Chapter 1

Typing

I find typing really fun, but all my friends think its keyboring

Klyiff

1.1 Introduction

Typing is more than just the process of hitting keys on a keyboard. In the programming language space, typing refers to categorizing types of data and how to determine how to use the data given to you. Fun fact: typing came from philosopher and mathematician Bertrand Russell [1]. This was then later used to make a simply typed lambda calculus (future chapter) which some can argue is the basis for all typed functional *and* imperative languages. We don't care about that and it's soundness here but just a fun fact for you to know.

Let's first begin by considering the following:

```
(* Invalid Ocaml *)
let f () = if 1 then 3 + 5 else 'a'

(* Valid python *)
def f():
  if 1:
    return 3 + 5
  else:
    return 'a'
```

These two programs seem to do similar things, but this only works in one language and not the other. This is because their **type system** is different. A type system is a series of rules that do two things (kinda). First they assign a **type** to values/expressions/variables (kinda). Second, they describe what you can do with values/expressions/variables of a certain type.

We can ultimately say that a type is a identifier that describes a property of something. For example, Cliff's hair can fall under the type of 'black' because it has certain properties we associate with black (eg. absorbing a certain amount of light waves preventing the reflection of the visible light spectrum). We could say that 42 is a real number because it has properties we associate with real numbers: being on the continuous number line. 42 is also a integer because it has certain properties (being a whole number for instance). Colloquially, you may hear someone say: "Wow you're my type" [2] and this kinda means that you fulfil all typical attributes someone likes.

[1] See Russell's Paradox

[2] I assume, I wouldn't know

For programming languages, we think of types as *data types*. 5 is an int because it fulfils certain properties (stored in 4 bytes in base 2 for examples) whereas 1.2 is a float because it fulfils different properties (stored in 4 bytes using IEEE-754 standard).

However, what is known as **type checking** is typically what people think of when they think of typing. This focuses on the second point: applying the typing rules to determine what can be done with certain data types.

1.2 Type Checking

Let us go back to the earlier example we provided:

```
1 (* Invalid Ocaml *)
2 let f () = if 1 then 3 + 5 else 'a'
3
4 (* Valid python *)
5 def f():
6   if 1:
7     return 3 + 5
8   else:
9     return 'a'
```

If we tried to run the OCaml code, something would prevent us from compiling the code because of a few reasons. UTop gives us the following reason: `"[1] has type int but an expression was expected of type bool because it is in the condition of an if-statement."`. However, the python code runs fine as is. We said this is because their type system was different. Now that we know the definition of a type system, we can say that the rules oh how we use data differs in each language. In OCaml, we cannot use `int` types in the conditional check. In Python however, we are allowed to use `int`s to do so. Another example is that OCaml puts a restriction on what can be added to an `int` (just other `int`s), whereas Python has a less strict rule system about what you can add to an `int`.

In order to check the types, a type checker has to run. This is typically done before the code is run (static type checking) or during code execution (dynamic type checking). A type checker's rules is similar to operational semantic rules, except instead of talking about what occurs, it describes what is allowed to occur. Let' see an example.

1.2.1 Our First Type Rule

Let's begin with our very first rule.

$$\frac{}{G \vdash true : bool}$$

Let us define some things. Very much like the environment in operational semantic rules, we need something to store data for our type rules. In this case we will call it G, the *context*. Thus, we can read this rule as "the expression true has type bool in the context G". Much like the environment A, G is a partial function that maps variable names to types. We will get back to this in a bit, let's first see some more rules:

$$\frac{}{G \vdash false : bool} \qquad \frac{}{G \vdash n : Int}$$

These are constant rules which say that false will always have type bool, regardless of the context, and an integer constant will always have type int. With this as our basis, what about variables which could vary? We now go back to examining the context G.

We said that G was a function that took in a variable and returned a type. We could say that for some variable x, G(x) would return x's type. Put into a rule, we could say the following:

$$\frac{}{G \vdash x : G(x)}$$

Simply put, given a context G, x evaluates to type `G(x)`. I like to write this rule however as the following:

$$\frac{G(x) = t}{G \vdash x : t}$$

Just makes it similar to how I write my Operational Semantic rule for variable lookup.

