Hindley-Milner Type Inference

CSE 340 FALL 2021
Rida Bazzi

Polymorphism with Type parameters: Functions

In C, one can have the following declarations

```
int max (int x, int y)
{
    if x > y return x else return y;
}

float max (float x, float y)
{
    if x > y return x else return y;
}
```

The two declarations are essentially the same, differing only in the types of the parameters.

In C++, we can merge the two declaration in one **template**

```
template <class T>
T maxt( T x, T y)                    // I call it maxt to distinguish it
{                                    // from the max function in C++
     if (x > y)
        return x;
     else
        return y;
}
```

In this declaration, we are providing a type parameter, so we can call

```
maxt<int>(3,5)

maxt<float>(3.5,4.5)
```

We can also omit the type parameter and instead call

```
maxt(3,5)  to compare two integers or
maxt("abc","def") to compare two strings
```

The compiler figures out how to instantiate the type <T> based on the argument types.

But if we try to call

```
maxt(3,5.5)
```

the compiler is not able to determine a type T to match the template. In this case we will need to call

```
maxt<float>(3,5.5);
```

Polymorphism with Type parameters: Data Types

We can also declare polymorphic data types by specifying type parameters for the data type that is being declared (struct or class). Here is an example

```
#include <iostream>
#include <string>
#include <typeinfo>

using namespace std;

template <class T1 , class T2>
class mypair {
        T1 first;   // first and second have types T1 and T2
        T2 second;  // respectively. T1 and T2 can be different
public:
        mypair(T1 f, T2 s)
        {
          first = f;
          second = s;
        }

        void mprint() // typeid() prints information about the types
        { cout << "(" << first << " " << typeid(first).name() << ","
                << second << " " << typeid(second).name() << ")" << endl;
        }
};


int main()
{
  int i;
  mypair<int,float> p1 (1,1.2);
  mypair<float,int> p2 (1.2,1);
  mypair<int *,int> p3 (&i,1);

  p1.mprint();
  p2.mprint();
  p3.mprint();
}
```

When I execute this code in my environment, I get

```
(1 i,1.2 f)
(1.2 f,1 i)
(0x7fff5e344bcc Pi,1 i)
```

Note that in the output, i, f, and Pi stand for integer, float and pointer to int respectively.

Type Inference with auto in C++

Since C++11, the auto keyword can be used to deduce the type of a declared variable at initialization.

For example, we can declare

```
auto a = 1+2;          // a is int
auto b = a + 1.0;      // b is float
```

Support for auto in new contexts has been provided in C++14

Consider the following code

```
struct rectangle {
    int x;
    int y;

    rectangle(int a, int b) {
           x = a; y = b;
    }
};

template <class T> auto mult(T x, T y) {
    return x*y;
}
int operator*(const rectangle& n1, const rectangle& n2) {
    return n1.x*n1.y*n2.x*n2.y;
}
bool operator>(const rectangle& n1, const rectangle& n2) {
    return n1.x*n1.y > n2.x*n2.y;
}
int main() {
     rectangle r1 = rectangle(5,7), r2 = rectangle(3,10);
     cout << mult<int>(3,9) << "\n";
     cout << mult<rectangle>(r1,r2) << "\n";
}
```

The return type of mult() is not explicitly specified. It is inferred by the compiler from the particular use. If the arguments are float, the return type is float. If the arguments are rectangles, the return type is int.

Implicit Polymorphism

Some language, such as Haskell and OCaml, infer the most general polymorphic types for declarations.

I will be giving examples from OCaml

Note. You can try these examples yourself by using an online editor such as

https://try.ocamlpro.com/

The command line lets you evaluate expressions and introduce associations between names and expressions

We start with the following expression

```
# b = 1;;
Line 1, characters 0-1:
Error: Unbound value b
```

in OCaml = is the comparison operator and `b = 1` is a `bool` (boolean) expression. Since the name b has not been introduced, the compiler complains that the value `b` is unbound (not associated with a value).

