Confessions of a Used Programming Language Salesman Getting the Masses Hooked on Haskell

Article *in* ACM SIGPLAN Notices · October 2007

CITATIONS
5

READS
1,995

1 author:

Erik Meijer
Delft University of Technology
137 PUBLICATIONS **4,999** CITATIONS

SEE PROFILE

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. [31]

Abstract

For many years I had been fruitlessly trying to sell functional programming and Haskell to solve real world problems such as scripting and data-intensive three-tier distributed web applications. The lack of widespread adoption of Haskell is a real pity since functional programming concepts are key to curing many of the headaches that plague the majority of programmers, who today are forced to use imperative languages. If the mountain won't come to Mohammed, Mohammed must go to the mountain, and so I left academia to move to industry. Instead of trying to convince imperative programmers to forget everything they already know and learn something completely new, the trick is to infuse existing imperative object-oriented programming languages with functional programming features. As a result, functional programming has finally reached the masses, except that it is called Visual Basic 9 instead of Haskell 98.

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Applicative (functional) languages, Object-oriented languages

General Terms Languages, Theory, Human Factors

1. Introduction

Nearly all (business) applications 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: **R**elations in the data tier + **O**bjects in the business tier + and **X**ML in the presentation tier, i.e. the infamous **ROX** triangle [103].

Not only is each data model fundamentally different, but each comes strongly coupled with its own programming language. Tabular data with SQL queries, objects with imperative languages such as Java, C♯, or Visual Basic, and XML trees with XPath, and XQuery or XSLT. 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 provide 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 unrelated special cases. conveniently, the same theories also form the foundation of functional languages, in particular those of the pure and lazy kind such as Haskell.

This paper is a personal account of my journey to democratize the three-tier distributed programming problem using (lazy) functional programming concepts. It starts

with my naive attempt to use Haskell as the language to write three-tier distributed data intensive applications, then continues with my brief flirtation with designing a series of completely new languages from scratch including Mondrian, XMλ, and Cω, and ends when accidentally discover the Change Function [37] and finally succeed by incorporating functional programming and monads into the LINQ framework, C$\sharp$ 3.0 and Visual Basic 9.

2. The Great Internet Hype 1.0

In 1997 I was fortunate enough to spend a sabbatical at the Oregon Graduate Institute. This was the zenith of the Internet bubble and we got inspired to apply Haskell, the world's finest imperative language [53], to solve the ROX problem using pure functional programming. This abruptly ended my "banana man" [74] period and instead of looking for practical applications of beautiful theory, I now started looking for beautiful theory to solve practical problems [88].

2.1 Interfacing Haskell to the Outside World

One of the premises of Landin's seminal paper "The Next 700 Programming Languages" [60] is that most languages can be considered as a collection of primitive building blocks plus glue to compose smaller programs into larger programs. Haskell's lazy evaluation and monads make Haskell a very powerful glue language [50, 49], but for a long time it was hard to access externally implemented libraries in Haskell, so there was little to glue together. As a result, most Haskell programs lived in a closed world (with notable exceptions such as [35]).

The first thing we needed to tackle was making it dead simple to interface Haskell with external, imperative, code. Interfacing Haskell to native C libraries requires at the lowest level just four things:

1. We need to put imported native pointers under control of the Haskell garbage collector.
2. Dually, we to pin exported Haskell references such that they are not moved around by the Haskell garbage collector.
3. We need to wrap imported native function pointers (either statically or dynamically loaded) as Haskell closures.
4. Dually, we need to export Haskell closures (either statically defined or dynamically created) as native function pointers.

This basic functionality is available as the standard Foreign Function Interface (FFI) for Haskell 98 [22]. Having a basic FFI mechanism theoretically enables interop between Haskell and other languages, but it requires 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. This is were tooling is dearly needed.

2.1.1 GreenCard and HaskellDirect

The first attempt at making FFI easy to use was Greencard [87]. Programmers put special directives in their code, and the GreenCard preprocessor then generates all the low-level interfacing and marshalling boilerplate code based on these directives.

H/Direct [98, 99] was an attempt to automatically generate FFI boilerplate code from a standalone 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 binary software components were the silver bullet and so it made sense to use a (more or less) language-*independent* external description for components.

2.1.2 COM

COM is a binary-component standard that, at its core, shows a minimalist design that 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. In particular COM does not impose a "common" type system. COM components are simply vtables of function pointers that implement a set of interfaces (identified by GUIDS) and inherit from a base interface `IUnknown`. The `IUnknown` interface determines the identity of a component and defines the first three entries of each COM vtable: `QueryInterface`, `AddRef` and `Release`. The `QueryInterface` method allows navigating between the set of interfaces implemented by the current component, the `AddRef` and `Release` methods allow the current component to participate in a reference-counting garbage collection protocol.

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 programming language that intrinsically supported COM, gave it a bad reputation. Nevertheless, even after more than 10 years, we still believe that COM is a great component model. And we are not the only ones; on the Windows platform, COM is alive and kicking. On Linux, the cross-platform COM model (XPCOM) is the basis for many open source projects such as Mozilla.

One interesting artifact that arose from interfacing Haskell

[100] and SML [92] 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 [45, 36, 43, 42].

We used phantom types in a variety of different ways. The most direct use is to represent typed pointers using a type synonym `Ptr a = Addr`. The unused type parameter a allows us to distinguish between a pointer to an integer `Ptr Int` and say a pointer to a pointer to an integer `Ptr (Ptr Int)`. We also heavily used phantom types to model interface inheritance [62].

2.1.3 Automation and MS Office

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. Many of Microsoft's applications, such as Word, Excel, Outlook, Powerpoint, and Visio, are all fully scriptable using Automation, and the pre-.NET Visual Basic dialects such as VBA and VB6 are programming-language abstractions over COM Automation.

Because of the additional level of indirection introduced by Automation, it is extremely simple for *any* language to use and create COM components. The language infrastructure just has to provide a generic binding for the single COM interface `IDispatch`. From there on programmers can access any Automation component. Similarly, the language only needs to provide one generic factory method that wraps a collection of function pointers into an `IDispatch` interface. From there on programmers can easily create new Automation components [64].

In our experience Haskell was an ideal language for COM Automation, for instance within half a day we scripted Visio to create a visual front end for Hawk [61] and we created an elegant combinator language for scripting MS Agents [77]. So we naturally assumed that if we exposed Automation in Haskell, the world would instantly fall for functional programming.

2.2 Server-side Scripting Using Haskell

Since the Internet bubble was still expanding in the late 90's, another 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.

2.2.1 Perl for Swine

The *Perl For Swine* library [68] leverages algebraic data types and higher-order functions of Haskell to create an embedded domain-specific language of HTML-generating combinators, and a worker-wrapper-style abstraction of the actual CGI protocol. The user writes a simple worker function of type `[(String, String)] -> IO HTML`, which gets enveloped into an `IO ()` action that parses the query string of the incoming HHTP request into a list of name-value pairs passes it to the worker and wraps the result using the proper MIME type.

