The essence of data access in Cω

The power is in the dot!

Gavin Bierman[1], Erik Meijer[2], and Wolfram Schulte[3]

[1] Microsoft Research, UK. gmb@microsoft.com
[2] Microsoft Corporation, USA. emeijer@microsoft.com
[3] Microsoft Research, USA. schulte@microsoft.com

Abstract. In this paper we describe the data access features of Cω, an experimental programming language based on C$^\sharp$ currently under development at Microsoft Research. Cω targets distributed, data-intensive applications and accordingly extends C$^\sharp$'s support of both data and control. In the data dimension it provides a type-theoretic integration of the three prevalent data models, namely the object, relational, and semi-structured models of data. In the control dimension Cω provides elegant primitives for asynchronous communication. In this paper we concentrate on the data dimension. Our aim is to describe the *essence* of these extensions; by which we mean we identify, exemplify and formalize their essential features. Our tool is a small core language, FCω, which is a valid subset of the full Cω language. Using this core language we are able to formalize both the type system and the operational semantics of the data access fragment of Cω.

1 Introduction

Programming languages, like living organisms, need to continuously evolve in response to their changing environment. These evolutionary steps are typically quite modest: most commonly the provision of better or reorganized APIs. Occasionally a more radical evolutionary step is taken. One such example is the addition of generic classes to both Java [6] and C$^\sharp$[25].

We should like to argue that the time has come for another large evolutionary step to be taken. Much software is now intended for distributed, web-based scenarios. It is typically structured using a three-tier model consisting of a *middle tier* containing the business logic that extracts relational data from a *data services tier* (a database) and processes it to produce semi-structured data (typically XML) to be displayed in the *user interface tier*.

It is the writing of these middle tier applications that we should like to address. These applications are most commonly written in an object-oriented language such as Java or C$^\sharp$ and have to deal with relational data (essentially SQL tables), object graphs, and semi-structured data (XML, HTML).

In addition, these applications are fundamentally concurrent. Because of the inherent latency in network communication, the more natural model of concurrency is asynchronous. Accordingly, Cω provides a simple model of asynchronous (one-way) concurrency based on the join calculus [12]. For the rest of this paper, we shall focus exclusively on the data access aspects of Cω; the concurrency primitives have been discussed

elsewhere [3]. Thus when we write Cω, we mean the language excluding the concurrency primitives.

Unfortunately common programming practice, and native API support for data access (e.g. JDBC and ADO.NET) leave a lot to be desired. For example, consider the following fragment taken (and mildly adapted) from the JDBC tutorial to query a SQL database (a user-supplied country is stored in variable `input`).

```
Connection con = DriverManager.getConnection(...);
Statement stmt = con.createConnection();
String query = "SELECT * FROM COFFEES WHERE Country='"+input+"'";
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
  String s = rs.getString("Cof_Name");
  float n = rs.getFloat("Price");
  System.out.println(s+" - "+n);
}
```

Using strings to represent SQL queries is not only clumsy but also removes any possibility for static checking. The impedance mismatch between the language and the relational data is quite striking; e.g. a value is projected out of a row by passing a string denoting the column name and using the appropriate conversion function. Perhaps most seriously, the passing of queries as strings is often a security risk (the ‘‘script code injection” problem—e.g. consider the case when the variable `input` is the string `"' OR 1=1; --"`) [17].

Unfortunately API support in both Java and C$\sharp$ for XML and XPath/XQuery is depressingly similar (even those APIs that map XML values tightly to an object representation still offer querying facilities by string passing).

Our contention is that object-oriented languages need to evolve to support data access satisfactorily. This is hardly a new observation; a large number of academic languages have offered such support for both relational and semi-structured data (see, e.g. [1, 2, 20, 19, 15, 4]). In spite of the obvious advantages of these languages, it appears that their acceptance has been hampered by the fact that they are ‘‘different” from more mainstream application languages, such as Java and C$\sharp$. For example, HaskellDB [19] proposes extensions to the lazy functional language, Haskell; and TL [20] is a hybrid functional/imperative language with advanced type and module systems. We approach this language support problem from a different direction, which is to extend the common application languages themselves rather than creating another new language.

Closer to our approach is SQLJ [24]. This defines a way of embedding SQL commands directly in Java code. Moreover the results of SQL commands can be stored in Java variables and *vice versa*. Thus SQL commands are statically checked by the SQLJ compiler. SQLJ compilation consists of two stages; first to pre-process the embedded SQL, and second the ‘pure Java’ compilation. Thus the embedded SQL code is not part of the language *per se* (in fact all the embedded code is prefixed by the keyword `#sql`). The chief difference is that Cω offers an integration of both the XML and relational data models with an object model.

Design objectives of Cω The aim of our project was to evolve an existing language, C$\sharp$, to provide first-class support for the manipulation of relational and semi-structured data. (Although we have started with C$\sharp$, our extensions apply equally well to other object-oriented languages, including Java.)

Addressing the title of our paper, the essence of the resulting language, Cω, is twofold: its extensions to the C$\sharp$ type system and, perhaps more importantly, the elegant provision of query-like capabilities (the sub-title of our paper). Cω has been carefully designed around a set of core design principles.

1. Cω is a coherent extension of (the safe fragment of) C$\sharp$, i.e. C$\sharp$ programs should be valid Cω programs with the same behaviour.
2. The type system of Cω is intended to be both as simple as possible and closely aligned to the type system in the XPath/XQuery standard. Our intended users are C$\sharp$ programmers who are familiar with XPath/XQuery.
3. From a programming perspective, the real power of Cω comes from its query-like capabilities. These have been achieved by generalizing member access to allow simple XPath-like path expressions.

Paper organization The rest of the paper is organized as follows. In §2 we give a comprehensive overview to the Cω programming language.[4] In §3.1 we identify and formalize FCω, a core fragment of Cω. In §3.2 we detail a simpler fragment, ICω, and in §3.4 show how FCω can be compiled to ICω. Using this compilation, we are able to show a number of properties of FCω in §3.5, including a type soundness theorem. We briefly discuss some related work in §4 and conclude in §5.

2 An introduction to Cω

Our design goal was to evolve C$\sharp$ to provide an integration of the object, relational and semi-structured data models. One possibility would be to add these data models to our programming language in an orthogonal way, e.g. by including new types `XML<`S`>` and `TABLE<`R`>`, where S and R are XML and relational schema respectively. We have sought to integrate these models by *generalization*, rather than by ad-hoc specializations. In the rest of this section we shall present the key ideas behind Cω, and give a number of small programs to illustrate these ideas. This section should serve as a programmer's introduction to Cω. We assume that the reader is familiar with C$\sharp$/Java-like languages.

2.1 New types

Cω is an extension of C$\sharp$, so the familiar primitive types such as integers, booleans, floats are present, as well as classes and interfaces. In this section we shall consider in turn the extensions to the type system—streams, anonymous structs, discriminated unions, and content classes—and for each consider the new query capabilities.

[4] An preliminary version of Cω was (informally) described in [22]. We have subsequently simplified the language, and our chief contribution here is a formalization (§§3–4).

Streams The first structural type we add is a stream type; streams represent ordered homogeneous collections of zero or more values. For example, `int*` is the type for homogeneous sequences of integers. Streams in Cω are aligned with iterators, which will appear in C$^\sharp$ 2.0. Cω streams are typically generated using iterators, which are blocks that contain `yield` statements. For example, the `FromTo` method:

```
virtual int* FromTo(int b, int e){
  for (i = b; i <= e; i++) yield return i;
}
```

generates a finite, increasing stream of integers. Importantly, it should be noted that, just as for C$^\sharp$, invoking such a method body does *not* immediately execute the iterator block, but rather immediately returns a closure. (Thus Cω streams are essentially lazy lists, in the Haskell sense.) This closure is consumed by the `foreach` statement, e.g. the following code fragment builds a finite stream and then iterates over the elements, printing each one to the screen.

```
int* OneToHundred = FromTo(1,100);
foreach (int i in OneToHundred) Console.WriteLine(i);
```

A vital aspect of Cω streams is that they are always *flattened*; there are no nested streams of streams. Cω streams thus coincide with XPath/XQuery sequences which are also flattened. This alignment is a key design decision for Cω: it enables the semantics of our generalized member access to match the path selection of XQuery. We give further details later.

In addition, flattening of stream types also allows us to efficiently deal with recursively defined streams. Consider the following recursive variation of the function `FromTo` that we defined previously:

```
virtual int* FromTo2(int b, int e){
  if (b>e) yield break;
  yield return b;
  yield return FromTo2(b+1,e);
}
```

The statement `yield break;` returns the empty stream. The non-recursive call `yield return b` yields a single integer. The recursive call `yield return FromTo2(b+1,n);` yields a stream of integers. As the type system treats the types `int*` and `int**` as equivalent this is type correct.

Without flattening we would be forced to copy the stream produced by the recursive invocation, leading to a quadratic instead of a linear number of `yields`:

```
virtual int* FromTo3(int b, int e){
  if (b>e) yield break;
  yield return b;
  foreach (int i in FromTo3(b+1,e))  yield return i;
}
```

Note that Cω's flattening of stream types does *not* imply that the underlying stream is flattened via some coercion; every element in a stream is `yield`-ed at most once. As we will see in the operational semantics (§3.3), iterating over a stream will effectively perform a depth-first traversal over the n-ary tree produced by the iterator.

Cω offers a limited but extremely useful form of *covariance* for streams. Covariance is allowed provided that the conversion on the element type is the identity; for example `Button*` is a subtype of `object*` whereas `int*` is *not* (as the conversion from `int` to `object` involves boxing). This notion is a simple variant of the notion of covariance for arrays in C$^\sharp$, although it is statically safe (unlike array covariance) as we can not overwrite elements of streams.

The rationale for this is that implicit conversions should be limited to constant-time operations. Coercing a stream of type `Button*` to type `object*` takes constant-time, whereas coercing `int*` to `object*` would be linear in the length of the stream, as the boxing conversion from `int` to `object` is not the identity.

