

1 Program Generation

Popular high-level programming languages provide programmers with many abstraction techniques. For the sake of performance, many programmers avoid using these abstractions because their use incurs a run-time performance penalty. Techniques in program generation afford the programmer many of the same abstraction techniques but without such a run-time performance overhead. Multi-stage languages perform program generation and provide static typing of generated code.

Literature often suggests that program generation can be used to optimize a program with respect to performance. But a program written by hand and optimized by the compiler can always match the performance of any generated program. Instead, multi-stage programming allows programmers to write programs using powerful abstractions, but without having to pay a run-time overhead for these abstractions.

In the following example we see the use of abstraction: the square function will return the square of an integer, but it is defined using the more general power function. Such an abstraction makes code more structured and easier to maintain but degrades performance.

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

let squareC = λx. power (x 2)
```

Staging can be modeled in a general-purpose language by using strings and concatenation. For example, we can write a function that returns code (in the form of a string) that represents some computation instead of performing that computation directly. The following example is written in OCaml:

```
let rec pow x n = if n = 0 then '1' else x ^ '*' ^ (pow x (n - 1))
```

```
let squareC = ``λx.`` ^ power (``x`` 2)
```

If squareC is run at this point, the following string will be produced:

```
``λ x. x * x * x * 1``
```

This string is constructed from repeated recursive calls to power. That is, first the initial “x” is generated, then another “x” is concatenated to it, and so forth until the final “1” is added and returned.

EXERCISE: Run the previous example in OCaml - NOT MetaOCaml.

What we have just seen is the state of the art in multi-stage programming. Strings and datatypes can be used to differentiate between different stages of computation. This example shows the idea behind MSP: fragment the code to show what can be done now and what can be done later. In the case above, brackets correspond to quotations marks.

Yet, with strings one could produce nonsensical code. For instance, what if we replaced every “*” (multiplication symbol) with a “?” (question mark)? The code would still typecheck because the code excluding the strings is still valid. Yet the final string that is produced would look something like

```
``λ x x ? x ? x ? 1``
```

which is obviously wrong. What is needed here is some sort of mechanism to look inside the strings and make sure the code that is being produced is indeed valid with respect to the type system. Strings can not be used because when strings are introduced they prohibit the typechecker from looking inside of them to inspect their correctness.

2 MSP constructs

There are three basic constructs in MSP. These constructs include brackets, escapes, and the run construct. Type constructors will return types of the

form `<_>`. For instance, `<1+1>` will have type `<int>`. In the MetaOCaml implementation expressions will have type `int` code, but more is needed than just code: programmers want to be able to use the generated code. This is where the `run` construct comes in. The `run` construct is the exclamation point (`!`). It allows the programmer to execute code that has been generated using brackets and escapes. More specifically, `run` does a lot more: `run` will perform lexical analysis, parse, type check, compile, load (into memory) and run the code presented to it.

2.1 MSP vs. Multi-level languages

Brackets and escapes alone form the basis of multi-level languages, whereas brackets, escapes, and the `run` construct will form the basis of multi-stage languages. The `run` construct allows programmers to stage their programs.

3 Classifiers

One of the goals of this class is to develop an intuitive understanding of environment classifiers. There currently is no intuitive and straightforward definition for an environment classifier. They show up when staged code is submitted to MetaOCaml and the system returns a type like `(α ,int)` code. The classifier is the α . This extra parameter is usually written in superscript format when in class and in casual presentation, but in MetaOCaml it is in the form of a tuple along with the expression type (`int`, `char`, etc). As Dr. Pasalic stated, we can ignore this, for now.

4 GCD Example

The following is an example of the greatest common divisor algorithm (GCD). As an exercise to the reader, it would be helpful to find out what has been done with GCD in the partial evaluation community (can GCD be staged?). On our first pass over this code, we as a class were unable to

stage it because of the nature of the algorithm. Below is the algorithm we implemented:

```
let gcd a b = if (a mod b = 0 ) then b else gcd b (a mod b)
```

As can be seen, if either parameter (a or b) is to be the early data and the other to be the late data, one will run into complications because the conditional statement in the if statement depends on knowing both values. But, as Dr. Taha mentioned in class, useful work can still be done on the branches. Yet another roadblock arises because the else branch uses both values. Finally, if the algorithm is changed by flipping the parameters in the gcd function, it will not terminate. A question to ponder: what if the modulus function (mod) is staged as well? The definition of mod uses the division (div) function - what if div is staged as well? This line of questioning must stop somewhere, but it is possible that the same problem will show up here as it did in the gcd function.

As further reading, the following links provide information on staging the greatest common divisor algorithm:

<http://repository.readscheme.org/ftp/papers/brics/BRICS-RS-97-1.pdf>
<http://repository.readscheme.org/ftp/papers/brics/danvy-ln.pdf>