Unifying Tables, Objects and Documents

Erik Meijer and Wolfram Schulte

Microsoft Corporation

Abstract. This paper proposes a number of type-system and language extensions to natively support relational and hierarchical data within a statically typed object-oriented setting. In our approach SQL tables and XML documents become first class citizens that benefit from the full range of features available in a modern programming language like $C^\sharp$ or Java. This allows objects, tables and documents to be constructed, loaded, passed, transformed, updated, and queried in a unified and type-safe manner.

1 Introduction

The most important current open problem in programming language research is to increase programmers productivity, that is to make it easier and faster to write correct programs [38]. The integration of data access in mainstream programming languages is of particular importance — millions of programmers struggle with this every day. Data sources and sinks are typically XML documents and SQL tables, but they don't merge nicely into a statically typed object-oriented setting in which most production software is written.

This paper addresses how to integrate tables and documents into modern object-oriented languages by providing a novel type-system and corresponding language extensions.

1.1 The Need for a Unification

Distributed web-based applications are predominantly structured using a three-tier model that most commonly consists of a *middle tier* containing the business logic that extracts relational data from a *data services tier* and munches it into hierarchical data that is displayed in the *user interface tier*. The middle tier is often programmed in an object-oriented language such as Java or $C^\sharp$.

As a consequence, middle tier programs have to deal with relational data (SQL tables), object graphs, and hierarchical data (HTML, XML). Unfortunately these three different worlds are not very well integrated. As the following ADO.Net based example shows, access to a database in this style involves sending a string representation of a SQL query over an explicit connection via a stateful API and then iterating over a weakly typed representation of the result set:

```
SqlConnection Conn = new SqlConnection(...);
SqlCommand Cmd = new SqlCommand("SELECT Name,HP FROM Pokedex",Conn);
Conn.Open();
SqlDataReader Rdr = Cmd.ExecuteReader();
```

Creating HTML or XML documents is then done by emitting document fragments in string form, without separating the model and presentation:

```
while (Rdr.Read()) {
 Response.Write("<tr><td>");
 Response.Write(Rdr.GetInt32(0));
 Response.Write("</td><td>");
 Response.Write(Rdr.GetString(1));
 Response.Write("</td></tr>");
}
```

Communication between the different tiers using untyped strings is obviously very brittle with lots of opportunities for errors and zero probability for static checking. The cynical thing is that due to the poor integration, performance suffers badly as well.

The next code fragment rewrites the same functionality using a hypothetical language that unifies objects, tables and documents.

```
tr* pokemon =
  select <tr>
          <td>{Name}</td><td>{HP}</td>
         </tr>
  from Pokedex;

Table t =
 <table>
   <tr><th>Name</th><th>HP</th></tr>
   {pokemon}
 </table>;

Response.Write(t);
```

In this case, strongly typed XML values are first-class citizens (i.e. the XML literal `<table>...</table>` has type static `Table`) and SQL-style `select` queries are build-in. There is ample opportunity for static checking, and because the SQL and XML type-systems are integrated into the language, the compiler can do a better job in generating efficient code.

1.2 Growing a Language

It is easy to criticize the current lack of integration between tables, objects and documents, but it is much harder to come up with a design that gracefully unifies these separate worlds. No main-stream programming language has yet emerged that realizes this vision [7].

Often language integration only deals with SQL *or* with XML, but usually not with both [12,26,15,19,2,11,29]. Alternatively they start from a completely new language such as XQuery, or XDuce or CDuce [6,24,44,10]. Approaches based on language binding using some kind of pre-compiler such as XSD.exe, Castor, or JAXB [31,1] do not achieve a real semantic integration. The impedance mismatch between the different type-systems then leads to strange anomalies or unnatural mappings. Another popular route to integrate XML and SQL is by means of domain specific embedded languages [25] using functional language such as Scheme or Haskell [35,36,33,34,30,27,20,42,46,12] as the host. In our experience however, the embedded DSL approach does not scale very well, and it is particularly difficult to encode the domain specific type-systems [40] and syntax into the host language.

In his invited talk at OOPSLA98 [22], Guy Steele remarked that

> ... from now on, a main goal in designing a language should be to plan for growth. The language should start small, and the language must grow as the set of users grows.

This paper shows how to grow a modern object-oriented language (we take $C^\sharp$ as the host language, but the same approach will work with Java, Visual Basic, C++, etc.) to encompass the worlds of tables and documents by adding new types and expressions. In the remainder of this paper we will discuss:

Streams (Section 2) Streams are homogenous sequences of values of variable length. A database table consists of zero or more tuples; in the document world nodes can have zero or more sub-documents of the same kind, and in the object world we often work with (lazy) streams of values.

Tuples (Section 3) Tuples are heterogeneous sequences of values of fixed length. As we have just noticed, a database table is a stream of tuples; in the document world the `sequence` construct is used to model groups of sub-documents that must be present in a particular order, and finally several proposals have been made to extend Java and other object-oriented languages with tuples [43,28].

Unions (Section 4) Unions represent a choice between values of different type. They play a very important role in semi-structured documents [8] and many schemas use the `choice` construct to model alternatives. Union types also occur naturally in the result-types of queries.

Content Classes (Section 5) Content classes are ordinary classes whose members can be anonymous (unnamed). We use content classes to model top-level elements and complex types in document schemas.

Queries (Section 6) Finally we will extend our repertoire of accessors of our new types to match the expressive power of XPath and SQL queries. These accessors include implicit (homomorphic extension) and explicit (apply-to-all) mapping over streams, filtering, transitive member access, and relational select and join.