Next we introduce `b`

```
# let b = 1;;
val b : int = 1
```

`let` is used to introduce an association between b and the value 1. You can understand it to mean make `b = 1` evaluate to true.

Notice how the compiler deduces that `b` is int because it is assigned the `int` value 1

Functions in OCaml

So far, we have only introduced `b` which has the value `1`

Now if we try to evaluate the expression **b = 1** we get true

```
# b = 1;;
- : bool = true
```

Notice that the type of the expression is given as **bool** (boolean) and the value is true

If we evaluate **b = 2**, we should get false

```
# b = 2;;
- : bool = false
```

Before introducing functions, we note that in OCaml function application is done by writing the function next to its argument as follows: **f x**. **f x** is **f** applied to **x**, or a call to **f** with argument **x**.

The following definition defines a function which, given an argument **x**, returns **x**. Note that there is no type specified for **x**. This function is <u>**polymorphic**</u>. It can take a parameter **x** of any type and returns **x**.

```
# let f x = x;;
val f : 'a -> 'a = <fun>
```

That is why the type of f is specified **'a -> 'a** which is a function that takes an argument of type **'a** and returns a result of type **'a**. The name **'a** is used to denote an unconstrained type that can be any type.

The following is a definition of a function that takes an argument of any type and returns an integer.

```
# let g x = 1;;
val g : 'a -> int = <fun>
```

We can evaluate the result of applying **g** to **f** and we get **1**:

```
# g f;;
- : int = 1
```

Functions and Operators

In OCaml, the operator + is only used to add integers. So, if we define

```
# let sum i j = i +j;;
val sum : int -> int -> int = <fun>
```

we are defining a higher order function `sum` that takes two parameters `i` and `j` and returns `i + j`. Since `i` and `j` are added using the + operator for integers, we conclude that `i` and `j` must be integers for the declaration to be correctly typed.

Let us examine `sum`. Application in OCaml is left associative as in lambda calculus. So `sum i j` is the same as `(sum i) j` . Given an integer value `v` of `i`, `(sum i)` is a function that takes an argument `j` and returns `v + j` where `v` is the value of `i`.

So, sum is a function that takes one integer argument and returns a function that takes one integer argument and returns an integer

That is why the type of `sum` is `int -> int -> int`

We call `sum` as follows

```
# sum 3 4;;
- : int = 7
```

The following form emphasizes that sum is indeed a function of one parameter that returns a function of one parameter that returns an integer

```
# (sum 3) 6;;
- : int = 9
```

Let us examine it

$$\underbrace{(\texttt{sum}\ 3)}_{\text{function}}\ \underbrace{6}_{\text{argument}}$$

`(sum 3)` is a function of `int` that returns `int`, so `sum` is a function of `int` (3 in the example) and the result is `(sum 3)` which is a function from `int` to `int`

Functions in OCaml

We can define a new function sum5 which is the result of applying sum to 5

```
# let sum5 = sum 5;;
val sum5 : int -> int = <fun>
```

Note that sum5 has type int -> int

We can use sum5 as a function:

```
# sum5 4;;
- : int = 9
```

More traditional functions

We can define a function with more "traditional" arguments, using tuples as follows

```
# let sum_tuple (i,j) = i+j;;
val sum_tuple : int * int -> int = <fun>
```

sum_tuple is a function of one parameter which is a pair. The pair has type int * int which is the cartesian product of int and int

We call sum_tuple as follows

```
# sum_tuple (3,4);;
- : int = 7
```

We cannot call sum_tuple as follows

```
# sum_tuple 3 4;;
Line 1, characters 0-9:
Error: This function has type int * int -> int
It is applied to too many arguments; maybe you forgot a `;'.
```

and we cannot call sum as follows

```
# sum (3,4);;
Line 1, characters 4-9:
Error: This expression has type 'a * 'b but an expression was expected of type int
```

Type Inference Constraints

Notice that in all the definitions we had so far, we did not declare the type of any of the variables, functions, or arguments. The types are inferred from the usage. This is like auto of C++ on steroids!

