Unifying Tables, Objects and Documents

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

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

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♯ 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 programmer productivity, that is to make it easier and faster to write correct programs [29]. 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 are currently rather poorly supported in common object-oriented languages.

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 typically structured using a three-tier model that consists of a *middle tier* that contains the business logic that extracts relational data from a *data services tier* and processes it into hierarchical data that is displayed in the *user interface tier* (alternatively, in a B2B scenario, this hierarchical data might simply be transferred to another application). The middle tier is typically programmed in an object-oriented language such as Java or C♯.

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 usually 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();
```

HTML or XML documents are then created 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 fragile with lots of opportunities for silly errors and no possibility for static checking. In fact, representing queries as strings can be a security risk (the so-called 'script code injection' problem). Finally, due to the poor integration, performance suffers badly as well.

1.2 Previous attempts

It is not an easy task to gracefully unify the worlds of objects, documents and tables, so it should not come as a surprise that no main-stream programming language has yet emerged that realizes this vision.

Often language integration only deals with SQL *or* with XML but not with both [4, 14, 6, 9, 18]. Alternatively they start from a completely new language such as XQuery or XDuce [3, 12], which is a luxury that we cannot afford. Approaches based on language binding using some kind of pre-compiler such as XSD.exe, Castor, or JAXB 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 [13] typically using a functional language such as Scheme or Haskell [26, 27, 22, 23, 19, 15, 10, 32, 33, 4] as the host. In our experience however, the embedded domain specific language approach does not scale very well, and it is particularly difficult to encode the domain specific type systems [30] and syntax into the host language.

1.3 The Xen solution

The examples above demonstrate that at a foundational level there is an impedance mismatch between the XML, SQL and object data-models. In our opinion the impedance mismatch is too big to attempt a complete integration. (The impedance mismatch between the object and XML data-models is treated in detail in a companion paper [25].)

Given these problems, our approach is to first take as our starting point the type system and object data-model of the middle tier programming language. This is the computational model that programmers are familiar with, and that is supported by the underlying execution engine.

We look at XML fidelity in terms of being able to serialize and deserialize as many possible documents that are expressible by some given XML schema language, and not at how closely we match one of the XML data models in our programming language once we have parsed an XML document. In other words, we consider XML 1.0 as simply syntax for serialized object instances of our enriched host language. For SQL fidelity we take the same approach: we require that SQL tables can be passed back and forth without having the need to introduce additional layers like ADO.NET.

Hence, rather than trying to blindly integrate the whole of the XML and SQL data-models, we enrich the type system of the object-oriented host language (in our case C♯) with a small number of new type constructors such as streams, tuples, and unions. These have been carefully designed so that they integrate coherently with the existing type system.

On top of these type system extensions, we then add two new forms of expressions to the base language: generalized member access provides path expressions to traverse hierarchical data, and comprehension queries to join elements from different collections. Rather than limiting comprehension queries to tabular data or path expressions on hierarchical data, we allow both forms of expressions to be used on any collection, no matter whether the data is in-memory or remotely stored in a database. To seamlessly handle queries on both on remote data sources and local data sources we use similar deferred execution implementation techniques as in HaskellDB [19]. Depending on the data source, the result of a query is either a materialized collection or a SQL program that can be sent to the database.

The result is Xen, a superset of C♯ that seamlessly blends the worlds of objects, tables and documents. The code fragment below shows how Xen is able to express the same functionality that we have seen previously.

```
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 Xen, 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 built-in. Xen thus allows for static checking, and because the SQL and XML type systems are integrated into the language, the compiler can do a better job at generating efficient code that might run on the client but which also might be sent to the server.

Although Xen by design does not support the entirety of the XML stack and some of the more advanced features of SQL, we believe that our type system and language extensions are rich enough to support many potential scenarios. For example we have been able to program the complete set of XQuery Use Cases, and several XSL stylesheets, and we can even serialize the classic XML Hamlet document without running into any significant fidelity problems.

The next sections show how we have grown a modern object-oriented language (we take C♯ 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 (§2) and expressions (§3).

2 The Xen type system

In this section we shall cover the extensions to the C♯ type system—streams, tuples, discriminated unions, and content classes—and for each briefly consider the new query capabilities. In contrast to nominal types such as classes, structs, and interfaces, the new Xen types are mostly *structural* types, like arrays in Java or C♯.

We will introduce our type system extensions by example. Formal details of the Xen type system can be found in a companion paper [24].