A key programming feature of Cω is generalized member access; as the subtitle suggests the familiar 'dot' operator is now much more powerful. Thus if the receiver is a stream the member access is mapped over the elements, e.g. `OneToHundred.ToString()` implicitly maps the method call over the elements of the stream `OneToHundred` and returns a value of type `string*`. This feature significantly reduces the burden on the programmer. Moreover, member access has been generalized so it behaves like a *path expression*. For example, `OneToHundred.ToString().PadLeft(10)` converts all the elements of the stream `OneToHundred` to a string, and then pads each string, returning a stream of these padded strings.

Sometimes one wishes to map more than a simple member access over the elements of a stream. Cω offers a convenient shorthand called an *apply-to-all expression*, written e`.{`$\overline{s}$`}`, which applies the block `{`$\overline{s}$`}`, where $\overline{s}$ denotes a sequence of statements, to each element in the stream e.[5] The block may contain the variable `it` which plays a similar role as the implicit receiver argument `this` in a method body and is bound to each successive element of the iterated stream. (Such expressions are reminiscent of Smalltalk `do:` methods.) For example, the following code first creates the stream of natural numbers from 1 to 256, converts each of the elements to a hex string, converts each of these to upper case, and then applies an apply-to-all expression to print the elements to the screen:

```
FromTo(1,256).ToString("x").ToUpper().{ Console.WriteLine(it); };
```

Anonymous structs The second structural type we add are anonymous structs, which encapsulate heterogeneous ordered collections of values. An anonymous struct is like a tuple in ML or Haskell and is written as `struct{int i; Button;}` for example. A value of this type contains a member `i` of type `int` and an unlabelled member of type `Button`. We can construct a value of this type with the expression: `new{i=42, new Button()}`.

To access components of anonymous structs we (again) generalize the notion of member access. Thus assuming a value `x` of the previous type, we write `x.i` to access

[5] We shall adopt the FJ shorthand [18] and write $\overline{x}$ to mean a sequence of x.

the integer value. Unlabelled members are accessed by their position; for example `x[1]` returns the `Button` member. As for streams, member access is lifted over unlabelled members of anonymous structs. To access the `BackColor` property of the `Button` component in variable `x` we can just write `x.BackColor`, which is equivalent to `x[1].BackColor`.

At this point we can reveal even more of the power of Cω's generalized member access. Given a stream `friends` of type `struct{string name;int age;}*`, the expression `friends.age` returns a stream of integers. The member access is over *both* structural types. The following query-like statement prints the names of one's friends:

```
friends.name.{ ConsoleWriteLine(it);};
```

Interestingly, Cω also allows repeated occurrences of the same member name within an anonymous struct type, even at different types. For example, assume the following declaration: `struct{int i; Button; float i;} z;` Then `z.i` projects the two `i` members of `z` into a new anonymous struct that is equivalent to `new{z[0],z[2]}` and of type `struct{int;float;}`.

Cω provides a limited form of covariance for anonymous structs, just as for streams. For example, the anonymous struct `struct{int;Button;}` *is* a subtype of `struct{int; Control;}`. However it is *not* a subtype of `struct{object; Control;}` since the conversion from `int` to `object` is not an identity conversion. Cω does not support width subtyping for anonymous structs.

Choice types The third structural type we add is a particular form of discriminated union type, which we call a choice type. This is written, for example, `choice{int; bool;}`. As the name suggests, a value of this type is either an integer or a boolean, and may hold either at any one time. Unlike unions in C/C++ and variant records in Pascal where users have to keep track of which type is present, values of a discriminated union in Cω are implicitly tagged with the static type of the chosen alternative, much like unions in Algol68. In other words, discriminated union values are essentially a pair of a value and its static type.

There is no syntax for creating choice values; the injection is implicit (i.e. it is generated by the compiler).

```
choice{int;Button;} x = 3;
choice{int;Button;} y = new Button();
```

Cω provides a test, e `was` τ, on choice values to test the value's *static* type. Thus `x was int` would return `true`, whereas `y was int` would return `false`.

Assuming that an expression e is of type `choice{`$\overline{\tau}$`}`, the expression e `was` τ is true for *exactly one* τ in $\overline{\tau}$. This invariant is maintained by the type system. The only slight complication arises from subtyping, e.g.

```
choice{Control; object;} z = new Button();
```

As `Button` is a subtype of both `Control` and `object`, which type tag is generated by the compiler? A choice type can be thought of as providing a *family* of overloaded constructor methods, one for each component type. Just as for standard object creation in

Java/C♯, the *best* constructor method is chosen. In the example above, clearly `Control` is better than `object`. Thus `z was Control` returns `true`. The notion of ‘‘best” for Cω is the routine extension of that for C♯.

As the reader may have guessed, member access has also been generalized over discriminated unions. Here the behaviour of member access is less obvious, and has been designed to coincide with XPath. Consider a value `w` of type `choice{char; Button;}`. The member access `w.GetHashCode()` succeeds irrespective of whether the value is a character or a `Button` object. In this case the type of the expression `w.GetHashCode()` is `int`.

However the member may not be supported by all the possible component types, e.g. `w.BackColor`. Classic treatments of union types would probably consider this to be type incorrect [23, p.207]. However, Cω’s choice types follow the semantics of XPath where, for example, the query `foo/bar` returns the `bar` nodes under the `foo` node if any exist, and *the empty sequence* if none exist. Thus in Cω, the expression `w.BackColor` is well-typed, and will return a value of type `Color?`. This is another new type in Cω and is a variant of the nullable type to appear in C♯ 2.0. A value of type `Color?` can be thought of as a singleton stream, thus it is either empty or contains a single `Color` value (when `w` contains a `Button`). Again, we emphasize that this behaviour precisely matches that of XPath.

Cω follows the design of C♯ in allowing all values to be boxed and hence all value types are a subtype of the supertype `object`. Thus both anonymous structs and choice types are considered to be subtypes of the class `object`.

Content classes To allow close integration with XSD and other XML schema languages, we have included the notion of a *content class* in Cω. A content class is a normal class that has a single *unlabelled* type that describes the content of that class, as opposed to the more familiar (named) fields. The following is a simple example of a content class.

```
class friend{
  struct{ string name; int age; };
  void incAge(){...}
}
```

Again we have generalized member access over content classes. Thus the expression `Bill.age` returns an integer, where `Bill` is a value of type `friend`.

From an XSD perspective, classes correspond to global element declarations, while the content type of classes correspond to complex types. Further comparisons with the XML data model are immediately below, but a more comprehensive study can be found elsewhere [21].

2.2 XML programming

It should be clear that the new type structures of Cω are sufficient to model simple XML schema. For example, the following XSD schema

```
<element name="Address"><complexType><sequence>
  <choice>
    <element name="Street" type="string"/>
    <element name="POBox" type="int"/>
  </choice>
  <element name="City" type="string"/>
</sequence></complexType></element>
```

can be represented (somewhat more succinctly!) as the Cω content class declaration:

```
class Address {
  struct{
    choice{ string Street; int POBox; };
    string City;
  };
}
```

The full Cω language supports XML literals as syntactic sugar for serialized object graphs. For example, we can create an instance of the `Address` class above using the following literal:

```
Address a = <Address>
              <Street>13 Elm St</Street>
              <City>Hollywood</City>
            </Address>;
```

The Cω compiler contains a validating XML parser that deserializes the above literal into normal constructor calls. XML literals can also contain typed holes, much as in XQuery, that allow us to embed expressions to compute part of the literal. This is especially convenient for generating streams.

The inclusion of XML literals and the semantics of the generalized member access mean that XQuery code can be almost directly written in Cω. For example, consider one of the XQuery Use Cases [9], that processes a bibliography file (assume that this is stored in variable `bs`) and for each book in the bibliography, lists the title and authors, grouped inside a `result` element. The suggested XQuery solution is as follows.

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

The Cω solution is almost identical:

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

The full Cω language adds several more powerful query expressions to those discussed in this paper. For instance, filter expressions e`[`e'`]` return the elements in the stream e that satisfy the boolean expression e'. As labels can be duplicated in anonymous structs and discriminated unions, the full language also allows type-based selection. For example, given a value `x` of type `struct{ int a; struct{string a;};}` we can select only the `string` member `a` by writing `x.string::a`.

Transitive queries are also supported in the full Cω language: the expression $e...\tau::m$ selects all members m of type τ that are transitively reachable from e. Transitive queries are inspired by the XPath descendant axis.

2.3 Database programming

Relational tables are merely streams of anonymous structs. For example, the relational table created with the SQL declaration:

```
CREATE TABLE Customer (name string, custid int);
```

can be represented in Cω: `struct{string name; int custid}* Customer;`

In addition to path-like queries, the full Cω language also supports familiar SQL expressions, including `select-from-where`, various joins and grouping operators. Perhaps more importantly, these statements can be used on *any* value of the appropriate type, whether that value resides in a database or in memory; hence, one can write SQL queries in Cω code that does not access a database! One of the XQuery use-cases [9] asks to list the title prices for each book that is sold by both booksellers `A` and `BN`. Using a `select` statement and XML-literals, this query can be written in Cω as the following expression:

```
select <book-with-prices>
         <title>{a.title}</title>
         <price-A>{a.price}</price-A>
         <price-BN>{bn.price}</price-BN>
       </book-with-prices>
from book a in A.book, book bn in BN.book
where a.title == bn.title
```

Note the use of XML placeholders `{a.title}` and `{bn.price}`: when this code is evaluated new titles and new prices are computed from the bindings of the `select-from-where` clause.

So far we have shown how we can query values using generalized member and SQL expressions, but as Cω is an imperative language, we also allow to perform updates. This paper, however, focuses on the type extensions and generalized member access only.

