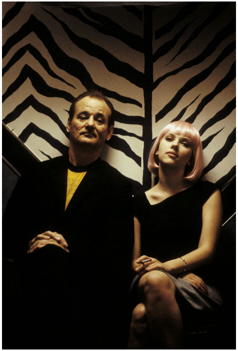
Lost in translation: Formalizing proposed extensions to C$^\sharp$

Gavin Bierman[1] Erik Meijer[2] Mads Torgersen[2]

[1]Microsoft Research, Cambridge

[2]Microsoft Corporation



ooPSLA 2007

LINQ (Language INtegrated Query)

- An approach to addressing the impedance mismatch problem
- Aim to make data programming:
  - Easy
  - Regular
  - Extensible
- Extend C$^\sharp$ and Visual Basic to make this programming even more pleasant

LINQ (Language INtegrated Query)

- An approach to addressing the impedance mismatch problem
- Aim to make data programming:
  - Easy
  - Regular
  - Extensible
- Extend C$^\sharp$ and Visual Basic to make this programming even more pleasant

And all this with no changes to the CLR (just some new libraries)

Demo!

Aims of “lost in translation”

Subject of study: The new language features

Several different perspectives:

- To provide a concrete semantics for C$^\sharp$3.0
- (Actually to formalize some C$^\sharp$2.0 features for the first time!)
- To show how the new features in C$^\sharp$3.0 can be compiled into existing C$^\sharp$2.0 features
- To demonstrate/prove that no changes to the CLR are needed

Aims of “lost in translation”

Subject of study: The new language features

Several different perspectives:

- To provide a concrete semantics for C$\sharp$3.0
- (Actually to formalize some C$\sharp$2.0 features for the first time!)
- To show how the new features in C$\sharp$3.0 can be compiled into existing C$\sharp$2.0 features
- To demonstrate/prove that no changes to the CLR are needed
    - To show how to build a C$\sharp$3.0 compiler on the cheap (but don’t tell the lawyers!)

Translation of C$^\sharp$3.0

Two stages:

1. Translate query expressions into method invocations
2. Translate the remaining C$^\sharp$3.0 code into C$^\sharp$2.0

Query expression compilation

```
var names =
   from c in Customers
   where c.City == "Redmond"
   where c.Credit > 10000
   orderby c.Age descending
   select c.Name;
```

Query expression compilation

```
var names =
   from c in Customers
   where c.City == "Redmond"
   where c.Credit > 10000
   orderby c.Age descending
   select c.Name;
```

```
var names =
   from c in Customers
      .Where((c)=>c.City=="Redmond")
   where c.Credit > 10000
   orderby c.Age descending
   select c.Name;
```

Query expression compilation

```
var names =
   from c in Customers
      .Where((c)=>c.City=="Redmond")
   where c.Credit > 10000
   orderby c.Age descending
   select c.Name;
```

Query expression compilation

```
var names =
   from c in Customers
      .Where((c)=>c.City=="Redmond")
   where c.Credit > 10000
   orderby c.Age descending
   select c.Name;
```

```
var names =
   from c in Customers
      .Where((c)=>c.City=="Redmond")
      .Where((c)=>c.Credit>10000)
   orderby c.Age descending
   select c.Name;
```

Query expression compilation

```
var names =
   from c in Customers
      .Where((c)=>c.City=="Redmond")
      .Where((c)=>c.Credit>10000)
   orderby c.Age descending
   select c.Name;
```

Query expression compilation

```
var names =
   from c in Customers
      .Where((c)=>c.City=="Redmond")
      .Where((c)=>c.Credit>10000)
   orderby c.Age descending
   select c.Name;
```

```
var names =
   from c in Customers
      .Where((c)=>c.City=="Redmond")
      .Where((c)=>c.Credit>10000)
      .OrderByDescending((c)=>c.Age)
   select c.Name;
```

Query expression compilation

```
var names =
   from c in Customers
      .Where((c)=>c.City=="Redmond")
      .Where((c)=>c.Credit>10000)
      .OrderByDescending((c)=>c.Age)
   select c.Name;
```

Query expression compilation

```
var names =
   from c in Customers
      .Where((c)=>c.City=="Redmond")
      .Where((c)=>c.Credit>10000)
      .OrderByDescending((c)=>c.Age)
   select c.Name;
```

```
var names =
   Customers
      .Where((c)=>c.City=="Redmond")
      .Where((c)=>c.Credit>10000)
      .OrderByDescending((c)=>c.Age)
      .Select((c)=>c.Name);
```

