The essence of data access in Cω

Gavin Bierman, Erik Meijer, Wolfram Schulte

19th ECOOP, Glasgow, Scotland

28th July 2005

Summary of talk

1. Background

   What's the problem? Why another programming language? What's new?

2. Introduction to Cω

   Informal introduction

3. Formalization

   (Very briefly) Type system; operational semantics; breaking news

4. Related work
5. Conclusions

What’s the problem?

- Many typical web-based applications need to
  - create/read relational data
  - create queries to be executed on a database server
  - create/read semistructured data (XML/HTML)
  - deal with inherent (asynchronous) concurrency
- Leads to three-tier software architecture

Can't I do this in Java/C# already?

- Relational Data/Queries:
  - Use APIs to create/access relational data (relational data as objects)
  - Use APIs to create queries (queries as strings)
- Semistructured data:
  - As above:
    - SS data as objects
    - Queries as strings
- Upper and lower tiers are very fragile ☹
- (Asynchronous) concurrency:
  - As many APIs as there are concepts!

Guiding principle

Microsoft Research Cambridge

Put features in the language itself, rather than in libraries

- Provide better level of abstraction
- Make invariants and intentions more apparent (part of the interface)
- Give stronger compile-time guarantees (types)
- Enable different implementations and optimizations
- Expose structure for other tools to exploit (example: static analysis)

Part I: A Cω primer

Design brief

- Take C# as a starting point (warts and all)
- No changes to the semantics of C#
- No changes to the CLR (1.1)
- Minimal extensions:
  - Don’t simply tack on the entire relational and semi-structured data models
  - Rather offer similar expressive power, but more in the spirit of an object-oriented language
  - XML literals and (some fragment of) SQL

Cω (partial) castlist

- Data access extensions:
  - Originally the brainchild of Erik and Wolfram


- Concurrency:
  - Due to Nick Benton, Luca Cardelli and Cédric Fournet
  - aka “Polyphonics”
- Compiler gurus:
  - Herman Venter, Claudio Russo and others

What's been added to C#?

- Data access:
  - New types
    - Streams
    - Anonymous structs
    - Choice types
    - Content classes
  - Generalized member access
  - XML literals
  - SQL expressions
- Concurrency:
  - New type
    - **`async`**
  - Generalized notion of method header to define chords

Described at ECOOP 2002!

New type #1: Streams

- New type: **`T*`** (sequence of **`T`**)
- Close relative of **`IEnumerable<T>`** (C# 2.0)
- Streams generated by statement blocks that **`yield`** values
- Streams consumed by **`foreach`** loop
- No nested streams (no **`T**`**)
  - Just like in XQuery/XPath

Example stream processing

Microsoft Research Cambridge

```
public static int* FromTo(int s, int e)    {
      for (i = s; i <= e; i++) yield return i;
}

int* OneToTen = FromTo(1,10);

foreach(int j in OneToTen) {
      Console.WriteLine(j);
}
```

Note the co-routine like behaviour

New type #2: Anonymous structs

- New type:

```
struct{int i; string s; string s;}
```

- Constructed in obvious way:

```
struct{int i; string s; string s;} tup =
          new{i=42, s="Erik", s="Wolfram"};
```

- (Like ML records but ordered, duplicates allowed)

New type #3: Choice

- New type: `choice{T;S;}`
- A value of this type is either a value of type `T` or a value of type `S`, example:

```
choice{string; Button; int;} u = “hello”;
u = new Button();
u = 42;
```

New type #4: Content class

- Like a normal class, except it contains a single unnamed member
- Useful for XML fidelity

```
public class book {
  struct{
    string title;
    choice{struct{editor editor;}*;
           struct{author author;}*;
    }
    string publisher;
    decimal price;
  }
}
```

XML Literals

- Now we have the types, we can write (strongly typed) XML literals in our code:

```
<!ELEMENT bib  (book*)>
<!ELEMENT book (title,
         (author* | editor*),
publisher, price)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT publisher (#PCDATA)>
<!ELEMENT price  (#PCDATA)>
```

```
public class bib
  { struct{ book book; }*; }
public class book
  { struct {string title;
            choice {
             struct { editor editor;}*;
             struct { author author;}*;}
            string publisher;
            decimal price;}
  }
```

```
bib = <bib>
       <book year="1994">
        <title>TCP/IP Illustrated</title>
        <author><last>Stevens</last>
               <first>W.</first>
        </author>
        <publisher>Addison-Wesley</publisher>
        <price> 65.95</price>
       </book>
       <book>
        <title>The Economics of Technology and
               Content for Digital TV</title>
        <editor><last>Gerbarg</last>
                <first>Darcy</first>
                <affiliation>CITI</affiliation>
        </editor>
        <publisher>Kluwer Academic</publisher>
        <price>129.95</price>
       </book>
      </bib>;
```

Generalized member access

Microsoft Research Cambridge

- A key programming feature of Cω
- Generalize member access (the ‘dot’ operator) from just objects to all the new datatypes
- Aim: Make the dot behave like the ‘/’ in XPath

The power is in the dot!