In addition to the basic CGI library, we also integrated the Hugs interpreter (`mod_haskell`) into the Apache Web server.

2.2.2 Haskell Server Pages (HSP)

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

Systems such as ASP and PHP facilitate this form of 'templated' Web site development by means of special static HTML pages with embedded *holes* the dynamically generated pieces of the page. The implementation of ASP and PHP is just a simple preprocessor that turns each line of HTML into a `Response.Write` statement and leaves embedded code as-is. This results in a non-compositional model where it is impossible to arbitrarily nest further HTML inside code, and code inside that HTML, etc.

Haskell Server Pages [84] takes the idea of HTML templates a step further by expanding HTML literals embedded in arbitrary Haskell expressions 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 [102], and as the MSc thesis of Niklas Broberg [34]. The PLT Schemework in the area of web (both client and server) programming is truly impressive [18].

2.2.3 HaskellDB

As mentioned earlier, almost all web applications involve dealing with relational database in form form or another. HaskellDB [63] is a domain-specific library for programming against relational data. Just like the HTML combinators, most domain-specific languages are implemented via *shallow* embedding [111]. That is, we define a set of base combinators that embody the semantic algebra of the embedded language, and glue the primitive operations together into bigger denotations using the host language's abstraction mechanism.

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. In a second step we evaluate these trees, or compile them into an external target language for execution.

The `Query a` monad in HaskellDB is a modified state monad where the state contains an explicit representation of the query that is being executed. The function `runQuery :: Query a -> IO [a]` compiles the this representation into SQL, submits it to a back-end database, and returns a collection of rows as the result. When executing the SQL, the database does not know, or have to know, that this SQL was generated by evaluating a Haskell comprehension. HaskellDB has lingered for a long time, but recently it has been revisited and improved [6].

A similar approach is used by Lennart Augustsson to generate VBA programs to script Excel for financial modeling, the hardware description language Lava [30] that generates VHDL or other external hardware description languages and the image generation library Pan [39].

2.3 Client-Side Scripting Using HaskellScript

With the browser wars were in full swing, Microsoft wanted to support both JavaScript as well as VBScript in its Internet Explorer browser. Any script engine that implements the *ActiveX Script Engine* interfaces can be embedded into a host application that implements the *ActiveX Script Host* interfaces. Examples of hosts are Internet Explorer, Windows shell, and the IIS Web server.

HaskellScript [77] is an implementation of the ActiveX Script Engine interfaces for Haskell. It uses a custom COM interface, `IScript`, layered on top of the Hugs interpreter. In our experience Haskell was an excellent language for DHTML and shell scripting, but despite this, HaskellScript reached the astronomical popularity of dynamically typed scripting languages such as Python, Lua, and most recently Ruby.

2.4 Lambada

Soon after we made our bet on H/Direct, COM and COM Automation, binary component models fell out of fashion in favor of language-*specific*, meta-data driven, models such as Java and .NET. Also, we speculated that one of reasons for the low adoption rate of our work might be the fact that the technology was Windows-specific. So, we tried to sell Haskell as the ultimate component glue language a second time by interfacing Haskell to Java [73] via JNI [66][1].

Also the Haskell-Java binding failed to gain any traction within the Haskell community, and we began exploring the insane idea of creating new languages for the Web from scratch.

2.4.1 XMλ

With HSP we already veered outside the strict boundaries of pure embedded domain-specific languages and made baby steps towards adding new syntax to Haskell. The HaskellDB experiment convinced me that the Haskell type-system is incredibly expressive, but not quite expressive enough to easily encode the idiosyncrasies of XML Schema and DTDs [101]. So it seemed obvious to design a stand-alone language, XMλ [82], to filter, query, pattern match, and transform XML documents. The language turned out to be an "Edsel".

XMλ featured a complicated DTD-based static type inference system that supported polymorphism and higher-order functions. As a result of this complexity we did not even succeed to get the implementation beyond a very early prototype stage. That did not prevent us from looking at even more advanced type systems. Subsequent work on type-indexed rows [96] marked the lost weekend of my typoholic years. And it slowly started to dawn on me that you can overdo static typing. Unfortunately, I had not yet hit bottom.

2.4.2 Internet Scripting Using Mondrian

Inspired by the conceptual minimalism of the "De Stijl" movement, Mondrian [71, 78, 52] was an experiment to reduce Haskell to its bare essence: higher-order functions, lazy evaluation, pattern matching, and monads; and for the rest, piggybacking as much as possible on the underlying object-oriented .NET framework. Just like modern

[1] Sheng Liang, one of the designers of JNI, worked on monadic interpreters in a previous life [65].

art does not evokes strong positive feelings in most people, the Mondrian language failed to seduce the functional language community at all, let alone the non-FP folks I was ultimately trying to win over.

In many respects F♯ [8] is the moral successor of Mondrian, except that it uses the strict and imperative functional language OCaml [1] as its basis. By dropping laziness in strict functional languages such as Scheme[55], SML[85], OCaml[1], Scala [19] and F♯, the purity and semantic beauty of true functional programming is lost. When programming in Haskell, laziness is one of those things that you rely on all the time without noticing it; it is something you only realize once you miss it.

2.4.3 Lazy versus Strict

Strictness *is* an effect, and most strict functional languages usually include other side-effects such as exceptions, imperative updates, and continuations. Semantically, therefore there is no real difference between a strict, higher-order, functional language, and a strict, higher-order, object-oriented language. One might argue that OO languages are actually more powerful than strict functional languages since objects are closures that contain multiple functions (methods) [104] while inheritance and virtual methods correspond to open data types and functions [67].

One of the most useful benefits of a pure language like Haskell is that initial algebras and final co-algebras coincide [76, 44]. This means that Haskell data structures can finite, infinite, or both. This means that in Haskell there is no problem to define a recursive list such as `nat = 0 : map (+1) nat`. In strict languages, values of normal data types or classes (initial algebras) are always finite, and the previous recursive list definition would not terminate. To define potentially infinite values you have to resort to co-algebras, or in layman's terms you have to implement an explicit pull model. The prototypical example of a co-algebraic, or pull-model, data type is the `IEnumerator` interfaces (in Java `iterator`)

```
interface IEnumerator<T>
{
   bool MoveNext();
   T Current;
}
```

The IEnumerator interface is an imperative version of the `unfoldr :: (a -> Maybe (b,a)) -> a -> [b]` function in Haskell where the state component a is implicit and the result `Maybe (b,a)` is simulated by first calling `MoveNext`, which side-effects `Current` when it succeeds. Precisely because of laziness ("deferred execution"), the `IEnumerator` interface and its factory interface `IEnumerable` play a key role in the LINQ framework described below.

