From C# to C∞

Luca Cardelli

Microsoft Research

Rotor Ws, Pisa, 2003-04-23

Enabler

- Rotor (and the CLR), re-enable language experimentation.
  - A pre-approved multi-language environment.
  - With a flexible attitude towards further innovation.
  - Open to the wider community.
- What should we do with this freedom?
  - Add Generics/Polymorphism (Done!)
    - Generic CLR/C#, Gyro
  - Integrate Functional Languages (Done!)
    - SML.NET, F#, ILX
  - What else?

New Programming Models

- We are in the middle of a transition in programming models (and possibly PLs)
  - More radical than C to C++ (objects) or C++ to Java (strong typing).
- What's on the other side of the transition?
  - Not clear, but lots of small and big steps are being taken into uncharted territory.
- We have a Cambrian explosion of programming models.
  - Lots of badly misshaped things are going to evolve before architectures settle down.

A Transition Caused by:

- A new emphasis on computation on WANs
  - Wide area data integration
    - XML is “net data”.
    - Need to integrate this new data into PLs.
    - **BIG DEBATE: hide, blend-in, or redesign?**
  - Wide area flow integration
    - Messages nor RPC, schedules not threads.
    - Need to integrate these new flows into PLs.
    - **BIG DEBATE: hide, blend-in, or redesign?**
- Impact
  - Not easy to fit this new stuff into existing PLs.
  - Ideal for Rotor research…

Data Integration

- PL data has traditionally been "triangular" (trees), while persistent data has traditionally been "square" (tables).
  - This has caused huge integration problems.
- Now, *BIG NEWS*, persistent data is triangular too! (XML)
  - New opportunity for PL integration.
  - However, the type systems for PL data (based on tree matching) and XML (based on tree automata) are still deeply incompatible.
  - Wouldn't it be nice to "program directly against the schemas" in a well-typed way?

Flow Integration

- Wouldn't it be nice to hide concurrency from programmers?
  - SQL does it well.
  - UI packages do it fine.
  - RPC does it ok.
  - But we are moving towards more asynchrony, I.e. towards more visible concurrency (BizTalks, etc.)
  - You can hide all concurrency some of the time, and you can hide some concurrency all the time, but you can't hide all concurrency all the time.

Retrofitting

- It would be wrong to see these new net data and net flow features as "just" new Domain Specific Languages (DSLs).
- This are fundamental new concepts that may result in new kinds of "general purpose programming languages" for WANs.
  - Analogy with past transitions: C++ was not just a DSL for objects, and Java was not just a DSL for type safety.
  - We need more than just an XML DSL and a flow DSL.
- Keep an open mind: there may be great opportunities around the corner, either as evolutions of C#, or new languages.

Reliability

- Whether or not we merge new programming models into PLs, we need analysis tools for these new situations.
  - Data: e.g.: data consistency checks.
  - Flow: e.g.: behavioral type system.
- The new programming models are sophisticated (i.e. “evil”), and require equally sophisticated (i.e. “good”) analysis tools to protect developers.
- **BIG Debate/Trade-off: Checked annotations, vs. inferred information.**

What Else, Indeed

- **Semistructured Data**
  - XDuce
  - TQL
  - Spatial Data Types
  - *<Unnamable>* (Erik Meijer and Wolfram Shulte)
- **Concurrent Flows**
  - BizTalk
  - BPEL
  - Polyphonic C#
  - Sharpie
  - *<Unnamable>* (Greg Meredith)

