodInfo method);
    public static BinaryExpression LessThanOrEqual(Expression left,
                                                   Expression right);
```

The following is from the v1 spec ... except liftToNull semantics which now describes the code that shipped ...

Left and right must be non-null. If a non-null method is supplied, that becomes the implementing method for this node. It must represent a non-void static method with two arguments, or an exception occurs.

Otherwise, if either of left.Type and right.Type contains a user definition of the corresponding binary operator (in the form of a static non-void op_... method with two arguments, using the CLS name), the MethodInfo representing that becomes the implementing method. Otherwise, if the corresponding operator is predefined in C# for left.Type and right.Type, the implementing method is null. Otherwise, an exception occurs.

If the implementing method is non-null, then:

- if left.Type and right.Type are assignable to the corresponding argument types of the implementing method, the node is non-lifted and the node type is the return type of the implementing method.
- If the following is true, then the BinaryExpression's Type is lifted to bool?:
    - left.Type and right.Type are both value types and both nullable
    - the corresponding non-nullable types are equal to the corresponding argument types of the implementing method
    - the return type of the implementing method is bool
    - you supply liftToNull as true

  If liftToNull is false, the Type property represents bool. If the method return type is not bool, and both arguments are nullable, then the result will always be a nullable regardless of the “liftToNull” flag. This accommodates languages like VB that return null if an argument is null and languages like C# that return false if an argument is null.
- Otherwise an exception occurs.

ReferenceEqual and ReferenceNotEqual are provided as convenience factories that ensure pointer comparisons without having to wrap each operand in Convert to Object expressions. The factories also aid readability of code. If either operand is a value type, then the factories throw an exception.

The resulting BinaryExpression has:

- Left and Right set to left and right, respectively
- Type set as described above
- Method set to the implementing method
- If the node is lifted, IsLifted is true and IsLiftedToNull is equal to the liftToNull argument; otherwise both are false.
- Conversion set to null.

4.6.15 General Factory Methods

Expression has the following general factory methods for BinaryExpressions:

```
public static BinaryExpression MakeBinary
     (ExpressionType binaryType, Expression left, Expression right);
public static BinaryExpression MakeBinary
     (ExpressionType binaryType, Expression left, Expression right,
      Boolean liftToNull, MethodInfo method);
public static BinaryExpression MakeBinary
     (ExpressionType binaryType, Expression left, Expression right,
      Boolean liftToNull, MethodInfo method,
      LambdaExpression conversion);
```

The following is from the v1 spec ...

Based on the value of binaryType, MakeBinary will return the result of calling the corresponding factory method above with the same parameters. If binaryType is not appropriate for any of the above factory methods, MakeBinary throws an ArgumentException.

All the requirements and guarantees of the called factory method apply.

4.7 TypeBinaryExpression Class

This class represents type tests. It can have node kinds TypeIs or TypeEqual. The former has the semantics of the IsInst CLR instruction (example below of how that is different than C#'s semantics), and it tests the Expression value for having a sub type of the TypeOperand value. TypeEqual tests for an exact type match in essence; for example, a boxed int will equal Int32 and Nullable<Int32> because both are valid types for a boxed int.

Example distinction between TypeIs node kind and C#'s 'is' operator:

```
using System;
using System.Linq;
using System.Linq.Expressions;

class Test {
  public static void Main() {
      Func<bool> func1
          // C# compiles the 'is' here directly instead of using
```

```
            // an ET and Expression.Compile().
            = () => new[] { DayOfWeek.Friday } is int[];
        Expression<Func<bool>> expr1
            // C# emits a TypeIs node for 'is' here, which is a bug
            // in 2008.
            = () => new[] { DayOfWeek.Friday } is int[];
        Console.WriteLine(func1());
        Console.WriteLine(expr1.Compile()());

    // When the above code is compiled and run:
    //    Expected: prints False and False
    //    Actual: prints False and True
```

The issue is that C# appropriate regards enums and integers as mutually type-distinct, though there are explicit conversions between them. However, the CLR compares an array with elements of the same underlying type as type equal, and arrays of enums in the CLR are just arrays of ints (in this case).

4.7.1 Class Summary

```
public sealed class TypeBinaryExpression : Expression {
    public Expression Expression { get; }
    public Type TypeOperand { get; }
    public TypeBinaryExpression Update(Expression expression)
```

4.7.2 Expression Property

This property returns the expression that produces a value for checking if its type is the type represented by the Type property.

Signature:

```
public Expression Expression { get; }
```

4.7.3 TypeOperand Property

This property returns the type to test whether Expression results in a value of this type.

Signature:

```
public Type TypeOperand { get; }
```

4.7.1 Update Method

This method creates a new expression that is like this one, but using the supplied children. If all of the children are the same, this method returns this expression instead of creating a new one. This is useful when implementing ExpressionVisitors to rewrite the current node only when you need to do so.

Signature:

```
public TypeBinaryExpression Update(Expression expression)
```

4.7.2 Factory Methods

Expression has the following factory methods for TypeBinaryExpressions:

```
public static TypeBinaryExpression TypeEqual
    (Expression expression, Type type);
public static TypeBinaryExpression TypeIs(Expression expression,
                                          Type type);
```

The following is from the v1 spec ...

Expression and type must be non-null. The resulting TypeBinaryExpression has:

- Node kind TypeIs or TypeEqual, as appropriate
- Type set to bool
- Expression and TypeOperand properties equal to expression and type arguments.

4.8 UnaryExpression Class

This class represents a many kinds of operations such as basic arithmetic, side-effecting arithemetic, control flow, and so on. With only a couple of exceptions, all the operations fit the shape of having a single operand as their primary input.

These nodes have many node kinds for the various operations they can support. See section 4.4 for more details on the semantics of various node kinds used with UnaryExpressions.

4.8.1 Class Summary

```
public sealed class UnaryExpression : Expression {
    public Boolean IsLifted { get; }
    public Boolean IsLiftedToNull { get; }
    public MethodInfo Method { get; }
    public Expression Operand { get; }
    public UnaryExpression Update(Expression operand)
```

4.8.2 IsLifted Property

This property returns true if the node represents an operator call that takes non-nullable parameters, but the calls passes nullable arguments.

Signature:

```
public Boolean IsLifted { get; }
```

4.8.3 IsLiftedToNull Property

This property returns true if the node represents a call to an operator that returns a nullable type. If a nullable argument evaluates to null (Nothing in Visual Basic), the operator returns a null reference (Nothing in Visual Basic).

Signature:

```
public Boolean IsLiftedToNull { get; }
```

4.8.4 Method Property

This property returns the MethodInfo associated the operation to further specify its semantics. The value may be null if the Type property represents a numeric or boolean type.

Signature:

```
public MethodInfo Method { get; }
```

4.8.5 Operand Property

This property returns the expression that models the single argument of the operation. The property's value may be null (when the node kind is Throw).

V1 guaranteed this was never null, but we allow null for the Throw node kind (at least).

Signature:

```
public Expression Operand { get; }
```

TypeAs and Convert node kinds use the Expression.Type property as the implicit operand for those operations.

4.8.6 Update Method

This method creates a new expression that is like this one, but using the supplied children. If all of the children are the same, this method returns this expression instead of creating a new one. This is useful when implementing ExpressionVisitors to rewrite the current node only when you need to do so.

Signature:

```
public UnaryExpression Update(Expression operand)
```

4.8.7 ArrayLength Factory Method

```
public static UnaryExpression ArrayLength(Expression array);
```

The following is derived from the v1 spec ...

Array must be non-null, and array.Type must represent an array type.

The resulting UnaryExpression has

- Node kind ArrayLength
- Operand set to expression
- Type set to the type int
- Method set to null
- IsLifted and IsLiftedToNull set to false

4.8.8 Conversion Factory Methods

Expression has the following factory methods for UnaryExpressions representing conversion operations:

```
public static UnaryExpression Convert
    (Expression expression, Type type, MethodInfo method);
public static UnaryExpression Convert(Expression expression,
                                      Type type);
public static UnaryExpression ConvertChecked
```

```
        (Expression expression, Type type, MethodInfo method);
    public static UnaryExpression ConvertChecked(Expression expression,
                                                 Type type);
```

The following is derived from the v1 spec ... with updates for correctness or new behaviors

Expression and type must be non-null. If a non-null method is supplied, that becomes the implementing method for this node. It must represent a non-void static method with one argument, or an exception occurs.

If method is null, the factories essentially prefer primitive type convert first, then implicit conversions (order determined by order reflection gives them and searching the list for the first one), then explicit conversions last. If either expression.Type or type contains a user definition of the implicit or explicit conversion operator (in the form of a static non-void op_Implicit or op_Explicit method with one argument), the MethodInfo representing that becomes the implementing method. ~~If more than one such method exists, an exception occurs~~. Otherwise,

- If both expression.Type and type represent value types, and both are numeric, Boolean, nullable, or non-nullable enumeration types, the implementing method is null.
- If either of expression.Type or type is a reference type, and an explicit boxing, unboxing or reference conversion exists from expression.Type to type, the implementing method is null.
- Otherwise, an exception occurs.

If the implementing method is non-null, then

- if expression.Type is assignable to the argument type of the implementing method, and the return type of the implementing method is assignable to type the node is non-lifted.
- If either or both of expression.Type or type is a nullable value type, and the corresponding non-nullable value types are equal to the argument type and the return type of the implementing method, respectively, then the node is lifted.
- Otherwise an exception occurs.

If the implementing method is null, then expression.Type and type are both numeric or boolean types. Furthermore

- If type is bool or bool? then expression.Type must also be either bool or bool?, or an exception occurs.
- If both expression.Type and type are non-nullable, the node is non-lifted.
- Otherwise the node is lifted.

Note, the Convert factories throw an exception if you try to convert to void. To convert to void so that you squelch the result of an expression (that is, execute it only for side-effects), you must finesse this with:

```
Expression.Block(typeof(void), expr)
```

The resulting UnaryExpression has NodeType Convert or ConvertChecked, respectively, Operand equal to expression, Type equal to type and Method equal to the implementing method. If the node is lifted, IsLifted is true, otherwise it is false. IsLiftedToNull is always false.

4.8.9 Functional Increment and Decrement Factory Methods

Expression has the following factory methods for UnaryExpressions representing functional increment and decrement operations:

```
public static UnaryExpression Decrement(Expression expression,
                                        MethodInfo method);
public static UnaryExpression Decrement(Expression expression);

public static UnaryExpression Increment(Expression expression,
                                        MethodInfo method);
public static UnaryExpression Increment(Expression expression);
```

If method is null, and the expression's Type property represents a numeric type, then the node uses .NET primitives for adding or subtracting one. If the expression's Type is not numeric, then the factory searches for a user defined op_Decrement or op_Increment implementation on the type.

4.8.10 Side-effecting Pre- and Post- Increment and Decrement Factory Methods

Expression has the following factory methods for UnaryExpressions representing side-effecting pre- and post- increment and decrement operations:

```
public static UnaryExpression PostDecrementAssign
    (Expression expression);
public static UnaryExpression PostDecrementAssign
    (Expression expression, MethodInfo method);
public static UnaryExpression PostIncrementAssign
    (Expression expression);
public static UnaryExpression PostIncrementAssign
    (Expression expression, MethodInfo method);

public static UnaryExpression PreDecrementAssign
    (Expression expression);
public static UnaryExpression PreDecrementAssign
    (Expression expression, MethodInfo method);
public static UnaryExpression PreIncrementAssign
    (Expression expression, MethodInfo method);
public static UnaryExpression PreIncrementAssign
    (Expression expression);
```

See section 4.4 for more information.

4.8.11 Numeric Negation and Plus Factory Methods

Expression has the following factory methods for UnaryExpressions representing numeric negation and "plus" (CLR op_UnaryPlus methods) operations:

```
public static UnaryExpression Negate(Expression expression);
public static UnaryExpression Negate(Expression expression,
                                     MethodInfo method);
public static UnaryExpression NegateChecked(Expression expression,
                                            MethodInfo method);
public static UnaryExpression NegateChecked
    (Expression expression);

public static UnaryExpression UnaryPlus(Expression expression);
public static UnaryExpression UnaryPlus(Expression expression,
                                        MethodInfo method);
```

The following is derived from the v1 spec ... with updates for correctness or new behaviors

Expression must be non-null. If a non-null method is supplied, that becomes the implementing method for this node. It must represent a non-void static method with one argument, or an exception occurs.

Otherwise, if expression.Type contains a user definition of the unary plus or minus operator respectively (in the form of a static one-argument non-void op_UnaryPlus or op_UnaryNegation method), the MethodInfo representing that becomes the implementing method. Otherwise, if expression.Type is a numeric type, the implementing method is null. Otherwise, an exception occurs.

If the implementing method is non-null, then

- if expression.Type is assignable to the argument type of the implementing method, the node is non-lifted, and the node type is the return type of the implementing method.
- If the following is true, then the UnaryExpression's Type is lifted:
  - expression.Type is a nullable value types
  - the corresponding non-nullable type is equal to the corresponding argument type of the implementing method
  - the return type of the implementing method is non-nullable value type

  Also, the factories then lift the node type to the corresponding nullable type for the method's return type.
- Otherwise an exception occurs.

If the implementing method is null, then expression.Type is a numeric type. The node type is then expression.Type. Furthermore

- if expression.Type is non-nullable, the node is non-lifted.
- Otherwise the node is lifted.

The resulting UnaryExpression has:

- node kind UnaryPlus or Negate
- Operand set to expression
- Type set as described above
- Method set to the implementing method

If the node is lifted as described above, IsLifted and IsLiftedToNull are true; otherwise, they are false.

4.8.12 Logical and Bit Negation Factory Methods

Expression has the following factory methods for UnaryExpressions representing logical and bit negation operations:

```
public static UnaryExpression Not(Expression expression);
public static UnaryExpression Not(Expression expression,
                                  MethodInfo method);

public static UnaryExpression OnesComplement
    (Expression expression)
public static UnaryExpression OnesComplement
     (Expression expression, MethodInfo method)
```

The following is derived from the v1 spec ... with updates for correctness or new behaviors

Expression must be non-null. If a non-null method is supplied, that becomes the implementing method for this node. The method must represent a non-void static method with one argument, or an exception occurs.

Otherwise, if expression.Type contains a user definition of a unary not operator (in the form of a static non-void op_LogicalNot or op_OnesComplement method with one argument), the MethodInfo representing that method becomes the implementing method. The factories look first for logical then for bitwise op_... methods. Otherwise, if expression.Type is a numeric or boolean type, the implementing method is null. Otherwise, an exception occurs.

For legacy purposes, the Not factory still uses the Not node kind for a bitwise negation. The OneComplement factory uses the OnesComplement node kind. We introduced the new node kind for DLR interoperability in the DynamicMetaObject protocol, and for meta-programming scenarios where the node's intent is readily manifest.

If the implementing method is non-null, then

- if expression.Type is assignable to the argument type of the implementing method, the node is non-lifted, and the node type is the return type of the implementing method.
- If the following is true, then the UnaryExpression's Type is lifted:
  - expression.Type is a nullable value types
  - the corresponding non-nullable type is equal to the corresponding argument type of the implementing method
  - the return type of the implementing method is non-nullable value type

  Also, the factories then lift the node type to the corresponding nullable type for the method's return type.
- Otherwise an exception occurs.

If the implementing method is null, then expression.Type may be numeric, Boolean, or user-defined type. The node type is then expression.Type. If the type is numeric or Boolean, then:

- if expression.Type is non-nullable, the node is non-lifted.
- Otherwise the node is lifted.

The resulting UnaryExpression has:

- node kind Not if you called Not, OnesComplement if you called OnesComplement
- Operand set to expression
- Type set as described above
- Method set to the implementing method

If the node is lifted as described above, IsLifted and IsLiftedToNull are true; otherwise, they are false.

4.8.13 IsTrue and IsFalse Factories

Expression has the following factory methods for UnaryExpressions representing testing if a value is a true value or a false value:

```
public static UnaryExpression IsFalse(Expression expression) {
public static UnaryExpression IsFalse(Expression expression,
                                      MethodInfo method) {
public static UnaryExpression IsTrue(Expression expression) {
public static UnaryExpression IsTrue(Expression expression,
```

```
                                        MethodInfo method) {
```

Expression must be non-null. If a non-null method is supplied, that becomes the implementing method for this node. The method must represent a non-void static method with one argument, or an exception occurs.

Otherwise, if expression.Type contains a user definition of a unary op_false or op_true operator, respectively, the MethodInfo representing that method becomes the implementing method. The user-defined type must also implement the op_BitwiseAnd operator method if it implements op_False, and op_BitwiseOr if it implements op_True. Otherwise, if expression.Type implicitly converts to boolean, the implementing method is null. Otherwise, an exception occurs.

If the implementing method is non-null, then

- if expression.Type is assignable to the argument type of the implementing method, the node is non-lifted, and the node type is the return type of the implementing method.
- If the following is true, then the UnaryExpression's Type is lifted:
    - expression.Type is a nullable value types
    - the corresponding non-nullable type is equal to the corresponding argument type of the implementing method
    - the return type of the implementing method is non-nullable value type

  Also, the factories then lift the node type to the corresponding nullable type for the method's return type.
- Otherwise an exception occurs.

If the implementing method is null, then expression.Type may be implicitly convertible to Boolean, or user-defined type. The node type is then expression.Type.

The resulting UnaryExpression has:

- node kind IsTrue or IsFalse
- Operand set to expression
- Type set as described above
- Method set to the implementing method

If the node is lifted as described above, IsLifted and IsLiftedToNull are true; otherwise, they are false.

4.8.14 Quote Factory Method

Expression has the following factory methods for UnaryExpressions representing a specialized quoting operation for LINQ expressions:

```
public static UnaryExpression Quote(Expression expression);
```

The following is derived from the v1 spec ... with a correction to the constraint on the expression parameter and usage.

It would be rare that an ET producer would need to use this factory or node kind. It exists with special semantics for use in LINQ v1 language features. In v-next+1 we'll consider a complete quasi-quoting model. See section 4.4.42 for more information.

Expression must be non-null. Expression.Type must be Expression<T>, where T is a delegate type.

The resulting UnaryExpression has:

- node kind Quote
- Operand set to expression
- Type set to expression.Type.
- Method as null
- IsLifted and IsLiftedToNull as false

4.8.15 Throw Flow Control Factory Mehtods

Expression has the following factory methods for UnaryExpressions representing dynamic flow control:

```
public static UnaryExpression Rethrow();
public static UnaryExpression Rethrow(Type type);

public static UnaryExpression Throw(Expression value, Type type);
public static UnaryExpression Throw(Expression value);
```

Passing null for the value is equivalent to calling the Rethrow factory. This is for convenience.

The factories take a type even though of course a throw never returns a result. Allowing the type to be other than void means you can use these expressions in value positions where the sub expression's type must match the containing expression's type. For example, you could construct "condition ? foo : throw(e)". Otherwise, you'd have to awkwardly construct a Block to hold the Throw so that you could fake matching the type.

Value.Type must represent a type that is not a value type.

These factories result in nodes with node kind Throw.

4.8.16 Reference Conversion Factory Methods

Expression has the following factory methods for UnaryExpressions representing reference conversion operations:

```
public static UnaryExpression TypeAs(Expression expression,
                                     Type type);
```

Expression and type must be non-null, and type must represent a reference type or nullable type.

The resulting UnaryExpression has:

- node kind TypeAs
- Operand set to expression
- Type set to type
- Method is null
- IsLifted and IsLiftedToNull are false

4.8.17 Unboxing (as Pointer to Box's Value) Factory Method

Expression has the following factory methods for UnaryExpressions representing unboxing (to an interior pointer) operations:

```
public static UnaryExpression Unbox(Expression expression,
                                    Type type);
```

The resulting interior pointer from this node has a the type represented by the type argument, which becomes the Unbox node's Type property. The type argument must represent the boxed value's value type. The operand expression's Type property must represent an interface type or type Object; a value type would not be boxed for any reason other than to be passed or assigned to an interface type or type Object. If the operand's Type property does not represent the type Object, then the Unbox node's Type property must represent a type that implements the operand's interface type.

The resulting node has:

- Node kind Unbox
- Operand set to expression
- Type set to type
- Method set to null
- IsLifted and IsLiftedToNull set to false

4.8.18 General Factory Methods

Expression has the following general factory methods for UnaryExpressions:

```
public static UnaryExpression MakeUnary
    (ExpressionType unaryType, Expression operand, Type type);
public static UnaryExpression MakeUnary
    (ExpressionType unaryType, Expression operand, Type type,
     MethodInfo method);
```

The following is derived from the v1 spec …

Based on the value of unaryType, MakeUnary returns the result of calling the corresponding factory method above with operand, type, and method (where applicable). If unaryType does not correspond to one of these factory methods, an ArgumentException will be thrown.

All the requirements and guarantees of the called factory method apply.

4.8.19 NewDelegate Expression (V-next+1)

Cut from .NET 4.0, planned for v-next+1.

We plan to add support for creating delegate values in v-next+1. For example, if you want to represent this expression:

```
new MyDelegate(obj.MyMethod);
```

You would need to generate a call to Delegate.CreateDelegate:

```
Expression.Convert
    (Expression.Call
        (typeof(Delegate)
            .GetMethod("CreateDelegate",
                       new[] { typeof(Type), typeof(object),
                               typeof(MethodInfo) }),
          Expression.Constant(typeof(MyDelegate)),
```

```
                Expression.Constant(myObj, typeof(object)),
                Expression.Constant(myObj.GetType()
                                        .GetMethod("MyMethod"))),
            typeof(MyDelegate))
```

Creating an ET like the following would be much easier:

```
        Expression.NewDelegate
            (typeof(MyDelegate),
             Expression.Constant(myObj, typeof(object)),
             myObj.GetType().GetMethod("MyMethod"))
```

Compilation would need to finesse the MethodInfo pointer to an IntPtr for the delegate type's constructor methods (using the efficient ldftn IL instruction).

4.9 BlockExpression Class

This class represents a sequence of expressions. Each expression executes in order, squelching the result of each except the last expression. The value of the Block node is the result of the last expression in the body, ignoring any exits via GotoExpression. The BlockExpression uses the Block node kind.

If the BlockExpression.Type represents a type that is void, then the result of the last expression is automatically "converted to void" or squelched. If the Block's type is other than void, then the last expression's Type property must represent a type that is reference assignable to the block's type.

Each block also has a collection of variables whose scope is the body of the block. Ignoring closures, the variables' lifetimes are limited to the block.

The automatic conversion to void for the last expression also provides a simple work around for converting to void. The Convert factories throw an exception if you try to convert to void. To convert to void so that you squelch the result of an expression (that is, execute it only for side-effects), you can use: BlockExpression: Expression.Block(typeof(void), expr).

Variables must be listed using ParameterExpressions as lexical variables in a BlockExpression to (in effect) define them in some sub tree. To reference a variable, you alias the ParameterExpression object used to define the variable. Note, while Parameter node references are what determine variable binding, you can declare the same Parameter object in nested BlockExpressions. The ET compiler resovles the references to the tightest containing Block that declares the Parameter.

See section 4.4.50 for more details on the semantics of BlockExpressions, especially for unique binding semantics and guarantees (for example, within a loop).

4.9.1 Class Summary

```
public sealed class BlockExpression : Expression {
    public ReadOnlyCollection<Expression>
           Expressions { get; }
    public Expression Result { get; }
    public ReadOnlyCollection<ParameterExpression> Variables { get; }
    public BlockExpression Update
        (IEnumerable<ParameterExpression> variables,
         IEnumerable<Expression> expressions)
```

4.9.2 Expressions Property

This property returns the collection of expressions that form the body of the block.

Signature:

```
public ReadOnlyCollection<Expression>
       Expressions { get; }
```

4.9.3 Result Property

This property is a convenience for accessing the last expression in this block's Expressions collection.

Signature:

```
public Expression Result { get; }
```

4.9.4 Variables Property

This property returns the collection of variables scoped to this BlockExpression.

Signature:

```
public ReadOnlyCollection<ParameterExpression>Variables { get; }
```

4.9.5 Update Method

This method creates a new expression that is like this one, but using the supplied children. If all of the children are the same, this method returns this expression instead of creating a new one. This is useful when implementing ExpressionVisitors to rewrite the current node only when you need to do so.

Signature:

```
public BlockExpression Update
     (IEnumerable<ParameterExpression> variables,
      IEnumerable<Expression> expressions)
```

4.9.6 Factory Methods

The Expression class has the following factory methods for BlockExpression:

```
public static BlockExpression Block
     (IEnumerable<ParameterExpression> variables,
      params Expression[] expressions);
public static BlockExpression Block
     (Type type,
      IEnumerable<ParameterExpression> variables,
      params Expression[] expressions);
public static BlockExpression Block
     (IEnumerable<ParameterExpression> variables,
      IEnumerable<Expression> expressions);
public static BlockExpression Block
     (Type type,
      IEnumerable<ParameterExpression> variables,
      IEnumerable<Expression> expressions);
public static BlockExpression Block
```

```
        (IEnumerable<Expression> expressions);
    public static BlockExpression Block
        (Type type,
         IEnumerable<Expression> expressions);
    public static BlockExpression Block
        (Expression arg0, Expression arg1, Expression arg2,
         Expression arg3, Expression arg4);
    public static BlockExpression Block
        (params Expression[] expressions);
    public static BlockExpression Block
        (Type type, params Expression[] expressions);
    public static BlockExpression Block(Expression arg0,
                                        Expression arg1);
    public static BlockExpression Block
        (Expression arg0, Expression arg1, Expression arg2);
    public static BlockExpression Block
        (Expression arg0, Expression arg1, Expression arg2,
         Expression arg3);
```

When type is supplied, if it is void, then the last expression's result in the block does not have to be assignable to type, and the result is automatically "converted to void" or squelched. If type is supplied as non-void, then the last expression's Type must represent a type that is reference-assignable to the block's type. When type is not supplied, the block's Type property is set to the last expression's Type property.

No variables element may be null. If any element is duplicated, the factory throws an exception.

Post CLR 4.0 (remove IsByRef error check): If any of the ParameterExpressions representing the local variables has IsByRef set to True, then you must initialize them with an AssignRef BinaryExpression node. Otherwise, if the Body of the Block references the variable, or the variable appears as the Left expression of an Assign BinaryExpression node, there will be compile time error (or possibly an error when the code executes).

4.10 ConstantExpression Class

This class models a literal constant in code. Its value is the Value object. Its node kind is Constant. A ConstantExpression may have any Value, and the value may not have any syntactic representation in any programming language.

Note, when using this node, if the Value is not serializable (more specifically, compilable to IL), then the ET containing this node will not be serializable.

4.10.1 Class Summary

```
public class ConstantExpression : Expression {
    public Object Value { get; }
```

4.10.2 Value Property

This property returns the expression that models the value of the constant expression.

4.10.3 Factory Methods

Expression has the following factory methods for ConstantExpressions:

```
public static ConstantExpression Constant(Object value);
public static ConstantExpression Constant(Object value, Type type);
```

If type is supplied, it must be non-null. If value is not null, type must be assignable from the actual run-time type of value.

The resulting ConstantExpression has:

- Node kind Constant
- Value set to value
- Type set to type if supplied. If value is not null, then Type is value's type. If value is null, Type is Object.

4.11 ConditionalExpression Class

This class represents an if-then-else for value. The node kind is Conditional. See section 4.4 for more details on the semantics.

4.11.1 Class Summary

```
public sealed class ConditionalExpression : Expression {
    public Expression IfFalse { get; }
    public Expression IfTrue { get; }
    public Expression Test { get; }
    public ConditionalExpression Update
        (Expression test, Expression ifTrue, Expression ifFalse)
```

4.11.2 IfFalse Property

This property returns the expression to evaluate if the Test expression results in false.

Signature:

```
public Expression IfFalse { get; }
```

4.11.3 IfTrue Property

This property returns the expression to evaluate if the Test expression results in false.

Signature:

```
public Expression IfTrue { get; }
```

4.11.4 Test Property

This property returns the expression to evaluate as the Test of the condition, determining which sub expression control flows to next.

Signature:

```
public Expression Test { get; }
```

4.11.5 Update Method

This method creates a new expression that is like this one, but using the supplied children. If all of the children are the same, this method returns this expression instead of creating a new one. This is useful when implementing ExpressionVisitors to rewrite the current node only when you need to do so.

Signature:

```
public ConditionalExpression Update
    (Expression test, Expression ifTrue, Expression ifFalse)
```

4.11.6 Factory Methods

Expression has the following factory methods for ConditionalExpressions:

```
public static ConditionalExpression Condition
    (Expression test, Expression ifTrue, Expression ifFalse);
public static ConditionalExpression Condition
    (Expression test, Expression ifTrue, Expression ifFalse,
     Type type);
public static ConditionalExpression IfThen
    (Expression test, Expression ifTrue)
public static ConditionalExpression IfThenElse
    (Expression test, Expression ifTrue, Expression ifFalse);
```

Test, ifTrue, and ifFalse must all be non-null. Test.Type must represent the bool type. If you do not supply the type argument, then IfTrue.Type and IfFalse.Type must represent the same type. If you do supply type, and it is not void, then IfTrue.Type and IfFalse.Type must both be reference-assignable to the supplied type. If type is void, then the sub expression types do not have to match, and any resulting value is "converted to void" or squelched.

The resulting ConditionalExpression has:

- Test, IfTrue, and IfFalse properties set to test, ifTrue and ifFalse, respectively
- Node kind Conditional
- Type set to ifTrue.Type if type is not supplied

4.12 DynamicExpression Class

This class represents a dynamic operation that must be bound at runtime of the expression tree. The semantics is to create a DLR CallSite for caching of implementations of the operation given the different kinds of objects passed to the CallSite during the program's execution. The Dynamic node has a DLR CallSiteBinder that determines the exact semantics of the operation given the runtime operands. The binder encapsulates the language semantics for the creator of the node, as well as any payload meta data that informs the binder how to compute an implementation of the operation at runtime.

For more on expected semantics, see other node kinds' statements as to their semantics in DynamicExpression nodes. You might also see the sub classes of DynamicMetaDataBinder and language documentation on their dynamic semantics. See the documents at on the DLR Codeplex site, sites-binders-dynobj-interop.doc and library-authors-introduction.doc.

To enable ET consumers to meta-program with ETs, languages should provide and document factories for their binders. For example, if you are rewriting a static operation into a dynamic one, or breaking a dynamic operation into two sub operations, you'll need to call the Expression.Dynamic factory. You'll need a CallSiteBinder from the language that captures the semantics you want, and you'll need to supply the binder with the appropriate payload meta data it needs to compute how to perform the operation at runtime.

4.12.1 Class Summary

```
public class DynamicExpression : Expression {
    public ReadOnlyCollection<Expression>
           Arguments { get; }
    public CallSiteBinder Binder { get; }
    public Type DelegateType { get; }
    public DynamicExpression Update(IEnumerable<Expression> arguments)
```

4.12.2 Arguments Property

This property returns the argument expressions that are the operands to the operation.

Signature:

```
    public ReadOnlyCollection<Expression>
           Arguments { get; }
```

4.12.3 Binder Property

This property returns the binder that the DLR calls at runtime to compute an implementation of the operation given the runtime types of the operands. The binder may include private payload meta data that further refines it computation (for example, whether an integer operation should throw on overflow or implicitly roll over to infinite precision integers.

Signature:

```
    public CallSiteBinder Binder { get; }
```

4.12.4 DelegateType

This property returns the delegate used to construct the CallSite<T> that manages the caching of dispatch rules for this operation. The first argument of the delegate type's Invoke method must be of type CallSite.

Signature:

```
    public Type DelegateType { get; }
```

4.12.5 Update Method

This method creates a new expression that is like this one, but using the supplied children. If all of the children are the same, this method returns this expression instead of creating a new one. This is useful when implementing ExpressionVisitors to rewrite the current node only when you need to do so.

Signature:

```
    public DynamicExpression Update(IEnumerable<Expression> arguments)
```

4.12.6 Factories

The Expression class has the following factories for creating DynamicExpressions:

```
public static DynamicExpression Dynamic
    (CallSiteBinder binder, Type returnType, Expression arg0,
     Expression arg1);
public static DynamicExpression Dynamic
    (CallSiteBinder binder, Type returnType, Expression arg0,
     Expression arg1, Expression arg2);
public static DynamicExpression Dynamic
    (CallSiteBinder binder, Type returnType, Expression arg0,
     Expression arg1, Expression arg2, Expression arg3);
public static DynamicExpression Dynamic
    (CallSiteBinder binder, Type returnType,
     IEnumerable<Expression> arguments);
public static DynamicExpression Dynamic
    (CallSiteBinder binder, Type returnType,
     params Expression[] arguments);
public static DynamicExpression Dynamic
    (CallSiteBinder binder, Type returnType, Expression arg0);
public static DynamicExpression MakeDynamic
    (Type delegateType, CallSiteBinder binder,
     IEnumerable<Expression> arguments);
public static DynamicExpression MakeDynamic
    (Type delegateType, CallSiteBinder binder,
     params Expression[] arguments);
public static DynamicExpression MakeDynamic
    (Type delegateType, CallSiteBinder binder, Expression arg0)
public static DynamicExpression MakeDynamic
    (Type delegateType, CallSiteBinder binder, Expression arg0,
     Expression arg1)
public static DynamicExpression MakeDynamic
    (Type delegateType, CallSiteBinder binder, Expression arg0,
     Expression arg1, Expression arg2)
public static DynamicExpression MakeDynamic
    (Type delegateType, CallSiteBinder binder, Expression arg0,
     Expression arg1, Expression arg2, Expression arg3)
```

If delegateType is supplied, the first argument to the delegate's Invoke method must be of type CallSite, and the parameter count must equal the number of arguments expressions supplied plus one.

4.13 CallInfo Class

This class represents argument information at DynamicExpression call sites. The information is in DLR binder objects (for example, InvokeMemberBinder, CreateInstanceBinder, GetIndexBinder).

This class is in the System.Dynamic namespace since it is only used in DynamicExpression binders.

4.13.1 Class Summary

```
public sealed class CallInfo {
```

```
    public int ArgumentCount { get; }
    public ReadOnlyCollection<string> ArgumentNames {get; }
```

4.13.2 ArgumentCount Property

This property returns the total number of argument expressions. For example "foo(1, 2, bar=3, baz=4)" has a count of four.

Signature:

```
    public int ArgumentCount { get; }
```

4.13.3 ArgumentNames Property

This property return the names used for any named arguments. If there are N names, then they are the names used in the last N argument expressions. For example "foo(1, 2, bar=3, baz=4)" has a collection of "bar" and "baz".

Signature:

```
    public ReadOnlyCollection<string> ArgumentNames {get; }
```

4.13.4 Factory Methods

The Expression class has the following factory methods for creating CallInfos:

```
    public static CallInfo CallInfo(int argCount,
                                    params string[] argNames)
    public static CallInfo CallInfo(int argCount,
                                    IEnumerable<string> argNames)
```

4.14 DebugInfoExpression Class

This class represents a point in an ET where there is debugging information (a la .NET sequence points). To clear the sequence point information, use an instance of this class with IsClear set to True.

4.14.1 Class Summary

```
public sealed class DebugInfoExpression : Expression {
    public SymbolDocumentInfo Document { get; }
    public virtual Int32 EndColumn { get; }
    public virtual Int32 EndLine { get; }
    public virtual Int32 StartColumn { get; }
    public virtual Int32 StartLine { get; }
    public virtual bool IsClear { get; }
```

4.14.2 Document Property

This property returns an object that contains information about the source code file for this sequence point.

4.14.3 StartLine Property

This property returns the starting line number in the file for this sequence point. This value is one-based, and it is an inclusive bound.

4.14.4 StartColumn Property

This property returns the starting column for this sequence point. This value is one-based, and it is an inclusive bound. (The underlying call in the .NET BCL is documented incorrectly.)

4.14.5 EndLine Property

This property returns the ending line number in the file for this sequence point. This value is one-based and must be greater than or equal to the start line.

4.14.6 EndColumn Property

This property returns the ending column for this sequence point. This value is one-based, and it is an exlusive bound. (The underlying call in the .NET BCL is documented incorrectly.) EndColumn should be greater than the StartColumn if StartLine equals EndLine.

The .NET 4.0 and Codeplex v1 DLR code only checks that the EndColumn is greater than or equal to the StartColumn when they are on the same line. The underlying .NET BCL call does no checking, just stored the data.

4.14.7 IsClear Property

This property returns False if the node sets debugging sequence point information, and it returns True if it clears the debugging information.

Signature:

```
public bool IsClear { get; }
```

4.14.8 Factory Methods

The Expression class has the following factory methods for creating DebugInfoExpressions:

```
public static DebugInfoExpression DebugInfo
    (SymbolDocumentInfo document, Int32 startLine,
     Int32 startColumn, Int32 endLine, Int32 endColumn);
public static DebugInfoExpression ClearDebugInfo
    (SymbolDocumentInfo document)
```

4.15 SymbolDocumentInfo Class

This class represents document information for debugging sequence point data. See DebugInfoExpression and http://msdn.microsoft.com/en-us/library/system.diagnostics.symbolstore.isymboldocument.aspx .

4.15.1 Class Summary

```
public class SymbolDocumentInfo {
```

```
public Guid DocumentType { get; }
public String FileName { get; }
public Guid Language { get; }
public Guid LanguageVendor { get; }
```

4.15.2 DocumentType Property

This property returns the documents unique type identifier. It defaults to the guid for a text file.

Signature:

```
public Guid DocumentType { get; }
```

4.15.3 FileName Property

This property returns the source file name.

Signature:

```
public String FileName { get; }
```

4.15.4 Language Property

This property returns the language's unique identifier, if any.

Signature:

```
public Guid Language { get; }
```

4.15.5 LanguageVendor Property

This property returns the language vendor's unique identifier, if any.

Signature:

```
public Guid LanguageVendor { get; }
```

4.15.6 Factory Methods

The Expression class has the following factory methods for creating SymbolDocumentInfos:

```
public static SymbolDocumentInfo SymbolDocument(String fileName,
                                                Guid language);
public static SymbolDocumentInfo SymbolDocument(String fileName);
public static SymbolDocumentInfo SymbolDocument
    (String fileName, Guid language, Guid languageVendor);
public static SymbolDocumentInfo SymbolDocument
    (String fileName, Guid language, Guid languageVendor,
     Guid documentType);
```

If documentType is not supplied, it defaults to the guid indicating text. If language or languageVendor are not supplied, they default to Guid.Empty.

See MSDN for more information on the members/parameters: and http://msdn.microsoft.com/en-us/library/system.diagnostics.symbolstore.isymboldocument.aspx .

4.16 TryExpression Class

This class represents represents try-catch control flow, including finally and fault blocks. TryExpression uses this node kind.

First the Body executes. Under normal execution, control flows from the body to the Finally expression, if any. The value of the TryExpresson is the Body expression's value. If an object is thrown (directly or indirectly) from within the Body, and there's no dynamically intervening appropriate catch handler, the CatchBlocks are considered in order to find one with a Test type that is assignable from the thrown object. If there is such a CatchBlock, execution flows to its Body and then to the Finally expression, if any. The CatchBlock's body produces the value for the TryExpression in this case.

If non-null, the Finally expression always executes regardless of dynamic flow of control to or through the TryExpression.

If Fault is non-null, then Finally is null, and Handlers is empty. The Fault expresson executes if an object is thrown from the Body, directly or indirectly. Control flows from the Fault expression to some appropriate catch handler, or possibly an process unhandled throw error block. If execution flows from the Body expression to the Finally expression, if any, and no objects are thrown, then the Fault expression does NOT execute.

4.16.1 Class Summary

```
public sealed class TryExpression : Expression {
    public Expression Body { get; }
    public Expression Fault { get; }
    public Expression Finally { get; }
    public ReadOnlyCollection<CatchBlock>
           Handlers { get; }
    public TryExpression Update(Expression body,
                                IEnumerable<CatchBlock> handlers,
                                Expression @finally, Expression fault)
```

4.16.2 Body Property

This property returns the expression to execute upon entering the TryExpression. This expression provides the value for the TryExpression, unless dynamic flow of control lands in a CatchBlock.

Signature:

```
    public Expression Body { get; }
```

4.16.3 Fault Property

This property returns the Expresson that executes if an object is thrown from the Body, directly or indirectly. Control flows from the Fault to other code that catches the object. Note, if Fault is non-null, then Finally is null, and Handlers is empty.

Signature:

```
    public Expression Fault { get; }
```

4.16.4 Finally Property

This property returns the Expression that executes either when the body completes or when a selected CatchBlock completes. The Finally expresson executes regardless of whether an object is thrown while the Body executes.

Signature:

```
public Expression Finally { get; }
```

4.16.5 Handlers Property

This property returns the collection of CatchBlocks to consider executing if an object is thrown while Body executes.

Signature:

```
public ReadOnlyCollection<CatchBlock>
       Handlers { get; }
```

4.16.6 Update Method

This method creates a new expression that is like this one, but using the supplied children. If all of the children are the same, this method returns this expression instead of creating a new one. This is useful when implementing ExpressionVisitors to rewrite the current node only when you need to do so.

Signature:

```
public TryExpression Update(Expression body,
                            IEnumerable<CatchBlock> handlers,
                            Expression @finally, Expression fault)
```

4.16.7 Factory Methods

These methods create expressions that represent try-catch control flow constructs.

Signatures:

```
public static TryExpression TryCatch
    (Expression body, params CatchBlock[] handlers);
public static TryExpression TryCatchFinally
    (Expression body, Expression finally,
     params CatchBlock[] handlers);
public static TryExpression TryFault(Expression body,
                                     Expression fault);
public static TryExpression TryFinally(Expression body,
                                       Expression finally);
public static TryExpression MakeTry
    (Type type, Expression body, Expression finally,
     Expression fault, IEnumerable<CatchBlock> handlers);
```

Body represents the instructions to execute under the try Expression.

Handlers is a collection of CatchBlocks, where control flows if one of their exception Test types is assignable from a thrown exception from within the body. The cases and tests within cases are executed and checked in the order they appear in the lists.

Finally is the expression to execute before flow of control exits the TryExpression, regardless of non-local exits.

Fault is the expression to execute when an exception occurs. If fault is supplied, there can be no finally supplied nor any cases. If there is no exception thrown, the fault expression does not execution, while a finally expression always executes.

Note: handlers can catch non-exception types, and Throw can throw them as well.

If type is not supplied, or supplied as non-void, then Body.Type and each CatchBlock.Type must be reference-assignable to the type. If type is not supplied, then the TryExpression.Type is Body.Type. When type is supplied as void, the types of sub expressions do not matter, and the any resulting value is "converted to void" or squelched. If the Body.Type is void, then every CatchBlock.Type must be void or have a null CatchBlock.Body.

4.17 CatchBlock Class

This class represents represents catch clauses for a TryExpression. These are not expressions because they cannot occur anywhere in an ET. They can only appear within a TryExpression. See the Handlers property of TryExpression for more information.

Note, the Variable effectively creates a lexical scope for the catch block, so re-using a ParameterExpression here that you've used in a containing BlockExpression or LambdaExpression will shadow those variables in the outer lexical scopes.

4.17.1 Class Summary

```
public sealed class CatchBlock {
    public Expression Body { get; }
    public Expression Filter { get; }
    public Type Test { get; }
    public ParameterExpression Variable { get; }
    public CatchBlock Update(ParameterExpression variable,
                             Expression filter, Expression body)
```

4.17.2 Body Property

The property returns the expression to execute if exeution transfers to this CatchBlock, which depends on Test and Fitler.

Signature:

```
    public Expression Body { get; }
```

4.17.3 Filter Property

This property returns the Filter expression that (if non-null) must evaluate to true for this CatchBlock to be chosen when looking for a catch handler.

Signature:

```
    public Expression Filter { get; }
```

4.17.4 Test Property

This property returns the type of thrown object this CatchBlock handles.

Signature:

```
public Type Test { get; }
```

4.17.5 Variable Property

This property returns the ParameterExpression that represents the variable that will be bound to the thrown object for purposes of executing the Filter expression and Body expressions.

Signature:

```
public ParameterExpression Variable { get; }
```

4.17.6 Update Method

This method creates a new expression that is like this one, but using the supplied children. If all of the children are the same, this method returns this expression instead of creating a new one. This is useful when implementing ExpressionVisitors to rewrite the current node only when you need to do so.

Signature:

```
public CatchBlock Update(ParameterExpression variable,
                         Expression filter, Expression body)
```

4.17.7 Factories

The Expression class has the following factories for creating CatchBlocks:

```
public static CatchBlock Catch(ParameterExpression variable,
                               Expression body);
public static CatchBlock Catch(Type type, Expression body);
public static CatchBlock Catch
    (ParameterExpression variable, Expression body,
     Expression filter);
public static CatchBlock Catch(Type type, Expression body,
                               Expression filter);

public static CatchBlock MakeCatchBlock
    (Type type, ParameterExpression variable, Expression body,
     Expression filter)
```

Then type is not supplied, the CatchBlock handles objects thrown of type represented by variable.Type.

Variable must not be IsByRef. Its Type property must be the same as that supplied in the type parameter.

If filter is supplied, its Type property must represent Boolean.

4.18 MethodCallExpression Class

This class represents static and instance method calls. Its node kind is Call.

If you want to invoke a callable object, such as applying a delegate or LambdaExpression to a list of arguments, then you should use an InvocationExpression.

There is a factory method that uses this node type to represent fetching an element from a multi-dimensional array (still has node kind Call). It will be obsolete in a future version. You should use IndexExpression to get and set array elements.

See section 4.4 for more details on the semantics of the Call node kind.

4.18.1 Class Summary

```
public class MethodCallExpression : Expression {
    public ReadOnlyCollection<Expression>
           Arguments { get; }
    public MethodInfo Method { get; }
    public Expression Object { get; }
    public MethodCallExpression Update(Expression @object,
                                       IEnumerable<Expression>
arguments)
```

4.18.2 Arguments Property

This property returns the read-only collection of argument expressions. If there are no arguments, this is an empty collection.

Signature:

```
    public ReadOnlyCollection<Expression>
           Arguments { get; }
```

4.18.3 Method Property

This property returns the MethodInfo representing the method to call, and it is guaranteed to be non-null.

If the MethodInfo represents a static method, then Object returns null.

To be obsolete in a future version, if the node represents fetching an element from a multi-dimensional array, then this property returns a MethodInfo for a public instance method named Get on the Object.Type type.

Signature:

```
    public MethodInfo Method { get; }
```

4.18.4 Object Property

This property returns the expression that models the target object that is the self argument for the method call. If the call is to a static method, then this returns null.

To be obsolete in a future version, if the node represents fetching an element from a multi-dimensional array, then this property returns the array expression.

Signature:

```
    public Expression Object { get; }
```

4.18.5 Update Method

This method creates a new expression that is like this one, but using the supplied children. If all of the children are the same, this method returns this expression instead of creating a new one. This is useful when implementing ExpressionVisitors to rewrite the current node only when you need to do so.

Signature:

```
    public MethodCallExpression Update(Expression @object,
                                       IEnumerable<Expression>
arguments)
```

4.18.6 General Call Factories

Expression has the following general call factory methods for MethodCallExpressions:

```
    public static MethodCallExpression Call
        (MethodInfo method, params Expression[] arguments);
    public static MethodCallExpression Call
        (MethodInfo method, Expression arg0, Expression arg1,
         Expression arg2, Expression arg3, Expression arg4);
    public static MethodCallExpression Call
        (MethodInfo method, Expression arg0, Expression arg1);
    public static MethodCallExpression Call
        (Expression instance, MethodInfo method,
         params Expression[] arguments);
    public static MethodCallExpression Call
        (Expression instance, MethodInfo method, Expression arg0,
         Expression arg1, Expression arg2);
    public static MethodCallExpression Call
        (Expression instance, MethodInfo method, Expression arg0,
         Expression arg1);
    public static MethodCallExpression Call(MethodInfo method,
                                            Expression arg0);
    public static MethodCallExpression Call
        (Expression instance, String methodName, Type[] typeArguments,
         params Expression[] arguments);
    public static MethodCallExpression Call
        (Type type, String methodName, Type[] typeArguments,
         params Expression[] arguments);
    public static MethodCallExpression Call
        (Expression instance, MethodInfo method,
         IEnumerable<Expression> arguments);
    public static MethodCallExpression Call
        (MethodInfo method,
         IEnumerable<Expression> arguments);
    public static MethodCallExpression Call
        (MethodInfo method, Expression arg0, Expression arg1,
         Expression arg2, Expression arg3);
    public static MethodCallExpression Call
        (MethodInfo method, Expression arg0, Expression arg1,
         Expression arg2);
    public static MethodCallExpression Call(Expression instance,
                                            MethodInfo method);
```

The following is derived from the v1 spec ... with updates for correctness or new behaviors

Method must be non-null. If method is an instance method, instance must be supplied as non-null. Instance's Type property must be assignable to the declaring type of the member represented by method. If the method is a static method, instance must be null (breaking bug fix from v1); otherwise, the factories throw an ArgumentException.

If arguments is omitted or null, there are no arguments. If provided, arguments must have the same number of elements as the number of parameters for the method. Each of the elements of arguments must be non-null, and the types of the values they represent must be assignable to the type of the corresponding parameter of method. There is a special case the factory handles when an element of arguments has a Type property representing a type that is not assignable to the corresponding parameter type. If the parameter's type is a sub type of LambdaExpression, and the argument Expression object itself (that is, the Expression node) is of a type that is assignable to the parameter's type, then the argument Expression node is wrapped in a Quote node. This supports a legacy ETs v1 behavior for how C# chose to implement expression such as "Expression<Func<...>> = (...) => ...".

Two overloads take methodName and resolve the MethodInfo for you. Language implementers or DLR CallSiteBinder implementers should never use these overloads since they may not have the same semantics of method resolution as your language. When using these overloads, instance, type, and methodName must not be null. These overloads search instance.Type (or type if you used that overload) and base types for methodName, case-INsensitively. Type parameters must match typeArguments, and parameter types must match argument expression types. If no method or more than one compatible method is found, these overloads throw an exception. Otherwise these overloads invoke Call with the instance (if supplied), the MethodInfo for the found method, and arguments to return a result.

The resulting MethodCallExpression has the Object and Method properties equal to instance and method, respectively. The Arguments property has the same elements as arguments, except that some elements may be wrapped in Quote nodes as described above. The Type property represents the return type of the method denoted by method. The NodeType property is Call.

4.18.7 Obsolete Multi-dimensional Array Index Factory

The ArrayIndex factories will be obsolete in lieu of the more general IndexExpression factory methods.

```
public static MethodCallExpression ArrayIndex
    (Expression array, params Expression[] indexes);
public static MethodCallExpression ArrayIndex
    (Expression array,
     IEnumerable<Expression> indexes);
```

A MethodCallExpression can represent fetching an array element from an array with rank greater than one. The value of the Method property must be a MethodInfo describing the public instance method named Get on a multi-dimensional array type.

The following is derived from the v1 spec ...

Array and indices must be non-null. Array.Type must represent an array type, and its rank must match the number of elements in indices. For each Expression, E, in indices, E.Type must represent the int type.

The MethodCallExpression's Object property is the array. The Arguments property is the result of ReadOnlyCollectionExtensions.ToReadOnlyCollection(indices). The Method property is the MethodInfo describing the public instance method named Get on the type represented by array.Type.

4.19 POST CLR 4.0 -- ComplexMethodCallExpression Class

We have this node type to describe function calls with unsupplied arguments, named argument values, etc.

See section 2.10 for more info.

4.19.1 Class Summary

```
public sealed class ComplexMethodCallExpression : MethodCallExpression
{
    public ReadOnlyCollection<Expression>
           Arguments { get; }
    public Expression[] ArgumentEvaluationOrder { get; }
    public ArgumentDescription[] ArgumentDescriptions { get; }
```

4.20 InvocationExpression Class

This class represents invoking callable objects. These nodes use the Invoke node kind. See section 4.4 for more details on the semantics.

4.20.1 Class Summary

```
public sealed class InvocationExpression : Expression {
    public ReadOnlyCollection<Expression>
           Arguments { get; }
    public Expression Expression { get; }
    public InvocationExpression Update
        (Expression expression, IEnumerable<Expression> arguments)
```

4.20.2 Arguments Property

This property returns the read-only collection of argument expressions that produce values to pass to the Expression object. This never returns null, using an empty collection for no arguments.

Signature:

```
    public ReadOnlyCollection<Expression>
           Arguments { get; }
```

4.20.3 Expression Property

This property returns the expression that models the callable object that is invoked on the Arguments to produce the result of this node. The expression represents either a System.Delegate or a LambdaExpression.

Signature:

```
public Expression Expression { get; }
```

4.20.4 Update Method

This method creates a new expression that is like this one, but using the supplied children. If all of the children are the same, this method returns this expression instead of creating a new one. This is useful when implementing ExpressionVisitors to rewrite the current node only when you need to do so.

Signature:

```
public InvocationExpression Update
    (Expression expression, IEnumerable<Expression> arguments)
```

4.20.5 Factory Methods

Expression has the following factory methods for InvocationExpressions:

```
public static InvocationExpression Invoke
    (Expression expression, params Expression[] arguments);
public static InvocationExpression Invoke
    (Expression expression,
     IEnumerable<Expression> arguments);
```

The following is derived from the v1 spec ... with updates for correctness or new behaviors

Expression must be non-null. Expression.Type must represent a delegate type D or a type Expression<D> where D is a delegate type. The list of parameters for D must have the same length as arguments, or 0 if arguments is null. Each of the elements of arguments must be non-null, and the types of the values they represent must be assignable to the type of the corresponding parameter of D. There is a special case the factory handles when an element of arguments has a Type property representing a type that is not assignable to the corresponding parameter type. If the parameter's type is a sub type of LambdaExpression, and the argument Expression object itself (that is, the Expression node) is of a type that is assignable to the parameter's type, then the argument Expression node is wrapped in a Quote node. This supports a legacy ETs v1 behavior for how C# chose to implement expression such as "Expression<Func<...>> = (...) => ...".

The resulting InvocationExpression has the Expression property equal to expression. The Arguments property has the same elements as arguments, except that some elements may be wrapped in Quote nodes as described above. The Type property represents the return type of D.

Implementation note: the expression compiler may choose to inline the invocation of a lambda expression, rather than create a separate CLR method.

4.21 IndexExpression Class

This class represents array access and indexed properties (property gets that take arguments). This node uses the Index node kind. An IndexExpression is allowed as an l-value, for example, with BinaryExpression and node kind Assign.

There is redundant modeling for accessing arrays (not assignment) due to ETs v1 backward compatibility. For LINQ features, VB and C# will continue to emit ETs with BinaryExpression (node kind ArrayIndex) for single dimension arrays and MethodCallExpression (method info is Array.Get) for mulit-dimensional arrays. VB and C# will emit IndexExpression for l-value locations when they extend their lambda support in a future version.

4.21.1 Class Summary

```
public class IndexExpression : Expression {
    public ReadOnlyCollection<Expression> Arguments { get; }
    public PropertyInfo Indexer { get; }
    public Expression Object { get; }
    public IndexExpression Update(Expression @object,
                                  IEnumerable<Expression> arguments)
```

4.21.2 Arguments Property

Signature:

```
    public ReadOnlyCollection<Expression> Arguments { get; }
```

4.21.3 Indexer Property

Signature:

```
    public PropertyInfo Indexer { get; }
```

4.21.4 Object Property

Signature:

```
    public Expression Object { get; }
```

4.21.5 Update Method

This method creates a new expression that is like this one, but using the supplied children. If all of the children are the same, this method returns this expression instead of creating a new one. This is useful when implementing ExpressionVisitors to rewrite the current node only when you need to do so.

Signature:

```
    public IndexExpression Update(Expression @object,
                                  IEnumerable<Expression> arguments)
```

4.21.6 Factory Methods

The Expressions class has the following factory methods for creating IndexExpressions:

```
    public static IndexExpression Property
```

```
        (Expression instance, PropertyInfo indexer,
         IEnumerable<Expression> arguments);
    public static IndexExpression Property
        (Expression instance, PropertyInfo indexer,
         params Expression[] arguments);
    public static IndexExpression Property
        (Expression instance, String propertyName,
         params Expression[] arguments);

    public static IndexExpression ArrayAccess
        (Expression array, IEnumerable<Expression> indexes);
    public static IndexExpression ArrayAccess
        (Expression array, params Expression[] indexes);

    public static IndexExpression MakeIndex
        (Expression instance, PropertyInfo indexer,
         IEnumerable<Expression> arguments);
```

Array must be an expression with a Type property representing an array (array.Type.IsArray is true). Indexes must all have a Type property that represents int32.

Indexer must not be null. Its PropertyType must not be a ByRef type, and it must not represent the void type. If Indexer represents as static property, then instance must be null, and vice versa.

Arguments must not be null, and it must not be empty. Arguments must have the same number of elements as the number of parameters for the property. Each of the elements of arguments must be non-null, and the types of the values they represent must be assignable to the type of the corresponding parameter of property. There is a special case the factory handles when an element of arguments has a Type property representing a type that is not assignable to the corresponding parameter type. If the parameter's type is a sub type of LambdaExpression, and the argument Expression object itself (that is, the Expression node) is of a type that is assignable to the parameter's type, then the argument Expression node is wrapped in a Quote node. This supports a legacy ETs v1 behavior for how C# chose to implement expression such as "Expression<Func<...>> = (...) => ...".

One overload takes propertyName and resolves the PropertyInfo for you. Language implementers or DLR CallSiteBinder implementers should never use this overload since it may not have the same semantics of resolution as your language. Instance and propertyName must not be null. This overload searches instance.Type and base types for propertyName, case-INsensitively. Parameter types must be reference assignable from corresponding argument expression types. If no method or more than one compatible method is found, this overload throws an exception.

4.22 LoopExpression Class

This class represents an infinite loop, executing its Body repeatedly until a sub expression of the body exits the loop via a GotoExpression.

Loop nodes have a Break property with a LabelTarget. When control transfers to this label with a value, the value becomes the result of the LoopExpression. The Break label can be null, in which case the Loop's Type property represents the void type.

LoopExpression.Type is the same as LoopExpression.Break.Type if the Break label is non-null.

4.22.1 Examples

4.22.1.1 While Less Than Ten

```
    // Counts up to 10 and breaks.  Roughly:
    //   int i = 0;
    //   while (true) {
    //     if (i < 123) ++i; else break;
    //   }
    var i = Expression.Variable(typeof(int), "i");
    var b = Expression.Label();
    var lambda = Expression.Lambda<Action>(
        Expression.Block(
            new[] { i },
            Expression.Assign(I, Expression.Constant(0)),
            Expression.Loop(
                Expression.IfThenElse(
                    Expression.LessThan(i, Expression.Constant(10)),
                    Expression.PreIncrementAssign(i),
                    Expression.Break(b)
                ),
                b
            )
        )
    );
```

4.22.1.2 ForEach over IEnumerable

Here's an example of using LoopExpression to produce a ForEach loop:

```
// variable = Expression.Variable(typeof(object), "value");
public static Expression ForEach
    (ParameterExpression variable, Expression enumerable,
     Expression body) {
    ParameterExpression temp =
        Expression.Variable(typeof(IEnumerator), "$enumerator");
    @break = Expression.Label();
    return Expression.Block(
        new[] { temp, variable },
        Expression.Assign(temp,
                          Expression.Call(
                              enumerable,
                              typeof(IEnumerable)
                                      .GetMethod("GetEnumerator"))),
        Expression.Loop(
            Expression.Block(
                Expression.Condition(
                    Expression.Call(
                        temp,
                         typeof(IEnumerator)
                                  .GetMethod("MoveNext")),
                    Expression.Empty(),
```

```
                    Expression.Break(@break)),
                Expression.Assign(
                    variable,
                    Expression.Convert(
                        Expression.Property(
                            temp,
                            typeof(IEnumerator)
                                    .GetProperty("Current")),
                                                variable.Type)),
                body),
        @break));
}
```

4.22.1.3 Lexical Semantics with Blocks and Loops Involved

Note, these examples do not use LoopExpression, but they apply to LoopExpression and show how LoopExpressions can be thought of as reducing to Lablels and Goto's. These examples illustrate an interesting property of ETs and how to create strong lexical scoping for variables used within an iteration. They also show an underlying .NET issue with initializing variable storage locations vs. leaking through previous contents.

The following MakeLoop and MakeLoop2 (loop with lambda) produce expressions that when executed would create the resulting string shown in the table:

	Plain loop	loop with lambda

Tree's ToString	{
	var str;
	var count;
	Start:;
	{
	var i;
	(count += 1);
	(i += 1);
	(str = Concat(str, i.ToString(), "|"));
	};
	IIF((count < 10), goto Start,);
	str;
	}
		{
		var str;
		var count;
		Start:;
		{
		var i;
		(count += 1);
		(i += 1);
		(str = Concat(str, i.ToString(), "|"));
		() => i;
		};
		IIF((count < 10), goto Start,);
		str;
		}

result	1|2|3|4|5|6|7|8|9|10|	1|1|1|1|1|1|1|1|1|1|

Things to note are that the variable "i" is not explicitly initialized in either case, and the lambda expression in MakeLoop2 creates a closure over "i".

```
private static Expression MakeLoop(){
    LabelTarget start = Expression.Label("Start");
    ParameterExpression i = Expression.Parameter(typeof(int), "i");
    ParameterExpression count = Expression.Parameter(typeof(int),
                                                     "count");
    ParameterExpression str = Expression.Parameter(typeof(String),
```

```
                                                    "str");
    return Expression.Block(new ParameterExpression[] { str, count
},
        Expression.Label(start),
        Expression.Block(new ParameterExpression[] { i },
            Expression.AddAssign(count, Expression.Constant(1)),
            Expression.AddAssign(i, Expression.Constant(1)),
            Expression.Assign(
                str,
                Expression.Call(
                    typeof(String)
                       .GetMethod(
                           "Concat",
                           new Type[] { typeof(String),
                                        typeof(String),
                                        typeof(String) }),
                    str,
                    Expression.Call(i, "ToString", Type.EmptyTypes),
                    Expression.Constant("|")
                    )
                )
            ),
        Expression.IfThen(
            Expression.LessThan(count, Expression.Constant(10)),
            Expression.Goto(start)
            ),
        str
        );
}

private static Expression MakeLoop2() {
    LabelTarget start = Expression.Label("Start");
    ParameterExpression i = Expression.Parameter(typeof(int), "i");
    ParameterExpression count = Expression.Parameter(typeof(int),
                                                     "count");
    ParameterExpression str = Expression.Parameter(typeof(String),
                                                   "str");
    return Expression.Block(new ParameterExpression[] { str, count
},
        Expression.Label(start),
        Expression.Block(new ParameterExpression[] { i },
            Expression.AddAssign(count, Expression.Constant(1)),
            Expression.AddAssign(i, Expression.Constant(1)),
            Expression.Assign(
                str,
                Expression.Call(
                    typeof(String)
                       .GetMethod(
                           "Concat",
                           new Type[] { typeof(String),
                                        typeof(String),
                                        typeof(String) }),
                    str,
                    Expression.Call(i, "ToString", Type.EmptyTypes),
                    Expression.Constant("|")
                    )
                ),
```

```
            Expression.Lambda(i)
            ),
        Expression.IfThen(
            Expression.LessThan(count, Expression.Constant(10)),
            Expression.Goto(start)
            ),
        str
        );
}
```

The lambda create a closure over "i", which create a unique lexical binding in each iteration of the loop. .NET leaks old values into the variable for each iteration when there is no closure, so you get a string with one to ten in it. When there is a closure, .NET re-initializes the memory as it should, so you get a string with all ones.

If you want to capture closures over unique bindings/values of "i" in an interation, ETs correctly compile with those semantics using Goto's or Loops (when the Block is within the iteration bounds). However, you need to correctly initialize the loop variable that you close over. The following MakeLoop3 is the same as MakeLoop2 except that it uses an extra variable, "i_", that is outside of the Block that is inside the loop bounds. The extra variable counts one to ten and is the initialization value for "i" each time the code enters the inner Block that is within the loop bounds.

```
private static Expression MakeLoop3() {
    LabelTarget start = Expression.Label("Start");
    ParameterExpression i_ = Expression.Parameter(typeof(int),
"i_");
    ParameterExpression i = Expression.Parameter(typeof(int), "i");
    ParameterExpression count = Expression.Parameter(typeof(int),
                                                     "count");
    ParameterExpression str = Expression.Parameter(typeof(String),
                                                   "str");
    return Expression.Block(new ParameterExpression[]
                                { str, count, i_ },
        Expression.Label(start),
        Expression.Block(new ParameterExpression[] { i },
            Expression.Assign(i, i_),
            Expression.AddAssign(count, Expression.Constant(1)),
            Expression.AddAssign(i, Expression.Constant(1)),
            Expression.Assign(
                str,
                Expression.Call(
                    typeof(String)
                        .GetMethod(
                            "Concat",
                            new Type[] { typeof(String),
                                         typeof(String),
                                         typeof(String) }),
                    str,
                    Expression.Call(i, "ToString",
                                    Type.EmptyTypes),
                    Expression.Constant("|")
                    )
                ),
            Expression.Lambda(i),
```

```
                Expression.Assign(i_, i)
                ),
            Expression.IfThen(
                Expression.LessThan(count, Expression.Constant(10)),
                Expression.Goto(start)
                ),
            str
            );
    }
```

You can invoke the MakeLoop* functions in a console application's Main with the following:

```
    Console.WriteLine(Expression.Lambda<Func<string>>(MakeLoop3())
                                .Compile()());
```

4.22.2 Class Summary

```
public sealed class LoopExpression : Expression {
    public Expression Body { get; }
    public LabelTarget BreakLabel { get; }
    public LabelTarget ContinueLabel { get; }
    public LoopExpression Update(LabelTarget breakLabel,
                                 LabelTarget continueLabel,
                                 Expression body)
```

4.22.3 Update Method

This method creates a new expression that is like this one, but using the supplied children. If all of the children are the same, this method returns this expression instead of creating a new one. This is useful when implementing ExpressionVisitors to rewrite the current node only when you need to do so.

Signature:

```
    public LoopExpression Update(LabelTarget breakLabel,
                                 LabelTarget continueLabel,
                                 Expression body)
```

4.22.4 Factory Methods

These methods create LoopExpressions. The semantics of a LoopExpression is to loop forever, so the body Expression must explicitly exit the loop with a GotoExpression.

Signatures:

```
    public static LoopExpression Loop(Expression body);
    public static LoopExpression Loop
        (Expression body, LabelTarget break, LabelTarget continue);
    public static LoopExpression Loop(Expression body,
                                      LabelTarget break);
```

Body is the expression to repeatedly execute.

Break identifies a location you can use in a GotoExpression from inside of the Body expression so that the code will terminate the loop and continue execution after the loop. If the Goto has a value, then it is the value of the LoopExpression. If not supplied, Break is null.

Continue identifies a location you can use in a GotoExpression from inside of the Body expression so that the code will terminate the current iteration and continue execution at the start of body. If not supplied, Continue is null.

Expression.Type is Expression.Break.Type if Break is non-null; otherwise, it is the void type.

4.23 POST CLR 4.0 -- ForExpression Class

4.23.1 Class Summary

```
public sealed class ForExpression : Expression {
    public Expression Body { get; }
    public Expression Increment { get; }
    public LabelTarget BreakLabel { get; }
    public LabelTarget ContinueLabel { get; }
    public Expression Test { get; }
```

4.24 POST CLR 4.0 -- ForEachExpression Class

Here's an example of using LoopExpression to produce a ForEach loop:

```
// variable = Expression.Variable(typeof(object), "value");
public static Expression ForEach
    (ParameterExpression variable, Expression enumerable,
     Expression body) {
    ParameterExpression temp =
        Expression.Variable(typeof(IEnumerator), "$enumerator");
    @break = Expression.Label();
    return Expression.Block(
        new[] { temp, variable },
        Expression.Assign(temp,
                          Expression.Call(
                              enumerable,
                              typeof(IEnumerable)
                                      .GetMethod("GetEnumerator"))),
        Expression.Loop(
            Expression.Block(
                Expression.Condition(
                    Expression.Call(
                        temp,
                         typeof(IEnumerator)
                                  .GetMethod("MoveNext")),
                    Expression.Empty(),
                    Expression.Break(@break)),
                Expression.Assign(
                    variable,
                    Expression.Convert(
                        Expression.Property(
                            temp,
                            typeof(IEnumerator)
```

```
                                        .GetProperty("Current")),
                                                    variable.Type)),
                    body),
            @break));
}
```

4.24.1 Class Summary

```
public sealed class ForEachExpression : Expression {
    public Expression Body { get; }
    public Expression Iterable { get; }
    public LabelTarget BreakLabel { get; }
    public LabelTarget ContinueLabel { get; }
```

4.25 POST CLR 4.0 -- WhileExpression Class

SIDENOTE

IPy will need to build a PythonWhileExpression to handle the Python Else clause if IPy wants to support meta-programming via ETs that is closer to the programs users write. The IPy-specific ET node could reduce to a WhileExpr:

```
PythonWhile <cond>: <whilebody> else: <elsebody>  -->

While (<cond> ? true : Block { <elsebody>; false }) <whilebody>
```

4.25.1 Class Summary

```
public sealed class WhileExpression : Expression {
    public Expression Body { get; }
    public LabelTarget BreakLabel { get; }
    public LabelTarget ContinueLabel { get; }
    public Expression Test { get; }
```

4.26 POST CLR 4.0 -- RepeatUntilExpression Class

4.26.1 Class Summary

```
public sealed class RepeatUntilExpression : Expression {
    public Expression Body { get; }
    public LabelTarget BreakLabel { get; }
    public LabelTarget ContinueLabel { get; }
    public Expression Test { get; }
```

4.27 GotoExpression Class

This class represents an unstructured flow of control. It has node kind Goto and a sub kind that captures the intent of the goto (break, return, etc.) for meta-programming purposes.

The Goto refers to a LabelTarget that a LabelExpression must refer to somewhere in the ET, and it is the LabelExpression that sets the target location for the flow of control. The LabelExpression must be lexically in the same LambdaExpression.Body as the GotoExpression.

The semantics of the LabelTarget chosen as the destination is lexically scoped in a sense. If all LabelTargets in the LambdaExpression are unique, then the LabelTarget in the GotoExpression simply must be found within the LambdaExpression containing the Goto. If the same LabelTarget is used multiple times within a LambdaExpression, then the GotoExpression targets the first matching LabelTarget found while searching up the ET to the Lambda root. This is a convenience for re-writers or tree builders that re-use sub trees that contain LabelExpressions and GotoExpressions so that the sub trees behave as expected unto themselves.

The Goto can optionally deliver a value to the location, as expressed by a non-null Expression property. If this property is non-null, then the expression Type property must represent a type that is reference assignable to the type represented by Target.Type. However, if Target.Type is void, the GotoExpression.Expression.Type can represent anything since the ET compiler will automatically convert the result to void or squelch the value.

See section 4.4 for more details on the semantics of GotoExpression, as well as the introductory section on iterations, lexical exits, and gotos (2.4).

4.27.1 Class Summary

```
public sealed class GotoExpression : Expression {
    public GotoExpressionKind Kind { get; }
    public LabelTarget Target { get; }
    public Expression Value { get; }
    public GotoExpression Update(LabelTarget target, Expression value)
```

4.27.2 Kind Property

This property returns the kind of Goto node this is. This property has no semantic bearing and only exists for debugging or documentation as to the intent of the goto.

Signature:

```
    public GotoExpressionKind Kind { get; }
```

4.27.3 Target Property

This property returns the LabelTarget to which some LabelExpression within the same LambdaExpression must refer. This identifies the target location of the goto. This property is never null.

Signature:

```
    public LabelTarget Target { get; }
```

4.27.4 Value Property

This property returns the Expression that provides the value to carry to the target location when transferring control. This property may be null.

Signature:

```
public Expression Value { get; }
```

4.27.5 Update Method

This method creates a new expression that is like this one, but using the supplied children. If all of the children are the same, this method returns this expression instead of creating a new one. This is useful when implementing ExpressionVisitors to rewrite the current node only when you need to do so.

Signature:

```
public GotoExpression Update(LabelTarget target, Expression value)
```

4.27.6 Factory Methods

Expression has the following factory methods for GotoExpression:

```
public static GotoExpression Break(LabelTarget target);
public static GotoExpression Break(LabelTarget target,
                                   Expression value);
public static GotoExpression Break(LabelTarget target,
                                   Type type);
public static GotoExpression Break(LabelTarget target,
                                   Expression value, Type type);

public static GotoExpression Continue(LabelTarget target);
public static GotoExpression Continue(LabelTarget target,
                                      Type type);

public static GotoExpression Goto(LabelTarget target);
public static GotoExpression Goto(LabelTarget target,
                                  Expression value);
public static GotoExpression Goto(LabelTarget target,
                                  Type type);
public static GotoExpression Goto(LabelTarget target,
                                  Expression value, Type type);

public static GotoExpression MakeGoto
    (GotoExpressionKind kind, LabelTarget target,
     Expression value);

public static GotoExpression Return(LabelTarget target,
                                    Expression value);
public static GotoExpression Return(LabelTarget target);
public static GotoExpression Return(LabelTarget target,
                                    Type type);
public static GotoExpression Return(LabelTarget target,
                                    Expression value, Type type);
```

Target must be the target of some LabelExpression lexically within the same LambdaExpression.Body that contains the resulting GotoExpression. The factory does not confirm this, so if this is not true, you'll get a compile-time error.

Value, if supplied is an expression to execute, and its value is delivered to the target location (that is, the value remains on the IL stack upon transferring control to the new location). The

Expression's Type property must represent a type that is reference-assignable to the type represented by the target's Type property.

There are two special cases the factory handles when value's Type property represents a type that is not assignable to the type represented by target.Type. First, if Target.Type is void, the GotoExpression.Expression.Type can represent anything since the ET compiler will automatically convert the result to void or squelch the value. Second, if target's type is a sub type of LambdaExpression, and the value Expression object itself (that is, the Expression node) is of a type that is assignable to the target's type, then the value Expression node is wrapped in a Quote node. This supports a legacy ETs v1 behavior for how C# chose to implement expression such as "Expression<Func<...>> = (...) => ...".

4.28 GotoExpressionKind Enum

This enum represents the kinds of GotoExpressions the DLR supports. These values are for debugging or documentation convenience only for capturing the intended use of the goto.

4.28.1 Type Summary