2.1 Streams

Streams represent ordered homogeneous collections of zero or more values. In Xen streams are most commonly generated by `yield return` blocks. Stream generators are like ordinary methods except that they may yield multiple values instead of returning a single time. The following method `From` generates a finite stream of integers $n, n+1, ..., m$:

```
static int* From(int n, int m){ while(n<=m) yield return n++; }
```

From the view of the host language, streams are typed refinements of C♯'s *iterators*. Iterators encapsulate the logic for enumerating elements of collections.

Given a stream, we can iterate over its elements using C♯'s existing `foreach` statement. For instance, the following loop prints the integers from n to m.

```
foreach(int i in From(n,m)) { Console.WriteLine(i); }
```

Streams and generators are not new concepts. They are supported by a wide range of languages in various forms [11, 20, 17, 21, 28]. Our approach is a little different in that:

- We classify streams into a hierarchy of streams of different length.
- We automatically flatten nested streams.
- We identify the value `null` with the empty stream.

To keep type-checking tractable, we restrict ourselves to the following stream types: $T*$ denotes possibly empty and unbounded streams, $T?$ denotes streams of at most one element, and $T!$ denotes streams with exactly one element. We will use $T?$ to represent optional values, where the non-existence 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 We write $S <: T$ to denote that type S is a subtype of type T and we write $S \cong T$ to denote that S is equivalent to T. Xen observes the axioms $T! <: T$, $T <: T?$ and $T? <: T*$. For instance $T? <: T*$ reflects the fact that a stream of at most one element is also a stream of at least zero elements. Non-stream types T into the subtype hierarchy by placing them between non-null values $T!$ and possibly null values $T?$. Thus allows for example to assign the value `3` to the type `int?`.

Like C♯ arrays, streams are covariant. For unbounded streams the upcast of the elements is via an identity conversion. This restriction guarantees that upcasts of streams are always constant time, and that the object identity of the stream is maintained. Suppose `Button` is a subclass of `Control`, then this rule says that `Button*` is a subtype of a stream of controls `Control*`. If the conversion is not the identity, we have to explicitly copy the stream. For example, we can convert stream `xs` of type `int*` into a stream of type `object*` using the apply-to-all expression `xs.{ return (object)it; }`. Optional and non-null types are covariant with respect to arbitrary conversions on their element types.

In Xen streams are always flattened, there are no nested streams of streams. At the theoretical level this implies a number of type equivalences, for example $T*? \cong T*$ reflects the fact that at most one stream of zero or more elements flattens into a single stream of zero or more elements.

Flattening of stream types is essential to efficiently deal with recursively defined streams. Consider the following recursive variation of the function `From` that we defined previously:

```
int* From(int n, int m){
  if (n>m) {
```

```
    yield break;
  } else {
    yield return n++; yield return From(n,m);
  }
}
```

The recursive call `yield return From(n,m);` yields a stream forcing the type of `From` to be a nested stream. The non-recursive call `yield return n++;` yields a single integer thus forcing the return type of `From` to be a normal stream. 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`:

```
int* From(int n, int m){
  if (n >m) {
    yield break;
  } else {
    yield return n++;
    foreach(int it in From(n,m)) yield return it;
  }
}
```

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. Iterating over a stream effectively perform a depth-first traversal over the n-ary tree produced by the stream generators.

Non-nullness. The type $T!$ denotes streams with exactly one element, and since we identify `null` with the empty stream, this implies that values of type $T!$ can never be `null`.

Being able to express that a value cannot be `null` via the type system allows *static* checking for `null` pointers (see [7, 8] 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 function `IPAddress.Parse`. 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 an `ArgumentNullException`. Nothing of this is apparent in the type of the `Parse` method in C♯. In Java the signature of `Parse` would at least 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`.

2.2 Anonymous structs

Tuples, or *anonymous structs* as we call them, encapsulate heterogeneous ordered collections values of fixed length. Members of anonymous structs can optionally be labelled, and labels can be duplicated, even at different types. Members of anonymous structs can be accessed by label or by position. Anonymous structs are value types, and have no object identity.

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

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

The members of an anonymous struct may be unlabelled, for example, we can create a tuple consisting of a labelled `Button` and an unlabelled `TextBox` as follows:

```
struct{Button enter; TextBox;} x =
  new(enter=new Button(), new TextBox());
```

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

As mentioned earlier, members of tuples can be accessed either by position, or by label. For example:

```
int m = new(47,11)[0];
Button b = x.enter;
```

As for streams, tuples are covariant provided that the upcast-conversion that would be applied is the identity. Subtyping is lifted over field declarations as expected. This means that we can assign `new(enter=new Button(), new TextBox())` to a variable of type `struct{Control enter; Textbox;}`.

