

Confessions of a Used Programming Language Salesman

Getting the Masses Hooked on Haskell

Erik Meijer

Microsoft SQL Server
emeijer@microsoft.com

When considering the past or the future, dear apprentice, be mindful of the present. If, while considering the past, you become caught in the past, lost in the past, or enslaved by the past, then you have forgotten yourself in the present. If, while considering the future, you become caught in the future, lost in the future, or enslaved by the future, then you have forgotten yourself in the present. Conversely, when considering the past, if you do not become caught, lost, or enslaved by the past, then you have remained mindful of the present. And if, when considering the future, you do not become caught, lost, or enslaved in the future, then you have remained mindful of the present. [14]

Abstract

Programmers in the real world wrestle every day to overcome the impedance mismatch between relational data, objects, and XML. For the past ten years we have been working on solving this problem by applying principles from functional programming, in particular monads and comprehensions. By viewing data as monads and formulating queries as comprehensions, it becomes possible to unify the three data models and their corresponding programming languages instead of considering each as a separate special case.

To bring these theoretical ideas within the reach of mainstream programmers, we have worked tirelessly on transferring functional programming technology from pure Haskell, via $C\omega$ to the upcoming versions of C# 3.0 and Visual Basic 9 and the LINQ framework. Functional programming has finally reached the masses, except that it is called Visual Basic instead of Lisp, ML, or Haskell!

Nearly all business applications simply boil down to transforming data from one form to the other, for instance an order-processing system written in an object-oriented language that creates XML billing statements from customer and orders information in a relational database.

As a result, programmers constantly need to juggle three very

disparate data models: **Relations** in the data tier + **Objects** in the business tier + and **XML** in the presentation tier = ROX.

Not only is each data model fundamentally different, but each comes strongly coupled with its own programming language, typically SQL for relational data, an imperative language such as Java, C#, or Visual Basic for objects, and XQuery or XSLT for XML. The deep impedance mismatch between the three inhabitants of the ROX triangle is the reason that many programmers in the real world are pulling their hair out on a daily basis.

Fortunately in the first half of the previous century mathematicians were working on esoteric theories such as *Category Theory* and *Lambda Calculus*. These mathematical theories provided deep insight into the algebraic nature of collections and operations on collections. By leveraging this theoretical basis it is actually possible to unify the three data models and their corresponding programming languages instead of considering them as three separate special cases.

After a long journey through theoretical computer science, database theory, functional programming, and scripting, abstract concepts such as monoids, lambda-expression, and comprehensions have finally reached the day-to-day world of ordinary programmers. The LINQ framework effectively introduces monads, monad comprehensions, and lambda expressions into the upcoming versions of C# 3.0 and Visual Basic 9.

This paper is a personal account of my journey to democratize the three-tier distributed programming problem. It starts with my attempt to use Haskell as the language to write three-tier distributed data intensive applications, then continues with my brief flirtation with the Internet Scripting Language Mondrian, the $C\omega$ language, the LINQ framework and C# 3.0 and Visual Basic 9, and ultimately comes to a happy end with my devotion to Visual Basic.

<warning>

The paper is a work in progress. I particularly welcome any pointers to related work and will gladly include these in a future version.

</warning>

1. The Great Internet Hype

In 1997 I was fortunate enough to spend a sabbatical at the Oregon Graduate Institute with Simon Peyton Jones. This was the zenith of the Internet bubble and we got inspired to apply Haskell, the world's finest imperative language, to solve the

[copyright notice will appear here]

ROX problem. This ended my “bananas” [36] period and I turned from a theorist into a practitioner.

Until that time, Haskell programs usually lived in a fairly closed world (with notable exceptions such as [17]), so the first thing we tackled was to make it dead simple to interface Haskell with imperative code.

1.1 Interfacing Haskell to the Outside World

One of the premises of Landin’s seminal paper “The Next 700 Programming Languages” [25] is that most languages can be considered as a collection of primitive building blocks plus glue to combine smaller blocks into larger blocks.

Haskell’s lazy evaluation and monads make Haskell a very powerful glue [23], but for a long time it was hard to access externally implemented libraries in Haskell, so there was little to glue together.

Interfacing Haskell (or any other language) to the outside world requires at the lowest level just four building blocks:

- We need to be able to wrap an external function pointer (either statically or dynamically loaded) as a Haskell function.
- We need to be able to put pointers to external data under control of the Haskell garbage collector.
- We need to wrap a Haskell function as an external function pointer.
- We need to be able to pin Haskell values in memory such that they are not moved around by the Haskell garbage collector.

This basic functionality is available as the standard Foreign Function Interface (FFI) for Haskell 98 [10].

Having these a basic FFI mechanism makes it possible to interoperate with other languages, but it required the determination of a monk to actually make things work because of the large amount of low-level plumbing needed to marshall values across language boundaries.

1.1.1 GreenCard and HaskellDirect

The first attempt at making FFI easy to use was the Greencard [45] preprocessor. Using Greencard programmers put special directives in their code that were used by the preprocessor to generate all the low-level primitives to do the dirty interfacing and marshallng work. HaskellDirect [49, 50] was an attempt to automatically generate FFI boilerplate code from a stand-alone IDL description for either an external library or a Haskell program.

IDL is a quite powerful, but rather messy, underspecified, and complex type-system. However, in the late nineties we all believed that software components were the silver bullet and so it made sense to use a (more or less) language-independent external description for components. Unfortunately, soon after we made our bet, binary component models like COM and Corba fell out of fashion in favor of language-specific, meta-data driven, models such as Java and .NET. As a result, the investments in H/Direct never paid off.

1.1.2 COM

COM is a binary component standard that at its core shows a minimalist design that to a geek evokes the same feelings of deep beauty and elegance as category theory. The model just imposes a handful of “axioms” that components and component consumers must observe.

Alas, the infrastructure around COM (such as the registry, the ingenious but complex OLE protocols, and the ever-changing marketing names) as well as the lack of a good programming language (not C++!) that natively supported COM, gave it a bad reputation. Nevertheless, we still believe that COM is a great component model. In fact, the XPCOM component model, which is the basis for many open source projects such as Mozilla, is basically a clone of COM. Seen in that light it is surprising that nobody has ported either the Haskell [51] or the SML COM infrastructure [47] to work with XPCOM.

One thing that did stick from our work of interfacing Haskell to COM is the notion of *phantom types*. A phantom type is a parametrized type whose type parameter is *not* used in its RHS. When we first discovered phantom types many people would not believe that they were legal; now they are the standard mechanism for advanced-type hacking in Haskell[9, 18, 21, 20].

