Adding dynamic types to C$^\sharp$

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$^\sharp$ 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$^\sharp$!), 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, Vedea.

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:

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

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:

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

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:

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

and add to subtype relation:

`dynamic` x = e;

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

Have we hosed the type system?

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

Have declarative type system, with subtype rule:

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

and add to subtype relation:

$\texttt{dynamic}\ x = e;$

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

$\texttt{SomeStaticMethodExpectingS(}\ x\ \texttt{);}$

Have we hosed the type system?

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

Have declarative type system, with subtype rule:

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

and add to subtype relation:

$\texttt{dynamic}\ x = e;$

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

$\texttt{SomeStaticMethodExpectingS}(\ x\ );$

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

Have we hosed the type system?

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

Have declarative type system, with subtype rule:

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

and add to subtype relation:

$\texttt{dynamic}\ x = e;$

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

$\texttt{SomeStaticMethodExpectingS}(\ x\ );$

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

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

Have we hosed the type system?

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

Have declarative type system, with subtype rule:

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

and add to subtype relation:

```
dynamic x = e;
```

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

```
SomeStaticMethodExpectingS( x );
```

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

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

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!

Bidirectional type systems: Revision

- First used by Pierce and Turner [1998]
- Distinguishes between checking and synthesis relations
- The difference is in the mode:

Checking	$\Gamma \vdash e <:_{\text{¡}} \sigma$	INPUTS:	Γ, e, σ
		OUTPUT:	Yes/No
Synthesis	$\Gamma \vdash e \uparrow \sigma$	INPUTS:	Γ, e
		OUTPUT:	σ

- It's very algorithmic ☺

Conversion versus synthesis in C$^\sharp$

Type conversion

```
T x = e;
```

$$\Downarrow$$

$\vdash$ `e` $<$:¡ `T`

Type synthesis

```
var y = e;
```

$$\Downarrow$$

$\vdash \mathtt{e} \uparrow \mathtt{T}$

Conversion versus synthesis in C$^\sharp$

Type conversion

```
T x = e;
```

$$\Downarrow$$

$$\vdash \texttt{e} <:_{¡} \texttt{T}$$

Type synthesis

```
var y = e;
```

$$\Downarrow$$

$$\vdash \texttt{e} \uparrow \texttt{T}$$

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

Conversion versus synthesis in C$^\sharp$

Type conversion

```
T x = e;
```

$$\Downarrow$$

$$\vdash \texttt{e} <:\text{¡}\ \texttt{T}$$

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

Type synthesis

```
var y = e;
```

$$\Downarrow$$

$$\vdash \texttt{e} \uparrow \texttt{T}$$

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

Implicit type conversions (aka subtyping)

"Subtyping with coercions"

$$[\text{IC-Refl}]\ \frac{}{\sigma_1 <:_{\text{¡}} \sigma_1 \leadsto \bullet}$$

$$[\text{IC-ByteToInt}]\ \frac{}{\texttt{byte} <:_{\text{¡}} \texttt{int} \leadsto \mathsf{ByteToInt}(\bullet)}$$

$$[\text{IC-Val-Obj}]\ \frac{}{\gamma <:_{\text{¡}} \texttt{object} \leadsto \mathsf{Box}[\gamma](\bullet)}$$

$$[\text{IC-Sub}]\ \frac{\texttt{C}_1\texttt{<}\overline{\sigma_1}\texttt{>} : \texttt{C}_2\texttt{<}\overline{\sigma_2}\texttt{>}}{\texttt{C}_1\texttt{<}\overline{\sigma_1}\texttt{>} <:_{\text{¡}} \texttt{C}_2\texttt{<}\overline{\sigma_2}\texttt{>} \leadsto \bullet}$$

C$^\sharp$ bidirectional type system: more fully

Checking	$\Gamma \vdash e <:\text{¡}\ \sigma$	"e can be implicitly converted to σ"
Synthesis	$\Gamma \vdash e \uparrow \sigma$	"e synthesizes a type σ"

C$^\sharp$ bidirectional type system: more fully

Checking	$\Gamma \vdash e <:¡\ \sigma \rightsquigarrow E$	"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:

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

Implicit conversion relation

Key rule:

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

Implicit conversion relation

Key rule:

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

Implicit conversion relation

Key rule:

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

Dealing with `dynamic`

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

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

Dealing with `dynamic`

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

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

Dealing with `dynamic`

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

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

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:

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

Dealing with `dynamic`, continued

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

   Recall:

$$[\text{IC-Synth}]\ \frac{\Gamma \vdash e_1 \uparrow \sigma_0 \rightsquigarrow E_1 \qquad \sigma_0 \neq \texttt{dynamic} \qquad \sigma_0 <:_{\text{¡}} \sigma_1 \rightsquigarrow C}{\Gamma \vdash e_1 <:_{\text{¡}} \sigma_1 \rightsquigarrow C[E_1]}$$

Dealing with `dynamic`, continued

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

   Recall:

$$[\text{IC-Synth}]\ \frac{\Gamma \vdash e_1 \uparrow \sigma_0 \leadsto E_1 \qquad \sigma_0 \neq \texttt{dynamic} \qquad \sigma_0 <:_{\text{i}} \sigma_1 \leadsto C}{\Gamma \vdash e_1 <:_{\text{i}} \sigma_1 \leadsto C[E_1]}$$

$$[\text{IC-Dynamic}]\ \frac{\Gamma \vdash e_1 \uparrow \texttt{dynamic} \leadsto E_1}{\Gamma \vdash e_1 <:_{\text{i}} \sigma_1 \leadsto}$$

Dealing with `dynamic`, continued

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

   Recall:

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

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

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

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

$$\frac{}{\Gamma \vdash e_1 . m\texttt{<}\overline{\sigma_1}\texttt{>}(\overline{e_2}) \uparrow \qquad \rightsquigarrow}$$

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

$$\frac{\Gamma \vdash e_1 \uparrow \texttt{dynamic} \rightsquigarrow E_1}{\Gamma \vdash e_1\,.\,m\texttt{<}\overline{\sigma_1}\texttt{>(}\overline{e_2}\texttt{)} \uparrow \qquad \rightsquigarrow}$$

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

$$\frac{\Gamma \vdash e_1 \uparrow \texttt{dynamic} \rightsquigarrow E_1}{\Gamma \vdash e_1\,.\,m\texttt{<}\overline{\sigma_1}\texttt{>(}\overline{e_2}\texttt{)} \uparrow \texttt{dynamic} \rightsquigarrow}$$

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

$$\frac{\Gamma \vdash e_1 \uparrow \texttt{dynamic} \leadsto E_1 \qquad \Gamma \vdash \overline{e_2} \uparrow^{+} \overline{\sigma} \leadsto \overline{E_2}}{\Gamma \vdash e_1\texttt{.}m\texttt{<}\overline{\sigma_1}\texttt{>(}\overline{e_2}\texttt{)} \uparrow \texttt{dynamic} \leadsto}$$

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

$$\frac{\Gamma \vdash e_1 \uparrow \texttt{dynamic} \rightsquigarrow E_1 \qquad \Gamma \vdash \overline{e_2} \uparrow^{+} \overline{\sigma} \rightsquigarrow \overline{E_2}}{\Gamma \vdash e_1\texttt{.}m\texttt{<}\overline{\sigma_1}\texttt{>(}\overline{e_2}\texttt{)} \uparrow \texttt{dynamic} \rightsquigarrow \mathsf{MInvoke}[m](E_1 : \texttt{dynamic}, \overline{E_2} : \overline{\sigma})}$$

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.

$$
\begin{array}{l}
DE ::= \\
\quad \mathsf{Convert}[\sigma](E\colon \sigma) \\
\quad \mathsf{MemberAccess}[f](E\colon \sigma) \\
\quad \mathsf{DInvoke}(E\colon \sigma, \overline{E\colon \sigma}) \\
\quad \mathsf{ObjectCreate}[\rho](\overline{E\colon \sigma}) \\
\quad \mathsf{MInvoke}[m](E\colon \sigma, \overline{E\colon \sigma})
\end{array}
$$

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

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$^\sharp$:
   - 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?