On Meta Programming and Code Generation in F# *

Pouya Larjani
Department of Computing and Software, McMaster University
larjanp@mcmaster.ca

Abstract

Meta programming is used to write programs that manipulate other programs. In this paper, we are interested in the use of meta programming for code generation generally and in syntactically correct, type safe methods for manipulating code fragments specifically. We require a strictly typed programming language with self-hosting meta programming support. Some techniques for type safe code generation are developed in the MetaOCaml programming language and have been published before. We develop the required machinery and techniques in the F# programming language in a similar manner using the support for typed code quotations in the F# programming language and the LINQ (Language Integrated Query) features of the .NET platform. This machinery allows us to compose code fragments that are syntactically correct with type and scope safety guarantees. We also develop a domain-specific language for the construction of code generators using F# workflows.

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

General Terms Code Generation

Keywords F#, Meta-Programming, Code Generation, Domain Specific Languages

1. Introduction

Throughout the history of computer programming many different definitions have been given for meta programming – such as manipulating strings representing code or handling representations of abstract syntax trees, template meta programming, and multi stage code compilation [12, 13, 20] – but they all share the same common goal: To manipulate or reason about other programs[1]. With advancements in the concepts of programming languages we see more tools and machinery available for writing better meta programs. In this paper we are concerned about *language integrated* meta programming, where the components of a meta program are available constructs inside the language itself and there is no more need of a meta language for writing such programs. In this case "meta program" is perhaps a misnomer and a suggested better name is *multi stage* programming [4, 19] where the focus is on multiple stages of compilation for obtaining code. In this paper we will assume the two terminologies are interchangeable.

Language integrated meta programming has three main components as used in List (see [1] for full description):

1. **Quote**: (quasi-quote) To obtain a representation of the code inside the quotation as a value in the program.
2. **Eval**: To run the piece of code that a value represents.
3. **Splice**: To compose different fragments of code together.

The splicing concept is the more interesting of these operations and deserves more explanation. The real power of meta programming lies within the splice operation for composing code fragments together. One can modify code and produce new combinations of code fragments using the splice operation and perform computations on code such as variable renaming, partial code compilation [14], or even optimizations by replacing specialized routines depending on the context [3, 4].

Of course having such power over code generation may lead to some undesirable results where the produced code is meaningless, but with the development of more organized meta programming components integrated into languages such as MetaML [20], MetaOCaml [12], Haskell [13] and F# [15] we have access to more attractive features such as static guarantees and type safety of splicing to ensure that the produced code fragments make sense (from the compiler's perspective). With more careful constructs and proper operations we may also convince a person that the produced code is correct as well.

1.1 Outline

The focus on this paper is on techniques for provably correct code generation in F# [16]. We will discuss the implementation of some important features of multi stage programming from [19] in F# as well as other developed techniques (such as those in [4]) for the generation of code.

In Section 1.2 we will have a brief overview of the syntax of meta programming elements in F# and its key similarities and differences with other languages that have integrated meta programming features. The next section discusses in detail the features of meta programming included and missing in F# quotations and explains the improvements and combinator implementations to overcome the shortcomings of the base implementation. This section contributes two quotation transformers for uniquely renaming variables and code reduction as well as a pretty-printer for F# quotations. Section 3 demonstrates the construction of an extensible and flexible code generator using many of the features included in F# ([10] for full features) such as computation expressions (i.e. monads), customized pattern recognizers ([17] Chapter 9), and LINQ expression trees [2]. This section contributes a framework for code generation and a workflow for generating code with continuations

* This work was supported by NSERC.

[1] The "other" program may even be the code to the meta program itself, in case of self-modifying programs.

as well as a useful example of how we can benefit from the additional functionality. Section 4 contains the major contributions of multi stage programming to code efficiency and the future work that can be done to further improve the functionality.

1.2 Meta Programming Elements in F#

Don Syme introduced the meta programming components of F# programming language and its uses in generation of dynamic queries and data parallel programs in his paper "Leveraging .NET Meta-programming Components from F#" [15]. This paper will build on the concepts introduced by Syme. The basic syntax of F# is very similar to other languages in the `ML` family and we will assume the reader has some familiarity with it. For a more detailed view of F# language please refer to [16]. The main point of comparison for meta programming in this paper will be MetaOCaml, see [6] for comparison with other meta programming languages such as Template Haskell and C++.

A typed code quotation in F# is constructed by surrounding a syntactically valid and type correct code fragment in `<@` and `@>` symbols (similar to the `.< ... >.` construction in MetaOCaml). F# quotations have have the syntax and grammar rules of F# language (see [10]); additionally, they are augmented with typing information of the quoted code and therefore valid compositions of code quotations are guaranteed to be type-correct [15]. In that respect, F# quotes are similar to MetaML [20] and MetaOCaml [12] — but unlike Lisp quotations which are untyped and are simply lifted data lists without requiring them to be executable[2]. The quotes are of type `Expr<'a>` (alternatively denoted by `'a Expr` which is more familiar for OCaml readers) where `'a` is the type of the inner expression.

The splice syntax in F# is done through the `%` operator inside of a quotation similar to `.~` in MetaOCaml. The "holes" in a quotation are typed, and the code that is being spliced at a location must match that type. This ensures that the generated quotation after splicing is still well typed as well as syntactically valid. For example, the function

```
> let Seq a b = <@ begin %a; %b end @>;;
val Seq : Expr<unit> -> Expr<'a> -> Expr<'a>
```

is s sequencing operator and produces code that sequences two other code fragments after each other. Notice the first code fragment must have type `unit` due to the semantics of `;` and the final type of the sequence expression is the type of the second code fragment.

