

Adding dynamic types to C#

Gavin Bierman

Microsoft Research

Joint work with Mads Torgersen and Erik Meijer

ECOOP, June 2010

Today's themes

Today's themes

Fashion:



Today's themes

Fashion:



Mystery:



Today's themes

Fashion:



Mystery:



Maths:

Today's themes

Fashion:



Mystery:



Maths:



Congrats Erik!

Theme #1: Fashion



What are C# devs doing?
What's difficult to do?
What would they like to do?

Helping the customers

1. Silverlight programming: inter-op with Javascript objects
2. Office inter-op
3. Dynamic languages envy

Helping the customers

1. Silverlight programming: inter-op with Javascript objects
 - ▶ Lots of annoying casts
2. Office inter-op
3. Dynamic languages envy

Helping the customers

1. Silverlight programming: inter-op with Javascript objects
 - ▶ Lots of annoying casts
 - ▶ Spin: strongly typed
2. Office inter-op
3. Dynamic languages envy

Helping the customers

1. Silverlight programming: inter-op with Javascript objects
 - ▶ Lots of annoying casts
 - ▶ Spin: strongly typed
 - ▶ Reality: **stringly typed**
2. Office inter-op
3. Dynamic languages envy

Helping the customers

1. Silverlight programming: inter-op with Javascript objects
 - ▶ Lots of annoying casts
 - ▶ Spin: strongly typed
 - ▶ Reality: **stringly typed**
2. Office inter-op
 - ▶ Lots of annoying casts
3. Dynamic languages envy

Helping the customers

1. Silverlight programming: inter-op with Javascript objects
 - ▶ Lots of annoying casts
 - ▶ Spin: strongly typed
 - ▶ Reality: **stringly typed**
2. Office inter-op
 - ▶ Lots of annoying casts
3. Dynamic languages envy
 - ▶ Ha ha ha ha ...

Visual Studio 2010



- ▶ Launched April 12, 2010
- ▶ Many great things (F#!), but...
- ▶ One big addition: **Dynamic Language Runtime (DLR)**
 - ▶ To support dynamic languages on the CLR
 - ▶ Shared infrastructure for IronPython and IronRuby compilers
 - ▶ Source code available! (from codeplex)
 - ▶ Can be used by any DL compiler writer e.g. Nua, Vedeo.

Helping the customers

1. Silverlight programming: inter-op with Javascript objects
 - ▶ Lots of annoying casts
 - ▶ Spin: strongly typed
 - ▶ Reality: **stringly typed**
2. Office inter-op
 - ▶ Lots of annoying casts
3. Dynamic languages envy
 - ▶ Ha ha ha ha ...

Helping the customers

1. Silverlight programming: inter-op with Javascript objects
 - ▶ Lots of annoying casts
 - ▶ Spin: strongly typed
 - ▶ Reality: **stringly typed**
2. Office inter-op
 - ▶ Lots of annoying casts
3. Dynamic languages envy
 - ▶ Ha ha ha ha ...



Can't we use the DLR?

Theme # 2: Mystery



Taken from: Sherlock Holmes and the temple of Hejlsberg

What's going on?

Have we hosed the type system?

[Taken from: "Gradual Types for objects" (Siek and Taha)]

Have declarative type system, with subtype rule:

$$\frac{\Gamma \vdash e : T \quad T \leq S}{\Gamma \vdash e : S}$$

and add to subtype relation:

Have we hosed the type system?

[Taken from: "Gradual Types for objects" (Siek and Taha)]

Have declarative type system, with subtype rule:

$$\frac{\Gamma \vdash e : T \quad T \leq S}{\Gamma \vdash e : S}$$

and add to subtype relation:

`dynamic` $x = e;$

Have we hosed the type system?

[Taken from: "Gradual Types for objects" (Siek and Taha)]

Have declarative type system, with subtype rule:

$$\frac{\Gamma \vdash e : T \quad T \leq S}{\Gamma \vdash e : S}$$

and add to subtype relation:

`dynamic` $x = e;$

$$\frac{}{T \leq \text{dynamic}}$$

Have we hosed the type system?

[Taken from: "Gradual Types for objects" (Siek and Taha)]

Have declarative type system, with subtype rule:

$$\frac{\Gamma \vdash e : T \quad T \leq S}{\Gamma \vdash e : S}$$

and add to subtype relation:

`dynamic` $x = e;$

$$\frac{}{T \leq \text{dynamic}}$$

`SomeStaticMethodExpectingS(x);`

Have we hosed the type system?