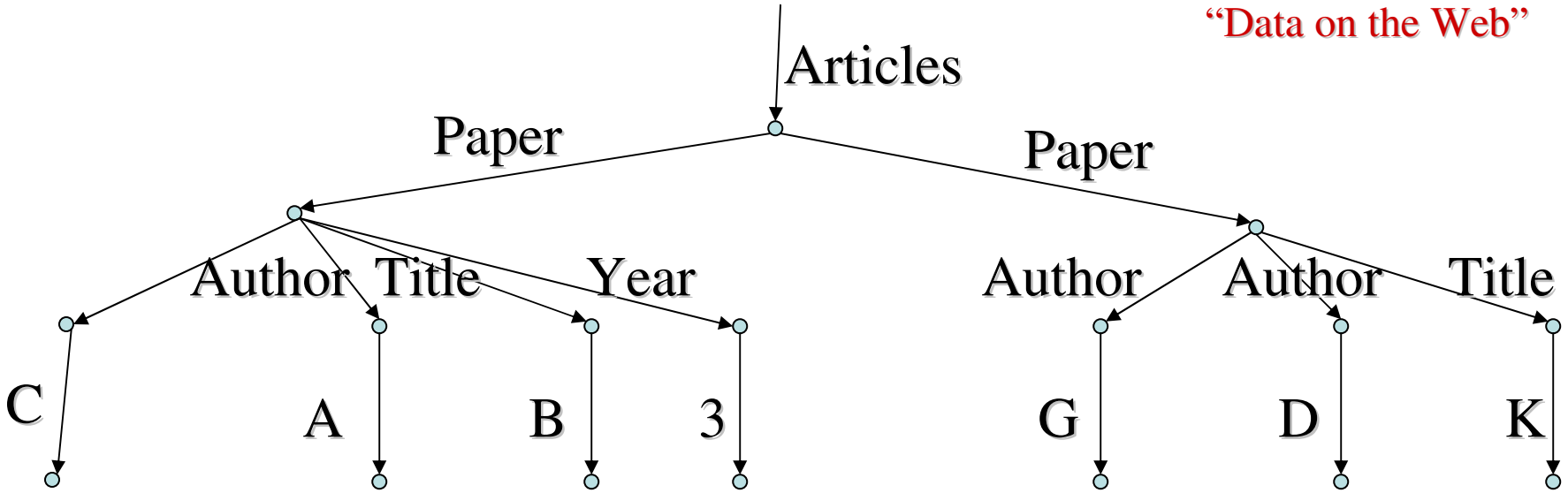
Pre-Conclusions

- New opportunities for Rotor research…
- Both data and flow extensions are famously incompatible with standard languages:
  - Schemas vs. (object) types
    "impedence mismatch"
  - Concurrency vs. inheritance
    "inheritance anomaly"

REVIEW: Semistructured Data

(I.e.: XML after parsing)

Abiteboul, Buneman, Suciu:
"Data on the Web"



- A tree (or graph), unordered (or ordered). With labels on the edges.
- Invented for "flexible" data representation, for quasi-regular data like address books and bibliographies.
- Adopted by the DB community as a solution to the "database merge" problem: merging databases from uncoordinated (web) sources.
- Adopted by W3C as "web data", then by everybody else.

It's Unusual Data

- Not really arrays/lists:
  - Many children with the same label, instead of indexed children.
  - Mixture of repeated and non repeated labels under a node.
- Not really records:
  - Many children with the same label.
  - Missing/additional fields with no tagging information.
- Not really variants (tagged unions):
  - Labeled but untagged unions.
- Unusual data.
  - Yet, it aims to be the new universal standard for interoperability of programming languages, databases, e-commerce...

Needs Unusual Languages

- New *flexible* types and schemas are required.
  - Based on "regular expressions over trees" reviving techniques from tree-automata theory.
- New processing languages required.
  - Xduce [Pierce, Hosoya], Cduce, our own…
  - Various web scripting abominations.
- New query languages required. Various approaches:
  - From simple: Existence of paths through the tree.
  - To fuzzy: Is a tree "kind of similar" to another one?
  - To fancy: Is a tree produced by a tree grammar?
  - To popular: SQL for trees/graphs, for some value of "SQL".

