When stuck, install a type-system

Before we begin...

Welcome back to week whatever it is of the programming languages course! Just for the sake of better mutual understanding, let's give that handout syntax once again here:

- 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

By now, you've written a lot of `SimpleScala` , and one of the things you probably **love** is getting stuck. Most of the reasons behind getting stuck are hidden away behind our semantics table being less restrictive than it should be. For example, the rule that defines anonymous function callsreads $e_1(e_2)$ and permits **any** expression to be an anonymous function. This way, the following is quite okay in `SimpleScala` , although it gets stuck really fast:

$$(2 + 3)\ (7)$$

Today, we present a way to fix this by checking these weird cases through a fine filter, by doing type checking. Type checking means assigning tags, called types, to values, and having a predefined notion of which types can play well together in which situations.

We'll be learning about types themselves. What you currently know is probably that these things exist, that, when asked, you should start with "um, int, bool, char?", but not much more. We'll be learning about ways to combine types in several ways, and what these things are useful for. At the end of this handout, you will be capable of thinking in types, your mind will be ruined and you will be a slave of the dreaded empire!

We'll also be learning about how to write and read this kind of thing, and we'll be applying all of that to our `SimpleScala` language. At the end of the third assignment (which is you coding all of this), you will have a language that doesn't let you run your program if there's a chance of getting stuck because of a stupid mistake written at 2:00AM.

2 Types

> A type system is the most cost effective unit test you'll ever have.
>
> — Peter Hallam, designer of C#, then technical lead of the C# compiler, then javascript wiz for Google (he's seen the good, the bad and the ugly)

Types are annotations that will stop our programs from doing certain bad things. They aren't a cure-all, but are extremely useful for making the runtime of our application more stable. Depending on how deep the rabbit hole we drop, type systems can have many properties and solve some not-so-obvious problems.

Type systems can be of many shapes and sizes, and there's confusion galore about the terminology describing them. Most people know of the terms **strongly** and **weakly** typed, but mix them with the terms **statically** and **dynamically** typed, while these terms have nothing to do with each other. To talk about these, maybe we should talk about errors a bit. These terms can be useful while thinking about your future type system.

As a general compass, this is how errors are classified:

- A **trapped error** makes the program halt immediately and signal the error.
- An **untrapped error** allows the execution to proceed.
- A program is **safe** if it doesn't have untrapped errors.

Those error types affect the kind of language we are building in many ways. Depending on our choice of properties, we have...

- A language that can guarantee that all of its programs are safe, is considered, errm, **safe**.
- A safe language that also can guarantee that its programs don't have some trapped errors (a set of forbidden errors, chosen by the language designer) is called **strongly typed**. Otherwise, it's a **weakly typed** language.
- Independent of that, a **statically typed** language, a type system is used at compile time to filter out potentially ill-behaved programs.
- In a **dynamically typed** language, runtime checks are used to detect and reject ill-behaved programs.

> ***Sidenote:*** You can basically skip all of this if you're doing this just for the grade. If it's boring you (if not, send a message to Miroslav on whichever channel you want, include the word "meticulous" and win a surprise coffee-and-cake break!), but there's also...
>
> - Further down the road, a statically typed language can be **explicitly typed**, in which the programmer annotates variables with type information (think Java)
> - Opposite of that, an **implicitly typed** looks like a dynamically typed program, with no obvious type information given by the programmer, but rather infered by the compiler or interpreter
> - something in-between dynamically and statically typed languages, called **gradually** typed languages,
> - **linear** type systems, in which every variable of a type is used at most once,
> - or an **ordered** type system, in which variables are used exactly once and in the order in which they are defined.
> - For a different thing, a **dependent** type system can work with types dependent on values, like saying that there's a type of "pairs of integers where the first is larger than the second", for instance
>
> Type systems are a big topic for PL researchers, and if you're ever wondering what we do for a living in the PL lab, this is a safe bet.

3 Typed SimpleScala

Lo and behold, we will now start changing our SimpleScala language to add types to it. Nothing changed in the semantics (that would be a bit counter-productive, as we're building types to help us enforce the semantics), but the syntax has to change a bit. Mostly we are adding **type annotations** to variables and a new kind of definitions, called **type definitions**.