2.3 Streams+anonymous structs = tables

Relational data is stored in tables, which are sets of rows. Sets can be represented by streams, and rows by anonymous structs, thus streams and anonymous structs 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	

This table can be modelled by the variable `Pokedex` below:

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

struct{string Name; int HP; Kind Kind; string? Evolved;
}* Pokedex;
```

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

2.4 Discriminated union

A value of a *discriminated union* holds (at different times) any of the values of its members. Like anonymous structs, the members of discriminated unions can be labelled or unlabelled.

Discriminated unions often appear in content classes (see §2.5 below). The type `Address` uses a discriminated union to allow either a member `Street` of type `string` or a member `POBox` of type `int`:

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

The second situation in which discriminated unions are used is in the result types of generalized member access (see §3.2). For example, when `p` has type `Pokemon`, the wildcard expression `p.*` selects all members of `p` which returns a stream containing all the members of a Pokemon and has type

```
choice{string; int; Kind; string?}*
```

Using the subtype rules for `choice` and streams this is equivalent to `choice{string?; int; Kind;}*`.

Unlike unions in C/C++ and variant records in Pascal where users have to keep track of which type is present, values of an discriminated unions in Xen are implicitly tagged with the static type of the chosen alternative, much like unions in Algol68. In other words, discriminated unions in Xen are essentially a pair of a value and its static type. The type component can be tested with the conformity test e `was` T. The expression e `was` T is true for *exactly one* T in the union. This invariant is maintained by the type system. You can get the value component of a discriminated union value by downcasting.

Labelled members of discriminated unions are just nested singleton anonymous structs, for example `choice{int Fahrenheit; int Celsius;}` is a shorthand for the more verbose `choice{struct{int Fahrenheit;}; struct{int Celsius;};}`. Discriminated unions are idempotent (duplicates are removed), associative and commutative (nesting and order are ignored).

Values of non-discriminated unions can be injected into a discriminated union. This rule allows us to conveniently inject values into a discriminated union as in the example below:

```
choice{int Fahrenheit; int Celsius;} = new(Fahrenheit=47);
```

Finally, streams distribute over nested discriminated unions, Again this is essential for recursively defined streams as in the following example which returns a stream of integers terminated by `true`:

```
choice{int; bool;}* f(int n) {
  if(n==0){
    yield return true;
  } else {
    yield return n;
    yield return f(--n);
  }
}
```

2.5 Content classes

Now that we have introduced streams, anonymous structs, and discriminated unions, our type system is rich enough to model a large part of the XSD schema

language; our aim is to cover as much of the essence of XSD [31] as possible whilst avoiding most of its complexity.

The correspondence between XSD particles such as `<sequence>` and `<choice>` with local element declarations and the type constructors `struct` 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 `(0, 1)`, and for $T!$ it is `(1,1)`.

The content class `Address` that we defined in §2.4 corresponds to the XSD schema `Address` below:

```
<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>
```

A Xen content class is simply a normal C♯ class with a single unlabelled member and zero or more methods. As a consequence, the 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 two elements are mapped onto the following two type declarations:

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

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

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

are mapped to ordinary fields of the containing type:

```
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. Further details of the relationship between the XML and Xen data-models can be found in a companion paper [25].

3 Xen expressions

In the previous sections we have concentrated on the Xen type system. In this section we will consider new Xen expression forms to construct, transform, query and combine Xen values.

3.1 XML constructors

Xen internalizes XML serialized objects into the language, conveniently allowing programmers to use XML fragments as object literals. For instance, we can create a new instance of an `Address` object using the following XML object literal:

```
Address a = <Address>
              <Street>One Microsoft Way</Street>
              <City>Redmond</City>
            </Address>;
```

The Xen compiler contains a validating XML parser that analyzes the XML literal and "deserializes" it at compile time into code that will construct the correct `Address` instance. This allows Xen programmers to treat XML fragments as first-class values in their code.

XML literals can also have placeholders to describe *dynamic* content (anti-quoting). We use the XQuery 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.

Note that XML literals are treated by Xen as just object constructors, there is nothing special about content classes. In fact, we can write XML literals to construct values of any type, for example, the assignment

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

creates an instance of the standard `Button` class and sets its `Text` field to the string `"Click Me"`.