We used phantom types in a variety of different ways. The most direct was to represent typed pointers using the type synonym `Ptr a = Addr`, which defines `Ptr` to be a synonym for addresses. However, the type parameter allows us to distinguish between a pointer to an integer `Ptr Int` and a pointer to a pointer to an integer `Ptr (Ptr Int)`. We also heavily used phantom types to model interface inheritance [26].

1.1.3 Automation

COM Automation is a reflexive layer built on top of COM that makes it easy for dynamic and scripting languages to access and create COM components.

The pre-.NET Visual Basic dialects are in many respects a programming language abstraction over COM Automation.

Because of the additional level of indirection introduced by Automation, it becomes much easier for *any* language to use and create COM components. The language infrastructure just has to provide a binding for the single COM interface `IDispatch` and from there on programmers can access any Automation component. Similarly the language needs to provide one generic factory method that wraps a collection of function pointers into an `IDispatch` interface, and from there on programmers can easily create new Automation components [28].

Many of Microsoft’s applications, such as Word, Excel, Outlook, Powerpoint, and Visio, are all fully scriptable using Automation, so we naturally assumed that if we exposed Automation in Haskell, the world would instantly fall for functional programming. For some odd reason, this never happened.

1.2 Server-side Scripting Using Haskell

Now that it had become easy to call imperative code from within Haskell on the one hand and expose Haskell functions to imperative code on the other hand, we turned our attention to the original goal of creating distributed three-tier applica-

tions and solving the ROX impedance mismatch. Since the Internet bubble was still expanding, naturally the first problem we tackled was writing dynamic HTML pages in Haskell.

Until then, most dynamic Web pages were written in Perl using thin wrappers on top of the basic CGI protocol. The reason that Perl was, and still is, popular for this task is that one can use regular expressions to parse query strings into hashes of name-value pairs that represent the data posted by the Web page to the server, and use “here” documents as simple text-based templates to generate dynamic HTML.

1.2.1 Perl for Swine

The *Perl For Swine* library [31] leverages algebraic data types and higher-order functions of Haskell by providing a very simple domain-specific language of HTML-generating combinators, and a worker-wrapper style abstraction of the actual CGI protocol. The user just writes a worker function of type `Query -> IO HTML`, which then gets wrapped into an `IO ()` action that parses the query string of the incoming HTTP request into a list of name-value pairs `[(String, String)]` and wraps the result using the proper MIME type.

In addition to the basic library, we also integrated the Hugs interpreter into the Apache Web server. The `mod_haskell` system eliminates the startup overhead for the Hugs interpreter when serving up a page, and it gives access to full power of the Apache Web server.

For some reason, `mod_haskell` never even got close to the astronomical popularity of `mod_perl`. The CGI library lives on as the `Network.CGI` library.

1.2.2 Haskell Server Pages

While using combinators to generate HTML is very powerful and concise, it is quite hard to generate beautiful HTML pages in a completely programmatical way. Professional Web sites consist of a combination of static content designed by professional artists sprinkled with dynamically generated fragments.

Systems such as ASP and PHP facilitate this form of Web site development via static HTML templates (designed by an artist) with embedded *holes* for those parts of the page that need to be generated dynamically (written by a programmer). The implementation of ASP and PHP is just a simple preprocessor that turns each line of HTML into a `Response.Write` statement. As a result, it is not possible to arbitrarily nest further HTML inside code, and code inside that HTML, etc. which completely breaks compositionality.

Haskell Server Pages [44] take the idea of HTML templates a step further by expanding HTML literals into calls to the Perl For Swine HTML generating combinators to allow arbitrary nesting of *concrete* HTML syntax and code. In addition HSP also introduced the notion of pattern matching against HTML.

An HSP-derived pre-processor is available as part of the WASH system [52], and as the MSc thesis of Niklas Broberg [16].

1.2.3 HaskellDB

HaskellDB [27] is a domain-specific library for programming against relational data. Usually domain-specific languages are

implemented via *shallow* embedding into Haskell or any other host language. That is, we define a set of base combinators that embody the semantic algebra of the language you are embedding, and glue these together into bigger denotations using the host language’s abstraction mechanism.

For example, the parser monad `Parse a` provides primitive operations such as `char :: Char -> Parser Char` and combinators such as `many :: Parser a -> Parser [a]` to build composite parsers, which all denote interpreter functions of type `String -> [(a,String)]`.

The main innovation in HaskellDB is the idea of a *domain-specific embedded compiler*, or *deep* embedding. In this case we define a set of base combinators that embody the abstract syntax tree of the embedded language and use the host language’s abstraction mechanism to build bigger abstract syntax trees. Given such an abstract syntax tree, in a second step we evaluate these trees, or compile them into a target language and execute that, to obtain the value they denote.

For example, the `Query a` monad in HaskellDB is a state monad where the state contains the expression tree that is constructed by the query. The `runQuery :: Query a -> IO [a]` compiles the underlying expression to SQL, submits it to a back-end database, and returns a collection of rows as the result.

HaskellDB has lingered for a long time, but recently it has been revisited and improved [2].

1.2.4 XMLambda

With HSP we had veered outside the boundaries of pure embedded domain-specific languages by introducing new syntax, and the HaskellDB experiment convinced me that while the Haskell type-system is incredibly expressive, it might be preferable to design a stand-alone DSL to filter, query, pattern match, and transform XML documents. XMLambda [42] was an experiment to design such a language that turned out to be an “Edsel” (the wrong language at the wrong time). It featured a complicated DTD-based static type inference system that supported polymorphism and higher-order functions.

Subsequent work on type-indexed rows [48] marked the lost weekend of my typholic years. And it slowly started to dawn on me that you can overdo static typing. Unfortunately, I had not yet hit bottom.

1.3 Client-Side Scripting Using Haskell

Although things were quiet on the server front, dominated by Apache and Perl, the browser wars were in full swing, with rapid and exciting innovations around DHTML and client-side scripting. This was a propitious time to introduce Haskell as a hot, new client-side scripting language.

1.3.1 HaskellScript

Because Microsoft wanted to support both JavaScript as well as VBScript in its Internet Explorer browser, it defined the *ActiveX Script Host* interfaces that allowed *any* script engine to be hosted in Internet Explorer and other hosts such as the Windows shell and the IIS Web server. Netscape Navigator also supported a plug-in model, but as far as we know, nobody has

attempted to write a Haskell plugin for that.