There are many examples of blending object-oriented and strict functional languages including C♯ 3.0 and Visual Basic 9, F♯, Scala, Java 7 (for which there are no less than 4 proposals to add closures) [11, 2, 3, 16]. Reconciling lazy and strict functional languages, however, is still an open research problem [54, 94, 109], and will remain so for a long time. Until that moment, pure lazy functional languages such as Haskell will remain a niche. The perceived pain of adoption for purity is simply to high compared to that of strict functional/OO languages.

3. The Change Function

After failing to sell solving real world problems to the Haskell community, as well as failing to sell Haskell to the real world, I started to wonder whether pushing for pure functional languages was actually the right strategy. Maybe I thought, I should sell my soul to the most popular programming paradigm, objects, and to the company that has the biggest market share, Microsoft. In retrospect, what happened, is that I unknowingly discovered the *Change Function* [37].

The Change Function is a simple theory that predicts the success of new technology. Producers of new technology are overly optimistic and think "Build it and they will come". Their conviction is that their technology is superior to anything that already exists. This is reflected in claims such as "Haskell programs are at least 10 shorther than the corresponding C programs". Moreover, they believe their currently inadequate implementations will radically improve over time. This is reflected in claims such as "Soon, real soon, the speed of Haskell programs will approach that of highly optimized C code". In short, they predict the chance of success as a function of the following product:

$$Supplier{-}centric\ Adoption\ Model = F(10X\ better * Moore's\ Law)$$

Obviously, I had been trapped into the supplier-centric point of view.

Normal users have a very different take on new technology. They hate technology for technology's sake (probably the display on your VCR is also still blinking). They just want to get their work done with the least amount of effort, and moreover, they have little desire to learn new things. According to Pip Coburn therefore, the chance actual chance of success for new technology is a function of the following

ratio:

$$Change\ Function = F(Perceived\ crisis/Perceived\ pain\ of\ adoption)$$

By focusing on finding beautiful solutions for real problems I had already nailed the numerator portion of the Change Function, from now on my goal in life would be to also drive the denominator down to zero to maximize my chance of success.

4. Microsoft

My main reason to join Microsoft as the company to infuse mainstream programming languages with monads and other functional programming features was that I believed the quickest road to success would be to work on adding support for exotic language features to the new *Common Language Runtime* [75].

4.1 Cω and The Dark XML Ages

While I was working in the CLR team on relaxed delegates and lightweight code generation, XML kept gaining momentum. The time was ripe to pitch to chairman Bill Gates himself the idea of making XML a first-class citizen in C♯. With much encouragement from Don Box (who convinced us to drop DTDs and adopt XSD instead), Wolfram Schulte and I submitted a Thinkweek paper [79] on this topic.

As it turned out, BillG liked our proposal, and the XML team led by William Adams funded started a small incubation team to write a prototype data-centric C♯ extension with support for relational tables and XML. The language was originally called X♯, along the way the # suffix became verboten so we changed the codename to Xen. When we joined efforts with the Polyphonic C♯ group [24] in Cambridge, we finally settled on Cω [81, 80, 27].

4.1.1 The Sucking Black Hole of XSD

To appreciate the difficulties of XML let us now make a short excursion to the world of XML schema [21], which must be one of the most complex artifacts invented by mankind. The complexity of XSD is baffling since the existing solution, DTDs, is really perfectly fine solution.

The main problem of XSD is not that it is gratuitously uses XML as its *concrete* syntax, but the fact that it is completely over-engineered. The most confusing feature of XML is the notion of `complexType`. The example below defines a schema for values 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 shape of the members of regular types; perhaps interfaces come closest to this idea. 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 wishful interpretation of mapping `complexType` to classes, the schema above would translate to the following class:

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

This immediately begs the question what to do with the declaration `<element name="Point" type="PointType"/>`. That has to be mapped to a type as well because it represents `Point` documents. Now some elements are mapped to types and some are mapped to fields, and this discrepancy causes an incoherence with the semantics of XPath where all path selections return collections of elements, never of complex types

We could go on for pages talking about the subtleties of mapping XSD to objects [58, 57, 59], 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.

4.1.2 Type-System Extensions

When you throw the challenge of defining an XML-centric language at an academic language designer, their first reaction is to invent a cleaner version of XSD with lots of fancy structural (regular expression) types and elaborate subtyping rules [48, 25, 47, 20]. In very much the same spirit, in Cω we attempted to extend the CLR type system

with a large arsenal of structural types:

$$
\begin{array}{rcl}
T & ::= & N \mid T\texttt{[]} \\
& \mid & T\texttt{(}\ldots\texttt{,}T\texttt{,}\ldots\texttt{)} \\
& \mid & T \texttt{|} T \mid T\texttt{\&}T \\
& \mid & T\texttt{-} \mid T\texttt{!} \mid T\texttt{?} \mid T\texttt{+} \mid T\texttt{*} \\
& \mid & \texttt{struct \{}\ldots\texttt{,}T[m]\texttt{,}\ldots\texttt{\}}
\end{array}
$$

These new structural types include function types T(...,T, ...); union | and intersection types &; an exotic family of stream types, - for exact types (dynamic type equals static type), ! for non-null types (streams with exactly one element), ? for optional types (streams with either zero or one element), + for non-empty streams, and * for possibly empty streams; and optionally labeled records struct { ..., [T] m , ...}.