[Taken from: "Gradual Types for objects" (Siek and Taha)]

Have declarative type system, with subtype rule:

$$\frac{\Gamma \vdash e : T \quad T \leq S}{\Gamma \vdash e : S}$$

and add to subtype relation:

`dynamic` $x = e;$

$$\frac{}{T \leq \text{dynamic}}$$

`SomeStaticMethodExpectingS(x);`

$$\frac{}{\text{dynamic} \leq S}$$

Have we hosed the type system?

[Taken from: "Gradual Types for objects" (Siek and Taha)]

Have declarative type system, with subtype rule:

$$\frac{\Gamma \vdash e : T \quad T \leq S}{\Gamma \vdash e : S}$$

and add to subtype relation:

`dynamic` $x = e;$

$$\frac{}{T \leq \text{dynamic}}$$

`SomeStaticMethodExpectingS(x);`

$$\frac{}{\text{dynamic} \leq S}$$

$$\text{[Transitivity]} \frac{T \leq \text{dynamic} \quad \text{dynamic} \leq S}{T \leq S}$$

Have we hosed the type system?

[Taken from: "Gradual Types for objects" (Siek and Taha)]

Have declarative type system, with subtype rule:

$$\frac{\Gamma \vdash e : T \quad T \leq S}{\Gamma \vdash e : S}$$

and add to subtype relation:

`dynamic` $x = e;$

$$\frac{}{T \leq \text{dynamic}}$$

`SomeStaticMethodExpectingS(x);`

$$\frac{}{\text{dynamic} \leq S}$$

$$\text{[Transitivity]} \frac{T \leq \text{dynamic} \quad \text{dynamic} \leq S}{T \leq S}$$

Oops! S+T solution: Drop transitivity ☹

Theme #3: Maths



What's going on?

But for the non-mathematically inclined...

Summary:

But for the non-mathematically inclined...

Summary:

- ▶ We did the proper "featherweight" thing

But for the non-mathematically inclined...

Summary:

- ▶ We did the proper "featherweight" thing
- ▶ It works!
 - ▶ `dynamic` can be added as an extension not a breaking change!

Conversion versus synthesis in C#

Type conversion

$$\begin{array}{c} T \ x = e; \\ \Downarrow \\ \vdash e <:; T \end{array}$$

Type synthesis

$$\begin{array}{c} \text{var } y = e; \\ \Downarrow \\ \vdash e \uparrow T \end{array}$$

Conversion versus synthesis in C#

Type conversion

$$\begin{array}{c} T \ x = e; \\ \Downarrow \\ \vdash e <: T \end{array}$$

Type synthesis

$$\begin{array}{c} \text{var } y = e; \\ \Downarrow \\ \vdash e \uparrow T \end{array}$$

```
Button x = null; //Works
```

Conversion versus synthesis in C#

Type conversion

$$\begin{array}{c} T \ x = e; \\ \Downarrow \\ \vdash e <: T \end{array}$$

```
Button x = null; //Works
```

Type synthesis

$$\begin{array}{c} \text{var } y = e; \\ \Downarrow \\ \vdash e \uparrow T \end{array}$$

```
var y = null; //Fails
```

Implicit type conversions (aka subtyping)

"Subtyping with coercions"

$$\text{[IC-Ref]} \frac{}{\sigma_1 <:_i \sigma_1 \rightsquigarrow \bullet}$$

$$\text{[IC-ByteToInt]} \frac{}{\text{byte} <:_i \text{int} \rightsquigarrow \text{ByteToInt}(\bullet)}$$

$$\text{[IC-Val-Obj]} \frac{}{\gamma <:_i \text{object} \rightsquigarrow \text{Box}[\gamma](\bullet)}$$

$$\text{[IC-Sub]} \frac{C_1 \langle \overline{\sigma}_1 \rangle : C_2 \langle \overline{\sigma}_2 \rangle}{C_1 \langle \overline{\sigma}_1 \rangle <:_i C_2 \langle \overline{\sigma}_2 \rangle \rightsquigarrow \bullet}$$

C[#] bidirectional type system: more fully

Checking $\Gamma \vdash e <:_i \sigma$ “ e can be implicitly converted to σ ”