HaskellScript [37] is an implementation of the ActiveX Script Engine interfaces on top of a custom COM interface, IScript, on top of the standard Hugs Server.

In this way, it is possible to embed the Hugs interpreter into any application that wants to make itself scriptable, such as games, the browser, the shell, etc. In our experience Haskell was an excellent language for DHTML and shell scripting, but for unknown reasons HaskellScript never came near the astronomical popularity of scripting languages such as Python, Lua, and most recently Ruby.

1.3.2 Lambada

Since we speculated that one of the reasons for the low adoption rate of H/Direct and friends might be the fact that the technology was Windows-specific, we tried to sell Haskell as the ultimate component glue language a second time by interfacing Haskell to Java [35] via JNI [30]. Interestingly, Sheng Liang, one of the designers of JNI, worked on monadic interpreters in a previous life [29].

However, also the Haskell-Java binding failed to gain any traction within the Haskell community. By that time however, the .NET Framework had appeared, and with its promise of a multi-language runtime, it seemed a good idea to revisit the old dream [33] of creating a radically simplified functional scripting language from scratch.

1.4 Internet Scripting Using Mondrian

Inspired by the conceptual minimalism of the "De Stijl" movement, Mondrian [38, 24] was an experiment to reduce Haskell to its bare essence: higher-order functions, lazy evaluation, and monads and for the rest piggybacking as much as possible on the underlying .NET framework. In many respects F^\sharp [3] is the moral successor of Mondrian, except that it uses the strict and imperative functional language OCaml [1] as its basis.

Just like modern art does not evoke strong positive feelings in people, the Mondrian language failed to seduce the (functional) language community at all. Apparently, I lack any salesmanship skills, so to save the common programmer I decided to sell my soul to the most popular programming paradigm, objects, and to the company that has the biggest market share, Microsoft.

2. The Dark XML Ages

While I was working in the CLR team on low-level features such as relaxed delegates and lightweight code generation, XML kept gaining momentum. The time was ripe to pitch the idea of making XML a first-class citizen in C^\sharp to chairman Bill Gates himself. With much encouragement from Don Box, Wolfram Schulte and I submitted a Thinkweek paper [39] on this topic. The language was originally called X^\sharp , but company marketing decreed that the # suffix was verboten so we changed it to Xen. Later, when we joined efforts with the Polyphonic C^\sharp group [11] in Cambridge, the name was finally changed to C_ω .

2.1 C_ω

The goal of C_ω [41, 40, 12] was to enhance the CLR type-system to encompass more of the XSD type-system so that

we could achieve a shallow embedding of as much of XML as possible into C^\sharp . Hence in C_ω there is no XML; instead XML is just a serialization format for C_ω objects.

To appreciate the difficulties of XML let us now make a short excursion to the world of XML schema [8], perhaps the best example of a south-pointing chariot in computer science. XSD must be one of the most complex artifacts invented by mankind, where simply using DTDs (or a compass instead of the chariot) would be a perfectly fine solution.

2.1.1 The Sucking Black Hole of XSD

The main problem of XSD is not that it gratuitously uses XML as its concrete syntax, but the fact that it is completely over-engineered for the problem it attempts to solve. The most confusing aspect of XSD is the notion of *complexType*. The example below defines the schema for XML elements of the form `<Point><x>4711</x><y>13</y></Point>`:

```
<complexType name="PointType">
  <sequence>
    <element name="x" type="integer"/>
    <element name="y" type="integer"/>
  </sequence>
</complexType>

<element name="Point" type="PointType"/>
```

The idea of *complexTypes* is to describe the shape of the *content* of elements, supposedly to aid reuse. No programming language we know of introduces this kind of additional layer of types to describe the inner shape of its regular type. While there are no values of type *complexType*, the fact that they are called *type* however leads many people to believe that in a shallow embedding of XML into objects, *complexTypes* and not elements should be mapped to classes.

Under that interpretation, the schema above would translate to the following class:

```
Class PointType
  x As Integer
  y As Integer
End Class
```

However, this begs the question what to do with the declaration `<element name="Point" type="PointType"/>`? One possibility is to map it to a wrapper class `Point`, with an implicit conversion to `PointType`. In any case there is a discrepancy: some elements are mapped to types and some are mapped to fields, causing an incoherence with the semantics of XPath where all path selections return collections of elements.

We could go on for pages talking about the subtleties of mapping XSD to objects, but we cut it short by observing that any attempt that does not uniformly map elements to types is fundamentally flawed because XML values are node-labeled trees, while objects are edge-labeled graphs.

2.1.2 Type-System Extensions

The solution to the XSD mapping problem we attempted in C_ω was to extend the CLR type-system with various structural

types:

```
T ::= N | T[]
    | T(..., T, ...)
    | T | T | T&T
    | T- | T! | T? | T+ | T*
    | struct { ..., T[m], ... }
```

These new structural types include disjoint union `|`, a family of stream types, `-` for exact types, `!` for non-null types, i.e. streams with exactly one element, `?` for optional types, i.e. streams with either zero or one element, `+` for non-empty streams, and `*` for possibly empty streams, and optionally labeled records `struct { ..., [T] m, ... }` to allow the specification of richer object models than just classes with a collection of fields.

In addition, `Cω` added function types `T(..., T, ...)` (which show up in LINQ as `Func(Of ..., T, ..., S)`) and intersection types (`&`).

In `Cω` we can define a schema for email messages such as

```
msg = <Email>
    <To>BillG</To>
    <From>Erik</From>
    <Body>
    <P>Visual Basic is also my
        favorite language</P>
    </Body>
</Email>
```

using the following type declaration

```
class Email {
    string To;
    string From;
    string? Subject;
    struct{ string P; }* Body
}
```

`Cω` also has type inference for local variables, and the inferred type for the `msg` variable is `Email`.

XML literals in `Cω` were just serialized objects, and the compiler translated such literals into constructor calls of the type denoted by the literal. Just like Haskell Server Pages, `Cω` XML literals could contain arbitrarily nested expression and statement holes.

2.1.3 Generalized Member Access

The slogan of `Cω` is “The Power Is In The Dot!” which refers to the fact that in `Cω` we aggressively lift member access over all structural types. For example, given a collection `bs` of type `Button*`, we can write `bs.BackColor` to return the individual colors of each button in the collection. The explicit notation for lifting uses an anonymous block expression `bs.{ return it.BackColor; }`.

The reason we introduced lifting over structural types is that doing so makes writing path expressions really convenient, especially since nested streams were automatically flattened.