3 The essence of Cω

In the rest of this paper we study formally the essence of Cω, by which we mean we identify its essential features. We adopt a formal, mathematical approach and define a core calculus, Featherweight Cω, or FCω for short, similar to core subsets of Java such as FJ [18], MJ [5] and ClassicJava [11]. This core calculus, whilst lightweight, offers a similar computational ‘‘feel” to the full Cω language: it supports the new type constructors and generalized member access. FCω is a completely valid subset of Cω in that every FCω program is literally an executable Cω program.

The rest of this section is organized as follows. In §3.1 we define the syntax and type system for FCω. Rather than give an operational semantics directly for FCω we prefer to first ''compile out" some of its features, in particular generalized member access. This both greatly simplifies the resulting operational semantics and demonstrates that Cω's features do not require extensive new machinery. Thus in §3.2 we define a target language, Inner Cω, or ICω, for this ''compilation". ICω is essentially the same language, but for a handful of new language constructs and a much simpler type system. In §3.3 we give an operational semantics for ICω programs. In §3.4 we specify the compilation of FCω programs into ICω programs. This translation is, on the whole, quite straightforward. We conclude the section in §3.5 by stating some properties of our calculi and the compilation. Most important is the type-soundness property for ICω. Space prevents us from providing any details of the proofs, but they are proved using standard techniques and are similar to analagous theorems for fragments of Java [18, 5].

3.1 A core calculus: FCω

Syntax An FCω program consists of one or more class declarations. Each class declaration defines zero or more methods and contains exactly one unlabelled type that we call the *content type*. (We can code up a conventional C$^\sharp$/Cω class declaration with a number of field declarations using an anonymous struct.) FCω follows C$^\sharp$ and requires methods to be explicitly marked as `virtual` or `override`. Given a program we assume that there is a unique designated method within the class declarations that serves as the entry point.

Program	p	$::= \overline{cd}$
Class Definition	cd	$::=$ `class` c`:` c `{`τ`;`$\overline{md}$`}`
Method Definition	md	$::=$ `virtual` τ m`(`$\overline{\tau}$ $\overline{x}$`){`$\overline{s}$`}`
		$\mid$ `override` τ m`(`$\overline{\tau}$ $\overline{x}$`){`$\overline{s}$`}`

FCω supports two main kinds of types: *value types* and *reference types*. As usual, the distinguished type `void` is used for methods that do not return anything; `null` is only used to type `null` references, as with C$^\sharp$. Value types include the base types `bool` and `int` and the structural types: anonymous structs and discriminated unions. Reference types are either class types or streams. As usual only reference types have object identity and are represented at runtime by references into the heap. We assume a designated special class `object`.

Types

τ	$::= \gamma$	Value types
	$\mid \rho$	Reference types
	$\mid$ `void` $\mid$ `null`	Void and null types

Value Types

γ	$::= b$	Base types
	$\mid$ `struct{`$\overline{fd}$`}`	Anonymous structs
	$\mid$ `choice{`$\overline{\kappa}$`}`	Choice types

Base Types

$b ::=$ `bool` $\mid$`int`

Reference Types

ρ	$::= c$	Classes
	$\mid \sigma$`*`	Stream types
	$\mid \sigma$`?`	Singleton stream type

Field Definition

fd	$::= \tau\, f$`;`	Named member
	$\mid \tau$`;`	Unnamed member

We employ the shorthand κ and σ to denote any type *except* a choice type and stream type (singleton or non-singleton), respectively. As Cω flattens stream types, we have made the simplification to FCω of removing nested stream types altogether from the

type grammar. We have also simplified FCω choice types so that the members are unlabelled and we also exclude (for simplification) nested choice types. These can be coded up in FCω using unlabelled anonymous structs.

FCω expressions, as for C$^\sharp$, are split into ordinary expressions and promotable expressions. Promotable expressions are expressions that can be used as statements. We assume a number of built-in primitive operators, such as `==`, `||` and `&&`. In the grammar we write $e \oplus e$, where $\oplus$ denotes an instance of one of these operators. We do not formalize these operators further as their meaning is clear.

Expression

$e ::=$	$b \mid i$	Literals
$\mid$	$e \oplus e$	Built-in operators
$\mid$	x	Variable
$\mid$	`null`	Null
$\mid$	`(`τ`)`e	Cast
$\mid$	e `is` τ	Dynamic typecheck
$\mid$	e `was` κ	Static typecheck for choice values
$\mid$	`new` τ`(`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

We have made a simplification in the interests of space to restrict apply-to-all expressions to contain an expression rather than a sequence of statements. This simplifies the typing rules, but as apply-to-all expressions can be coded using `foreach` loops it is not a serious restriction.

Statements in FCω are standard. As mentioned earlier we have adopted the `yield` statement that will appear in C$^\sharp$ 2.0 to generate streams.

Statement	$s ::=$	`;`	Skip
	$\mid$	pe`;`	Promoted expression
	$\mid$	`if (`e`)` s `else` s	Conditional
	$\mid$	τ 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 (`σ x `in` e`)` s	Foreach loop
	$\mid$	`while (`e`)` s	While loop
	$\mid$	`{`$\overline{s}$`}`	Block

In what follows we assume that FCω programs are well-formed, e.g. no cyclic class hierarchies, correct method body construction, etc. These conditions can be easily formalized but we suppress the details for lack of space.

Subtyping Before we define the typing judgements for FCω programs we need to define a number of auxiliary relations. First we define the subtyping relation. We write $\tau <: \tau'$ to mean that type τ is a subtype of type τ'. The rules defining this relation are as follows.

$$\frac{}{\tau <: \tau}\ [\text{Refl}] \qquad \frac{\tau <: \tau' \qquad \tau' <: \tau''}{\tau <: \tau''}\ [\text{Trans}] \qquad \frac{}{\gamma <: \texttt{object}}\ [\text{Box}] \qquad \frac{\texttt{class}\ c : c'}{c <: c'}\ [\text{Sub}]$$

$$\frac{}{\texttt{null} <: \rho}\ [\text{Null}] \qquad \frac{\tau <: \tau' \qquad f = f'}{\tau\ f <: \tau'\ f'}\ [\text{FD}] \qquad \frac{\sigma <: \sigma' \qquad IdConv(\sigma, \sigma')}{\sigma\texttt{*/?} <: \sigma'\texttt{*/?}}\ [\text{Stream}]$$

$$\frac{}{\sigma\texttt{*} <: \texttt{object}}\ [\text{SBox}] \qquad \frac{}{\sigma\texttt{?} <: \sigma\texttt{*}}\ [\text{SSub}] \qquad \frac{}{\sigma <: \sigma\texttt{?}}\ [\text{Sing}]$$

$$\frac{\overline{fd} <: \overline{fd'} \qquad IdConv(\overline{fd}, \overline{fd'})}{\texttt{struct\{}\overline{fd}\texttt{\}} <: \texttt{struct\{}\overline{fd'}\texttt{\}}}\ [\text{Struct}] \qquad \frac{}{\kappa <: \texttt{choice\{}\kappa\texttt{;}\ \overline{\kappa'}\texttt{\}}}\ [\text{SubChoice}]$$

$$\frac{\overline{\kappa} <: \overline{\kappa'} \qquad IdConv(\overline{\kappa}, \overline{\kappa'})}{\texttt{choice\{}\overline{\kappa}\texttt{\}} <: \texttt{choice\{}\overline{\kappa'}\ \overline{\kappa''}\texttt{\}}}\ [\text{Choice}]$$

Most of these rules are straightforward. The rule [Stream] contains notation ($\sigma\texttt{*/?}$) that we use throughout this paper. It is uses to denote two instances of the rule, one where we select the left of the '/' in all cases (in this case $\sigma\texttt{*}$) and one where we select the right in all cases. It does *not* include cases where we individually select left and right alternatives. The rules [Stream] and [Struct] make use of a predicate $IdConv$, which relates two types τ and τ' if there is an identity conversion between them. Thus $IdConv(\texttt{Button}, \texttt{object})$ but not $IdConv(\texttt{int}, \texttt{object})$. In this short paper we shall not give its straightforward definition.

Generalized member access As we have seen a key programming feature of Cω is generalized member access. Capturing this behaviour in the type system can be tricky, but we have adopted a rather elegant solution, whereby we define two auxiliary relations. The first, written $\tau.f : \tau'$, tells us that given a value of type τ accessing member f will return a value of type τ'. We define a similar relation for function member access, written $\tau.m(\overline{\tau'}) : \tau''$. Having generalized member access captured by a separate typing relation greatly simplifies the typing judgements for expressions. As generalized member access is a key feature of Cω, we shall give it in detail.

The definition of this relation over stream types is as follows.

$$\frac{\sigma.f : \sigma'}{\sigma\texttt{*}.f : \sigma'\texttt{*}} \qquad \frac{\sigma.f : \sigma'\texttt{*/?}}{\sigma\texttt{*}.f : \sigma'\texttt{*}} \qquad \frac{\sigma.m(\overline{\tau}) : \sigma'}{\sigma\texttt{*}.m(\overline{\tau}) : \sigma'\texttt{*}} \qquad \frac{\sigma.m(\overline{\tau}) : \sigma'\texttt{*/?}}{\sigma\texttt{*}.m(\overline{\tau}) : \sigma'\texttt{*}} \qquad \frac{\sigma.m(\overline{\tau}) : \texttt{void}}{\sigma\texttt{*}.m(\overline{\tau}) : \texttt{void}}$$

The first two rules map the member access over the stream elements, making sure that we do not create a nested stream type. The next two rules for function member access are similar. The last rule captures the intuition that mapping a `void`-valued method over a stream, forces the evaluation of the stream and does not return a value.

