To explore brave new worlds...

Before we begin...

Welcome back to week whatever it is of the programming languages course! If you're still reading this, and are confused, here's what you forgot:

- Whatever text is written in a regular, black-on-white font is essential and couldn't be skipped. This is usually something you should take your time and read through carefully. If it's **bold** or *italic*, doubly so!
- If text appears in a darker grey box with a **"Sidenote"** caption, it's usually an interesting sidenote on the topic in question. These are usually there to tell you things you would come to realize for yourselves at some later point.
- If text appears in a dashed, lighter grey box, with a caption like **"Don't drink too much Kool-aid"**, this is usually an example, counter-example, question or something to spend some time thinking about. We're interested in your opinions, if you'd give them.
- Code segments and equations are given in a `monospaced font`, with explanations usually following or preceding them.
- Anything else is a comment.

1 Overview

Even though you may not feel like it, I assure you that you have written an interpreter for a language very similar to Scala, you've enriched it with a type-system, and now that you can write sensible code in there, you can work on expanding it. We've lost some freedom and got some security with the addition of the type system, and now, let's see if we can get some of that freedom back, by going polymorphic!

2 Parametric Polymorphism

For to be free is not merely to cast off one's chains, but to live in a way that respects and enhances the freedom of others.

Nelson Mandela

Parametric polymorphism, which is a mouthful, is also a mechanism for making a language more expressive while maintaining full static type safety.

Polymorphism (greek, πολυς meaning "many"and μορφη meaning "forms") is a wide term, generally being really abstract: it means that some interface can handle multiple types. How this is implemented makes it parametric, ad-hoc or subtype polymorphism.

Sidenote: Ad-hoc polymorphism is what you might know as "function overloading", where you define a function of the same name, but different types, and the resolution of which one is called depends on the call arguments.
Subtype polymorphism is when we denote different instances by name. This is just the good old inheritance-based stuff you know from object-oriented classes. This is sometimes referred to as just, simply, polymorphism in object-oriented circles.

Parametric polymorphism is when code is written but instead of mentioning specific types, we leave holes (also known as *type variables*) to be filled at some point later. These holes are generic in nature, so the types associated with them are sometimes also called generic types.

For this to work, we have to, yet again, change our syntax and typing rules to fit. This change, however, is quite small compared to adding a type-checker, and mostly constitutes of minor additions. This ended being the shortest and most to-the-point handout yet, as I didn't want to bog you down with extra stuff. Without further ado...

2.1 Syntax Additions

For starters, let's introduce type variables into our syntax. There are several moments where we're going to use them. Let's call this T, and the set of all type variables *TypeVariable*.

$$T \in \mathit{TypeVariable}$$

As for the syntactic rules, we can have either type variables themselves or user-defined types with type variables. Basically, the only difference you'll be seeing is that we're adding the $[\vec{\tau}]$ to everything.

$$\tau \in \mathit{Type} ::= \textbf{string} \mid \textbf{boolean} \mid \textbf{integer} \mid \textbf{unitType} \mid \tau_1 \Rightarrow \tau_2 \mid (\ \vec{\tau}\ ) \mid un[\vec{\tau}] \mid T$$

In expressions, we can either have named functions or constructors which take type arguments.

$$e \in \mathit{Exp} ::= fn[\vec{\tau}](e) \mid cn[\vec{\tau}](e) \mid \ \ldots$$

Both of these are supplemented by ways to actually define them, which we just extend a bit:

$$\mathit{def} \in \mathit{Def} ::= \textbf{def}\ fn[\vec{T}](x : \tau_1) : \tau_2 = e$$
$$\mathit{tdef} \in \mathit{UserDefinedTypeDef} ::= \textbf{algebraic}\ un[\vec{T}] = \overrightarrow{\mathit{cdef}}$$

Now, it may be weird that in some places, we have a $[\vec{\tau}]$ and somewhere $[\vec{T}]$. To understand this, let me give you and example from Scala:

```
def foo[T](t: T) = t
...
foo[Int](5)
```

The definition of a function requires a type variable, so we use a free type variable T to mark whatever the actual type will be. Once the time comes to call the function, instead of using a T, we use an actual type, *Int* in this case, which is in the set of types τ. This is exactly how `SimpleScala` will be from now on.

