# Two Lightweight DSLs for Rich UI Programming

Sean McDirmid

Microsoft Advanced Technology Center (ATC), Beijing China
smcdirm@microsoft.com

**Abstract.** User interfaces are evolving beyond bitmaps to include animation and special effects that utilize powerful graphics hardware. Unfortunately, the APIs used to implement these features are often not programmer friendly and can result in verbose code that is written in multiple languages. This paper describes our experience in improving UI library usability through **lightweight domain-specific languages (DSL)** that are limited in scope to ease the use of library features rather than whole libraries. Lightweight DSL code is evaluated without meta-programming by using a hosting language's conventional extensibility mechanisms such as operating overloading and automatic conversions. As a result, lightweight DSLs are easy to implement while their code can easily be modularized and manipulated by host language abstractions. We demonstrate the effectiveness of our technique through two C# lightweight DSLs for expressing databinding and pixel shading in Microsoft's WPF UI library.

## 1 Introduction

Because of increasingly powerful graphics hardware, a rich user interface (UI) can be built out of high-fidelity animations and special effects that are exposed in recent UI toolkits such as Microsoft's Windows Presentation Foundation (WPF) [17], Apple's Core Animation [2], and Sun's JavaFX [21] libraries. However, mainstream languages such as Java or C# alone are inadequate for programming many features of these UI toolkits. The toolkits often have declarative APIs that are awkward to use in a general purpose language; e.g., constraints in a UI toolkit that support retained graphics and animation. Additionally, the UI toolkits often have functionality that cannot even be expressed in a general purpose language; e.g., pixel shaders execute according to a very different simple parallel programming model.

Rich UI programming is often improved through domain specific languages (DSLs). Examples of such languages include Sun's JavaFX Script [22], which has dedicated animation and constraint constructs, Microsoft's HLSL [15], which is a low-level C-like language that is used to program pixel shaders, or Microsoft's XAML [18], which is a markup language for UI layout and styling. Unfortunately, DSLs are expensive to both build and use: compilers and tools must be built and programmers have to learn yet another language. As a result, the DSL approach works best if the benefits are large; i.e., if an entire library can be improved by the approach. However, often only some features of a library can benefit from the DSL where other features difficult or impossible to use in the DSL. For example, XAML improves the encoding of styling and layout in WPF but cannot be used to express event handlers, while HLSL can only be used to express pixel shaders. Such DSLs must then be combined with code in other languages; e.g., C# "code behind" is used to express what XAML cannot, while HLSL-based pixel shaders are enabled in WPF through C# boilerplate code. Such multi-language solutions burden the developer with additional complexity in the form of bridge code as well as requiring them to switch between multiple languages.

Rather than use UI DSLs, UI programming constructs can instead be directly added to a general purpose language as an *embedded DSL*. For example, FrTime [3] enhances Scheme using macros [9] with functional reactive programming (FRP) constructs known as *signals* [6] that directly support retained graphics and animation. Through scheme macros, expressions can be "quoted" as data that can then be re-interpreted in some custom way, which is a form of meta-programming. LINQ [16] enables such meta-programming in C#: C# code that undergoes syntax and type checking by the compiler can be transformed into reified expression trees that undergo custom interpretation. Using LINQ, it is possible to implement multiple DSLs to handle various features of WPF by re-interpreting C# code. Such DSLs reuse the existing C# language and would benefit fully from IDE features such as Intellisense (code completion). Unfortunately, these DSLs are still fairly expensive to implement as a full expression tree interpreter must be implemented for the DSL. More importantly, quoted DSL code cannot be freely mixed with unquoted host language code and so the DSL code cannot easily be modularized or otherwise configured using host language abstractions. This limits many useful techniques such as executing loops in the host language as a form of partial evaluation or using virtual methods to configure and select DSL code.

This paper presents experience on a hybrid approach that relies on a general purpose language's conventional syntax and typing extension constructs to host multiple *lightweight DSLs* whose scopes are limited to supporting a limited aspect of a domain; e.g., constraints or pixel shading in the UI domain. A lightweight DSL is based on *placeholder values* that can only be composed into expressions and cannot be viewed directly from within the host language. Placeholder value expressions are instead translated into the code that the programmer would have written without the DSL. As an example, consider a statement in a lightweight databinding DSL for C#/WPF that expresses a layout constraint:

```
b.CenterTop.Bind = s.LeftBottom+PointSg.Make(s.Value*s.Width,0);
```

This statement aligns a button `b` directly under a slider `s`'s user-draggable slide thumb. Each of `LeftBottom`, `Value`, and `Width` are placeholder values that are composed into an expression that represents the point beneath the slider's thumb. The value of this expression changes at run time either by moving the slider's thumb, changing the slider's position, or by changing the slider's width. The expression is bound to the `CenterTop` placeholder value of a button via the `Bind` operation of the databinding DSL. The DSL's runtime will then enforce the layout constraint by creating and configuring WPF multi-binding objects. These details are hidden to the programmer: because of C#'s support for operator overloading, custom property syntax, and extension methods, bind appears as a very normal-looking arithmetic assignment operation. The approach also improves on safety: the DSL ensures that `CenterTop` is only bound to point expressions, whereas no static checking is performed when conventional WPF databinding APIs are used.

Lightweight DSLs do not depend on meta-programming–instead library functionality is explicitly prepared for use in the DSL through an auxiliary library. For example, each WPF dependency property must be manually wrapped as a placeholder value for use in the constraint DSL, while all scalar operations must be manually wrapped as operations on placeholder values. To suit the needs of the lightweight DSL, additional placeholder values and operator semantics are specified during wrapping to enhance functionality and improve safety. For example, the databinding DSL supports invertible operations where a `Right` placeholder value can be defined as `Left + Width` and `b.Right.Bind = s.Left` is equivalent to `b.Left.Bind = s.Left - b.Width`. Finally, explicit wrapping also ensures that aspects of the host language that the DSL cannot support cannot be accessed by the programmer.

Our technique has been used to build two lightweight DSLs that support the programming of rich UIs in Microsoft's WPF library using C#. The first lightweight DSL wraps WPF databinding with an improved constraint-like syntax. WPF databinding creates a constraint over a dependency property so that it is always the current value of an expression over one or more other dependency properties. By operating on dependency properties through a databinding DSL, constraint operations can be expressed in a more concise, safe, and reusable way. The second lightweight DSL is used to express WPF pixel shaders. In this case, the parameters of the shader, including the current pixel's texture and coordinate, are placeholder values that can be operated on to form the shader's resulting pixel. The DSL code is then compiled into a shader via automatic translation into HLSL. By writing pixel shaders in C# rather than HLSL, boilerplate is eliminated while programmers gain more safety and can take advantage of higher-level language features. For example, even though a pixel shader cannot support virtual method dispatch, this feature can be used in the construction of the pixel shader. Additionally, since similar HLSL code is generated to what would be written by hand, little or no run-time performance is sacrificed.

The rest of this paper is organized as follows. Section 2 provides background on existing techniques and problems in rich UI programming. Section 3 then shows how two C# lightweight DSLs can improve on existing techniques, making it easier to program rich UI experiences. Section 4 describes how lightweight DSLs are implemented and we discuss the implications of our approach in Section 5. Comparisons with existing techniques is presented together with related work in Section 6. We conclude in Section 7. Finally, the source code along with examples for DSLs described in this paper is available at `http://www.codeplex.com/bling`.

