Leveraging .NET Meta-programming Components from F#

Integrated Queries and Interoperable Heterogeneous Execution

Don Syme
Microsoft Research, Cambridge
dsyme@microsoft.com

Abstract

Language-integrated meta-programming and extensible compilation have been recurring themes of programming languages since the invention of LISP. A recent real-world application of these techniques is the use of small meta-programs to specify database queries, as used in the Microsoft LINQ extensions for .NET. It is important that .NET languages such as F# are able to leverage the functionality provided by LINQ and related components for heterogeneous execution, both for pragmatic reasons and as a first step toward applying more disciplined, formal approaches to these problems. This paper explores the use of a modest meta-programming extension to F# to access and leverage the functionality of LINQ and other components. We do this by demonstrating an implementation of language integrated SQL queries using the LINQ/SQLMetal libraries. We also sketch two other applications: the execution of data-parallel quoted F# programs on a GPU via the Accelerator libraries, and dynamic native-code compilation via LINQ.

Categories and Subject Descriptors D.3.3 [*Programming*]: Language Constructs and Features

General Terms Languages

Keywords Meta-programming, Functional Programming, LINQ, GPUs, Reflection, Database Languages, Domain Specific Languages

1. Introduction

Language-integrated meta-programming has been a recurring theme of programming languages since the invention of LISP. However, in the context of the traditional goals of the ML language design (e.g. type safety, static guarantees, efficient compilation, equational reasoning and modular programming) meta-programming has had a poor reputation: until the advent of more disciplined systems such as Meta ML [30], meta-programs were rightly regarded as simply abstract syntax trees ("the equational theory of LISP fexprs is trivial" [33]), dynamic entities offering few static guarantees, and applications of meta-programming were relatively uncompelling.

An interesting new development is the advent of a broad range of high-quality components for executing program fragments on platforms such as .NET. These are readily accessible through interoperable versions of ML such as F# [28]. As a result applications can now be surprisingly heterogeneous: programs may generate both SQL and JavaScript, heterogeneous between server, database and browser. Even the presence of components such as the .NET CLR or JVM cannot always be taken for granted, as seen in the growing use of Graphics Processing Units (GPUs) as general purpose computing devices [19, 31].

Given this heterogeneity it is somewhat inevitable that people turn toward extensible compilation and intensional[1] meta-programming to bring a degree of uniformity to programming. For example, mainstream languages are now using small meta-programs to specify SQL queries, as in the Microsoft LINQ extensions for Visual Basic 9 and C# 3.0 [20, 9], an initiative that forms part of the background to this paper. One of the motivations of LINQ is to re-use existing infrastructure (e.g. rich editing environments) in the context of embedded domain-specific languages. The end result is that environments like .NET are moving to incorporate a layer of intensional meta-programming, including components to translate meta-programs to languages such as SQL.

This paper explores three applications of intensional meta-programming on the .NET platform using F# and a modest meta-programming extension to F#. The application areas and corresponding contributions of this paper are as follows:

1. We present the first implementation of language-integrated SQL queries for an ML dialect;
2. We present experimental examples of dual-mode GPU execution for data-parallel programs, where data-parallel array programs can be run either directly on a CPU or as meta-programs translated to GPU pixel shader code;
3. We present preliminary results for runtime code generation based on the LINQ dynamic expression compiler.

This work should be seen as a set of experiments in potential application areas for meta-programming: in a sense it is also a series of experiments in domain-specific language embeddings. However the focus is on showing how we can *reuse* existing execution components, thus indicating that meta-programming may be escaping its one-language-centered history, at least in the context of .NET. We do not claim specific advances in the theory of meta-programming, and indeed if anything examples 2 and 3 highlight the importance of applying approaches based on partial evaluation

ML'06 September 16, 2006, Portland, Oregon, USA.

[1] By "intensional meta-programming" we mean systems where the equational properties of program fragments (e.g. extensionality) are not necessarily preserved through meta-programming, e.g. because meta-programming exposes entire abstract syntax trees. This covers most forms of meta-programming, with the notable exception of meta-logic and staged computation systems.

and staged computation, since in both domains sophisticated patterns of program generation are common.

We consider the above results remarkable for what can be achieved with relatively little effort. This is because we leverage stable, well-tested and well-designed components written outside F# — for SQL and dynamic compilation we use the LINQ libraries from Microsoft, and for GPU execution we utilize the Accelerator library from Microsoft Research [31].

In the remainder of this section we give background on LINQ and related approaches. In §2 we describe F# and the quotation mechanism we use in this paper. In §3 we describe SQL-integrated queries in F# and their implementation via the LINQ libraries. In §4 we describe experiments with dual-mode CPU/GPU programming, and in §5 we describe experiments with runtime code generation. We summarize related work and future directions in §6.

1.1 Approaches to Interoperable, Heterogeneous Meta-programming

An excellent comparative overview of meta-programming for domain-specific languages can be found in Czarnecki et al. [7].

A crucial long-term goal of meta-programming research is to preserve the properties of source languages (e.g. extensionality) while giving disciplined frameworks for program generation and execution — in the realm of code generation this has been achieved through staged computation [30, 29, 25]. Recent work has shown how to extend staged computation to include the generation of C code — a form of "implicitly heterogeneous" execution [8]. It is interesting to look at the viability of these techniques for the applications described in this paper, a topic we consider in §6.

However, we have not used staged type systems in this paper, and as such this paper risks infuriating those working on "taming the beast" of meta-programming through this technique. We ask for forbearance, on the grounds that sometimes it is necessary to investigate application areas, both for pragmatic reasons and in order to develop an understanding of how more disciplined approaches might be applied to programming in worlds such as .NET and Java.

1.2 LINQ, DLINQ and LINQ Meta-programming

Part of the background to this paper is the rapid adoption by Microsoft of the use of meta-programming in the Microsoft LINQ extensions for C# 3.0 and Visual Basic 9. The appendix describes the relevant technical features by example. For the purposes of this paper the salient features are:

- C# 3.0 includes a generics, extensions to the Java/.NET nominal OO model to permit functional/aggregate operations on data structures, an extensible dot-notation, a modicum of local type inference and syntactic sugar for the use of functional/aggregate operations. The functional programming community can take considerable credit for the inclusion of these features [16, 1, 5].
- The .NET platform will include several important components which constitute the functionality we seek to leverage in §3 and §5:
  - `System.Query.dll`, containing the definition of the type `Expression<T>`, a representation of typed abstract syntax trees for expressions. This DLL also contains a dynamic compiler for these, generating garbage-collected .NET IL code through .NET "light-weight code generation" [23], and executed as native code;
  - `SQLMetal.exe`, a program generator builds type-annotated object models for database schema;
  - `System.Data.DLinq.dll`, a compiler from LINQ expressions that use `SQLMetal` object models to SQL.

The end result is that C# program fragments such as

```
db.Employees
  .Where(e => e.City = "London")
  .Select(e => e.Address);
```

are executed as SQL code on the database, e.g.

```
SELECT [t0].[Address]
FROM [Employees] AS [t0]
WHERE [t0].[City] = London
```

Further C# examples are given in the Appendix. Above we we use underlining to indicate implicitly quoted fragments of the program.

The specification of acceptable constructs and their translation to SQL is itself a lengthy document, part of the LINQ distribution [20]. This specification notes many constructs that are not translatable (and which will give runtime errors), as well as some deliberate semantic differences between C# and C#-expressions-when-implicitly-quoted-and-run-as-SQL, which we'll call LINQ-SQL. For example SQL's date/time values are rounded to .000, .003 or .007 seconds, so are less precise than .NET's, and "Math.Round" is given a different semantics. In real-world heterogeneous programming this kind of mismatch seems unavoidable.

2. F# and F# Quotations

In this section we describe F#, and then describe the quotation mechanism we use in this paper.[2][3]

2.1 F#

F# is a multi-paradigm .NET language explicitly designed to be an ML suited to the .NET environment. It is rooted in the Core ML design, and in particular has a core language largely compatible with that of OCaml.