The growth of our experimental language is controlled by applying the following design principles:

Denotables values should be (easily) expressible If programmers can declare a variable of a certain type, it must be possible to write an expression of that type in a convenient way.

Expressible values should be denotable If programmmers can write an expression of a certain type, it must be possible to declare a variable whose static type precisely matches that of the expression.

No forced identity Programmers should never be forced to introduce either nominal identity of types, or object identity of values (aliasing).

Orthogonality There should be no special cases that discriminate between tables, documents and objects. Operations should work uniformly across the three worlds.

Flexibility The new types should have rich subtyping relationships that ease in writing type correct and evolvable software [9].

2 Streams

Streams are *generically typed refinements of iterators*, the pair of twin interfaces `IEnumerable` and `IEnumerator` in $C^\sharp$, or the corresponding `Iterator` interface in Java. Iterators encapsulate the logic for enumerating elements of collections.

The `foreach` loop of $C^\sharp$ makes it very convenient to *consume* values of type `IEnumerable` (future versions of Java will have a similar construct). For instance, since type `string` implements the `IEnumerable` interface, we can iterate over all the characters in a string using a simple `foreach` loop:

```
foreach(char c in s) Console.WriteLine(c);
```

The `foreach` loop in $C^\sharp$ is syntactic sugar for the following (simplified) `while` loop that calls into the `IEnumerable` and `IEnumerator` interfaces:

```
IEnumerator e = ((IEnumerable)s).GetEnumerator();
while (e.MoveNext()) { char c = (char)e.Current;
  Console.WriteLine(c);
}
```

While consuming an iterator is easy, it is much more difficult to write a generator that *implements* the `IEnumerable` (or the underlying `IEnumerator`) interface. In order to implement the `IEnumerable` interface on type `string` for instance, we have to manually create a state-machine that iterates over the individual characters in the string via `MoveNext` and exposes the current character via the `Current` property:

```
class string: IEnumerable {

  IEnumerator GetEnumerator() { return new Chars(this); }

  private class Chars : IEnumerator {
    private string s; private int i = 0; private char c;

    Chars(string s) { this.s = s; }

    public bool MoveNext() {
      if (i < s.Length) {
        c = s[i++]; return true;
      } else {
        return false;
      }
    }

    public char Current { get { return c; } }
  }
}
```

Note that this implementation does not correctly handle the extreme cases of calling `Current` before the first call to `GetNext` and calling it after `GetNext` has returned `false`.

In $C^\sharp$ and Java iterators are denotable, but not easily expressible. Moreover, the type `IEnumerable` is not very accurate since it does not convey the element type of the iterator. In other words, iterators of a particular type are expressible, but not precisely denotable.

We remedy both problems by introducing a new type of streams and a new statement to generate streams:

- The type T`*` denotes homogenous streams of arbitrary length with elements of type T. Type T`*` is a subtype of both `IEnumerable` and `IEnumerator`.
- Stream generators are like ordinary methods except that they may yield multiple values instead of returning a single time. The `yield` e statement returns the value of expression e into the `Current` property of its corresponding stream and suspends execution until `MoveNext` is called at which time execution resumes. Upon termination of the iterator `MoveNext` returns `false`.

Using streams and generators it becomes much simpler to enumerate all the characters in a string. The helper method `char* explode(string s)` generates the stream of the individual characters of string `s`. The `GetEnumerator` method of class `string` then simply explodes itself:

```
class string: IEnumerable {

  public IEnumerator GetEnumerator() { return this.explode(); }

  private char* explode() {
    int e = this.Length; for(int i = 0; i < e; i++) yield s[i];
  };
}
```

In this case maintaining the state is implicit in the control-flow of the `explode` function and in particular the borderline cases are handled correctly by definition.

Streams and generators are not new concepts. They are supported by a wide range of languages in various forms [21,5,39,28,32,37], and in particular future versions of $C^\sharp$ will also support iterators. Our approach is a little different in that:

- We classify streams into a hierarchy of streams of different length (`!`, `?`, `+`, `*`, see below).
- We automatically flatten streams of streams (see Section 2.2).
- Our streams are covariant (see below).
- We identify the value `null` with the empty stream (see Section 2.1).

To keep type-checking tractable, we restrict ourselves to the following four stream types: T`*` denotes possibly empty and unbounded streams with elements of type T, T`+` denotes non-empty possibly unbounded streams with elements of type T, T`?` denotes streams of at most one element of type T, and T`!` denotes streams with exactly one element of type T. We will use T`?` to represent optional values, where the nonexistence is represented by the value `null` and analogously we use T`!` to represent non-null values.

The different stream types form a natural subtype hierarchy, where subtyping corresponds to stream inclusion:

$$\begin{aligned} T\texttt{!} &<: T\texttt{+} \\ T\texttt{+} &<: T\texttt{*} \\ T\texttt{?} &<: T\texttt{*} \end{aligned}$$

For instance the subtype relation $T\texttt{!} <: T\texttt{+}$ reflects the fact that a stream of exactly one element is also a stream of at least one element.

We embed non-stream types T into the hierarchy by placing them between non-null values $T\texttt{!}$ and possibly null values $T\texttt{?}$:

$$\begin{array}{l} T\texttt{!} <: T \\ T <: T\texttt{?} \end{array}$$

This inclusion allows programmers to precisely state their intentions with respect to `null` values: $T\texttt{!}$ means `null` is not allowed, $T\texttt{?}$ means `null` is expected, and T means `null` is exceptional.