## 2 Background

Rich graphics supported by modern graphics hardware are going beyond their traditional uses in graphics-intensive applications such as games to be used in rich user interfaces for productivity applications. UI toolkits such as WPF support the construction of such rich user interfaces through high-level APIs that hide the complexity of lower-level game-oriented toolkits such as DirectX [14] in the case of WPF. One important way that WPF and similar toolkits hide complexity from UI programmers is through a *retained mode* API where rendering occurs automatically as UI objects properties, which are called *dependency properties* in WPF, are changed. Examples of WPF dependency properties include the position and size of a widget, its background color, the size and face of its font, the value of a slider, and so on. WPF's retained-mode API is coupled with an animation engine where dependency properties can be bound to values that change during the course of an animation. For example, a widget's left position property can be animated to go from `0` to `1024` in `500` milliseconds, causing the widget to move to the right across the screen. Additionally, a databinding engine allows dependency properties to be bound to expressions over other dependency properties so that the properties' values are always that of their bound expressions. We elaborate on databinding in this section.

WPF animation and databinding are imperative operations that establish declarative effects. In the simplest case, one property is animated or databound; e.g., consider C# code that binds the left property of widget `A` to the left property of widget `B`:

```
var b = new Binding(widgetB, LeftProperty);
BindingOperations.SetBinding(widgetA, LeftProperty, b);
```

A `Binding` object establishes the declarative effect that takes effect when the call to `SetBinding` is made. Databinding code starts to get very verbose when multiple properties are bound through an expression. As an example, consider C# code that binds the horizontal center of widget `A` to the horizontal center of widget `B`:

```
var mb = new MultiBinding() { Converter = new MyConverter() };
mb.Bindings.Add(new Binding(widgetB, LeftProperty));
mb.Bindings.Add(new Binding(widgetB, WidthProperty));
mb.Bindings.Add(new Binding(widgetA, WidthProperty));
BindingOperations.SetBinding(widgetA, LeftProperty, mb);
```

Since horizontal center is not a property, we must compute it manually by operating on three properties: the left property of widget `B` and the width properties of both widget `B` and `A`. Proprerty values are then combined in a `MyConverter` object:

```
class MyConverter : IMultiValueConverter {
 object Convert(..., object[] params, ...) {
  double leftB = (double) params[0], widthB=(double) params[1],
         widthA= (double) params[2];
  double leftA = leftB + widthB/2 - widthA/2;
  return leftA;
 } ... }
```

The `params` object is populated with the current values of the properties that are added to the `MultiBinding` object. The logic in the `Convert` method operates on these values to find the goal left property value of widget `A`. This example demonstrates how databinding verbosity stems from having the programmer explicitly manage multiple low-level details:

1. The programmer must declare each property involved in the bind.

2. The programmer must remember the intended type of a dependency property along with its sequence in the bind when they write the code of the value converter; e.g., the width property for widget `B` is of type double and is at index `1` in `params`.
3. The programmer must manually rewrite constraints so that only target properties appear on left hand sides; e.g., the constraint `leftA + widthA/2 = leftB + widthB/2` must be written by the programmer as `leftA = leftB + widthB/2 - widthA/2`. This problem is exacerbated when WPF's support for two-way databinding is considered.

By having to manage these details, it is very difficult for the programmer to reuse databinding code; e.g., the index of the dependency property might change, or the value converter code will change completely depending on the constraints that are being expressed. As a result, the verbosity problem leads to a reusability problem that makes databinding programmer unfriendly.

## 2.1 Pixel Shaders

WPF also supports custom GPU-accellerated *pixel shader* effects that post-process each pixel of a targeted UI widget. Although pixel shaders have been used in graphics-intensive applications such as games, only very recently have pixel shaders been applied to user interfaces with support from user interface toolkits such as WPF or JavaFX [23]. A WPF pixel shader effect can implement color-modifying effects, such as embossing or de-coloring a UI widget, or pixel-distoring effects, such as swirling a UI widget or rippling a wave through it. The HLSL language [15] used to encode a WPF shader is a restricted C-like language with very simple control-flow and data-access capabilities; e.g., only 96 instructions, no dynamic looping, and no global data[1]. The HLSL code is compiled into bytecode that is compiled further according to the graphics card on the user's machine.

WPF pixel shaders are completely compatible with WPF's retained mode API: pixel shader parameters are normal dependency properties that can be both databound and animated. A rich UI can then seamlessly use the effect in their UI while animating or databinding the effect's parameters. Example:

```
Grayscale effect = new Grayscale();
widget.Effect = effect;
var b = new Binding(slider, ValueProperty);
BindingOperations.SetBinding(effect, IntensityProperty, b);
```

This code installs a `GrayScale` effect on a `widget` to de-color it according to the current slide value of a `slider`, which in our example is between `0` and `1`. Databinding is used to bind the `effect`'s `IntensityProperty` dependency property to `slider`'s `ValueProperty`. The `IntensityProperty` in turn controls the intensity of the effect: at `0` it is unapplied and `widget` retains its normal color, while at `1` it is fully applied and `widget` is displayed in grayscale. This behavior is encoded through the following HLSL code:

[1] WPF supports Shader Model 2; later shader models have more capabilities.

```
sampler2D implicitInput : register(s0);
float intensity : register(c3);

float4 main(float2 uv : TEXCOORD) : COLOR {
  float4 orig = tex2D(implicitInput, uv);
  float avg = (orig.r + orig.g + orig.b) / 3;
  float4 gray = float4(avg, avg, avg, orig.a);
  return lerp(orig, gray, intensity);
}
```

The `main` method of this code is applied to every pixel of `widget` where the color returned will replace the pixel. The `tex2D` function returns a color in the `widget`'s implicit input texture (`implicitInput`) for the current pixel being processed (`uv`). The `lerp` function interpolates between the first and second color arguments according to the `intensity` value. After the HLSL code is compiled, additional C# boilerplate code is needed to make the pixel shader available in WPF as an effect:

```
class Grayscale : ShaderEffect {
 static DependencyProperty InputProperty = ShaderEffect.
  RegisterShaderSamplerProperty(``Input'',typeof(Grayscale),0);
 static DependencyProperty IntensityProperty = Register(
   ``Intensity'', typeof(double), typeof(Grayscale),
       new UIPropertyMetadata(PixelShaderConstantCallback(3)));
 GrayScale() {
  PixelShader = new PixelShader()
    { UriSource = Global.MakePackUri("grayscale.ps") };
  UpdateShaderValue(InputProperty);
  UpdateShaderValue(IntensityProperty);
 }
}
```

All shader effects have an input property (`InputProperty`) that is implicitly bound to the texture of the UI element they are applied to. The `0` term in `InputProperty`'s definition indicates that it is located in the `s0` sampled texture register of the pixel shader, which corresponds to the `sampler2D implicitInput : register (s0);` statement in the HLSL code. Additional textures that are sampled in a shader are assigned successive sampled texture registers `s1`, `s2`, `s3`, and so on. Likewise, the `3` term in the definition of `IntensityProperty` indicates that the intensity parameter is located in parameter register `c3`, which corresponds to the `float intensity : register(c3);` statement in the HLSL code. Additional parameters can be passed in through other parameter registers `c0`, `c1`, `c2`, and so on. Additional bookkeeping methods called in the constructor (`UpdateShaderValue`) ensure that the pixel shader is re-run when the parameters change.

