Comp 511 Notes

Inscribed by Anthony Castanares, Seth Fogarty, and Kristin Y. Rozier

February 16, 2005

1 Staging annotations in 'C and MetaOCaml

Note that for these notes ! is the dereference operator, not the shorthand for .!, the run operator. *run* will be used to designate .!.

There are two kinds of code pieces in 'C, `cspec` and `vspec`.
This section concerns the methods of creating and handling `cspec` code. This code refers to expressions, statements, and values. The operators on `cspec`, and their corresponding MetaOCaml variants are:

MetaOCaml	'C
<>	′
$\sim$	@
run	$compile()$
$\sim (lift!_)$	$
$lift(!x)$	‘$x

1.1 Lifting in MetaOCaml

Simple values in MetaOCaml are automatically lifted. Function values are not, but integers and other base types are 'inlines' directly into the code.

```
let x = 1 in
 <x>
-> <1>

let x = fun x -> x in <x>
-> <%x>
```

This removes the need for a primitive lift operator. Note that this is a reasonable optimization in a functional programming language, where values are not mutable. The only reason to keep a name instead of a value is that the value associated with that name might change. This is not possible in MetaOCaml without the use of references.

1.2 The $ operator

In 'C the $ operator is a little strange. It is similar to a splice-and-lift in MetaOCaml. It's purpose is to inject the rvalue of a variable into a piece of code, much like MetaOCaml already does for base values. Everything in 'C is statically scoped, the use of the $ operator is the difference between returning a location (lvalue) and returning the value (rvalue).

```
int x = 1
return 'x
-> 'x

int x = 1
return `$x
-> `1
```

The correspondence in OCaml requires the use of references. This is not a feature of the translation of the metaprogramming constructs, but is a feature of the translation of C code to OCaml code.

```
int x = 1 return `$x
-> `1
```

Corresponds to

```
let x = ref 1 in
 lift (!x)
-> <1>
```

and

```
int x = 1
return 'x
-> `*[pointer/lvalue x]
```

corresponds to

```
let x = ref 1 in
 <!x>
-> <!%x>
```

Note that the second example allows you to return garbage. You have a lvalue for x, but once x falls out of scope the location can be claimed for other areas. This is a problem that is considered in 'C, but not permanently solved.

2 Other 'C Operators

The following special 'C operators are frequently used but have no direct equivalents in MetaOCaml:

- '_
- param
- local
- push_init
- push

In 'C, you can have a function with no type signature at compile time because it is created dynamically so you use `push` to provide parameters to it. That is, `push` makes a list of arguments. Then `param` creates the parameter for the function.

Here is an example of the use of `param`:

'C	MetaOCaml
`int vspec v = param (int, 0)` `c = '(0 + v)`	`<fun x -> 0 + x>`

The operator `param` by itself does not have a direct equivalent in MetaOCaml. It would be something like

```
C[c := '(0 + v)] in MetaOCaml
```

So another way to write the previous function in MetaOCaml would be:

```
<fun x -> ~( ... C[<0 + x>]...)>
```

Here is an example of combining the use of `param` with the `push` operator:

‘C	MetaOCaml

`int vspec v = param(int, 0)`
`c = compile (‘return(0 + v))`
`void cspec args = push_init();`
`push(args, ‘(17);`
`‘c(args);`
	`let c = .!<fun x \rightarrow 0 + x>`
	`<c 17>`

The `local` operator creates a dynamic variable name that is in the local scope. Here is an example of local:

‘C	MetaOCaml

`int vspec i = local(int);`
`c = compile (`
`  ‘{i = 17; return (i + 1)})`
	`<let i = ref 0 in`
	`  ~ ...`
	`  C[i := 17; !i]>`

Note the variable re-naming which happens automatically in MetaOCaml in this example. The ‘C code is harder to type-check and is more complex to avoid re-use of variable names. ‘C does do other things automatically, like the implicit @-signing feature, which Walid and Seth find very confusing.

2.1 Dependant Types

The feature of ‘C of having dynamically determined numbers and types of arguments to functions is not allowed in MetaOCaml. For instance, you can't accomplish the following in ML without any tagging and untagging:

$$\pi n : int.A^n \rightarrow B$$

As an aside, here is a function which can be written in MetaOCaml using tagging and untagging:

```
f 1(a) = a
f 2(a, (b)) = b
f 3(a, (b, (c))) = c
```

For this function, we will have the first argument in the first stage and then we use that information to specialize the function. This is the closest

approximation we have to the unique property of ‘C that its functions can take undetermined numbers of arguments. (In MetaOCaml there is also a way to accomplish this same functionality by cheating, using an array and switching the OCaml checks to off.) There may also be a viable alternative way of implementing this functionality in ‘C, which would be to adapt the mechanism that printf uses since printf also takes in a dynamically determined number and type of arguments.

The most important question here is: how do we do static checking before we generate on this type of function? The answer is that you need dependent types.

3 Continuation of ’C

In comparison to ’C, MetaOCaml can implement goto statements using CPS continuations. The ’C paper commented on Fabius. Fabius will attempt to apply curried functions which leads to code explosion: if big expressions are bound to let statements then the evaluation of those bindings can explode. As an example, look at the following code where two functions are evaluated in ML and in Fabius:

```
let f a b c =
   let x = a + b in
     x + c

let g = f 1 2 in g

ML => fun c -> let x = 1 + 2 in x + c

Fabius => fun c -> 3 + c
```

The ML evaluation will return a function expecting the third argument to f (c), with the work remaining to be done as the body of the function. Note, however, that the expression `1 + 2` is not reduced or simplified to `3`. Instead it is left unevaluated. Fabius will evaluate `1 + 2`, bind it to `x`, and replace `x` within the body of the expression. Fabius uses runtime code generation with currying to get this result.

The lack of static typing in 'C makes programming in 'C more difficult than in MetaOCaml. In MetaOCaml, type checking is done so that it does not interfere with programming; this is not the case with 'C.

The ML evaluation will return a function expecting the third argument to f (c), with the work remaining to be done as the body of the function. Note, however, that the expression `1 + 2` is not reduced or simplified to `3`. Instead it is left unevaluated. Fabius will evaluate `1 + 2`, bind it to `x`, and replace `x` within the body of the expression. Fabius uses runtime code generation with currying to get this result.

The lack of static typing in 'C makes programming in 'C more difficult than in MetaOCaml. In MetaOCaml, type checking is done so that it does not interfere with programming; this is not the case with 'C.

'C performs heavyweight compilation as early as possible and saves compilation tasks such as register allocation and memory functions to the end of compilation. This type of strategy yields very fast run-time performance but limits the amount of optimizations that can be performed on the code. For instance, if lots of compilation is done in a hurry, say in one pass, that leaves no more time to go back over the code and optimize for performance. The following example illustrates this issue. Say we have the following code fragments generated from one program:

```
<17>     <x>     <~f  ~x>     <fun x -> ~(f <x>)>
```

If we want to perform the same static compilation strategy that 'C uses, we won't know how these code fragments will be combined during execution. This single program can produce these difference code fragments under different inputs. We can not perform any optimizations statically by looking at these outputs. That is the issue with optimizing early: one is limited in their options for optimizing. Jumbo has options that can be turned on for using an early compilation strategy but many issues arise using it like linking problems, problems with cross-stage persistence of values, etc.