1.2.2 Type Restrictions

Now that we have some basic type rules under our belt, we can discuss some rules that put restrictions on what what certain expresions can be. Let us start with the OCaml if expresion.

$$\frac{G \vdash e_1 : bool \qquad G \vdash e_2 : t \qquad G \vdash e_3 : t}{G \vdash if\ e_1\ then\ e_2\ else\ e_3 : t}$$

This says that given some context G, if e_1 has type bool, and e_2 and e_3 both have type t, then the expression $if\ e_1\ then\ e_2\ else\ e_3$ has type t. This only holds true if e_1 is a bool. If this rule cannot be held, then the type checker will return an error.

Let's see some more rules.

$$\frac{G \vdash e_1 : int \qquad G \vdash e_2 : int \qquad G \vdash + : int \to int \to int}{G \vdash e_1 + e_2 : int}$$

$$\frac{G \vdash e_1 : bool \qquad G \vdash e_2 : bool \qquad G \vdash \&\& : bool \to bool \to int}{G \vdash e_1 \&\& e_2 : int}$$

$$\frac{G \vdash e : int \qquad G \vdash eq0 : int \to bool}{G \vdash eq0\ e : bool}$$

$$\frac{G \vdash e_1 : t \qquad G \vdash e_2 : t \qquad G \vdash =: t \to t \to bool}{G \vdash e_1 = e_2 : bool}$$

Notice there is a restriction placed upon the expressions which dictate how each operator is to be used. For example, the + operator can only work on `ints`. For the penultimate rule, let us assume there is a function called `eq0` which returns true if the input is 0 and false otherwise. Another interesting note is the last rule. The restriction is not upon any particular type, but that the two arguments must be of the same time.

1.2.3 Let and Functions

Technically let expressions are the internal workings of a function call which is why I grouped let expressions and functions together, but this is outside the scope of this class (kinda. See lambda Calculus).

When talking about let expressions and functions, we will start to need to extend the context, as well as use it to store data. We will start with let expression before moving to functions. Below is an example let expression type rule for the OCaml language:

$$\frac{G \vdash e_1 : t_1 \qquad G, x : t_1 \vdash e_2 : t_2}{G \vdash let\ x = e_1\ in\ e_2 : t_2}$$

This says that if e_1 has type t_1, and if in the extended G context with the binding of the variable x to the type t_1, the expression $_2$ has type t_2, then the expression $let\ x = e_1\ in\ e_2$ has type t_2.

Moving onto functions, we will be using the anonymous function syntax of below:

$$\frac{G, x : t_1 \vdash e : t_2}{G \vdash fun\ (x : t_1) \to e : (t_1 \to t_2}$$

This one is tricky. To read this we say, that if we have a function which takes in a parameter x which has type t_1, and has an expression e that has the type t_2 in the context $G, x : t_1$, then the expression `fun x -> e` has type $t_1 \to t_2$. This can

then be chained as needed for any function through currying.

Lastly we have the function call. This requires us to use the above function type rule as a basis.

$$\frac{G \vdash e_1 : (t_1 \rightarrow e_2) \qquad G \vdash e_2 : t_1}{G \vdash e_1\ e_2 : t_2}$$

This says that in some context G, if e_1 has type $t_1 \rightarrow t_2$, and e_2 has type t_1, then the expression $e_1\ e_2$ has type t_2. Notice how this is just a generalization of the let expression[3].

1.2.4 Type Proofs

Now that we can read rules, we can now make proofs about the types of expressions we have in OCaml. Very much like we can use OpSem rules to proove correctness of a program, we can use the previously defined rules to make a proof about the type system for the language.

For example: consider the type proof for the expression: `let x = 3 in if true then x else x + 3`

$$\dfrac{\dfrac{}{G \vdash 3 : int} \quad \dfrac{\dfrac{}{G, x : int \vdash true : bool} \quad \dfrac{}{G, x : int \vdash x : int} \quad \dfrac{\dfrac{}{G, x : int \vdash x : int} \quad \dfrac{}{G, x : int \vdash 3 : int} \quad G, x : int \vdash + : (int \rightarrow int \rightarrow int)}{G, x : int \vdash x + 3 : int}}{G, x : int \vdash \text{if } true \text{ then } x \text{ else } x + 3 : int}}{G \vdash \text{let } x = 3 \text{ in if } true \text{ then } x \text{ else } x + 3 : int}$$

Notice how this is very similar to OpSem proofs, but instead of showing what everything evaluates to, it instead shows the types of everything. It is important to note that if a proof cannot be made, then it fails to type check.