Unlike the splicing in MetaOCaml, the splice operation in F# does not avoid variable capture and no variable renaming is performed during splicing. This can lead to unsafe code generation when multiple code combinators use the same variable name in their generated code, or even the case of applying the same code combinator multiple times. In the following sections we will develop the quotation transformers that perform renaming of variables to create unique names during splicing and solve the variable capture problem.

There is no immediate equivalent of the "run" operator (such as `.!` in MetaOCaml) in F#. We can execute a quotation using the LINQ expression compiler by calling the `Expr<'a>.Eval()` method. Unlike the quotation evaluation of MetaOCaml which is a pure compile-time operation, F# quotations can be compiled and run on demand at any stage. This allows for greater flexibility and partial evaluation during run-time. Section 2.4 discusses how we can use the LINQ compiler to our advantage in multi stage code generation.

[2] F# has untyped code quotation with `<@@` and `@@>` style of quoting, but they do not have the same semantics as Lisp quotes.

F# quotations are algebraic data types defined as an expression tree augmented with attributes and typing information. This is quite similar to the definition of quotations in Template Haskell and can be explicitly created using generator functions and analyzed using pattern matching — unlike MetaOCaml's approach of representing code quotations [20].

The internal tree representation of a quote in F# is defined through the algebraic type:

```
type Tree =
    | CombTerm   of ExprConstInfo * Expr list
    | VarTerm    of Var
    | LambdaTerm of Var * Expr
    | HoleTerm   of Type * int
```

The type `Var` is defined as the name of a variable (as a string) bundled with the type of the variable and `ExprConstInfo` type carries various information about different expressions. While the `Expr` type utilizes the `Tree` type internally, this structure is not exposed to the users of the `Expr` type outside of the module and instead F# uses quotation generator methods and active pattern recognizers [18] to safely compose or decompose quotations.

Since the internal representation of an expression tree is low-level (almost similar to Lisp's method of using s-expressions to represent a quote [1]), it is possible for users to generate an expression tree which does not represent any valid code. As we will see in the next section one of the main requirements of generating correct code is for the quotation syntax to be grammatically correct — which could easily be violated if the code generators have access to the internal expression tree.

To satisfy this requirement, the `Expr` type only allows construction of expressions through a series of specialized construction methods that only produce proper[3] expression trees. For example, the method:

```
Expr.Sequential : Expr * Expr -> Expr
```

produces proper expressions representing a sequential operator similar to the example above.

The `Quotations.Patterns` module defines a collection of active pattern recognizers for deconstruction of the `Expr` data type. These pattern recognizers allow high-level pattern matching using the proper expression format over the internal tree representation of the quotation. The name of the generator method for each high-level quotation construct is often the same as the name of the pattern recognizer for it, and we will be using both of them throughout this paper. The type and definition of a method (such as `Sequential` method above) differs according to the context of its appearance whether it is a constructor or a deconstructor (pattern match). For more information refer to [15] for expression types and [18] for active pattern recognizers.

These quotation constructor methods and pattern recognizers in the F# library provide us a rich environment for generating and analyzing syntactically correct code quotations.

2. Features of Multi-Stage Programming

A multi stage program (meta program) has many advantages to the direct implementation of the same program such as having the flexibility of choosing from different specializations of a routine depending on the current task at hand and without using abstraction layers. Another advantage of multi stage programs is the ability to partially compile code during the different stages of compilation and/or execution according to the currently available data

[3] Improper expressions are trees that conform to the `Tree` data structure, but do not denote any valid syntax.

[4]. While being advantageous in many respects, multi stage programs also have the caveat of being more complex to program and have the possibility of composing wrong code mixtures from correct smaller code fragments, as in Section 2.1 for example. In this section we discuss how we can take better advantage of the available features and how to simplify the code generation process while avoiding the mistakes that produce incorrect code.

In general, we can break down the requirements of correct code generation into following steps:

1. Syntax correctness: Every code quotation must conform to the syntax rules of the language and be guaranteed to parse correctly.
2. Type correctness: Code quotations must confer to the language's typing rules and be guaranteed to compile correctly.
3. Scoping correctness: Variables do not escape their scopes at run time and do not cause any naming clashes in the code.
4. Semantic correctness of combinations: Each code combinator has a guarantee (formally as a proof or informally as documentation) to perform the correct action and generate code quotation that conforms to these requirements.

We can gain concise and correct syntax by the fact that code quotations are parsed by the compiler in multiples stages and each quotations must adhere to the language syntax rules and parsed into an abstract syntax tree. The type correctness is also a built-in feature of languages with strongly typed quotations. The next 2 items are the main sources of discussion in the following sections.

2.1 Variable Captures and Renaming in Splicing

A variable capture occurs when name of a variable in scope is reused in a sub-expression for a different purpose. Such name clashes are common issues when using code combinators to construct meta programs and are not always avoidable by the programmer.

A common example for variable capture is when a code combinator is applied twice. Consider the following function:

```
let Fun (f: Expr<'a> -> Expr<'b>) : Expr<'a->'b> =
    let v = Var.Global("t", typeof<'a>)
    Expr.Lambda(v, f (v |> Expr.Var |> Expr.Cast))
    |> Expr.Cast
```

Which wraps a (code transformer) function inside a code quotation, i.e.

```
(Fun >> Eval) f = (Lift >> f >> Eval)
```

This combinator introduces a new variable named `t` as argument of the lambda function. A nested application of this combinator now generates a naming clash[4]:

```
let add = Fun (fun x ->
              Fun (fun y -> <@ (%x) + (%y) @>))
val add : Expr<(int -> int -> int)> =
    <@ fun t -> fun t -> (t)+(t) @>
```

As we can see this is clearly not the intention of the program and the naming clash between the two calls to the combinator is semantically altering the generated code. Such issues always arise when the combinator is generating code that introduces a new variable without having sufficient information about the context of the external expressions and the currently used names — commonly used with *let* bindings and lambda abstractions as well as the imperative *for* loops. Although in the sample combinator defined above the naming clash can be avoided with a minor adjustment, this issue is rectified when we further define more combinators causing possible naming clashes and introduce the continuation-passing style in Section 3 and we require a more robust and versatile solution.