3.2 Stream generators, iterators and lifting

To make the creation of streams as concise as possible, we allow *anonymous method bodies* as expressions. In the example below we assign the (conceptually) infinite stream of positive integers to the variable `nats`:

```
// 0, 1, 2, ...
int* nats = { int i=0; while(true) yield return i++; };
```

Our stream constructors (`*`, `?`, `!`) are functors, and hence we implicitly *lift* operations on the element type of a stream (such as member or property access and method calls) over the stream itself. For instance, to convert each individual string in a stream `ss` of strings to uppercase, we can simply write `ss.ToUpper()`.

We do not restrict this lifting to member access. Xen generalizes this with an *apply-to-all* block. We can write the previous example as `ss.{ return it.ToUpper(); }`. The implicit argument `it` refers successively to each element of the stream `Ss`.

In fact, the apply-to-all block itself can yield a stream, in which case the resulting nested stream is flattened in the appropriate way. For example (where `nats` is a stream of integers):

```
// 1, 2,2, 3,3,3, 4,4,4,4, ...
int* rs = nats.{ for(i=0; i<it; i++) yield return 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 write:

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

Apply-to-all blocks can be stateful, so we can use them to do reductions (in the functional community called *folds*). For example, we can sum all integers in an integer stream `xs` as follows:

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

We need to be careful when lifting over non-null types, since the fact that the receiver object is not `null` does not imply that its members are not `null` either:

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

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

Optional types provide a standard implementation of the `null` design pattern; when a receiver of type $T?$ is `null`, accessing any of its members returns `null`:

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

In Objective-C [16] this is the standard behaviour for any object that can be `null`.

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 type `choice{bool; int;}+` when `xs` is defined as:

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

Lifting over discriminated unions 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, the result type is the union of the types of the respective members.

Binary and unary operators are lifted element-wise over streams. For example we can add two optional integers `x+y` to get another optional integer. If either `x` or `y` is `null` the result of adding them is `null` as well. Lifting of optional types implements SQL's three-value logic.

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 `nats[it%2==1]`. For each element in the stream to be filtered, the predicate is evaluated with that element bound to `it`. Only if the predicate is true the element becomes part of the new stream.

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

In fact, filters can be encoded using an apply-to-all block:

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

3.3 Further generalized member access

As we have seen, Xen elegantly generalizes familiar C♯ member access resulting in compact and clear code. However we should like to provide more flexible forms of member access: Xen provides *wildcard*, *transitive* and *type-based* access. These forms are similar to the concepts of nametest, abbreviated relative location paths and name filters in XPath [1], but have been adapted to work uniformly on object graphs.

Wildcards provide access to all members of a type without needing to specify the labels. For example, suppose that we want to have all fields of an `Address`:

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

The wildcard expression returns the content of all accessible fields and properties of the variable `Microsoft` in their declaration order. In this case `"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`s of all `Book`s in the source stream `books`:

```
Book F = <Book>
          <Title>Faust</Title>
          <Author>Goethe</Author>
         </Book>;
Book K = <Book>
          <Title>De Klompeniers</Title>
          <Author>Jac. Broersen</Author>
         </Book>;

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

Transitive member access abstracts from the concrete representation of a tree; as long as the mentioned member is reachable and accessible, its value is returned.

Looking for just a field name might not be sufficient, especially for transitive queries where there might be several reachable members with the same name, but of different type. In that case we allow 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::*`.

3.4 Comprehensions

The previous sections presented our solutions to querying documents. However for accessing relational data, which we model as streams of anonymous structs, simple SQL queries are more natural and flexible. Here we only consider 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*. Here are two simple SQL-style comprehension queries:

```
Pokemon* ps1 =
  select * from Pokedex where Kind == Normal;
struct{string Name; Kind Kind;}* ps2 =
    select Name, Kind from Pokedex;
```

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:

```
ps2 = select Name, Kind from Pokedex;
```

without having to declare the type of `ps2`.

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 comprehensions 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 Pokemons that have evolved from each other:

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

Again, we should like to emphasize the elegant integration of data in Xen. 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 excessive syntactic burden for the programmer as the result types of queries are inferred.

4 Conclusion

The language extensions proposed in this paper support both the SQL [2] and the XML schema type system [31] 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 capture much of the expressive power of XPath [1], XQuery [3] and XSLT [5], 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.

In summary, we have shown that it is possible to have both SQL tables and XML documents as first-class citizens 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.

Acknowledgments

We should 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. This paper was written whilst Bierman was in the University of Cambridge Computer Laboratory and supported by EU AppSem II.