1.2.5 Subtyping

For the most part typing is an ontological problem. Ontology deals with figuring out what it means to be something. We said earlier that for data types, it is typically the case that an entity x is type t because it has properties associated with t. In the introduction we said that 42 was an real number because it has properties that real numbers have. For philosophers, many ontological ventures try and figure out the core properties that make something itself with no overlap. We don't care about that. We acknolwedge that some things share properties. Again, 42 is both a real and an integer. Subtyping talks about entities that have multiple properties.

Let's consider the very classic example of squares and rectangles. We say that all squares are rectangles. That is whie squares have square properties, squares also have rectangles. So we say that squares are a (sub)type of rectangle. In particular, we can apply the Liskov substitution principle, what talks about types S, T and some property P.

$$Subtype(S, T) : \forall x \in T.P(x) \Rightarrow \forall y \in S.P(x)$$

This basically means that I can use S where I expect T is S is a subtype of T. In terms of properties, if all values of a type T have a property, then all values of a subtype S also satisfy that property. For example: A property of a rectangle is have four internal right angles. All Squares satisfy this principle. Sure, Square satisfy more properties (eg. having the same length and width), but they also satisfy all rectangle properties. Because we are talking about satisfying properties and not the number properties, subtypes are more specific than their supertype.

With this in mind, we can now consider why (or why not) OCaml and C disallow/allow the addition of ints and floats. In OCaml, the only type rule about the + operator is something like the following:

$$(ocaml - int - add)\frac{G \vdash e_1 : int \qquad G \vdash e_2 : int \qquad (+) : int \rightarrow int \rightarrow int}{G \vdash e_1 + e_2 : int}$$

whereas something in C could have something like:

[3] Consider that $let\ x = 3\ in\ x + 1$ is the same as $(fun\ x \rightarrow x + 1)\ 3$

$$(c-add)\frac{G \vdash e_1 : int \qquad G \vdash e_2 : int \qquad (+) : int \to int \to int}{G \vdash e_1 + e_2 : int}$$

$$(c-add)\frac{G \vdash e_1 : float \qquad G \vdash e_2 : float \qquad (+) : float \to float \to float}{G \vdash e_1 + e_2 : float}$$

$$(c-add)\frac{G \vdash e_1 : float \qquad G \vdash e_2 : int \qquad (+) : float \to int \to float}{G \vdash e_1 + e_2 : float}$$

$$(c-add)\frac{G \vdash e_1 : int \qquad G \vdash e_2 : float \qquad (+) : int \to float \to float}{G \vdash e_1 + e_2 : float}$$

However if we generalize ints and floats to a larger supertype (like numbers), we can use one rule:

$$(c-add)\frac{G \vdash e_1 : number \qquad G \vdash e_2 : number \qquad (+) : number \to number \to number}{G \vdash e_1 + e_2 : number}$$

Now all we need is some way to say that ints are numbers and floats are numbers. We can do so by introducing a subtype rule:

$$(int-num)\frac{G \vdash n : int \qquad int <: number}{G \vdash n : number}$$

The $<:$ means subtype[4]. So we read this as "In the context G, n is of type number if n is of type int in the context G and that ints are a subtype of number". We can then say something similar with floats:

$$(floats-num)\frac{G \vdash n : float \qquad float <: number}{G \vdash n : number}$$

These rules can be abstracted away to a general subsumption rule:

$$\frac{G \vdash s : S \qquad S <: T}{G \vdash e : T}$$

It is important to note that subtypes are reflexive and transitive:

$$x <: x \qquad x <: y \land y <: z \Rightarrow x <: z$$

Records

Many record-supported languages (OCaml included) will use subtyping to typecheck records. With OCaml records, there are 3 main subtyping rules to consider when type checking records.