Before defining the rules for member access over anonymous structs, we need to define rules for member access over named field definitions. This is pretty straightforward and as follows.

$$\frac{}{\tau\ f.f : \tau} \qquad \frac{\tau.m(\overline{\tau'}) : \tau''}{\tau\ f.m(\overline{\tau'}) : \tau''}$$

Now we consider the rules for generalized member access over anonymous structs. First we give the degenerate cases where only one component supports the member access.

$$\frac{\exists! k \in \{1 \ldots n\}.\ fd_k.f : \tau_k}{\texttt{struct\{}fd_1\texttt{;} \ldots fd_n\texttt{;\}}.f : \tau_k} \qquad \frac{\exists! k \in \{1 \ldots n\}.\ fd_k.m(\overline{\tau'}) : \tau''}{\texttt{struct\{}fd_1\texttt{;} \ldots fd_n\texttt{;\}}.m(\overline{\tau}) : \tau''}$$

The non-degenerate cases are then as follows.

$$\frac{\exists S \subseteq \{1 \ldots n\}.|S| \geq 2 \wedge p = |S| \wedge \forall k \in [1..p].fd_{S_k}.f : \tau_k}{\texttt{struct\{}fd_1\texttt{;} \ldots fd_n\texttt{;\}}.f : \texttt{struct\{}\tau_1\texttt{;} \ldots \tau_p\texttt{;\}}}$$

$$\frac{\exists S \subseteq \{1 \ldots n\}.|S| \geq 2 \wedge p = |S| \wedge \forall k \in [1..p].\ fd_{S_k}.m(\overline{\tau}) : \tau'_k}{\texttt{struct\{}fd_1\texttt{;} \ldots fd_n\texttt{;\}}.m(\overline{\tau}) : \texttt{struct\{}\tau'_1\texttt{;} \ldots \tau'_p\texttt{;\}}}$$

Thus a subset, S, of the components support the member, and we map the member access over these components in order. The overall return type is an anonymous struct of the component return types.

We now consider the rules for generalized member access over choice types. Again we consider these rules depending on how many components support the member access. First we give the simple case when *all* possible components support the member access.

$$\frac{\forall k \in \{1 \ldots n\}.\ \kappa_k.f : \tau}{\texttt{choice\{}\kappa_1\texttt{;} \ldots \kappa_n\texttt{;\}}.f : \tau} \qquad \frac{\forall k \in \{1 \ldots n\}.\ \kappa_k.m(\overline{\tau}) : \tau'}{\texttt{choice\{}\kappa_1\texttt{;} \ldots \kappa_n\texttt{;\}}.m(\overline{\tau}) : \tau'}$$

We also have the case when only one of the possible components supports the member access. These rules are as follows (we omit the nested cases).

$$\frac{\exists! k \in \{1 \ldots n\}.\ \kappa_k.f : \sigma \quad n > 1}{\texttt{choice\{}\kappa_1\texttt{;} \ldots \kappa_n\texttt{;\}}.f : \sigma\texttt{?}} \qquad \frac{\exists! k \in \{1 \ldots n\}.\ \kappa_k.m(\overline{\tau}) : \sigma \quad n > 1}{\texttt{choice\{}\kappa_1\texttt{;} \ldots \kappa_n\texttt{;\}}.m(\overline{\tau}) : \sigma\texttt{?}}$$

The reader will recall that the return type of this generalized member access involves a singleton stream type. Finally we give the cases where more than one of the possible components supports the member access.

$$\frac{\exists S \subseteq \{1 \ldots n\}.|S| \geq 2 \wedge p = |S| \wedge \forall k \in [1..p].\ \kappa_{S_k}.f : \kappa'_k}{\texttt{choice\{}\kappa_1\texttt{;} \ldots \kappa_n\texttt{;\}}.f : \texttt{choice\{}\kappa'_1\texttt{;} \ldots \kappa'_p\texttt{;\}?}}$$

$$\frac{\exists S \subseteq \{1 \ldots n\}.|S| \geq 2 \wedge p = |S| \wedge \forall k \in [1..p].\ \kappa_{S_k}.m(\overline{\tau}) : \kappa'_k}{\texttt{choice\{}\kappa_1\texttt{;} \ldots \kappa_n\texttt{;\}}.m(\overline{\tau}) : \texttt{choice\{}\kappa'_1\texttt{;} \ldots \kappa'_p\texttt{;\}?}}$$

Generalized member access over singleton streams is relatively straightforward; the only complication being again to ensure that no nested streams are generated.

$$\frac{\sigma.f\colon\sigma'}{\sigma?.f\colon\sigma'?} \quad \frac{\sigma.f\colon\sigma'*/?}{\sigma?.f\colon\sigma'*/?} \quad \frac{\sigma.m(\overline{\tau})\colon\sigma'}{\sigma?.m(\overline{\tau})\colon\sigma'?} \quad \frac{\sigma.m(\overline{\tau})\colon\sigma'*/?}{\sigma?.m(\overline{\tau})\colon\sigma'*/?}$$

Finally we need to define rules for generalized member access over classes. Clearly these need to reflect the standard C♯ semantics: function member access on classes searches the class hierarchy until a matching method is found. If we find a matching method $\tau' m(\overline{\tau''})$ in class c, we need to check the actual types of the arguments to the types expected by m. This behaviour is given by the following two rules.

$$\frac{\texttt{class}\ c\colon c'\texttt{\{}\tau\texttt{;}\overline{md}\texttt{\}} \quad \tau' m(\overline{\tau''}) \in \overline{md} \quad \overline{\tau} <: \overline{\tau''}}{c.m(\overline{\tau})\colon\tau'}$$

$$\frac{\texttt{class}\ c\colon c'\texttt{\{}\tau\texttt{;}\overline{md}\texttt{\}} \quad \tau' m(\overline{\tau''}) \notin \overline{md} \quad c'.m(\overline{\tau})\colon\tau'}{c.m(\overline{\tau})\colon\tau'}$$

Next we consider the rules for generalized field access. There is a small subtlety here concerning recursive class definitions; consider the following recursive class `List` of lists of integers: `class List { struct{ int head; List; } }`

Given an instance `xs` of type `List`, we do not want `xs.head` to recursively select all `head` fields in `xs`. However simply unfolding the content type and using the rules given earlier for generalized access over anonymous structs that is precisely what would happen!

There are a number of solutions, but in order to make the Cω type system as simple as possible, we follow e.g. Haskell and SML and break recursive cycles at nominal types. In our setting that means that we simply do not perform member lookup on nominal members of the content of nominal types. Using these refined rules, the result type of `xs.head` is `int`.

Formalizing this is trivial but time-consuming. We define another family of generalized member access judgements, written $\tau \bullet f\colon\tau'$, which is identical to the previous rules except they are not defined for nominal types. We elide the definitions here.

To define field access on nominal types, we first define formally the content type of a class, written $content(c)$ for some class c, as follows.

$$\frac{\texttt{class}\ c\texttt{:object\{}\tau\texttt{;}\overline{md}\texttt{\}}}{content(c) = \tau} \qquad \frac{\texttt{class}\ c : c'\texttt{\{}\tau\texttt{;}\overline{md}\texttt{\}} \quad content(c') = \tau'}{content(c) = \texttt{struct\{}\tau'\texttt{;}\tau\texttt{;\}}}$$

The rule for generalized member access over classes simply searches for the member f on the content type of class c, and is given by the following rule.

$$\frac{content(c) = \tau \quad \tau \bullet f : \tau'}{c.f\colon\tau'}$$

Generalized index access As we mentioned earlier, elements of anonymous structs can be accessed by position. This is captured by the following rule.

$$\frac{type(fd_i) = \tau_i}{\texttt{struct \{}fd_1\texttt{;} \ldots fd_n\texttt{;\}[}i\texttt{]:}\tau_i}$$

As the reader might have expected, this index access is generalized over the other types; the rather routine details are omitted.

Typing judgements We are now able to define typing judgements for FCω. We define three relations corresponding to the three syntactic categories of expressions, promotable expressions and statements. For all three judgements we write Γ to mean a partial function from program identifiers to types. The judgements for expressions and promotable expressions are written $\Gamma \vdash e : \tau$ and $\Gamma \vdash pe : \tau$, respectively. These are given in Fig. 1.

Most of these rules are routine; we shall discuss a few of the more interesting details here. In the rule [TStruct], we have made use of a typing judgement for a binding expression. This is defined as follows:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash f \texttt{ = } e : \tau\ f}$$

The compactness of the rules [TField], [TIndex] and [TMeth] shows the elegance of having captured generalized member access with auxiliary relations.

The rules [TAAExp1] and [TAAExp2] ensure that the return type of an apply-to-all expression is not nested. The rule [TAAExp3] ensures the appropriate mixed flattening of streams. The rule [TAAExp4] captures the intuition that applying a `void`-typed expression to a stream forces the evaluation of that stream and hence the overall type is also `void`.

The typing judgement for FCω statements is written $\Gamma; \tau \vdash s$ and is intended to mean that a statement s is well-typed in the typing environment Γ. If it returns a value (either via a normal `return` or a `yield return`) then that value is of type τ.

The rules [TForeach1] and [TForeach2] reflect the fact that the type of the stream elements can be cast to the type of the bound variable. This can be either via an upcast ([TForeach1]) or a downcast ([TForeach2]) (again this matches C$\sharp$ 2.0).

3.2 An inner calculus: ICω

Rather than consider further our featherweight calculus FCω, we shall in fact define another core calculus for Cω. This inner calculus, called ICω, is intended to be similar but lower-level than FCω; it can be thought of as the internal language of a compiler.

The chief simplification in ICω is that its type system does *not* support generalized member access. The intention is that we compile out generalized member access when translating FCω programs into ICω programs. We give some details of this compilation in §3.4. Apart from a simplified type system, we can define quite simply an operational semantics for ICω; this is given in §3.3.