The query expression pattern (subset)

```
class C<T>
{
       public C<T> Where(Func<T,bool> predicate);
       public C<U> Select<U>(Func<T,U> selector);
       public C<V> SelectMany<U,V>(Func<T,C<U>> selector,
              Func<T,C<U>,V> resultSelector);
       public C<V> Join<U,K,V>(C<U> inner, Func<T,K> outerKeySelector,
              Func<U,K> innerKeySelector, Func<T,U,V> resultSelector);
       public C<V> GroupJoin<U,K,V>(C<U> inner,
              Func<T,K> outerKeySelector,
              Func<U,K> innerKeySelector,
              Func<T,C<U>,V> resultSelector);
       public O<T> OrderBy<K>(Func<T,K> keySelector);
       public O<T> OrderByDescending<K>(Func<T,K> keySelector);
       public C<G<K,T>> GroupBy<K>(Func<T,K> keySelector);
       public C<G<K,E>> GroupBy<K,E>(Func<T,K> keySelector,
              Func<T,E> elementSelector);
}
```

A step back

- Before we think about translating C$^\sharp$3.0
- ...Let's think about C$^\sharp$ 2.0 for now

A step back

- Before we think about translating C$^\sharp$3.0
- ...Let's think about C$^\sharp$ 2.0 for now
- ......but let's think about it formally!

Bidirectional type systems

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

Checking	$\Gamma \vdash e \downarrow \tau$	INPUTS:	Γ, e, τ
		OUTPUT:	Yes/No
Synthesis	$\Gamma \vdash e \uparrow \tau$	INPUTS:	Γ, e
		OUTPUT:	τ

- It's very algorithmic ☺

Typing relations in C$^\sharp$2.0

Checking	$\Gamma \vdash e \downarrow_{\mathsf{i}} \tau$	"e can be implicitly converted to τ"
	$\Gamma \vdash e \downarrow_{\mathsf{x}} \tau$	"e can be explicitly converted to τ"
	$\Gamma \vdash s \downarrow \phi$	"statement s can be converted to ϕ"
Synthesis	$\Gamma \vdash e \uparrow^{\mathsf{s}} \phi$	"e synthesizes a type ϕ"
Inference	$\Gamma; \overline{X} \vdash e \sim \tau \hookrightarrow \theta$	"e matches type τ with free type parameters $\overline{X}$ and infers the substitution θ"

Typing relations in C$^\sharp$2.0

Checking	$\Gamma \vdash e \downarrow_{\mathsf{i}} \tau$	"e can be implicitly converted to τ"
	$\Gamma \vdash e \downarrow_{\mathsf{x}} \tau$	"e can be explicitly converted to τ"
	$\Gamma \vdash s \downarrow \phi$	"statement s can be converted to ϕ"
Synthesis	$\Gamma \vdash e \uparrow^{\mathsf{s}} \phi$	"e synthesizes a type ϕ"
Inference	$\Gamma; \overline{X} \vdash e \sim \tau \hookrightarrow \theta$	"e matches type τ with free type parameters $\overline{X}$ and infers the substitution θ"

Fact

$\Gamma \vdash e \uparrow^{\mathsf{s}} \tau$ *iff* $\Gamma; X \vdash e \sim X \hookrightarrow [X \mapsto \tau]$

Type synthesis relation

$$\boxed{\Gamma \vdash e \uparrow^{\mathsf{s}} \tau}$$

$$\frac{}{\Gamma \vdash b \uparrow^{\mathsf{s}} \texttt{bool}} \quad \frac{}{\Gamma \vdash i \uparrow^{\mathsf{s}} \texttt{int}} \quad \frac{}{\Gamma, x\colon \tau \vdash x \uparrow^{\mathsf{s}} \tau}$$

$$\frac{\Gamma \vdash e \downarrow_{\mathsf{x}} \tau}{\Gamma \vdash (\tau)e \uparrow^{\mathsf{s}} \tau} \quad \frac{\Gamma \vdash e \uparrow^{\mathsf{s}} \tau_1 \quad \mathit{ftype}(\tau_1, f) = \tau_2}{\Gamma \vdash e.f \uparrow^{\mathsf{s}} \tau_2}$$