First we can say that if we have a record like $\{x : int\}$, then we can say this is a record with a label x that is of type int. As discussed earlier, a subtype is a more specific type of its supertype. So in this case, any record with a label x that is of type int is its subtype, irregardless of how many other labels and types are in the record. For example $\{x : int; y : int\} <: \{x : int\}$.

This is because a longer record is more informative and hence describes a smaller number of entities. To formalize this rule we can say the following:

$$\{l_i : T_i^{i \in 1 \ldots n+k}\} <: \{l_i : T_i^{i \in 1 \ldots n}$$

All this says is that records with labels $l_1, l_2, \ldots, l_{n+k}$ are a subtype of records with labels $l_1, L_2, \ldots l_n$ (assuming the label-type pairs are the same: $\{x : int; y : float\} \not<: \{x : string\}$).

Additionally, the types of the individual fields (of a particular label l) can be subtypes of each other and that also is fine. For example: $\{x : int\} <: \{x : number\}$. We can generalize this with the following rule:

$$\frac{\forall i . S_i <: T_i}{\{l_i : S_i^{i \in 1 \ldots n}\} <: \{l_i : T_i^{i \in 1 \ldots n}\}}$$

[4] $\sqsubseteq$ is an alternative notation. I like <: because you can actually type that on a keyboard

To put the past two rules together we could say $\{x : int; y : \{a : int; b : int\}\} <: \{y : \{a : number\}\}$

The last rule is pretty straightforward: the order of the labels should not matter. So $\{x : int; y : string\} <: \{y : string; x : int\}$.

Functions

Functions can also be subtyped. For functions, our input can get more general while our output gets more specific when mvoing from type to subtype. This is represented as:

$$\frac{U <: S \qquad T <: P}{S \to T <: U \to P}$$

Using our square-rectangle idea:

- If our function takes in squares, then it takes in entities that satisfy rectangle properties.
- If our function returns rectangles, then it can also return any square.

1.3 Type Systems

The question may then arise: "why do so many type systems exist?". Ultimately because the language philosophy is tied to the type system. Languages were designed a certain way to solve problems, and the type system reflects that. I don't need a type system that enforces buffer-overflow checks if there are no arrays in the language. So in this regard, we categorize type systems based of properties the type system and how these properties effect the language itself.

One property we do care about a type system is how useful it is to enforce good and bad programs. In something like OCaml `if 1 then true else "a"` is a bad program, and ocaml will not compile it. OCaml's type system rejects this program. This is good because it saves runtime crashes later down the line.

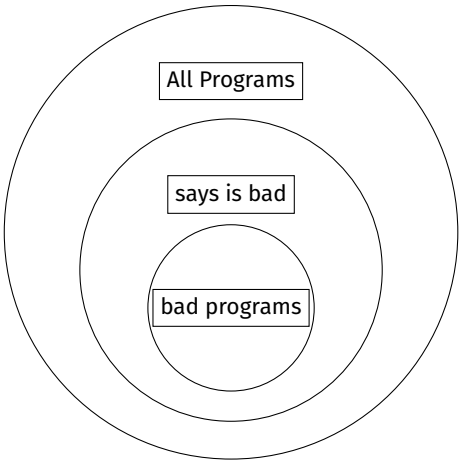
In this context, a type system may have the following (but never all 3):

- Soundness
- Completeness
- Decidability

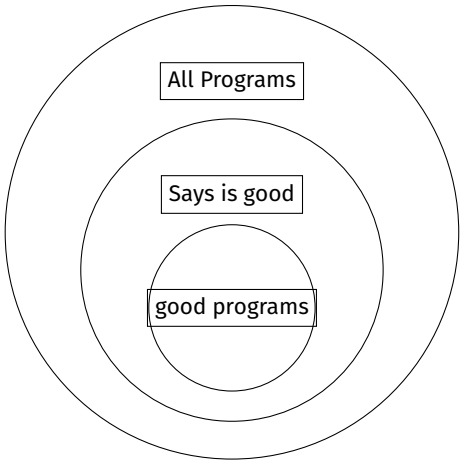
The first is **soundness**. A sound type system will reject all bad programs. The easiest sound type system will reject everything. This is because soundness doesn't care about accepting good programs. If I reject all programs, then I do end up rejecting all bad programs. Sure, I end up rejecting good programs, but that's not part of the definition.