3.1 Type annotations

The two places which changed from handout 2 are the following:

$$x \Rightarrow e \qquad\qquad \textbf{def}\ fn(x) = e$$

These now look like this:

$$(x : \tau) \Rightarrow e \qquad\qquad \textbf{def}\ fn(x : \tau_1) : \tau_2 = e$$

What this is saying is that now the variable x and the function fn must align themselves with types, and these types are the following:

$$\tau \in \mathit{Type} ::= \textbf{String} \mid \textbf{Boolean} \mid \textbf{Int} \mid \textbf{Unit} \mid \tau_1 \Rightarrow \tau_2 \mid (\ \vec{\tau}\ ) \mid \mathit{userDefined}$$
$$\mathit{userDefined} \in \mathit{UserDefinedTypeName}$$

This makes SimpleScala much closer to what we have in Scala, as we now denote our types when defining functions. All of our basic building blocks have types assigned to them, including strings, booleans, integers and unit. We also have a type to represent our function values, denoted as $\tau_1 \Rightarrow \tau_2$, meaning "from type to type". Note that a curried function will basically have one function type lead into another: $\tau_1 \Rightarrow (\tau_2 \Rightarrow \tau_3)$; we'll write this with no parentheses, like $\tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3$, and you want to fight us about it, please do so during office hours and discussions. We'll actually award points for a worthy discussion!

> ***Don't drink too much Kool-Aid:*** *The definition of "worthy" can be found somewhere in the Marvel universe and is the same as in "Whoever is worthy can lift the hammer Mjolnir and harness the power of THOR!"*

You can imagine that combining tuple types, function types and the as-of-yet-undefined-but-alluring user defined types, and filling the holes with basic types can give us some great types to work with! In terms of what this means for program safety, the more complex the type system, the more problems it can cover. These function definitions are a bit different than before, and for your use in the code, we make a globally available map *fdefs*, which keeps a pair of two types: one for the argument (x) and one for the function's return value.

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

It's important to understand that the type of the return value and the type of the function itself are two different things: the return type is τ_2, for example, while the function type is $\tau_1 \Rightarrow \tau_2$.

3.2 Type definitions

One of the overwhelming problems with SimpleScala , causing depression and existential angst, was the thought of not being able to isolate ourselves to using just some limited subset of correct values. For example, it would be nice if lists would work with just `Cons(Int)` and `Nil(Unit)`, but this isn't the case, as we can construct a `Cons(1, Cons(2, Banana(7)))`. With the addition of types, we'll introduce a special kind of type, called **algebraic** types. These are types made for matching, consisting of several disjoint parts. You've seen this in Scala, there we call them **sealed traits** with **case classes**.

> ***Sidenote:*** The name **algebraic** comes from the word "algebra", which itself comes from the "al-jabr", meaning "reunion of broken parts", first used in the famous book "Al-kitab al-mukhtaṣar fī ḥisāb al-ğabr wal-muqābala" or "The Compendious Book on Calculation by Completion and Balancing". The author of the book was none other than Persian mathematician Muḥammad ibn Mūsā al-Kwārizmī. The translation of his works have built the foundation of the western scientific view of the 12th century. The latin variant of Al-khwarismi's name, algorismi, is well-known to us today, as the root of the word "algorithm".

An algebraic type is defined with the new syntax rule:

$$\begin{aligned} tdef \in UserDefinedTypeDef &::= \textbf{algebraic}\ un = \overrightarrow{cdef} \\ cdef \in ConstructorDefinition &::= cn(\tau) \end{aligned}$$

The constructors in *cdef* are separated by a |, so for example (to start off your next assignment and go back to lists):

$$\textbf{algebraic}\ \texttt{List}\ =\ \texttt{Cons}(\textbf{Int})\ |\ \texttt{Nil}(\textbf{Unit})$$

This makes the type `List` valid in our programs, and furthermore, lets us constrain the pattern-matching of a list to only a `Cons`(**Int**) and `Nil`(**Unit**). Many stuck situations just dispersed by themselves!

As with the function definition map *fdefs*, our user defined types are defined in a similar globally accessible map:

$$tdefs \in TypeDefs = UserDefinedTypeName \to (ConstructorName \to Type)$$

To see how this structure keeps the data about our types, consider the `List` from above. It would generate the following entries in the code:

$$tdefs = [\texttt{List} \to [\texttt{Cons} \to \textbf{Int}, \texttt{Nil} \to \textbf{Unit}]]$$

3.3 Program definition

Our program was previously defined as:

$$prog \in Program ::= \overrightarrow{def}\ \ e$$

This just won't do anymore, because we need to add our type definitions in now. This is going to change it only slightly:

$$prog \in Program ::= \overrightarrow{tdef}\ \ \overrightarrow{def}\ \ e$$

As you can see, we put all the type definitions before all the function definitions, before evaluating that one expression that represents our entry-point.

To check the full syntax, take a look at the end of this handout.

4 Typing rules

Now that we have the architecture to support our fancy type system, we need to actually form the rules of how types intermingle. These rules form **type judgements** which mark the correct constructions of types in our programs. Let's learn how to read them.