$\boxed{\Gamma \vdash e : \tau \text{ and } \Gamma \vdash pe : \tau}$

$$\frac{}{\Gamma \vdash i : \texttt{int}}\ [\text{TInt}] \qquad \frac{}{\Gamma \vdash b : \texttt{bool}}\ [\text{TBool}] \qquad \frac{}{\Gamma, x : \tau \vdash x : \tau}\ [\text{TId}] \qquad \frac{}{\Gamma \vdash \texttt{null} : \texttt{null}}\ [\text{TNull}]$$

$$\frac{\Gamma \vdash e : \tau' \quad (\tau' <: \tau) \vee (\tau <: \tau')}{\Gamma \vdash \texttt{(}\tau\texttt{)}\, e : \tau}\ [\text{TSub}] \qquad \frac{\Gamma \vdash e : \tau' \quad (\tau' <: \tau) \vee (\tau <: \tau')}{\Gamma \vdash e\ \texttt{is}\ \tau : \texttt{bool}}\ [\text{TIs}]$$

$$\frac{\Gamma \vdash e : \texttt{choice\{}\overline{\kappa'}\ \kappa; \overline{\kappa''}\texttt{\}}}{\Gamma \vdash e\ \texttt{was}\ \kappa : \texttt{bool}}\ [\text{TWas}] \qquad \frac{\Gamma \vdash \overline{be} : \overline{fd}}{\Gamma \vdash \texttt{new \{}\overline{be}\texttt{\}} : \texttt{struct\{}\overline{fd}\texttt{\}}}\ [\text{TStruct}]$$

$$\frac{\Gamma \vdash e : \tau \quad \tau <: \mathit{content}(c)}{\Gamma \vdash \texttt{new}\ c\texttt{(}e\texttt{)} : c}\ [\text{TNew}] \qquad \frac{\Gamma \vdash e : \tau \quad \tau.f : \tau'}{\Gamma \vdash e.f : \tau'}\ [\text{TField}]$$

$$\frac{\Gamma \vdash e : \tau \quad \tau[i] : \tau'}{\Gamma \vdash e\texttt{[}i\texttt{]} : \tau'}\ [\text{TIndex}] \qquad \frac{\Gamma \vdash x : \tau \quad \Gamma \vdash e : \tau' \quad \tau' <: \tau}{\Gamma \vdash x\texttt{=}e : \tau}\ [\text{TAss}]$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \overline{e'} : \overline{\tau'} \quad \tau.m(\overline{\tau'}) : \tau''}{\Gamma \vdash e.m(\overline{e'}) : \tau''}\ [\text{TMeth}] \qquad \frac{\Gamma \vdash e : \sigma\texttt{*/?} \quad \Gamma, \texttt{it} : \sigma \vdash e' : \sigma'}{\Gamma \vdash e\texttt{.\{}e'\texttt{\}} : \sigma'\texttt{*/?}}\ [\text{TAAExp1}]$$

$$\frac{\Gamma \vdash e : \sigma\texttt{*/?} \quad \Gamma, \texttt{it} : \sigma \vdash e' : \sigma'\texttt{*/?}}{\Gamma \vdash e\texttt{.\{}e'\texttt{\}} : \sigma'\texttt{*/?}}\ [\text{TAAExp2}] \qquad \frac{\Gamma \vdash e : \sigma\texttt{*/?} \quad \Gamma, \texttt{it} : \sigma \vdash e' : \sigma'\texttt{?/*}}{\Gamma \vdash e\texttt{.\{}e'\texttt{\}} : \sigma'\texttt{*}}\ [\text{TAAExp3}]$$

$$\frac{\Gamma \vdash e : \sigma\texttt{*/?} \quad \Gamma, \texttt{it} : \sigma \vdash e' : \texttt{void}}{\Gamma \vdash e\texttt{.\{}e'\texttt{\}} : \texttt{void}}\ [\text{TAAExp4}]$$

$\boxed{\Gamma; \tau \vdash s}$

$$\frac{}{\Gamma; \tau \vdash \texttt{;}}\ [\text{TSkip}] \qquad \frac{\Gamma; \tau \vdash \overline{s}}{\Gamma; \tau \vdash \texttt{\{}\overline{s}\texttt{\}}}\ [\text{TNest}] \qquad \frac{\Gamma \vdash pe : \tau}{\Gamma; \tau' \vdash pe\texttt{;}}\ [\text{TProm}] \qquad \frac{}{\Gamma; \texttt{void} \vdash \texttt{return;}}\ [\text{TRetV}]$$

$$\frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma; \tau \vdash s}{\Gamma; \tau \vdash \texttt{while (}e\texttt{)}\ s}\ [\text{TWhile}] \qquad \frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma; \tau \vdash s_1 \quad \Gamma; \tau \vdash s_2}{\Gamma; \tau \vdash \texttt{if (}e\texttt{)}\ s_1\ \texttt{else}\ s_2}\ [\text{TIf}]$$

$$\frac{\Gamma \vdash e : \tau' \quad \tau' <: \tau}{\Gamma; \tau \vdash \texttt{return}\ e\texttt{;}}\ [\text{TRet}] \qquad \frac{}{\Gamma; \sigma\texttt{*} \vdash \texttt{yield break;}}\ [\text{TYieldB}]$$

$$\frac{\Gamma \vdash e : \sigma' \quad \sigma' <: \sigma}{\Gamma; \sigma\texttt{*} \vdash \texttt{yield return}\ e\texttt{;}}\ [\text{TYield1}] \qquad \frac{\Gamma \vdash e : \sigma'\texttt{*/?} \quad \sigma' <: \sigma'' \quad \Gamma, x : \sigma''; \tau \vdash s}{\Gamma; \tau \vdash \texttt{foreach (}\sigma''\ x\ \texttt{in}\ e\texttt{)}\, s}\ [\text{TForeach1}]$$

$$\frac{\Gamma \vdash e : \sigma\texttt{*} \quad \sigma\texttt{*} <: \sigma'\texttt{*}}{\Gamma; \sigma'\texttt{*} \vdash \texttt{yield return}\ e\texttt{;}}\ [\text{TYield2}] \qquad \frac{\Gamma \vdash e : \sigma'\texttt{*/?} \quad \sigma'' <: \sigma' \quad \Gamma, x : \sigma''; \tau \vdash s}{\Gamma; \tau \vdash \texttt{foreach (}\sigma''\ x\ \texttt{in}\ e\texttt{)}\, s}\ [\text{TForeach2}]$$

Fig. 1. Typing judgements for FCω expressions, promotable expressions and statements

The grammar of ICω is then a simple varianr of the grammar for FCω. Some extra expression and statement forms are added (which reflects the lower-level nature of ICω) and likewise a couple are removed from the grammar as they are redundant. We do not expect these new syntactic forms to be made available to the Cω programmer (although they could be). The extensions are as follows:

Expression

$e ::= \ldots$		
$\mid$ `new` τ`(`$\overline{s}$`)`	Closure creation	
$\mid$ `new (`κ`,`$e`)`	Choice creation	
$\mid$ e`.Content`	Class content	
$\mid$ e `at` κ	Choice content	

Promotable expression

$pe ::= \ldots$	
$\mid$ τ`({`$\overline{s}$`})`	Block expression

Statement

$s ::= \ldots$	
$\mid$ `yield return (`τ`,`$e`);`	Typed yield

Thus ICω includes expressions to create closure and choice elements. We include an operator e`.Content` to extract the content element from an object e. Given an element e of a choice type, we add an operation e `at` κ to extract its κ-valued content. (If it is of another type, this will raise an exception.) We add (typed) block expressions to ICω, and in addition we provide a typed `yield` statement.

The two syntactic forms that we removed from the grammar of FCω are: (1) We remove field accesses e`.`f completely; they are replaced by positional access, i.e. e`[`i`]`; and (2) We remove the untyped `yield` statement; all `yield`s in ICω are explicitly typed.

We can define typing judgements for ICω expressions and statements, which are written $\Gamma \rhd e\colon\tau$ and $\Gamma;\tau \rhd s$, respectively. Most of these rules are identical to those for FCω; we shall just give the rules for the new syntactic forms. The rules for creating closure and choice elements are as follows:

$$\frac{\Gamma;\sigma\texttt{*/?} \rhd \overline{s}}{\Gamma \rhd \texttt{new}\ \sigma\texttt{*/?(}\overline{s}\texttt{)}\colon\sigma\texttt{*/?}} \qquad \frac{\Gamma \rhd e\colon\kappa' \quad \kappa' <\colon \kappa}{\Gamma \rhd \texttt{new (}\kappa\texttt{,}e\texttt{)}\colon\texttt{choice\{}\kappa\texttt{;\}}}$$

The typing rules for extracting the content of content class and choice elements are as follows:

$$\frac{\Gamma \rhd e\colon c}{\Gamma \rhd e\texttt{.Content}\colon \mathit{content}(c)} \qquad \frac{\Gamma \rhd e\colon\texttt{choice\{}\kappa\texttt{;}\overline{\kappa'}\texttt{\}}}{\Gamma \rhd e\ \texttt{at}\ \kappa\colon\kappa}$$

The typing rule for block expressions and `yield` statements are as follows:

$$\frac{\Gamma;\tau \rhd \overline{s} \quad \tau \neq \texttt{void}}{\Gamma \rhd \tau\texttt{(\{}\overline{s}\texttt{\})}\colon\tau} \qquad \frac{}{\Gamma;\sigma\texttt{*/?} \vdash \texttt{yield break;}}$$

$$\frac{\Gamma \rhd e\colon\sigma' \quad \sigma' <\colon \sigma}{\Gamma;\sigma\texttt{*/?} \rhd \texttt{yield return (}\sigma'\texttt{,}e\texttt{);}} \qquad \frac{\Gamma \rhd e\colon\sigma\texttt{*/?} \quad \sigma\texttt{*/?} <\colon \tau \quad \tau \neq \texttt{object}}{\Gamma;\tau \rhd \texttt{yield return (}\sigma\texttt{*/?,}e\texttt{);}}$$

3.3 Operational semantics for ICω

In this section we formalize the dynamics of ICω by defining an operational semantics. We follow FJ [18] and MJ [5] and give this in the form of a small-step reduction relation,

although a big-step evaluation relation can easily be defined. Hence we use evaluation contexts to encode the evaluation strategy in the now familiar way [11]—the definition of ICω evaluation contexts is routine and omitted. First we define the value forms of ICω expressions and statements (where bv is the value form of a binding expression):

Expression values

$v ::= b \mid i \mid$ `null` $\mid$ `void`		Basic values
$\mid r$		Reference
$\mid$ `new {`$\overline{bv}$`}`		Struct value
$\mid$ `new (`κ`,`v`)`		Choice value

Statement values

$sv ::=$ `;`		Skip
$\mid$ `return` v`;`		Return value
$\mid$ `return;`		
$\mid$ `yield return (`τ`,`v`);`		Typed yield value
$\mid$ `yield break;`		End of stream value

Evaluation of ICω expressions and statements takes place in the context of a state, which is a pair (H, R), where H is a heap and R is a stack frame. A heap is represented as a finite partial map from references r to runtime objects, and a stack frame is a finite partial map from variable identifiers to values. A runtime object, as for C$\sharp$, is a pair (τ, cn) where τ is a type and cn is a canonical, which is either a value or a closure. A closure is the runtime representation of a stream and is written as a pair $(R, \overline{s})^{\alpha}$ where R is a stack frame and $\overline{s}$ is a statement sequence. The superscript flag α indicates whether the closure is fresh or a clone. We will explain this distinction later. In what follows we assume that expressions and statements are well-typed.