The next two rules reflect the facts that `null` (we use $\emptyset\texttt{?}$ for the null-type) is a possible value of any reference type, but that value types are never `null`:

$$\begin{array}{lll} \emptyset\texttt{?} & <: T, & T \text{ is a reference type} \\ T & <: T\texttt{!}, & T \text{ is a value type} \end{array}$$

Like arrays, streams are covariant. This means that subtyping on the element types is lifted to subtyping on streams. The special case for the `null` type says that possibly-null values can be `null`:

$$\frac{S <: T}{S\texttt{*} <: T\texttt{*}}$$

$$\emptyset\texttt{?} <: T\texttt{?}$$

Let `Button` be a subtype of `Control`, then the first rule says that `Button*` is a subtype of a stream of controls `Control*`. The second rule says for instance that `null` can be assigned to a variable of type `int?`.

2.1 Nullness

The type $T\texttt{!}$ denotes streams with exactly one element, and since we identify `null` with the empty stream, this implies that values of type $T\texttt{!}$ can never be `null`. Dually, the type $T\texttt{?}$ denotes streams with either zero (that is `null`) or exactly one element.

Values of type $T\texttt{?}$ model the explicit notion of nullability as found in SQL by providing a standard implementation of the `null` design pattern [23]; when a receiver of type $T\texttt{?}$ is `null`, accessing any of its members returns `null` instead of throwing an exception as in $C^\sharp$ or Java:

```
string? t = null;
int? n = t.Length; // n = null
```

In Objective-C [3] this is the standard behavior for any receiver object that can be `null`. In section 6.1 we show how member-access is lifted over streams in general.

Being able to express that a value *cannot* be `null` via the type system allows static checking for `null` pointers (see [16,18] for more examples). This turns many (potentially unhandled) dynamic errors into compile-time errors.

One of the several methods in the .NET base class library that throws an `ArgumentNullException` when its argument is `null` is the `IPAddress.Parse` function. Consequently, the implementation of `IPAddress.Parse` needs an explicit `null` check:

```
public static IPAddress Parse(string ipString) {
  if (ipString == null)
    throw new ArgumentNullException("ipString");
  ...
}
```

Dually, clients of `IPAddress.Parse` must be prepared to catch and deal with a possible `ArgumentNullException`. Nothing of this is apparent in the type of the `Parse` method in $C^\sharp$. In Java at least the signature of `Parse` would show that it possibly throws an exception.

It would be much cleaner if the type of `IPAddress.Parse` indicated that it expects its `string` argument to be non-`null`:

```
public static IPAddress Parse(string! a);
```

Now, the type-checker statically rejects any attempt to pass a string that might be `null` to `IPAddress.Parse`.

The proof obligation for returning a non-null stream T`!` or T`+` is similar to proving the definite assignment rule in $C^\sharp$ or Java. For statement blocks that return or yield non-empty streams, each non-exceptional execution path should return or yield at least one non-null value. The type-checker will therefore accept the first definition of `FromTo` but will reject the second:

```
int+ FromTo(int s, int d, int e) {
  yield s; while(s <= e) yield s += d;
}

// Type error
int+ FromTo(int s, int d, int e) {
  while(s <= e){ yield s; s += d; }
}
```

Non-empty streams T`+` are implicitly convertible to possibly empty streams T`*`; we can forget the fact that a stream has at least one element. It is in general *not* safe to downcast from a possibly empty stream T`*` to a non-empty stream T`+`. At first sight we might think that testing if the stream contains at least one non-null value would suffice. Alas this is not true. By cunningly using side-effects, the generator function `OnlyOnce()` only yields `4711` the first time it is evaluated and every subsequent evaluation produces an empty stream:

```
bool Done = false;
int* OnlyOnce() {
  if(!Done){ Done = true; yield 4711; }
};
int+ xs = (int+)OnlyOnce();   // 1. cast succeeds
int+ xs = (int+)OnlyOnce();   // 2. cast fails
```

To prevent such loopholes, down casting from T`*` to T`+` will only succeed if the dynamic type of the underlying stream is T`+`.

2.2 Flattening

We have to be very careful to ensure that every value in a (nested) stream is yielded at most once, otherwise we might end up with a quadratic instead of a linear number of yields when generating certain (recursive) streams [45]. For instance this happens if a nested stream like `[[...[[[],0],1],...],n-1]` gets recursively flattened into the non-nested stream `[0,1,...,n-1]` as in the next example:

```
// Iota(n) generates the stream [0,1,..,n-1]
int* Iota(int n){
  if(n>0){
    foreach(int i in Iota(--n)) yield i;
    yield n;
  }
}
```

Note that we are forced to flatten the stream produced by the recursive invocation of `Iota(n)` to generate a stream of the required type `int*`. Apart from these typing issues, there is absolutely no reason that the actual instance of a nested stream should be flattened since we can easily iterate over the leaf elements (the *yield*) of a nested stream.

So all that is required to type-check generators of nested streams is to flatten the *type* of a stream, which again does not imply that the underlying implementation of streams gets flattened as well. Table 2.2 gives the general flattening rules for nested streams T^{ij} of all possible combinations of stream constructors: The rule T`*+` $=$ T`+`, for instance, reflects the fact that a non-empty stream of possibly empty stream flattens into a non-empty stream, while T`+*` $=$ T`*` reflects that a possibly empty stream of non-empty streams flattens to a possibly empty stream.

T^{ij}	j = !	?	+	*
i = !	!	?	+	*
?	!	?	+	*
+	+	*	+	*
*	+	*	+	*

Fig. 1. Flattening rules for streams

Using the flattening rules, we can now write a linear time version of the `Iota` function that returns a nested stream of streams of type `int*`:

```
// Iota(n) generates the stream [[...[[[],0],1],...],n-1]
int* Iota(int n){
  if(n>0){
    yield Iota(--n); yield n;
  }
}
```

3 Tuples

Tuples are *heterogeneous* sequences of *optionally labelled* values of *fixed* length. Another way of viewing tuples is as anonymous structs whose members are ordered, in particular tuples have no object identity.