XML literals in Cω are just serialized objects, and the compiler translated such literals into constructor calls of the type denoted by the literal. For example, we can define a schema for email messages written in XML syntax 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
}
```

Just like Haskell Server Pages, Cω XML literals can contain arbitrarily nested expression and statement holes. Cω also has type inference for local variables, and the inferred type for the msg variable is Email.

4.1.3 Generalized Member Access

The slogan of Cω is "The Power Is In The Dot!" which refers to the fact that in Cω we 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; } to map the lambda expression function(Button it){ return it.BackColor; } over the collection bs.

4.1.4 Query Comprehensions

Besides generalized member access and explicit lifting, Cω also supports XPath-style filter expressions such as buttons[it.BackColor = Color.Red], using it to bind the implicit parameter of the filter predicate and SQL-style SELECT FROM WHERE GROUPBY comprehensions. The compiler has built-in knowledge about queries over streams (list comprehensions) and about queries over remote databases (the query monad). It was possible to overload the comprehension syntax for other types via compiler plug-ins.

4.1.5 Nullable types

In the Whidbey version of the .NET Framework, nullable types were introduced in C$^\sharp$ 2.0 using the same ? syntax as Cω. In C$^\sharp$ 2.0, conversions and binary operators over T? are lifted, but 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? interpreted as streams of at most one element to T* c.q. IEnumerable<T> which represent stream of zero or more elements.

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. Without going through such a two step process, boxing the null value of a value type T would not be result in the null pointer, i.e., (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.

5. Language Integrated Query

As the Cω incubation was winding down, the C$^\sharp$ team started to spin up the design work for C$^\sharp$ 3.0, and several ex-members of Cω and ObjectSpaces teams went over to C$^\sharp$ to spread their intellectual DNA and conceive LINQ.

The goal of **L**anguage **In**tegrated **Q**uery (LINQ) is to unify programming against relational data, objects, and XML. In contrast to my earlier efforts, this time we managed to strike a nice balance between providing domain-specific libraries and adding general purpose language extensions. Surprisingly, LINQ requires no extensions to the underlying CLR and all new C$^\sharp$ (and Visual Basic) features are defined

as syntactic sugar on top of C♯ 2.0 [28]. Both the libraries and the language extensions are heavily inspired by functional programming, in particular the Haskell (list) prelude and monads and monad comprehensions [108]. All of the various language extensions such as lambda expressions, anonymous types, type inference, meta-programming, etc. are valuable by themselves. The total is really more than just the sum of the parts.

At the same time, I rekindled up my interest in scripting and dynamic languages [72], 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, following the approach suggested by Gilad Bracha [32]. In turn, this led me to the realization that Visual Basic was the ideal language for the road ahead because it is the only widespread language (yet) that allows static typing where possible and dynamic typing where necessary. So besides joining the C♯ design team, I also joined the Visual Basic design team (see §5.8.2 and §6) and changed the title on my business card [5].

5.1 Query Comprehensions

Central to LINQ is the introduction of *query comprehensions* into both Visual Basic and C♯. query comprehensions are a generalization of Haskell's monad (or list) comprehensions to include support for SQL-style operations such as joins, grouping and aggregation, and sorting. For example, the following query filters and sorts a collection of programmers by `Age` and `Name`, groups them by their favorite `Language` and aggregates the size and names of each group

```
var q =
  from p in programmers
  where p.Age > 20
  orderby p.Name descending
  group p.Name by p.Language into g
  select new{ Language = g.Key
            , Size = g.Count()
            , Names = g
            }
```

Just like Haskell, comprehensions are convenient syntactic sugar on top of a specific design pattern. In Haskell the pattern is defined via the `Monad` and `MonadPlus` type classes for monad comprehensions (and using `concatMap` for list comprehensions, which is the `>>=` for the list monad), LINQ defines a set of *standard sequence operators* for this purpose (see §5.2). A possible translation of the above query into the underlying standard sequence operators could look like:

```
  var q = programmers
    .Where(p => p.Age > 20)
    .OrderByDescending(p => p.Age)
    .GroupBy(p => p.Language, p.Name)
    .Select(g => new{ Language = g.Key
                    , Size = g.Count()
                    , Names = g
                    });
```

With the recent paper [107] on extending Haskell's list comprehensions with grouping and sorting, the circle of influence between Haskell and LINQ has been closed. Other recent languages that support comprehensions are Fortress [10] and Scala [19].

In Visual Basic, comprehensions are design to minimize the need for dropping down to the underlying primitive operators and lambda expressions. Comprehensions are fully compositional, and act as a pipeline that transforms collections of tuples into collections of tuples (similar to the tuple-based translation of XQuery [93]).

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* (which was removed from Haskell 98), 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(APrice, BPrice) < 100
```

In the (mechanical and unreadable) 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(Function(A)
      BarnesAndNoble.Select(Function(B)
         New With { .A = A, .B = B})).
   Where(Function(_It_)
      _It_.A.ISBN = _It_.B.ISBN).
   Select(Function(_It_) New With {
      .Title = _It_.A.Title,
      .PriceA = _It_.A.Price,
```

```
      .BPrice = _It_.B.Price}).
    Where(Function(_It_)
       Max(_It_.APrice, _It_.BPrice)<100)
```

5.2 Standard Query Operators

The higher-kinded shape of a generic type `M<T>` that supports a simplified version of the standard query operator pattern contains the well-know higher-order monadic and list processing functions such as `filter`, renamed to `Where`; `map`, renamed to `Select`; and of course >>= (bind), renamed to `SelectMany`:

```
class M<T>
{
   M<T> Where(Func<T, bool> p);
   M<S> Select<S>(Func<T,S> f);
   M<T> SelectMany<S>(Func<T,M<S>> f);
 }
```

Other standard query operators include `OrderBy`, `GroupBy`, `Join`, `GroupJoin`, `Aggregate` (Haskell's `foldr`), etc.

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 with signature `T Where(this Q src, Func<R,S> p)`. 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>`, see §5.3) and most importantly, the methods can take expressions trees (see §5.7) instead of just delegates.

It is quite surprising that Haskell is one of the very few languages that allows higher-kinded type variables. If, on the term level, parametrizing over functions is useful, doing the same on the level of types sounds like an obvious thing to do. As far as we know Scala is the only other language besides Haskell that also support higher-kinded types [86]

5.3 Extension Methods

In both Visual Basic and C♯, 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♯). In particular, 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♯ 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,...)`.

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<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♯ 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>(
    this 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.

5.4 Object Initializers

Introducing query comprehensions in the language forces us into a much more expression-oriented style than the usual statement-oriented style that people are used to in imperative languages. To facilitate this, both C♯ and Visual Basic introduce *object initializers*, which correspond closely to labeled construction in Haskell. An object initializer such as (using C♯ syntax):

```
var p =
   new Person
      { First="John", Last="Doe" };
```

creates a new instance of `Person` and then assigns values to the `First` and `Last` 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 it. 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", ... }
   }
```

5.5 Anonymous Types

In queries we often want to combine several values by projecting out a subset of their members without having to declare and introduce a new nominal type. This is exactly the reason that functional languages and Cω support tuples and/or labeled *records*. For Java, C♯, and Visual Basic programmers that grew up with monomorphic nominal types where arrays as the sole structural type, the concept of anonymous types is still revolutionary.

In Visual Basic, we create an anonymous record with a `Name` and a `City` member by using syntax similar to that of nominal object initializers:

```
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, neither C♯ nor Visual Basic supports (width or depth) record subtyping.

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

In C♯, equality on anonymous types is defined structurally, that is, the compiler generates implementations of `Equals` and `GetHashCode` based on the structure of the type. To ensure soundness, anonymous types in C♯ are immutable. In Visual Basic, the fields of anonymous types are mutable, except those that are marked with the `Key` modifier. The `Equals` and `GetHashCode` methods are defined in terms of `Key` fields only [106].

5.6 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 monomorphic types, such as `Hashtable`, regardless of whether their "real" type, such as `HashTable<int, List<string>>`, is complex or not. With the advent of generics and anonymous types, values can be typed much more precisely, which makes explicit typing painful. In particular it is painful to have to duplicate the type in in the constructor and at the variable declaration:

```
HashTable<int,List<string>> tedious =
   new HashTable<int,List<string>>();
```

The problem is less pressing in Visual Basic (and C++) that already supports a concise syntax for combining local variable declarations and constructor calls