References

1. A. Berglund, S. Boag, D. Chamberlin, M.F. Fernandez, M. Kay, J. Robie, and J. Simeon. XML path language 2.0. `http://www.w3.org/TR/xpath20/`.
2. G.M. Bierman and A. Trigoni. Towards a formal type system for ODMG OQL. Technical Report 497, University of Cambridge Computer Laboratory, 2000.
3. S. Boag, D. Chamberlin, M.F. Fernandez, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML query language. `http://www.w3.org/TR/xquery/`.
4. A.S. Christensen, A. Muller, and M.I. Schwartzbach. Static analysis for dynamic XML. In *Proceedings of PlanX*, 2002.
5. J.J. Clark. XSL Transformations 1.0. `http://www.w3.org/TR/xslt`.
6. R. Connor, D. Lievens, and F. Simeoni. Projector: a partially typed language for querying XML. In *Proceedings of PlanX*, 2002.
7. M. Fahndrich and R.M. Leino. Declaring and checking non-null types in an object-oriented language. In *Proceedings of OOPSLA*, 2003.
8. C. Flanagan, R. Leino, M. Lillibridge, C. Nellson, J. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of PLDI*, 2002.
9. V. Gapeyev and B.C. Pierce. Regular object types. In *Proceedings of ECOOP*, 2003.
10. P. Graunke, S. Krishnamurthi, S.V.D. Hoeven, and M. Felleisen. Programming the web with high-level programming languages. In *Proceedings of ASE*, 2001.
11. R. Griswold and M. Griswold. *The Icon programming language*. Prentice Hall, 1990.
12. H. Hosoya and B.C. Pierce. XDuce: A typed XML processing language (preliminary report). In *Proceedings of WebDB*, number 1997 in LNCS, pages 226–244, 2000.
13. P. Hudak. Building domain specific embedded languages. *ACM computing surveys*, 28(4), 1996.
14. M. Kempa and V. Linnemann. On XML objects. In *Proceedings of PlanX*, 2002.
15. O. Kiselyov and S. Krishnamurthi. SXSLT: A manipulation language for XML. In *Proceedings of PADL*, 2003.
16. S. Kochan. *Programming in Objective C*. Sams, 2003.
17. A. Krall and J. Vitek. On extending Java. In *Proceedings of JMLC*, 1997.
18. T. Lahiri, S. Abiteboul, and J. Widom. Ozone: Integrating structured and semistructured data. In *Proceedings of DBPL*, 2000.
19. D. Leijen and E. Meijer. Domain specific embedded compilers. In *Proceedings of USENIX Conference on Domain-specific languages*, 1999.
20. B. Liskov, R. Atkinson, T. Bloom, E. Moss, J.C.Schaffert, R. Scheiffer, and A. Snyder. *CLU reference manual*. Springer Verlag, 1981.
21. B. Liskov, M.Day, M. Herlihy, P. Johnson, and G. Leavens. ARGUS reference manual. Technical report, MIT, 1987.
22. E. Meijer. Server side web scripting in Haskell. *Journal of Functional Programming*, 10(1), 2000.
23. E. Meijer, D. Leijen, and J. Hook. Client-side web scripting with HaskellScript. In *Proceedings of PADL*, 2002.

24. E. Meijer, W. Schulte, and G.M. Bierman. The essence of Xen. Submitted for publication, 2003.
25. E. Meijer, W. Schulte, and G.M. Bierman. Programming with circles, triangles and rectangles. In *Proceedings of XML 2003*, 2003.
26. E. Meijer and M. Shields. XMλ: a functional language for constructing and manipulating XML documents. Unpublished paper, 1999.
27. E. Meijer and D. van Velzen. Haskell server pages. In *Proceedings of Haskell workshop*, 2000.
28. S. Murer, S. Omohundro, D. Stoutamire, and C.Szyperski. Iteration abstraction in Sather. *ACM ToPLAS*, 18(1):1–15, 1996.
29. T.A. Proebsting. Disruptive programming language technologies. Unpublished note, 2002.
30. M. Shields and E. Meijer. Type-indexed rows. In *Proceedings of POPL*, 2001.
31. J. Simeon and P. Wadler. The essence of XML. In *Proceedings of POPL*, 2003.
32. P. Thiemann. WASH/CGI: Server side web scripting with sessions and typed compositional forms. In *Proceedings of PADL*, 2002.
33. N. Welsh, F. Solsona, and I. Glover. SchemeUnit and SchemeQL: Two little languages. In *Proceedings of Workshop on Scheme and functional programming*, 2002.