The pure logic of a pixel shader is that of a simple pixel to pixel function with multiple parameters. However, the encoding of this logic can be insignificant when compared to all of the boilerplate code that must be written around it. As with databinding, the verbosity results from having the programmer manage low-level details. The programmer must remember the intended type of a parameter or texture along with its sequence between the C# boilerplate and the HLSL code; e.g., the `intensity` parameter for `Grayscale` is of type double and is at parameter register `3` in C#, which is of type `float` and register `c3` in HLSL. Because of these details, HLSL and C# boilerplate code are both difficult to reuse. Additionally, although HLSL code can be reused as procedures, HLSL itself has little support for abstraction–it does not support virtual methods or similar kinds of indirection. Coupled with immature IDE support, e.g., no code completion, programming in HLSL can be a very archaic experience when compared to programming in a mature high-level languages like C#.

### 2.2 Implications

Both databinding and pixel shaders are powerful WPF technologies that are hindered by verbose low-level abstractions. If these technologies are not used very often, then verbosity of the abstractions does not matter. For example, WPF databinding is not often used for doing layout–panels with specific layout schemes are used instead. Also, if programmers are expected to often reuse existing effects rather than frequently write custom effects, then it is not very important that pixel shader effects require lots of boilerplate code. However, as these technologies are better understood, useful possibilities can arise that can benefit from more their more intensive use even if such uses were not considered originally. For example, databinding provides a simple and flexible way of expressing custom layouts, while more custom pixel shaders could lead to more immersive lighting effects in UIs. For this reason, the abstractions under which these UI technologies are accessed should be improved.

## 3 Lightweight DSLs

Both the databinding and pixel shader features in WPF are based on a simple kind of pure expression logic without any complex control flow or imperative effects. For databinding, expressions exist on both sides of the constraint being encoded; e.g., consider the layout constraint `leftA + widthA/2 = leftB + widgetB/2` that was encoded using databinding in Section 2. For pixel shaders, expressions are simply used to transform pixels; e.g., the gray scale shader logic described in Section 2.1. Ideally, such expression-oriented code could be expressed directly with little or no boilerplate in a hosting general purpose language. For example, we could express the above layout constraint in C# with only one line of code:

```
(widgetA.Left+widgetA.Width/2).Bind=widgeB.Left+widgetB.Width/2;
```

If the expressions were values in the hosting language, then the hosting language's abstractions could be used to modularize, reuse, and otherwise manipulate the expressions. For example, we could define in C# a property `HorizontalCenter` that is defined on UI elements to be `Left + Width / 2`. The above code could then be expressed concisely as `widgetA.HorizontalCenter.Bind=widgetB.HorizontalCenter`. The ability to form such expressions comes from what we call a *lightweight DSL*, whose characteristics are as follows:

1. It (the lightweight DSL) targets only one library feature, e.g., databinding, and not an entire domain, e.g., UI;
2. It provides an expression-oriented syntax that can be used to concisely encode logic for the target feature;
3. It is fully hosted by a high-level general purpose language where expressions in the DSL are first-class values that can be fully manipulated by the hosting language's abstraction constructs; and finally
4. Its expressions can still be composed and operated on after passing through the host language's abstraction constructs.

The first three characteristics are typical of an *embedded DSL* where a lightweight DSL is a special kind of embedded DSL. The last characteristic distinguishes a lightweight DSL in that it has full access to the power of its host language, which is not a given in embedded DSLs where expressions are treated as pure data. We demonstrate the usefulness of full host language access in this section. Beyond full access, a lightweight DSL is distinguished from a normal embedded DSL by a lightweight metaprogramming-free implementation strategy, which we describe in Section 4. We describe two lightweight DSLs in this section: a databinding DSL and pixel shader DSL that respectively target WPF's databinding and pixel shader effect features.

A lightweight DSL is based on *placeholder values* that encapsulate *DSL values*, which are produced during DSL code execution. Considering C#, a placeholder value's type is of the form *`DSLValue<T>`* where *`DSLValue`* is some generic class defined by the DSL that encapsulates a DSL value of type *`T`*. For example, `Signal<double>` is the type of a double databinding placeholder value while `Shader<Point>` is the type of a point pixel shader placeholder value. The DSL value represented by a placeholder value is hidden from the host language by its type; i.e., the placeholder value represents the future result of a *staged computation*. Such encapsulation is necessary if the DSL value might be unavailable when the host language code is evaluated; e.g., a value in a pixel shader DSL is only available when executing on the GPU. Hiding the DSL value is also convenient if its use in the host language is less important; e.g., a value in a databinding DSL is primarily meant to establish a binding between multiple dependency properties and the "current" value is only of limited use and should not normally be used.

Depending on the semantics of the DSL, a placeholder value can support most of the basic operations of the DSL value it wraps in addition to other operations that are relevant to the DSL. For example, both `Signal<double>` and `Shader<double>` support `+`, `-`, `*`, and `/`. How such operations are supported depends on the semantics and limitations of the DSL. For example, 64-bit double operations are mapped to 32-bit float operations in a pixel shader DSL because this is what is supported. Also, many operations in the databinding DSL can automatically be inverted in some way; e.g., `z = x + y` can automatically be inverted by the DSL implementation as `z - y = x` but not as `z - x = y`, while `z = x / y` can be inverted as `z * y = x` but not as `x / z = y`. This convention allows the databinding DSL to support the reuse of expressions that can appear on both sides of a constraint equation as well as transparently support two-way databinding. Consider `widgetA.HorizontalCenter.Bind = widgetB.HorzintalCenter`, which according to `HorizontalCenter`'s definition expands out to:

```
(widgetA.Left+widgetA.Width/2).Bind=widgeB.Left+widgetB.Width/2;
```

The databinding DSL's implementation then automatically rewrites this constraint using the specified inverse of `+`:

```
widgetA.Left.Bind=widgeB.Left+widgetB.Width/2-widgetA.Width/2;
```

The above constraint is then directly translated by the databinding DSL into the appropriate WPF databinding operations. Note that `+` along with other operations that can be inverted are no longer technically commutative. For example, the expression `Left + Width / 2` no longer has the same meaning as `Width / 2 + Left`: in the former case, widget `A` will be moved as widget `B` moves or is resized, while in the latter case, widget `A` will be resized as widget `B` moves or is resized. By sacrificing commutativity, expressions in databinding DSL become reusable and the DSL begins to resemble a more powerful imperative constraint language such as Kaleidoscope [8]. Expressions over placeholder values can even be combined together to form higher-level properties that are not directly provided in WPF. Consider:

```
LeftTop        PointSg.Make(Left, Top);
RightBottom    PointSg.Make(Left+Width,Top+Height);
CenterPosition PointSg.Make(Left+Width/2,Top+Height/2);
```

The `PointSg.Make` method combines two double placeholder values of the databinding DSL into a point placeholder value, which when bound will bind pair-wise each component. Example:

```
widgetA.RightBottom.Bind = widgetB.CenterPosition
```

This code is rewritten by the databinding DSL implementation into two component binding expressions:

```
widgetA.Left.Bind=widgetB.Left + widgetB. Width/2 - widgetA.Width
widgetA. Top.Bind=widgetB. Top + widgetB.Height/2 - widgetA.Height
```