F# takes interoperability as axiomatic. By this we mean that the language design must enable and encourage the use of functionality implemented outside the language, whether that functionality is accessed via programmatic frameworks such as .NET or by other means. Interoperability has been an increasingly important theme of ML language design: e.g. OCaml highlights the importance of simple, direct models of compilation and representation, partly to enable a simple C FFI; MLj [3] and SML.NET [2] highlight the importance of utilizing runtime components from environments such as the JVM and .NET. F# extends these themes, particularly that of high-level interoperability [27].

Linguistically, F# includes:

- The standard constructs of Core ML, essentially as implemented by OCaml;
- Type inference based on an instantiation of HM(X) with subtype and operator overloading constraints [21];
- A .NET-style nominal object model including classes, single inheritance, object expressions, properties and nominal interfaces associated with object values;
- A dot notation, where overloading is resolved in a type-directe fashion based on the type information available on a left-to-right, outside-in analysis of a file.

[2] F# is described in detail on the F# website [28], but has not been described in detail in previous publications.

[3] The finalized design of meta-programming support for F# may vary greatly from that described here, and indeed the process of writing this paper has clarified many issues to the author. We encourage interested readers to consider this an invitation to assist in the design process!

- Nested modules and module signatures, but not functors. Modules may not be used as values or re-constrained under different signatures.

Like other .NET languages, F# derives much of its power from its reliance on .NET:

- Garbage collection;
- JIT and "install-time" compilation;
- Cross-language, interoperable generics;
- Relatively high performance, especially for floating point;
- Concurrent GC and SMP support;
- A vast array of high-quality libraries, including Windows Forms and Managed DirectX;
- Debuggers, CPU profilers and memory profilers;
- Portability across any Common Language Infrastructure (CLI) [10] implementation, of which Microsoft's .NET Framework is one.

Some F# specific tools are added:

- A cross-module optimizing compiler;
- F# for Visual Studio — an interactive development environment with on-the-fly type checking, automatic drop-down menus and balloon tips based on contextual type information;
- F# Interactive — an interactive read/eval loop, optionally hosted in Visual Studio. .NET libraries can be dynamically loaded, and code entered interactively is optimized and run as native code.

F# embraces interoperability with CLI paradigms, for example:

- F# types and code can be used directly from other CLI languages;
- F# both generates and consumes generic CLI code, i.e. ML polymorphism is compiled as CLI generics, and generic definitions from other CLI languages can be used as polymorphic definitions to F# code;
- A simple, direct model of compilation is used, and optimization settings do not change the binary interface of F# components;

As far as ML dialects go, F# supports an atypical number of dynamic techniques. For example, all F# type variables have intensional representations available at runtime, largely as a side effect of the use of CLI generics to implement type parameterization. Thus the type-erasure property does not hold for F# code, but careful use of this feature allows generic code to return code specialized by type representation and also permits the implementation of some interesting polytypic functions. We make use of intensional type representations at several points in this paper. CLI implementations typically pass and store of type representations in a manner reminiscent of the dictionaries use to implement Haskell's type classes, in particular the `Typeable` class of Haskell.

Similarly, F# supports both subtyping and type-rediscovery through downcasts, this time as a side-effect of the fact that all .NET reference type values carry runtime types. All F# types are safely[4] convertible to a universal type `obj` (`System.Object`), where the original type is rediscovered by using a pattern-matching downcast. This permits the easy implementation of generic functions such as a generic pretty printer `print_any`.

Some other points relevant to this paper are:

[4] Here we use the word "safely" in the sense of "avoiding memory corruption."

```
val (|>) : 'a -> ('a -> 'b) -> 'b
val (>>) : ('a -> 'b) -> ('b -> 'c) -> '('a -> 'c)

type seq<'a> = System.Collections.Generic.IEnumerable<'a>
module IEnumerable : sig
  val map      : ('a -> 'b)   -> #seq<'a> -> seq<'b>
  val filter   : ('a -> bool) -> #seq<'a> -> seq<'a>
end
```

Figure 1. Some functions and type abbreviations used in this paper

- Some signatures from the F# standard library are shown in Fig 1;
- Both prefix and postfix syntax can be used for types, e.g. `int list` and `list<int>` are equivalent;
- The syntax `#`τ is used to mean "τ or any of its subtypes", shorthand for a new type variable α with a subtype constraint at its (inferred) binding site;
- Subtyping does not extend through any structural types, e.g. the tuple, list and function type constructors are all invariant. This avoids many of the technical difficulties of combining subtyping with ML type inference, at the expense of requiring the insertion of some explicit coercions;
- Operator overloading is supported, based on .NET standards, typically resolved according to the inferred nominal type information of the left-hand parameter of a binary operator;
- The default semantics for hashing, equality and comparison can be specified for each new concrete F# type definition, where concrete types are records, unions and class definitions.

F# shares many similarities with SML.NET [2], though there are important differences, both as languages and in terms of the intended use of the language implementations. Both languages have interoperability extensions. SML.NET is a highly-optimizing whole-program compiler for a standard language, while F# is a more experimental language for efficient, interactive symbolic programming, often in data-rich contexts. It is partly the desire to bring type-safety and language integration to interactive data acquisition that has led to the work described in this paper.

2.2 F# Quotations

This paper utilizes a modest experimental meta-programming extension to F# called F# quotations. These allow the capture of type-checked expressions as structured terms, typically of type `Quotations.Typed.Expr<`α`>` for some α (here abbreviated α `expr`), or as a non-type-annotated "raw" quotation of type `Quotations.Raw.Expr` (abbreviated `rexpr`). These structured terms can then be interpreted, analyzed and compiled to alternative languages. The terms are presented after type-checking and after the removal of pattern matching into discriminate/project form.

In the terminology and analysis of Czarnecki et al. [7], F# quotations are a manifestation of "template" meta-programming (or perhaps a category called "term" meta-programming is justified). Quotation literals are parsed and type-checked statically, but quotations can also be created programatically at runtime. They permit the reuse of the F# parser and type-checker, though not the F# code generator — the purpose of the mechanism is to reuse alternative execution machinery unrelated to F#. They statically guarantee the syntactic validity of quoted fragments and the typing of quoted literals. They permit the inspection of quoted fragments, at least by "library writers" (see below). They are a sub-language of F#, as you can't quote entire F# modules, and support a limited form of

"cross-stage persistence" (see §2.8). They can be used in separately compiled modules.

2.3 Quotations as Terms

Putting syntax aside, F# quotations are similar to many other representations of typed lambda expressions, and other systems use techniques close to those described here, e.g. see ReFLect [14]. However, from the perspective of interoperability it is important to understand how F# quotations relate to existing reified constructs on the .NET platform. We also use the syntax of quotation literals extensively in this paper.

Quotations are intended to be used in two modes: one where the "library writer" performs intensional analysis on raw terms, and one where "average users" write quotation literals and use functionality provided by the library writer. Quotation processing libraries effectively act as a form of compiler extension. This distinction is not currently enforced, but in future versions .NET security permissions [17] may be required to access intensional representations.

As far as the library writer is concerned raw quotation terms are an abstract type that can be projected to the typical "code" algebra up to a family of constants `rconst`, a type of variable names `rvname`, and an reified representation of types `rtype` (types and constants are discussed in §2.6):[5]

```
type rexpr ⇝
  | Const of rconst * rtype list
  | App   of rexpr * rexpr
  | Var   of rvname * rtype
  | Lam   of (rvname * rtype) * rexpr
```

Two extra constructs represent holes and nested quotations:

```
type rexpr ⇝
  ...
  | Quote of rexpr
  | Hole of rtype
```

Raw quotation terms carry internal type annotations. They can be constructed programmatically, but runtime checks are applied to ensure only well-typed `rexpr`s are constructed, in the style of the term algebras for LCF theorem proving systems [13].[6] Variable names are effectively strings, and are observable, but the default equality, hashing and comparison routines are up to alpha-equivalence.