```
public enum GotoExpressionKind {
    Goto,
    Return,
    Break,
    Continue
```

4.28.2 Members

The Summary section shows the type as it is defined (to indicate values of members), and this section documents the intent of the members.

`Goto`	The node is a basic goto targeting a label. The Value is a void DefaultExpression.
`Return`	The node represents a 'return' or exit from a lambda. The Value may be a void DefaultExpression or any expression.
`Break`	The node represents a goto within a loop to the exit target of the loop. The Value may be void or any expression.
`Continue`	The node represents a goto the beginning of a loop to abort the current iteration and continue with the next iteration and bindings of the iteration variables. The Value is a void DefaultExpression.

4.29 LabelExpression Class

This class represents an unstructured flow of control target location. It has node kind Label.

The expression points to a LabelTarget. GotoExpressions refer to the same target object to designate that they jump to this LabelExpression location in the ET. The LabelTarget enables the ET to be acyclic by avoiding having Goto nodes point directly to Label nodes. Rewriting ETs behaves better regarding with LabelExpression identity when the LabelTarget object is distinct due to changes not propagating as far and forcing more rewrites via GotoExpressions.

The target has a Type property because Goto's can transfer control to a location with a value. The type allows factory methods and the ET compiler to verify static typing intent within the ET. See section 2.4 for more information.

A Goto can optionally deliver a value to the LabelExpression's location. In case execution flows through the label in a structured way (not via a jump), it has a DefaltValue expression that provides the result of the LabelExpression. The label's location is AFTER the DefaultValue expression. If an unstructured flow of control lands at this LabelExpression's location, the GotoExpression provides the result for the LabelExpression.

The LabelExpression.Type property gets its value from Target.Type. Any GotoExpression that references the same Target must have a Type property that represents a type that is reference assignable to the type represented by Target.Type. There are two exceptions, see the factory method documentation for an explanation.

See section 4.4 for more details on the semantics of GotoExpression and LabelExpression, as well as the introductory section on iterations, lexical exits, and gotos (2.4).

4.29.1 Example

This shows how to use a label to exit a lambda and using the label's default expression. Often the LabelExpression will be the Body of the lambda, making the LabelExpression's DefaultValue effectively serve as the lamba's Body. However, this example is the best rendering of the code snippet. You could of course have written the ET purely expression-based (no return's) where the lambda's Body was a ConditionExpression with a ConstantExpression for the consequence and alternative expressions.

```
// Simple return example:
//   (int x) => {
//     if (x < 0) return -1;
//     return 1;
//   }
var x = Expression.Parameter(typeof(int), "x");
// Get LabelTarget
var r = Expression.Label(typeof(int));
var lambda = Expression.Lambda<Func<int, int>>(
    Expression.Block(
        Expression.IfThen(
            Expression.LessThan(x, Expression.Constant(0)),
            Expression.Return(r, Expression.Constant(-1))),
        // Add LabelExpr to define label target location and add
        // default value if fall through to label target.
        Expression.Label(r, Expression.Constant(1))),
    x
);
```

4.29.2 Class Summary

```
public sealed class LabelExpression : Expression {
    public Expression DefaultValue { get; }
    public LabelTarget Target { get; }
    public LabelExpression Update(LabelTarget target,
                                  Expression defaultValue)
```

4.29.3 DefaultValue Property

This property returns the default expression that provides the result of the LabelExpressions. If a GotoExpression transfers control to the Target, then it provides an expression that provides the result for the LabelExpression. If this property is null, then this object's Type property represents the void type.

Signature:

```
    public Expression DefaultValue { get; }
```

4.29.4 Target Property

This property returns the target identity for the LabelExpression. GotoExpressions refer to the same target object to designate that they jump to this LabelExpression location in the ET.

Signature:

```
    public LabelTarget Target { get; }
```

4.29.5 Update Method

This method creates a new expression that is like this one, but using the supplied children. If all of the children are the same, this method returns this expression instead of creating a new one. This is useful when implementing ExpressionVisitors to rewrite the current node only when you need to do so.

Signature:

```
    public LabelExpression Update(LabelTarget target,
                                  Expression defaultValue)
```

4.29.6 Factory Methods

The Expression class has the following factory methods for creating LabelExpressions:

```
    public static LabelExpression Label(LabelTarget target);
    public static LabelExpression Label(LabelTarget target,
                                        Expression defaultValue);
```

Target identifies the LabelExpression so that a GotoExpressions can refer to the same target object to designate that it jumps to this LabelExpression location in the ET.

DefaultValue, if supplied is an expression to execute, and its value is the result of the LabelExpression if execution reaches the label without jumping. The defaultValue's Type property must represent a type that is reference-assignable to the type represented by the target's Type property.

There are two special cases the factory handles when defaultValue's Type property represents a type that is not assignable to the type represented by target.Type. First, if Target.Type is void, the defaultValue.Type can represent anything since the ET compiler will automatically convert the result to void or squelch the value. Second, if target's type is a sub type of LambdaExpression, and the defaultValue Expression object itself (that is, the Expression node) is of a type that is assignable to the target's type, then the value Expression node is wrapped in a Quote node. This supports a legacy ETs v1 behavior for how C# chose to implement expression such as "Expression<Func<...>> = (...) => ...".

4.30 LabelTarget Class

This class represents the identity of a LabelExpression to which GotoExpressions refer that want to jump to that LabelExpression. The LabelTarget enables the ET to be acyclic by avoiding having Goto nodes point directly to Label nodes. Rewriting ETs behaves better regarding with LabelExpression identity when the LabelTarget object is distinct due to changes not propagating as far and forcing more rewrites via GotoExpressions.

The target has a Type property because Goto's can transfer control to a location with a value. The type allows factory methods and the ET compiler to verify static typing intent within the ET. See section 2.4 for more information.

4.30.1 Class Summary

```
public sealed class LabelTarget {
    public String Name { get; }
    public Type Type { get; }
```

4.30.2 Name Property

This property returns the name of the label. This is useful purely for debugging or pretty printing purposes since it has no semantic effects.

Signature:

```
    public String Name { get; }
```

4.30.3 Type Property

This property returns the type expected of any value delivered to the label's location of control flow. Any GotoExpression.Expression that targets this LabelTarget must have a Type property that represents a type that is reference assignable to the type represented by this Type property. However, if Target.Type is void, the GotoExpression.Expression.Type can represent anything since the ET compiler will automatically convert the result to void or squelch the value.

Signature:

```
    public Type Type { get; }
```

4.30.4 Factory Methods

The Expression class has the following factory methods for creating LabelTargets:

```
    public static LabelTarget Label(Type type, String name);
```

```
public static LabelTarget Label(Type type);
public static LabelTarget Label();
public static LabelTarget Label(String name);
```

4.31 MemberExpression Class

This class represents accessing an instance member, property or field, of an object or a static member of a type. It can be used as the Left expression of a BinaryExpression with node kind Assign.

4.31.1 Class Summary

```
public sealed class MemberExpression : Expression {
    public Expression Expression { get; }
    public MemberInfo Member { get; }
    public MemberExpression Update(Expression expression)
```

4.31.2 Expression Property

This property returns the Expression representing the object or type on which to access Member.

Signature:

```
public Expression Expression { get; }
```

4.31.3 Member Property

This property return the Expression representing the member to access on Expression. This is never null.

Signature:

```
public MemberInfo Member { get; }
```

4.31.4 Update Method

This method creates a new expression that is like this one, but using the supplied children. If all of the children are the same, this method returns this expression instead of creating a new one. This is useful when implementing ExpressionVisitors to rewrite the current node only when you need to do so.

Signature:

```
public MemberExpression Update(Expression expression)
```

4.31.5 Factory Methods

Expression has the following factory methods for MemberExpressions:

```
public static MemberExpression Field(Expression expression,
                                     String fieldName);
public static MemberExpression Field
    (Expression expression, Type type, String fieldName);
public static MemberExpression Field(Expression expression,
                                     FieldInfo field);
```

```
public static MemberExpression MakeMemberAccess
    (Expression expression, MemberInfo member);

public static MemberExpression Property(Expression expression,
                                        PropertyInfo property);
public static MemberExpression Property
    (Expression expression, MethodInfo propertyAccessor);
public static MemberExpression Property(Expression expression,
                                        String propertyName);
public static MemberExpression Property
    (Expression expression, Type type, String propertyName);
public static MemberExpression PropertyOrField
    (Expression expression, String propertyOrFieldName);
```

The following is derived from the v1 spec ... with updates for correctness or new behaviors

Field, property, and propertyAccessor must be non-null. PropertyAccessor must represent a property accessor method. If the member represented by field, property, or propertyAccessor are instance members, expression must be non-null. Expression's Type property must be assignable to the declaring type of the member represented by field, property, or propertyAccessor.

For the methods taking a string and performing name resolution, expression, fieldName, propertyName, and propertyOrFieldName must be non-null. These factory methods search through expression.Type and its base types for fields, properties, or both. The found members have the name fieldName, propertyName, or propertyOrFieldName, respectively. These factories prefer Public members over non-public. Additionally, given PropertyOrField, the factory prefers properties over fields. If there is not exactly one result found that matches, these factories throw an exception. Otherwise, these factories pass expression and the found fieldInfo or propertyInfo to the Field or Property factory method above.

For MakeMemberAccess, member must be non-null. Based on the type of member, this factory calls one of the other factory methods with expression and member as arguments. All the requirements and guarantees of the called factory method apply.

The returned MemberExpression has

- Node kind MemberAccess
- Expression property set to expression
- Member property set to field, property, or the property referred to by propertyAccessor
- Type set to the FieldType or PropertyType property

4.32 MemberInitExpression Class

This class represents instantiating a type and filling in members before returning the instance. Its node kind is MemberInit. For example,

```
// new MyClass { Foo = “hello”, Bar = “world” } becomes:
MyClass obj = new MyClass();
obj.Foo = “hello”;
obj.Bar = “world”;
```

This node reduces to a block that instantiates the type and sets the supplied members. This node type is a ETs v1 type that, even with blocks and assignments in ETs v2, is still useful for convenience and meta-programming uses.

This node uses sub types of MemberBinding as helper model classes for individual member values and what methods to use to add the element.

4.32.1 Class Summary

```
public sealed class MemberInitExpression : Expression {
    public ReadOnlyCollection<MemberBinding>
           Bindings { get; }
    public NewExpression NewExpression { get; }
    public MemberInitExpression Update
        (NewExpression newExpression,
        IEnumerable<MemberBinding> bindings)
```

4.32.2 Bindings Property

This property returns the collection of MemberBindings that represent how to fill in the members of the new instance that this node returns.

Signature:

```
    public ReadOnlyCollection<MemberBinding>
           Bindings { get; }
```

4.32.3 NewExpression Property

This property returns a NewExpression whose semantics is to create an instance of the type represented by the node's Type property. The resulting object returned when this expression executes has its instance members filled in as described by its MemberBindings.

Signature:

```
    public NewExpression NewExpression { get; }
```

4.32.4 Update Method

This method creates a new expression that is like this one, but using the supplied children. If all of the children are the same, this method returns this expression instead of creating a new one. This is useful when implementing ExpressionVisitors to rewrite the current node only when you need to do so.

Signature:

```
    public MemberInitExpression Update
        (NewExpression newExpression,
        IEnumerable<MemberBinding> bindings)
```

4.32.5 Factory Methods

Expression has the following factory methods for MemberInitExpressions:

```
    public static MemberInitExpression MemberInit
        (NewExpression newExpression, params MemberBinding[] bindings);
```

```
public static MemberInitExpression MemberInit
    (NewExpression newExpression,
     IEnumerable<MemberBinding> bindings);
```

The following is derived from the v1 spec …

NewExpression and bindings must be non-null. For each element of bindings, the Member property must represent a member of the type represented by newExpression.Type.

The resulting MemberInitExpression has the NewExpression property equal to newExpression, and Bindings has the same elements as bindings. The Type property is equal to newExpression.Type.

4.33 MemberBinding Class

This class is the base class for representing member initialization expressions within MemberInitExpressions. This is a supporting type, not an Expression. Each subtype of this type (MemberListBinding, MemberAssignment, and MemberMemberBinding) has a unique kind enum value to support switching on the binding kinds in languages that do not have type switch expressions.

No one should derive from this class, and the constructor will be made obsolete in future versions.

4.33.1 Class Summary

```
public abstract class MemberBinding {
    // This constructor is now obsolete in spec.  It will be obsolete
in
    // code in v-next+1, then removed in v-next+2.
    protected MemberBinding(MemberBindingType type, MemberInfo member);
    public MemberBindingType BindingType { get; }
    public MemberInfo Member { get; }
```

4.33.2 MemberBinding Constructor

No one should derive from this class, and the constructor will be made obsolete in

v-next+1, then removed in v-next+2.

4.33.3 BindingType Property

This property returns the kind of binding this object represents, which is one to one with the concrete sub type.

```
public MemberBindingType BindingType { get; }
```

4.33.4 Member Property

This property returns the method that sets the member. It is never null.

Signature:

```
public MemberInfo Member { get; }
```

4.34 MemberBindingType Enum

This enum provides member for tagging MemberBinding sub types. Each MemberBinding subtype has a unique MemberBindingType value to support switching on the binding kinds in languages that do not have type switch expressions.

4.34.1 Type Summary

```
public enum MemberBindingType {
    Assignment,
    MemberBinding,
    ListBinding
```

4.34.2 Type Members

The Summary section shows the type as it is defined (to indicate values of members), and this section documents the intent of the members.

`Assignment`	The MemberBinding is an instance of MemberAssignment. This models setting a member to a scalar value, for example, the setting of Foo in: new Blah {Foo = 5, ...}
`MemberBinding`	The MemberBinding is an instance of MemberMemberBinding. This models setting a member to a new instance of a type with the supplied member values, for example, the setting of Foo in: new Blah {Foo = {Bar = ..., Baz = ...}, ...}
`ListBinding`	The MemberBinding is an instance of MemberListBinding. This models setting a member to a list of values, for example, the setting of Foo in: new Blah {Foo = {1, 2, 3, ...}, ...}

4.35 MemberListBinding Class

This class represents the setting of an instance member to a list of elements. This is a supporting type used in MemberInitExpressions. For example:

```
New Foo { bar = {1, 2, 3}}
```

4.35.1 Class Summary

```
public sealed class MemberListBinding : MemberBinding {
    public ReadOnlyCollection<ElementInit>
           Initializers { get; }
```

```
public MemberListBinding Update
    (IEnumerable<ElementInit> initializers)
```

4.35.2 Initializers Property

This property returns the collection of ElementInit model objects for what values to add to the member collection and how to add each value.

Signature:

```
public ReadOnlyCollection<ElementInit>
       Initializers { get; }
```

4.35.3 Update Method

This method creates a new expression that is like this one, but using the supplied children. If all of the children are the same, this method returns this expression instead of creating a new one. This is useful when implementing ExpressionVisitors to rewrite the current node only when you need to do so.

Signature:

```
public MemberListBinding Update
    (IEnumerable<ElementInit> initializers)
```

4.35.4 Factory Methods

Expression has the following factory methods for MemberMemberBindings:

```
public static MemberListBinding ListBind
    (MemberInfo member,
     IEnumerable<ElementInit> initializers);
public static MemberListBinding ListBind
    (MemberInfo member, params ElementInit[] initializers);
public static MemberListBinding ListBind
    (MethodInfo propertyAccessor,
     IEnumerable<ElementInit> initializers);
public static MemberListBinding ListBind
    (MethodInfo propertyAccessor,
     params ElementInit[] initializers);
```

The following is derived from the v1 spec ...

Member must be non-null, and must represent a field or property. Let T represent a FieldType or PropertyType for member. PropertyAccessor must represent a property accessor method. Let T also represent the property's type. T must be assignable to IEnumerable. Each method in ElementInits must be callable on instances of T for adding elements to the member.

4.36 MemberMemberBinding Class

This class represents the recursive initializing of one object's member with the creation of another object and setting the second object's members. This is a supporting type used in MemberInitExpressions. For example, the setting of Foo in:

```
new Blah {Foo = {Bar  = ..., Baz = ...}, ...}
```

4.36.1 Class Summary

```
public sealed class MemberMemberBinding : MemberBinding {
    public ReadOnlyCollection<MemberBinding>
           Bindings { get; }
    public MemberMemberBinding Update
        (IEnumerable<MemberBinding> bindings)
```

4.36.2 Bindings Property

This property returns the collection of MemberBindings that describe how to initialize members of an object. The object is the initialization value a member who MemberInitExpression recursively contains this MemberMemberBinding model object. This property is never null.

4.36.3 Update Method

This method creates a new expression that is like this one, but using the supplied children. If all of the children are the same, this method returns this expression instead of creating a new one. This is useful when implementing ExpressionVisitors to rewrite the current node only when you need to do so.

Signature:

```
public MemberMemberBinding Update
    (IEnumerable<MemberBinding> bindings)
```

4.36.4 Factory Methods

Expression has the following factory methods for MemberMemberBindings:

```
public static MemberMemberBinding MemberBind
    (MemberInfo member,
     IEnumerable<MemberBinding> bindings);
public static MemberMemberBinding MemberBind
    (MemberInfo member, params MemberBinding[] bindings);
public static MemberMemberBinding MemberBind
    (MethodInfo propertyAccessor,
     IEnumerable<MemberBinding> bindings);
public static MemberMemberBinding MemberBind
    (MethodInfo propertyAccessor, params MemberBinding[] bindings);
```

The following is derived from the v1 spec ...

Member must be non-null, and must represent a field or property. Let T be the FieldType or PropertyType. PropertyAccessor must represent a property accessor method of a property. Let T also be the property's PropertyType. Bindings must be non-null, and for each element of bindings, the Member property must represent a member of T.

4.37 MemberAssignment Class

This class models assigning a scalar value to a MemberInitExpression's member, for example, the setting of Foo in:

```
new Blah {Foo = 5, ...}
```

4.37.1 Class Summary

```
public sealed class MemberAssignment : MemberBinding {
    public Expression Expression { get; }
    public MemberAssignment Update(Expression expression)
```

4.37.2 Expression Property

This property returns the Expression that models the value to assign to the MemberInitExpression member.

Signature:

```
    public Expression Expression { get; }
```

4.37.3 Update Method

This method creates a new expression that is like this one, but using the supplied children. If all of the children are the same, this method returns this expression instead of creating a new one. This is useful when implementing ExpressionVisitors to rewrite the current node only when you need to do so.

Signature:

```
    public MemberAssignment Update(Expression expression)
```

4.37.4 Factory Methods

Expression has the following factory methods for MemberAssignments:

```
    public static MemberAssignment Bind(MethodInfo propertyAccessor,
                                        Expression expression);
    public static MemberAssignment Bind(MemberInfo member,
                                        Expression expression);
```

The following is derived from the v1 spec … with additions on binding to generic properties.

Member must be non-null, and must represent a field or property. PropertyAccessor must represent a property accessor method. Expression must be non-null, and its Type property must be assignable to the FieldType or PropertyType of the field or property represented by member or propertyAccessor.

You cannot simply bind to generic base type properties. For example, the following code does not work:

```
public class OrderBase<T> {
    public T OrderName { get; set; }
}

public class Order : OrderBase<string> {
    public string OrderID { get; set; }
}

class Program {
    static void Main(string[] args) {
        // Get handles for the setter method and the Order classs
        Type orderType = typeof(Order);
```

```
            RuntimeTypeHandle orderHandle = orderType.TypeHandle;
            PropertyInfo nameProp = orderType.GetProperty("OrderName");
            MethodInfo nameMethod = nameProp.GetSetMethod();
            RuntimeMethodHandle nameHandle = nameMethod.MethodHandle;

            // Now get a MethodInfo for the setter method
            MethodInfo nameMethodFromHandle =
                        (MethodInfo)MethodBase
                                    .GetMethodFromHandle(nameHandle,
                                                         orderHandle);
            ConstantExpression constant = Expression.Constant("Ben");
            // Attempt to bind.  Fails with:
            // System.ArgumentException: The method
            // 'TestBinding.OrderBase`1[System.String].set_OrderName'
            // is not a property accessor
            MemberAssignment binding =
                        Expression.Bind(nameMethodFromHandle, constant);
        }
    }
```

The Bind factory method ultimately fetches the DeclaringType from the MethodInfo, which doesn't work in this case. Bind fetches this so that it can get the PropertyInfo, which this node holds onto as convenience to tree walkers and consumers. You can construct the PropertyInfo for a generic base class by tweaking some of the DLR's internal code and calling it as a helper:

```
    MemberBinding binding =
        Expression.Bind(GetProperty(propertyAccessor,
                                    Type.GetTypeFromHandle(
                                             edmProperty
                                             .PropertyDeclaringType)),
                        valueReader);

    private static PropertyInfo GetProperty(MethodInfo setterMethod,
                                            Type declaringType) {
        BindingFlags bindingAttr = BindingFlags.NonPublic |
                                   BindingFlags.Public |
                                   BindingFlags.Instance;
        foreach (PropertyInfo propertyInfo in
                 declaringType.GetProperties(bindingAttr)) {
            if (propertyInfo.GetSetMethod(nonPublic: true) ==
                setterMethod) {
                return propertyInfo;
            }
        }
    }
```

4.38 ListInitExpression Class

This class represents a collection construction and initialization. It has the ListInit node kind. You can use this node to create any type that supports an "add" method (case-insensitive). For example,

```
// new MyList { “hello”, “world” } becomes:
MyList list = new MyList();
list.Add(“hello”);
```

list.Add(“world”);

This node reduces to a block that instantiates the type and adds the supplied elements. This node type is a ETs v1 type that, even with blocks and assignments in ETs v2, is still useful for convenience and meta-programming uses.

This node uses ElementInit as a helper model class for individual element values and what method to use to add the element.

4.38.1 Class Summary

```
public sealed class ListInitExpression : Expression {
    public ReadOnlyCollection<ElementInit>
           Initializers { get; }
    public NewExpression NewExpression { get; }
    public ListInitExpression Update
        (NewExpression newExpression,
         IEnumerable<ElementInit> initializers)
```

4.38.2 Initializers Property

This property returns a read-only collection of ElementInit objects describing each element and how to add it to the new instance of this node's Type.

Signature:

```
    public ReadOnlyCollection<ElementInit>
           Initializers { get; }
```

4.38.3 NewExpression Property

This property returns the NewExpression that creates an instance of this node's Type.

Signature:

```
    public NewExpression NewExpression { get; }
```

4.38.4 Update Method

This method creates a new expression that is like this one, but using the supplied children. If all of the children are the same, this method returns this expression instead of creating a new one. This is useful when implementing ExpressionVisitors to rewrite the current node only when you need to do so.

Signature:

```
    public ListInitExpression Update
        (NewExpression newExpression,
         IEnumerable<ElementInit> initializers)
```

4.38.5 Factory Methods

Expression has the following factory methods for ListInitExpressions:

```
    public static ListInitExpression ListInit
        (NewExpression newExpression,
         params ElementInit[] initializers);
```

```
public static ListInitExpression ListInit
     (NewExpression newExpression, MethodInfo addMethod,
      params Expression[] initializers);
public static ListInitExpression ListInit
     (NewExpression newExpression,
      IEnumerable<ElementInit> initializers);
public static ListInitExpression ListInit
     (NewExpression newExpression,
      IEnumerable<Expression> initializers);
public static ListInitExpression ListInit
     (NewExpression newExpression,
      params Expression[] initializers);
public static ListInitExpression ListInit
     (NewExpression newExpression, MethodInfo addMethod,
      IEnumerable<Expression> initializers);
```

The following is derived from the v1 spec … with updates for correctness or new behaviors

NewExpression and initializers must be non-null. The type represented by newExpression.Type must implement System.Collections.IEnumerable. If a non-null addMethod is given, it must represent an instance method named "Add" with exactly one parameter. The Type property of all elements of initializers must represent a type that is assignable to the parameter type of addMethod.

The resulting ListInitExpression has:

- Node kind set to ListInit
- NewExpression set to newExpression
- Type set to newExpression.Type
- If initializers is given as an IEnumerable<ElementInit> or an ElementInit[], then the Initializers property is simply equal to initializers. Otherwise, this factory determines an addMethod to use and creates ElementInit objects for a collection of initializers as specified below.

If you supply a non-null addMethod, then that is the add method of the ListInitExpression. Otherwise, this factory searches for a single-argument instance method with a name equal to "Add" (ignoring case) on newExpression.Type and its base type. If exactly one method is found, that is the add method; otherwise, this factory throws an exception.

The Initializer property is then a list of ElementInit objects, one for each element of initializers, with the AddMethod property being the add method, and the Arguments property being a ReadOnlyCollection<Expression> containing the corresponding element of initializers as its single element.

4.39 ElementInit Class

This class represents elements to be added to a new instance of a type in the ListInitExpression and MemberListBinding classes. This is a supporting type used in some expression nodes, and it is not an expression.

4.39.1 Class Summary

```
public sealed class ElementInit {
```

```
public MethodInfo AddMethod { get; }
public ReadOnlyCollection<Expression> Arguments { get; }
public ElementInit Update(IEnumerable<Expression> arguments)
```

4.39.2 AddMethod Property

This property returns the method that will be used to add an element to an object whose intantiation and initialization are modeled in a ListInitExpression or MemberListBinding.

Signature:

```
public MethodInfo AddMethod { get; }
```

4.39.3 Arguments Property

This property returns the collection of argument expressions for this object's AddMethod.

Signature:

```
public ReadOnlyCollection<Expression> Arguments { get; }
```

4.39.4 Update Method

This method creates a new expression that is like this one, but using the supplied children. If all of the children are the same, this method returns this expression instead of creating a new one. This is useful when implementing ExpressionVisitors to rewrite the current node only when you need to do so.

Signature:

```
public ElementInit Update(IEnumerable<Expression> arguments)
```

4.39.5 Factory Methods

Expression has the following factories for ElementInit objects:

```
public static ElementInit ElementInit
    (MethodInfo addMethod,
     IEnumerable<Expression> arguments);
public static ElementInit ElementInit
    (MethodInfo addMethod, params Expression[] arguments);
```

The following is derived from the v1 spec ... with updates for correctness or new behaviors

AddMethod and arguments must be non-null. AddMethod must represent an instance method named “Add” (ignoring case). Arguments must have the same number of elements as the number of parameters the method represented by addMethod takes. Each of the elements of arguments must be non-null, and the types of the values they represent must be assignable to the type of the corresponding parameter of addMethod. There is a special case the factory handles when an element of arguments has a Type property representing a type that is not assignable to the corresponding parameter type. If the parameter's type is a sub type of LambdaExpression, and the argument Expression object itself (that is, the Expression node) is of a type that is assignable to the parameter's type, then the argument Expression node is wrapped in a Quote node. This supports a legacy ETs v1 behavior for how C# chose to implement expression such as "Expression<Func<...>> = (...) => ...".

The resulting ElementInit has the AddMethod property equal to addMethod and the Arguments property equal to arguments.

4.40 NewExpression Class

This class represents calling a constructor to instantiate a type. These nodes have the New node kind. These are distinct from MethodCallExpression for a couple of reasons. One is that object instantiation is often a distinct linguistic feature as opposed to being functionality that naturally fits other features like regular function calls. This distinction is worth explicitly representing for meta-programming purposes. The second reason is that in .NET ConstructorInfos are not the same as MethodInfos requiring a distinct type.

The NewExpression uses the New node kind. It represents calling a constructor to create a new object. Given a New node, exp, let T be the C# name of the declaring type of exp.Constructor, and let e1…en be the comma-separated list of C# expressions equivalent to the corresponding nodes in exp.Arguments. Then the C# equivalent of exp is "new T(e1…en)".

4.40.1 Class Summary

```
public class NewExpression : Expression {
    public ReadOnlyCollection<Expression>
           Arguments { get; }
    public ConstructorInfo Constructor { get; }
    public ReadOnlyCollection<System.Reflection.MemberInfo>
           Members { get; }
    public NewExpression Update(IEnumerable<Expression> arguments)
```

4.40.2 Arguments Property

This returns the arguments to the constructor invocation. This never returns null, returning an empty collection for the default constructor.

Signature:

```
    public ReadOnlyCollection<Expression>
           Arguments { get; }
```

4.40.3 Constructor Property

This returns the ConstructorInfo for the constructor. This never returns null.

Signature:

```
    public ConstructorInfo Constructor { get; }
```

4.40.4 Members Property

When constructing an anonymous type, this returns MemberInfos describing the members of the type to construct. For example, the C# expression “new { Foo = x, Bar = y }” could be represented by a NewExpression with Members having the PropertyInfos for “Foo” and “Bar”.

This never returns null, returning an empty collection for non-anonymous types.

Signature:

```
public ReadOnlyCollection<MemberInfo>
       Members { get; }
```

4.40.5 Update Method

This method creates a new expression that is like this one, but using the supplied children. If all of the children are the same, this method returns this expression instead of creating a new one. This is useful when implementing ExpressionVisitors to rewrite the current node only when you need to do so.

Signature:

```
public NewExpression Update(IEnumerable<Expression> arguments)
```

4.40.6 Factory Methods

Expression has the following factory methods for creating NewExpressions:

```
public static NewExpression New
    (ConstructorInfo constructor,
     IEnumerable<Expression> arguments,
     params MemberInfo[] members);
public static NewExpression New(ConstructorInfo constructor);
public static NewExpression New(Type type);
public static NewExpression New(ConstructorInfo constructor,
                                params Expression[] arguments);
public static NewExpression New
    (ConstructorInfo constructor,
     IEnumerable<Expression> arguments);
public static NewExpression New
    (ConstructorInfo constructor,
     IEnumerable<Expression> arguments,
     IEnumerable<MemberInfo> members);
```

The following is derived from the v1 spec … with updates for correctness or new behaviors

Constructor and type must be non-null. Type must represent a type that has a constructor that takes no arguments.

If arguments is omitted or null, the factory stores an empty collection. If provided, arguments must have the same number of elements as the number of parameters for the constructor. Each of the elements of arguments must be non-null, and the types of the values they represent must be assignable to the type of the corresponding parameter of addMethod. There is a special case the factory handles when an element of arguments has a Type property representing a type that is not assignable to the corresponding parameter type. If the parameter's type is a sub type of LambdaExpression, and the argument Expression object itself (that is, the Expression node) is of a type that is assignable to the parameter's type, then the argument Expression node is wrapped in a Quote node. This supports a legacy ETs v1 behavior for how C# chose to implement expression such as "Expression<Func<...>> = (...) => ...".

If provided, members must have the same number of elements as arguments. Each of the elements of members must be non-null. Each must be a gettable instance PropertyInfo, FieldInfo, or MethodInfo that is available on the declaring type of constructor. You might expect the members to be setters, but these exist purely to capture the names of members, for pretty

printing, or for debugging uses. They do not affect the semantics or results of NewExpression nodes. The corresponding element of arguments for a particular members element must have a type assignable to the type of the member.

The resulting NewExpression has:

- Node kind New
- Constructor property set to constructor, or the constructor of the type that takes no arguments.
- Arguments property set to arguments, except that some elements may be “quoted” as described above. If arguments is omitted or null, Arguments is an empty collection.
- Members property set to the same elements as members. If members is omitted or null, Members is an empty collection.
- Type property set to the declaring type of the constructor denoted by constructor, or set to the type argument.

Note that when member infos are supplied we map any get_ or set_ member infos to prop infos, and this is a breaking change from v1.

4.41 NewArrayExpression Class

This class represents creating a new array. It uses the NewArrayInit and NewArrayBounds node kinds. NewArrayInit represents making a one-dimensional array by specifying a list of elements, for example, in C# "new T[]{e1…en}". NewArrayBounds represents making a new array by specifying its bounds for each dimension, for example, in C# "new T[e1…en]".

4.41.1 Class Summary

```
public class NewArrayExpression : Expression {
    public ReadOnlyCollection<Expression>
           Expressions { get; }
    public NewArrayExpression Update
        (IEnumerable<Expression> expressions)
```

4.41.2 Expressions Property

This property returns the collection of Expressions that provide values for initializing a single dimensional array or the integer bounds for each dimension of an array, depending on the node kind.

Signature:

```
    public ReadOnlyCollection<Expression>
           Expressions { get; }
```

4.41.3 Update Method

This method creates a new expression that is like this one, but using the supplied children. If all of the children are the same, this method returns this expression instead of creating a new one.

This is useful when implementing ExpressionVisitors to rewrite the current node only when you need to do so.

Signature:

```
public NewArrayExpression Update
    (IEnumerable<Expression> expressions)
```

4.41.4 Factory Methods

Expression has the following factory methods for NewArrayExpression nodes:

```
public static NewArrayExpression NewArrayBounds
    (Type type,
     IEnumerable<Expression> bounds);
public static NewArrayExpression NewArrayBounds
    (Type type, params Expression[] bounds);

public static NewArrayExpression NewArrayInit
    (Type type,
     IEnumerable<Expression> initializers);
public static NewArrayExpression NewArrayInit
    (Type type, params Expression[] initializers);
```

The following is derived from the v1 spec … with updates for correctness or new behaviors

For the …Bounds factories Type and bounds must be non-null. Type represents the element type. Each element of bounds must be non-null, and its Type property must represent an integral type. The resulting NewArrayExpression has node kind NewArrayBounds. The Type property represents an array type with rank equal to the length of bounds and the element type represented by type. The Expressions property has the same elements as bounds.

For the …Init factories type and initializers must be non-null. Type represents the element type. Each element of initializers must be non-null and have a Type property that represents a type assignable to the type represented by type. There is a special case the factory handles when an element of initializers has a Type property representing a type that is not assignable to type. If the initializer's type is a sub type of LambdaExpression, and the element Expression object itself (that is, the Expression node) is of a type that is assignable to type, then the element Expression node is wrapped in a Quote node. This supports a legacy ETs v1 behavior for how C# chose to implement expression such as "Expression<Func<...>> = (...) => ...".

The resulting NewArrayExpression has node kind NewArrayInit. The Type property represents an array type with rank 1 and the element type represented by type. The Expressions property has the same elements as initializers.

4.42 LambdaExpression Class

This abstract class represents a function definition, and it always manifests as Expression<T>. Evaluating a LambdaExpression produces a delegate. It has node kind Lambda.

The LambdaExpression may be in the context of (that is, it is a sub ET of) other lambdas or BlockExpressions. When this occurs, and the inner LambdaExpression contains

ParameterExpression nodes that refer to variables defined by outer lambdas or blocks, the expression compiler creates closure environments when needed, lifting variables.

4.42.1 Examples

Iterative Fact:

```
static Func<int, int> ETFact() {
    var value = Expression.Parameter(typeof(int), "value");
    var result = Expression.Parameter(typeof(int), "result");
    var label = Expression.Label(typeof(int));
    var lambda = Expression.Lambda<Func<int, int>>(
        Expression.Block(
            new[] { result },
            Expression.Assign(result, Expression.Constant(1)),
            Expression.Loop(
                Expression.IfThenElse(
                    Expression.GreaterThan(value,
                                           Expression.Constant(1)),
                    Expression.MultiplyAssign
                       (result,
                        Expression.PostDecrementAssign(value)),
                        Expression.Break(label, result)),
                label)),
        value);
    return lambda.Compile();
```

Recursive Fact with StrongBox to refer to function recursively:

```
using System;
using System.Linq.Expressions;
using System.Runtime.CompilerServices;

namespace LambdaRecursion
{
    class Program
    {
        static void Main(string[] args)
        {
            // fact = (x) => x <= 1 ? 1 : x*fact(x-1)
            var x = Expression.Parameter(typeof(int), "x");
            var factorial = new StrongBox<Func<int, int>>();
            var lambda = Expression.Lambda<Func<int, int>>(
                Expression.Condition(
                    Expression.LessThanOrEqual
                        (x, Expression.Constant(1)),
                    Expression.Constant(1),
                    Expression.Multiply(
                        x,
                        Expression.Invoke(
                            Expression.Field
                               (Expression.Constant(factorial),
                                "Value"),
                            Expression.Subtract
                               (x, Expression.Constant(1))))),
                x);
```