Data Descriptions

- We want to *talk about* (specify/ query/ constrain/ type) the possible structure of data, for many possible reasons:
  - Typing (and typechecking): for language and database use.
  - Constraining (and checking): for policy or integrity use.
  - Querying (and searching): for semistructured database use.
  - Specifying (and verifying): for architecture or design documents.
- A *description* is a formal way of talking about the possible structure of data.
  - We should go after a general framework: a very expressive language of descriptions.
  - Special classes of descriptions can be used as types, schemas, constraints, queries, and specifications.

Example: Typing

Data

```
Cambridge[
  Eagle[
    chair[0] |
    chair[0]
  ]
]
```

In Cambridge there is (nothing but) a pub called the Eagle that contains (nothing but) two empty chairs.

Description

```
Cambridge[
  Eagle[
    chair[0] |
    T
  ] | T
]
```

data matches description

In Cambridge there is (at least) a pub called the Eagle that contains (at least) one empty chair.

Example: Queries

With match variables $\mathcal{X}$: *Who is really sitting at the Eagle?*

matches?

Eagle[
  chair[¬**0** ∧ $\mathcal{X}$] |
  T
]

Yes: $\mathcal{X}$ = *John*[**0**]
Yes: $\mathcal{X}$ = *Mary*[**0**]

With *select-from*:

from Eagle[...]
match Eagle[*chair*[¬**0** ∧ $\mathcal{X}$] | **T**]
select person[$\mathcal{X}$]

Single result:
person[*John*[**0**]] |
person[*Mary*[**0**]]

Current Work

- We investigate
  - Powerful languages of data description, based on “spatial logics”.
  - Akin to “description logics” of some time ago, but seen more as type systems.
  - Special cases are regular expressions over trees (XML query, etc.)
  - Lots of open problems in this area (typing and subtyping algorithms).

REVIEW: Concurrent Flows

- Distribution => concurrency + latency
  => asynchrony
  => more concurrency
- Message-passing, event-based programming, dataflow models
- For programming languages, coordination (orchestration) languages & frameworks, workflow

Language support for concurrency

- Why?
  - Make invariants and intentions more apparent (part of the interface)
  - Good software engineering
  - Allows the compiler much more freedom to choose different implementations
  - Also helps other tools

.NET today

- Java-style “monitors”
- OS shared memory primitives
- Clunky delegate-based asynchronous calling model
- Hard to understand, use and get right
  - Different models at different scales
  - Support for asynchrony all on the caller side – little help building code to *handle* messages (must be thread-safe, reactive, and deadlock-free)

Polyphonic C#

- An extension of the C# language with new concurrency constructs
- Based on the **join calculus**
  - A foundational process calculus like the π-calculus but better suited to asynchronous, distributed systems
  - It adapts remarkably well to classes and methods.
- A single model which works both for
  - local concurrency (multiple threads on a single machine)
  - distributed concurrency (asynchronous messaging over LAN or WAN)
- It is different
- But it's also simple – if Mort can do any kind of concurrency, he can do this

In one slide:

- Objects have both **synchronous** and ***asynchronous*** methods.
- Values are passed by ordinary method calls:
  - If the method is synchronous, the caller blocks until the method returns some result (as usual).
  - If the method is **async**, the call completes at once and returns void.
- A class defines a collection of ***chords*** (synchronization patterns), which define what happens once a particular *set* of methods have been invoked. One method may appear in several chords.
  - When pending method calls match a pattern, its body runs.
  - If there is no match, the invocations are queued up.
  - If there are several matches, an unspecified pattern is selected.
  - If a pattern containing *only* async methods fires, the body runs in a new thread.

A simple buffer

```
class Buffer {
  String get() & async put(String s) {
     return s;
  }
}
```

A simple buffer

```
class Buffer {
  String get() & async put(String s) {
     return s;
  }
}
```

- An ordinary (synchronous) method with no arguments, returning a string

A simple buffer

```
class Buffer {
  String get() & async put(String s) {
     return s;
  }
}
```

- An ordinary (synchronous) method with no arguments, returning a string
- An asynchronous method (hence returning no result), with a string argument

A simple buffer

```
class Buffer {
  String get() & async put(String s) {
      return s;
  }
}
```

- An ordinary (synchronous) method with no arguments, returning a string
- An asynchronous method (hence returning no result), with a string argument
- Joined together in a chord

A simple buffer

```
class Buffer {
  String get() & async put(String s) {
     return s;
  }
}
```

- Calls to `put()` return immediately (but are internally queued if there's no waiting `get()`).
- Calls to `get()` block until/unless there's a matching `put()`
- When there's a match the body runs, returning the argument of the `put()` to the caller of `get()`.
- Exactly which pairs of calls are matched up is unspecified.