Type-annotated terms (α `expr`) are also an abstract type, though library writers can access raw terms from typed terms, and a casting operator is available to annotate a raw term.

```
Typed.to_raw    : 'a expr -> rexpr
Typed.of_raw    : rexpr -> 'a expr
```

`Typed.of_raw` applies a runtime type check to ensure that the type annotation is correct based on internal type annotations in the raw expression and F#'s intensional representation of the type variable `'a`, thus enforcing a correspondence between the language type system and the runtime type annotations on raw terms.[7]

2.4 Quotation Literals

Quotation literals quote expressions and are any uses of the family of parenthetical operators beginning `<@... ...@>`, e.g. `<@ @>` and `<@| |@>`. Here are some simple instances of typed quotation literals:

```
<@ 1 + 1 @>            : int expr
<@ (fun x -> x + 1) @> : (int -> int) expr
```

User-defined operators are also permitted, e.g. `<@+ +@>`. Each operator captures a typed abstract syntax tree form of the enclosed expression and invokes the given operator on the result. Quotation operators involving `@@` (e.g., `<@@ @@>`) are "raw" operators, typically used within implementations of quotation processing libraries. The following illustrate some simple instances of raw quotation:

```
<@@ 1 + 1 @@>            : rexpr
<@@ (fun x -> x + 1) @@> : rexpr
```

As far as average users are concerned, quotations are "hygienic" because quotation literals must be closed, up to the use of "top-level" definitions. As in ReFLect [14], top-level definitions are part of the algebra of constants and include all public definitions in any accessible F# module as well as definitions entered in the read-eval loop.

A current limitation is that `rexpr`s do not include quantification over type variables (there are no $\forall$ or $\exists$ types or Λ terms): only top-level constants may be polymorphic. Thus generalization is not performed at `let` bindings within quoted terms, so the following quotation term will give a compile-time error:

```
<@@ let f x = (x,x) in f 1, f "1" @@>
```

2.5 Quotation Template Literals

Quotation literals may also contain holes `_`, which allow them to be used as functions that generate expressions at runtime by "splicing in" other expressions (a form of anti-quotation), or as templates by matching against input expressions. Some example uses of templates are:

```
<@  1 + _  @> : int expr -> int expr
<@  1 + _ + _  @> : int expr -> int expr -> int expr
<@@  1 + _  @@> : rexpr -> rexpr
```

The query form is often used in a pattern match as follows:

```
let decompose inp =
  match <@@| sin(_) * sin(_) |@@>(inp) with
  | Some(v1,v2) -> ...
  | None -> ...
```

Library writers can define operators that give alternative meanings to templates, e.g. the holes may be interpreted according to a different (e.g. higher-order) matching algorithm (c.f. higher-order matching for code pattern literals [24]). We do not use literal templates extensively in this paper, so do not describe them in detail here. However they are used throughout the implementations of the quotation processing functions we describe.

When a library writer defines a new raw quotation operator the template is passed to the implementation of the operator as type `Raw.Template<`τ_1`,`τ_2`>` (abbreviated `rtempl<`τ_1`,`τ_2`>`). Here τ_1 and τ_2 are phantom types indicating the number of quotation holes in the quoted term. The pre-defined raw quotation literal operators have the following types:

```
(<@@. .@@>)     : rtempl<'a,'b> -> rtempl<'a,'b>
(<@@| |@@>)     : rtempl<'a,'b> -> rexpr -> 'a option
(<@@   @@>)     : rtempl<'a,'b> -> 'b
```

The phantom types are, respectively, a tuple of the types returned by a successful query, and the iterated function type representing the use of the template as a generator. A similar mechanism is used for the "typed" quotation operators, e.g. the pre-defined quotation literal operators have the following types:

[5] We use ⇝ to indicate that the abstract type can effectively be projected (viewed) as the given algebra through the use of auxiliary functions, see §2.7. The algebra doesn't exist as an algebraic data type in the F# quotation library.

[6] These runtime checks are not present in the released implementation at the time of writing, though we don't foresee any problems in adding them.

[7] Again, these runtime checks are not present in the released implementation at the time of writing.

```
(<@. .@>) : templ<'a,'b,'c> -> templ<'a,'b,'c>
(<@| |@>) : templ<'a,'b,'c> -> 'a expr -> 'b option
(<@   @>) : templ<'a,'b,'c> -> 'c
```

Operators to fill and match templates are defined as polytypic functions in the F# library, but could just as easily be redefined by the library writer. At present quotations literals may not contain holes for types, though this would clearly be useful when working with polymorphic constructs.

2.6 Quoted Types

The reified types and constants of the quoted language correspond to the types of .NET and the constants of the F# language. Reified types (`rtypes`) use a standard first-order representation of types, where type variables are indexed by integer index.

```
type rtype ⇝
  | VarType   of int
  | AppType   of rtycon * rtype list
```

The algebra of reified types is tantalizingly close to the reified representation of types on the .NET platform itself, i.e. the .NET type `System.Type`. It would be an enormous simplification if F# could leverage this representation and avoid any additional reified representation of types. For ground types this works satisfactorily, with some caveats for large tuple types mentioned below. However, the `System.Type` representation uses "graph binding" for type variables, where type variables are effectively pointers to a direct, exact binding location within a .NET program. This would require all type variable binders in F# source code to have a corresponding binding in compiled .NET code, an unacceptable constraint when compiling an ML language.

On the other hand, we accept a 1:1 correspondence between bindings of F# concrete type constructors (e.g. records and unions) and the existence of named .NET type definitions: this correspondence is, for example, part of the reason why it is easy to interoperate with F# code from other .NET languages. Thus our reified view of type constructors does leverage the corresponding .NET representation of type constructors (which also happens to be `System.Type` — the .NET construct serves multiple roles). We add an additional case to handle large F# tuple types, which only have a corresponding isomorphic matching set of `System.Type` constructors up to a fixed tuple size:[8]

```
type rtycon ⇝
| TupleTyOp  of int   // length
| NamedTyOp of System.Type
```

No syntax currently exists for quoted `rtype` literals. F# already includes a construct to reify F# types as .NET `System.Type` values, e.g. "`type int`" or "`type ('a * 'a)`". An F# library functions exist to project these into the `rtycon` and `rtype` algebra.

2.7 Programmatic access through expression families

The expression types α `expr` and `rexpr` are abstract, and are built and deconstructed using projection/injection pairs, which we call "families":

```
type ('a,'b) family =
 { Make: 'a -> 'b;
   Query: 'b -> 'a option; }
```

A library of compositional operators is provided for this type, e.g.

```
val fMap : ('a -> 'b) * ('b -> 'a)
           -> ('a,'c) family -> ('b,'c) family
val fOrElse : ('a,'b) family
               -> ('a,'b) family
               -> ('a,'b) family
```

The following families form a complete decomposition of raw expressions:[9]

```
type 'a rexprFamily = ('a, rexpr) family
val rvar = (rvname * rtype)

val efVar    : rvname rexprFamily
val efHole   : rtype rexprFamily
val efApp    : (rexpr * rexpr) rexprFamily
val efLambda : (rvar * rexpr) rexprFamily
val efQuote  : rexpr rexprFamily
val efConst  : (rconst * rtype list) rexprFamily
```

Decomposition operations are not provided directly on the α `expr` type due to the difficulty of statically maintaining correct type annotations.[10]

The algebra of constants `rconst` is not directly accessible but is revealed through expression families, e.g., the following families reveal the existence of constants for sequencing and integers:

```
val efInt32 : int32 rexprFamily
val efSeq : (rexpr * rexpr) rexprFamily
```

There are currently 30 expression families which together cover all constant constructions. Some of these cover the use of methods and types from other .NET languages, e.g. `efMethodCall`.