```
Dim sweet As
   New HashTable(Of Int, List(Of String))()
```

Typically, type inference algorithms look at all uses of a variable and infer a most general type via unification or constraint-solving. 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. Type inference in the presence of overloading and subtyping is a hard problem, and has been a very active research area for many years [90]. The presence of the über type `Object` makes inferred types based on taking least upper bounds degenerate to `Object` pretty quickly. However, using the same variable at two disparate types with `Object` as their closest supertype is most always an error. The situation even gets murkier due to (user-defined) implicit conversions.

Inferring types for function arguments in an object-oriented

language is also non-obvious. For example, what would be the inferred type for the parameter `x` of the lambda-expression `x => x.Foo` when there are multiple types (classes or interfaces) in scope that have a `Foo` property, each of which can have a different return type. Haskell type classes artificially avoid this problem by insisting that a method cannot be defined in different classes in scope.

The 80/20 solution is to infer types only from the initializer expression of just local variable declarations and never invent new types during inference. 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. The pragmatic choice we made is a classic example of "worse is better" [4]. However, programming language researchers should keep pushing for "the right thing" [8, 19, 26, 14, 10].

5.7 Expression Trees

One of the biggest hassles of deep embedding is creating representations of embedded programs with bound variables. Because Haskell 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 simply 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. Pushing the burden upon the consumer is 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 Visual Basic 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 MSIL 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 MSIL 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<Func<..>>` 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.

5.7.1 LINQ-to-SQL

LINQ-to-SQL is a domain-specific library for accessing relational data that makes heavy use of expression trees by implementing the standard query operators on the `IQueryable` interface that mirrors those defined on `IEnumerable` by using `Expression (< Func< ...>>` instead of delegates `Func<...>`. Just like HaskellDB, the LINQ-to-SQL infrastructure then compiles these expression trees into SQL and creates objects from the result of running the query on a remote database. LINQ-to-SQL 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.

5.8 XML integration

Ideally, XML should just be a wire-serialization format, and therefore completely hidden from the programmer. However when developers are forced to deal with raw XML, it should be as convenient and simple as possible to create, manipulate, and query XML. For this reason, LINQ introduces LINQ-to-XML [69] to replace the standard W3C DOM. On top of LINQ-to-XML, Visual Basic 9 supports deep embedding of XML via XML literals, and special axis member syntax that abbreviate the most common access patterns to select child elements and attributes.

In combination with query comprehensions the XML support in Visual Basic largely eliminates the need for big standard special-purpose XML-manipulation languages such as XQuery. As always happens, XQuery started as a "declarative" domain specific language, but is evolving into a full-fledged imperative programming language anyhow [56]. Time will tell if users are willing to make the effort of abandoning existing OO languages and switch to XML and XQuery, or that they rather stick with what they already know extended with what they need to get their job done. Note that XQuery and LINQ share the same monad-based semantic foundation [40, 70].

5.8.1 LINQ-to-XML API

The standard W3C DOM API is document-centric, which means that elements and attributes can only exist in the context of a specific document; elements and attributes are not first-class values. Due to this document-centricity, 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: imperative construction does not fit very well in the expression-oriented style required by LINQ.

Accessing nodes using the DOM is 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.

In LINQ-to-XML elements and attributes are first-class values that are constructed via normal constructor calls (*functional construction*), independent of any particular document context. When an already parented node is added as a child of another parent node, the child node is automatically cloned and reparented.

All XPath axes, such as `Parent`, `Descendants`, `Elements`, `Attributes`, and so on, are available as consistently named (extension) methods on nodes and collections of nodes. The latter, closely reflects member lifting of Cω.

5.8.2 XML Literals

While the LINQ-to-XML API is already a major improvement over the DOM, it is not yet simple enough. On top of LINQ-to-XML's functional construction, Visual Basic allows XML literals that the compiler translates into LINQ-to-XML 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 effectively following LINQ-to-XML calls (in reality the compiler unfolds the constructor calls and immediately generates the underlying imperative statements to add child nodes to the parent):

```
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 (quasi-quotation [55]) 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 document declaration `<? xml version="1.0" ?>` causes the inferred type of the variable `Rock` to be `XDocument`.

Unlike Cω and previously mentioned research-oriented XML-centric programming languages, we do not assume any schema information in order to deal with XML documents. Instead Visual Basic optionally layers[32] XSD schema information on top of the CLR type-system to guide Intellisense in the IDE for XML literal construction and axis members. This type information has no, and cannot have any, impact on the runtime behavior of the program; the underlying LINQ-to-XML API is a completely schema-agnostic XML object model.

5.8.3 Namespaces

We gladly got rid of XSD schema in Visual Basic 9, but there is no way around XML namespaces. As James Clark remarks[17], 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, LaTeX document, etc.

In Visual Basic, there are two ways to declare a namespace prefix: by using a global `Imports <xmlns:prefix=URI>` namespace declaration, or by a local `xmlns:prefix=URI` declaration inside an XML element.

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

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

Global namespace declarations scope over the whole program, while local namespace declarations scope over their embedded elements and attributes, but not inside expression holes.

5.8.4 Axis Members

In Visual Basic, we have introduced special syntax for the three most common XPath axis: `Children`, `Descendants`, and `Attributes`. The child axis `cd.Children("Title")` is written using mnemonic syntax `cd.<Title>` designed specifically to emphasize XML element access as opposed to ordinary member access; the descendant axis `CDs.Descendants("Artist")` is similar, but uses three dots `CDs...<Artist>`; again to emphasize the difference with regular member access; the attribute axis `cs.Attributes("Genre")` is abbreviated as `cd.@Genre`.

Element (and attribute) names are expanded to their fully qualified names based on the closest namespace prefix declaration in scope. Hence in the example above, the fully qualified name of the element `<Y:Year>` is `<{http://www.freedb.org}Year>`. So in order to access that node we must use the global prefix declaration `X` for the same namespace URI and write `CD.<X:Year>`.

6. 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. 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♯ with slightly more verbose syntax. Nothing is further from the truth. Visual Basic is the ultimate language to democratize programming against the Cloud.

6.0.5 Transactions, Joins, and Morphisms

The advent of multi-core processors will put highly parallel machines on the desktops of normal people. Transactions [46] are perhaps the only way imperative programmers can deal with concurrency because transactions allow you them continue to think as if your program runs in isolation. While shared-nothing, message passing, concurrency as supported by Erlang [23] might be 10X better, they require programmers to make a complete paradigm switch and hence according to the Change Function the chance of success is small (both obviously for Erlang greater than zero).