4.1 The typing environment

To start off, we need to introduce the most basic typing environment, which is a mapping from free variables to types (similar to the value environment ρ, which maps from variables to values). The typing environment is usually called Γ. To say that a variable x is of type τ in our environment, we'd write:

$$\Gamma \vdash x : \tau$$

You can remember to read this as "In the environment Γ, x has type τ". Of course, Γ is just a map; a map from variable to type:

$$\Gamma \in TypeEnv = Variable \to Type$$

4.2 Inference rules

We use inference rules to construct more complicated uses in the typing environment. Inference rules tell us what we may infer, and under which premises. We usually write them as:

$$\frac{premises}{conclusion}$$

In our case, both the *premises* and the *conclusion* are typing rules, so, for example, we'd have:

$$\frac{x \in keys(\Gamma) \quad \tau = \Gamma(x)}{\Gamma \vdash x : \tau}$$

We read this as "if the premises are that x is a variable in Γ, and that τ is what x is mapped to in Γ, the conclusion is that in Γ, x is of type τ".

Usually you'll find the names of the rules next to the rules themselves; the one above would be called (VAR), as it defines how variables are typed.

A conclusion with no premise is called an **axiom**, and is written like this:

$$\frac{}{\Gamma \vdash b : \textbf{Boolean}} \text{ (BOOLEAN)}$$

We read this as "in the environment Γ, an expression which represents booleans is of type **Boolean**. It's important to register that the names of these expressions aren't actual variables, but rather syntactic expressions. An example which makes this obvious is the following:

$$\frac{\Gamma \vdash e_1 : \textbf{String} \quad \Gamma \vdash e_2 : \textbf{String}}{\Gamma \vdash e_1 + e_2 : \textbf{String}} \ (+_{string})$$

This is just what it looks like: if both e_1 and e_2 are of type **String** in Γ, then their concatenation is also of type **String** in Γ. Your next assignment will be figuring out how to write all these rules into `SimpleScala` , so take your time and examine the handout, and the rules at the end. As with the last time, the helper functions are given.

4.2.1 So... uhm... what?

Once you have inference rules, anything type-correct can be infered from these simple rules. If you have a complicated expression, and can't figure out what type it is, you can simplify it using these rules. If you can't get there, that's a **type error**.

You might be wondering where this method starts and where it ends. Your program is only one expression, after all, so it will have to match some of our rules. If it does, then it will have potential premises about its sub-expressions, which, again, should match some rules, and so on. This goes on, recursively, until it reaches axioms, after which there's nothing to match and if nothing failed, we're okay.

> ***Sidenote:*** We ♡ type errors because they help us find whatever plagues our programs before our code gets used in a nuclear powerplant (and yes, one of your TAs does have his code running in at least one room with lots of radiation, somewhere in Russia and South America!)
> Please be safe.

4.3 Notation

4.3.1 Length of a list

The length of a list is marked by the absolute value symbol: $|\cdot|$. If you see this $|\vec{a}|$, that means the length of $\vec{a}$.

4.3.2 Tuple vs list notation

Tuples use a dot to separate them, as in: $(\tau_1 \cdot \tau_2)$. Lists don't use this, and rather go with ::.

4.4 Helpers

4.4.1 Create the Γ environment of a block

The aptly called `blockGamma` function takes a list of val-expressions and a type environment (old, pre-block Γ), and returns the new type environment, with all the variables from the val-expressions mapped to their respective types.

4.4.2 Retrieve the types of the elements of a tuple

Given a list of expressions, `tupleTypes` gets each of their types.

4.4.3 Create the Γ environment of a tuple

Very similar to `blockGamma`, the `tupGamma` function takes in a list of variables, their respective types and an old type environment and returns a new Γ, filled with those variable-to-type mappings.

4.4.4 Check if all the cases of a user-defined type are sane

This function is our pattern-matching type-safe jackpot! Given a list of cases and a user defined type name, `casesSane` returns true if the following things are true:

- Each constructor name used in the cases is mentioned in the user-defined type for the case
- No two cases passed to this function share the same name
- No case mentioned in the user-defined types is missing in the list of cases

4.4.5 Get the type of each case in a list of cases

What the description says, only better — that's `casesTypes`. Takes a list of cases, a type environment and a mapping from constructor names to types, it returns a list of types.

4.4.6 Check whether all given elements of a list are the same

This is a weird one, but very useful. Given a list of all the same elements, it returns one of them. Gets stuck if they aren't all the same or if the list is empty. For example, `asSingleton`$((1, 1, 1)) = 1$, but `asSingleton`$((1, 1, 2))$ gets stuck. Really useful for the pattern matching type rule, right?

5 Done!

You may notice that we're not really holding your hand so much on this one. This is for many reasons, some of which have to do with us being slightly chaotic evil, some of which are just us wanting you to figure it out for yourselves. As always, we're open for questions, but we really hope you read through this multiple times, before asking some trivial thing like "so, what's Γ?", because it helps both sides. We'll try our best to answer, but please, respect our time as we respect yours.