MetaOCaml and MetaML automatically handle variable renaming when splicing code by creating unique names such that the name clashes and variable captures do not occur [19], but F# does not automatically rename variables and leaves it to the programmer to ensure uniqueness of variable names. Therefore we need to develop the machinery in code combinators to produce unique names when it is necessary.

The approach in this paper is to construct a function called `MakeUnique` that renames the variables (including function arguments, let bindings, and for loop counters) in an expression before splicing takes place. In the example above, making all variables unique would produce the code
`fun t_1 -> fun t_2 -> (t_1)+(t_2).`

This function can be defined by structural recursion on the data type defining code expressions and a local dictionary of unique names[5]:

```
let unique: Var -> Var = ...
// Ensures given variable is unique
let subs: Var -> Var -> Expr -> Expr = ...
// Applies a variable renaming to an expression

let MakeUnique = function
    | Var v -> Expr.Var (unique v)
    | Lambda (n, e) -> let n' = unique n
                       Expr.Lambda(n', subs n n' e)
    | Let (n, v, e) -> let n' = unique n
                       Expr.Let(n', v, subs n n' e)
    | e -> e
```

Using the `MakeUnique` function in the code combinators solves the issue with variable captures. This is an essential factor in correctness of the generated code to obtain scoping safety in addition to the type safety and syntax correctness provided by meta programming features of F#. We can further on simplify the process of adjusting variable names by defining an expression transformer:

```
let rec TraverseExpr f e = ...
// Traverses the expression and maps the given function to each
term

let Rename (x: Expr<'a>) : Expr<'a> =
    TraverseExpr MakeUnique x |> Expr.Cast
```

which is called at every stage of code generation inside combinators for unique renaming of variables.

For efficiency purposes, the `MakeUnique` transformer contains a local dictionary of the uniquely generated variable names such that an expression is never renamed twice in the process of code generation.

2.2 Code Reduction at Compilation

Partial evaluation in one of the more attractive benefits of multi stage programming for code optimization. If at a certain stage of code generation all the data for evaluation of a sub-expression is readily available, the code generator can reduce the entire sub-expression to the result of its evaluation (i.e. partially evaluate the program) to improve the post-compilation execution time at the cost of performing the evaluation at the current stage.

[4] Due to an unfortunate naming clash for the splicing operator, we put the splice within brackets (such as `(%x)`) to avoid parsing the operation as the binary modulus operator which has precedence over splice.

[5] The full code for this paper can be accessed at http://www.cas.mcmaster.ca/~larjanp/fscodegen

The reduction can only occur during splicing of expressions. When a code combinator is generating a new code fragment by splicing other expressions it can evaluate the sub-expression fully if all the data is present prior to splice operation – for example reduce a function call which has all of its arguments evaluated to the result of the call. We will see in Section 3 how this operation can be improved further by flow of information in the code generation monad.

Another weakness of the splice operation in F# in comparison with MetaOCaml is exposed when attempting the code reductions as explained above. F# splice (%) operation does not automatically perform the desired reduction and needs the data to be prepared prior to splicing to apply the proper reduction. For example the *apply* operation for code is defined in MetaOCaml:

```
# let app f x = .< .~f .~x >.;;
val app : ('a, 'b -> 'c) code ->
  ('a, 'b code) -> ('a, 'c) code
# app .< fun x -> x + x >. .<1>.;;
- : ('a, int) code =
  .<((fun x_1 -> (x_1 + x_1)) 1)>.
```

but the same definition in F# does not produce equivalent code:

```
> let app f x = <@ (%f) (%x) @>;;
val app : Expr<('a -> 'b)> -> Expr<'a> -> Expr<'b>
> app <@ fun x -> x + x @> <@ 1 @>;;
val it : Expr<int> =
  <@ SpliceExpression (fun x -> x + x) 1 @>
```

In the above example, the desired reduction did not take place during the code generation time and the computation is delayed to the runtime call to `SpliceExpression` function to perform the beta reduction — even if `SpliceExpression` is identity at run time, it should be performed during compilation time since all the information to reduce the expression is present at compilation. In fact, the current quotation evaluation using LINQ (Section 2.4) is not able to correctly execute this code and results in a run-time error for a first class usage of splicing. Until this issue is fixed in F#, we can use an alternative method of creating another quotation transformer which automatically translates `SpliceExpression` expressions into appropriate function applications which can be beta reduced at evaluation stage. Similar to the `Rename` transformer developed in Section 2.1 we may define a `Reduce` transformer for this purpose, but we can also take an alternative approach to avoiding this issue while applying the other transformations: Our goal is to construct the resultant expression such that the superfluous function application is removed — and meanwhile perform the variable renaming mentioned in previous section as well. Or solution is to prepare the representation of the code using proper code combinators instead of quasi quotations.

```
> let Apply (f: Expr<'a->'b>) (x: Expr<'a>)
      : Expr<'b> =
      Expr.Application(Rename f, Rename x)
      |> Expr.Cast;;
> Apply <@ fun x -> x + x @> <@ 1 @>;;
val it : Expr<int> = <@ (fun x_1 -> x_1+x_1) 1 @>
```

An explicit call to `Expr.Application` constructs the type of expression desired. We will use this style of code construction extensively in the following sections in replacement of the built-in splice operation.

