David Olson

Eric Seamons

Part 1

It would be possible to create this sort of a type system.  There are a finite number of data structures and a finite number of primitive types.  Many combination types would need to be created, such as lists of numbers, lists of strings, lists of lists, and so on.  While possible, this isn't a wise level of abstraction.  This leaves us with a large number of types to declare and worry about when programming and when type checking and compiling.  The much easier way is to consider any list to be of the same type, which is a list of anything.  Then, when the program deals with the actual contents of the list, the type checker worries about what types are actually in the list.  The final type of a program generally isn't a list anyways.

The idea that our boss has isn't amazing, but has a familiar feel to it.  C++ and Java are strict about declaring the primitive type that will be stored in data structures.  The compiler wants to make sure that the correct type of items are being put into the data structure, for your benefit.  That isn't to say that Java would consider ArrayList< ArrayList <Integer>> a type, but it is a combination of types that the compiler recognizes and does type checking on.

Part 2

For both function definitions and applications, and other parts of the language syntax, type checking will not change if we implement the language lazily, but the type checker itself is not lazy.  Type checking and lazy interpretation have the similar strictness points.  Primitives such as numbers don't need to be calculated further, and their types are already known.  Function definitions are just descriptions of the function, not a calculation.  The type checker will figure out the type of the body while a lazy interpreter will calculate the body only when needed, but that does not change the return type at all.  Function applications are a strict point of both when calculations need to be done and when in the type checker we are just left with the return type.

Part 3

Type checking is very important in programming, because some data types cannot have certain operations performed on them.  For example, the plus operator can perform on numbers, but it cannot perform on booleans. In many languages the plus operator can also perform on strings, but the two variables that are being operated on need to be of the same type. Type Checking is especially important in programming environments, because type errors are some of the hardest errors to find. Without pre-runtime type checking, the programmer could spend hours caught up on a simple type error, especially in large applications.

This assignment has made us more self-aware of type errors. When writing code, we need to “type check” and make sure that whatever variable we are using an operator on is always of the correct type. For example, if we are using a dynamically typed language and the program accepts user input, we need to make sure that we check the user input so that we are not performing an operation on an incorrect type. Keeping track of the type of variables is important in programming.