The library defines a number of additional families and combinators for performing more interesting queries on raw expressions, e.g.

```
val efLambdas : (rvar list * rexpr) rexprFamily
val efAnd     : (rexpr * rexpr)     rexprFamily
val efLet     : (rvar * rexpr)      rexprFamily
```

These both abstract the encoding of F# language constructs into the term syntax, and help control the complexity of expression manipulation and construction.

A final family of constants permits the reification of any .NET object value into the term structure:

```
val efLiftedValue    : obj rexprFamily
```

Constants of this kind can be generated using the following function:

```
val lift : 'a -> 'a expr
```

2.8 Optional persistence of top-level definitions

The primary technical distinction between F# quotations and C# (implicit) quotations is that top-level F# definitions can be optionally compiled to exhibit "persistence", i.e. compiled as both as F# compiled code and as reified definitions for use by quotation processing environments. This applies to F# code compiled with the `--quotation-data` flag. This only applies to "top-level" definitions in the sense of §2.4.

Uses of top-level definitions appear as a variety of constant node in the algebra of expressions:

[8] In the current released version of F# the algebra of type constructors also includes cases for arrays and function types. These are not required, since satisfactory `System.Type` values exist to represent these.

[9] As usual, the lack of "views" [32] in ML-family languages means we lose the static information that the families form a disjoint and complete decomposition of the type.

[10] GADTs [34] could be used to permit an algebraic view of the α `expr` type that preserves type information.

```
type topDefData
val efAnyTopDefn : (topDefData * rtype list) rexprFamily
```

The `topDefData` represents linking information (i.e. the name of a top-level definition and the .NET library it lives in), and the type arguments represent the instantiation for uses of polymorphic functions.

The reified definition of top-level constructs can be accessed using the library function `resolveTopDefn`:

```
let sinExpr = <@@ sin @@>;;

// Show that this is a use of a top-level definition:
let sinTopDefData,sinTypeArgs =
 match efAnyTopDefn.Query(sinExpr) with
 | Some(res) -> res
 | None -> failwith "no match";;

// Resolve the top level definition to its body
resolveTopDefn(sinTopDefData,sinTypeArgs);;

val it : rexpr option
   = Some <@@ fun x -> System.Math.Sin(x) @@>
```

which indicates that the F# `sin` function is implemented as a .NET method call to the method `System.Math.Sin`. `resolveTopDefn` may return `None` if the definition is not available or includes some of the corner constructs not supported by quotations (e.g. "inline .NET assembly code", used in some places by the F# library). This function also applies the type instantiation to the body of the definition.

Persistent definitions let us use common functions as macros in embedded languages. For example, we frequently use the operator `|>` within quotations, knowing that most quotation translations will implement the expansion of the macro, typically by inserting `let`-bindings to maintain sharing, rather then by substitution:

```
let (|>) x f = f x
```

3. Language Integrated SQL Queries with F#

We now turn to the three applications where we leverage .NET components through quotation meta-programming, which we label FSQL (running quoted F# programs as SQL via the LINQ DLinq libraries), FGPU (likewise as GPU pixel shader code via the Accelerator libraries) and FCPU (native code generation from closed quotations using the LINQ dynamic compiler). We emphasize three things:

- Our aim is to show that it is *possible* to leverage .NET meta-programming components via F#, and meta-programming in particular. Whether it is *desirable* to do so depends on the technical qualities of the component being accessed, among other things.
- We concentrate on examples where we don't programmatically generate quoted programs. For example, we don't take an input list and generate a fresh SQL query based on a fold over that list. We speculate that an emphasis on program generation would immediately lead to a stronger emphasis on staged computation, and indeed that a staged computation mechanism could be useful and orthogonal to the mechanisms used here.
- The use of F# meta-programming is not, strictly speaking, necessary to access the underlying components — you can explicitly create the relevant data (e.g. expression tree values) and call the libraries directly. However in two of three cases this feels considerably more awkward — for the case of GPU programming the tradeoffs feel more marginal.

For FSQL and FCPU we assume the existence of the following function, accessible to meta-programming library writers, that translates F# quotations to LINQ expression trees, for the subset of F# expressions that correspond to C# expressions:

```
val proj2: ('a -> 'b) expr -> Expression<Func<'a,'b>>
```

Our first example, FSQL, is language-integrated SQL in F# by using the LINQ-SQL libraries. We ignored a number of details, e.g. the use of nulls in SQL tables, which is handled largely through .NET `Nullable` types. We first show the simplistic translation of the C# example from §1.2 to F#. However, in the case of queries that combine and refine multiple relations (e.g. joins, or grouping) the F# version reveals non-orthogonalities hidden by the implicit typing of the C# presentation. We then give an alternative approach for F# to account for this. For examples we use the .NET object-model API to the "Northwind" database provided with the LINQ Preview [20]. This API was generated by the SQLMetal tool and was also used in the examples in §1.2. For completeness, here is how a database connection is established from F#:[11]

```
let connString =
  @"AttachDBFileName='c:\My Documents\Northwind.mdf';
    Server='.\SQLEXPRESS';
    Integrated Security=SSPI;
    enlist=false"

let db = new Northwind(connString)
```

Here `db` has type `Northwind`, which has, among other things, properties `Customers` and `Employees` of types of the same names, each subtypes of `query<Customer>` and `query<Employee>` (we abbreviate `IQueryable<`τ`>` as `query<`τ`>`).

3.1 Simple Queries

A direct translation of the first C# example from §1.2 would involve the use of operations on the LINQ `IQueryable` type, shown in Fig 2. Here `db` has type `Northwind`, a subtype of `System.Query.IQueryable<Employee>`.

```
let query =
  db.Employees
  |> IQueryable.where  <@ fun e -> e.City = "London" @>
  |> IQueryable.select <@ fun e -> e.Address @>
```

We have used database names such as `where` and `select` — later we need more sophisticated database operations such as joins, and a consistent database naming terminology helps identify the permitted language subset and its correspondence to the LINQ specification. Type-checking and inference ensure that `e` is known to have type `Employee`, which in turn allows us to resolve the dot-notation overload `City`. The corresponding SQL code is the same as in §1.2:

```
SELECT [t0].[Address]
FROM [Employees] AS [t0]
WHERE [t0].[City] = London
```

The (approximate) intended semantics of the LINQ-SQL libraries is that the execution of the meta-programs should be the same as if the programs were converted to equivalent programs over in-memory lists, where the database table is treated as an in-memory enumerable data structure. Indeed, the .NET type `IQueryable<`τ`>` is a subtype of `IEnumerable<`τ`>`, indicating (but in no way enforcing) the intended correspondence. As it happens, the following F# program would achieve the same result, at the expense of dragging the entire database table in-memory:

[11] As in C#, strings beginning with '@' are "verbatim" strings that can include backslashes.

```
type seq<'a> = System.Collections.Generic.IEnumerable<'a>
type q<'a> = System.Query.IQueryable<'a>
module IQueryable : sig
  val select : ('a -> 'b)      -> #q<'a> -> q<'b>
  val where  : ('a -> bool)    -> #q<'a> -> q<'a>
  val selectn: ('a -> seq<'b>) -> #q<'a> -> q<'b>
end

module Sequence : sig
  val select    : ('a -> 'b)   -> #seq<'a> -> seq<'b>
  val where     : ('a -> bool) -> #seq<'a> -> seq<'a>
  val selectn: ('a -> seq<'b>) -> #seq<'a> -> seq<'b>
end

module IQueryable = struct
  open System.Query
  let where qf c =
    System.Query.Queryable.Where (c,proj2 qf)
  ...
end

module Sequence = struct
  open System.Query
  let F1 f = new Func<_,_>(f)
  let where f c = System.Query.Sequence.Where (c,F1 f)
  ...
end
```

Figure 2. Operators needed for the first, awkward approach: use an F# view of the LINQ `IQueryable` and `ISequence` operators. `System.Query.Queryable.Where` etc. are the fully qualified names for the C# extension methods used in §1.2.

```
open IEnumerable;;
let query =
  db.Employees
  |> IEnumerable.filter (fun e -> e.City = "London")
  |> IEnumerable.map    (fun e -> e.Address)
```

Here `IEnumerable.filter` is one the operations defined shown in Fig 1.