The general code reduction expression transformer can be defined recursively similar to the `Rename` transformer. This function will search for an application of the `SpliceExpression` function and reduce that into an application call that can be beta reduced during compilation:

```
let MakeSplice = function
    | Application (Call(None, m, [Value(p,_)]), v)
        when (m.Name = "SpliceExpression") &&
             (p :? Expr) ->
        Expr.Application(p :?> Expr, v)
    | e -> e

let Reduce (x: Expr<'a>) : Expr<'a> =
    TraverseExpr MakeSplice x |> Expr.Cast
```

This can further be improved by replacing the string comparison for method name with a proper comparison of `MethodInfo` objects.

2.3 Code Generation Combinators

The F# quotations provided us with required machinery to ensure the syntactic and type correctness of generated code fragments, and in the last few sections we have developed the required techniques for avoiding unwanted variable captures and allowing compile-time function applications. The next step is to provide appropriate code combinators as the basic building blocks of the code generator (as opposed to directly constructing quotations in the process) such that every combinator produces code from its arguments that adheres to the four requirements defined earlier.

In the discussion code provided in above, some type restrictions and casting is required to ensure the type correctness of the combinators. It is important to note that the base data type for representation of code and its generators (the type `Expr` above) are *untyped* code, and thus the combinators that will be using them directly need additional type constraints for the casting from untyped quotation `Expr` to typed quotation `Expr<'a>` to be valid.

The two most basic combinators are `Lift` and `Eval`, where the former lifts a value from its base type to quoted type (similar to quote) and the latter evaluates a quotation from its quoted type into the base type. Ideally, in an end-user application there should not be any quotations present and the only code constructions are through lifting basic values with `Lift` and combining them using other combinators, with a final evaluation from the code domain to the value domain using `Eval`. The evaluation operation will be discussed further in Section 2.4.

In order to achieve this goal, we will need an appropriate combinator equivalent to each of the language constructs, such as *let* statements (without variable naming clashes), function applications (as seen above), lambda abstraction with care for variable captures, sequencing of code fragments (`;` operation), control structure and program flow combinators (*if* statements, *for* and *while* loops, etc...), and some ease-of-use convenience combinators (i.e. lifted references or lists).

```
let Let n (a: Expr<'v>) (f: Expr<'v> -> Expr<'e>)
    : Expr<'e> =
    let v = Variable<'v> n
    Expr.Let(v, a, f (v |> Expr.Var |> Expr.Cast))
    |> Expr.Cast

let Sequence (a: Expr<'b>) (b: Expr<'a>)
    : Expr<'a> =
    Expr.Sequential(Rename a, Rename b) |> Expr.Cast

let While c b =
    let c', b' = Rename c, Rename b
    <@ while (%c') do (%b') done @>

...
```

We will be using these combinators extensively during code generation and provide monadic versions of each in Section 3.

2.4 Printing, Compilation and Evaluation

Readers familiar with F# and its quotations system have noticed at this point that the quotation results of samples above do not resemble any of the output from the F# interactive environment. F# by default prints the algebraic data type representing the code fragment (as in the AST of the code) without any pretty-printing applied. To obtain results that are easy to read and resemble the syntax of F# more closely we need to develop our own quotation pretty-printer to attach to the interactive session. Note that this is purely a cosmetic enhancement during development and presentation, and the existence of this printer has no effect on the execution or the result of any code generator.

```
let rec PrintQuote = function
    | Var v -> v.Name
    | Value(a, _) -> a.ToString()
    | Lambda(v, e) -> "fun " + v.Name + " -> " +
                      PrintQuote e
    | Let(n, v, e) -> "let " + n.Name + " = " +
                      PrintQuote v + " in " +
                      PrintQuote e
    ...
```

The compilation and evaluation process for F# quotations differs greatly from the "run" method (`.!`) provided with MetaOCaml. Evaluation of a quote in F# is done through external functions that produce and compile LINQ computation expressions [2] as opposed to a built-in run operation of MetaOCaml [19]. The `Expr<'a>.Eval` method is defined in the `Microsoft.FSharp.Linq.QuotationEvaluation` [9] library and is executed at the end of each stage in computation and generation of code. Thus the combinator `Eval` is defined as:

```
let EvalLINQ (x: Expr<'a>) : 'a = x.Eval()
```

The LINQ dynamic compiler generates native and optimized code through the .NET JIT (Just-In-Time) compiler from the expressions trees produced from F# quotations [15]. The performance of this evaluation and benefits and drawbacks are discussed in Section 4.1.

3. Construction of a Code Generator

We have addressed the main topics and concerns with meta programming in F# to generate code. With the ability to generate code fragments that are syntactically valid and strongly typed with guarantees on syntax correctness, type correctness and variable scope correctness, we can now compose code fragments using our combinators to generate derivative programs. One of the benefits of substituting different specializations of sub-programs is to overcome the execution overhead caused by abstraction, but the main power of using code generators lies within the concept of compile time beta reduction for partial evaluation as demonstrated in [3] and [4] for generating specialized solvers for Gaussian Elimination.

Ideally, we would like to have a domain-specific language (DSL) that captures all the required code generation routines in a convenient environment for the developer. In most modern functional programming languages (Haskell, OCaml, and F# for instance) we can eliminate the need for an external DSL by using the special features in language such as development of custom operators for code combinators and special monads for composing code fragments. In [4], Carette and Kiselyov have used a syntactic extension to OCaml (`pa_monad` [5]) to write monadic syntax similar to Haskell. In F# we can use the concept of a *workflow* to develop the required monads (For example, see [17] Chapter 9).

3.1 Continuation-Passing Style

Before we explain the usage of workflows in the code generator, there is one more issue to address: How do we assign names or use code fragments that have not been generated yet? What happens when a combinator that's generating an inner portion of a routine requires the program to have extra parameters, or add definitions prior to execution of this code?