Type synthesis relation

$\boxed{\Gamma \vdash e \uparrow^{\mathsf{s}} \tau}$

$$\frac{}{\Gamma \vdash b \uparrow^{\mathsf{s}} \texttt{bool}} \qquad \frac{}{\Gamma \vdash i \uparrow^{\mathsf{s}} \texttt{int}} \qquad \frac{}{\Gamma, x\colon \tau \vdash x \uparrow^{\mathsf{s}} \tau}$$

$$\frac{\Gamma \vdash e \downarrow_{\mathsf{x}} \tau}{\Gamma \vdash (\tau)e \uparrow^{\mathsf{s}} \tau} \qquad \frac{\Gamma \vdash e \uparrow^{\mathsf{s}} \tau_1 \quad \mathit{ftype}(\tau_1, f) = \tau_2}{\Gamma \vdash e.f \uparrow^{\mathsf{s}} \tau_2}$$

Note

There are no synthesis rules for `null` or AMEs!

Implicit conversion relation

$\boxed{\Gamma \vdash e_1 \downarrow_{\mathsf{i}} \tau_1}$

$$\frac{}{\Gamma \vdash \texttt{null} \downarrow_{\mathsf{i}} \rho} \qquad \frac{dtype(D)(\overline{\tau}) = \overline{\tau_0} \rightarrow \phi \quad \Gamma, \overline{x}\colon \overline{\tau_0} \vdash \overline{s} \downarrow \phi}{\Gamma \vdash \texttt{delegate}\texttt{(}\overline{\tau_0}\ \overline{x}\texttt{)}\texttt{\{}\overline{s}\texttt{\}} \downarrow_{\mathsf{i}} D\texttt{<}\overline{\tau}\texttt{>}}$$

$$\frac{\Gamma \vdash e \uparrow^{\mathsf{s}} \tau_1 \quad \tau_1 <: \tau_2}{\Gamma \vdash e \downarrow_{\mathsf{i}} \tau_2}$$

$\boxed{\Gamma \vdash e_1 \downarrow_{\mathsf{x}} \tau_1}$

$$\frac{\Gamma \vdash e \downarrow_{\mathsf{i}} \tau}{\Gamma \vdash e \downarrow_{\mathsf{x}} \tau} \qquad \frac{\Gamma \vdash e \uparrow^{\mathsf{s}} \tau_1 \quad \tau_1 <:^{\mathsf{x}} \tau_2}{\Gamma \vdash e \downarrow_{\mathsf{x}} \tau_2}$$

And now on to the translation!

Typing relations (recall)

Checking	$\Gamma \vdash e \downarrow_{\mathsf{i}} \tau$	"e can be implicitly converted to τ"
	$\Gamma \vdash e \downarrow_{\mathsf{x}} \tau$	"e can be explicitly converted to τ"
	$\Gamma \vdash s \downarrow \phi$	"statement s can be converted to ϕ"
Synthesis	$\Gamma \vdash e \uparrow^{\mathsf{s}} \phi$	"e synthesizes a type ϕ"

Typing relations into translations

Checking	$\Gamma \vdash e \downarrow_{\mathsf{i}} \tau \rightsquigarrow e'$	"e can be implicitly converted to τ yielding e'"
	$\Gamma \vdash e \downarrow_{\mathsf{x}} \tau \rightsquigarrow e'$	"e can be explicitly converted to τ yielding e'"
	$\Gamma \vdash s \downarrow \phi \rightsquigarrow s'$	"statement s can be converted to ϕ yielding s'"
Synthesis	$\Gamma \vdash e \uparrow^{\mathsf{s}} \phi \rightsquigarrow e'$	"e synthesizes a type ϕ yielding e'"

Key fact

It's all about type-directed translation!

Simple example

$$\frac{dtype(D)(\overline{\tau}) = \overline{\tau_0} \rightarrow \tau_1 \quad \Gamma, \overline{x_0} : \overline{\tau_0} \vdash e_1 \downarrow_{\mathsf{i}} \tau_1 \rightsquigarrow e_{11}}{\Gamma \vdash \texttt{(}\overline{x_0}\texttt{) => } e_1 \downarrow_{\mathsf{i}} D\texttt{<}\overline{\tau}\texttt{>} \rightsquigarrow \texttt{delegate(}\overline{\tau_0}\ \overline{x}\texttt{)\{ return } e_{11}\texttt{; \}}}$$