3.2 The Problem of Nested Queries

The use of subsets of functional languages as query languages is largely understood, e.g. see Buneman et al. [6]. Neither F# nor C# 2.0 are relational languages, so it is not surprising that specifying relational programs becomes a little harder than shown in the previous section. C# 3.0 and other languages have worked around this limitation by adding a relational comprehension notation explicitly for use with linearly ordered data structures. For example, consider the following C# comprehension:

```
from e  in db.Employees,
     et in e.EmployeeOffices
where e.City == "Seattle"
select new e.Name, et.Office;
```

Adding a comprehension notation to F# is beyond the scope of this paper, so here we work only with the explicit forms. The above C# code is syntactic sugar for the following (`SelectMany` corresponds to `map >> concat`):

```
db.Employees
  .SelectMany(e =>
    ((IEnumerable<Office>)e.EmployeeOffices)
      .Where(et => e.City == "London")
      .Select(et => new (e.Name,et.Office)))
```

Note the irregularities in the explicit C# code: the inner lambda expressions are quoted only once, and the inner tables are treated as `IEnumerable` "in memory" data structures rather than `IQueryable` database tables. The natural direct translation of this code to F# is:

```
db.Employees
|> IQueryable.selectn <@ fun e ->
  e.EmployeeOffices
  |> IEnumerable.where  (fun et ->  e.City = "London")
  |> IEnumerable.select (fun et -> e.Name, et.Office) @>
```

and indeed we can arrange things such that the above program generates correct SQL. However, on the outside of the quotation we manipulate values of type `IQueryable` and on the inside `IEnumerable`. This is unappealing, and stems from artifices used in the C# presentation that give the outward appearance that the outer structure of the query is composed from a number of quoted fragments. However, this structure goes only "one level deep." We suspect that C# programmers seeking to understand the feature "in depth" may also find the C# approach strange, and indeed that it would make the programmatic generation of larger SQL queries more difficult.

3.3 The Second Approach

To avoid the lack of uniformity outlined in the last section we take the following approach:[12]

1. Represent queries by closed quoted programs that manipulate `IEnumerable` values;
2. Define SQL comprehension-building operators that work over `IEnumerable` values;
3. Provide a function `SQL` that compiles this the query program to the corresponding LINQ expression trees.

In particular, we define a function:

```
val SQL : ('db -> IEnumerable<'a>) expr
           -> 'db -> IEnumerable<'a>
```

This function succeeds if the quoted F# program represents a valid SQL query, in the subset outlined in Fig 4. Some of the operators used to build queries are shown in Fig 3. Here is an example of its use:

```
open Query

let results =
  db |> SQL <@ fun db ->
       db.Employees
       |> selectn (fun e ->
           e.EmployeeOffices
           |> where  (fun et -> e.City = "London")
           |> select (fun et -> e.Name, et.Office)) @>
```

The SQL function processes the quoted term to the form described above, then translates the remaining fragments to LINQ expressions using `proj2`. When the resulting LINQ expression/program is processed by LINQ-SQL library this generates:

```
SELECT [t0].[Name], [t1].[Office]
FROM [Employees] as [t0], [EmployeeOffices] as [t1]
WHERE ([t0].[City] = 'London')
  AND ([t1].[EmployeeID] = [t0].[EmployeeID])
```

Although syntactically slightly longer, this approach has a number of advantages:

[12] This technique does not appear in the currently released F#/Linq sample in the F# distribution, as the process of writing this paper clarified the need for a uniform treatment of inner queries. We expect to release a full sample implementation prior to the workshop.

```
type seq<'a> = IEnumerable<'a>
module Query : sig
  val select : ('a -> 'b)   -> #seq<'a> -> seq<'b>
  val where  : ('a -> bool) -> #seq<'a> -> seq<'a>
  val selectn: ('a -> #seq<'b>) -> #seq<'a> -> seq<'b>
  ...
end

module Query = struct
  let F1 f = new Func<_,_>(f)
  let where f c = System.Query.Sequence.Where(c,F1(f))
  ...
end
```

Figure 3. The better approach: query operations operate on IEnumerable.

- We only use one set of operators, over the type `IEnumerable`.
- The function `SQL` represents the essence of the underlying technique: it implements an embedded domain-specific language.
- The quoted program corresponds directly to the query operating over in-memory data-structures, and indeed can be run as such.
- The query program is *closed* (hence must accept the database as a parameter, though use of the `|>` notation makes this palatable). This means we can reuse it against other databases (or multiple connections to the same database, each of which SQLMetal represents by a different database object), and, if the functionality were made available by LINQ, we could amortize the cost of the translation to SQL.

3.4 The FSQL Embedded Language

The essence of the embedded quotation language accepted by the `SQL` function is shown in Fig 4. Some aspects of the language are dictated by the translatable primitives of the LINQ-SQL libraries, specified in the LINQ-SQL documentation. Furthermore, as shown in Fig 3, the operators such as `where` are themselves simply defined in terms of LINQ operators. The `SQL` function remaps all uses of `System.Query.Sequence.*` to the corresponding `System.Query.Queryable.*` operations. This means that the F# subset is effectively that specified by the LINQ-SQL documentation, with some additions such as the use of F# quotation macros. This means we defer most aspects of the specification of the accepted subset to the LINQ-SQL documentation.

LINQ-SQL query programs may also have a "terminating computation" that processes the results generated by an SQL query. For example, the following program constructs F# tuple values as part of the final selection. LINQ-SQL knows nothing of F# tuples, but automatically detects the residue that must be run in-memory.

```
let query =
  db |> SQL <@ fun db ->
    db.Employees
    |> where  (fun e -> e.City = "London")
    |> select (fun e -> (e.Name,e.Address))
  @>
```

4. Accelerating F# Code on the GPU

We next describe the use the Accelerator and DirectX libraries [31] to execute quoted F# programs on a GPU. To quote:

> There is significant interest in using GPUs for general-purpose programming... GPUs have an explicitly parallel programming model and deliver much higher performance for some floating-point workloads than comparable CPUs. It is, however, difficult for most programmers to make use of this computing power.... The Accelerator programs ... often significantly outperform the C++ programs, by up to 18x.

```
query = fun (db : 'db) -> seq
seq
  = unary-seqop seq
  | binary-seqop (seq,seq)
  | db.TableName
  | ...                                  (See notes)
unary-seqop
  = select proj
  | where proj
  | selectn(fun var -> seq)
  | ...                                  (See notes)
binary-seqop
  = join proj proj (fun v -> unary-seqop)
  | ...                                  (See notes)
proj
  = (fun var -> expr)
expr
  = expr.PropertyName
  | var
  | let var = expr in expr
  | expr {+,=,*,-,/,%} expr              (Over a limited range of types)
  | ...                                  (See notes)
```

Figure 4. The embedded FSQL language. The specification is in terms of operators from the `Query` module. The *expr* category also includes the large expressions translated by LINQ-SQL. All syntax categories include the use of F# quotation macros with *expr* arguments (see §2.8), translated by substitution. Additionally, any operators that also macro-expand to uses of the LINQ `System.Query.Sequence.*` operations and which are specified as SQL-acceptable by the LINQ-SQL documentation are also translatable.

The Accelerator library provides a programmatic API for the description of data-parallel array programs in terms of point-wise operations (`+`, `*`, etc.), reduction operations (sum-across-rows, sum-across-columns etc.) and transformation operations (shift, replicate, section, pad, transpose and others). These operations manipulate "parallel array types" that represent GPU programs over particular data types (e.g. `FPA` for floating-point array computations). Programmers must explicitly convert between data-parallel arrays and normal arrays.