In contrast to the databinding DSL, the expressions in the pixel shader DSL are used in simple pixel transforming functions. Example:

```
widget.Effect = Shaders.Apply<double>(slider.Value,
 (Shader<Brush> input,Shader<Point> uv,Shader<double> v) => {
 Shader<Color> orig = input[uv];
 Shader<double> gs = (orig.R + orig.G + orig.B) / 3;
 return Lerp(orig, FromArgb(gs,gs,gs,input[uv].A), v);     });
```

This C# code completely expresses the pixel shader used as an example in Section 2– no other boilerplate is necessary. The `Shaders.Apply` method takes as an argument shader logic that is encoded in the pixel shader DSL, and returns a fully configured effect for this pixel shader. The `Shader` type is then a wrapper around various built-in and WPF types, where the WPF type `Brush` is used to represent sampled textures. As appropriate, C# idioms are used to access pixel shader functionality; e.g., simple array access syntax in `input[uv]` replaces the explicit sample call in HLSL `tex2D(input, uv)`. The full power of C#'s IDE editor is also available, resulting in a more programmer-friendly environment for creating pixel shaders; e.g., the `R`, `G` and `B` properties are listed as possible code completions for the `orig` variable.

Because the parameters of a pixel shader are dependency properties, the `Shaders-.Apply` method can accept multiple placeholder values from the databinding DSL. In our example, `slider.Value` is of type `Signal<double>` and represents the slider's slide value. For each signal that is passed into the `Shaders.Apply` method, a corresponding shader value is passed into the pixel shader GPU block. For example `(input, uv, v) =>` is a preamble for pixel shader logic (using C# closure syntax) with standard arguments `input` and `uv`, which represent the texture and pixel coordinate being processed, along with intensity `v`, which represents the value of the `slider.Value` signal within the pixel shader and is accordingly of type `Shader<double>`. The implementation of the pixel shader DSL then transparently maps dependency property and pixel shader parameters to the correct registers, eliminating the need for the programmer to perform their own bookkeeping on parameter passing.

## 3.1 Lifting

Most embedded DSLs support either the automatic or manual lifting of code from the host language into the DSL, where *lifting* allows the code to eventually be applied to DSL values. By contrast, lifting in a lightweight DSL is either restricted or impossible. The databinding DSL supports the lifting of a code via a `MapSg` class that targets one or more signals. To support the databinding DSL's semantics, its `MapSg` class supports the specification of inverse code along with the code being lifted. Example:

```
Signal<double> result = new MapSg<double,double,double>
  (operandA, operandB, (x,y) => x + y, (z,y) => z - y);
```

This C# code defines the `operandA + operandB` operation of the databinding DSL where the first and second arguments are the signals involved in the operation while the third and fourth arguments are the code being lifted and this code's inverse, respectively. Obviously, not all operations support inversion, e.g., consider `Max` and `Min`. Such operations are lifted in the databinding DSL without support for inversions and then are restricted to only being used on the right side of a bind.

In contrast to the databinding DSL, the pixel shader DSL does not support lifting. The reason for this difference is that the expressions over databinding DSL values are eventually evaluated in hosting language inside a C# value converter of the form described in Section 2, while expressions over pixel shader DSL values are eventually evaluated on the GPU. Instead of lifting, the pixel shader DSL supports basic operations through HLSL's built-in operations. In both the databinding and pixel shader DSL, more complicated operations can be formed by composing expressions over placeholder values. Consider code that computes the area of a triangle in the databinding DSL:

```
Signal<double> TriangleArea(Signal<Point> A,
           Signal<Point> B, Signal<Point> C) {
 var t0 = (C.X - A.X) * (B.Y - A.Y);
 var t1 = (B.X - A.X) * (C.Y - A.Y);
 return .5 * (t0 - t1).Abs();        }
```

By expressing code in the databinding DSL as opposed to just lifting the function from C#, we can take advantage of inversion, although in this case it is of limited use. Equivalent code with different types is used to express triangle area for the pixel shader DSL:

```
Shader<double> TriangleArea(Shader<Point> A,
           Shader<Point> B, Shader<Point> C) {
 var t0 = (C.X - A.X) * (B.Y - A.Y);
 var t1 = (B.X - A.X) * (C.Y - A.Y);
 return .5 * (t0 - t1).Abs();        }
```

Unfortunately, because placeholder values in different DSLs have different types, code cannot easily be modularized across multiple DSLs. This code duplication could be eliminated lifting support, which the pixel shader DSL lacks. The typing problem can be solved and the duplicate definition eliminated in host languages that support dynamic typing or higher-kinded generics [19].

## 3.2 Full Compatibility and Staged Evaluation

The placeholder values of a lightweight DSL are first-class values in the host language meaning that they are data that can be passed into and returned by host language methods or stored in host language fields and variables. Additionally, placeholder values can still be composed via operations after they are used as data in the host language. Consider C# `Sample` method for a pixel shader DSL that finds the average color of the pixels surrounding a specified coordinate:

```
Shader<Color> Sample(Shader<Brush> input,Shader<Point> uv,
                              double radius,int npoint) {
 Shader<Color> ret = Colors.Transparent;
 for (int i = 0; i < npoint; i++) {
  Shader<Point> xy = uv +
   (i * (360d / npoint)).AsDegrees.SinCos() * radius;
  ret = ret + (input[xy] / npoint);
 } return ret; }
```

The first two arguments of `Sample` are placeholder values for the texture being sampled and the center point that is being sampled around. The second two arguments of `Sample` are host language values for the radius of the sample and the number of points that we want to sample around the center point. Since the latter two arguments are host language values, they are known immediately and not later on the GPU. The for loop along with the whole second term of the addition that forms `xy` are evaluated in C#, which is a form of partial evaluation. When the resulting placeholder values are evaluating on the GPU, the for loop is completely unrolled where the `sin` and `cos` operations have already been evaluated into constants. As all loops must be unrolled within a WPF pixel shader, no expressiveness or performance is lost from this approach.

Lightweight DSLs support the encoding of pure expressions and do not support imperative effects. However, this does not mean that imperative programming cannot be used to combine these expressions together: placeholder values can be stored and read from variables where this capability is very useful in practice. Consider the following code:

```
MyContainer container = ...;
Signal<Point> at = new Point(0,0);
foreach (UIElement e in container.Children)              {
  e.LeftTop.Bind = at;
  at = PointSg.Make(at.X + e.Size.X + container.HGap, 0); }
```

The above code lays out the children of a container from left to right. Since the `at` variable refers to databinding values, the layout will automatically adjust itself as the size of the children or gap changes. The interleaving of staged and unstaged code can be a bit confusing to programmers. However, since the lightweight DSL's code is formed from pure expressions that encode neither loops nor side effects, understanding this code as executing all at once is not far off from the resulting semantics. The only difference is that the `at` variable accumulates placeholder values that allows databinding over the next child's position, as opposed to just the current value of the next child's position.