Synthesis $\Gamma \vdash e \uparrow \sigma$ “ e synthesizes a type σ ”

C[#] bidirectional type system: more fully

Checking $\Gamma \vdash e <:_i \sigma \rightsquigarrow E$ “ e can be implicitly converted to σ
yielding E ”

Synthesis $\Gamma \vdash e \uparrow \sigma \rightsquigarrow E$ “ e synthesizes a type σ
yielding E ”

Implicit conversion relation

Key rule:

$$[\text{IC-Synth}] \frac{}{\Gamma \vdash e_1 <:_i \sigma_1 \rightsquigarrow}$$

Implicit conversion relation

Key rule:

$$[\text{IC-Synth}] \frac{\Gamma \vdash e_1 \uparrow \sigma_0 \rightsquigarrow E_1}{\Gamma \vdash e_1 <:_j \sigma_1 \rightsquigarrow}$$

Implicit conversion relation

Key rule:

$$[\text{IC-Synth}] \frac{\Gamma \vdash e_1 \uparrow \sigma_0 \rightsquigarrow E_1 \quad \sigma_0 <:_j \sigma_1 \rightsquigarrow C}{\Gamma \vdash e_1 <:_j \sigma_1 \rightsquigarrow}$$

Implicit conversion relation

Key rule:

$$[\text{IC-Synth}] \frac{\Gamma \vdash e_1 \uparrow \sigma_0 \rightsquigarrow E_1 \quad \sigma_0 <:_i \sigma_1 \rightsquigarrow C}{\Gamma \vdash e_1 <:_i \sigma_1 \rightsquigarrow C[E_1]}$$

Dealing with `dynamic`

1. Add *one* new rule to type conversion/subtyping

$$[\text{IC-Dynamic}] \frac{}{\sigma <: \text{dynamic} \rightsquigarrow}$$

Dealing with `dynamic`

1. Add *one* new rule to type conversion/subtyping

$$[\text{IC-Dynamic}] \frac{}{\sigma <:_{\text{j}} \text{dynamic} \rightsquigarrow ?}$$

Dealing with `dynamic`

1. Add *one* new rule to type conversion/subtyping

$$\text{[IC-Dynamic]} \frac{\sigma <:_{\text{j}} \text{object} \rightsquigarrow C}{\sigma <:_{\text{j}} \text{dynamic} \rightsquigarrow C}$$

Dealing with `dynamic`, continued

2. Add special `dynamic` case for all rules that synthesize types in their hypotheses.

Dealing with `dynamic`, continued

2. Add special `dynamic` case for all rules that synthesize types in their hypotheses.

Recall:

$$\text{[IC-Synth]} \frac{\Gamma \vdash e_1 \uparrow \sigma_0 \rightsquigarrow E_1 \qquad \sigma_0 <_i \sigma_1 \rightsquigarrow C}{\Gamma \vdash e_1 <_i \sigma_1 \rightsquigarrow C[E_1]}$$

Dealing with `dynamic`, continued

2. Add special `dynamic` case for all rules that synthesize types in their hypotheses.

Recall:

$$\text{[IC-Synth]} \frac{\Gamma \vdash e_1 \uparrow \sigma_0 \rightsquigarrow E_1 \quad \sigma_0 \neq \text{dynamic} \quad \sigma_0 <_i \sigma_1 \rightsquigarrow C}{\Gamma \vdash e_1 <_i \sigma_1 \rightsquigarrow C[E_1]}$$

Dealing with `dynamic`, continued

2. Add special `dynamic` case for all rules that synthesize types in their hypotheses.

Recall:

$$[\text{IC-Synth}] \frac{\Gamma \vdash e_1 \uparrow \sigma_0 \rightsquigarrow E_1 \quad \sigma_0 \neq \text{dynamic} \quad \sigma_0 <_i \sigma_1 \rightsquigarrow C}{\Gamma \vdash e_1 <_i \sigma_1 \rightsquigarrow C[E_1]}$$

$$[\text{IC-Dynamic}] \frac{\Gamma \vdash e_1 \uparrow \text{dynamic} \rightsquigarrow E_1}{\Gamma \vdash e_1 <_i \sigma_1 \rightsquigarrow}$$