4.1 An Example

Our aim in this section is to show the execution of a limited subset of closed quotation programs via Accelerator. We use the Game of Life as our sample. Fig 5 shows the `nextGeneration` function of the Game of Life written using standard functional programming over integer and boolean in-memory 2D F# arrays (using the F# `Array2` module).

We first show how to explicitly generate and evaluate the corresponding GPU program via calls to the Accelerator API. Fig 6 shows the F# code that generate a GPU program to compute one generation, using floating point numbers to represent the grid.

The style of programming shown in Fig 6 is quite reasonable, as ML makes an excellent meta-language for explicitly calling program-generation APIs.

However, there are benefits if we can see GPU execution as a special case of executing CPU programs: for example we may eventually wish to support the optimized execution of existing F# matrix programs. As a step in this direction, we have prototyped a function `accelerate` that takes a quoted program fragment over 2D arrays and compiles it to a GPU program:

```
let matrix f = Array2.init dimx dimy f
let K c = matrix (fun _ _ -> c)
let ttrue, ffalse = K true, K false
let zero, one, two, three = K 0, K 1, K 2, K 3
let (.&&) a b = matrix (fun i j -> a.(i,j) && b.(i,j))
let (.||) a b = matrix (fun i j -> a.(i,j) || b.(i,j))
let (.-)  a b = matrix (fun i j -> a.(i,j) - b.(i,j))
let (.+)  a b = matrix (fun i j -> a.(i,j) + b.(i,j))
let (.=)  a b = matrix (fun i j -> a.(i,j) = b.(i,j))
let neg a     = matrix (fun i j -> - a.(i,j))
let rotate a dx dy =
  matrix (fun i j -> a.((i+dx)%dimx,(j+dy)%dimy))
let count a = matrix (fun i j -> int_of_bool a.(i,j))

let nextGeneration(a) =
  let N dx dy = rotate (count a) dx dy in
  let sum =  N (-1) (-1) .+ N (-1) 0 .+ N (-1) 1
          .+ N 0 (-1)    .+             N 0    1
          .+ N 1  (-1)   .+ N   1  0 .+ N 1    1  in
  (sum .= three) .|| ((sum .= two) .&& a);;
```

Figure 5. The F# code to compute the Game of Life on the CPU using in-memory two-dimensional arrays. We assume the `dimx/dimy` parameters are fixed.

```
open Microsoft.Research.DataParallelArrays

let shape = [| dimx; dimy |]
let zero = new FPA(0.0f, shape)
let one = new FPA(1.0f, shape)
let two = new FPA(2.0f, shape)
let three = new FPA(3.0f, shape)
let And (a:FPA) (b:FPA) = FPA.Min(a, b)
let Or (a:FPA) (b:FPA) = FPA.Max(a, b)
let Rotate (a:FPA) i j = a.Rotate([| i;j |])

let Equals (a:FPA) (b:FPA) =
  let cond = -(FPA.Abs(a - b)) in
  FPA.Cmp(cond, one, zero)

let NextGeneration (a:FPA) =
  let N dx dy = Rotate a dx dy in
  let sum = N (-1) (-1) + N 0 (-1) + N 1 (-1)
          + N (-1) 0               + N 1 0
          + N (-1) 1    + N 0 1    + N 1 1 in
  Or (Equals sum three) (And (Equals sum two) a)

let initial = Array2.init(fun i j -> i=0 or j=0 or i=j)
let step0  = new FPA(initial);;
let step1 = NextGeneration(step0).Eval();
```

Figure 6. Generating the GPU version of the Life program by explicit calls to Accelerator. We assume the `shape` parameter is fixed.

```
let nextGenerationGPU = accelerate <@ nextGeneration @>
let step1 = nextGenerationGPU(step0)
```

Figure 7. Generating the Life program by quoting the CPU program and calling the `accelerate` quotation compiler.

```
array-prog = fun (arr-var : 'db[,]) -> arr
arr
  = Array2.init const const (fun idx-var idx-var -> elem-exp)
  | arr-var
  | ...                                (See notes)
elem-exp
  = arr-var.(idx-exp,idx-exp)
  | const
  | (+,-,*,/,%,&&,||,min,max,=,>,<,<=,>=,<>)  elem-exp*
  | if elem-exp then elem-exp else elem-exp
  | ...                                (See notes)
idx-exp
  = idx-var
  | idx-exp (+,-) const (%) const    (Shift/Rotate)
  | ...                                (See notes)
const =   any numeric constant
```

Figure 8. The embedded language of data-parallel array comprehensions accepted by our prototype of `accelerate`. The input must be well-typed. All syntax categories include the use of F# quotation macros and let-bindings (see §2.8), translated by substitution.

```
val accelerate: ('a[,] -> 'a[,]) expr -> 'a[,] -> 'a[,]
```

Fig 7 shows the use of this with a quotation of the `nextGeneration` function. Note that Fig 7 achieves essentially the same thing as Fig 6 in just one line.

4.2 The FGPU language

The language accepted by our prototype of `accelerate` is shown in Fig 8.[13] This language clearly relates to a subset of APL data-parallel programs [15]. In future work we will seek to extend this language to a richer and more complete subset of F#.

Programs such as those in Fig 7 are handled correctly by `accelerate`. This is a form of *dual mode* execution where the same program text can be run as either a CPU program or a data-parallel GPU quoted program.

However, while dual-mode execution is convenient it comes at the cost of semantic precision, as GPUs do not implement true 32-bit IEEE arithmetic. We return to this topic in §6. Furthermore dual-mode execution may only make sense if substantial programs are already available, in a form suitable for GPU compilation through the `accelerate` function.

We have not, as yet, verified that the performance gains over C++ [31] also apply to code generated via F# quotations. However, given that the cost of quotation processing can be easily amortized we see no reason why this should not be the case.

For completeness we show part of the implementation of the `accelerate` function. This portion specifies the translation of the binary operators of this language is shown below. This emphasizes the role that raw quotation literals (see §2.4) play in the specification of declarative quotation processors:

[13] There are some minor discrepencies with our implementation, e.g. rotation is handled by a more adhoc case.

```
let rec compileComprehension arrv sz1 sz2 v1 v2 eleme =
  // Case (fun i j -> ...  op <l> <r> ... )
  match
   List.first
     (compileBinOp arrv sz1 sz2 v1 v2 eleme)
     [ <@@ max  @@>, (fun l r -> FPA.Max(l,r));
       <@@ min  @@>, (fun l r -> FPA.Min(l,r));
       <@@ (||) @@>, (fun l r -> FPA.Max(l,r));
       <@@ (&&) @@>, (fun l r -> FPA.Min(l,r));
       <@@ (+)  @@>, (fun l r -> FPA.Add(l,r));
       <@@ (-)  @@>, (fun l r -> FPA.Sub(l,r));
      ]
     with
  | Some res -> res
  | _ -> ...
```

5. Runtime Native Code Generation from F# Quotations

For completeness we show the generation of code from F# quotation values. We emphasize that staged computation provides a more controlled framework for arbitrary code generation. However, even without such a framework runtime code generation or interpretation may be a necessary workhorse when implementing portions of an embeddded language, e.g. when post-processing results being returned from a database connection. Furthermore, staged computation has not yet been implemented in combination with native-code generation, and the code generated by the LINQ dynamic compiler has a very special combination of properties:

- It is compiled and run as native code, via JIT compilation to CIL;
- It is garbage collected, through the use of .NET "light-weight code generation".[14]

In many long-lived applications the GC of generated code is essential, so much so that programs work around the lack of code GC by spawning new processes or .NET application domains. Thus this combination makes an excellent example of how non-trivial execution machinery can be leveraged by F#. It also, we hope, acts as a motivation for the implementation of staged compilation on the .NET platform.