By returning placeholder values from virtual methods and properties, we can augment lightweight DSLs with the more expressive object-oriented features of the host language. The lightweight DSLs themselves do not support objects–the values that they manipulate are limited to either primitives (double, int) or stateless structures (point, color). Given the ability to mix staged DSL code with unstaged host language code, OO features can be used in expressing behavior even if they do not directly influence the evaluation of DSL code. Example:

```
class MyContainer : Canvas
{ virtual Signal<double> HGap { get { return 10; } } }
```

The class `MyContainer` defines a virtual property `HGap` (horizontal gap) that is a databinding placeholder value of type `Signal` but is initially bound to the constant 10. As a result, the children of a direct instance of `MyContainer` are laid out horizontally with a 10 pixel gap. The `HGap` property can be overridden in a sub-class of `MyContainer` to return a different placeholder value:

```
class ExtendedContainer : MyContainer
{ Slider slider = ...;
  override Signal<double> HGap { get { return slider.Value; }}}
```

The children of direct instances of `ExtendedContainer` are now laid out horizontally with a gap that depends on the current value of the slider. As the slider's value changes, the layout will change automatically to account for the new gap value.

The use of object-oriented abstractions to organize and modularize DSL code is especially useful in building pixel shaders. HLSL is fairly primitive when compared to contemporary languages: only procedures and compiler directives can be used to modularize and configure HLSL code. In contrast, the pixel shader DSL provides support for object-oriented programming via unstaged host language code. Consider the following shader effect expressed in the pixel shader DSL:

```
Shader<Color> Average(Shader<Brush> input,
                      Shader<Point> uv, Shape geom)
{ Shader<Color> s = geom.Sample(input, uv, 8);
  return input[uv].Lerp(s, .5);                }
```

`Shape` is an abstract class that declares the `Sample` method that the `Rectangle` and `Circle` sub-classes implement differently. Since `geom` in the above code is a host language value, the call to `Sample` occurs unstaged and can take advantage of the C#'s support for virtual dispatch. If `geom` is bound to an instance of `Circle`, sampling occurs along a circle around point `uv` in texture `input`; while if `geom` is bound to an instance of `Rectangle`, sampling occurs along the edge of a rectangle. Through more aggressive use of object-oriented abstractions, pixel shader functionality can be reused in a framework fashion via object composition and filling in abstract methods. In this way, programming a pixel shader closely resembles other programming done in C#.

## 4 Implementation

Because a lightweight DSL is based on pure expressions, it does not need to be implemented through explicit meta-programming. Instead, the DSL can be implemented by building expressions trees through conventional method calls, ideally with sugar through lightweight syntax extension constructs such as operator overloading. Expression trees are processed according to the purpose of the DSL: a databinding DSL will populate and install binding objects, while a pixel shader DSL will generate HLSL code that is then compiled into a pixel shader. Without meta-programming, libraries require more preparation to be usable in the DSL. For example, in the databinding DSL each dependency property and operation must be explicitly "enabled" by writing some code that makes it useable in the DSL. On the other hand, no general interpreter needs to be implemented, access to library features that are not available in the DSL can be restricted by not enabling them in the DSL.

The amount of syntactic sugar that can support a lightweight DSL influences its usability. For example, Java's lack of support operator overloading means a Java-hosted lightweight DSL is very verbose. In contrast, Scala [20] supports the overloading of arbitrary operators and even have operators that are right associative. This paper focuses on C# as a host language, which lies somewhere between Java and Scala in terms of syntactic extensibility: most basic operators can be overloaded, new operators cannot be defined, and some operators (e.g., && but not &) are off limits. The language's support for automatic coercions also affects DSL syntax as it determines how constants are transformed into placeholder values. A couple of restrictions complicate lightweight DSL syntax in C#: operators and automatic coercions can only be defined in types that either targets. Placeholder value types are generic, e.g., `Signal<T>` and `Shader<T>`, and so operators and automatic coercions cannot be defined within specific instances of these types, e.g., `Signal<double>` or `Shader<float>`. Instead, we must create "brand" types for specific instances of the generic placeholder type, such as `DoubleSg` for `Signal<double>` and `FloatSh` for `Shader<float>`. Such types can then hold operators and coercions; e.g., the add operator for `DoubleSg` or a coercion from float to an instance of `FloatSh`. All operations should consume and produce brand types to ensure that coercions occur on arguments and that operators are available on result types. Methods that return or consume generic placeholder types, such as `Map` or `Bind`, cannot return a brand type and so the user must explicit convert values to achieve the brand type or force a coercion. Example:

```
Signal<double> result0 =
 new MapSg(x0, 20d.Sg(), (x,y) => x + y, (z,y)=>z-x);
DoubleSg result1 = result0.Sg();
DoubleSg result2 = result1 + 10;
```

Map operators over generic placeholder value types. Since "20d" is not a databinding DSL placeholder value, we must manually convert it into one via a call to `Sg`, which is an extension method that coerces to `DoubleSg` values. Because `Map` also returns a generic value, we must call the `Sg` method on the result value of type `Signal<double>` into a value of type `DoubleSg`, which then allows access operators like plus. The need for branded types is a limitation of C# and would not necessarily be needed for lightweight DSLs that were implemented in other languages, such as Scala.

## 4.1 Databinding DSL Implementation

Evaluation in the databinding DSL is implemented through what amounts to functional origami: functions associated with non-leaf expression nodes are applied and inverted to form the flat logic that is used to combine leaf dependency properties into the expression's value. Along the way, a binding object is populated with dependency properties where the flat logic forms the value converter of the binding. This is all enabled in a generic signal class that represents all placeholder values in the databinding DSL:

```
abstract class Signal<T> {
 Signal<T> Bind {  set {
  MultiBinding binding = new MultiBinding();
  this.Write(binding, value.Read(binding));
 }     };
 abstract Func<object[],T> Read(MultiBinding binding);
 abstract void Write(MultiBinding binding,Func<object[],T> fX);}
```

C#'s property syntax is used so that `Bind` resembles a property with only a setter: in `c.Bind = v`, the `set` handler is called with `value` bound to `v`. The `Bind` setter creates a multibinding object, populates it by reading from right side of the bind, and installs it by writing to the left side. The `Read` method returns an "accessor" function that computes the signal's value when given a list of all dependency property values in the binding, which will be provided by a value converter. The `Write` method then consumes an accessor function for the expression being bound to. The leaf expression trees of the databinding DSL are dependency object/property pairs known as unit signals:

```
class UnitSg<T> : Signal<T> {
 DependencyObject target; DependentProperty property;
 override Func<object[],T> Read(MultiBinding binding) {
  int index = binding.Bindings.Count;
  binding.Bindings.Add(new Binding(target, property));
  return params => (T) params[index];    }

 override void Write(MultiBinding binding,Func<object[],T> fX){
  binding.Converter = (IMultiValueConverer) fX;
  BindingOperations.SetBinding(target, property, binding);   }}
```

The type parameter `T` is bound to the understood type of the dependency property; e.g., for the `LeftProperty` of widget `w`, a unit signal is created with `new UnitSg<double>(w, LeftProperty)` where the left property's type is `double`. The `Read` method of `UnitSg` adds the dependency object and property to the binding being formed, and returns an accessor function that uses the index of the binding to access the dependency object/property's value in the value converter. The `Write` method then converts `fX`, which is the accessor function of the expression being bound to, into the value converter of the binding. The non-leaf expression trees of the databinding DSL are map operations that compose two signals:

```
class MapSg<T,R,S> : Signal<T> {
 Signal<R> argA; Signal<S> argB;
 Function<R,S,T> f; Function<T,S,R> g;
 override Func<object[],T> Read(MultiBinding binding)
 { Function<object[],R> fA = argA.Read(binding);
   Function<object[],S> fB = argB.Read(binding);
   return params => f(fA(params), fB(params));        }

 override void Write(MultiBinding binding,Func<object[],T> fX)
 { Function<object[],S> fB = argB.Read(binding);
   argA.Write(binding, params => g(fX(params), fB(params))); }}
```