For more advanced scenarios that require complex synchronization patterns, we believe that Cω style join patterns [24, 95] are most attractive. Visual Basic support so-called declarative event handling that allows you to specify a disjunction of events that are handled by a particular handler. For instance, the `Handles` clause below specifies that the method `__Click_Or_Focus` will run when either the `Button.Click`, or the `Textbox.Focus` is received

```
Sub __Click_Or_Focus
   (S As Object, E As EventArgs)
   Handles Button.Click, TextBox.Focus
      ...
End Sub
```

Declarative join patterns are a natural extension of this model where the `When` clause specifies the handler that has to be run when a *conjunction* of "messages" have been received. To the caller messages just look like methods, except that asynchronous message calls return immediately whereas synchronous message calls may block. The `OnePlaceBuffer` below specifies that when both a `Put` and `__Empty` message are queued, the `__Put_And_Empty` handler is fired, and when both the `Take` and `__Contains` messages are queued, the `__Take_And_Contains` handler is fired.

```
Class OnePlaceBuffer(Of T)
   Public Synchronous Put(t As T)
   Public Synchronous Take() As T

   Asynchronous __Empty()
   Asynchronous __Contains(t As T)

   Sub __Put_And_Empty(t As T)
   When Put, _Empty
      Me.__Contains(t)
   End Sub

   Function __Take_And_Contains(t As T) As T
   When Take, __Contains
      Me.__Empty()
      Return t
   End Function

   Public Sub New()
      Me.__Empty()
   End Sub
End Class
```

This should be easy to grasp for programmers that already have a basic understanding of event handling.

It is also interesting to see a resurging interest in morphisms and program transformations in the context of massively parallel computing [38]. The query syntax in Visual Basic 9 was designed to support aggregate comprehensions specifically to enable writing MapReduce-style queries directly in Visual Basic instead of using a special domain specific language such as Sawzall [91].

6.1 Static Typing Where Possible, Dynamic Typing Where Needed

The artificial separation between the supporters of dynamically and statically typed languages is rather unfortunate [72]. In particular, dynamically typed languages miss a great chance to leverage static information about programs that the compiler and tools can leverage. Not writing types does not imply no static types [12]. On the other hand, we should not forget that all statically typed languages need a few drops of dynamism (downcasts, reflection, array bounds checking) to make things run smoothly. Every useful Haskell program somehow relies on `unsafePerformIO`.

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`, member resolution is phase-shifted to runtime since at that point the dynamic type of the receiver has become its static type.

The type rule for the statically typed method invocation is standard. The relation $\texttt{R} \bullet \texttt{m(S) As T} \rightsquigarrow \texttt{M}$ encodes the member lookup and overload resolution for a method `m` to find the code `M` to call when the receiver has static type `R` and the argument has static type `S`.

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

In the late-bound case, when the receiver has type `Object` and the previous rule does not apply, static method resolution failed. Instead of bailing out and reporting a type error, Visual Basic instead defers member lookup and overload resolution to run-time by inserting the `LateCall` function, passing it the name of the method, the receiver, and the actual argument.

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

At runtime, when executing the `LateCall`, we determine 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:

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

Late binding in Visual Basic implements a form of multi-methods since late calls are resolved based on the dynamic types of *all* their arguments.

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 it is allowed to 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`. The compiler inserts casting stub `Function(X As A)CType(f(X), R)` of the exact type required by the delegate.

6.2 Possible Language Enhancements

The intersection between dynamic and static languages has proven to be a fertile ground for language designers [41, 97, 105, 33] and we see a lot of interesting opportunities to improve Visual Basic in this respect.

6.2.1 Dynamic Identifiers

Inspecting the type rule for late binding, it is clear that there is no reason that the method name in late-bound calls needs to be statically determined. Dynamic identifiers allow calls 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 $[\![e.m(a)]\!] = [\![e]\!]\texttt{.("m")}([\![a]\!])$. This kind of interpretation is extremely useful for data-driven test harnesses [83]. In a future version of Visual Basic, we also hope to allow the replacement of constants by variables in any place where the runtime infrastructure allows us to compute corresponding entities at runtime. This would make Visual Basic as dynamic as possible within the limitations set by the underlying CLR.

6.2.2 Explicit Relationships

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 $\mapsto$ primary key relationship) as opposed to from the parent to the child. As a result, it becomes possible to create explicit relationships [29] 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 useful extension that aids dynamism and enables explicit relationships are dynamic interfaces which let you implement an interface on an existing type after the fact. This is much like in Haskell were we can create an instance of a given type for a type class independent of the definition of that type [110].

6.2.3 Contracts

The 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. For instance the type of the `Slice` method below specifies that it takes an array and two integers and returns an array, but it does not convey any *relevant* information at all about the intended semantics of the method; given an array `src`, a start index `i` such that `0 =< i`, an end index `j` such that `i =< j =< src.Length`, the array elements withing the indicated range are copied into a new array:

```
T[] Slice(this T[] src, int i, int j){...}
```

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 [15, 9, 13] on the other extreme.

While we feel that pre- and post-conditions are ready to promote from the research world to the real world, we do not believe that object invariants are ready for prime time yet [89].

7. Conclusions and Final Musings

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

Since lambda expressions, meta-programming, monads, and comprehensions are abstract mathematical concepts, it might seem remarkable that they show up in contemporary languages such as C♯ 3.0, Visual Basic 9, and Java, as well as more experimental hybrid functional/OO languages such as Fortress, Scala, Fortress, and F♯. It is especially remarkable since, in the Haskell community, monads are still considered as one of the most advanced and difficult features to master. The explanation for this apparent contradiction is that the successful programming languages understand developer inertia and obey the Change Function. They offer a solution to deep developer pain with a very low adoption threshold. The reason why most research languages die a quick death and successful languages a slow one is because they do not realize users are in charge and instead promise miracles of improved productivity and automatic correctness, coupled with an unbounded optimism that in the future their implementations will be fast enough thanks to better and more advanced compiler optimizations [7]. Sometimes a research language targets a severe enough crisis that is larger that the perceived pain of adoption, and hence it gathers enough velocity to escape death. A good example of a language in this category is Erlang. Hopefully, Haskell will escape death as well.

Reaching out to real users is a necessary condition for success but by no means sufficient. Successfully transferring technology from research to the mainstream also requires that all research problems be solved upfront, 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 the research results and simplify the contributions as much as possible, but not more. Research should not try to become advanced development, that amounts to eating our own seed corn. Product development should not attempt to do research, that diminishes the chances to ship anything.

It necessarily takes a long time for research ideas to surface in the real world. The reason is simply that it just takes time for the really good ideas to float up and mature and for the bad ideas to sink down and wither away. Skimming the cream from the research results requires patience. It is very hard to plan innovation.

There is one aspect of the divisions between research and practice that I do not know how to solve. In practice most development effort goes into the "noise" that researchers abstract away in order to drill down to the core of the problem. However, it is often in this noise that the hard implementation problems are hidden. For example, most programming language research papers define some for of "featherweight" subset [51] of the full programming language to define and study the properties of the dynamic and static semantics of that language. The irony is that such minimalistic subsets gives rise to a nice and simple formalization, whereas language implementers actually need help formalizing the rough edges of the language, not the beautiful and clean subset. If something is easy to formalize, it is probably easy to implement. If something is hard to formalize, it is for sure *really* hard to implement.