Simple example

$$\frac{dtype(D)(\overline{\tau}) = \overline{\tau_0} \rightarrow \tau_1 \quad \Gamma, \overline{x_0} : \overline{\tau_0} \vdash e_1 \downarrow_{\mathsf{i}} \tau_1 \rightsquigarrow e_{11}}{\Gamma \vdash \texttt{(}\overline{x_0}\texttt{) => } e_1 \downarrow_{\mathsf{i}} D\texttt{<}\overline{\tau}\texttt{>} \rightsquigarrow \texttt{delegate(}\overline{\tau_0}\ \overline{x}\texttt{)\{ return } e_{11}\texttt{; \}}}$$

Content

Lambda expressions are translated to AMEs

Simple example

$$\frac{\Gamma \vdash e_1 \uparrow^{\mathsf{S}} \tau_2\texttt{[ ]} \leadsto e_{11} \quad x \notin dom(\Gamma) \quad \Gamma, x\colon \tau_2 \vdash s_1 \downarrow \phi \leadsto s_{11}}{\Gamma \vdash \texttt{foreach (var}\ x_1\ \texttt{in}\ e_1\texttt{)}\ s_1 \downarrow \phi \leadsto \texttt{foreach (}\tau_2\ x_1\ \texttt{in}\ e_{11}\texttt{)}\ s_{11}}$$

Synthesis/Conversion difference exposed

$$\frac{\Gamma \vdash e_1 {\uparrow}^{\mathsf{s}} \tau_1 \rightsquigarrow e_{11} \quad x \notin dom(\Gamma) \quad \Gamma, x\colon \tau_1 \vdash \overline{s_1} \downarrow \phi \rightsquigarrow \overline{s_{11}}}{\Gamma \vdash \texttt{var}\ x = e_1\,;\ \overline{s_1} \downarrow \phi \rightsquigarrow \tau_1\ x = e_{11}\,;\ \overline{s_{11}}}$$

$$\frac{\Gamma \vdash e_1 {\downarrow}_{\mathsf{i}} \tau_1 \rightsquigarrow e_{11} \quad x \notin dom(\Gamma) \quad \Gamma, x\colon \tau_1 \vdash \overline{s_1} \downarrow \phi \rightsquigarrow \overline{s_{11}}}{\Gamma \vdash \tau_1\ x = e_1\,;\ \overline{s_1} \downarrow \phi \rightsquigarrow \tau_1\ x = e_{11}\,;\ \overline{s_{11}}}$$

Synthesis/Conversion difference exposed

$$\frac{\Gamma \vdash e_1 \uparrow^{\mathsf{s}} \tau_1 \rightsquigarrow e_{11} \quad x \notin dom(\Gamma) \quad \Gamma, x\colon \tau_1 \vdash \overline{s_1} \downarrow \phi \rightsquigarrow \overline{s_{11}}}{\Gamma \vdash \texttt{var}\ x = e_1\,;\ \overline{s_1} \downarrow \phi \rightsquigarrow \tau_1\ x = e_{11}\,;\ \overline{s_{11}}}$$

$$\frac{\Gamma \vdash e_1 \downarrow_{\mathsf{i}} \tau_1 \rightsquigarrow e_{11} \quad x \notin dom(\Gamma) \quad \Gamma, x\colon \tau_1 \vdash \overline{s_1} \downarrow \phi \rightsquigarrow \overline{s_{11}}}{\Gamma \vdash \tau_1\ x = e_1\,;\ \overline{s_1} \downarrow \phi \rightsquigarrow \tau_1\ x = e_{11}\,;\ \overline{s_{11}}}$$

All these fail!

```
var a = null;
var b =
  delegate(int x){return x;};
var c  = (x)=> x;
```

Synthesis/Conversion difference exposed

$$\frac{\Gamma \vdash e_1 \uparrow^{\mathsf{s}} \tau_1 \rightsquigarrow e_{11} \quad x \notin dom(\Gamma) \quad \Gamma, x\colon \tau_1 \vdash \overline{s_1} \downarrow \phi \rightsquigarrow \overline{s_{11}}}{\Gamma \vdash \mathtt{var}\ x = e_1\,;\ \overline{s_1} \downarrow \phi \rightsquigarrow \tau_1\ x = e_{11}\,;\ \overline{s_{11}}}$$

$$\frac{\Gamma \vdash e_1 \downarrow_{\mathsf{i}} \tau_1 \rightsquigarrow e_{11} \quad x \notin dom(\Gamma) \quad \Gamma, x\colon \tau_1 \vdash \overline{s_1} \downarrow \phi \rightsquigarrow \overline{s_{11}}}{\Gamma \vdash \tau_1\ x = e_1\,;\ \overline{s_1} \downarrow \phi \rightsquigarrow \tau_1\ x = e_{11}\,;\ \overline{s_{11}}}$$