The second is **completeness**. A complete type system will accept all good programs. The easiest complete type system is one that accepts all programs. This is because completeness doesn't care about rejecting bad programs. If I accept all programs, then I do end up accepting all good programs. Sure, I end up accepting bad programs, but that's not part of the definition.

Consider the following Venn-diagrams:



Sound



Complete

If a type system is both sound and complete: then it rejects all bad programs, and accepts all good programs. This is ultimately the goal, but this cannot be practically achieved. This is due to the last property: **decidability**.

Decidability is a property of the type system implementation. If we wrote a type checker, the typew checker would have to come back and say yes or no: accept or reject the program. The issue here is, in order to do this, you would need to be able to analyze every possible program that could exist, and doing so reduces down to the halting problem. The halting program is a problem that has been proved to be undecidable, that is there is no possible solution for general purpose programs. The halting problem states: there is no program that can check to see if a all general-purpose programs will terminate or not. That is, for all programs p, there is no program H that can exist where $H(p)$ is true for all p (where $H(p)$ is true if the program halts and false otherwise).

If we cannot check if a program terminates, there is no way to type check the entire program. So type checkers will prematurely end early with a result of accept or reject. Because it prematurely ends, it can either to reject unknown or accept unknown programs. Thus, a practical type checker can be sound or complete, but not both.

This leads us to talk about properties of programs. The two properites are:

- well typed vs ill-typed
- well defined vs ill-defined

A program is **well typed** if the type system accept it. A program is **ill-typed** if it is rejected by the type system. Something like `if true then false else true` is well typed in OCaml, whereas `if 1 then true else "a"` is ill-typed in OCaml.

A well-defined program is one where all aspects of the program have a semantic definition. An ill-defined program is one where there is no semantic definition in the language. For example:

```c
#include <stdio.h>
int main(char* args){
  int* buff;
  printf("%d",buff[4]);
  return 1;
}
```

This C program is well typed - this program will compile. But there is no definition or deterministic behavior from this program should we run it. Reading out of bounds in C has unspecifiec behavor. Hence it is ill-defined.

Additionally, we could have programs that are well-defined but won't be accepted by the type system. For example:

```
let f g = (g 1, g "hello") in f (fun x -> x)
```

We can look at this and understand that this should probably return a `int * string` tuple of `1,"hello"`, but this is not actually allowed in OCaml.[5]

Then, putting all this together, we can say a language is type safe if all well-typed programs are well defined. Typically this is really hard to prove for very large languages (and I suspect most of them aren't completely type safe) but it's accepted that baring a bugs in the language implementation, if the language has an intended meaning, then its type safe. So C is not type safe because C knows and accepts that there is undefined behavior in the language. Python on the other hand, says that throwing an error/crashing is expected behavior so its typically considered a type safe language.

1.4 Type Inference

Made with help from Maya Popova

1.4.1 Introduction

In languages like OCaml where we do not explicitly have to tell the compiler the types of data, but we still want to statically type check, we need to be able to infer the types of data and variables in order to determine if we are using values correctly. Many languages use's a Hindley-Milner algorithm to perform type inference and we will talk about parts of it here.

If you're observant, you may have noticed that in all the examples we've given, we either ask you about pre-created functions (like eq) or we tell you the expected type for the parameter (like $fun(x : t_1) \longrightarrow e_1$) instead of just $fun x \longrightarrow e_1$. This is because the latter case requires type-inference, not just type-checking.

Let's skip back to when we were learning about typing OCaml expressions. An expression like 4 is simple to type - we know 4 is an int. Similarly, $let\ x\ =\ 4\ in\ if\ true\ then\ x\ or\ x$ is simple to prove, because we know x has the type of the value it's bound to, which is int, and so we can keep track of this in the environment and find that the full expression must also have type int.

