

1 Discussion of the Gentle Introduction to MSP

After reading “A Gentle Introduction to Mutli-stage Programming”, a few topics were addressed. These included the validity of the lack of using strings to produce syntactically correct programs, a more thorough explanation of the interpreter presented in the text, and a methodology for creating staged programs.

1.1 Syntax Correct Strings??

The paper claimed when building program generators, strings can construct and combine code fragments concisely, however they cannot *automatically verify* that generated programs are syntactically correct. The example that paper presented was “f (, y)”. The argument was made that you can check for syntax correctness with strings. The response to this was that (*at compile time*) a programming language has to make a decision to either treat a string normally or treat it as *generating code*. One can think of the $\langle \dots \rangle$ as a “*special*” type of string. If there was not a clear separation such as this, it would hard for a programming language to decipher whether or not to treat strings normally or to treat them as generating fragments.

1.2 The Interpreter

There were a few issues that came up concerning the interpreter presented in the paper. There was confusion in the functionality of the ext function. Its purpose was to model environments as functions. Let’s say we have an environment as followed:

Variable	Value
x	13
y	4

We wish to model this environment as a function. A simple version of this function would be:

```
fun s → if s == "x" 13 else if s == "y" then 4 else raise Exception
```

Now if we wish to add "z" → 7 to the environment we would use the ext function:

```
let ext env s v = fun s' → s == s' then v else env s'
```

Another issue that was brought up was the need for two different environments. To ensure type correctness there needed to be an environment that looks up variables and one to look up functions. One impressive feature of the interpreter was its length. Even though it had limited capabilities, it was compact. The C version of this would be longer, however, the length Perl version is still being debated.

1.3 Abstraction without the penalty

Throughout the course, the motivation behind Multi-Staged Programming was to be able to make more reusable and robust code without paying the penalty for the abstraction. One item that this paper introduced was having a baseline for writing staged programs. The idea is to write the *non-staged* version of a program. Then one should think of the *ideal* generated code that is produced for specific cases. Finally, the staged-version of the program should be designed to generate the *desired* code.

2 Homework

Write the CPS and Monad version of the Ackermann function.