For example when generating a *let*-statement, we would like to have a combinator `<@ let v = %value in %expr @>`, but the `expr` generator must know what the variable bound to `value` is called before generating the code. Readers may have noticed at this point that the definition of the `Let` combinator in previous section required a function of type `Expr<'v> -> Expr<'e>` instead of the expression (where `'v` is the type of the introduced variable) to perform the variable substitution in *expr* prior to using it in the generator. We would like to expand on this concept of passing required information to future generators automatically instead of asking for such substituting functions.

The solution proposed in [14] and [4] is to use a continuation function in our generators using continuation-passing style (CPS). In this case each combinator outputs a continuation that will generate the code when run, and accept other code generator continuations as code parameters. This method is used extensively in [4] to program code combinators and we will follow the same design for our combinators in F#. For full details on benefits and usage of CPS refer to [14].

Before we introduce the idea of workflows, we will first explore the continuation-passing monad as described above. The state-continuation monad used in [4, 14] can simply be defined as the F# type:

```
type StateCPSMonad<'s, 'v, 'w> =
     's -> ('s -> 'v -> 'w) -> 'w
```

Where `'s` is the state, `'v` is the value type, and `'w` is the type of the final answer. In the code generator the type of value and answer are in fact expressions over types `'v` and `'w`. We will not be using the monadic state in any of the combinators and examples in this paper and will only define them here to show the equivalent version of the monad defined in [4] in F#, but the type of the state that used in more advanced examples is a sequence of state variables with type `'s`. Thus, a better definition of the state-continuation monad used for code generation is:

```
type CodeGen<'a, 'v, 'w> =
     StateCPSMonad<seq<'a>, Expr<'v>, Expr<'w>>
```

Followed by the two monadic operations of *return* and *bind*:

```
let ret:  Expr<'v> -> CodeGen<'s,'v,'w> =
    fun a s k -> k s a
let bind: CodeGen<'s,'a,'w> ->
          (Expr<'a> -> CodeGen<'s,'b,'w>) ->
          CodeGen<'s,'b,'w> =
    fun m f s k -> m s (fun s' k' -> f k' s' k)
```

The result of a code generator is a continuation that can produce the code given the initial conditions — which are often the empty state and the *reset* continuation:

```
let Reset s k = k
let Generate m = m [] Reset
```

This gives us a primitive notion of a code generator monad that the referenced texts have proposed. In order to use the added functionality correctly, we will need to define the monadic versions of the combinators from Section 2.3 first.

3.2 Monadic Code Combinators

Let-statements are one of the most important and problematic expressions in code generation. As shown in [14] the main question is when and where to introduce the variable bindings. They must maintain the dependency between the defined variables, but also need to be generated *before* the code fragment that utilizes them. This was the motivating move to use CPS notation so that code generators can see their continuations beforehand. Let statements may also cause variable capture and shadowing issues discussed in Section 2 that needs to be resolved when generating the final statement. Knowing the importance and issues associated with Let statements, it comes as no wonder that we would define them first.

Looking back at the definition of the Let combinator:

```
let Let name (a: Expr<'v>)
    (f: Expr<'v> -> Expr<'e>) : Expr<'e> =
    let v = UniqueVars.GetNew name typeof<'v>
    Expr.Let(v, a, f (v |> Expr.Var |> Expr.Cast))
    |> Expr.Cast
let LetM name a = fun s k -> Let name a (k s)
```

We now realize why the combinator required a function to generate the body of the statement, as it can be lifted easily into the monadic version LetM (of type `string -> Expr<'a> -> CodeGen<'s,'a,'w>`).

We can demonstrate this new code generator with a simple example using explicit calls to ret and bind

```
> Generate <|
    bind (LetM "x" <@ 4 @>) (fun a ->
    bind (LetM "x" <@ 7 @>) (fun b ->
    ret  (Apply (Apply <@ (+) @> a) b)));;
val it : Expr<int> =
  <@ let x_1 = 4 in let x_2 = 7 in (x_1)+(x_2) @>

> Eval it;;
val it : int = 11
```

We called both the variables "x" deliberately to ensure the variable renaming is taking place correctly[6] and applied a simple addition to the two values to generate the body of the let statements. There are many points to improve here for readability: Switch to a Haskell-style monadic syntax that wraps the bind operations in a friendly syntax (such as pa_monad [5] and F# workflows [17]), incorporate LetM into a new variant of binding operator (given that we are not bound to names used in syntax extension), introduce a new combinator Apply2 to make the application of binary function easier, or even introduce the operator-combinator @+ to infix the operation.

```
let Apply2 f x y = Apply (Apply f x) y
```

There are many other useful combinators to define, such as sequencing expressions:

```
let Sequence (a: Expr<'b>) (b: Expr<'a>)
    : Expr<'a> =
    Expr.Sequential(Rename a, Rename b) |> Expr.Cast
let SequenceM a b = fun s k ->
    k s (Sequence (a s Reset) (b s Reset))
```

Or as a different alternative, a combinator needed to append a statement prior to the generated code from the continuation:

```
let PrependM a b = fun s k ->
    Sequence (a s Reset) (k s b)
```

And the primary looping combinator for *while* loops:

```
let While c b =
    let c', b' = Rename c, Rename b
    <@ while (%c') do (%b') done @>
let WhileM c b = fun s k ->
    k s (While (c s Reset) (b s Reset))
```

We will see more of these combinators as we progress in development of the full code generator.

3.3 F# Computation Expressions (Workflows)

As described earlier, workflows are equivalent features to monads in F# with some additional DSL elements as introduced in [17] and [10]. We will be using the workflows in F#[7] to refine the monadic code generator defined earlier. Next we will produce alternate implementations to the code generation combinators that operate on the codegen monad, and lastly a short example of a code generator is presented.