The `Read` method of `MapSg` populates each argument of the map by reading then using the resulting accessor functions (`fA` and `fB`) combined with the map's code function (`f`) to create a an accessor function that computes the maps value in the value converter. The `Write` method of `MapSg` encodes the trick that allows databinding DSL expressions to be reused on either side of the bind: we read the second argument and then combine its accessor function with the accessor function of the expression being bound to via an inverse function `g`. The result of `g`'s application forms a new accessor function that removes the map's second argument from the bound-to expression, given whatever remove means for the map's operation; e.g., for addition this means subtraction. This new accessor function is then used in the `Write` call on the map's first argument. For example, `x + y = z` is transformed into an install on `x` using the expression `z - y`.

C#'s extension method constructs allow signal expressions to be modularized as methods that are accessed through existing existing UI classes. Example:

```
static DoubleSg Left(this UIElement self)
{ return new UnitSg<double>(self, Canvas.LeftProperty).Sg(); }
```

The `Sg` call creates a branded placeholder value as described earlier in the section. Map's can then be used to define operators inside branded placeholder types. Consider an operator in `DoubleSg`:

```
static DoubleSg operator+(DoubleSg opA, DoubleSg opB)
{ return new MapSg<double,double,double>
    (opA,opB, (x,y)=>x+y, (z,y)=>z-y).Sg(); }
```

This code expresses the + operation as described before.

Variations of `UnitSg` and `MapSg` exist to handle different special situations. For example, some dependency properties in WPF exist in read/write pairs; e.g., `Actual-WidthProperty` is a read only property that provides the actual width of a widget, while `WidthProperty` is a write mostly property that allows width to be encoded manually: we combine these properties into one signal for programmer convenience. Other classes exist to create points out of related properties and expressions; e.g., `Left` and `Top` double signals can be combined into the `LeftTop` point signal, which allows a 2D position to be bound in one bind statement.

### 4.2 Pixel Shader DSL Implementation

In contrast to the databinding DSL, pixel shader DSL evaluation occurs through generated HLSL code. Because pixel shaders are defined as functions, the DSL can automatically keep track of register indices and types, generating the preamble for the HLSL code without programmer intervention. Code generation is a matter of walking the expression tree and generating temporary variables for each operation that are used in successive operations.

WPF pixel shaders only support as inputs primitives and certain immutable values like brushes, points, and colors. These types in turn are mapped to a few types supported in HLSL: sampled textures, floats, and float tuples of an arity up to four. Programmers are not exposed to these low-level types: the mapping to low-level types occurs DSL during code generation. For example, a placeholder value of type `Shader<Color>` and `Shader<Point4D>` that the programmer uses will both map to a `float4` value in the generated HLSL code. Because high-level types are preserved in the DSL, the `Shader<Color>` type need only support properties and methods specific to color, whereas `float4` is a multi-use low-level type.

WPF pixel shader code that executes on a GPU is extremely restricted: no branching, only 96 arithmetic instructions, and no global memory. Because of these restrictions, pixel shaders are small and extremely easy to optimize. For this reason, we cannot observe a run-time performance difference between a pixel shader written in the DSL (with generated HLSL code) or hand written in HLSL. The pixel shader DSL also does not support any kind of control flow: one cannot create loops or branches as an expression tree. Instead, loops are unrolled while conditions are expressed via a `Condition` operator. However, the same loop unrolling and predication occurs to HLSL code through the HLSL compiler. The overhead of HLSL compilation is significant: on the order of tens of milliseconds is required to load the HLSL compiler and compile the generated code. However, this overhead is only paid for when the pixel shader is first created.

## 5 Discussion

The databinding and pixel shader DSLs improve over existing WPF programming practices by eliminating lots of boilerplate code that is difficult to write and interferes with reuse. Quantifying this improvement in terms of lines of code (LOC) is straightforward: for both DSLs, all code that does not encode logic needed to express the intended behavior is completely eliminated. However, benchmarking this reduction in real programs is difficult as the databinding and pixel shader technologies are rarely used given either their newness, the inconvenience of their abstractions, or their limited use. Given the last barrier, WPF databinding is designed for the very specific case of mapping application data into a user interface, while WPF pixel shader support is meant to build libraries of effects that are reused in multiple applications. However, the use cases of these technologies expands when they become easier to use.

Consider WPF databinding as a simple declarative constraint system. One well-known application of constraints in user interfaces is the flexible specification of layouts [4]. Typically, layout is expressed in WPF through panels that implement a specific layout scheme. For example, a `WrapPanel` packs widgets sequentially next to each other wrapping to the next row or column as needed, while a `DockPanel` allows widgets to be stuck to the edges of the panel. The standard practice of building complex layouts in WPF is to nest panels, which requires some up front planning and is still restricted in the layouts that can be expressed. In contrast, databinding can be used to express arbitrary layout constraints inside a `Canvas` panel: just bind the `Left`, `Top`, `Width`, and `Height` dependency properties. By wrapping these properties up into higher-level signals such as `LeftTop`, `CenterPosition`, and `Size`, we can express arbitrary positioning and size constraints on widgets with respect to each other. Example:

```
chessPiece.CenterPosition.Bind = container.Center;
gleam.CenterTop.Bind = chessPeice.CenterTop;
gleam.Rotate(chessPiece.CenterPosition - gleam.LeftTop).
  AnimateTo(360.Degrees);
```

This code centers a chess piece image on the screen with a gleam highlight stuck to its top. The gleam is then rotated around the chess piece's center by 360 degrees in an animation; we subtract gleam's left-top position from the piece's center position since a WPF rotation is specified relative to the target being rotated.

Another application of databinding is to synchronize multiple animations where standard practice in WPF is to use a `Storyboard`. The problem with this approach is that it only works through the animation engine–there is no way to grab and control the clock directly if we want to. Through databinding, animation can be specified as a constraint on a clock property:

```
DoubleSg Clock = ...; Rectangle rectA, rectB = ...;
rectA.Left.Bind = Clock * container.Width - rectA.Width;
rectB.Left.Bind = ((Clock - .5) * 2).Max(0) * container.Width -
  rectB.Width;
```

In this code, rectangle `A` moves across the screen according to the value of a `Clock` signal, which is between zero and one. Rectangle `B` moves across the screen during the last half of the clock, starting later and moving faster. To begin the animation, we animate clock; e.g., `Clock.Animate(0, 1)`. The clock can also be set directly to `.1`, `.5`, or `.75` to jump to states in the animation. We can even bind `Clock` to a slider, e.g., `Clock.Bind = slider.Value`, and enable user control of the animation. This latter technique is especially useful in debugging animations.

Custom pixel shaders on a per-application basis become desirable when UIs are enhanced with realistic lighting and other effects that further draw a user into a UI. Although shader code can be reused as an HLSL library of procedures, without the pixel shader DSL the programmer would still have to switch languages, set up their project, and write C# boilerplate code. With the pixel shader DSL, pixel shader code can be configured in C# through partial evaluation and C#'s conventional modularity constructs.