In Fig. 2 we define the evaluation relation for ICω expressions, written $S, e \rightarrow S', e'$, which means that given a state S, expression e reduces by one or possibly more steps to e' and a (possibly updated) state S'. (We use an auxiliary function $value$ defined as follows: $value(f$ `=` $v) \stackrel{\text{def}}{=} v$, $value(v) \stackrel{\text{def}}{=} v$.) These rules are routine.

As is usual we have a number of cases that lead to a predicable error state, e.g. following a dereference of a `null` object. These errors in ICω are $CastX$, $ChoiceX$, $NullX$ and $NullableX$. We say that a pair S, e is *terminal* if e is one of these errors, or it is a value.

The evaluation relation for ICω promotable expressions is written $S, pe \rightarrow S', pe'$ and is also given in Fig. 2. The rules for method invocation deserve some explanation: they are differentiated according to whether the method is `void`-returning. If it is not then the method body is unfolded, and executed until it is of the form `return` v`;` where v is a value. This value is then the result of the method invocation. If the method is `void`-valued, then we unfold the method body and execute it until it is of the form `return;`. The result is the special value `void`.

The evaluation relation for statements is written $S, s \rightarrow S', s'$ and in Fig. 3 we give just some of the interesting cases, which are those dealing with `foreach` loops. As we have mentioned, Cω streams are aligned with C$\sharp$ 2.0 iterators: there the `foreach` loop is actually syntactic sugar: first of all an `IEnumerator<T>` is obtained from the iterator block (which should be of type `IEnumerable<T>`) using the `GetEnumerator` method. This is walked over using `MoveNext` and `Current` members. Semantically important is that `GetEnumerator` actually copies the enumerable object. In our semantics we faithfully encode this by tagging closures, and creating clones as appropriate. Thus whilst iterating over a stream we update the reference in place (rules [FVC], [FSC] and [FNC]), but every `foreach` creates its own copy from a fresh original (rules [FVF], [FSF] and [FNF]). In rule [FBr] we write α to range over both clone and fresh.

Expressions

$$\frac{}{(H,R), x \to (H,R), R(x)} \qquad \frac{H(r) = (\tau', cn) \quad \tau' <: \tau}{(H,R), \texttt{(}\tau\texttt{)}r \to (H,R), r}$$

$$\frac{H(r) = (\tau', cn) \quad \tau' \not<: \tau}{(H,R), \texttt{(}\tau\texttt{)}r \to (H,R), \mathit{CastX}} \qquad \frac{H(r) = (\tau', cn) \quad \tau' <: \tau}{(H,R), r\ \texttt{is}\ \tau \to (H,R), \texttt{true}}$$

$$\frac{H(r) = (\tau', cn) \quad \tau' \not<: \tau}{(H,R), r\ \texttt{is}\ \tau \to (H,R), \texttt{false}} \qquad \frac{}{S, \texttt{new (}\kappa, v\texttt{) was }\kappa \to S, \texttt{true}}$$

$$\frac{\kappa \neq \kappa'}{S, \texttt{new (}\kappa, v\texttt{) was }\kappa' \to S, \texttt{false}} \qquad \frac{r \notin dom(H)}{(H,R), \texttt{new}\ c\texttt{(}v\texttt{)} \to (H[r \mapsto (c,v)], R), r}$$

$$\frac{r \notin dom(H)}{(H,R), \texttt{new}\ \sigma\texttt{*/?(}\overline{s}\texttt{)} \to (H[r \mapsto (\sigma\texttt{*/?}, (R, \overline{s})^{\mathsf{fresh}})], R), r}$$

$$\frac{H(r) = (c, cn)}{(H,R), r\texttt{.content} \to (H,R), cn} \qquad \frac{}{S, \texttt{null.content} \to S, \mathit{NullX}}$$

$$\frac{0 \leq i \leq n}{S, \texttt{new \{}bv_0, .., bv_n\texttt{\}[}i\texttt{]} \to S, \mathit{value}(bv_i)} \qquad \frac{}{S, \texttt{new (}\kappa, v\texttt{) at }\kappa \to S, v} \qquad \frac{\kappa \neq \kappa'}{S, \texttt{new (}\kappa, v\texttt{) at }\kappa' \to S, \mathit{ChoiceX}}$$

Promotable expressions

$$\frac{}{(H,R), x\ \texttt{=}\ v \to (H, R[x \mapsto v]), v} \qquad \frac{(H,R), \overline{s} \to^* (H',R'), \texttt{return}\ v\texttt{;}\overline{s'}}{(H,R), \tau\texttt{(\{}\overline{s}\texttt{\})} \to (H',R'), v}$$

$$\frac{}{S, \texttt{null.}m\texttt{(}\overline{v}\texttt{)} \to S, \mathit{NullX}} \qquad \frac{\begin{array}{l}H(r) = (c, \_) \quad \mathit{method}(m,c) = \tau'(\overline{\tau}\ \overline{x})\{\overline{s}\} \quad \tau' \neq \texttt{void} \\ (H,[\,]), \texttt{\{}c\ \texttt{this}\ \texttt{=}\ r\texttt{;}\overline{\tau}\ \overline{x}\ \texttt{=}\ \overline{v}\texttt{;}\overline{s}\texttt{\}} \to^* (H',R'), \texttt{return}\ v'\texttt{;}\overline{s'}\end{array}}{(H,R), r\texttt{.}m\texttt{(}\overline{v}\texttt{)} \to (H',R), v'}$$

$$\frac{\begin{array}{l}H(r) = (c, \_) \quad \mathit{method}(m,c) = \texttt{void}\ (\overline{\tau}\ \overline{x})\{\overline{s}\} \\ (H,[\,]), \texttt{\{}c\ \texttt{this}\ \texttt{=}\ r\texttt{;}\overline{\tau}\ \overline{x}\ \texttt{=}\ \overline{v}\texttt{;}\overline{s}\texttt{\}} \to^* (H',R'), \texttt{return}\ \texttt{;}\overline{s'}\end{array}}{(H,R), r\texttt{.}m\texttt{(}\overline{v}\texttt{)} \to (H',R), \texttt{void}}$$

Fig. 2. Evaluation rules for ICω expressions and promotable expressions

$$\frac{}{S, \texttt{foreach (}\sigma\ x\ \texttt{in null)}\, s \to S, ;}\ \text{[FNull]}$$

$$\frac{H(r) = (\tau', (R', \overline{s'})^{\alpha}) \quad (H, R'), \overline{s'} \to^* (H', R''), \texttt{yield break ;}\overline{s''}}{(H, R), \texttt{foreach (}\sigma\ x\ \texttt{in}\ r\texttt{)}\ s \to (H', R), ;}\ \text{[FBr]}$$

$$\frac{\begin{array}{ll} H(r) = (\tau', (R', \overline{s'})^{\mathsf{fresh}}) & r' \notin dom(H') \\ (H, R'), \overline{s'} \to^* (H', R''), \texttt{yield return (}\sigma', v\texttt{);}\overline{s''} & v \neq \texttt{null} \end{array}}{\begin{array}{l}(H, R), \texttt{foreach (}\sigma\ x\ \texttt{in}\ r\texttt{)}\ s \to \\ (H'[r' \mapsto (\tau', (R'', \overline{s''})^{\mathsf{clone}})], R), \texttt{\{\{}\sigma\ x\ \texttt{=}\ v; s\texttt{\}}\ \texttt{foreach (}\sigma\ x\ \texttt{in}\ r'\texttt{)}\ s\texttt{\}}\end{array}}\ \text{[FVF]}$$

$$\frac{\begin{array}{l} H(r) = (\tau', (R', \overline{s'})^{\mathsf{clone}}) \\ (H, R'), \overline{s'} \to^* (H', R''), \texttt{yield return (}\sigma', v\texttt{);}\overline{s''} \quad v \neq \texttt{null} \end{array}}{\begin{array}{l}(H, R), \texttt{foreach (}\sigma\ x\ \texttt{in}\ r\texttt{)}\ s \to \\ (H'[r \mapsto (\tau', (R'', \overline{s''})^{\mathsf{clone}})], R), \texttt{\{\{}\sigma\ x\ \texttt{=}\ v; s\texttt{\}}\ \texttt{foreach (}\sigma\ x\ \texttt{in}\ r\texttt{)}\ s\texttt{\}}\end{array}}\ \text{[FVC]}$$

$$\frac{\begin{array}{ll} H(r) = (\tau', (R', \overline{s'})^{\mathsf{fresh}}) & r' \notin dom(H') \\ (H, R'), \overline{s'} \to^* (H', R''), \texttt{yield return (}\sigma'\texttt{*}, v\texttt{);}\overline{s''} & v \neq \texttt{null} \end{array}}{\begin{array}{l}(H, R), \texttt{foreach (}\sigma\ x\ \texttt{in}\ r\texttt{)}\ s \to \\ (H'[r' \mapsto (\tau', (R'', \overline{s''})^{\mathsf{clone}})], R), \texttt{\{foreach (}\sigma\ x\ \texttt{in}\ v\texttt{)}\ s\ \texttt{foreach (}\sigma\ x\ \texttt{in}\ r'\texttt{)}\ s\texttt{\}}\end{array}}\ \text{[FSF]}$$