However, when working with functions, things become more interesting. Let's consider this example.

$$fun\ x \longrightarrow 1 + x$$

You, a human (probably), can pretty easily tell me that this is an $int \longrightarrow int$ function. But if we want to tell a computer to do this, we need to understand exactly what's happening at each step. We begin by realizing that this is a function. Being a function, the final type will be in the form $(type\ of\ x) \longrightarrow (return\ type)$. Since x is a parameter, we can't know anything about it yet! So we have to move on to try and find the return type by finding the type of $x + 1$, without having added anything to the environment.

$$1 + x$$

Okay, this is +, which we know is an operator that can only act on two ints and return an int. We know trivially that 1 is an int. But what about x? We don't know anything about x! The type of x isn't in our context (environment) G yet, so when we try to find its type, we fail - undeclared variable! DeclareError!

Clearly, this is not correct. But given the rules we've seen so far, we don't really have a way to do this. Step 1 of getting around this is that when we encounter a new variable that we can't yet know anything about (since it's being declared as a function parameter), we have to give it a temporary type, say $'a$, and put it in the context. Now let's go back to the body of the function $1 + x$, knowing that the type of $x = 'a$.

Again, this is +, which we know is an operator that can only act on two ints and return an int. We know trivially that 1 is an int. But what about x? Now we know what x is - it's type $'a$! Great - that's not exactly an int, but it could be, so that's fine. So not only can we say that there is no type error and the body/return type of the whole function is int, but we can also say that we now know that $'a$ must $= int$. We have found a constraint, C, that says $'a = int$.

This is great - we now know the input type is $'a$, the output type is int, and the "constraint" that $'a = int$. So we can "unify" the constraint with the type, and get full type $int \longrightarrow int$. The next two sections will talk more about "constraints" and "unification", and the section after that will talk about a problem that a naive approach to doing this might face.

[5] This is called paramatric polymorphism or rank 2 polymorphism

1.4.2 Constraints

Ultimately, when performing type inference we are going need to figure out what things are based on how we use them. We are "constrained" by how we use values. If I say something like `x = 1`, we can restrict x to type `int`. This is the basis of constraint-based type inference.

To see an example of this, let's look at this if expression. To be simple, we're gonna look at x and y as valid variables whose type we don't know yet without explicitly thinking about this in the context of a function.

```
1 if x then y else 4
```

In this example, we can constrain x to a bool because it is a guard expression, and y to be the same type as 4. Since 4 is a constant, we know its type as an int, meaning we can assume y to be an int. From an algorithmic standpoint, we could create a rule like the following:

$$\frac{G \vdash e_1 : t_1, C_1 \qquad G \vdash e_2 : t_2, C_2 \qquad G \vdash e_3 : t_3, C_2}{G \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t, C_1 \cup C_2 \cup C_3 \cup \{t_1 = bool, t = t_2, t = t_3\}}$$

We can read this as: Given some context G, the expression if e_1 then e_2 else e_3 has type `t` with some set of constraints $C_1 \cup C_2 \cup C_3 \cup \{t_1 = bool, t = t_2, t = t_3\}$. The first three sets of constraints C_i are the combination of all the constraints created in all the sub-expressions e_1, e_2, e_3, and three new constraints are the following:

- The type t_1 (the type of the guard) must be constrained to a `bool`
- The return type t should be the same as the type of e_2: t_2
- The return type t should be the same as the type of e_3: t_3

Due to transitivity, the last two constraints also say that e_2 and e_3 should have the same type. We could have also just said that t_2 should be equal to t_3 and that the output type is t_2. Eh. We also could have written those constraints on the top line, but it makes more sense to have them alongside the if itself, since the use of if specifically gives us those constraints. Regardless, we have an idea of how we can put constraints and type inference into our typing algorithm.

For expressions where no inference is needed, we can make the constraint set empty:

$$\frac{}{G \vdash n : int, \{\}}$$

So, putting that together, here's a filled-out type inference proof for the original expression (let's say we called y type a and x type b earlier as our "placeholder" types.

$$\frac{\dfrac{G, x : a, y : b(x) = a}{G, x : a, y : b \vdash x : a, \{\}} \quad \dfrac{G, x : a, y : b(y) = b}{G, x : a, y : b \vdash y : b, \{\}} \quad \dfrac{}{G, x : a, y : b \vdash 4 : int, \{\}}}{G, x : a, y : b \vdash \text{if } x \text{ then } y \text{ else } 4 : t, \{\} \cup \{\} \cup \{\} \cup \{a = bool, t = b, t = int\}}$$

So the final type is t with the constraints $\{'a = bool, t =' b, t = int\}$. None of these constraints are contradictory, and we can unify these together to find out that the full type must be int.