```
factorial.Value = lambda.Compile();
Console.WriteLine(factorial.Value(5));
```

Recursive Fact just using ParameterExpression variables for recursive reference:

```
ParameterExpression input = Expression.Parameter(typeof(int));
var test = Expression.GreaterThan(input, Expression.Constant(1));
var factorial = Expression.Parameter(typeof(Func));
Expression body = Expression.Block(
                    Expression.Condition(
                        test,
                        Expression.Multiply(
                            input,
                            Expression.Invoke(
                                factorial,
                                Expression.Subtract(
                                    input,
                                    Expression.Constant(1)))),
                        Expression.Constant(1)));
var fact = Expression.Lambda>(body, input);
var block = Expression.Block(
                new ParameterExpression[] { factorial },
                Expression.Assign(factorial, fact),
                Expression.Invoke(factorial,
Expression.Constant(5)));
Expression.Lambda(block).Compile()();
```

4.42.2 Class Summary

```
public abstract class LambdaExpression : Expression {
    public Expression Body { get; }
    public String Name { get; }
    public ReadOnlyCollection
               <ParameterExpression>
           Parameters { get; }
    public Type ReturnType { get; }
    public void TailCall { get; }
    public Delegate Compile();
    public Delegate Compile(DebugInfoGenerator debugInfoGenerator)
    public void CompileToMethod(MethodBuilder method);
    public void CompileToMethod(MethodBuilder method,
                                DebugInfoGenerator debugInfoGenerator);
    public Expression<TDelegate> Update
       (Expression body, IEnumerable<ParameterExpression> parameters)
```

4.42.3 Body Property

This property returns the body expression of the lambda. It is never null.

This node's Body.Type is reference assignable to ReturnType with one exception. ReturnType may be void when Body.Type is not, for example:

```
static int NonVoidMethod() {
    return 123;
}
```

```
static void Main(string[] args) {
    Expression<Action> e = () => NonVoidMethod();
}
```

For convenience in this case, the types do not have to match, and the lambda will automatically convert the result to void or squelch it.

Signature:

```
public Expression Body { get; }
```

4.42.4 Name Property

This property returns the name of the lambda, which is used for debugging or pretty printing only. It has no semantic value. It may be null.

Signature:

```
public String Name { get; }
```

4.42.5 Parameters Property

This property returns the collection of ParameterExpressions that declare parameter variables for the lambda. The number and types are consistent with the delegate type in the Type property for a LambdaExpression.

Signature:

```
public ReadOnlyCollection
           <ParameterExpression>
       Parameters { get; }
```

4.42.6 ReturnType Property

This property returns the type that is the return type of the delegate type in this node's Type property. This node's Body.Type is reference assignable to ReturnType with one exception. ReturnType may be void when Body.Type is not, for example:

```
static int NonVoidMethod() {
    return 123;
}
static void Main(string[] args) {
    Expression<Action> e = () => NonVoidMethod();
}
```

For convenience in this case, the types do not have to match, and the lambda will automatically convert the result to void or squelch it.

4.42.7 TailCall Property

This property returns whether compiling this LambdaExpression should attempt to use tail calls for any value returning expressions. If true, this is not a guarantee since some calls cannot be tailed called (for example, there may be write backs needed for some properties or ref args).

Signature:

```
public void TailCall { get; }
```

4.42.8 Compile* Methods

These methods compile the LambdaExpression instance. Compile returns a delegate that you can immediately invoke to execute the ET. The delegate has the type represented by this nodes Type property.

DebugInfoGenerator, if supplied, indicates the compiler should emit sequence point and local variable information. It also implements members the compiler uses to emit this information.

Compiling an Expression<D> which contains calls of unsafe code may cause an exception to get thrown.

CompileToMethod is useful for hosted languages that want to provide some pre-compilation support so that their host applications do not have to read source code and compile it on each start up. It is up to the language using this method to prepare the assembly and write it to disk.

Signatures:

```
public Delegate Compile();
public Delegate Compile(DebugInfoGenerator debugInfoGenerator)
public void CompileToMethod(MethodBuilder method);
public void CompileToMethod(MethodBuilder method,
                            DebugInfoGenerator debugInfoGenerator);
```

4.42.9 Update Method

This method creates a new expression that is like this one, but using the supplied children. If all of the children are the same, this method returns this expression instead of creating a new one. This is useful when implementing ExpressionVisitors to rewrite the current node only when you need to do so.

Signature:

```
public Expression<TDelegate> Update
   (Expression body, IEnumerable<ParameterExpression> parameters)
```

4.42.10 Factory Methods

Expression has the following factory methods for LambdaExpressions:

```
public static LambdaExpression Lambda
    (Expression body, params ParameterExpression[] parameters);
public static LambdaExpression Lambda
    (Expression body, bool tailcall,
     params ParameterExpression[] parameters);
public static LambdaExpression Lambda
    (Expression body,
     IEnumerable<ParameterExpression>
     parameters);
public static LambdaExpression Lambda
    (Expression body, bool tailcall,
     IEnumerable<ParameterExpression>
     parameters);
public static LambdaExpression Lambda
    (Expression body, String name,
     IEnumerable<ParameterExpression>
     parameters);
```

```
public static LambdaExpression Lambda
    (Expression body, String name, bool tailcall,
     IEnumerable<ParameterExpression>
     parameters);

public static Expression<TDelegate> Lambda<TDelegate>
    (Expression body, String name,
     IEnumerable<ParameterExpression>
     parameters);
public static Expression<TDelegate> Lambda<TDelegate>
    (Expression body, String name, bool tailcall,
     IEnumerable<ParameterExpression>
     parameters);
public static Expression<TDelegate> Lambda<TDelegate>
    (Expression body, params ParameterExpression[] parameters);
public static Expression<TDelegate> Lambda<TDelegate>
    (Expression body, bool tailcall,
     params ParameterExpression[] parameters);
public static Expression<TDelegate> Lambda<TDelegate>
    (Expression body,
     IEnumerable<ParameterExpression>
     parameters);
public static Expression<TDelegate> Lambda<TDelegate>
    (Expression body, bool tailcall,
     IEnumerable<ParameterExpression>
     parameters);

public static LambdaExpression Lambda
    (Type delegateType, Expression body, String name,
     IEnumerable<ParameterExpression>
     parameters);
public static LambdaExpression Lambda
    (Type delegateType, Expression body, String name,
     bool tailcall,
     IEnumerable<ParameterExpression> parameters);
public static LambdaExpression Lambda
    (Type delegateType, Expression body,
     params ParameterExpression[] parameters);
public static LambdaExpression Lambda
    (Type delegateType, Expression body, bool tailcall,
     params ParameterExpression[] parameters);
public static LambdaExpression Lambda
    (Type delegateType, Expression body, bool tailcall,
     params ParameterExpression[] parameters);
public static LambdaExpression Lambda
    (Type delegateType, Expression body,
     IEnumerable<ParameterExpression>
     parameters);
public static LambdaExpression Lambda
    (Type delegateType, Expression body, bool tailcall,
     IEnumerable<ParameterExpression>
     parameters);
```

The following is derived from the v1 spec ... with updates for correctness or new behaviors

When no delegate type is specified via generic or other parameters, body must be non-null. If you supply more than 16 parameters, the factory throws an exception; you can use Expression.GetDelegateType and another factory method to work around this. These methods construct a delegate type based on body.Type and the Type property of each parameter, if supplied. When possible, the delegate type is from the System.Action (if body has void Type) or System.Func (if body is value returning). These factories then pass the delegate type, body, and parameters to one of the other Lambda methods.

When you supply the delegate types via the generic parameter, it must be a delegate type. If supplied via the delegateType parameter, it must be a non-null delegate type. Body must be non-null, and body.Type must represent a type that is assignable to the return type of the delegate type. There are two special cases the factory handles when body's type is not assignable to the return type. The first is that if the delegate's return type is void, in which case the compiler pops the IL stack should Body.Expression leave a value on it. The second case is if the return type is a sub type of LambdaExpression, and the body expression object itself (that is, the Expression node) is of a type that is assignable to the return type. In this case the factory wraps the body Expression node in a Quote node. The list of parameters for the delegate must have the same length as parameters, or be 0 if parameters is null. If parameters is non-null, all its elements must be non-null, and their Type properties must represent types that are assignable from the corresponding parameter types of the delegate.

If tailcall is unsupplied, it defaults to False. When true, this parameter indicates that compiling this LambdaExpression should attempt to use tail calls for any value returning expressions. True is not a guarantee since some calls cannot be tailed called (for example, there may be write backs needed for some properties or ref args). There are also .NET CLR constraints when JIT'ing these calls within dynamic methods. You cannot call from security transparent dynamic methods to critical callees, which may be allowed in CLR 4.0 on x86 (it works on x64). If all the methods are in the same assembly, or for a dynamic method calling a dynamic method (such as, recursing on itself), JIT should use tail call on CLR 3.5 and 4.0.

Name, if not null, is used for debugging or pretty printing only. It has no semantic value. If you need to detect internal helper frames used by the DLR (for example, CallSite cache delegates), see System.Runtime.CompilerServices.IsInternalFrame.

The resulting object is always an Expression<Tdelegate> with:

- Node kind Lambda
- Body and Parameters set to body and parameters, respectively. If parameters is null, Parameters is an empty collection.
- TailCall property set to the tailcall parameter or False
- Type property set to the delegate type

4.43 Expression<TDelegate> Class

This class is the concrete type instantiated for LambdaExpressions. See that type for more information.

4.43.1 Class Summary

```
public sealed class Expression<TDelegate> : LambdaExpression {
    public new TDelegate Compile();
```

```
public new TDelegate Compile(DebugInfoGenerator debugInfoGenerator)
```

4.44 DebugInfoGenerator Class

This abstract class represents a debug information writer for use with LambdaExpression Compile methods. This is in the System.RuntimeCompilerServices namespace.

There is a static factory method for generating a default writer that emits a pdb file.

This class allows you to own the debug info writing, or to intercept it. You might create a default writer and wrap it in a customer writer. The custom writer can create other ways to get at the info (for example, for creating a debugging experience within an RIA under Silverlight where normal pdb's aren't available). Your custom writer can also delegate to the default writer for creating normal pdb's as well.

4.44.1 Class Summary

```
public abstract class DebugInfoGenerator {
    public static DebugInfoGenerator CreatePdbGenerator() {
        return new SymbolDocumentGenerator();
    public abstract void MarkSequencePoint
        (LambdaExpression method, int ilOffset,
         DebugInfoExpression sequencePoint);
```

4.44.2 DebugInfoGenerator Method

This static factory method returns a default info write that generates a pdb file.

Signature:

```
    public static DebugInfoGenerator CreatePdbGenerator() {
        return new SymbolDocumentGenerator();
```

4.44.3 MarkSequencePoint Method

This method takes sequence point information from LambdaExpression.Compile* methods. Method is the LambdaExpression being compiled, and ilOffset is the instruction counter offset into the dynamic methods IL. SequencePoint is the debug info marker expression in the ET for which this sequence point is being emitted.

Signature:

```
    public abstract void MarkSequencePoint
        (LambdaExpression method, int ilOffset,
         DebugInfoExpression sequencePoint);
```

4.45 ParameterExpression Class

This class represents a reference to a variable defined in the containing context. This node uses the Parameter node kind as a legacy from ETs v1.

Variables must be listed using ParameterExpressions as parameters for LambdaExpression or as lexicals in a BlockExpression to in effect define them in some sub tree. To reference a variable, you alias the ParameterExpression object used to define the variable. Note, while Parameter

node references are what determine variable binding, you can declare the same Parameter object in nested BlockExpressions. The ET compiler resovles the references to the tightest containing Block that declares the Parameter.

The Name property is purely for debugging or pretty printing purposes and has no semantic value whatsoever.

Closure environments and lifting variables happens automatically as needed when compiling ETs.

4.45.1 Class Summary

```
public class ParameterExpression : Expression {
    public Boolean IsByRef { get; }
    public String Name { get; }
```

4.45.2 IsByRef Property

This property returns whether the variable is a parameter passed by reference instead of by value. This can only be true for parameters to LambdaExpression, not for variables in BlockExpression. Parameters marked IsByRef cannot be lifted to a closure environment.

An example of creating a ByRef parameter follows:

```
delegate void RefDelegate(ref int a);
...
// In some code somewhere ...
    var parameter = Expression.Parameter(typeof(int).MakeByRefType(), "x");
    var lambda = Expression.Lambda<RefDelegate>
        (Expression.Assign(parameter, Expression.Constant(123)),
         new[] { parameter });
    var d = lambda.Compile();
    Console.WriteLine(lambda.Parameters[0].IsByRef);
    int x = 0;
    d(ref x);
    Console.WriteLine(x);
```

Signature:

```
public Boolean IsByRef { get; }
```

4.45.3 Name Property

This property returns the string name of the variable for debugging or pretty printing only. It has no semantic meaning, and it may be null.

Signature:

```
public String Name { get; }
```

4.45.4 Factory Methods

Expressions have the following factory methods for ParameterExpressions:

```
public static ParameterExpression Parameter
    (Type type, String name);
```

```
public static ParameterExpression Parameter(Type type)
public static ParameterExpression Variable
    (Type type, String name);
public static ParameterExpression Variable(Type type)
```

Type must be non-null. The resulting node has Type and Name set to the factory arguments.

4.46 RuntimeVariablesExpression Class

This class represents variables that need to be accessed for getting/setting as lifted closure variables. This class uses the RuntimeVariables node kind.

The Type property of this expression kind is IRuntimeVariables. This could have been IList<IStrongBox>, but having IRuntimeVariables enables the implementation of the closure environment to change over time.

4.46.1 Class Summary

```
public sealed class RuntimeVariablesExpression : Expression {
    public ReadOnlyCollection<ParameterExpression> Variables { get; }
    public RuntimeVariablesExpression Update
        (IEnumerable<ParameterExpression> variables)
```

4.46.2 Variables Property

This property returns the collection of ParameterExpressions representing the variables to close over and make available first class at runtime.

Signaure:

```
public ReadOnlyCollection<ParameterExpression> Variables { get; }
```

4.46.3 Update Method

This method creates a new expression that is like this one, but using the supplied children. If all of the children are the same, this method returns this expression instead of creating a new one. This is useful when implementing ExpressionVisitors to rewrite the current node only when you need to do so.

Signature:

```
public RuntimeVariablesExpression Update
    (IEnumerable<ParameterExpression> variables)
```

4.46.4 Factory Methods

The Expression class has the following factory methods for creating RuntimeVariableExpressions:

```
public static RuntimeVariablesExpression RuntimeVariables
    (IEnumerable<ParameterExpression>
     variables);
public static RuntimeVariablesExpression RuntimeVariables
    (params ParameterExpression[] variables);
```

If any member of variables is null, these throw an ArgumentNullException.

4.47 IRuntimeVariables Interface

This interface provides access to local variables explicitly requested to be lifted into a closure via a RuntimeVariablesExpression.

4.47.1 Class Summary

```
public interface IRuntimeVariables {
    int Count { get; }
    object this[int index] { get; set; }
```

4.47.2 Count Property

This property returns the number of variables captured by the RuntimeVariablesExpression.

Signature:

```
int Count { get; }
```

4.47.3 This Property

This property takes an index, zero to Count - 1, to get or set a closed over variable requested via a RuntimeVariablesExpression.

Signature:

```
object this[int index] { get; set; }
```

4.48 SwitchExpression Class

This class represents a switch or select expression, where possible cases are considered to find a match to a condition value. These nodes use the Switch node kind. Below are summary semantics, but see section 4.4 for more details on the semantics of the Switch node kind.

These sorts of constructs vary widely across languages (kinds of values you can select over, whether cases can be merged, whether cases fall through to one another, how default cases are handled, etc. SwitchExpression neither matches C# nor VB exactly, but it has a nice simplicity that supports many uses in those languages.

At a high level, this node's semantics is to evaluate the SwitchValue expression, then to evaluate each SwitchCase's TestValues in order. For each test value, if the SwitchCase.Comparison (invoked on the SwitchValue and TestValue) return True, then the corresponding SwitchCase.Body executes. If no case fires, then the DefaultBody executes. The value resulting from the SwitchExpression is the last expression executed, which is typically the last expression of the selected case body.

If you want the effect of case fall through, then you can use GotoExpression and construct the target case as follows. The case's body can be a BlockExpression, and the first expression in it can be a LabelExpression with a null Expression property. The expression compiler detects patterns for eliminating the goto's.

4.48.1 Class Summary

```
public sealed class SwitchExpression : Expression {
    public ReadOnlyCollection<SwitchCase>
           Cases { get; }
    public MethodInfo Comparison { get; }
    public Expression DefaultBody { get; }
    public Expression SwitchValue { get; }
    public SwitchExpression Update(Expression switchValue,
                                   IEnumerable<SwitchCase> cases,
                                   Expression defaultBody)
```

4.48.2 Cases Property

This property returns the collection of SwitchCases describing each branch of the SwitchExpression. Each case is considered in the order in which it appears in the collection.

Signature:

```
    public ReadOnlyCollection<SwitchCase>
           Cases { get; }
```

4.48.3 Comparison Property

This property returns the function that takes as its first argument the SwitchExpression.SwitchValue and each case's test value as the second argument. If it returns True, then the body of the case containing the test value executes.

Signature:

```
    public MethodInfo Comparison { get; }
```

4.48.4 DefaultBody Property

This property returns the Expression to execute if no cases match the SwitchValue. This property may return null, but only if this SwitchExpression object's Type property represents void.

Signature:

```
    public Expression DefaultBody { get; }
```

4.48.5 SwitchValue Property

This property returns the test condition value to be compared with all the cases' test values.