$$\frac{\begin{array}{l} H(r) = (\tau', (R', \overline{s'})^{\mathsf{clone}}) \\ (H, R'), \overline{s'} \to^* (H', R''), \texttt{yield return (}\sigma'\texttt{*}, v\texttt{);}\overline{s''} \quad v \neq \texttt{null} \end{array}}{\begin{array}{l}(H, R), \texttt{foreach (}\sigma\ x\ \texttt{in}\ r\texttt{)}\ s \to \\ (H'[r \mapsto (\tau', (R'', \overline{s''})^{\mathsf{clone}})], R), \texttt{\{foreach (}\sigma\ x\ \texttt{in}\ v\texttt{)}\ s\ \texttt{foreach (}\sigma\ x\ \texttt{in}\ r\texttt{)}\ s\texttt{\}}\end{array}}\ \text{[FSC]}$$

$$\frac{\begin{array}{ll} H(r) = (\tau', (R', \overline{s'})^{\mathsf{fresh}}) & r' \notin dom(H') \\ (H, R'), \overline{s'} \to^* (H', R''), \texttt{yield return (}\tau, \texttt{null)}\texttt{;}\overline{s''} & \end{array}}{\begin{array}{l}(H, R), \texttt{foreach (}\sigma\ x\ \texttt{in}\ r\texttt{)}\ s \to \\ (H'[r' \mapsto (\tau', (R'', \overline{s''})^{\mathsf{clone}})], R), \texttt{foreach (}\sigma\ x\ \texttt{in}\ r'\texttt{)}\ s\end{array}}\ \text{[FNF]}$$

$$\frac{\begin{array}{l} H(r) = (\tau', (R', \overline{s'})^{\mathsf{clone}}) \\ (H, R'), \overline{s'} \to^* (H', R''), \texttt{yield return (}\tau, \texttt{null)}\texttt{;}\overline{s''} \end{array}}{(H, R), \texttt{foreach (}\sigma\ x\ \texttt{in}\ r\texttt{)}\ s \to (H'[r \mapsto (\tau', (R'', \overline{s''})^{\mathsf{clone}})], R), \texttt{foreach (}\sigma\ x\ \texttt{in}\ r\texttt{)}\ s}\ \text{[FNC]}$$

Fig. 3. Evaluation rules for ICω `foreach` loops

Rules [FSF] and [FSC] embody the flattening of streams. To evaluate a `foreach` loop we first evaluate the stream until it `yield`s a value. If that value is itself a stream, then we should first execute the `foreach` loop on this stream.

3.4 Compiling FCω to ICω

In this section we give some details of the compilation of FCω into ICω. Much of this compilation is routine, so in the interests of space we shall concentrate only on the most interesting aspect: generalized member access.

We employ a ''coercion" technique, in that we translate the *implicit* generalized member access of FCω into an *explicit* ICω code fragment. This can be expressed as an inductively defined relation, written $|\tau.f\colon\tau'| \rightsquigarrow g$ and $|\tau.m(\overline{\tau'})\colon\tau''| \rightsquigarrow g$ for member and function member access respectively. A judgement $|\tau.f\colon\tau'| \rightsquigarrow g$ is intended to mean that if invoking a member f on an element of type τ returns an element of type τ', then g is the ICω coercion that encodes the explicit access of the appropriate member. In Fig. 4 we give some details of the compilation of generalized member access (GMA) for members, i.e. the $|\tau.f\colon\tau'| \rightsquigarrow g$ relation. (The version for function members (methods) is similar and omitted.) In the definition we have employed a function-like syntax for coercions, although they are really contexts, and we have dropped the types from various block expressions. We have used the shorthand `yield return`$'$`(`τ`,e);` to mean the statement sequence `yield return(`τ`,e);yield break;` and have also used two functions: `Value` that returns the element of a singleton stream or raises an exception if it empty, and `HasValue` that returns a boolean depending on whether the singleton stream has an element or not. These can be coded directly and their definitions are omitted.

For example, we can compile an instance of member access in FCω, $e\,.f$, as follows: we first compile the expression e into ICω, yielding e', and also generate a coercion, g, corresponding to the member access. The result of the compilation of $e\,.f$ is then simply $g(e')$. We write the compilation of, e.g. an expression, e, as $|\Gamma \vdash e\colon\tau| \rightsquigarrow e'$.

Incoherence by design Java and C$\sharp$ are by design incoherent [7]. Both languages use a notion of ''best" conversion when there is more than one conversion between two types. If there does not exist a best conversion, a compile-time error is generated. In compiling FCω to ICω we use this notion of a best conversion when dealing with rules that use subtyping. We do not formalize this notion of ''best" here; both the Java and C$\sharp$ language specifications give precise details. The new types in Cω do not complicate this notion greatly: For example, there are two conversions between `int` and `object`: one using the rule [Box], the other using the rules [SubChoice] and [Box] along with [Trans] (i.e. `int` $<:$ `choice{int;string;}` $<:$ `object`). It is clear that the first conversion is better. The other critical pairs are similarly easy to resolve.

3.5 Properties of FCω and ICω

In this section we briefly mention some properties of FCω and ICω and the compilation. We do not give any details of the proofs, as they are standard and follow analogous theorems for Java [18, 5]; details will appear in a forthcoming technical report.

Compiling GMA over streams

$$\frac{|\sigma.f\colon\sigma'| \rightsquigarrow g}{|\sigma*.f\colon\sigma'*| \rightsquigarrow z \mapsto z.\texttt{\{}g(\texttt{it})\texttt{\}}}$$

Compiling GMA over anonymous structs

$$\frac{\exists S \subseteq \{1 \ldots n\}.|S| \geq 2. \wedge\, p = |S| \wedge \forall k \in [1..p].\ |fd_{S_k}.f : \tau_k| \rightsquigarrow g_k}{|\texttt{struct\{}fd_1\texttt{;}\ldots fd_n\texttt{;\}.}f\colon\texttt{struct\{}\tau_1\texttt{;}\ldots\tau_p\texttt{;\}}| \rightsquigarrow z \mapsto \texttt{new\{}g_1(z\texttt{[1]})\texttt{,}\ldots\texttt{,}g_p(z\texttt{[}p\texttt{]})\texttt{\}}}$$

$$\frac{\exists!k \in \{1 \ldots n\}.\ |fd_k.f : \tau_k| \rightsquigarrow g}{|\texttt{struct\{}fd_1\texttt{;}\ldots fd_n\texttt{;\}.}f\colon\tau_k| \rightsquigarrow z \mapsto g(z\texttt{[}k\texttt{]})}$$

Compiling GMA over choice types

$$\frac{\exists S \subseteq \{1 \ldots n\}.|S| \geq 2 \wedge p = |S| \wedge \forall k \in [1..p].\ |\kappa_{S_k}.f : \kappa'_k| \rightsquigarrow g_k}{\begin{array}{l}|\texttt{choice\{}\kappa_1\texttt{;}\ldots\kappa_n\texttt{;\}.}f : \texttt{choice\{}\kappa'_1\texttt{;}\ldots\kappa'_p\texttt{;\}?}| \\ \rightsquigarrow z \mapsto \\ \quad \texttt{(\{if(}z\ \texttt{was}\ \kappa_{S_1}\texttt{)} \\ \quad\ \texttt{\{return new choice\{}\kappa'_1\texttt{;}\ldots\kappa'_p\texttt{;\}?(yield return'(}\kappa_{S_1}\texttt{,new(}\kappa_{S_1}\texttt{,}g_1(z\ \texttt{at}\ \kappa_{S_1})\texttt{)))\texttt{;)\texttt{;\}}} \\ \quad \cdots\texttt{if(}z\ \texttt{was}\ \kappa_{S_p}\texttt{)} \\ \quad\ \texttt{\{return new choice\{}\kappa'_1\texttt{;}\ldots\kappa'_p\texttt{;\}?(yield return'(}\kappa_{S_p}\texttt{,new(}\kappa_{S_p}\texttt{,}g_p(z\ \texttt{at}\ \kappa_{S_p})\texttt{)))\texttt{;)\texttt{;\}}} \\ \quad \texttt{else return null;\})}\end{array}}$$

$$\frac{|\kappa_k.f : \tau| \rightsquigarrow g_k \quad \forall k.1 \leq k \leq n}{\begin{array}{l}|\texttt{choice\{}\kappa_1\texttt{;}\ldots\kappa_n\texttt{;\}.}f : \tau| \rightsquigarrow z \mapsto \texttt{(\{ if(}z\ \texttt{was}\ \kappa_1\texttt{) return}\ g_1(z\ \texttt{at}\ \kappa_1)\texttt{;}\cdots \\ \qquad\qquad\qquad\qquad \texttt{if(}z\ \texttt{was}\ \kappa_n\texttt{) return}\ g_n(z\ \texttt{at}\ \kappa_n)\texttt{;\})}\end{array}}$$

$$\frac{\exists!k \in \{1 \ldots n\}.\ |\kappa_k.f : \sigma| \rightsquigarrow g \quad n > 1}{\begin{array}{l}|\texttt{choice\{}\kappa_1\texttt{;}\ldots\kappa_n\texttt{;\}.}f : \sigma\texttt{?}| \\ \rightsquigarrow z \mapsto \texttt{(\{if(}z\ \texttt{was}\ \kappa_k\texttt{) return new}\ \sigma\texttt{?(yield return'(}\sigma\texttt{,}g(z\ \texttt{at}\ \kappa_k)\texttt{));} \\ \qquad\quad \texttt{else return null;\})}\end{array}}$$

Compiling GMA over singleton streams

$$\frac{|\sigma.f\colon\sigma'| \rightsquigarrow g}{\begin{array}{l}|\sigma\texttt{?}.f\colon\sigma'\texttt{?}| \rightsquigarrow z \mapsto \texttt{(\{if (HasValue(}z\texttt{)) return new}\ \sigma'\texttt{?(yield return'(}\sigma'\texttt{,}g(\texttt{Value(}z\texttt{))));} \\ \qquad\qquad\quad \texttt{else return null;\})}\end{array}}$$

$$\frac{|\sigma.f\colon\sigma'*/\texttt{?}| \rightsquigarrow g}{\begin{array}{l}|\sigma\texttt{?}.f\colon\sigma'*/\texttt{?}| \rightsquigarrow z \mapsto \texttt{(\{if (HasValue(}z\texttt{)) return}\ g(\texttt{Value(}z\texttt{));} \\ \qquad\qquad\qquad\quad \texttt{else return null;\})}\end{array}}$$

Fig. 4. Compilation of Generalized Member Access

Our main result is that ICω is type-sound, which is captured by the following properties. (We use generalized judgements, e.g. $\Gamma \rhd (S, e) : \tau$ to mean that the expression e is well-typed and also that the state S is well-formed with respect to Γ, in the familiar way. As is usual [18] we also need to add ''stupid" typing rules for the formal proof.)