All these fail!

```
var a = null;
var b =
  delegate(int x){return x;};
var c  = (x)=> x;
```

All these work!

```
Button a = null;
Func<int,int> b =
  delegate(int x){return x;};
Func<Button,object> c = (x)=> x;
```

An awkward wart

There's a nasty with type inference and overloading resolution:

```
static void mybar<T>(int a, T b){ Console.WriteLine("1"); }
static void mybar<T>(int a, int b){ Console.WriteLine("2"); }
```

An awkward wart

There's a nasty with type inference and overloading resolution:

```
static void mybar<T>(int a, T b){ Console.WriteLine("1"); }
static void mybar<T>(int a, int b){ Console.WriteLine("2"); }
```

```
mybar(42,42);
```

An awkward wart

There's a nasty with type inference and overloading resolution:

```
static void mybar<T>(int a, T b){ Console.WriteLine("1"); }
static void mybar<T>(int a, int b){ Console.WriteLine("2"); }
```

```
mybar(42,42);          // Infers <int>; Prints 1
```

An awkward wart

There's a nasty with type inference and overloading resolution:

```
static void mybar<T>(int a, T b){ Console.WriteLine("1"); }
static void mybar<T>(int a, int b){ Console.WriteLine("2"); }
```

```
mybar(42,42);          // Infers <int>; Prints 1
```

```
mybar<int>(42,42);
```

An awkward wart

There's a nasty with type inference and overloading resolution:

```
static void mybar<T>(int a, T b){ Console.WriteLine("1"); }
static void mybar<T>(int a, int b){ Console.WriteLine("2"); }
```

```
mybar(42,42);          // Infers <int>; Prints 1
```

```
mybar<int>(42,42);     // Prints 2 :-(
```

Consequence of the wart

Normally we'd formalize local type inference as, e.g.

$$\Gamma \vdash e\,.\,m(\bar{f}) : \sigma \rightsquigarrow e\,.\,m\texttt{<}\bar{\tau}\texttt{>}(\bar{f})$$

But this doesn't work ☹

Several solutions possible; we use method descriptors from MSIL

Conclusions

- The LINQ project enables query-like facilities into the .NET languages
- C$^\sharp$3.0 (and VB 9.0) contains a number of FP-inspired extensions to enhance LINQ programming
- These new language features can be explained by type-directed translations (this paper)
- Formal techniques are useful specification tools ☺
  - Machine-assisted tools would be great here!

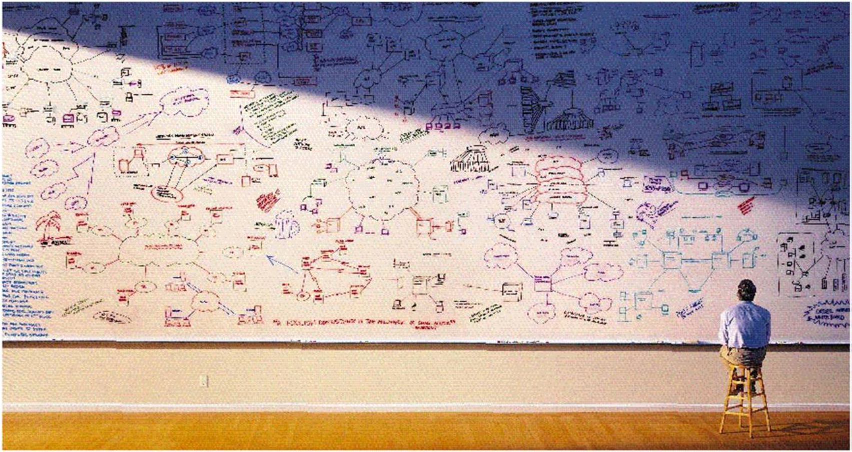
Related work

Far too much to mention!

- Close relative: Cω [Bierman, Meijer, Schulte]
- Lots and lots of other work (much academic):

  SQLJ, XJ, TL, HaskellDB, Fibonacci, Galileo, XDuce, CDuce, Napier88, OPJ, PJama, Thor, ...

Questions?



Or come to the Microsoft booth!