Don't drink too much Kool-Aid: *Okay, not really exactly: Scala can* ***infer*** *the types, so we can basically just write:*

```
def foo[T](t: T) = t
...
foo(5)
```

The type checker is powerful enough to recognize which type-variables come down to what based on the value (e.g. 5 is an integer). How would you do that in `SimpleScala` *?*

2.2 Type System Additions

Accordingly, we have to extend our type domains with type variables, too. Where we had function definitions map from function names to tuples of types (argument and return type), now we also include the type variable list present (might be empty!) in the definition. This is to say that a function that doesn't need type variables can choose to ignore them, but the reasoning is still polymorphic!

$$\mathit{fdefs} \in \mathit{NamedFunctionDefs} = \mathit{FunctionName} \rightarrow (\overrightarrow{\mathit{TypeVariable}} \times \mathit{Type} \times \mathit{Type})$$

Similarly, we change type definitions to include type variables.

$$tdefs \in TypeDefs = UserDefinedTypeName \to (\overrightarrow{TypeVariable} \times (ConstructorName \to Type))$$

Last but not least, we need to define a scope for type variables, which is a set of type variables currently available to a location in our program.

$$tscope \in TypeVarsInScope = \overline{TypeVariable}$$

2.3 Typing Rules

Now that that's said and done, let's take care of the actual changes in the type system. The largest one is this one:

$$\frac{\begin{array}{c}\texttt{typeOkList}(\vec{\tau_1}) \quad fn \in \texttt{keys}(fdefs) \quad (\overrightarrow{T} \cdot \tau_2 \cdot \tau_3) = fdefs(fn) \quad |\overrightarrow{T}| = |\vec{\tau_1}| \\ \tau_2' = \texttt{typeReplace}(\overrightarrow{T}, \vec{\tau_1}, \tau_2) \quad \tau_3' = \texttt{typeReplace}(\overrightarrow{T}, \vec{\tau_1}, \tau_3) \qquad \Gamma \vdash e : \tau_2'\end{array}}{\Gamma \vdash fn[\vec{\tau_1}](e) : \tau_3'} \text{ (NAMECALL)}$$

If we have a function call, with type arguments $\vec{\tau_1}$ and a return type τ_3', check whether the types in $\vec{\tau_1}$ are okay (more on that in the helpers section), and then does the same things as before except that all our definitions now have a richer type, with the $\overrightarrow{T}$ passing around. These are the variables, which have to be the same length as our concrete types that we pass in (e.g. if our function is parametrized by one type variable, we have to supply exactly one concrete type).

The biggest new piece here is the `typeReplace` helper, which actually does the replacement of type variables with concrete types. It basically fills the holes which type variables are, by reading from the currently available concrete types (passed in by τ_1), building a map between the type variables in T and those, and then bridging that gap. Here's an example of how `typeReplace` would work in practice:

$$\begin{aligned}&\texttt{typeReplace}([A], [\textbf{int}], \textbf{string}) = \textbf{string}\\&\texttt{typeReplace}([A], [\textbf{int}], A) = \textbf{int}\\&\texttt{typeReplace}([A], [\textbf{int}], A \Rightarrow (\ A, \textbf{string}\ )) = \textbf{int} \Rightarrow (\ \textbf{int}, \textbf{string}\ )\\&\texttt{typeReplace}([A], [B], A \Rightarrow A) = B \Rightarrow B\end{aligned}$$

We do this twice: once to get the type of the argument, and another time to get the return type. If these, in fact, weren't type variables, as the first example shows, `typeReplace` returns the concrete argument.

I sense a question incoming: what do we need τ_2' for? We never use it in the conclusion below the line! Yes, but the typing of our argument is implicit, as seen in the last (rightmost) part of the premise; e is of type τ_2', which we got from passing the type definition from $(\overrightarrow{T} \cdot \tau_2 \cdot \tau_3) = fdefs(fn)$ through the type replacement function; it must match up with the expected input type after type replacement.