Theorem 1 (Type soundness for ICω).

1. *If* $\Gamma \rhd (S, e) : \tau$ *and* $(S, e) \rightarrow (S', e')$ *then* $\exists \tau'$ *such that* $\Gamma \rhd (S', e') : \tau'$ *and* $\tau' <: \tau$.
2. *If* $\Gamma; \tau \rhd s$ *and* $(S, s) \rightarrow (S', s')$ *then* $\exists \tau'$ *such that* $\Gamma; \tau' \rhd (S', s')$ *and* $\tau' <: \tau$.
3. *If* $\Gamma \rhd (S, e) : \tau$ *then either* (S, e) *is terminal or* $\exists S', e'$ *such that* $(S, e) \rightarrow (S', e')$.
4. *If* $\Gamma; \tau \rhd (S, s)$ *then either* (S, s) *is terminal or* $\exists S', s'$ *such that* $(S, s) \rightarrow (S', s')$.

We can also prove that our compilation of FCω to ICω is type-preserving, i.e. if an FCω expression e in environment Γ has type τ , then there is a compilation of e resulting in an ICω expression e', such that e' in Γ also has type τ.

Theorem 2 (Type preservation of compilation).

1. *If* $\Gamma \vdash e : \tau$ *then* $\exists e'$ *such that* $|\Gamma \vdash e : \tau| \rightsquigarrow e'$ *and* $\Gamma \rhd e' : \tau$.
2. *If* $\Gamma; \tau \vdash s$ *then* $\exists s'$ *such that* $|\Gamma; \tau \vdash s| \rightsquigarrow s'$ *and* $\Gamma; \tau \rhd s'$.

4 Related work

Numerous languages have been proposed for manipulating relational and semi-structured data. For reasons of space we focus here only on those for semi-structured data (some of the languages for relational data were cited in §1).

A number of special-purpose functional languages [15, 4, 10] have been proposed for processing XML values. This stands in contrast to our approach, which aims at extending an existing widely-used object-oriented programming language.

The languages most similar to Cω are XJ [14] and Xtatic [13]. XJ adds XML and XPath as a first-class construct to Java, and uses logical XML classes to represent XSDs. In this way XJ allows compile time checking of XML fragments; however since the impedance mismatch between XML and objects is quite large, it does not deal with a mix of data from the the object and the XML world. One consequence is, for example, that XPath queries are restricted to work on XML data only.

Xtatic extends C$^\sharp$ with a separate category of regular expression types [16]. Subtyping is structural. While this gives a lot of flexibility this neither conforms with XML Schema, where subtyping is defined by name through restrictions and extensions, nor does it allow a free mix of objects and XML. Further, Xtatic uses pattern matching for XML projections, which fits well with the chosen type system but lacks first-class queries.

In contrast to XJ and Xtatic, Cω does not treat XML as a distinct and separate class. Its ingenuity lies in the uniform integration of the new stream, choice and struct types into the existing types and the generalization of member access— ''the power is in the dot". In fact, generalized member access in Cω achieves many of the benefits that

other type systems try to solve. For example, a long standing problem is how to write a query over data that comes from two sources that are similar, modulo some distribution rules, but not the same [8]. The type algebra of regular expression types often allows a factorization which makes this scenario possible. Generalized member access, on the other hand, handles this problem itself, without the need for distribution rules at the type level.

Another popular approach to deal with XML in an object-oriented language is by using so called data-bindings. A data-binding generates some strongly typed object representation from a given XML schema (XSD). JAXB for Java and xsd.exe in the .NET framework generate classes from a given XSD. However, it is often impossible to generate reasonable bindings, since the rich type system of XSDs cannot adequately be mapped onto classes and interfaces. As a consequence the resulting mappings are often weakly typed.

Cω takes a different but simpler view: XML is considered to be a serialization syntax for the rich type system of Cω. We are not tied to a particular XML data model. While Cω by design doesn't support the entirety of the full XML stack, in our experience Cω's type system and language extensions are rich enough to support realistic scenarios. We have written a large number of applications, including the complete set of XQuery Use Cases, several XSL stylesheets, and a substantial application (50KLOC) to manage TV listings.

5 Conclusions and future work

In this paper we have considered the problem of manipulating relational and semi-structured data within common object-oriented languages. We observed that existing methods using APIs provide poor support for these common application scenarios. Therefore, we have proposed a series of elegant extensions to C$^\sharp$ that provides type-safe, first-class access to, and querying of, these forms of data. We also have built a full compiler that implements our design. In this paper we have studied these extensions formally.

This work represents an industrial application of formal methods; on the whole, we found the process of formalizing our intuitions extremely useful, and indeed we managed to trap a number of subtle design flaws in the process. (In addition we had to formalize a fragment of C$^\sharp$, which was a little subtle in places. For example, we believe that this paper gives the first formal operational semantics for iterators.) That said, we also found it useful to be simultaneously developing a compiler. On a small number of occasions we found that our formalization was too high-level, in that it failed to capture some lower-level issues. Also whilst FCω is small enough to prove theorems about by hand, we should have liked to formalized a larger fragment of the language. At the moment, this seems unrealistic without more highly developed machine assistance.

One aspect of this project that we should like to consider further is the compilation. The Common Type System (CTS) for the Common Language Runtime (CLR) whilst general, lacks support for structural types. As our current compiler targets .NET 1.1, this means that the choice and anonymous structs types have to be ‘‘simulated”. In

future work, we plan to study extending the CLR with structural types. This would also enable more effective compilation of other languages that offer structural types, such as functional languages. It would also be interesting to study whether the lightweight covariance of Cω could be added to the CTS and other languages.

Implementation status A prototype Cω compiler is freely available. It covers the entire safe fragment of C$^\sharp$ and includes all the data access features described in this paper (and more) and also the ‘‘polyphonic" concurrency primitives [3]. (Available from `http://research.microsoft.com/comega`.)

References

1. A. Albano, G. Ghelli, and R. Orsini. Types for databases: the Galileo experience. In *Proceedings of DBPL*, 1989.
2. A. Albano, G. Ghelli, and R. Orsini. Fibonacci: A programming language for object databases. *Journal of Very Large Data Bases*, 4(3):403–444, 1995.
3. N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C$^\sharp$. *TOPLAS*, 26(5):769–804, 2004.
4. V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-centric general-purpose language. In *Proceedings of ICFP*, 2003.
5. G.M. Bierman, M.J. Parkinson, and A.M. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report 563, University of Cambridge, 2003.
6. G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to Java. In *Proceedings of OOPSLA*, 1998.
7. V. Breazu-Tannen, T. Coquand, C.A. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Information and computation*, 93(1):172–221, 1991.
8. P. Buneman and B.C. Pierce. Union types for semistructured data. In *Proceedings of IDPL*, 1998.
9. D. Chamberlin et al. XQuery use cases. `http://www.w3.org/TR/xquery-use-cases/`.
10. S. Boag et al. XQuery. `http://www.w3.org/TR/xquery`.
11. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proceedings of POPL*, 1998.
12. C. Fournét and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of POPL*, 1996.
13. V. Gapeyev and B.C. Pierce. Regular object types. In *Proceedings of ECOOP*, 2003.
14. M. Harren, M. Raghavachari, O. Shmueli, M. Burke, V. Sarkar, and R. Bordawekar. XJ: Integration of XML processing into Java. Technical report, IBM Research, 2003.
15. H. Hosoya and B.C. Pierce. XDuce: A typed XML processing language. In *Proceedings of WebDB*, 2000.
16. H. Hosoya, J. Vouillon, and B.C. Pierce. Regular expression types for XML. In *Proceedings of ICFP*, 2000.
17. M. Howard and D. LeBlanc. *Writing Secure Code*. Microsoft Press, 2003.
18. A. Igarashi, B.C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.
19. D. Leijen and E. Meijer. Domain Specific Embedded Compilers. In *Proceedings of Conference on Domain-Specific Languages*, 1999.
20. F. Matthes, S. Müßig, and J.W Schmidt. Persistent polymorphic programming in Tycoon: An introduction. Technical report, University of Glasgow, 1994.
21. E. Meijer, W. Schulte, and G.M. Bierman. Programming with circles, triangles and rectangles. In *Proceedings of XML*, 2003.

22. E. Meijer, W. Schulte, and G.M. Bierman. Unifying tables, objects and documents. In *Proceedings of DP-COOL,*, 2003.
23. B.C. Pierce. *Types and programming languages*. MIT Press, 2002.
24. J. Price. *Java programming with Oracle SQLJ*. O'Reilly, 2001.
25. D. Yu, A. Kennedy, and D. Syme. Formalization of generics for the .NET common language runtime. In *Proceedings of POPL*, 2004.