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.* [31]

References

[1] http://caml.inria.fr/

[2] http://docs.google.com/view?docid=ddhp95vd_6hg3qhc

[3] http://docs.google.com/view?docid=k73_1ggr36h

[4] http://dreamsongs.com/worseisbetter.html

[5] http://en.wikipedia.org/wiki/pimp_my_ride

[6] http://haskelldb.sourceforge.net/

[7] http://research.microsoft.com/~toddpro/papers/law.htm

[8] http://research.microsoft.com/fsharp/about.aspx

[9] http://research.microsoft.com/specsharp/

[10] http://research.sun.com/projects/plrg/fortress.pdf

[11] http://www.artima.com/weblogs/viewpost.jsp?thread=182412

[12] http://www.cis.upenn.edu/~bcpierce/types/archives/1988/msg00042.html

[13] http://www.cs.iastate.edu/~leavens/jml/documentation.shtml

[14] http://www.doc.ic.ac.uk/~scd/

[15] http://www.eiffel.com/

[16] http://www.javac.info/closures-v05.html

[17] http://www.jclark.com/xml/xmlns.htm

[18] http://www.plt-scheme.org/publications.html

[19] http://www.scala-lang.org/

[20] http://www.w3.org/tr/xquery/

[21] http://www.w3.org/xml/schema

[22] The Haskell 98 Foreign Function Interface 1.0, 2003.

[23] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[24] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern Concurrency Abstractions for C♯. *ACM TOPLAS*, 26(5):769–804, 2004.

[25] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: an XML-Centric General-Purpose Language. In *Proceedings ICFP*, 2003.

[26] Gavin Bierman. Formalizing and Extending C♯ Type Inference. In *Proceedings FOOL/WOOD*, 2007.

[27] Gavin Bierman, Erik Meijer, and Wolfram Schulte. The essence of Data Access in Cω. In *Proceedings ECOOP*, volume 3586 of *LNCS*, 2005.

[28] Gavin Bierman, Erik Meijer, and Mads Torgersen. Lost in Translation: Formalizing Proposed Extensions to C♯. In *Proceedings OOPSLA*, 2007.

[29] Gavin Bierman and Alisdair Wren. First-Class Relationships in an Object-Oriented Language. In *Proceedings ECOOP*, volume 3586 of *LNCS*, 2005.

[30] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: hardware design in haskell. *SIGPLAN Notices*, 34(1):174–184, 1999.

[31] Matthew Bortolin. *The Dharma of Star Wars*. Wisdom Publishers Inc., Boston, 2005.

[32] Gilad Bracha. Pluggable Type Systems. In *Proceedings OOPSLA Workshop On The Revival Of Dynamic Languages*, 2004.

[33] Gilad Bracha and David Griswold. Strongtalk: typechecking smalltalk in a production environment. *ACM SIGPLAN Notices*, 28(10):215–230, 1993.

[34] Niklas Broberg. Haskell Server pages Through Dynamic Loading. In *Proceedings Haskell Workshop*, 2005.

[35] Magnus Carlsson and Thomas Hallgren. FUDGETS - A Graphical User interface in a Lazy Functional Language. In *Proceedings FPCA*, 1993.

[36] James Cheney and Ralf Hinze. First-Class Phantom Types. Computer and Information Science Technical Report TR2003-1901, Cornell University, 2003.

[37] Pip Coburn. *The Change Function: Why Some Technologies Take Off and Others Crash and Burn*. 2006.

[38] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings OSDI*, 2004.

[39] Conal Elliott, Sigbjørn Finne, and Oege de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(2), 2003. Updated version of paper by the same name that appeared in SAIG '00 proceedings.

[40] Mary F. Fernandez, Jérôme Siméon, and Philip Wadler. A semi-monad for semi-structured data. In *Proceedings of the 8th International Conference on Database Theory*, volume 1973 of *LNCS*, 2001.

[41] Cormac Flanagan. Hybrid type checking. In *Proceedings POPL*, 2006.

[42] Matthew Fluet and Riccardo Pucella. Phantom types and subtyping. *Journal of Functional Programming*, 16(6):751–791, 2006.

[43] Matthew Fluet and Riccardo Pucella. Practical datatype specializations with phantom types and recursion schemes. *Electronic Notes on Theoretical Computer Science*, 148(2):211–237, 2006.

[44] Peter Freyd. Recursive types reduced to inductive types. In *Proceedings LICS*, 1990.

[45] Jeremy Gibbons and Oege de Moor, editors. *The Fun of Programming*. Cornerstones in Computing. Palgrave, 2003.

[46] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers

Inc., San Francisco, CA, USA, 1992.

[47] Matthew Harren, Mukund Raghavachari, Oded Shmueli, Michael G. Burke, Rajesh Bordawekar, Igor Pechtchanski, and Vivek Sarkar. XJ: Facilitating XML Processing in Java. In *Proceedings WWW*, 2005.

[48] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. *ACM TOPLAS*, 27(1):46–90, 2005.

[49] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4es):196, 1996.

[50] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[51] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3), 2001.

[52] Nigel Perry Jason Smith and Erik Meijer. Mondrian for .NET. *DDJ*, 2002.

[53] Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. In *Marktoberdorf Summer School*, 2000.

[54] Simon Peyton Jones, Mark Shields, John Launchbury, and Andrew Tolmach. Bridging the gulf: a common intermediate language for ml and haskell. In *Proceedings POPL*, 1998.

[55] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[56] Donald Kossmann, Michael Carey, Don Chamberlin, Mary Fernandez, Daniela Florescu, Giorgio Ghelli, Jonathan Robie, and Jerome Simeon. Xqueryp: An XML application development language. In *Proceedings XML*, 2006.

[57] R. Lämmel and E. Meijer. Mappings make data processing go 'round — An inter-paradigmatic mapping tutorial. In *Generative and Transformation Techniques in Software Engineering*, volume 4143 of *LNCS*, 2005.

[58] R. Lämmel and E. Meijer. Revealing the X/O impedance mismatch (Changing lead into gold). In *Datatype-Generic Programming*, volume 4719 of *LNCS*, 2007.

[59] Ralf Lämmel and Dave Remy. Functional OO Programming with Triangular Circles. In *Proceedings XML*, 2006.

[60] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, March 1966.

[61] John Launchbury, Jeffrey R. Lewis, and Byron Cook. On embedding a microarchitectural design language within haskell. In *ICFP '99: Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, 1999.

[62] Daan Leijen. Functional Components: COM Components in Haskell. Master's thesis, Department of Computer Science, University of Amsterdam, september 1998.