Both constructors and matching is literally the same thing, and is left to the readers as an exercise. Here's a question that might help you with solving constructor-based rules: what is m? What does it represent? Once you know this, send me a mail with the explanation and the title "m is for..." to win a small, but very unique (as in "there's literally one of these on this planet") prize! Now that that's done, here's a quick overview of the helpers used!

2.4 Helpers

These helpers are in three categories: those not mentioned do the exact same thing as before, and we didn't want to cut wood just so that you could have a duplicate page. The second category is of the ones that got affected by the changes but didn't change their original purpose. The third is of those that are here to fill in the functionality for parametric polymorphism, and are explained in detail.

2.4.1 typeOk

Returns **true** if the given type is valid. This means the following:

$\texttt{typeOk} \in \mathit{Type} \rightarrow \mathit{Boolean}$

$\texttt{typeOk}(\textbf{string}) = \textbf{true}$

$\texttt{typeOk}(\textbf{boolean}) = \textbf{true}$

$\texttt{typeOk}(\textbf{integer}) = \textbf{true}$

$\texttt{typeOk}(\textbf{unitType}) = \textbf{true}$

$\texttt{typeOk}(\tau_1 \Rightarrow \tau_2) =$

    $\texttt{typeOk}(\tau_1) \wedge \texttt{typeOk}(\tau_2)$

$\texttt{typeOk}((\ \vec{\tau}\ )) =$

    $\texttt{typeOkList}(\vec{\tau})$

$\texttt{typeOk}(un[\vec{\tau}]) =$

    $\texttt{typeOkList}(\vec{\tau})$

$\texttt{typeOk}(T) = T \in \mathit{tscope}$

As you can see, everything is quite simple: basic types are always available, functions have to be okay for both the input and output types, tuples and user defined types are also recursively run (using a smaller helper, `typeOkList`, which just traverses through the types and repeatedly calls `typeOk` on the head) and type variables are okay if they are in the current scope.

2.4.2 typeReplace

Performs type replacement of type variables with concrete types. The first parameter is the sequence of type variables in play. The second parameter is the sequence of concrete types these should map to. The third is the target expression which we will be doing the replacements in.

If this target is a concrete type, the result is that same type. If not, there are several options:

- the target is a type variable: in this case, replace it if it has a mapping in the map made from the first and second argument
- the target is a function type $\tau_1 \Rightarrow \tau_2$: recurse into both τ_1 and τ_2, return with their more concrete forms τ'_1 and τ'_2, and return a function type $\tau'_1 \Rightarrow \tau'_2$
- the target is a tuple type $\vec{\tau_1}$, run through it recursively, returning with the more concrete form for every element of the tuple, which will constitute the resulting type
- the target is a user-defined type $un[\vec{\tau_1}]$, behave the same as you did with the tuple case, but wrap the result in the *un* type before returning

Formally, it uses three functions to help itself. The `makeMap` function, which hasn't been shown before (because it's used only internally) creates a map out of two lists. `typeReplace` then does the following:

$\texttt{typeReplace} \in \overrightarrow{\mathit{TypeVariable}} \times \overrightarrow{\mathit{Type}} \times \mathit{Type} \rightarrow \mathit{Type}$

$\texttt{typeReplace}(\vec{T}, \vec{\tau_1}, \tau_2) =$

    $\texttt{typeReplaceHelper}(\texttt{makeMap}(\vec{T}, \vec{\tau_1}), \tau_2)$

The `typeReplaceHelper` function is used to actually traverse the types:

```
typeReplaceHelper ∈ $(\mathit{TypeVariable} \rightarrow \mathit{Type}) \times \mathit{Type} \rightarrow \mathit{Type}$
typeReplaceHelper($m$, $\textbf{string}$) = $\textbf{string}$
typeReplaceHelper($m$, $\textbf{boolean}$) = $\textbf{boolean}$
typeReplaceHelper($m$, $\textbf{integer}$) = $\textbf{integer}$
typeReplaceHelper($m$, $\textbf{unitType}$) = $\textbf{unitType}$
typeReplaceHelper($m, \tau_1 \Rightarrow \tau_2$) =
    let $\tau_1'$ = typeReplaceHelper($m, \tau_1$)
    let $\tau_2'$ = typeReplaceHelper($m, \tau_2$)
    $\tau_1' \Rightarrow \tau_2'$
typeReplaceHelper($m$, ( $\vec{\tau_1}$ )) =
    let $\vec{\tau_2}$ = typeReplaceHelperList($m, \vec{\tau_1}$)
    ( $\vec{\tau_2}$ )
typeReplaceHelper($m, un[\vec{\tau_1}]$) =
    let $\vec{\tau_2}$ = typeReplaceHelperList($m, \vec{\tau_1}$)
    $un[\vec{\tau_2}]$
typeReplaceHelper($m, T$) = $m(T)$
```