2.1.4 Nullable types

In the Whidbey version of the .NET Framework, nullable types were introduced in `C# 2.0` using the same `?` syntax as `Cω`. In `C# 2.0`, conversions and binary operators over `T` are lifted, but normal member access is not. The `T?` type constructor is constrained to take a non-nullable value type `T` as its argument, so nullable types cannot be nested. Unlike `Cω`, there is no implicit conversion from `T?` to `T*`.

The biggest impact of `Cω` on the real world has been to ensure that nullable types in the CLR are coherent; that is when a `null` value of type `T?` is boxed to `object`, it is mapped to the `null` pointer, and when a non-`null` value `t` of type `T?` is boxed to `object`, the value is first unwrapped and then boxed. The conversion sequence goes like this: `(object)(T)t`. Without going through two steps, boxing `null` would not be `null` `(object)null != null`. Also first upcasting to nullable and then boxing would not give the same result as boxing, that is, `(object)(T?)t != (object)t`.

2.1.5 Query Comprehensions

Besides generalized member access and explicit lifting, `Cω` also supports filter expressions of the form

```
buttons[ it.BackColor = Color.Red ]
```

and SQL-style comprehensions. The compiler has built-in knowledge about queries over streams (list comprehensions) and about queries over remote databases (the query monad). It is possible to add other overloads by writing compiler plugins.

3. LINQ, Finally Getting Closer to the Real World

As the `Cω` incubation was winding down, the `C#` team started to spin up the design work for `C# 3.0`, and Matt Warren and myself went over to `C#` to spread the intellectual DNA that we amassed during the `Cω` work and before.

Concurrently I rekindled up my interest in scripting and dynamic languages [34], sparked by the staggering complexity of the `Cω` type-system. I became convinced that deep embedding is the best way to deal with XML in a language, with an optional and layered type-system on top. [15]. This led me to the realization that Visual Basic was the ideal language for the road ahead because it is the only widespread language that allows static typing where possible and dynamic typing where necessary.

It should be no surprise that the goal of LINQ is to unify programming against relational data, objects, and XML. In LINQ we have managed to strike a nice balance of libraries and language extensions. Moreover, all of the language extensions are valuable by themselves, so the total is really more than just the sum of the parts. Both Visual Basic and `C#` have parity with respect to each of the LINQ supporting features.

3.1 Anonymous Types

In queries we often want to project out certain members of a certain value and combine them with the projected members of another value. For instance, given a customer `C` and

its address A, we want to return the pair C.Name and A.City without having to declare and introduce a new nominal type. This is exactly the reason that functional languages and C ω have labeled *records* or anonymous types.

In Visual Basic, we create a record with a Name and a City member by writing

```
Dim Customer =  
  New With { .Name = "Bill", .City = "Seattle" }
```

Since extensible records and record subtyping are still open research problems, and because the underlying CLR runtime does not directly support structural types, there is no record subtyping.

In C \sharp anonymous types are *expressible* (they can be the result of an expression) but not *denotable* (you cannot say their type), so they cannot appear as argument or result types of methods, or be used as properties or fields; and type inference is absolutely necessary for expressions that return anonymous types. Within a single method, all structurally equivalent anonymous types are mapped to the same underlying compiler generated nominal type.

3.2 Object Initializers

Introducing query comprehensions in the language forces us into a much more expression-oriented style that is common in functional languages rather than the usual statement-oriented style that people are used to in imperative languages. To facilitate this, both C \sharp and Visual Basic introduce the notion of *object initializers*.

An object initializer such as (using C \sharp syntax):

```
new Person { First = "Jacques",  
             Last = "Chirac" }
```

corresponds to the C ω block expression

```
new Person().{ it.First = "Jacques";  
              it.Last = "Chirac"; return it; }
```

that creates a new instance and then assigns values to the fields or properties of the just-created instance.

Many types contain read-only members of mutable types; to initialize these we just supply a list of values for each member to initialize them. If we want to create a new instance for an embedded member, we recursively use an object initializer expression:

```
Dim Pair =  
  New Person With {  
    .Name = New Name With { ... },  
    .Address With { .City = "Seattle", ... }  
  }
```

3.3 Type Inference

In a purely nominal type system such as pre-generics Java or CLR type inference does not add much value. When the only compound types are arrays, most expressions have simple types, such as `Hashtable`, regardless of whether their “real” type, such as `HashTable(Of Integer, List(Of String))` is complex or not. However, with the advent of generics and

anonymous types, values can be typed much more precisely, making explicit typing much more painful.

Moreover, having expressible but non-denotable types makes type inference necessary since it is not even possible to write down the type of certain expressions, for instance `new{ x=4711, y=13 }`.

Traditionally languages with type inference look at all uses of a variable when inferring its most general type, usually via some form of unification or constraint-solving procedure. While this guarantees that inferred types are in some sense most precise, it also leads to hard-to-understand error messages, as every Haskell and SML user has experienced. The situation even gets murkier due to (user-defined) implicit conversions, while the presence of the uber type `Object` makes inferred types degenerate to `Object` pretty quickly where using a variable at two disparate types is most probably an error.

Type inference in the presence of overloading and subtyping is a hard problem, and has been a very active research area for many years [46]. Inferring types for function arguments is non-obvious. For example, what would be the inferred type for the argument X of the function `Function(X) Return X.Foo()` when there are multiple types (classes or interfaces) in scope that have a `Foo` method, each of which can be overloaded on their argument type. Inferring result types for functions is equally non-obvious in the presence of subtyping and overloading.

The 80/20 solution is to infer types only from the initializer expression of just local variable declarations. This is simple, simple to implement, and is conceptually closest to explicitly typed local declarations; the only difference is that the compiler will infer the type that the programmer would provide otherwise.

3.4 Extension Methods

Unlike Java, but like C++, in both Visual Basic and C \sharp , methods are non-virtual by default. An instance method is really nothing more than a static method with an implicit receiver (called `Me` in Visual Basic, and `this` in C \sharp). Calling an instance method does not involve any dynamic dispatching and the call is resolved completely statically.

Extension methods lift the restriction that instance methods need to be defined in the receiver’s class. In C \sharp 3.0 and Visual Basic 9, *any* static method can be marked as an extension method, and hence can be invoked using instance call syntax `e.f(...,a...)` instead of using the normal static call syntax that mentions the class C in which the method is defined `C.f(e,...,a,...)`. Both Visual Basic and Java (but not C \sharp) allow class imports. Using class imports we can write a static method call on a class C as `f(e,...,a...)` by omitting the class C.

