

Comp511 Notes

by Kristin Y. Rozier

January 19, 2005

1 Reminders

- There is a compile-time parameter for getting rid of the comments that show up in the general code and producing more concise output. Check `metaocaml -help`.
- Expect partial evaluators to have a hard time evaluating things because an understanding of the algorithm is required.

2 Comments on Chapter 1 reviews

- This chapter makes a clear distinction between multi-stage programming and other forms of meta-programming.

metaprogramming: program that manipulates as data another program

Code is first-class in multistage languages but you can't take code apart in multi-stage languages.

Multi-stage programs are pretty impoverished in terms of meta-programming support; they are models for program generation but not program manipulation and transformation.

- *Inference* is confusing. When you are starting to stage a program, use explicit type declarations in your functions to make things easier to understand.
- *Partial evaluation* is an automated form of binding-time analysis. For example: write the power function, think about where the staging constructs will go, then stage it. Partial evaluation does all of the stuff that you're supposed to do by hand in multi-stage programming.

In practice there are a lot of problems because it is not so easy to figure out where the staging annotations go. This is partly because there are frequently multiple ways to stage a program, even with the same order of inputs. There is no best way to stage a program; it all depends on what you want to do with it. By the time you have spent the time figuring out how the partial evaluation works, you are better off doing it by hand. There are currently more people using MetaOCaml by hand than using partial evaluators. There are usability issues here.

- There was a confusion about when to use the staging constructs.

In OCaml, there is no real need for lift because it is just defined as $lift(x) = \langle x \rangle$. For example, $lift(5) = \langle 5 \rangle$.

$\langle 1 + 5 \rangle \rightarrow \langle 1 + 5 \rangle$ is different from $lift(1 + 5) \rightarrow \langle 6 \rangle$. The main usefulness of lift is to make sure the function is evaluated earlier. The key thing is that you have static typing and that you know how you deal with it. How you use the staging constructs will become more clear with experience.

- Multi-level becomes multi-stage by adding the run construct (the staging part). That is the main jump.

3 Encoding

Guarantee	
None	string: $f(x) = x + 1$
syntactic correctness "CFG"	datatype: $fun("f", "x", +(var"x", int1))$
type safety "CSG"	MSP: $\langle f(x) = x + 1 \rangle$

What is encoding? If you have two sets A and B and an encode function E and a decode function D such that $D \circ E = id_A$ and $E \circ D = id_B$ (in other words, the relationship is one-to-one and onto), then B represents A . But this definition is too narrow. For instance, we can represent the natural numbers by reals. We don't need the function $E \circ D = id_B$. Even though this relation is not required for representation, it is true for the MSP example. There is no extra junk in this set.

3.1 Exercise: stage the power function in Perl

The power function has type $int \times float \rightarrow float$.

The staged power function has type $int \times \langle float \rangle \rightarrow \langle float \rangle$.

Annotated types with extra expressions don't fit naturally into type systems. Maybe it would be more intuitive if we subscripted the float variables. *The type constructor encodes the level that you are at.* There is a little bit of indirection there.

Here is an implementation of the power function in perl, followed by a nifty function which produces the staged version of the power function:

```
#!/usr/bin/perl
#by Kristin Y. Rozier

sub power {

    my ($exp, $base) = @_;

    if ($exp == 0) {    #base case
        return 1;
    } #end if
    elsif ($exp%2==0){ #if the exponent is even
        return (&power($exp/2, $base) * &power($exp/2, $base));
    } #end elsif
    else {              #the exponent is odd
        return $base*&power($exp-1, $base);
    } #end else

} #end power

$exp = 4;
$base = 2;

print "$base to the power of $exp = ";
$answer = &power($exp, $base);
print "$answer\n";

sub stage_power {

    my ($exp, $string) = @_;
```

```

    if ($exp == 0) {    #base case
        $string = $string."1";
        return $string;
    } #end if
    elsif ($exp%2==0){ #if the exponent is even
        $string = $string." "(&stage_power($exp/2, $string)
            ."*").&stage_power($exp/2, $string));
        return $string;
    } #end elsif
    else {              #the exponent is odd
        $string = $string."\"$base *".&stage_power($exp-1, $string);
        return $string;
    } #end else

} #end power

```

```

print "Function for the power of $exp = \";
$staged_power_string = &stage_power($exp, "");
print $staged_power_string."\"\\n";

```

```

print "Staged power function answer: ";
$answer = eval $staged_power_string;
print "$answer\\n";

```

The program above produces the following output:

```

% perl power.pl
2 to the power of 4 = 16
Function for the power of 4 = " $base *1*$base *1* $base *1*$base *1"
Staged power function answer: 16

```

4 Future Discussions

If we have time next class, we will stage Ackermann's function and List.map.

5 Perl is very very fast!

The perl code featured above was later refined to include timing information and benchmarked. The resulting code is located in this directory and named power.pl. The results of multi-staging are evident in its output below:

```
olympus(kyrozier)% perl power.pl
Unstaged power function:
timethis for 2: 2 wallclock secs ( 2.12 usr + 0.00 sys = 2.12 CPU)
 @ 339.15/s (n=719)
Result: 3 to the power of 4 = 81
```

```
Staging the power function:
timethis for 2: 2 wallclock secs ( 2.16 usr + 0.00 sys = 2.16 CPU)
 @ 194.44/s (n=420)
Function for the power of 4 = " &square ( &square ($base *1))"
```

```
Running the staged power function:
bigString is:
for ($x=0; $x<=200; $x++)
{
    &square ( &square ($base *1));
}
```

```
timethis for 2: 2 wallclock secs ( 2.22 usr + 0.00 sys = 2.22 CPU)
 @ 1099.10/s (n=2440)
Staged power function answer: 81
```

The staged version of the power function runs 1100 times a second!!!