Generalized member access

- Sequence:

```
string* ss;
ss.ToUpper();
```

string*

- Anonymous struct:

```
struct{int i; string s; string s;} x;
x.i;
x.s;
```

int

string*

- Choice:

```
choice{string;Person;int;} y;
y.Length;
```

choice{int;Float;?

XQuery Use Case Three

Microsoft Research Cambridge

For each book in the bibliography, list the title and authors, grouped inside a “result” element

XQUERY

```
for $b in $bs/book
return
 <result>
   {$b/title}
   {$b/author}
 <result>
```

Cω

```
foreach (b in bs.book)
{
  yield return <result>
                {b.title}
                {b.author}
               </result>;
}
```

First class SQL

- Elements of SQL have been built in to the language

```
String conn = "  ...Initial Catalog=Northwind;  ";
Database DB = new Database(conn);
Console.WriteLine("Please enter a city: ");
string input = Console.ReadLine();

res = select *
        from DB.Customers
       where City==input ;

Console.WriteLine("Customers from: "+input);
res.{Console.WriteLine(it.ContactName +
                       " --- " +
                       it.Phone);};
```

- In fact, we can use this construct over both in-memory and external data!

Part II: Formalization

Strategy

- Adopt a Featherweight Approach™
  - Identify a core fragment (subset) of the language
  - Give type system
  - Give operational semantics (preferably single-step)
  - Prove type soundness results
- Actually things are a little more complicated here:
  - Identify a core language, FCw
    - Define type system
  - Identify an inner language, ICw
    - Define type system
    - Type-directed compilation: FCw → ICw
    - Define operational semantics
    - Prove type soundness

Microsoft Research Cambridge

Expression

e	$::=$	$b \mid i$	Literals
	$\mid$	$e \oplus e$	Built-in operators
	$\mid$	x	Variable
	$\mid$	`null`	Null
	$\mid$	$(\tau)e$	Cast
	$\mid$	e `is` τ	Dynamic typecheck
	$\mid$	e `was` κ	Static typecheck for choice values
	$\mid$	`new` $\tau(e)$	Object creation
	$\mid$	`new` $\{\overline{be}\}$	Anonymous struct creation
	$\mid$	$e.f$	Field access
	$\mid$	$e[i]$	Field access by position
	$\mid$	pe	Promotable expression

Promotable expression

pe	$::=$	$x = e$	Variable assignment
	$\mid$	$e.m(\overline{e})$	Method invocation
	$\mid$	$e.\{e\}$	Apply-to-all

Binding expression

be	$::=$	$f = e$	Named binding
	$\mid$	e	Unnamed binding

Statement

s	$::=$	`;`	Skip
	$\mid$	pe`;`	Promoted expression
	$\mid$	`if` (e) s `else` s	Conditional
	$\mid$	$\tau\ x = e$`;`	Variable declaration
	$\mid$	`return` e`;`	Return statement
	$\mid$	`return;`	Empty return
	$\mid$	`yield return` e`;`	Yield statement
	$\mid$	`yield break;`	End of stream
	$\mid$	`foreach` $(\sigma\ x$ `in` $e)$ s	Foreach loop
	$\mid$	`while` (e) s	While loop
	$\mid$	$\{\overline{s}\}$	Block

See…it all fits on a single slide ☺

Operational semantics

- Transition relation of form:

$$(H, R), e \longrightarrow (H', R'), e'$$

- For expressions, pretty standard
- For statements, a little more tricky…

Semantics of foreach

- [Moving to C# 2.0] Tricky! For example:

```
foreach (T x in e) s
```

- is actually expanded to:

```
IEnumerator<T> enumerator =
((IEnumerable<T>)(e)).GetEnumerator();
try {
      while (enumerator.MoveNext()) {
            T x = (T)enumerator.Current;
            s;
      }
}
finally {
      enumerator.Dispose();
}
```

Iterator blocks

```
static IEnumerable<int> FromTo(int from, int to) {
	while (from <= to) yield return from++;
}
```

- gets rewritten to:

```
static IEnumerable<int> FromTo(int from, int to) {
	return new __Enumerable1(from, to);
}
```

- with **`GetEnumerator`** method:

```
public IEnumerator<int> GetEnumerator() {
	__Enumerable1 result = this;
	if (Interlocked.CompareExchange(ref __state, 1, 0) != 0)
	{	result = new __Enumerable1(__from, to);
		result.__state = 1;
	}
	result.from = result.__from;
	return result;
}
```

GetEnumerator

- From C# 2.0 documentation:

"The **first** time the **`GetEnumerator`** method is invoked, the enumerable **object itself** is returned. **Subsequent** invocations of the enumerable object's **`GetEnumerator`**, if any, return a **copy** of the enumerable object. Thus, each returned enumerator has its own state and changes in one enumerator will not affect another. The **`Interlocked.CompareExchange`** method is used to ensure thread-safe operation."