The function `DivMod` returns the quotient and remainder of its arguments as a tuple that contains two named integer fields `sequence{int Div, Mod;}`:

```
sequence{int Div, Mod;} DivMod(int x, int y) {
  return new(Div = x/y, Mod = x%y);
}
```

The members of a tuple do not need to be labelled, for example, we can create a tuple consisting of a labelled `Button` and an unlabelled `string` as follows:

```
sequence{Button b; string;} x = new(b=new Button(), "OK");
```

An unlabelled member of a *nominal* type is a shorthand for the same member implicitly labelled with its type.

Tuples can be picked apart constant indexers, `DivMod(47,11)[0]` for instance selects `47`, or by named member access, provided of course that tuple has a member m, for instance `x.b`.

3.1 Subtyping

Like streams, tuples are subject to a rich subtype hierarchy. The first subtype relation for tuples formalizes the fact that labels are optional and that we can forget them by upcasting:

$$\texttt{sequence}\{\ldots;\ T\ m\,;\ldots\} \ <:\ \texttt{sequence}\{\ldots;\ T\,;\ \ldots\}$$

Using this rule we see that we can assign `DivMod(47,11)` to a an unlabelled pair of integers of type `sequence{int; int;}`.

We can forget the ordering, nesting, and labels of a tuple by upcasting a tuple to a stream. The special cases give tighter types for the empty tuple (which gets converted to the empty stream `null`) and singleton tuple (which gets converted to its underlying value):

$$\begin{array}{l}\texttt{sequence}\{\} \ <:\ \emptyset\texttt{?}\\ \texttt{sequence}\{T\} \ <:\ T\\ \texttt{sequence}\{\ldots;\ T\,;\ \ldots\} \ <:\ \texttt{choice}\{\ldots; T\ \ ;\ldots\}\texttt{*}\end{array}$$

Using the last conversion, we can enumerate the values of any tuple as a stream, i.e. the tuple `new(4711, true, 'z', 3.14)` can be converted into the stream `[4711, true, 'z', 3.14]` of type `choice{int; bool; char; float}*`.

3.2 Non-Nullness for tuples

Even though tuples have no object identity, the fact that they are convertible to streams makes them subtly different from nominal value types.

Suppose that we would add the rule that tuples are not `null`, i.e., $\texttt{sequence}\{\ldots\}$ $<:$ $\texttt{sequence}\{\ldots\}\texttt{!}$. Then by applying this rule in combination with the singleton rule $\texttt{sequence}\{T\}$ $<:$ T we could assign the value `null` to a variable of non-null type `Button!`:

```
// Type error
sequence{Button;}  a = new(null);
sequence{Button;}! b = a;
Button!            c = b; // c = null
```

To maintain type-soundness soundness we have a weaker rule that states that a tuple is non-null if it has at least one member that is non-null. This guarantees that when the tuple is converted to a stream the resulting stream has the right cardinality. For singleton sequences the conversion also holds in the reverse direction:

$$\begin{array}{l}\texttt{sequence}\{\ldots;\ T\texttt{!}\ m\,;\ \ldots\} \ <:\ \texttt{sequence}\{\ldots;\ T\ m\,;\ \ldots\}\texttt{!}\\ \texttt{sequence}\{T\ m\,;\}\texttt{!} \ <:\ \texttt{sequence}\{T\texttt{!}\ m\,;\}\end{array}$$

By applying this rule in combination with the fact that `int <: int!`, we can show that the sequence of integers `new(1)` is convertible into a non-empty stream of type `sequence{int ; } <: sequence{int! ; } <: sequence{int ; }! <: int*! <: int+`.

3.3 Streams+Tuples = Tables

Relational data is stored in tables, which are sets of tuples. Sets can be represented by streams, thus streams and tuples together can be used to model relational data.

The table below contains some basic facts about Pokemon characters such as their name, their strength, their kind, and the Pokemon from which they evolved (see `http://www.pokemon.com/pokedex/` for more details about these interesting creatures).

Name	**HP**	**Kind**	**Evolved**
Meowth	50	Normal	
Rapidash	70	Fire	Ponyta
Charmelon	80	Fire	Charmander
Zubat	40	Plant	
Poliwag	40	Water	
Weepinbell	70	Plant	Bellsprout
Ponyta	40	Fire	

Each row in this table is a value of type `Pokemon` and the table itself is modelled as a variable `Pokedex` of type `Pokemon*`. The keyword `type` identifies the name on the left with the type expression on the right. It is just an abbreviation mechanism.

```
enum Kind {Water, Fire, Plant, Normal, Rock}

type Pokemon = sequence{
  string Name; int HP; Kind Kind; string? Evolved;
}

Pokemon* Pokedex;
```

The fact that basic Pokemon are not evolutions of other Pokemon shows up in that the `Evolved` column has type `string?`.

Representing tables is necessary for the integration of relational data, but it is not sufficient: we also have to provide operations that work on tables. We will introduce such query expressions in Section 6.3.

4 Unions

Union types often appear in content classes (see section 5 below). The type `Address` uses a union type `choice{ string Street; int POBox; }` to allow either a member `Street` of type `string` or a member `POBox` of type `int` as part of an `Address`:

```
class Address {
  sequence{
    choice{ string Street; int POBox; };
    string City; string? State; int Zip;
    string Country;
  };
}
```

The second situation in which union types are used is in the result types of generalized member access (see Section 6). For example, when variable `p` has type `Pokemon`, the expression `p.*` returns a stream containing all the members of a `Pokemon` instance which has type `choice{string; int; Kind; string?}*`. Using the subtype rules for `choice` and streams given below, we can show that this is isomorphic to `choice{string; int; Kind;}*`.