```
    public Expression SwitchValue { get; }
```

4.48.6 Update Method

This method creates a new expression that is like this one, but using the supplied children. If all of the children are the same, this method returns this expression instead of creating a new one. This is useful when implementing ExpressionVisitors to rewrite the current node only when you need to do so.

Signature:

```
public SwitchExpression Update(Expression switchValue,
                               IEnumerable<SwitchCase> cases,
                               Expression defaultBody)
```

4.48.7 Factory Methods

The Expression class has the following factory methods for creating SwitchExpressions:

```
public static SwitchExpression Switch(Expression value,
                                      params SwitchCase[] cases);
public static SwitchExpression Switch
    (Expression switchValue, Expression defaultBody,
     params SwitchCase[] cases)
public static SwitchExpression Switch
    (Expression switchValue, Expression defaultBody,
     MethodInfo comparison, params SwitchCase[] cases)
public static SwitchExpression Switch
    (Type type, Expression switchValue, Expression defaultBody,
     MethodInfo comparison, params SwitchCase[] cases)
public static SwitchExpression Switch
    (Expression switchValue, Expression defaultBody,
     MethodInfo comparison, IEnumerable<SwitchCase> cases)
public static SwitchExpression Switch
    (Type type, Expression switchValue, Expression defaultBody,
     MethodInfo comparison, IEnumerable<SwitchCase> cases)
```

Comparison defaults to the effect of using Expression.Equal. The comparison function gets invoked with the switch value as the first argument and each case's test value in turn as the second argument.

The Type property, if not supplied via the type parameter, is set to a case body's type, and all SwitchCase.Body.Type properties must represent the same type. This is for simplicity of the compiler in choosing a unifying type and for more explicit correctness by forcing ET producers to specify their exact intentions

If type is supplied as void, then the case bodies can be of any type, and any results are automatically "converted to void" or squelched. If the type is supplied as non-void, then the SwitchCase.Body.Type properties must represent types that are reference assignable to the type represented by the type parameter.

If DefaultBody is null, the Type property must represent void type. If the type parameter is unsupplied, then each SwitchCase.Body.Type must be void. If DefaultBody is non-null, then it's Type property must represent a type that is reference assignable to the type represented by the SwitchExpression's Type property.

If comparison is supplied, then every SwichCase TestValue must have a Type property that represents a type that is reference assignable to the second parameter of the comparison function. If comparison is not supplied, there are more constraints on the test values in case branches. All test value Expressions across every SwitchCase must have exactly the same Type property. The factories effectively call Expression.Equal on value and a case's test value to find a comparison method. These constraints are for simplicity of the compiler in choosing a unifying type and searching for an Equal method as well as simplicity for ET consumer. Note, if your intention is to get reference equality semantics, then you need to wrap the values in Convert to object expressions.

4.49 SwitchCase Class

This class represents a single case in a SwitchExpression. While the cases do fit the expression-based model and result in a value, they are not Expressions since they cannot appear anywhere in an ET, only in Switch nodes.

4.49.1 Class Summary

```
public sealed class SwitchCase {
    public Expression Body { get; }
    public ReadOnlyCollection<Expression>
           TestValues { get; }
    public SwitchCase Update(IEnumerable<Expression> testValues,
                             Expression body)
```

4.49.2 Body Property

This property returns the Expression to execute if one of this case's test values is equal to the containing Switch's SwitchValue.

Signature:

```
    public Expression Body { get; }
```

4.49.3 TestValues Property

This property returns the collection of test value Expressions to consider in order to determine whether to select this case.

Signature:

```
    public ReadOnlyCollection<Expression>
           TestValues { get; }
```

4.49.4 Update Method

This method creates a new expression that is like this one, but using the supplied children. If all of the children are the same, this method returns this expression instead of creating a new one. This is useful when implementing ExpressionVisitors to rewrite the current node only when you need to do so.

Signature:

```
    public SwitchCase Update(IEnumerable<Expression> testValues,
                             Expression body)
```

4.49.5 Factory Methods

The Expression class has the following factory for creating SwitchCases objects:

```
    public static SwitchCase SwitchCase(Expression body,
                                        params Expression[] testValues)
    public static SwitchCase SwitchCase
        (Expression body, IEnumerable<Expression> testValues)
```

If the SwitchExpression to which the resulting SwitchCase gets added has an explicit comparison function, then each of the testValues must have a Type property that represents a type that is reference assignable to the second parameter of the comparison function. Otherwise all test value Expressions must have exactly the same Type property. This is for simplicity of the compiler in choosing a unifying type and searching for an Equal method as well as simplicity for ET consumer.

4.50 ExpressionVisitor Class

This class provides a visitor framework for ET nodes. With Extension node kinds and reducibility, providing a walker model is important for saving everyone effort and having well-behaved extensions. Without a common walking mechanism built-in, everyone would have to fully reduce extension nodes to walk them. Reducing is lossy for meta-programming because usually you can't go back to the original ET, especially if you're rewriting parts of the tree.

The ExpressionVisitor class is abstract with two main entry points and many methods for sub classes to override. The entry points visit an arbitrary Expression or collection of Expressions. The methods for sub classes to override correspond to the node types. For example, if you only care to inspect or act on BinaryExpressions and ParameterExpressions, you'd override VisitBinary and VisitParameter. The methods you override all have default implementations that just visit their children. If the result of visiting a child produces a new node, then the default implementations construct a new node of the same kind, filling in the new children. If the child comes back identity equal, then the default implementations just returns the node passed in.

As an Extension node kind author, you should override Expression.VisitChildren to visit your sub expressions. Furthermore, if any children come back as new objects, you should reconstruct your node type with the new children, returning the new node as your result. By default Expression.VisitChildren reduces the expression to all common nodes and then calls the visitor on the result.

The following subsections only spec the members that have design motivation or some behavior worth noting. Otherwise, the methods by default visit their children and create new instances of the node types or helper objects (for example, CatchBlock) if any children changed when they were visited.

4.50.1 Rebinding When Children Nodes' Type Properties Change

There are constraints in the default visitor methods regarding Type property changes when children get rewritten. Changing the types of children can cause the parent node to rebind its semantics when it creates a new instance of itself. For example, an Add node of two integers could become an Add node of a user-defined integer wrapper and an integer, where the user-defined op_Add method adversely affects the original ET's semantics. Since accidental semantics changes are more likely programmer errors, the default visitor methods resist these changes.

If you intend to rewrite children node Type properties and cause rebinding in parent nodes, you'll need to explicitly override the parent node visitor method to control the rewriting and rebinding of the parent node's semantics. This should not mean you'll need to override all

methods. Most likely you'll only need to override BinaryExpression and UnaryExpression, maybe a couple of others.

Some nodes inherently resist when type changes occur from rewrites. If the parent node has an explicit MethodInfo stating its semantics, then the default implementations of the visit methods call the factories with the method info. The factories will check that the new operands are still assignable to the method's parameters.

When there is no explicit method info, if the children nodes are primitive types, the default implementations require the children's types to remain the same after rewriting. If the new Type properties represent reference types, then there are a few cases:

- Generally:
  - If the node has a methodinfo, the default implementation of the visitor method just calls the factory which confirms the methodinfo info still applies.
  - If there's no method info, and the children had value types, the rewritten children must have the same value types as before.
  - If there's no method info, and the children had reference types, the rewritten children must have Type properties representing types that are reference assignable to the original types represented by the children. Again, the default implementations check this by just calling the factory and ensuring the result type is still valid for the operation, and no new methodinfo got selected.
- Equal (and NotEqual): if the Equal/NotEqual node had reference equality, then the default visitor method preserves these semantics rather than rebinding to user-defined equality (operand Type properties don't matter then). Otherwise, the method info must fit the new children types.
- Switch: If the node had no methodinfo for comparing the switch value with the test values, and the new Swtich node with the rewritten children must not have an inferred methodinfo.
- Convert, Coalesce, TypeAs: these nodes are about ensuring their expression's Type aligns or unifies with other nodes. Since the methodInfo the conversion semantics, the general rules apply.

4.50.2 Class Summary

```
public abstract class ExpressionVisitor {

    protected ExpressionVisitor();

    protected static ReadOnlyCollection<T> Visit<T>
        (ReadOnlyCollection<T> nodes, Func<T,T> elementVisitor);
    protected ReadOnlyCollection<Expression> Visit
        (ReadOnlyCollection<Expression> nodes);
    // Visit is virtual just for consistency with the visitor sample
    // popular with ETs v1.  You should never need to override it.
    public virtual Expression Visit(Expression node);
    protected ReadOnlyCollection<T> VisitAndConvert<T>
        (ReadOnlyCollection<T> nodes, String callerName)
        where T : Expression;
    protected T VisitAndConvert<T>(T node, String callerName)
        where T : Expression;
    protected virtual Expression VisitBinary
        (BinaryExpression node);
```

```
protected virtual Expression VisitBlock
    (BlockExpression node);
protected virtual CatchBlock VisitCatchBlock(CatchBlock node);
protected virtual Expression VisitConditional
    (ConditionalExpression node);
protected virtual Expression VisitConstant
    (ConstantExpression node);
protected virtual Expression VisitDebugInfo
    (DebugInfoExpression node);
protected virtual Expression VisitDefault
    (DefaultExpression node);
protected virtual Expression VisitDynamic
    (DynamicExpression node);
protected virtual ElementInit VisitElementInit(ElementInit node);
protected virtual Expression VisitExtension
    (Expression node);
protected virtual Expression VisitGoto
    (GotoExpression node);
protected virtual Expression VisitIndex
    (IndexExpression node);
protected virtual Expression VisitInvocation
    (InvocationExpression node);
protected virtual Expression VisitLabel
    (LabelExpression node);
protected virtual LabelTarget VisitLabelTarget
    (LabelTarget node);
protected virtual Expression VisitLambda<T>
    (Expression<T> node);
protected virtual Expression VisitListInit
    (ListInitExpression node);
protected virtual Expression VisitLoop
    (LoopExpression node);
protected virtual Expression VisitMember
    (MemberExpression node);
protected virtual MemberAssignment VisitMemberAssignment
    (MemberAssignment node);
protected virtual MemberBinding VisitMemberBinding
    (MemberBinding node);
protected virtual Expression VisitMemberInit
    (MemberInitExpression node);
protected virtual MemberListBinding VisitMemberListBinding
    (MemberListBinding binding);
protected virtual MemberMemberBinding VisitMemberMemberBinding
    (MemberMemberBinding node);
protected virtual Expression VisitMethodCall
    (MethodCallExpression node);
protected virtual Expression VisitNew
    (NewExpression node);
protected virtual Expression VisitNewArray
    (NewArrayExpression node);
protected virtual Expression VisitParameter
    (ParameterExpression node);
protected virtual Expression VisitRuntimeVariables
    (RuntimeVariablesExpression node);
protected virtual Expression VisitSwitch
    (SwitchExpression node);
protected virtual SwitchCase VisitSwitchCase(SwitchCase node);
```

```
protected virtual Expression VisitTry
    (TryExpression node);
protected virtual Expression VisitTypeBinary
    (TypeBinaryExpression node);
protected virtual Expression VisitUnary
    (UnaryExpression node);
```

4.50.3 Visit<T> Method

This method iterates over nodes invoking elementVisitor on each. If any invocation returns a new instance of the T, then Visit<T> copies the collection, aliasing unchanged elements and point to the new ones elementVisitor created.

This method is for convenience, and it is used in the default implementations of several methods to walk children (for example, SwitchCases, CatchBlocks, MemberBindings, etc.). It is protected so that you can use it as well.

Signature:

```
protected static ReadOnlyCollection<T> Visit<T>
    (ReadOnlyCollection<T> nodes, Func<T,T> elementVisitor);
```

4.50.4 VisitLambda<T> Method

This method visits the parameters and the body of a LambdaExpression. It requires the generic parameter to solve a couple of rewriting issues. While the T is the same delegate type as LambdaExpression.Type, in some cases it isn't possible to pass the delegate type to the factories for constructing new LambdaExpressions. When some code, A, is rewriting a Lambda node from other code, B, that has a private delegate type, A cannot always invoke the factories with the delegate type (for example, in partial trust). Due to how access works in .NET, when the delegate type flow in via the T parameter, you can call the factories with the private delegate type.

Signature:

```
protected virtual Expression VisitLambda<T>
    (Expression<T> node);
```

4.50.5 VisitAndConvert<T> Method

This method effectively calls Visit on all the nodes, and if any node gets changed by the visitor, this method ensures the new node is the same type as the old node.

This is a convenience method for iterating nodes like BlockExpression.Variables or LambdaExpression.Parameters. It takes a string for the caller's name, and if there's a rewriting error, the this method throws an exception with a message saying you should override the calling visit method to handle iterating over the nodes or handle the rewrites yourself. This is protected so that you can use it too in derived visitors.

Signature:

```
protected ReadOnlyCollection<T> VisitAndConvert<T>
    (ReadOnlyCollection<T> nodes, String callerName)
    where T : Expression;
protected T VisitAndConvert<T>(T node, String callerName)
    where T : Expression;
```

4.50.6 VisitConstant Method

This method by default just returns the node with no recursive visiting.

Signature:

```
protected virtual Expression VisitConstant
    (ConstantExpression node);
```

4.50.7 VisitDebugInfo Method

This method by default just returns the node with no recursive visiting.

Signature:

```
protected virtual Expression VisitDebugInfo
    (DebugInfoExpression node);
```

4.50.8 VisitDynamic Method

This method by default just visits the argument expressions.

Signature:

```
protected virtual Expression VisitDynamic
    (DynamicExpression node);
```

4.50.9 VisitDefault Method

This method by default just returns the node with no recursive visiting.

Signature:

```
protected virtual Expression VisitDefault
    (DefaultExpression node);
```

4.50.10 VisitExtension Method

This method by default calls node.VisitChildren.

Signature:

```
protected virtual Expression VisitExtension
    (Expression node);
```

4.50.11 VisitLabelTarget Method

This method by default just returns the node with no recursive visiting.

Signature:

```
protected virtual Expression VisitLabelTarget
    (LabelTarget node);
```

4.50.12 VisitMember Method

This method by default just visits the object expression.

Signature:

```
protected virtual Expression VisitMember
    (MemberExpression node);
```

4.50.13 VisitNew Method

This method by default just visits the argument expressions. If any argument expression changes, then this method creates a new NewExpression with the same Members if there are any.

Signature:

```
protected virtual Expression VisitNew
    (NewExpression node);
```

4.50.14 VisitParameter Method

This method by default just returns the node with no recursive visiting.

Signature:

```
protected virtual Expression VisitParameter
    (ParameterExpression node);
```

4.51 POST CLR 4.0 -- GlobalVariableExpression Class

This type models a variable look up that requires hosting context. The variable gets looked up in a current ScriptScope in which he code is executing or along a chain of ScriptScopes including the ScriptRuntime.Globals ScriptScope.

This node's kind is Extension.

4.51.1 Class Summary

```
public sealed class GlobalVariableExpression : Expression {
    public Boolean IsLocal { get; }
    public String Name { get; }
```

4.52 POST CLR 4.0 -- GeneratorExpression

This type models code that can contain YieldExpressions. This is not shipping in CLR 4.0 and is only available on www.codeplex.com/dlr . This node reduces by generating an ET that embodies the state machine needed to return values, re-enter the generator, and re-establish any dynamic context such as try-catch's.

This node's kind is Extension.

While this expression is generally useful, it implements a couple of behaviors specific to Python and needs to be generalized, particularly around yield in finally blocks and re-entering the generator with a value as the result of a YieldExpression.

4.52.1 Class Summary

```
public sealed class GeneratorExpression : Expression {
    public Expression Body { get; }
```

```
public LabelTarget Target { get; }
```

4.52.2 Body Property

This property returns the code to build into the generator's state machine that computes values to yield.

4.52.3 Target Property

This property returns the representation of the place in the code that YieldExpressions will Goto to return values from the generator.

4.53 POST CLR 4.0 -- YieldExpression Class

This type models locations in GeneratorExpressions where the code you return a value, and where the generator's state machine needs to be able to re-enter to continue computing values. It is only valid to use YieldExpression nodes in a sub ET with a GeneratorExpression root node.

This node's kind is Extension.

While this expression is generally useful, it implements a couple of behaviors specific to Python and needs to be generalized, particularly around yield in finally blocks and re-entering the generator with a value as the result of a YieldExpression.

4.53.1 Class Summary

```
public sealed class YieldExpression : Expression {
    public LabelTarget Target { get; }
    public Expression Value { get; }
    public Int32 YieldMarker { get; }
```

4.53.2 Target Property

This property returns the representation of the place in the code that YieldExpressions will Goto to return values from the generator. It must match the Target property of the GeneratorExpression node that is the root of the sub ET containing this YieldExpression.

Signature:

```
public LabelTarget Target { get; }
```

4.53.3 Value Property

This property returns the expression that models the value the generator returns at this point in the code.

Signature:

```
public Expression Value { get; }
```

4.53.4 YieldMarker Property

This property returns a unique value within the GeneratorExpression that is the root of the sub ET containing this YieldExpression. This unique value identifies this YieldExpression for tree rewriters. The generator object returned from GeneratorExpression has a method on it

The uniqueness of this value is the responsibility of the node creator, and the DLR does not ensure it is unique.

Signature:

```
public Int32 YieldMarker { get; }
```

~~4.54 CUT Annotations Class~~

~~This class represents a collection of information associated with an Expression node instance. The information is keyed by a type. An Annotations object can have more than one element of a given type.~~

~~Annotations instances are immutable. Of course, users can add an annotation member that is an indirection point, from which they can maintain mutable information. You might do this in some processing pass where an annotation changes, but you do not want to incur the cost of copying the sub ET from the node to the root of its containing tree over and over.~~

~~4.54.1 Class Summary~~

```
[SerializableAttribute]
public sealed class Annotations : IEnumerable<System.Object>,
                                  IEnumerable {
    public static readonly Annotations Empty;

    public Annotations Add<T>(T annotation);
    public Boolean Contains<T>();
    public T Get<T>();
    public Annotations Remove<T>();
    public Boolean TryGet<T>(out T annotation);
```

~~4.54.2 Empty Field~~

~~This field returns the empty Annotations object from which you can start your own Annotations collection.~~

~~Signature:~~

```
    public static readonly Annotations Empty;
```

~~4.54.3 Add<T> Method~~

~~This method returns a copy of the Annotations object with the argument added.~~

~~*Should we guarantee new element is at the end for purposes of Get and TryGet semantics?*~~

~~Signature:~~

```
    public Annotations Add<T>(T annotation);
```

4.54.4 Contains<T> Method

This method returns true if there are any elements of type T, otherwise it returns false.

Signature:

```
public Boolean Contains<T>();
```

4.54.5 Get<T> Method

This method returns the first element of type T found in the Annotations object. If there are no such elements, then it returns the default instance of T.

Signature:

```
public T Get<T>();
```

4.54.6 TryGet<T> Method

This method returns true and sets its argument the first element of type T found in the Annotations object. If there are no such elements, then it returns false and sets the argument to the default instance of T.

Signature:

```
public Boolean TryGet<T>(out T annotation);
```

4.54.7 Remove<T> Method

This method returns a new Annotations object that is a copy of this Annotations object with all elements of type T removed from it.

Signature:

```
public Annotations Remove<T>();
```