There are many constructs allowed in a computation expression in F# such as *let*, *yield*, *while*, *try*, etc... but before implementing any of these features, we should first explain how a workflow is internally represented. Let us consider the following sample of a workflow (similar to the example given above):

```
sample {
    let! x1 = 4
    let! x2 = 7
    return (x1 + x2)
}
```

The let and return statement above are de-sugared according to the rules of computation expressions into:

```
sample.Delay(fun () ->
  sample.Bind(4, fun x1 ->
    sample.Bind(7, fun x2 ->
      sample.Return(x1 + x2))))
```

The expansion process from the (sugared) workflow notation into the (de-sugared) internal system notation automatically converts consecutive expressions into the continuation-passing style before execution. This greatly improves the readability and development of the code combinators from the notation in Section 3.1.

To further explain the other elements supported in the code generation language, we have defined the following interface for the workflow:

```
type Workflow =
    abstract Delay:
        (unit -> Expr<'a>) -> Expr<'a>
    abstract Bind:
        Expr<'a> * (Expr<'a> -> Expr<'b>)
        -> Expr<'b>
    abstract Return:
        'a -> Expr<'a>
    abstract ReturnFrom:
        Expr<'a> -> Expr<'a>
    abstract YieldFrom:
        Expr<'a> -> Expr<'a>
    abstract Combine:
        Expr<unit> * Expr<'a> -> Expr<'a>
    abstract While:
        (unit -> Expr<bool>) * Expr<unit>
```

[6] in retrospect, these two bindings could come from completely different sources that are not aware of each others naming conventions. In fact, we will completely remove the option of choosing specific names for variables in Section 3.3.

[7] The terms "workflow" and "computation expression" are used interchangeably in the F# literature.

```
        -> Expr<unit>
    abstract Zero:
        unit -> Expr<unit>
```

The entire computation expression is wrapped in a call to the `Delay` method of the workflow, and the `let!` (and `do!`) expressions are translated to the `Bind` method call — however, we left the normal `let` expression untransformed in order to allow the code generator to utilize local variable bindings that are not produced in resultant code. A `yield!` statement is transformed to a call to the `YieldFrom` method which we will primarily use as means to insert code fragments inside the code generator (similar to the `PrependM` combinator defined earlier) whereas sequenced calls to various statements are combined using the `Combine` method (equivalent to `SequenceM` earlier).

The implementation of the code generator workflow is quite similar to the monadic combinators defined earlier:

```
let codegen = {
    new Workflow with
    member cg.Delay v = v()
    member cg.Bind (v,f) = Let "t" v f
    member cg.Return v = <@ v @>
    member cg.ReturnFrom v = v
    member cg.YieldFrom v = v
    member cg.Combine (a,b) = Sequence a b
    member cg.While (c,e) = While (c()) e
    member cg.Zero () = <@ () @>
}
```

We can now used the `codegen` workflow as the DSL for defining the code generators. The example illustrated in Section 3.2 now becomes much more readable and easier to maintain:

```
> codegen {
    let! a = <@ x+1 @>
    let! b = <@ f y @>
    return! <@ (%a)*(%a) + (%b)*(%b) @> };;
val it : Expr<int> =
  <@ let t_1 = x+1 in let t_2 = f y in
     (t_1)*(t_1)+(t_2)*(t_2) @>
```

We have lost the ability of explicitly naming the introduced bound variables with the transformation to workflow syntax (in fact, every variable is now called `t_`*num*), but gained a significant amount of readability and maintainability in the trade-off. It is also important for readers to note that we have deliberately omitted the *state* variable of the state-continuation monad from [4] as it does not apply to the topics in this paper.

Additionally, if we wish to gain more control and flexibility over the continuations in the code generator, we can define a workflow for explicitly binding objects of the type `StateContMonad` from Section 3.2 similar to the definition in [14] and [4]:

```
type CGMonadBuilder() =
    member m.Return a = ret a
    member m.Bind (m, f) = bind m f
let mcodegen = CGMonadBuilder()
```

In which case the example from Section 3.2 becomes:

```
> Generate <| mcodegen {
    let! a = LetM "x" <@ 4 @>
    let! b = LetM "x" <@ 7 @>
    return Apply2 <@ (+) @> a b };;
val it : Expr<int> =
  <@ let x_3 = 4 in let x_4 = 7 in (x_3)+(x_4) @>
```

3.4 Example: The Power Unleashed

Readers may have noticed the theme of all the examples so far has been arithmetic. This was chosen because 1) the operations are familiar for everyone and simple to follow, and 2) it takes a relatively short amount of code to generate the result to demonstrate a construction or a flaw. We will work towards more complex samples of code generation using custom algebras in this section.

Consider the following definition of a ring:

```
type Ring<'t> =
    abstract zero: 't
    abstract one: option<'t>
    abstract add: 't -> 't -> 't
    abstract neg: 't -> 't
    abstract sub: option<'t -> 't -> 't>
    abstract mul: 't -> 't -> 't
    abstract inv: option<'t -> 't>
    abstract div: option<'t -> 't -> 't>
```

If member `one` is not `None` then the ring has multiplicative identity, and if `inv` is defined then the ring also has multiplicative inverses, making it a *domain*. Notice that `sub` and `div` are left optional since it is possible for the program to define them externally in terms of other operations, i.e. :

```
let mksub (r: Ring<_>) =
    match r.sub with
    | None ->
        Some (fun a b -> r.add a (r.neg b))
    | _ -> r.sub
let mkdiv (r: Ring<_>) =
    match r.div, r.inv with
    | None, Some inv ->
        Some (fun a b -> r.mul a (inv b))
    | _, _ -> r.div
```

The ring of integers that we have been using in most of examples so far does not have an inverse or division operation defined. We can implement them as follows:

```
let Ints = {
    new Ring<_> with
    member i.zero = 0
    member i.one = Some 1
    member i.add a b = a + b
    member i.neg a = -a
    member i.sub = Some (fun a b -> a - b)
    member i.mul a b = a * b
    member i.inv = None
    member i.div = None
}
```