The major advantage of extension methods over regular instance methods is that we can add extension methods to a receiver type after the fact, and moreover, we can add new methods to *any* type, including interfaces such as `IEnumerable(Of T)` and constructed types such as `string[]`.

This latter capability is key to defining the standard query operators over any type. For instance, using C \sharp syntax, the

definition of the standard query operator `selectMany` on `IEnumerable<T>` (the bind operator `>>=` of the list monad in Haskell) is defined as follows:

```
public static class Sequence
{
    static IEnumerable<S> selectMany<T,S>(
        IEnumerable<T> src, Func<T, IEnumerable<S>> f)
    {
        foreach(var t in src)
            foreach(var s in f(t))
                yield return s;
    }
}
```

Extension methods are a pure compile-time mechanism. The runtime type of the receiver is not actually extended with additional methods. In particular, reflection does not know anything about extension methods and hence late binding over extension methods is not possible.

In many ways this makes extension methods similar to the "method call" operator

```
receiver # method = method receiver
```

that we introduced in Haskell when we started using COM components, and which has been rediscovered as the `[>` operator in `F#` recently.

3.5 Expression Trees

One of the biggest hassles of deep embedding is to create representations of embedded programs that contain bound variables. Because Haskell at that time lacked quoting or any form of reifying its internal parse trees, HaskellDB required subtle hacks to create expression tree combinators that forced users to write predicates as `X!name .==. constant("Joe")`.

In Lisp or Scheme we would of course use quote and quasi quote to turn code into data and escape back to code. The problem with explicit quoting in Lisp is really the same as the HaskellDB mechanism: the API writer has to decide to use data or code, and then the user has to decide to quote or not. This is an example of what we call "retarded innovation".

One of the most exciting features of both `C# 3.0` and `Visual Basic 9` is the ability to create code as data by converting an inline function or lambda expression based on the expected static type of the context in which the lambda expression appears.

Assume we are given the lambda expression `Function(X)X>42`. When the target type in which that lambda expression is used is an ordinary delegate type, such as `Func(Of Integer, Boolean)`, the compiler generates IL for a normal delegate of the required type. On the other hand, when the target type is of the special type `Expression(Of Func(Of Integer, Boolean))` (or any other nested delegate type), the compiler generates IL that *when executed will create an intentional representation of the lambda expression that can be treated as an AST by the receiving API*.

The major advantage of this style of type-directed quoting via `Expression(Of ...)` is that it is now (nearly) transparent to the *consumer* of an API whether to quote or not. The user only has to remember to use lambda expressions as opposed

to ordinary delegate syntax.

Just like HaskellDB, the Dlinq part of the LINQ framework takes advantage of expression trees to define an implementation of the standard query pattern that as its effect computes a program that when executed computes a collection of results.

3.5.1 Standard Query Pattern

The higher-kinded shape of a generic type `C<T>` that supports a simplified version of the standard query pattern contains the well-know (monadic) operators `filter`, called `Where`; `map`, called `Select`; and of course `bind`, which is called `SelectMany`:

```
Class C(Of T)
Function Where
    (P As Func(Of T, Boolean)) As C(Of T)
Function Select(Of S)
    (F As Func(Of T,S)) As C(Of S)
Function SelectMany(Of S)
    (F As Func(Of T, C(Of S))) As C(Of S)
End Class
```

When Java and the CLR introduced generics, they unfortunately did not allow for parameterizing over *type constructors* as opposed to abstracting over just types. The consequence of this oversight is that is impossible to enforce the standard query-operator pattern using the CLR or Java type system.

Because of the purely syntactic way comprehensions are translated into the underlying sequence operators (as we will see in the next section), it is also possible to implement the pattern using non-generic types, for instance using a `Where` method of the shape:

```
Function Where
    (Src As Qs, Pred As Func(R, S)) As Ts
```

In this case, the type dependency between the element type of the source and the argument type of the predicate is lost, which means we cannot define typing rules at the level of query comprehensions themselves.

The upside of this flexibility is that we get more freedom to implement the standard query pattern. For example, the various methods could also be defined as extension methods (which we rely on for the implementation of the pattern over `IEnumerable<T>`) and most importantly, the methods can take expressions trees instead of just delegates.

The generic delegate type `Func(Of A, R)` represents a function of type `A -> R` or `R(A)`, but the methods in the patterns could equally well use some other delegate type with the same argument and result type.

3.5.2 Query Comprehensions

Just as in Haskell, where list and monad comprehensions are syntactic sugar for more complex expressions in terms of the standard monad operators, both `Visual Basic` and `C#` define special comprehension syntax that the compiler expands into the standard query operators. The main advantage of using comprehension syntax over the low-level operators is that query comprehensions introduce a more convenient scope for bound variables.

In Visual Basic, comprehensions are fully compositional, and act as a pipeline that transforms collections of tuples into collections of tuples. The following query joins all books from Amazon and Barnes and Noble by ISBN number and selects the price at each store as well as the title of the book, and finally filters out all books that cost more than a hundred dollars. Note the use of *punting*, where the compiler infers the record labels from the expression, in the Select clause:

```
Dim BookCompare =
  From A In Amazon, B In BarnesAndNoble
  Where A.ISBN = B.ISBN
  Select A.Title,
         PriceA = A.Price,
         BPrice = B.Price
  Where Max(A.Price, B.Price) < 100
```

In the de-sugared code that the compiler generates, the From clause of the query constructs the Cartesian product of the two source collections using a nested Select(Many), the Where clause then lifts the iteration variables A and B over the compiler-generated argument `_It_`, the Select projects the pair of A and B into a triple Title, APrice, and BPrice, and finally the last Where clause again lifts these iteration variables over the compiler-generated argument `_It_`:

```
Dim BookCompare =
  Amazon.SelectMany((A)
    BarnesAndNoble.Select((B)
      New{A,B})).
  Where(( _It_ ) _It_.A.ISBN = _It_.B.ISBN).
  Select(( _It_ ) New{ _It_.A.Title,
                    PriceA = _It_.A.Price,
                    BPrice = _It_.B.Price}).
  Where(( _It_ ) Max(_It_.APrice, _It_.BPrice) < 100)
```

The Visual Basic compiler contains a standard peephole optimizer that post-processes the generated code to eliminate unnecessary intermediate values.

3.6 XML integration

While ideally XML is just a serialization format that is hidden from the programmer, it has now become so pervasive that in those situations where we do need to deal with XML, it should be as convenient as possible.