When we deal with lists, `typeReplaceHelperList` is used to map over them. This effectively just performs Scala's `map` method, but in a way that is specific to type replacement.

```
typeReplaceHelperList ∈ $(\mathit{TypeVariable} \rightarrow \mathit{Type}) \times \overrightarrow{\mathit{Type}} \rightarrow \overrightarrow{\mathit{Type}}$
typeReplaceHelperList($m$, []) = []
typeReplaceHelperList($m, \tau_1 :: \vec{\tau_2}$) =
    let $\tau_1'$ = typeReplaceHelper($m, \tau_1$)
    let $\vec{\tau_2}'$ = typeReplaceHelperList($m, \vec{\tau_2}$)
    $\tau_1' :: \vec{\tau_2}'$
```

2.4.3 Small changes due to adding polymorphism

The functions which experienced some changes due to polymorphic typing alone, are `casesSane` and `casesTypes`.

In the case of `casesSane`, it's because it works with the *tdefs* structure, which now also holds the type variable list in it. This, however, doesn't affect much else.

As for `casesTypes`, which gets the type of each case in the list of cases, we now have to account for all the type replacements and type variables in the cases, and as such, this function changed somewhat to take that into account. This change, however, is barely visible to you, as you're just passing one more argument, which represents this list of type variables.

3 Done!

– Conan, what is best in life?

— Mongol General, to Conan, during dinner

Okay! Long one! It's 3:00 in the morning, and this chill lo fi playlist I'm on just gave me the chills by quoting a very profound thought Alan Watts wrote. Do ask all the questions you have, don't hold back.

Remember that you will surely pass this course, that's not the question. The question is how much you will learn and keep with you, now, for later, for circumstances unknown and problems yet unseen. Careful exploration of the unknown, guided and safe, is, after all, the best in life! If you disagree, write to us and explain what IS best in life.

– That is good! That is good!

— Mongol General, to Conan, after hearing his response

4 Poly-Typed SimpleScala

4.1 Syntax

$$x \in \mathit{Variable} \qquad \mathit{str} \in \mathit{String} \qquad b \in \mathit{Boolean} \qquad i \in \mathbb{Z} \qquad n \in \mathbb{N}$$

$$\mathit{fn} \in \mathit{FunctionName} \qquad \mathit{cn} \in \mathit{ConstructorName} \qquad \mathit{un} \in \mathit{UserDefinedTypeName}$$