We can inject any type T into a union containing that type; singleton labelled tuples are injected into labelled unions:

$$T \ <: \texttt{choice}\{T\,;\ \ldots\}$$
$$\texttt{sequence}\{T\ m\}\ <: \texttt{choice}\{T\ m\,;\ \ldots\}$$

Except for boxing, $\texttt{choice}\{\ldots\} <: \texttt{object}$, there is no implicit elimination rule for union types. In other words, $\texttt{choice}\{T\,; S\}$ is an upperbound for S and T, but not a least upperbound. The reason is that we do not consider `Control` and `choice{Button; Control;}` to be isomorphic, which would be the case with a least upperbound interpretation.

Choice types are idempotent (duplicates are removed), and associative and commutative (nesting and order of members are ignored):

$$\texttt{choice}\{\ldots; F\,;\ F\,;\ \ldots\} \ = \ \texttt{choice}\{\ \ldots;\ F\,;\ \ldots\}$$
$$\texttt{choice}\{\ldots;\ \texttt{choice}\{\ldots\}\,;\ \ldots\} \ = \ \texttt{choice}\{\ldots;\ \ldots;\ \ldots\}$$
$$\texttt{choice}\{\ldots;\ F\,;\ G\,;\ \ldots\} \ = \ \texttt{choice}\{\ldots;\ G\,;\ F\,;\ \ldots\}$$

Streams distribute over unions. Non-nullness and possibly nullness distribute in both ways, and any inner streams gets absorbed by an outer + or *:

$$\begin{aligned}
\texttt{choice}\{\ldots;T\,;\ldots\}\texttt{!} &= \texttt{choice}\{\ldots;T\texttt{!}\ ;\ldots\}\texttt{!}\\
\texttt{choice}\{\ldots;T\,;\ldots\}\texttt{?} &= \texttt{choice}\{\ldots;T\texttt{?}\ ;\ldots\}\texttt{?}\\
\texttt{choice}\{\ldots;T^i\,;\ \ldots\}\texttt{+} &= \texttt{choice}\{\ldots;\ T\,;\ldots\}\texttt{+}\\
\texttt{choice}\{\ldots;T^i\,;\ \ldots\}\texttt{*} &= \texttt{choice}\{\ldots;\ T\,;\ldots\}\texttt{*}
\end{aligned}$$

where i is any stream functor.

The flattening and distribution rules allow us to normalize streams of choices: inner stream functors can either be eliminated completely or can be moved out of the choice.

5 Content Classes, XSDs and XML

Now that we have introduced streams, tuples, and unions, our type system is rich enough to model a large part of the XSD schema language [17]; our aim is to cover as much of the essence of XSD [41] as possible while avoiding most of its complexity.

The correspondence between XSD particles such as `<sequence>` and `<choice>` with local element declarations and the type constructors `sequence` and `choice` with (labelled) fields should be intuitively clear. Likewise, the relationship of XSD particles with occurrence constraints to streams is unmistakable. For T`*` the attribute pair `(minOccurs, maxOccurs)` is `(0, unbounded)`, for T`+` it is `(1, unbounded)`, for T`?` it is `(0, 1)`, and for T`!` it is `(1,1)`.

The content class `Address` that we defined in Section 4 corresponds to the following XSD schema `Address`:

```
<element name="Address">
  <complexType>
    <sequence>
      <choice>
        <element name="Street" type="string">
        <element name="POBox" type="integer">
      </choice>
      <element name="City" type="string">
      <element name="State" type="string" minOccurs="0"/>
      <element name="Zip" type="integer"/>
      <element name="Country" type="string"/>
    </sequence>
  </complexType>
</element>
```

The only difference between a content class and a normal $C^\sharp$ class is the fact that the members of content class can be unlabelled (just like the members of tuples and unions). As a consequence, unlabelled content can only ever be accessed via its individually named children, which allows the compiler to choose the most efficient data layout.

The next example schema defines two top level elements `Author` and `Book` where `Book` elements can have zero or more `Author` members:

```
<element name="Author">
  <complexType>
    <sequence>
      <element name="Name" type="string"/>
    </sequence>
  </complexType>
</element>

<element name="Book">
  <complexType>
    <sequence>
      <element name="Title" type="string"/>
      <element ref="Author" minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
  </complexType>
</element>
```

In this case, the local element reference is modelled by an unlabelled field and the schema is mapped onto the following two content type declarations:

```
class Author { string Name; }
class Book { sequence{ string Title; Author*; } }
```

All groups such as the one used in the following schema for the complex type `Name`

```
<complexType name="Name">
  <all>
    <element name="First" type="string"/>
    <element name="Last" type="string"/>
  </all>
</complexType>
```

are mapped to ordinary fields of the containing type, i.e. without a `sequence`:

```
class Name { string First; string Last; }
```

As these examples show, both top-level element declarations and named complex type declarations are mapped to top-level types. This allows us to unify derivation of complex types and substitution groups of elements using standard inheritance.

5.1 XML Literals

XML literals are an intuitive way to construct instances of content classes by making XML serialization into a first class language construct. For example, we can define an `Address` instance by directly assigning an XML document that confirms to the schema for `Address` as follows:

```
Address Microsoft =
  <Address>
    <Street>One Microsoft Way</Street>
    <City>Redmond</City><Zip>98052</Zip>
    <Country>USA</Country>
  </Address>;
```

XML literals can also have placeholders to describe *dynamic* content (similar to anti-quoting as found in Lisp and other languages). We use the XQuery [6] convention whereby an arbitrary expression or statement block can be embedded inside an element by escaping it with curly braces:

```
Author NewAuthor(string name) {
  return <Author>{name.ToUpper()}</Author>;
}
```

Embedded expressions must return or yield values of the required type (in this case `string`). Validation of XML literals with placeholders is non-trivial and is the subject of a forthcoming paper.