Dealing with `dynamic`, continued

2. Add special `dynamic` case for all rules that synthesize types in their hypotheses.

Recall:

$$\text{[IC-Synth]} \frac{\Gamma \vdash e_1 \uparrow \sigma_0 \rightsquigarrow E_1 \quad \sigma_0 \neq \text{dynamic} \quad \sigma_0 <_{;i} \sigma_1 \rightsquigarrow C}{\Gamma \vdash e_1 <_{;i} \sigma_1 \rightsquigarrow C[E_1]}$$

$$\text{[IC-Dynamic]} \frac{\Gamma \vdash e_1 \uparrow \text{dynamic} \rightsquigarrow E_1}{\Gamma \vdash e_1 <_{;i} \sigma_1 \rightsquigarrow \text{Convert}[\sigma_1](E_1 : \text{dynamic})}$$

where `Convert` is the (runtime) implicit conversion (subtype) test and coercion insertion code.

Dealing with `dynamic`: method invocation (Dynamic receiver)

$\Gamma \vdash e_1.m\langle\bar{\sigma}_1\rangle(\bar{e}_2) \uparrow \quad \rightsquigarrow$

Dealing with `dynamic`: method invocation (Dynamic receiver)

$$\Gamma \vdash e_1 \uparrow \text{dynamic} \rightsquigarrow E_1$$

$$\Gamma \vdash e_1.m\langle\bar{\sigma}_1\rangle(\bar{e}_2) \uparrow \rightsquigarrow$$

Dealing with `dynamic`: method invocation (Dynamic receiver)

$$\Gamma \vdash e_1 \uparrow \text{dynamic} \rightsquigarrow E_1$$

$$\Gamma \vdash e_1.m\langle\bar{\sigma}_1\rangle(\bar{e}_2) \uparrow \text{dynamic} \rightsquigarrow$$

Dealing with `dynamic`: method invocation (Dynamic receiver)

$$\Gamma \vdash e_1 \uparrow \text{dynamic} \rightsquigarrow E_1$$

$$\Gamma \vdash \bar{e}_2 \uparrow^+ \bar{\sigma} \rightsquigarrow \bar{E}_2$$

$$\Gamma \vdash e_1.m\langle\bar{\sigma}_1\rangle(\bar{e}_2) \uparrow \text{dynamic} \rightsquigarrow$$

Dealing with `dynamic`: method invocation (Dynamic receiver)

$$\Gamma \vdash e_1 \uparrow \text{dynamic} \rightsquigarrow E_1$$

$$\Gamma \vdash \bar{e}_2 \uparrow^+ \bar{\sigma} \rightsquigarrow \bar{E}_2$$

$$\Gamma \vdash e_1.m\langle\bar{\sigma}_1\rangle(\bar{e}_2) \uparrow \text{dynamic} \rightsquigarrow \text{MInvoke}[m](E_1 : \text{dynamic}, \bar{E}_2 : \bar{\sigma})$$

Dynamic operators

Treated in paper as if they are special constructs in the target language, but in reality they're calls to methods in the DLR.

$DE ::=$

$\text{Convert}[\sigma](E: \sigma)$

$\text{MemberAccess}[f](E: \sigma)$

$\text{DInvoke}(E: \sigma, \overline{E}: \sigma)$

$\text{ObjectCreate}[\rho](\overline{E}: \sigma)$

$\text{MInvoke}[m](E: \sigma, \overline{E}: \sigma)$

$$\frac{|H| \vdash o: \sigma_1 <: \sigma_2 \rightsquigarrow E}{\langle H, ST, \text{Convert}[\sigma_2](o: \sigma_1), FS \rangle \rightsquigarrow \langle H, ST, E, FS \rangle}$$

Usual properties (preservation and progress)

Related work

[Too many to mention!]

Highly relevant:

- ▶ Siek and Taha. *Gradual typing for objects*. ECOOP'07.
- ▶ Ina and Igarashi. *Towards gradual typing for generics*. STOP'09.
- ▶ Wrigstad et al. *Integrating typed and untyped code in a scripting language*. POPL'10.

[Apologies to other "Highly relevant" authors☺]

Conclusions

Aims:

1. Adding `dynamic` to C#:
 - ▶ Why was it added?
 - ▶ Why is it useful?
 - ▶ How was it done?
2. Bidirectional type systems:
 - ▶ Elegant theory
 - ▶ Matches real world!
 - ▶ Provides nice way of supporting `dynamic` compared to previous work

Questions?