Beyond increased use, the pixel shader DSL improves modularity because pixel shader code that runs on the GPU can be located close to the external code that computes shader's parameters. Because shader parameters are dependency properties, the code used outside of the GPU can be expressed in the databinding DSL, creating a kind of synergy between the two DSLs. Consider the following method that distorts a widget based on the values of four relative corner positions (`ps`):

```
ShaderEffect Distort(PointSg[] ps) {
 DoubleSg volume = TriangleArea(ps[0], ps[1], ps[3]) +
                   TriangleArea(ps[2], ps[1], ps[3]);

 return ShaderEffects.Apply(volume, ps,
(TextureSh input,PointSh uv,FloatSh volumeX,PointSh[] psX) => {
  DoubleSh[] ds = new DoubleSh[4];
  for (int i = 0; i < 4; i++)
   ds[i] = TriangleArea(uv, psX[i], psX[(i + 1) % 4]);
  PointSh p = PointSh.Make(ds[3] / (ds[3] + ds[1]),
                           ds[0] / (ds[0] + ds[2]));
  DoubleSh volumeAt = ds[0] + ds[1] + ds[2] + ds[3];
  return (volumeAt <= volumeX) ? input[p] : Colors.Transparent;
 }); }
```

The first part of the `Distort` method computes the volume of the four relative bounding corners that distort the widget by adding the areas of the two dividing triangles together. Because this code is written in the databinding DSL, the `center` and `volume` variables refer to databinding placeholder values whose evaluations will change as the points are moved. The second part of this method creates the pixel shader using the `volume` value and the four corner points (`ps`), which are accessed through the `volumeX` and `psX` shader placeholder values within then pixel shader. Distortion is determined by computing the triangle areas of each corner edge and the current pixel (`uv`) to compute the horizontal and vertical position of the pixel (`p`) with respect to the four corners. To filter out pixels that lie outside of the polygon formed by the four corner points, we compare the volume at the current pixel (`volumeAt`) with the volume of the polygon that was computed through the databinding DSL. This comparison is then used to determine whether the distorted pixel (`input[p]`) or an empty pixel (`Transparent`) is returned, which ensures that the distortion is only rendered within its bounds.

## 6 Comparisons and Related Work

Our lightweight DSL approach are related to languages and techniques in multiple areas. Here we provide a deep, as opposed to a broad, overview of related work comparing the tradeoffs with our work as comprehensively as possible. The rest of this section describes dedicated DSLs (Section 6.1), meta-programming techniques (Section 6.2), embedded DSLs (Section 6.3), and host languages beyond C# (Section 6.4).

### 6.1 Dedicated DSLs and Specialized Languages

Dedicated DSLs are already involved in the construction of UIs: WPF already has XAML [18] to express graphics of the UI components along with declarative configuration, while HLSL [15] is used to write pixel shaders. Other UI markup language and shading languages exist for other UI platforms. Such UI languages are extremely limited: HLSL can only be used to encode pixel shader logic while XAML can only be used for certain parts of the UI. As a result, WPF solutions often involve multiple languages; e.g., XAML code, C# "code behind" to describe behavior that cannot be encoded in XAML, and HLSL code to implement the UI's pixel shaders. In contrast to this approach, our approach focuses on enhancing one host language through lightweight DSLs with the additional abstractions it needs to concisely encode UIs.

In contrast to WPF, Sun's JavaFX [21] takes a different approach with an integrated rich UI scripting language [22] that supports markup, animation, and databinding along with a complete OO language so that anything else can be effectively expressed. In particular, JavaFX Script's syntax for databinding is very similar to our databinding DSL. Consider:

```
var slider = SwingSlider{minimum: 0 maximum: 60 value: 0};
Stage {
 title: "Data Binding"
 scene: Scene {
  fill: Color.LIGHTGRAY;
   content: {
    slider, Circle {
     centerX: bind slider.value+50 centerY: 60 radius: 50
....
```

The last line of code binds the horizontal position of a circle to the slider's value added to 50 pixels. The equivalent databinding DSL code would be `circle.CenterPosition.X.Bind = slider.Value + 50`. JavaFX has advantages over our lightweight DSL approach by being able to support markup and being specifically optimized for UI. On the other hand, our approach can be hosted by an existing general purpose language and so does not require designing a new language and building the tools needed to support this language.

The work that we did on the databinding DSL is related to constraint programming languages that have often been oriented toward expressing UIs. As a representative example, Kaleidoscope [8] mixes objects, constraints, and imperative actions into a single integrated programming language. The resemblance between databinding in WPF and

Kaleidoscope constraints is striking: both distinguish between binding (`always`) and setting properties explicitly (`once`), and in both some properties are "held" in a more complicated equation where it is not clear which value should change. On the other hand, databinding is not as powerful as a constraint system: the constraints are limited to equality and relationships like greater than or within are not expressible. Still, the constraint equation syntax that the databinding DSL adds to C# brings general purpose languages closer to the capabilities of a constraint language like Kaleidoscope.

## 6.2 Meta-programming

Macros and other forms of meta-programming provide explicit support for extending a language with new abstractions. LISP macros [9] and derived macro systems allow code to be *quoted* as data that can then be traversed by a specialized evaluator to implement the semantics of a language extension. In contrast, a lightweight DSL forms expression trees explicitly through specialized APIs. LINQ [16] brings LISP-like macro capability to C# and extends C# with an embedded query language. LINQ's infrastructure enables code quoting so that it can be used to implement other kinds of DSLs. Example:

```
Expression<Func<Brush,Point,Color>> shader =
  (Brush input, Point uv) => {
  Color c = input.GetPixel(uv);
  var avg = (c.scR + c.scG + c.scB) / 3;
  return Color.FromArgb(avg, avg, avg, c.scA);  }
```

This code essentially encodes the logic of a grayscale pixel shader. Code within the quote undergoes syntactic and type checks as usual with the only difference being that expression trees are produced at run-time. Because the type of `shader` is `Expression-<...>`, the closure is transformed into data rather than a function that is ready for evaluation. This data can then either be compiled and executed, e.g., `shader.Compiler()-(texture, new Point(.5, .5))`, or the expression tree can undergo custom evaluation; e.g., generate a pixel shader. As for the latter custom evaluation, Brahma [1] can transform LINQ expression trees into HLSL code that can execute on a GPU to speed up certain computations.

The primary advantage of LINQ over our lightweight DSL approach includes the ability to completely re-interpret any C# syntax or re-direct any method call. In contrast, a lightweight DSL is limited to overloading certain operators and explicitly hooks into methods defined specifically for the DSL: we cannot re-interpert C# control flow primitives such as `if` and we cannot change the behavior of an existing method. On the other hand, our approach can be much cheaper for DSL implementors as it does not require building an expression tree traverser for a language with a rich syntax like C#. However, the main benefit of our approach is that it allows for expression trees to span C#'s abstraction boundaries (methods, virtual methods), which is very difficult to do in LINQ. As an example, the expression `TriangleArea(a, b,c)` is transformed by LINQ into a method call node: the `TriangleArea` method itself is not rendered as data! A LINQ re-interpreter must either manually associate `TriangleArea` method with special behavior or call it directly. The situation is much improved in a lightweight DSL: the version of the `TriangleArea` method that operates on placeholder values can explicitly be specified. For similar reasons, a lightweight DSL can support the mixing of host language and DSL values, while this is difficult or impossible to do in LINQ.