XML literals are just object constructors, there is nothing special about content classes. Hence we can write XML literals to construct values of *any* type, for example, the next assignment creates an instance of the standard `Button` class and sets its `Text` field to the string `"Click Me"`:

```
Button b = <Button>
             <Text>Click Me</Text>
           </Button>;
```

6 Generalized Member Access

In the previous sections we have concentrated on the type-system extensions to our hypothetical programming language. This section extends our repertoire of *expressions* to transform and query values of these new types.

6.1 Map, Filter, Fold

To make the creation of streams as concise as possible, we allow statement blocks (anonymous method bodies) as expressions. In the example below we assign the (lazy) infinite stream of positive integers to the variable `nats` by using an anonymous method body as an expression:

```
// block expression that yields the stream [0,1,2,...]
int* nats = { int i=0; while(true) yield i++; };
```

Our stream constructors (`*`, `+`, `?`, `!`) are functors, and hence we *implicitely lift member access on the element type of a stream over the stream itself*. For instance, to convert each individual string in a stream `Ss` of strings to uppercase, we can simple write `ss.ToUpper()`:

```
string* Ss = { yield "Hello"; yield "World!"; };
string* SS = Ss.ToUpper();
```

If both the stream and its elements have the same member no lifting takes place, and member access on the whole stream is the best match. For example, since `GetType()` is defined for both `string` and `string*`, the expression `Ss.GetType()` will return the dynamic type of the stream `Ss`.

If we nevertheless want to lift member access over a stream, we can use an *apply-to-all* block. For example, to get all the dynamic types of the elements of a stream we write `Ss.{ return it.GetType(); }`. The implicit argument `it` inside the apply-to-all block plays a similar role as the implicit argument `this` for methods and refers successively to each element of the stream `nats`.

As the next example shows, the apply-to-all block itself can yield a stream, in which case the resulting nested stream is flattened according to the rules of table 2.2:

```
// self-counting numbers: 1, 2,2, 3,3,3, 4,4,4,4, ...
int* rs = nats.{ for(i=1; i<it; i++) yield it; };
```

If an *apply-to-all* block returns `void`, no new stream is constructed and the block is eagerly applied to all elements of the stream. For example to print all the elements of a stream we can just map `Console.WriteLine` over each element:

```
nats.{ Console.WriteLine(it); };
```

Apply-to-all blocks can be stateful, so we can use them to do reductions or folds. For example, we can sum all integers in an integer stream `xs` by adding each element of the stream to a local variable `s`:

```
int sum(int* xs){
  int s = 0;
  xs.{ s += it; return; };
  return s;
}
```

Note that we need the `return` statement inside the block to ensure that the return type of the block is `void` such that the iteration is performed eagerly.

Often we want to filter a stream according to some predicate on the elements of the stream. For example, to construct a stream with only odd numbers, we filter out all even numbers from the stream `nats` of natural numbers using the filter expression

```
int* odds1 = nats[it%2 == 1];
```

For each element in the stream to be filtered, the predicate is evaluated with that element as `it`. Only if the predicate is true the element becomes part of the new stream.

On closer inspection, we realize that filters are just abbreviations of an apply-to-all-block:

```
int* odds2 = nats.{if (it%2 == 1) return it;};
```

Hence `odds1` and `odds2` denote streams that both have the same elements in the same order.

Lifting over non-null types is different from lifting over the other stream types, since the fact that the receiver object is not `null` does not imply that its members are not `null` either. For example when we create a new non-null `Button` instance using the default constructor, it's `Parent` field will definitively be null:

```
Button! b = <Button/>;
Control p = b.Parent; // Parent is null
```

Hence the return type of lifting over a non-null type is not guaranteed to return a non-null type.

The table 6.1 show how lifting of member-access interacts with streams types. Let T^j be a stream type, and m of type S^i be a member of the element type T that we want to lift over the stream . The result type of lifting m is then given by $s^{i \oplus j}$ (here _ denotes a non stream type):

$\oplus$	j=_	!	?	+	*
i=_		?	?	*	*
!	!	!	?	*	*
?	?	?	?	*	*
+	+	+	*	+	*
*	*	*	*	*	*

Fig. 2. Lifting over streams

Member access is not only lifted over streams, but over all structural types. For example the expression `xs.x` will return the stream `true, 1, 2` of union type `choice{bool; int;}+` when `xs` is defined as:

```
sequence{ bool x; sequence{ int x; }*; } xs =
  new( x=true, { yield new(x=1); yield new(x=2); } );
```

Lifting over union types introduces a possibility of nullness for members that are not in all of the alternatives.

Suppose `x` has type `choice{ int; string; }`. Since only `string` has a `Length` member, the type of `x.Length` is `int?` which reflects the fact that in case the dynamic type of `x` is `int`, the result of `x.Length` will be `null`. Since `int` and `string` both have a member `GetType()`, the return type of `x.GetType()` is `Type`:

```
choice{ int; string; } x = 4711;
int? n = x.Length;    // null
Type t = x.GetType(); // System.Int32
```

In case the alternatives of a union have a member of different type in common, we require a downcast before doing the member access.