$$T \in \mathit{TypeVariable}$$

$$\begin{aligned}
\tau \in \mathit{Type} &::= \textbf{string} \mid \textbf{boolean} \mid \textbf{integer} \mid \textbf{unitType} \mid \tau_1 \Rightarrow \tau_2 \mid (\ \vec{\tau}\ ) \mid \mathit{un}[\vec{\tau}] \mid T \\
e \in \mathit{Exp} &::= x \mid \mathit{str} \mid b \mid i \mid \textbf{unit} \mid e_1 \oplus e_2 \\
&\mid (x : \tau) \Rightarrow e \mid e_1(e_2) \mid \mathit{fn}[\vec{\tau}](e) \\
&\mid \textbf{if}\ (e_1)\ e_2\ \textbf{else}\ e_3 \\
&\mid \{\overrightarrow{\mathit{val}}\ e\} \\
&\mid (\ \vec{e}\ ) \mid e.\_n \\
&\mid \mathit{cn}[\vec{\tau}](e) \mid e\ \textbf{match}\ \{\overrightarrow{\mathit{case}}\} \\
\mathit{val} \in \mathit{Val} &::= \textbf{val}\ x = e \\
\mathit{case} \in \mathit{Case} &::= \textbf{case}\ \mathit{cn}(x) \Rightarrow e \mid \textbf{case}\ (\ \vec{x}\ ) \Rightarrow e \\
\oplus \in \mathit{Binop} &::= + \mid - \mid \times \mid \div \mid \land \mid \lor \mid < \mid \leq \\
\mathit{tdef} \in \mathit{UserDefinedTypeDef} &::= \textbf{algebraic}\ \mathit{un}[\overrightarrow{T}] = \overrightarrow{\mathit{cdef}} \\
\mathit{cdef} \in \mathit{ConstructorDefinition} &::= \mathit{cn}(\tau) \\
\mathit{def} \in \mathit{Def} &::= \textbf{def}\ \mathit{fn}[\overrightarrow{T}](x : \tau_1) : \tau_2 = e \\
\mathit{prog} \in \mathit{Program} &::= \overrightarrow{\mathit{tdef}}\ \overrightarrow{\mathit{def}}\ e
\end{aligned}$$

4.2 Type System

4.2.1 Type Domains

$$\begin{aligned}
fdefs \in \mathit{NamedFunctionDefs} &= \mathit{FunctionName} \rightarrow (\overrightarrow{\mathit{TypeVariable}} \times \mathit{Type} \times \mathit{Type}) \\
tdefs \in \mathit{TypeDefs} &= \mathit{UserDefinedTypeName} \rightarrow (\overrightarrow{\mathit{TypeVariable}} \times (\mathit{ConstructorName} \rightarrow \mathit{Type})) \\
cdefs \in \mathit{ConstructorDefs} &= \mathit{ConstructorName} \rightarrow \mathit{UserDefinedTypeName} \\
tscope \in \mathit{TypeVarsInScope} &= \overline{\mathit{TypeVariable}} \\
\Gamma \in \mathit{TypeEnv} &= \mathit{Variable} \rightarrow \mathit{Type}
\end{aligned}$$

4.2.2 Type Rules

$$\frac{x \in \mathtt{keys}(\Gamma) \quad \tau = \Gamma(x)}{\Gamma \vdash x : \tau}\ (\textsc{var}) \qquad \frac{}{\Gamma \vdash str : \textbf{string}}\ (\textsc{string}) \qquad \frac{}{\Gamma \vdash b : \textbf{boolean}}\ (\textsc{boolean})$$

$$\frac{}{\Gamma \vdash i : \textbf{integer}}\ (\mathbb{Z}) \qquad \frac{}{\Gamma \vdash \textbf{unit} : \textbf{unitType}}\ (\textsc{unit}) \qquad \frac{\Gamma \vdash e_1 : \textbf{integer} \quad \Gamma \vdash e_2 : \textbf{integer}}{\Gamma \vdash e_1 + e_2 : \textbf{integer}}\ (+_{int})$$

$$\frac{\Gamma \vdash e_1 : \textbf{string} \quad \Gamma \vdash e_2 : \textbf{string}}{\Gamma \vdash e_1 + e_2 : \textbf{string}}\ (+_{string}) \qquad \frac{\otimes \in \{-, \times, \div\} \quad \Gamma \vdash e_1 : \textbf{integer} \quad \Gamma \vdash e_2 : \textbf{integer}}{\Gamma \vdash e_1 \otimes e_2 : \textbf{integer}}\ (\textsc{arithOp})$$

$$\frac{\otimes \in \{\wedge, \vee\} \quad \Gamma \vdash e_1 : \textbf{boolean} \quad \Gamma \vdash e_2 : \textbf{boolean}}{\Gamma \vdash e_1 \otimes e_2 : \textbf{boolean}}\ (\textsc{boolOp}) \qquad \frac{\otimes \in \{<, \leq\} \quad \Gamma \vdash e_1 : \textbf{integer} \quad \Gamma \vdash e_2 : \textbf{integer}}{\Gamma \vdash e_1 \otimes e_2 : \textbf{boolean}}\ (\textsc{relOp})$$