### 6.3 Embedded DSLs

Embedded DSLs reduce the effort needed to implement a DSL by embedding the DSL in a host language. A lightweight DSL is essentially an embedded DSL implemented through the lightweight techniques described in this paper. Embedded DSLs related to UI programming include those from the field of functional reactive programming (FRP) [6], which include the Fran and Yampa [10] embedded DSLs for Haskell as well FrTime [3] for Scheme. Our WPF databinding DSL is influenced and we call the placeholder values in this DSL "signals" to denote their similarity with FRP signals. However, signals are much more formal abstractions: they support continuous time with either synchronous or asynchronous semantics. FRP signals can also express expressions where signals are accessed through other signals. Consider the expression `SelectedClock.Time` where both `SelectedClock` and the `Time` of a clock changes over time: this can be expressed in an FRP but not in the databinding DSL given limitations in the WPF databinding engine. On the other hand, our WPF databinding DSL is very useful as a "FRP-lite" that allows programmers to leverage many of the benefits of FRP without switching languages.

Various embedded DSLs also exist to express pixel shaders. With Sh [12, 13], pixel shaders are encoded in C++ using a "retained-mode" technique that strongly resembles our lightweight DSL technique: operating overloading, rather than template metaprogramming, is used to construct expression trees that are translated into GPU code at run-time. As with our approach, the full power of the host language (C++) is available to modularize shader code. Our contribution over this work is generalizing the technique to other languages, e.g., C#, and features, e.g., databinding. Shaders can also be expressed in embedded DSLs via PyFX [11] on Python or Vertigo [5] on Haskell. The benefits of these embedded DSLs match the strengths of their host languages; i.e., Python's dynamism and Haskell's strong typing and formalisms.

### 6.4 More Host Language Possibilities

Although this paper has focused on C# as a host language, we originally thought of lightweight DSLs to work with Scala [20] as a host language. Scala provides many features that especially support lightweight DSL specification: covariance, GADTs, type constructor polymorphism [19], implicit parameters and coercions, mixins, and very flexible operator overloading. With these features, the issues with our C# DSLs are not present in Scala; e.g., type constructor polymorphism allows us to abstract over different DSLs to avoid the duplication issue described in Section 3.1. On the other hand, our use of C# as a host language has been very effective and the convenience of being able to use a mainstream language counteracts these drawbacks to a point. We believe that over time either mainstream languages or those language that become mainstream will have more features that would benefit our technique.

Lightweight DSL techniques can also be effectively applied to languages like Java that do not support operator overloading, as shown in jMock [7]. In this case, method calls replace operators while more explicit conversions may be required. Although this verbose syntax can make lightweight DSLs somewhat inconvenient for programmers, they can still benefit from the enhanced reuse and complexity reductions that the DSLs provide.

## 7 Conclusions and Future Work

This paper has shown how to construct lightweight DSLs that improve the programming of rich UIs. By avoiding meta-programming and leveraging a host language's own extensibility constructs, lightweight DSLs are easy to implement and so can pay-off quickly. We developed the databinding and pixel shader DSLs over the course of two weeks to support our local prototyping effort, and our informal experience has been that we are able to produce rich UIs much faster than if conventional WPF approaches were used. Beyond improved syntax, the ability of using the host language to fully manipulate lightweight DSL code enables high-level UI constructs such as object-oriented pixel shaders or constraint-based layouts.

Future work includes exploring what other aspects of a rich UI library can be improved through their own lightweight DSLs. Promising candidate aspects include geometry, lighting, camera manipulation, event processing, and physics-driven interaction. In the future, the kinds of expressions manipulated in a lightweight DSL are also bound to get more complicated; e.g., sophisticated shaders that are not currently supported by WPF support dynamic looping, dynamic branching, and even limited forms of side effects. The expressiveness of lightweight DSL syntax should be improved to support these features–perhaps through a more extensible host language such as using Scala rather than C#. Alternatively, we could reconsider the use of meta-programming by enhancing it to support the seamless mixing of DSL code with host language code.

The two lightweight DSLs described in this paper are available in a library called Bling WPF. The library along with its source code can be downloaded from Codeplex at `http://www.codeplex.com/bling`.

## References

1. Ananth. *Brahma*. http://brahma.ananthonline.net/.
2. Apple Computer. *Core Animation*. http://www.apple.com/macosx/technology/coreanimation.html.
3. G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *ESOP*, pages 294–308, 2006.
4. J. Coutaz. A layout abstraction for user-system interface. *SIGCHI Bull.*, 16(3):18–24, 1985.
5. C. Elliott. Programming graphics processors functionally. In *Proceedings of the 2004 Haskell Workshop*. ACM Press, 2004.
6. C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of ICFP*, volume 32 (8) of *SIGPLAN Notices*, pages 263–273. ACM, 1997.
7. S. Freeman and N. Pryce. Evolving an embedded domain-specific language in java. In *Proceedings of OOPSLA*, pages 855–865, New York, NY, USA, 2006. ACM.
8. B. N. Freeman-Benson. Kaleidoscope: Mixing objects, constraints and imperative programming. In *Proceedings of OOPSLA and ECOOP*, volume 25 (10) of *SIGPLAN Notices*, pages 77–88. ACM, 1990.

9. P. Graham. Common lisp macros. *AI Expert*, 3(3):42–53, 1987.
10. P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming. In *Advanced Functional Programming*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer, 2002.
11. C. Lejdfors and L. Ohlsson. Pyfx: A framework for programming real-time. Technical report, Lund University, 2005.
12. M. McCool and S. D. Toit. *Metaprogramming GPUs with Sh*. A K Peters, 2004.
13. M. McCool, S. D. Toit, T. Popa, B. Chan, and K. Moule. Shader algebra. In *Proceedings of SIGGRAPH*, pages 787–795, New York, NY, USA, 2004. ACM.
14. Microsoft. *DirectX*. http://msdn.microsoft.com/en-us/library/aa139763.aspx.
15. Microsoft. *High Level Shader Language (HLSL)*. http://msdn.microsoft.com/en-us/library/bb509561(VS.85).aspx.
16. Microsoft. *The LINQ Project*. http://msdn.microsoft.com/en-us/vbasic/aa904594.aspx.
17. Microsoft. *Windows Presentations Foundation (WPF)*. http://msdn.microsoft.com/en-us/netframework/aa663326.aspx.
18. Microsoft. *XAML*. http://msdn.microsoft.com/en-us/library/ms752059.aspx.
19. A. Moors, F. Piessens, and M. Odersky. Generics of a higher kind. In *OOPSLA*, pages 423–438, New York, NY, USA, 2008. ACM.
20. M. Odersky and M. Zenger. Scalable component abstractions. In *Proceedings of OOPSLA*, pages 41–57, New York, NY, USA, 2005. ACM.
21. Sun Microsystems. *JavaFX*. http://www.sun.com/software/javafx/index.jsp.
22. Sun Microsystems. *JavaFX Script*. http://www.sun.com/software/javafx/script/.
23. Sun Microsystems. *Project Scene Graph Effects*. https://scenegraph-effects.dev.java.net/.