6.2 Wildcard, Transitive and Type-based Member-access

The only query form available in object-oriented languages is member access. But that is rather restrictive. To allow for more flexible forms of member access, we provide *wildcard*, *transitive* and *type-based* access. These forms are similar to the concepts of nametest, abbreviated relative location paths and name filters in XPath [14]. However we adapted them to work uniformly on object graphs.

Wildcards allow to access all accessible members of a type without having to know their names. Suppose that we want to have all fields of an `Address`, then we can write:

```
choice{string; int;}* addressfields = Microsoft.*;
```

The wild-card expression returns the content of all accessable fields and properties of the variable `Microsoft` in their declaration order. In this case the stream of strings `"One Microsoft Way"`, `"Redmond"`, `98052`, `"USA"`.

Transitive member-access, written as `e...m`, returns all accessible members *m* that are transitively reachable from `e` in depth-first order. The following declaration of `authors` (lazily) returns a stream containing all `Author` of all `Books` in the source stream `books`:

```
Book F = <Book>
           <Title>Faust</Title><Author>Goethe</Author>
         </Book>;
```

```
Book K = <Book>
           <Title>Max Havelaar</Title><Author>Multatuli</Author>
          </Book>;

Book* books = { yield F; yield K; };
string* authors = books...Author;
```

Transitive member access allows to abstract from the concrete representation of a document; as long as the mentioned member is reachable and accessible, its values are returned.

Looking for just a field name is often not sufficient, especially for transitive queries where there might be several reachable members with the same name but of different type. In that case we can add an additional type-test to restrict the matching members. A type-test on T selects only those members whose static type is a subtype of T. For instance, if we are only interested in Microsoft's `POBox` number, and `Zip` code, we can write the transitive query `Microsoft...int::*`.

Note that type based access is also useful for unnamed members, since even if they have no name, they do have a static type.

6.3 Select and Join

The previous sections presented our solutions to querying documents. However for accessing relational data, which we have modelled as streams of tuples, simpler SQL queries are sufficient. Here we only show the integration of the SQL `select-from-where` clause, and defer the discussion of more advanced features such as data manipulation and transactions to a future paper.

The fundamental operations of relational algebra are *selection*, *projection*, *union*, *difference* and *join*. Selection is similar to filter and transforms one stream of tuples into another stream of tuples. Here are two variations of selection:

```
Pokemon* normalPokemons1 =
  select *
  from Pokedex
  where Kind == Normal;
Pokemon* normalPokemons2 =
  select it
  from (Pokemon it in Pokedex)
  where it.Kind == Normal;
```

The first example uses the familiar SQL syntax. Its meaning is provided by the second form, which uses the explicit iterator variable `it` as we have seen before.

We use similar sugar to introduce names for projection. Projection produces a stream of tuples by selecting only certain columns in its input stream:

```
sequence{string Name; Kind Kind;}*
  pokemonAbstract1 =
    select Name, Kind
    from Pokedex;
sequence{string Name; Kind Kind;}*
  pokemonAbstract2 =
    select new(Name= it.Name, Kind=it.Kind)
    from (Pokemon it in Pokedex);
```

Again, the first declaration shows the traditional SQL syntax, where the second shows the unsugared representation, which explicitly builds the resulting tuple by projecting the required members.

In practice, the result types of SQL queries can be quite involved and hence it becomes painful for programmers to explicitly specify types. Since the compiler already knows the types of sub-expressions, the result types of queries can be inferred automatically. Providing type declarations for method local variables is not necessary, and we can simply write:

```
pokemonAbstract3 = select Name, Kind from Pokedex;
```

without having to declare the type of `pokemonAbstract3`.

Union and difference present no difficulty in our framework. They can easily be handled with existing operations on streams. Union concatenates two streams into a single stream. Difference takes two streams, and returns a new stream that contains all values that appear in the first but not in the second stream.

The real power of `select-from-where` comes from join. Join takes two input streams and creates a third stream whose values are composed by combining members from the two input streams. For example, here is an expression that selects pairs of Pokemeons which have evolved from each other:

```
select p.Name, q.Name
from p in Pokedex, q in Pokedex
where p.Evolved == q
```

Again, we would like to stress the fact that everything fits together. The select expression works on arbitrary streams, whether in memory or on the hard disk; streams simply virtualize data access. Strong typing makes data access secure. But there is no burden for the programmer since the result types of queries are inferred.

7 Conclusion

The language extensions proposed in this paper support both the SQL [4] and the XML schema type system [41] to a large degree, but we have not dealt with all of the SQL features such as (unique) keys, and the more esoteric XSD features such as redefine. Similarly, we already covered much of the expressive power of XPath [14], XQuery [6] and XSLT[13], but we do not support the full set of XPath axis. We are able to deal smoothly with namespaces, attributes, blocking, and facets however. Currently we are investigating whether and which additional features need to be added to our language.

Summarizing, we have shown that it is possible to have both SQL tables and XML documents as first order citizen in an object-oriented language. Only a bridge between the type worlds is needed. Building the bridge is mainly an engineering task. But once it is available, it offers the best of three worlds

8 Acknowledgments

We would like to acknowledge the support, encouragement, and feedback from Mike Barnett, Nick Benton, Don Box, Luca Cardelli, Bill Gates, Steve Lucco, Chris Lucas, Todd Proebstring, Dave Reed, Clemens Szyperksi, and Rostislav Yavorskiy and the hard work of the WebData languages team consisting of William Adams, Joyce Chen, Kirill Gavrylyuk, David Hicks, Steve Lindeman, Chris Lovett, Frank Mantek, Wolfgang Manousek, Neetu Rajpal, Herman Venter, and Matt Warren.