1.4.3 Unification

Once we have a list of constraints and the type of an expression, we can start applying the constraints to the type. This process is typically called unification. Unification will only unify useful constraints. If our constraints say something like `bool=bool`, then it is not useful. Let's consider we have the type: `'a -> 'b -> bool` and the constraints `{'a='b,'b=int}`. If we go through and unify, we should get that the final type should be `int -> int -> bool`. Additionally, if we get a conflicting constraint, like $\{t_1 = int; t_2 = bool, t_1 = t_2\}$, then we should fail to type check and throw an error or something.

1.4.4 Let Polymorphism (in OCaml)

So, we now know the general idea - first, type check as normal. If you can't know the type of something (like if it's a parameter of a function), give it a temporary value. Then, if we later find out that there is a constraint, save that constraint. Finally, at the end, unify the type with the temporary values in it with the set of constraints, and as long as there were no conflicts, we are good to go!

...Or are we? Consider the following:

```
let f x = x : 'a -> 'a
```

We know here that `f` is a function that takes in some type `'a` and returns that same type. Hence we can do both of the following:

```
let f x = x in
f true (* returns true:bool *)
let f x = x
f 3 (* returns 3:int *)
```

Consider what we can also do:

```
let f x = x in
if f true then f 1 else f 3
```

If we tried a naive approach to type inference, we might first find that f is some type $a \to a$. Then, we might first infer that a is a $bool$ because we first see `f true` in the guard, and add the constraint $a = bool$. Then, when we see `f` again in the true branch (`f 1`), we see that we are calling `f` with an input of type `int` and add the constraint $a = int$. Same in the false branch.

Oh no! One constraint says unknown type `f` is type $bool \to bool$ and the other says unknown type `f` is type $int \to int$. This is a constraint conflict, and so our type inference fails.

None of this is what we want to happen - our function is already defined fully and so we don't need to constrain it. We can apply all kinds of different types `'a,'b,'c...` to `f`, and they should all always work.

Let's consider our type checking rules for let, fun, and if expressions:

$$\frac{G \vdash e_1 : t_1 \qquad G, x : t_1 \vdash e_2 : t_2}{G \vdash let\ x = e_1\ in\ e_2 : t_2} \qquad \frac{G, x : t_1 \vdash e : t_2}{G \vdash fun\ (x : t_1) \to e : (t_1 \to t_2)} \qquad \frac{G \vdash e_1 : bool \qquad G \vdash e_2 : t \qquad G \vdash e_3 : t}{G \vdash if\ e_1\ then\ e_2\ else\ e_3 : t}$$

So let's write our proof about `let f = fun x -> x in if f true then f 1 else f 3`. I will skip a few steps for readability.

$$\dfrac{\dfrac{G, x :' a \vdash x :' a}{G \vdash \text{fun } x \to x :' a \to' a} \qquad \dfrac{G, f :' a \to' a \vdash f\ true : bool \qquad G, f :' a \to' a \vdash f\ 1 : int \qquad G, f :' a \to' a \vdash f\ 3 : int}{G, f :' a \to' a \vdash \text{if } f\ true \text{ then } f\ 1 \text{ else } f\ 3}}{G \vdash \text{let } f\ =\ \text{fun } x \to x \text{ in if } f\ true \text{ then } f\ 1 \text{ else } f\ 3}$$

If were to one-to-one our type checker with this proof, we get an issue: on the top right, `'a` should be both a `bool` and an `int`. This is because we are reusing the type `'a` and are assuming that `'a` should be a single type, not a variety of types.

Let's rewrite this proof, but now using a different name for the polymorphic `'a` each time we want to use the function, so that we don't assume that it should be a single type across each instance.

$$\dfrac{\dfrac{G, x :' a \vdash x :' a}{G \vdash \text{fun } x \to x :' a \to' a} \qquad \dfrac{G, f :' b \to' b \vdash f\ true : bool \qquad G, f :' c \to' c \vdash f\ 1 : int \qquad G, f :' d \to' d \vdash f\ 3 : int}{G, f :' a \to' a \vdash \text{if } f\ true \text{ then } f\ 1 \text{ else } f\ 3}}{G \vdash \text{let } f\ =\ \text{fun } x \to x \text{ in if } f\ true \text{ then } f\ 1 \text{ else } f\ 3}$$

We now say that `'a` could be any of the following: `'b,'c,'d`. The meaning is the same, but we avoid the issue that forces `f` to appear as if it has to "change" type rather than just accepting different types when it needs to.