A native code generator for a large subset of quoted F# code can be implemented by projecting to the LINQ `Expression<T>` type, here using the function `proj2` mentioned in §3. We then call the corresponding LINQ code generator `Expression.Compile`, generating a .NET delegate value which we explicitly invoke.

```
val compile : ('a -> 'b) expr -> ('a -> 'b)
let compile q =
  let f = (proj2 q).Compile() in
  fun x -> f.Invoke(x);;

(compile <@ fun x -> x + 3,3 @>) 3;;
val it : (int * int) = (6,6)
```

While program generation is easier with F# quotations than with other comparable techniques available on the .NET platform, it is substantially more difficult than in properly staged systems such as MetaOCaml [29]. For example, when generating code for the prototypical recursive `pown` function we must either build the quotation terms programmatically or manually apply beta-reduction optimizations to the generated code during compilation. If the latter is done then `pown` can be generated for a given `n` as follows:

[14] This is an under-emphasized feature of the .NET Common Language Runtime originally designed and implemented by Russi, Meijer and Pobar [23]. We know of no commercial-quality native-code generating virtual machine that provides a similar feature, though would be delighted to hear otherwise.

N	F# pow (native)	F# pown (native)	F# pown (LINQ)	ocamlopt pow (native)	ocamlc pow (byte)
1	0.56	0.12	2.45	0.78	16.9
5	1.40	1.34	4.09	2.56	32.5
10	3.40	3.06	7.62	5.92	52.1
100	49.6	51.8	56.6	71.4	392.6

Figure 9. Execution time (sec) for 10^8 executions of the `pow` and `pown` functions, 3Ghz Pentium 4. Native `pown` represents the hand-expanded and statically compiled version of the code generated by quotation programming.

```
let rec pown n =
 if n = 0 then <@ fun x -> 1 @>
 else (<@ fun x -> x * _ x @> (pown (n - 1)))
```

Based on our initial tests, .NET native code compilation gives decent performance for generated code. For example, the execution times of the following comparative program for varying N is shown in Fig 9. We also show comparative times for OCaml native and byte code.

```
let rec pow (acc:int) n x =
  if n = 0 then acc else  pow (x * acc) (n-1) x;;

// LINQ-generated
let p = pown N in for i = 1 to 100000000 do p 1 done;;
// Statically compiled
for i = 1 to 100000000 do pow 1 N 3 done;;
```

As it happens, for F# the figures show a marginal slow-down for the generated code, which is a linear sequence of multiplications, in contrast to the accumulating loop of the static `pow` function. This indicates that the `pow` function has been well-compiled with excellent branch-prediction for the JIT-generated native code. The slower relative performance of LINQ-compiled code in the case of $N = 1$ indicates that dynamically generated code has a small per-call overhead (approx 0.02μs/call). Not all Core ML expression constructs can be dynamically compiled via LINQ: for example the LINQ expression language lacks a recursion construct (though can call existing recursion operators) and will not compile tailcalls as such.

6. Related Work and Future Challenges

We have shown three applications of meta-programming where we have used F# to access three heterogeneous execution components available in the context of .NET, using a modest meta-programming extension called F# quotations.

Most other meta-programming research focuses on execution "within" a language implementation, through the use of code generation. Our primary contribution is to demonstrate the importance of interoperable heterogeneous meta-programming in the context of ML, particularly in cases where the execution machinery components are external to a language implementation. We also consider this a starting point for the application of disciplined techniques for meta-programming in the context of .NET, and a partial rational reconstruction of the meta-programming being adopted in the Microsoft LINQ initiative.

This work does not claim to be a contribution to the *theory* of meta-programming, but is an exploration of its importance in the context of a framework and language where numerous interesting heterogeneous execution components are readily available.

Above all it is driven by the belief that if you're going to do meta-programming and symbolic manipulation in the context of .NET, you may as well do it with one of the best symbolic programming languages available on the platform.

The three applications we have outlined all require additional development and refined specifications before they consitute polished domain-specific languages. In future work we plan to continue to the development and refinement of the components and to focus on the potential application of staging techniques and type systems to these domains.

6.1 Toward Semantically Clean Heterogeneous Meta-programming?

The author approaches meta-programming with the usual suspicions of an ML advocate: some uses of meta-programming are not compelling; combinators are often a sufficient substitute; core ML itself is itself an excellent meta-language; and over-zealous use can play havoc with our abilities to reason about programs.

In the context of typed functional languages recent work has been primarily focused on staged-computation systems that are very careful to preserve the equational properties of the language at all levels. This is not something we have attempted to do in this paper. Staged computation works especially well in the context of code-generation and compile-time macros. Relevant systems include Meta ML and its derivatives, MacroML [12], TemplateHaskell [25] and the distributed language MetaKlaim [11].

Recent work has shown how to extend staged compilation to include the generation of C code through "off-shoring" — a form of "implicitly heterogeneous" execution [8]. It is interesting to consider the challenges and pre-requisites for the application of implicitly heterogeneous staged computation in the domains of use explored by this paper. For example, we would need formal (and preferably machine-checkable) specifications of the execution semantics implemented by devices such as databases and GPUs. However, machine-checked proofs of correctness for the implementations of compiler extensions would themselves require extensive meta-programming in the form of theorem proving, a long standing interest of the author [26]. A form of intensional meta-programming has already been used to prove properties of F# programs that implement web services [4].

Disturbingly, two of the three target execution techniques (GPU, SQL) have, by specification, inconsistencies and quirks which mean they are *not* perfect off-shoring executors of F# code. For example, GPUs do not implement precise 32-bit IEEE floating point, by the very nature of the hardware used, and indeed, many GPUs come without a precise specification of their numerical characteristics, though that doesn't make them useless as computational devices. Additionally, the LINQ-SQL specification lists a number of ways in which the SQL code generated does not implement precisely the same semantics as the unquoted versions of the C# meta-programs. Pragmatically speaking, the existence of these heterogeneous execution components leaves the semantically-minded language provider with an unenviable choice: ban the use of such components within a meta-programming framework in order to preserve semantic purity; make their use difficult (e.g. by forcing the use of programmatic APIs or meta-programs which manipulate different types and thus reveal all semantic distinctions); or permit their use via intensional meta-programming (despite the existence of minor semantic discrepancies). For this paper we chose to demonstrate the latter route.

6.2 Other Related Work

The Accelerator library is by Ogelsby, Tarditi and Puri [31]. Compiling array aggregate expressions has a long history starting with APL [15]. LINQ originates from the C# design group led by Hejlsberg. Many projects have implemented language-integrated database bindings for high-level programming languages: indeed, the relative lack of such bindings for means ML is the exception rather than the rule [18, 22, 5].

Acknowledgments

We thank in advance the referees for their comments and observations. We also thank Greg Neverov, Jose Ogelsby, James Margetson, Gavin Bierman, Simon Peyton-Jones, Nick Benton and Ralf Herbrich for helpful discussions related to this work.

References

[1] B. Beckman, G. Bierman, and E. Meijer. LINQ: Reconciling Objects, Relations, and XML In the .NET Framework. In *Accepted to appear in SIGMOD*, 2006.

[2] N. Benton, A. Kennedy, and C. V. Russo. Adventures in interoperability: the SML.NET experience. In *PPDP '04: Proceedings of the 6th ACM SIGPLAN International conference on Principles and Practice of Declarative Programming*, pages 215–226, New York, NY, USA, 2004. ACM Press.

[3] P. N. Benton, A. J. Kennedy, and G. Russell. Compiling Standard ML to Java bytecodes. In *3rd ACM SIGPLAN International Conference on Functional Programming*, September 1998.

[4] K. Bhargavan, C. Fournet, A. Gordon, and S. Tse. Verified interoperable implementations of security protocols. In *Computer Security Foundations Workshop (CSFW)*, 2006.