A cute side-effect of how this course is done, is that you have our REPL which you can use to answer the questions like "is this program well-typed?" or "what is the type of this program?". However, the more essential questions, like "why is this program of this type?", which are more important for actually understanding stuff in life, you have to figure out for yourselves from that.

I'm really sorry I wasn't able to setup LaTeX so that I can print out Al-Khwarismi's name and his book's name in their original forms, but I'll make due somehow.

After doing this assignment, you will have formed an interpreted language with a type system on top, which is very cool. You have officially reached the step where you can write programs and get some nice output as a result. The next assignment after that is going to be a little bit different, so I won't even mention it here. Yet. Anyway, it's awkward silence at the end of handout time, so I'll just go into the tables and inference rules now.

5.1 Syntax

$$x \in \mathit{Variable} \qquad \mathit{str} \in \mathit{String} \qquad b \in \mathit{Boolean} \qquad i \in \mathbb{Z} \qquad n \in \mathbb{N}$$
$$\mathit{fn} \in \mathit{FunctionName} \qquad \mathit{cn} \in \mathit{ConstructorName} \qquad \mathit{un} \in \mathit{UserDefinedTypeName}$$

$$\tau \in \mathit{Type} ::= \textbf{String} \mid \textbf{Boolean} \mid \textbf{Int} \mid \textbf{Unit} \mid \tau_1 \Rightarrow \tau_2 \mid (\ \vec{\tau}\ ) \mid \mathit{un}$$

$$\begin{aligned} e \in \mathit{Exp} ::= {} & x \mid \mathit{str} \mid b \mid i \mid \textbf{unit} \mid e_1 \oplus e_2 \\ & \mid (x : \tau) \Rightarrow e \mid e_1(e_2) \mid \mathit{fn}(e) \\ & \mid \textbf{if}\ (e_1)\ e_2\ \textbf{else}\ e_3 \\ & \mid \{\ \overrightarrow{\mathit{val}}\ e\ \} \\ & \mid (\ \vec{e}\ ) \mid e.\_n \\ & \mid \mathit{cn}(e) \mid e\ \textbf{match}\ \{\overrightarrow{\mathit{case}}\} \end{aligned}$$

$$\mathit{val} \in \mathit{Val} ::= \textbf{val}\ x = e$$

$$\mathit{case} \in \mathit{Case} ::= \textbf{case}\ \mathit{cn}(x) \Rightarrow e \mid \textbf{case}\ (\ \vec{x}\ ) \Rightarrow e$$

$$\oplus \in \mathit{Binop} ::= + \mid - \mid \times \mid \div \mid \wedge \mid \vee \mid < \mid \leq$$

$$\mathit{tdef} \in \mathit{UserDefinedTypeDef} ::= \textbf{algebraic}\ \mathit{un} = \overrightarrow{\mathit{cdef}}$$

$$\mathit{cdef} \in \mathit{ConstructorDefinition} ::= \mathit{cn}(\tau)$$

$$\mathit{def} \in \mathit{Def} ::= \textbf{def}\ \mathit{fn}(x : \tau_1) : \tau_2 = e$$

$$\mathit{prog} \in \mathit{Program} ::= \overrightarrow{\mathit{tdef}}\ \overrightarrow{\mathit{def}}\ e$$

5.2 Type Rules

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

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

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

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

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

$$\frac{fn \in \texttt{keys}(fdefs) \quad (\tau_1 \cdot \tau_2) = fdefs(fn) \quad \Gamma \vdash e : \tau_1}{\Gamma \vdash fn(e) : \tau_2}\ (\textsc{nameCall}) \qquad \frac{\Gamma \vdash e_1 : \textbf{boolean} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \textbf{if}\ (e_1)\ e_2\ \textbf{else}\ e_3 : \tau}\ (\textsc{if})$$

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

$$\frac{cn \in \texttt{keys}(cdefs) \quad un = cdefs(cn) \quad \Gamma \vdash e : \tau \quad (tdefs(un))(cn) = \tau}{\Gamma \vdash cn(e) : un}\ (\textsc{constructor})$$

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

$$\frac{\begin{array}{c}\Gamma \vdash e : un \quad un \in \texttt{keys}(tdefs) \quad \texttt{casesSane}(\overrightarrow{case}, un) \\ \vec{\tau_1} = \texttt{casesTypes}(\overrightarrow{case}, \Gamma, tdefs(un)) \quad \tau_2 = \texttt{asSingleton}(\vec{\tau_1})\end{array}}{\Gamma \vdash e\ \textbf{match}\ \{\overrightarrow{case}\} : \tau_2}\ (\textsc{match-constructor})$$