$$\frac{\mathtt{typeOk}(\tau_1) \quad \Gamma[x \mapsto \tau_1] \vdash e : \tau_2}{\Gamma \vdash (x : \tau_1) \Rightarrow e : \tau_1 \Rightarrow \tau_2}\ (\textsc{anonFun}) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \Rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1(e_2) : \tau_2}\ (\textsc{anonCall})$$

$$\frac{\begin{array}{c}\mathtt{typeOkList}(\vec{\tau_1}) \quad fn \in \mathtt{keys}(fdefs) \quad (\overrightarrow{T} \cdot \tau_2 \cdot \tau_3) = fdefs(fn) \quad |\overrightarrow{T}| = |\vec{\tau_1}| \\ \tau_2' = \mathtt{typeReplace}(\overrightarrow{T}, \vec{\tau_1}, \tau_2) \quad \tau_3' = \mathtt{typeReplace}(\overrightarrow{T}, \vec{\tau_1}, \tau_3) \quad \Gamma \vdash e : \tau_2'\end{array}}{\Gamma \vdash fn[\vec{\tau_1}](e) : \tau_3'}\ (\textsc{nameCall})$$

$$\frac{\Gamma \vdash e_1 : \textbf{boolean} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \textbf{if}\ (e_1)\ e_2\ \textbf{else}\ e_3 : \tau}\ (\textsc{if}) \qquad \frac{\Gamma' = \mathtt{blockGamma}(\overrightarrow{val}, \Gamma) \quad \Gamma' \vdash e : \tau}{\Gamma \vdash \{\overrightarrow{val}\ e\} : \tau}\ (\textsc{block})$$

$$\frac{\vec{\tau} = \mathtt{tupleTypes}(\vec{e}, \Gamma)}{\Gamma \vdash (\ \vec{e}\ ) : (\ \vec{\tau}\ )}\ (\textsc{tup}) \qquad \frac{\Gamma \vdash e : (\ \vec{\tau}\ ) \quad \tau' = \mathtt{tupleAccess}(\vec{\tau}, n)}{\Gamma \vdash e.\_n : \tau'}\ (\textsc{acc})$$

$$\frac{\begin{array}{c}\mathtt{typeOkList}(\vec{\tau_1}) \quad cn \in \mathtt{keys}(cdefs) \quad un = cdefs(cn) \quad (\overrightarrow{T} \cdot m) = tdefs(un) \\ |\overrightarrow{T}| = |\vec{\tau_1}| \quad \tau_2 = m(cn) \quad \tau_2' = \mathtt{typeReplace}(\overrightarrow{T}, \vec{\tau_1}, \tau_2) \quad \Gamma \vdash e : \tau_2'\end{array}}{\Gamma \vdash cn[\vec{\tau_1}](e) : un[\vec{\tau_1}]}\ (\textsc{constructor})$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : (\ \vec{\tau_1}\ ) \quad |\vec{\tau_1}| = |\vec{x}| \\ \Gamma' = \mathtt{tupGamma}(\vec{x}, \vec{\tau_1}, \Gamma) \quad \Gamma' \vdash e_2 : \tau_2\end{array}}{\Gamma \vdash e_1\ \textbf{match}\ \{(\textbf{case}\ (\ \vec{x}\ ) \Rightarrow e_2) :: []\} : \tau_2}\ (\textsc{match-tup})$$

$$\frac{\begin{array}{c}\Gamma \vdash e : un[\vec{\tau_1}] \quad un \in \mathtt{keys}(tdefs) \quad \mathtt{casesSane}(\overrightarrow{case}, un) \quad (\overrightarrow{T} \cdot m) = tdefs(un) \\ |\overrightarrow{T}| = |\vec{\tau_1}| \quad \vec{\tau_2} = \mathtt{casesTypes}(\overrightarrow{case}, \Gamma, \overrightarrow{T}, \vec{\tau_1}, m) \quad \tau_3 = \mathtt{asSingleton}(\vec{\tau_2})\end{array}}{\Gamma \vdash e\ \textbf{match}\ \{\overrightarrow{case}\} : \tau_3}\ (\textsc{match-constructor})$$