Our rules

- We captured all this complication **and** flattening of nested streams in just 8 transition rules!

Could we improve?

[Post ECOOP submission]

- Three criticisms of these semantics:
  1. Complicated!
  2. Trying to model too much? (In-place updating of stream closure)
  3. Don't see that nice co-routine behaviour

Gavin's latest cool rules

1. Move to a frame-stack semantics, à la Felleisen, GMB/Parkinson/Pitts (Middleweight Java), and others

(H, VS, code, FS)

FS = Frame stack
= Composition of frames
= current evaluation context
= future continuation!

2. Drop the in-place updating
   - Do everything by copying (it's just a spec. after all!)

Yield a value

[E-YieldV] $\langle H, MS \circ (MS' \circ VS), \texttt{yield return}(\sigma, v);, FS \circ \mathsf{Foreach}(\sigma', x, r, s) \circ FS' \rangle$
$\rightsquigarrow \langle H', MS'' \circ VS, s, \mathsf{Foreach}(\sigma', x, r', s) \circ FS' \rangle$

$$\begin{aligned}
\text{where } & \mathsf{Foreach} \notin FS \\
& H(r) = (\tau, -) \\
& r' \notin \mathrm{dom}(H) \\
& H' \stackrel{\text{def}}{=} H[r' \mapsto (\tau, [MS, FS])] \\
& MS'' \stackrel{\text{def}}{=} MS'[x \mapsto v]
\end{aligned}$$

Stream evaluation rules (in total)

[E-Foreach] $\langle H,\ VS, \texttt{foreach}\ (\sigma\ x\ \texttt{in}\ r)\ s',\ FS\rangle$
$\rightsquigarrow \langle H, MS \circ VS, \overline{s}, \mathsf{Foreach}(\sigma, x, r, s') \circ FS\rangle$ where $H(r) = (-, [MS, \overline{s}])$

[E-ForeachNull] $\langle H,\ VS, \texttt{foreach}\ (\sigma\ x\ \texttt{in null})\ s',\ FS\rangle$
$\rightsquigarrow \langle H,\ VS,\ ;,\ FS\rangle$

[E-YieldV] $\langle H, MS \circ (MS' \circ VS), \texttt{yield return}(\sigma, v);, FS \circ \mathsf{Foreach}(\sigma', x, r, s) \circ FS'\rangle$
$\rightsquigarrow \langle H', MS'' \circ VS, s, \mathsf{Foreach}(\sigma', x, r', s) \circ FS'\rangle$ where $\mathsf{Foreach} \notin FS$
$H(r) = (\tau, -)$
$r' \notin \mathrm{dom}(H)$
$H' \stackrel{\mathrm{def}}{=} H[r' \mapsto (\tau, [MS, FS])]$
$MS'' \stackrel{\mathrm{def}}{=} MS'[x \mapsto v]$

[E-YieldBr] $\langle H, MS \circ VS, \texttt{yield break};, FS \circ \mathsf{Foreach}(\sigma, x, r, s) \circ FS'\rangle$
$\rightsquigarrow \langle H,\ VS,\ ;,\ FS'\rangle$ where $\mathsf{Foreach} \notin FS$

[E-YieldNest] $\langle H, MS \circ VS, \texttt{yield return}(\sigma\texttt{*},v);, FS \circ \mathsf{Foreach}(\sigma', x, r, s) \circ FS'\rangle$
$\rightsquigarrow \langle H',\ VS, \texttt{foreach}(\sigma'\ \texttt{x in}\ v)\ s, \mathsf{Foreach}(\sigma', x, r', s) \circ FS'\rangle$ where $\mathsf{Foreach} \notin FS$
$H(r) = (\tau, -)$
$r' \notin \mathrm{dom}(H)$
$H' \stackrel{\mathrm{def}}{=} H[r' \mapsto (\tau, [MS, FS])]$

Type soundness

- We can prove that our operational semantics satisfy a type safety property

- Proved in “standard way”:
  - Subject reduction theorem
  - Progress theorem

Related work

- Building database programming languages:
  - Too much work to mention!
  - Lots of academic work, e.g. Galileo, Fibonacci, Tycoon, HaskellDB, …
  - Industrial work: SQLJ, …
- Integrating XML with programming languages:
  - CDuce (ENS/INRIA), Links (Edinburgh)
  - Xtatic (UPenn)
  - XJ (IBM Research)

Lots more work to do here!