To truly fix this issue, we shouldn't store `f` as a `'a -> 'a` function. We need to introduce a new type: **type scheme**. Basically, we want to store the concept that `f` is type `'a -> 'a`, but also be fully aware that `'a` can take in any value without having to create a new constraint about it.

The notation for a type scheme looks like: `label.type`. It says that all labels (where a label is something like `'a,'b,'c,etc`) will be used in the following type.

For instance, the function we just talked about would be stored like `'a.'a->'a`. When we apply it to an int, we can briefly consider the label `'a` to be a totally new label `'b` for this application, so we have a temporary type for the function `'b -> 'b`, and we can create the constraint only from $\{'b = int\}$, leaving the original `'a` alone.

Formally, a type scheme says that a polymorphic type works for all values. Sometimes, it is written as $\forall a.t$. For all possible values of the labels a, this thing has type t.

When I first learned this, I found it confusing. Here is how I think about it. `fun x -> x` should have the type `'a -> 'a`. This should be true for all input values. So we can say, for all inputs of type `'a`, `fun x -> x` will take in types of `'a` and return something of type `'a`. Thus, we can say `fun x -> x` has type `'a.'a->'a`. Then, when we instantiate a particular call to `fun x -> x` (like in the case of an application), we can fork off a new copy of this function which will have type `'b -> 'b` where `'b` is a type not used before.

One thing to note is that this is used only in let bindings in the form of `let x =` e_1 `in` e_2. We find the type of e_1, and if it is a function, we calculate e_2 using the idea of the scheme. However, the type of e_1 itself is still just a regular arrow type, not scheme. This new type scheme will never be returned, it's just used in the algorithm.

Let's see an example: To begin, let's first include some rules we need. We are going to be using a new helper function, $make(t_x)$, which will create a "totally new" polymorphic type that has never been used before.

$$(var-lookup)\frac{G(x) = t}{G \vdash e : t, \{\}} \qquad (fun)\frac{make(t_x) \quad G, x : t_x \vdash e : t, C_1}{G \vdash fun\ x \rightarrow e : t_x \rightarrow t, C_1}$$

$$(app)\frac{make(t_x) \quad G \vdash e_1 : t_1, C_1 \quad G \vdash e_2 : t_2, C_2}{G \vdash e_1\ e_2 : t_x, \{t_1 = t_2 \rightarrow t_x\} \cup C_1 \cup C_2}$$

Then let's prove **`let id = fun x -> x in if id true then id false else id true`**. I will only show part of this since I only want to highlight this idea of the type scheme. I also will not include constraints here.

$$\frac{\dfrac{make(t_1) \quad \dfrac{G, x : t_1(x) = t_1}{G, x : t_1 \vdash x : t_1}}{G \vdash fun\ x \rightarrow x :' a \rightarrow' a} \qquad \dfrac{\dfrac{make('z) \quad \dfrac{G, id :' a.'a \rightarrow' a(id) = new\_version}{G, id :' a.'a \rightarrow' a \vdash id :' b \rightarrow' b}}{G, id :' a.'a \rightarrow' a \vdash id\ true : bool} \quad \ldots}{G, id :' a\ .'a \rightarrow' a \vdash \text{if } id\ true \text{ then } false \text{ else } true : bool, C : t_2}}{G \vdash \text{let } id = fun\ x \rightarrow x \text{ in if } id\ true \text{ then } false \text{ else } true : bool, C}$$

There are a few things here:

- `make()` will make a new type, useful when making a new 'a or something
- `new_version` will fork the version of the type scheme to a new typed function. This only occurs when var-lookup returns a Type Scheme
- Notice that when we put `id` in the environment, we make it a type scheme. This basically flags the id lookup rule to make a new version when its used