Since we are working with code generation, we cannot directly work on the ring of integers, but instead used the ring of *lifted* integers (lifted into the expressions domain), such as:

```
let LiftedInts = {
    new Ring<_> with
    member i.zero = <@ 0 @>
    member i.one = Some <@ 1 @>
    member i.add a b = <@ (%a) + (%b) @>
    member i.neg a = <@ -(%a) @>
    member i.sub = None
    member i.mul a b = <@ (%a) * (%b) @>
    member i.inv = None
    member i.div = None
}
```

Having the required machinery and data definitions, we can shed a new light on the all-familiar power example and introduce a new variant of it: A code generator that outputs the power generator given a ring

```
let rec Power (r: Ring<_>) n b =
    match n with
    | 1 -> b
    | n when n%2 = 1 -> codegen {
        let! b' = r.mul b b
        yield! r.mul (Power r (n/2) b') b }
    | n -> codegen {
        let! b' = r.mul b b
        yield! Power r (n/2) b' }
```

For example:

```
> Fun <| Power LiftedInts 11;;
val it : Expr<(int -> int)> = <@ fun t_1 ->
  let t_2 = (t_1)*(t_1) in
    let t_3 = (t_2)*(t_2) in
      let t_4 = (t_3)*(t_3) in
        (t_4)*(t_2)*(t_1) @>
```

The `Power` function above if a function that given a `Ring<'a>` ring, produces a power function of type `int -> 'a -> 'a`. At this point we have not prepared for raising a value to the power 0 since the ring may not have the `one` elements. If we modify the result of exponentiation function to be of type `option<'a>` we can allow the power 0 of a ring *if* the ring's multiplicative identity is defined. Furthermore we can allow negative exponents when the ring defines a unary division operator:

```
let Power' r n b =
    match n with
    | n when n>0 -> Power r n b |> Some
    | 0 -> r.one
    | n ->
        match r.inv with
        | None -> None
        | Some inv -> Power r (-n) b |> inv |> Some
```

Running it first on the lifted ring of integers which has no division, followed by ring of reals (floats of F#) which can perform division:

```
> Power' LiftedInts -11 <@ 2 @>;;
val it : Expr<int> option = None

> Power' LiftedReals -11 <@ 2. @>;;
val it : Expr<float> option = Some <@
  (1.)/
    (let t_39 = (2.)*(2.) in
      (let t_40 = (t_39)*(t_39) in
        (let t_41 = (t_40)*(t_40) in
          t_41)*(t_39))*(2.)) @>
```

4. Related Work and Improvements

In this paper we have analyzed the issues in meta programming elements of the F# programming language that arise when building a code generator. We have developed the appropriate techniques and machinery to overcome these issues and generate code quotations that match our requirements for producing provably correct code compositions — that is to have correctness of all syntax, semantics, typing and scoping. We achieve syntax and type correctness by using typed code quotations of F# while using the machinery developed in Section 2 to reach the scoping correctness requirement. Semantic correctness of combining code fragments is achieved by using the appropriate code combinators that can be proved correct externally — notice that this only shows correctness of the automatically combinations of other code fragments and not the entire program as a whole.

We then introduced the concept of workflows in F# and used them for construction of a domain-specific language for code generation using both the explicit and implicit continuation-passing style. These workflows allow a more convenient and maintainable method for composing code generators and implicitly use the other combinators and code transformers developed earlier for additional ease of use. We developed two code generator workflows: `mcodegen` for explicit CPS notation and `codegen` for implicit continuations and showed examples of how they can be used and the additional flexibility and ease of use that they offer.

The techniques and generators discussed in this paper can be improved further in many ways:

- The `codegen` workflow as defined here does not carry a state in its computations (unlike `mcodegen`). This is a relatively simple to augment the workflow, for example see [11].
- The `Rename` and `Reduce` code transformers need to be called explicitly inside the combinators when required. Integrating some of their functionality with the F# core may improve readability and development of the combinators and aid avoiding mistakes.
- The quotation printer outlined here is not conforming to F#'s standard of using a `Layout` type for printing objects. Although the output is more readable than F#'s built-in printer for quotes, it can still benefit greatly from the improved layout features in F# interactive.
- Many of the discussions contained performing different actions depending on whether there is static data present (ready to evaluate at the current stage, or already evaluated) or if the data is dynamic (code or data that will be available or evaluated at a later stage) while some of the generators shown earlier required an `option` type for the data to specify whether there is any data present at all. A different approach for handling all cases above is to define a new data type:

  ```
  type data<'a> =
     | DynamicData of Expr<'a>
     | StaticData of 'a
     | NoData
  ```

  All the combinators and monadic workflow operations would operate on this data type instead and produce appropriate static or dynamic information depending on how the arguments are resolved. In cases of mixed data (i.e. sequencing static code and dynamic code) they can be transformed between static and dynamic information using `Lift` or `Eval` operators. This is similar to the options workflow defined in [11] or the `Maybe` monad of Haskell.

4.1 On Efficiency of Dynamically Generated Code

A main concern of switching from the standard coding and abstraction techniques to meta programs is the efficiency of the (run-time) code generation process and the difference between the specialized version generated from a code generator as opposed to hand-written specialization of the program. The contributions of multi stage programming to increasing or decreasing the efficiency of code can be categorized into the following three categories:

1. Run-time code generation and evaluation: The sources of overhead computations performed during the stages of compilation/execution are the contributions from either the code generation combinators and the associated quotation transformers,

or the compilation performed during evaluation phase (a call to `Eval`). Quotation evaluation is a computationally expensive task relative to the other code combinators and it is the programmer's task to decide if the overhead of a one-time evaluation justifies the gained efficiency. As shown in [15] for a different implementation of the power function, the overhead of quotation evaluation becomes negligible as the size of execution increases.