[5] G. Bierman, E. Meijer, and W. Schulte. The essence of data access in Cω. In *Proceedings on the 19th European Conference on Object Oriented Programming*, pages 287–311, July 2005.

[6] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Rec.*, 23(1):87–96, 1994.

[7] K. Czarnecki, J. O'Donnell, J. Striegnitz, and W. Taha. Dsl implementation in metaocaml, template haskell, and c++.

[8] J. Eckhardt, R. Kaiabachev, E. Pasalic, K. Swadi, and W. Taha. Implicitly Heterogeneous Multi-stage Programming. In *Proceedings of GPCE '05*, 2005.

[9] ECMA International. ECMA Standard 334: C# language specification. See `http://www.ecma-international.org`.

[10] ECMA International. ECMA Standard 335: Common Language Infrastructure. See `http://www.ecma-international.org`.

[11] G. Ferrari, E. Moggi, and R. Pugliese. Metaklaim: a type safe multi-stage language for global computing. *Mathematical. Structures in Comp. Sci.*, 14(3):367–395, 2004.

[12] S. E. Ganz, A. Sabry, and W. Taha. Macros as multi-stage computations: type-safe, generative, binding macros in MacroML. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN International Conference on Functional Programming*, pages 74–85. ACM Press, 2001.

[13] M. Gordon, R. Milner, and C. Wadsworth. Edinburgh LCF: A mechanised logic of computation. In *Lecture Notes in Computer Science*, volume 78. Springer Verlag, 1979.

[14] J. Grundy, T. Melham, and J. O'Leary. A reflective functional language for hardware design and theorm proving. Technical Report PRG-RR-03-16, Oxford Univerity, Computing Laboratory, 2003.

[15] L. J. Guibas and D. K. Wyatt. Compilation and delayed evaluation in APL. In *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 1–8, New York, NY, USA, 1978. ACM Press.

[16] A. J. Kennedy and D. Syme. Design and implementation of Generics for the .NET Common Language Runtime. In *Programming Language Design and Implementation*. ACM, 2001.

[17] B. A. LaMacchia, S. Lange, M. Lyons, R. Martin, and K. T. Price. *.NET Framework Security*. Pearson Education, 2002.

[18] D. Leijen and E. Meijer. Domain specific embedded compilers. In *2nd USENIX Conference on Domain Specific Languages (DSL'99)*, pages 109–122, Austin, Texas, Oct. 1999. Also appeared in ACM SIGPLAN Notices 35, 1, (Jan. 2000), see also `http://www.haskell.org/haskellDB/`.

[19] D. Luebke, M. Harris, J. Krüger, T. Purcell, N. Govindaraju, I. Buck, C. Woolley, and A. Lefohn. Gpgpu: general purpose computation on graphics hardware. In *GRAPH '04: Proceedings of the conference on SIGGRAPH 2004 course notes*, page 33, New York, NY, USA, 2004. ACM Press.

[20] Microsoft Corporation. The LINQ May 2006 Preview, 2006. See `http://msdn.microsoft.com/data/ref/linq/`.

[21] M. Odersky, M. Sulzmann, and M. Wehr. Type Inference with Constrained Types. *TAPOS*, 5(1), 1999.

[22] J. A. Orenstein and D. N. Kamber. Accessing a relational database through an object-oriented database interface. In *VLDB '95: Proceedings of the 21th International Conference on Very Large Data Bases*, pages 702–705, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[23] J. Pobar and D. Russi. Late-bound invocation notes and Lightweight Code Gen, 2004. MSDN blog entry, See `http://blogs.msdn.com/joelpob/archive/2004/04/01/105862.aspx`.

[24] T. Sheard, Z. Benaissa, and E. Pasalic. DSL implementation using staging and monads. In *Domain-Specific Languages*, pages 81–94, 1999.

[25] T. Sheard and S. P. Jones. Template Meta-programming for Haskell. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16, New York, NY, USA, 2002. ACM Press.

[26] D. Syme. *Declarative Theorem Proving for Operational Semantics*. PhD thesis, University of Cambridge, 1998.

[27] D. Syme. ILX: Extending the .NET Common IL for Functional Language Interoperability. *Electronic Notes in Theoretical Computer Science*, 59(1), 2001.

[28] D. Syme and J. Margetson. The F# website, 2006. See `http://research.microsoft.com/fsharp/`.

[29] W. Taha and other contributors. MetaOCaml: a compiled, type-safe multi-stage programming language, 2006. See `http://www.metaocaml.org/`.

[30] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1–2):211–242, 2000.

[31] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: simplified programming of graphics processing units for general-purpose uses via data-parallelism. Technical Report MSR-TR-2005-184, Microsoft Research, December 2005.

[32] P. Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 307–313, New York, NY, USA, 1987. ACM Press.

[33] M. Wand. The theory of fexprs is trivial. *Lisp and Symbolic Computation*, 10(3):189–199, 1998.

[34] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 224–235, New York, NY, USA, 2003. ACM Press.

A. LINQ and LINQ Meta-programming

Part of the background to this paper is the rapid adoption by Microsoft of the use of meta-programming in the Microsoft LINQ extensions for C# 3.0 and Visual Basic 9. We briefly summarize this project, partly because one of the aims of meta-programming in F# is to ensure that F# programmers can take advantage of the "high-value" components being developed in the context of LINQ. LINQ is described by Microsoft as

> a set of extensions to the .NET Framework that encompass language-integrated query, set, and transform operations. It extends C# and Visual Basic with native language syntax for queries and provides class libraries to take advantage of these capabilities...

Part of LINQ is a set of modifications to the Java/.NET nominal OO model to permit functional/aggregate operations on data structures. These modifications include syntactic lambda-expressions, an extensible dot-notation and a modicum of local type inference, giving C# 3.0 program fragments such as:

```
var shortDigits = digits.Where(d => d.Length < 10);
```

where the environment contains (in pseudo-ML syntax):

```
digits : IEnumerable<string>;
.Length: string -> int
.Where : (IEnumerable<α>, Func<α,bool>)
          -> IEnumerable<α>
```

The "extension method" `Where` is better known to functional programmers as a filter.

As a result of LINQ the .NET platform will include a standard DLL `System.Query.dll` containing the definition of a type `Expression<T>`, a representation of typed expression syntax trees for both C# and Visual Basic expressions. This type and the libraries which accept and manipulate values of this type constitute the meta-programming layer of .NET we seek to leverage in §3 and §5 of this paper. C# syntactic lambda expressions are implicitly reified whenever they are used as a value of type `Expression<Func<A,B>>`.[15] For example, consider

```
var shortDigits = db.Where(d => d.Length < 10);
```

where the environment contains:

```
db: IQueryable<string>
.Where: (IQueryable<α>, Expression<Func<α,bool>>)
        -> IQueryable<α>
```

The call to `Where` uses the underlined syntactic lambda expression as type `Expression<Func<string,bool>>`, so the expression is passed as an abstract syntax tree (for clarity we use underlining wherever C# code is implicitly reified). In this example `db` represents a handle to a database table and `IQueryable` represents both a table (in the case of `db`) and a composed query over the table (in the case of the overall expression).

The LINQ SQLMetal tool builds type-annotated object models for database schema. For example, if a schema contains table `Employees` with columns `City` and `Address` a set of class definitions is generated, building an environment:

```
type Employees : IQueryable
.City: Employees -> System.String
.Address: Employees -> string
```

The reification of LINQ is a fairly limited facility: only expressions may be reified, and the LINQ expression type contain no nodes for recursion. Control constructs are generally absent since in OO languages they belong to the world of statements.

LINQ reified expressions must be closed apart from the use of constructs such as named types, methods, properties and fields, an approach we also use for F# quotations. Other expressions may be "spliced into" expression trees, and ground values may be lifted to expressions. Libraries are free to do what they want with quoted expressions.

[15] `Func<_,_>` is one of several representations of arity-1 function values on the .NET platform.