A simple buffer

```
class Buffer {
  String get() & async put(String s) {
      return s;
  }
}
```

- Does example this involve spawning any threads?
  - No. Though the calls will usually come from different pre-existing threads.
- So is it thread-safe? You don't seem to have locked anything…
  - Yes. The chord compiles into code which uses locks. (And that *doesn't* mean everything is synchronized on the object.)
- Which method gets the returned result?
  - The synchronous one. And there can be at most one of those in a chord.

Reader/Writer

…using threads and mutexes in Modula 3

An introduction to programming with threads.
Andrew D. Birrell, January 1989.

```
VAR i: INTEGER;
VAR m: Thread.Mutex;
VAR c: Thread.Condition;

PROCEDURE AcquireExclusive();
BEGIN
    LOCK m DO
        WHILE i # 0 DO Thread.Wait(m,c) END;
        i := -1;
    END;
END AcquireExclusive;

PROCEDURE AcquireShared();
BEGIN
    LOCK m DO
        WHILE i < 0 DO Thread.Wait(m,c) END;
        i := i+1;
    END;
END AcquireShared;
```

```
PROCEDURE ReleaseExclusive();
BEGIN
    LOCK m DO
        i := 0; Thread.Broadcast(c);
    END;
END ReleaseExclusive;

PROCEDURE ReleaseShared();
BEGIN
    LOCK m DO
        i := i-1;
        IF i = 0 THEN Thread.Signal(c) END;
    END;
END ReleaseShared;
```

Reader/Writer in five chords

```
public class ReaderWriter {
  public void Exclusive() & async Idle() {}
  public void ReleaseExclusive() { Idle(); }

  public void Shared() & async Idle()    { S(1); }
  public void Shared() & async S(int n) { S(n+1); }
  public void ReleaseShared() & async S(int n) {
    if (n == 1) Idle(); else S(n-1);
  }

  public ReaderWriter() { Idle(); }
}
```

A single private message represents the state:

none ←→ `Idle()` ←→ `S(1)` ←→ `S(2)` ←→ `S(3)` …

(exclusive) (available) (shared)

A pretty transparent description of a simple state machine, as it should be.

Features

- A clean, simple, new model for asynchronous concurrency in C#
  - Declarative, local synchronization
  - Model good for both local and distributed settings
  - Efficiently compiled to queues and automata
  - Easier to express and enforce concurrency invariants
  - Compatible with existing constructs, though they constrain our design somewhat
  - Minimalist design – pieces to build whatever complex synchronization behaviours you need
  - Solid foundations
  - Works well in practice

Implementation

- Translate Polyphonic C# -> C#
- Built on Proebsting & Hanson's lcsc
- Introduce queues for pending calls (holding blocked threads for sync methods, arguments for asyncs)
- Generated code (using brief lock to protect queue state) looks for matches and then either
  - Enqueues args (async no match)
  - Enqueues thread and blocks (sync no match)
  - Dequeues other args and continues (sync match)
  - Wakes up blocked thread (async match with sync)
  - Spawns new thread (async match all async)
- Efficient – bitmasks to look for matches, no PulseAlls,…

Conclusions

- New languages
  - Language evolution is driven by elegance (mental efficiency).
  - Language adoption is driven by need.
- We now *badly need* evolution in data handling and control flows.
  - Lots of inelegant need-driven hacks.
  - Some elegant designs here and there.
  - Let's put them together!
  - To C∞ … and beyond!