[63] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *Proceedings DSL*, Austin, Texas, 1999. Also appeared in ACM SIGPLAN Notices 35, 1, (Jan. 2000).

[64] 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*, 1999.

[65] Sheng Liang. *Modular Monadic Semantics and Compilation*. 1997.

[66] Sheng Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley, 1999.

[67] Andres Löh and Ralf Hinze. Open data types and open functions. In *Proceedings PPDP*, 2006.

[68] Erik Meijer. Server-Side Web Scripting in Haskell. *Journal of Functional Programming*, 10(1):1–18, january 2000.

[69] Erik Meijer and Brian Beckman. XLINQ: XML Programming Refactored (The Return Of The Monoids). In *Proceedings XML*, 2005.

[70] Erik Meijer, Brian Beckman, and Gavin Bierman. Linq: reconciling object, relations and XML in the .net framework. In *Proceedings SIGMOD international conference on Management of data*, 2006.

[71] Erik Meijer and Koen Claessen. The Design and Implementation of Mondrian. In *Proceedings Haskell Workshop*, 1997.

[72] Erik Meijer and Peter Drayton. Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages. In *Proceedings OOPSLA Workshop On The Revival Of Dynamic Languages*, 2004.

[73] Erik Meijer and Sigbjorn Finne. Lambada, Haskell as a Better Java. In *Proceedings Haskell Workshop*, 2000.

[74] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional Programming with Bananas, Lenses, Envelopes, and Barbed Wire. In *Proceedings FPCA*, volume 523 of *LNCS*, 1991.

[75] Erik Meijer and John Gough. Technical overview of the common language runtime, 2000.

[76] Erik Meijer and Graham Hutton. Bananas in space: extending fold and unfold to exponential types. In *Proceedings FPCA*, 1995.

[77] Erik Meijer, Daan Leijen, and James Hook. Client-side Web Scripting with HaskellScript. In *Proceedings PADL*, volume 1551 of *LNCS*, 1998.

[78] Erik Meijer, Nigel Perry, and Arjan van Yzendoorn. Scripting .NET Using Mondrian. In *Proceedings ECOOP*, volume 2072 of *LNCS*, 2001.

[79] Erik Meijer and Wolfram Schulte. XML Types for C$^\sharp$. BillG ThinkWeek Submission Winter 2001.

[80] Erik Meijer, Wolfram Schulte, and Gavin Bierman. Programming with Circles, Triangles and Rectangles. In *Proceedings XML*, 2003.

[81] Erik Meijer, Wolfram Schulte, and Gavin Bierman. Unifying Tables, Objects and Documents. In *Proceedings DP-COOL*, volume 27 of *John von Neumann Institute of Computing*, 2005.

[82] Erik Meijer and Mark Shields. XMLambda: A Functional Programming Language for Constructing and Manipulating XML Documents. Unpublished draft.

[83] Erik Meijer, Amanda Silver, and Paul Vick. Overview Of Visual Basic 9.0. In *Proceedings XML*, 2005.

[84] Erik Meijer and Danny van Velzen. Haskell Server Pages: Functional Programming and the Battle for the Middle Tier. In *Proceedings Haskell Workshop*, 2000.

[85] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[86] Adriaan Moors, Frank Piessens, and Martin Odersky. Towards equal rights for higher-kinded types. In *Proceedings MPOOL*, 2007.

[87] Thomas Nordin and Simon Peyton Jones. Green Card: a Foreign-language Interface for Haskell. In *Proceedings Haskell Workshop*, 1997.

[88] Andy Oram and Greg Wilson, editors. *Beautiful Code: Leading Programmers Explain How They Think*. O'Reilly, 2007.

[89] Matthew Parkinson. Class invariants: The end of the road? In *Proceedings IWACO*, 2007.

[90] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, Massachusetts, 2002.

[91] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with sawzall. *Science of Computer Programming*, 13(4):277–298, 2005.

[92] Riccardo Pucella, Erik Meijer, and Dino Oliva. Aspects de la Programmation d'Applications Win32 avec un Langage Fonctionnel. In *Journées Francophones des Langages Applicatifs*, 1999.

[93] Christopher Re, Jerome Simeon, and Mary Fernandez. A complete and efficient algebraic compiler for xquery. In *Proceedings ICDE*, 2006.

[94] Ben Rudiak-Gould, Alan Mycroft, and Simon Peyton Jones. Haskell is not not ml. In *Proceedings ESOP*, 2006.

[95] Claudio Russo. The joins concurrency library. In *Proceedings PADL*, volume 4354 of *LNCS*, 2007.

[96] Mark Shields and Erik Meijer. Type-Indexed Rows. In *Proceedings POPL*, 2001.

[97] Jeremy Graham Siek and Walid Taha. Gradual typing for objects. In *Proceedings ECOOP*, 2007.

[98] Erik Meijer Sigbjorn Finne, Daan Leijen and Simon Peyton Jones. H/Direct: A Binary Foreign Language Interface for Haskell. In *Proceedings ICFP*, 1998.

[99] Erik Meijer Sigbjorn Finne, Daan Leijen and Simon Peyton Jones. Calling Hell from Heaven and Heaven from Hell. In *Proceedings ICFP*, 1999.

[100] Erik Meijer Simon Peyton Jones and Daan Leijen. Scripting COM Components in Haskell. In *Proceedings Software Reuse*, 1998.

[101] P. Thiemann. A Typed Representation for HTML and XML Documents in Haskell. *Journal of Functional Programming*, 12(5):435–468, July 2002.

[102] Peter Thiemann. WASH Server Pages. In *Proceedings FLOPS*, 2006.

[103] Dave Thomas. The impedance imperative tuples+objects+infosets=too much stuff! *Journal of Object Technology*, 2(5):7–12, September-October 2003.

[104] Dave Thomas. Message oriented programming. *Journal of Object Technology*, 3(5):7–12, May-June 2004.

[105] Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: from scripts to programs. In *Proceedings OOPSLA Dynamic Language Symposium*, 2006.

[106] Mandana Vaziri, Frank Tip, Stephen Fink, and Julian Dolby. Declarative object identity using relation types. In *Proceedings ECOOP*, 2007.

[107] Phil Wadler and Simon Peyton Jones. Comprehensive comprehensions: comprehensions with "order by" and "group by", 2007.

[108] Philip Wadler. Comprehending monads. In *Proceedings LFP*, 1990.

[109] Philip Wadler. Lazy vs strict. *ACM Computing Surveys*, 28(2):318–320, 1996.

[110] Stefan Wehr, Ralf Lämmel, and Peter Thiemann. Javagi: Generalized interfaces for java. In *Proceedings ECOOP*, 2007.

[111] Martin Wildmoser and Tobias Nipkow. Certifying machine code safety: Shallow versus deep embedding. In *Proceedings TPHOLs*, volume 3223 of *LNCS*, 2004.