References

1. The castor project. http://castor.exolab.org/.
2. The navision x++ programming language.
3. The Objective-C Programming Language.
4. G. Bierman and A. Trigoni. Towards a Formal Type System for ODMG OQL. Technical Report 497, University of Cambridge, Computer Laboratory, 2000.
5. B.Liskov, R.Atkinson, T. Bloom, E. Moss, J.C.Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual*. LNCS 114. Springer=Verlag, 1981.
6. S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language. Technical report, W3C, November 2002.
7. T. Bray. XML Is Too Hard For Programmers. http://www.tbray.org/ongoing.
8. P. Buneman and B. Pierce. Union Types for Semistructured Data. In *International Database Programming Languages Workshop*, LNCS 1949, 2000.
9. L. Cardelli. Types for data-oriented languages. In *EDBT*, 1988.
10. L. Cardelli, P. Gardner, and G. Ghelli. Manipulating trees with hidden labels. In *FOSSACS*, 2003.
11. R. G. G. Cattell, D. K. Barry, R. Catell, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, and F. Velez. *The Object Data Standard ODMG 3.0*. Morgan Kaufmann Publishers, 2000.
12. A. S. Christensen, A. Muller, and M. I. Schwartzbach. Static Analysis for Dynamic XML. In *PlanX*, 2002.
13. J. Clark. XSL Transformations (XSLT). Technical report, W3C, November 1999.

14. J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0. Technical report, W3C, November 1999.
15. R. Connor, D. Lievens, and F. Simeoni. Projector: a partially typed language for querying XML. In *PlanX*, 2002.
16. M. Fahndrich and R. M. Leino. Declaring and checking Non-Null Types in an Object-Oriented Language. In *OOPSLA*, 2003.
17. D. C. Fallside. XML Schema Part 0: Primer. Technical report, W3C, May 2001.
18. C. Flanagan, K. Leino, M. Lillibridge, C. Nelson, J. Saxe, and R. Stata. Extended Static Checking for Java. In *PLDI*, 2002.
19. V. Gapeyev and B. Pierce. Regular object types. In *ECOOP*, 2003.
20. P. Graunke, S. Krishnamurthi, S. V. D. Hoeven, and M. Felleisen. Programming the Web with High-Level Programming Languages. In *Automated Software Engineering*, LNCS 2028, 2001.
21. R. Griswold and M. Griswold. *The Icon Programming Language (2nd edition)*. Prentice Hall, 1990.
22. J. Guy L. Steele. Growing a Language. *Journal of Higher-Order and Symbolic Computation*, 12(3), 1999.
23. K. Henney. Null Object, Something for Nothing. In *Seventh European Conference on Pattern Languages of Programs*, 2002.
24. H. Hosoya and B. C. Pierce. XDuce: A Typed XML Processing Language. In *International Workshop on the Web and Databases (WebDB)*, 2000.
25. P. Hudak. Building Domain Specific Embedded Languages. *ACM Computing Surveys*, 28(4), 1996.
26. M. Kempa and V. Linnemann. On XML Objects. In *PlanX*, 2002.
27. O. Kiselyov and S. Krishnamurthi. SXSLT: Manipulation Language for XML. In *PADL*, LNCS 2562, 2003.
28. A. Krall and J. Vitek. On Extending Java. In *JMLC*, 1997.
29. T. Lahiri, S. Abiteboul, and J. Widom. Ozone: Integrating structured and semistructured data. *Lecture Notes in Computer Science*, 1949:297+, 2000.
30. D. Leijen and E. Meijer. Domain Specific Embedded Compilers. In *2nd USENIX Conference on Domain-Specific Languages*, 1999.
31. T.-W. Lin. Java architecture for xml binding (jaxb): A primer, 2002.
32. B. Liskov, M. Day, M. Herlihy, P. Johnson, and G. Leavens. ARGUS Reference Manual. Technical Report MIT/LCS/TR-400, 1987.
33. E. Meijer. Server Side Web Scripting in Haskell. *Journal of Functional Programming*, 10(1), January 2000.
34. E. Meijer, D. Leijen, and J. Hook. Client Side Web Scripting with HaskellScript. In *PADL*, 2002.
35. E. Meijer and M. Shields. XMLambda: a Functional Language for Constructing and Manipulating XML Documents, 1999. http://www.cse.ogi.edu/ mbs.
36. E. Meijer and D. van Velzen. Haskell Server Pages. In *Haskell Workshop 2000*, 2000.
37. S. Murer, S. Omohundro, D. Stoutamire, and C. Szyperski. Iteration abstraction in Sather. *ACM Transactions on Programming Languages and Systems*, 18(1):1–15, January 1996.
38. T. Proebstring. Disruptive Language Technologies.
39. N. Schemenauer, T. Peters, and M. L. Hetland. Simple Generators. PEP-0255.
40. M. Shields and E. Meijer. Type-indexed Rows. In *POPL*, 2001.
41. J. Simeon and P. Wadler. The Essence of XML. In *POPL*, 2003.
42. P. Thiemann. WASH/CGI: Server Side Web Scripting with Sessions and Typed, Compositional Forms. In *Practical Aspects of Declarative Languages*, 2002.

43. C. van Reeuwijk and H. Sips. Adding tuples to java: a study in lightweight data structures. In *ACM Java Grande/ISCOPE*, 2002.
44. A. F. Véronique Benzaken, Giuseppe Castagna. CDuce: an XML-centric general-purpose language. In *ICFP*, 2003.
45. P. Wadler. The Concatenate Vanishes, 1989. http://www.research.avayalabs.com/user/wadler/papers/vanish/vanish.pdf.
46. N. Welsh, F. Solsona, and I. Glover. SchemeUnit and SchemeQL: Two Little Languages. In *Third Workshop on Scheme and Functional Programming*, 2002.