For this reason, LINQ introduces a new XML API called XLINQ [32], which replaces the standard DOM. On top of XLINQ, Visual Basic 9 supports deep embedding of XML via XML literals that allow cut and paste of arbitrary XML fragments.

3.6.1 XLinQ API

The standard W3C DOM API is document-centric, which means that elements and attributes exist in the context of a specific document, hence elements and attributes are not first-class values. Due to this document centrality, construction of nodes becomes imperative. You first create a node using a factory method on the target document and then explicitly add it as a child of another existing node. The DOM model is inside-out: objects can be created independently, free of the context of their container. This blocks modularity and reusability. Imperative construction does not fit very well in the expression-oriented style required by LINQ in general and query comprehensions in particular.

Accessing nodes using the DOM is extremely inconsistent, with many special cases. The methods `GetAttribute` or `GetAttributeNode` access a particular child *attribute*, but the `Item` default property (indexer) accesses child *elements*. The special `FirstChild` and `LastChild` methods exist for elements but not for attributes.

The XLinQ API is an alternative for the DOM where elements and attributes are first-class values and are constructed via normal constructor calls (*functional construction*), independent of any particular document context. When an already parented node is added to a child of a new parent, that node is automatically cloned. All XPath axes, such as `Parent`, `Descendants`, `Elements`, `Attributes`, and so on, are available as (extension) methods on nodes and collections of nodes. The latter, of course, closely reflects member lifting of $C\omega$.

3.6.2 XML Literals

While the XLinQ API is already a major improvement over the DOM¹, it is not yet simple enough. On top of XLinQ's functional construction, Visual Basic allows XML literals, (fragments of) XML documents that the compiler translates into XLinQ constructor calls. For instance, the declaration below

```
Dim CD = <CD Genre="rock">
  <Title>Stop</Title>
  <Artist>Sam Brown</Artist>
  <Year>1988</Year>
</CD>
```

is compiled into the following XLinQ calls:

```
Dim CD = New XElement("CD",
  New XAttribute("Genre", "rock"),
  New XElement("Title", "Stop"),
  New XElement("Artist", "Sam Brown"),
  New XElement("Year", 1988))
```

XML literals can contain expression holes at any position where the underlying API allows an argument of a type compatible with the expression plugged into the hole. For example, we can create an XML document with all rock CDs from the FreeDB database using the following simple query:

```
Dim Rock =
  <?xml version="1.0" ?>
  <CDs><%=
    From CD In FreeDB
    Where CD.Genre = "rock"
    Select <CD>
      <Title><%= CD.Title %></Title>
      <Artist><%= CD.Artist %></Artist>
      <Year><%= CD.Year %></Year>
    </CD>
  %></CDs>
```

The outer document (the `<? xml version="1.0" ?>` causes the inferred type of the variable `Rock` to be `XDocument`).

3.6.3 Namespaces

We gladly got rid of XSD schemas in Visual Basic 9, but there is no way around XML namespaces. As James Clark remarks[7],

¹DOM means brain dead in Dutch.

namespaces are one of the most confusing aspects of XML. Perhaps one of the main benefits of XML literals is the fact that users can copy and paste XML including namespaces into a Visual Basic program and start modifying it from there, in the same way many of us deal with make files and the like.

There are two ways to declare a namespace prefix: by using a global `Imports prefix = URI` declaration, and by a normal `xmlns:prefix=URI` declaration inside an element. Global `Imports` namespace declarations scope over the whole program, while normal `xmlns` namespace declarations scope over their embedded elements and attributes, but not inside expression holes.

```
Imports X = "http://www.freedb.org"

Dim CD = <Y:CD xmlns:Y="http://www.freedb.org"
  Genre="rock">
  <Y:Title>Live!</Y:Title>
  <X:Artist>Anouk</X:Artist>
  <Y:Year>1997</Y:Year>
</Y:CD>
```

As we will see next, global prefix declarations are useful for axis member selection.

3.6.4 Axis Members

In Visual Basic, we have introduced special syntax for the three most common axes: `Children`, `Descendants`, and `Attributes`. The child axis `cd.Children("Title")` is written using syntax that resembles an element `cd.<Title>`; the descendant axis `CDs.Descendants("Artist")` is similar, but uses three dots `CDs...<Artist>`; and lastly the attribute axis `cs.Attributes("Genre")` is abbreviated as `cd.@Year`.

We use a global prefix to access elements and attributes with qualified names. For example, given the second CD example, we must write `CD.<X:Year>` to access the `<Y:Year>` child since `Y` was declared via the local namespace declaration `xmlns:Y = "http://www.freedb.org"` and hence the fully qualified name of the element is `{http://www.freedb.org}Year`.

Unlike $C\omega$, we do not assume any schema information. Instead we will optionally layer [15] XSD information on top of the CLR type-system to guide Intellisense in the IDE for XML literal construction and axis members. However, this type information has no impact on the runtime behavior of the program.

3.7 DLinQ

Besides language extensions, standard query operators, and XLinQ, the fourth pillar of the LINQ framework is DLinQ; a domain-specific library for accessing relational data. The implementation of the standard query operators for DLinQ mirrors the Query monad of HaskellDB and uses expression trees `Expression(Of Func(Of ...))` instead of delegates `Func(Of ...)`. The DLinQ infrastructure then compiles these expression trees into SQL and creates objects from the result of running the query on a remote database.

DLinQ also provides the usual object-relational mapping infrastructure such as a context that tracks object identity of rehydrated rows, and tracks changes to the object graph to submit changes back to the underlying database.

4. The Great Internet Hype, Version 2.0

After a five-year hibernation, people have rediscovered DHTML and client-side scripting in combination with Web services under the monikers Web 2.0 and AJAX. It is my current belief that Visual Basic is the ultimate language to democratize programming against the Cloud, and hence to bring my quest to a happy end.

4.1 Visual Basic

People often snort at Visual Basic, either because they still have an outdated idea of "Basic" in mind, or because they think that Visual Basic .NET is just C^\sharp with slightly more verbose syntax. Nothing is further from the truth.

4.2 Static Typing Where Possible, Dynamic Typing Where Needed

As we have argued elsewhere [34], the artificial separation between the supporters of dynamically and statically typed languages is rather unfortunate.

Dynamically typed languages miss a great chance to leverage static information about programs that the compiler can infer. Not writing types does not imply no static types [6, 1]. On the other hand, the runtime correctness of programs as implied by most contemporary static type-systems is rather weak. Moreover, even statically typed languages need a few drops of dynamism (downcasts, reflection, array bounds checking) to make things run smoothly.