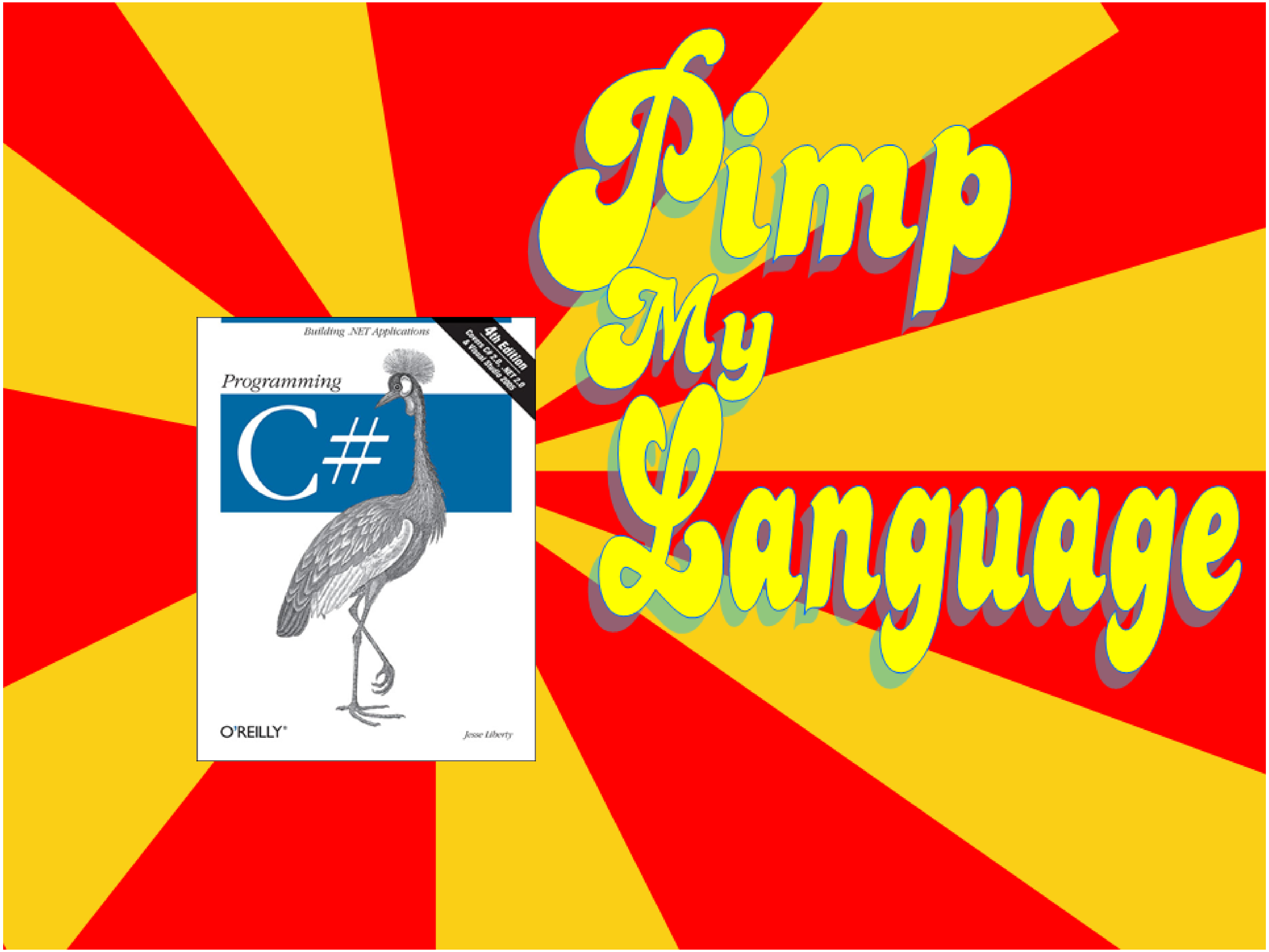
Summary

- We took (the entire) C# as our starting point
- We "grew" it with minimal extensions to allow
  - Relational/Semi-structured data manipulation/creation
  - Query-like constructs
  - (asynchronous concurrency primitives)
- We've proved it correct!
- Our hope: Cω is a language well-suited for data-intensive, distributed applications

- Compiler is freely available from:

  http://research.microsoft.com/comega

Another perspective…



Thank you! Any questions?

Microsoft Research Cambridge