2. Eliminating abstraction layers: A very visible performance gained by using multi stage programming is the elimination of abstraction layers and function call overheads inside the generated code. In the power example in Section 3.4 all of the calls to `Ring<'t>` interface's virtual members were eliminated in the shift from `Ints` to `LiftedInts`, such that a multiplication of two elements of the ring of integers is written as `(x * x)` as opposed to `(r.mul x x)`. Notice that the optimizing compiler cannot inline this function call automatically since `Ring<'t>.mul` is an abstract (virtual) method and is not resolved until a specific instance of the ring is given at run time. The efficiency gain from eliminating abstraction overhead may become very visible in complex calculations such as Gaussian Elimination as shown in [3, 4]. Using this feature also enables us to define more completely designed hierarchy that were previously infeasible to achieve due to the abstraction overhead. For example, a proper and complete algebraic definition of a ring in the examples above should have started with a basic description of a set and follow the algebraic hierarchy to the object *magma* and then *monoid* and so forth until it is possible to describe a much richer structure of a ring through this hierarchy. Using interfaces and abstract methods this object hierarchy would be extremely slow to execute and not feasible computationally, whereas using the meta programming techniques described here the final generated code is very optimized and readable without any overheads. Eliminating the abstraction overheads allows an automatic instantiation of the code to become much more similar to the hand-written specialized version of the program.

3. Partial evaluation and compile time beta reductions: A less visible but extremely powerful effect of engineering multi stage programs is the control gained over compile-time partial evaluation of code fragments [8]. Although the F# optimizing compiler is capable of performing a lot of partial evaluation optimizations, we seek to gain more control over this during the stages of compilation according to the present static and dynamic data. If a code generator or combinator has specific data or options present at the generation time, it may be able to completely avoid producing a code fragment for future evaluation and instead present a precomputed result for the operation. In the `Power'` example in Section 3.4 when the generator is given a negative exponent on a ring with no division (or a 0 exponent with a ring that has no identity) the generator replaced the entire body of the calculation with the `None` value, completely reducing the need to perform any computation — or assertions on existence of `one` or `div`. This optimization in programs can potentially reduce the run-time computational complexity of the program as opposed to the linear speed increase gained by removal of abstraction calls.

Acknowledgments

I would like to thank my thesis adviser William Farmer for his advice and comments on my research and specially on this paper, and Jacques Carette for pointing me in the right direction on this research and reviewing my experiments with code generation. This research is being pursued under the MathScheme project at McMaster University.

References

[1] A. B. Bawden. Quasiquotation in lisp. In *O. Danvy, Ed., University of Aarhus, Dept. of Computer Science*, pages 88–99, 1999.

[2] D. Box and A. Hejlsberg. LINQ: .NET language integrated query, 2007. http://msdn.microsoft.com/en-us/netframework/aa904594.aspx.

[3] J. Carette. Gaussian elimination: a case study in efficient genericity with metaocaml. *Science of Computer Programming*, 62(1):3–24, 2006. ISSN 0167-6423.

[4] J. Carette and O. Kiselyov. Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. *Science of Computer Programming*, In Press, Corrected Proof:–, 2008. ISSN 0167-6423.

[5] J. Carette, L. E. van Dijk, and O. Kiselyov. Syntax extension for monads in ocaml, 2008. http://www.cas.mcmaster.ca/~carette/pa_monad/.

[6] K. Czarnecki, J. O'Donnel, J. Striegnitz, and W. Taha. Dsl implementation in metaocaml, template haskell, and c++. 2004. http://www.cs.rice.edu/~taha/publications/journal/dspg04b.pdf.

[7] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 237–247, New York, NY, USA, 1993. ACM. ISBN 0-89791-598-4.

[8] R. Marlet, S. Thibault, and C. Consel. Efficient implementations of software architectures via partial evaluation. *Automated Software Engg.*, 6(4):411–440, 1999. ISSN 0928-8910.

[9] Microsoft Research. The `F#` power pack, 2010. http://fsharppowerpack.codeplex.com/.

[10] MSDN. Microsoft `F#` developer center and language reference, 2010. http://msdn.microsoft.com/en-us/library/dd233154(VS.100).aspx.

[11] E. Pentangelo. `M<'a>` Lib (F#/C# Monads Library), 2009. http://sharpmalib.codeplex.com/.

[12] Rice University Programming Languages Team – PLT. Metaocaml: A compiled, type-safe, multi-stage programming language, 2006. http://www.metaocaml.org/.

[13] T. Sheard and S. P. Jones. Template meta-programming for haskell. *SIGPLAN Not.*, 37(12):60–75, 2002. ISSN 0362-1340.

[14] K. Swadi, W. Taha, O. Kiselyov, and E. Pasalic. A monadic approach for avoiding code duplication when staging memoized functions. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 160–169, New York, NY, USA, 2006. ACM. ISBN 1-59593-196-1.

[15] D. Syme. Leveraging .NET meta-programming components from `F#`: integrated queries and interoperable heterogeneous execution. In *ML '06: Proceedings of the 2006 workshop on ML*, pages 43–54, New York, NY, USA, 2006. ACM. ISBN 1-59593-483-9.

[16] D. Syme and J. Margetson. `F#` programming language, 2010. http://research.microsoft.com/fsharp/.

[17] D. Syme, A. Granicz, and A. Cisternino. *Expert* `F#`. Apress, 2007.

[18] D. Syme, G. Neverov, and J. Margetson. Extensible pattern matching via a lightweight language extension. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, pages 29–40, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-815-2.

[19] W. Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999.

[20] W. Taha and T. Sheard. Metaml and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000. ISSN 0304-3975.

[21] E. Westbrook, M. Ricken, J. Inoue, Y. Yao, T. Abdelatif, and W. Taha. Mint: Java multi-stage programming using weak separability. ACM, 2010.