Visual Basic is unique in that it allows static typing where possible and dynamic typing where necessary. When the receiver of a member-access expression has static type `Object`, name resolution is phase shifted to runtime since at that point the dynamic type of the receiver has become its static type.

The rule for the static case is defined as usual. The term $R \bullet m(S)$ as $T \rightsquigarrow M$ encodes the member lookup and overload resolution for method `M` that finds the code `M` to call when the receiver has static type `R` and the argument has static type `S`.

$$\frac{\Gamma \vdash e \text{ As } R \rightsquigarrow E, \Gamma \vdash a \text{ As } S \rightsquigarrow A, R \bullet m(S) \text{ As } T \rightsquigarrow M}{\Gamma \vdash e.m(a) \text{ As } T \rightsquigarrow M(E, A)}$$

In the late-bound case, when the receiver has type `Object` and the previous rule does not apply, we cannot do the member look up and overload resolution at compile time, so instead we defer this to the `LateCall` function, passing it the name of the method, the receiver, and the actual argument.

$$\frac{\Gamma \vdash e \text{ As } R \rightsquigarrow \text{Object}, \Gamma \vdash a \text{ As } \text{Object} \rightsquigarrow A}{\Gamma \vdash e.m(a) \text{ As } \text{Object} \rightsquigarrow \text{LateCall}("m", E, A)}$$

At runtime, when executing the `LateCall`, we lookup the dynamic types of the receiver and the argument and do the member lookup at runtime to find the code `M` that actually needs to be called:

$$\frac{r.GetType() \rightarrow R, a.GetType() \rightarrow S, R \bullet m(S) \text{ As } T \rightsquigarrow M}{\text{LateCall}("m", r, a) \rightarrow M(r, a)}$$

In some sense, late binding in Visual Basic gives you a form of multi-methods since late calls are resolved based on the dynamic types of *all* their arguments.

4.3 More Dynamism

Looking at the type rule for late binding, it is clear that there is no reason that for late-bound calls, the method name needs to be statically determined. Visual Basic 9 therefore allows late-bound identifiers of the form $e.(m)(a)$ where m is any expression whose type is convertible to string. Note that dynamic identifiers make it quite easy to define a meta-circular interpreter for Visual Basic since $[e.m(a)] = [e].("m")(a)$. This kind of interpretation is extremely useful for data-driven test harnesses [43].

In a future version of Visual Basic, we also hope to allow the replacing of constants by variables in any place where the runtime infrastructure allows us to compute these at runtime, making Visual Basic as dynamic as possible given the limitations of the CLR.

Another aspect in which Visual Basic differs from statically typed languages such as C# and Java, is that the Visual Basic compiler inserts downcasts automatically, and not just upcasts. We are using this ability to relax the creation of delegates in such a way that you can create a delegate of type `Func(Of A,R)` from any function f that can be called with an actual argument of type A and assigned to a variable of type R ; we simply do this by creating a stub `Function(X As A) CType(f(X), R)` of the exact type required by the delegate.

Just like we extended late binding over normal object to XML via axis members, we are also planning to provide a similar mechanism for late-binding over deeply embedded relational data (that is ADO .Net DataSets). This form of late-binding is extremely useful to support developers of generic reporting, viewing, analysis, intelligence, visualization, and data-mining tools.

4.4 Type-System Extensions

One of the nice things about the relational model is the fact that relationships are external. That is, children point to the parents (foreign key \rightarrow primary key relationship) as opposed to from the parent to the child. As a result, it becomes possible to create explicit relationships [13] between types after the fact, without modifying the participating types. This is important when we want to relate data from different sources. For example relating descriptions of CDs from a web-services with my personal CD collection in iTunes. By adding support for explicit relationships in the language we can navigate such relationship via the familiar dot-notation instead of having to perform complicated joins using middle tables.

Another proposed extension that aids dynamism are dynamic interfaces that make it possible to implement an interface on an existing type, much like in Haskell we can create an instance of a given type for a type class independent of the definition of that type.

4.5 Contracts

Current static type systems as found in contemporary object-oriented languages are not expressive enough. They only allow you to specify the most superficial aspects of the contract between the caller and the callee. From a program specification point of view, our programs are extremely dynamically typed!

What we really need is a dial that we can turn from no static typing on the one extreme, to traditional static typing, to full contracts and invariants [5, 4] on the other extreme.

4.6 Concurrency and Transactions

One aspect of distributed data-intensive applications that we have not yet mentioned is the distributed part. We need to address concurrency not only for that reason, but also because the advent of multi-core processors will put highly parallel machines on the desktops of normal people. We believe that transactions [22] are the only way ordinary people can deal with concurrency, and hence we are very interested in investigating language support for transactions and transactional memory.

For more advanced scenarios that require complex synchronization patterns, we believe that $C\omega$ style join patterns [11] remain very attractive.

It is also interesting to see a resurging interest in morphisms and program transformations in the context of massively parallel computing [19].

In any case, bringing concurrency to the masses is one of the topics that is high on our agenda.

5. Enlightenment?

Transferring technology from research to the mainstream requires solving all research problems, leaving implementation as just a matter of engineering. While the goal of research is to push the envelope as hard as possible, the role of product development is to pick and choose from that envelope and simplify the contributions as much as possible, but not more.

It necessarily takes a long time for research ideas to surface in the real world. The reason is simply that it takes time for the really good ideas to float up and mature and for the bad ideas to sink down and whither away.

There is one aspect of the impedance mismatch between research and practice that I did not know how to solve, which is that in practice most effort goes into the "noise" that researchers abstract away from in order to drill down to the core of the problem.

Even though lambda expressions, meta programming, monads, and comprehensions have been around for many decades, it is rather remarkable that they show up in mainstream languages such as C# 3.0 and Visual Basic 9 and the LINQ framework. It is especially remarkable since, in the functional programming community, monads are not yet mainstream and by many considered to cause brain damage.

Functional programming has finally reached the masses, except that it is called Visual Basic instead of Lisp, ML, or Haskell.

Acknowledgments

The list of people to thank would be extremely long, and even then I would run the risk of forgetting someone. Instead of inadvertently stepping on someone's toes, let me instead say that the only reason a dwarf like me can see this far is that I have been standing on the shoulders of giants.

All things are subject to change, and nothing can last forever. Look at your hand, young one, and ask yourself, "Whose hand is this?" Can your hand correctly be called "yours"? Or is it the hand of your mother, the hand of your father. Reflect on the impermanent nature of your hand, the hand that you once sucked in your mother's womb. [14]

References

- [1] <http://caml.inria.fr/>
- [2] <http://haskelldb.sourceforge.net/>
- [3] <http://research.microsoft.com/fsharp/about.aspx>
- [4] <http://research.microsoft.com/specsharp/>
- [5] <http://www.eiffel.com/>
- [6] <http://www.haskell.org>
- [7] <http://www.jclark.com/xml/xmlns.htm>
- [8] <http://www.w3.org/xml/schema>
- [9] *The Fun of Programming*. Cornerstones in Computing. Palgrave, 2003.
- [10] The Haskell 98 Foreign Function Interface 1.0, 2003.
- [11] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern Concurrency Abstractions for C#. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804, 2004.
- [12] Gavin Bierman, Erik Meijer, and Wolfram Schulte. The essence of Data Access in C#. In *ECOOP*, volume 3586 of *LNCS*. Springer-Verlag, 2005.
- [13] Gavin Bierman and Alisdair Wren. First-Class Relationships in an Object-Oriented Language. In *ECOOP*, volume 3586 of *LNCS*. Springer-Verlag, 2005.
- [14] Matthew Bortolin. *The Dharma of Star Wars*. Wisdom Publishers Inc., Boston, 2005.
- [15] Gilad Bracha. Pluggable Type Systems. In *OOPSLA Workshop On The Revival Of Dynamic Languages*, 2004.
- [16] Niklas Broberg. Haskell Server pages Through Dynamic Loading. In *Haskell Workshop*, 2005.
- [17] Magnus Carlsson and Thomas Hallgren. FUDGETS - A Graphical User interface in a Lazy Functional Language. In *FPCS*, 1993.
- [18] James Cheney and Ralf Hinze. First-Class Phantom Types. Computer and Information Science Technical Report TR2003-1901, Cornell University, 2003.
- [19] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.
- [20] Matthew Fluet and Riccardo Pucella. Phantom Types and Subtyping, 2004.
- [21] Matthew Fluet and Riccardo Pucella. Practical Datatype Specializations with Phantom Types and Recursion Schemes, 2005.
- [22] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.
- [23] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.
- [24] Nigel Perry Jason Smith and Erik Meijer. Mondrian for .NET. *DDJ*, 2002.
- [25] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, March 1966.
- [26] Daan Leijen. Functional Components: COM Components in Haskell. Master's thesis, Department of Computer Science, University of Amsterdam, september 1998.
- [27] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *2nd USENIX Conference on Domain Specific Languages (DSL'99)*, pages 109–122, Austin, Texas, 1999. Also appeared in *ACM SIGPLAN Notices* 35, 1, (Jan. 2000).
- [28] Daan Leijen, Erik Meijer, and James Hook. Haskell as an Automation Controller. In *The 3rd International Summerschool on Advanced Functional Programming*, volume 1608 of *LNCS*. Springer-Verlag, 1999.
- [29] Sheng Liang. *Modular Monadic Semantics and Compilation*. 1997.
- [30] Sheng Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [31] Erik Meijer. Server-Side Web Scripting in Haskell. *Journal of Functional Programming*, 10(1):1–18, january 2000.
- [32] Erik Meijer and Brian Beckman. XLINQ: XML Programming Refactored (The Return Of The Monoids). In *XML*, 2005.
- [33] Erik Meijer and Koen Claessen. The Design and Implementation of Mondrian. In *Haskell Workshop*, 1997.
- [34] Erik Meijer and Peter Drayton. Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages. In *OOPSLA Workshop On The Revival Of Dynamic Languages*, 2004.
- [35] Erik Meijer and Sigbjorn Finne. Lambda, Haskell as a Better Java. In *Haskell Workshop*, 2000.
- [36] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional Programming with Bananas, Lenses, Envelopes, and Barbed Wire. In *FPCA*, volume 523 of *LNCS*. Springer-Verlag, 1991.
- [37] Erik Meijer, Daan Leijen, and James Hook. Client-side Web Scripting with HaskellScript. In *PADL*, volume 1551 of *LNCS*, pages 196–210. Springer-Verlag, 1998.
- [38] Erik Meijer, Nigel Perry, and Arjan van Yzendoorn. Scripting .NET Using Mondrian. In *ECOOP*, volume 2072 of *LNCS*. Springer-Verlag, 2001.
- [39] Erik Meijer and Wolfram Schulte. XML Types for C#. BillG ThinkWeek Submission Winter 2001.
- [40] Erik Meijer, Wolfram Schulte, and Gavin Bierman. Programming with Circles, Triangles and Rectangles. In *XML*, 2003.
- [41] Erik Meijer, Wolfram Schulte, and Gavin Bierman. Unifying Tables, Objects and Documents. In *DP-COOL*, volume 27 of *John von Neumann Institute of Computing*, 2005.
- [42] Erik Meijer and Mark Shields. XMLambda: A Functional Programming Language for Constructing and Manipulating XML Documents. Unpublished draft.
- [43] Erik Meijer, Amanda Silver, and Paul Vick. Overview Of Visual Basic 9.0. In *XML*, 2005.
- [44] Erik Meijer and Danny van Velzen. Haskell Server Pages: Functional Programming and the Battle for the Middle Tier. In *Haskell Workshop*, 2000.
- [45] Thomas Nordin and Simon Peyton Jones. Green Card: a Foreign-language Interface for Haskell. In *Proceedings of the Haskell Workshop*, 1997.
- [46] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, Massachusetts, 2002.
- [47] Riccardo Pucella, Erik Meijer, and Dino Oliva. Aspects de la Programmation d'Applications Win32 avec un Langage Fonctionnel, 2004.
- [48] Mark Shields and Erik Meijer. Type-Indexed Rows. In *POPL*, 2001.
- [49] Erik Meijer Sigbjorn Finne, Daan Leijen and Simon Peyton Jones. H/Direct: A Binary Foreign Language Interface for

Haskell. In *ICFP*, 1998.

- [50] Erik Meijer Sigbjorn Finne, Daan Leijen and Simon Peyton Jones. Calling Hell from Heaven and Heaven from Hell. In *ICFP*, 1999.
- [51] Erik Meijer Simon Peyton Jones and Daan Leijen. Scripting COM Components in Haskell. In *Software Reuse*, 1998.
- [52] Peter Thiemann. WASH Server Pages. In *FLOPS*, 2006.