In what follows we give more complex examples.

Before giving the examples, we list some of the constraints that we can deduce from some common usages

1. **Integer constants** ... , -4, -3, -2, -1, 0, 1, 2, 3, 4, ...

   v : int

   An integer constant has type **int**. Anywhere an integer constant appears, we should assign it the type int

   Example

```
# 3;;
- : int = 3
```

2. **Floating point constants** 1.2 , 3.7 , 0.2 , ...

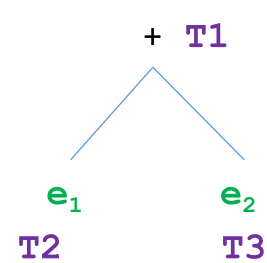
   v : float

   A floating point constant has type **float**

   Example

```
# 3.3 ;;
- : float = 3.3
```

3. **Integer addition** $e_1 + e_2$



   Constraint T1 = T2 = T3 = int

   if two expressions e_1 and e_2 are added together with the + operator , the two expressions must be of type int and the result is of type int

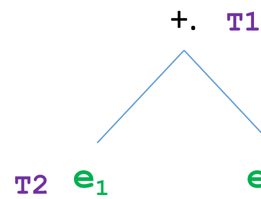
   Example

```
# let sum i j = i +j;;
val sum : int -> int -> int = <fun>
```

More Type Inference Constraints

4. Floating point addition e_1 +. e_2 **(notice the . after the +)**

Constraint T1 = T2 = T3 = float



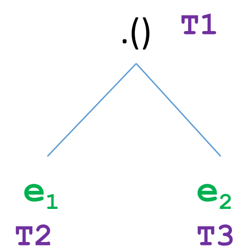
if two expressions e_1 and e_2 are added together with the +. operator , the two expressions must be of type float and the result is of type float

Example

```
# let sum_f i j = i +. j;;
val sum_f : float -> float -> float = <fun>
```

5. Array access e_1.(e_2)

Constraints T2 = T1 array
T3 = int



If the expression e_1.(e_2) is used, then e_1 must have type T array where T is the element type and e_2 must be an integer. The type of the expression e_1.(e_2) is T

Example

```
# let f x i = x.(i);;
val f : 'a array -> int -> 'a = <fun>
```

T_x T_i

Note how the type of the first parameter x is T_x = 'a array and the type of the second parameter i is T_i = int

More Type Inference Constraints

6. <u>Function application</u> `f x₁ x₂ ... xₖ`

x_1 : T_1

x_2 : T_2

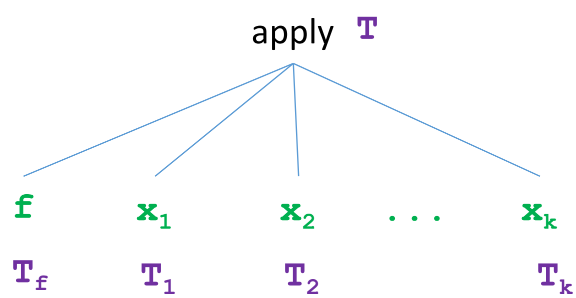
...

x_k : T_k

f : T_f

$f\ x_1\ x_2\ \ldots\ x_k$: T

If the epxression $f\ x_1\ x_2\ \ldots\ x_k$ is used, then f must be a function. If the types of x_1 , x_2 , ... , x_k are T_1 , T_2 , ..., **and** T_k respectively, then the type of f is T_1 -> T_2 -> T_3 -> ... -> T_k -> T where T is the return type of f



constraint T_f = T_1 -> T_2 -> T_3 -> ... -> T_k -> T

<u>Example</u>

```
# let f x y = x (y + 1);;
val f : (int -> 'a) -> int -> 'a = <fun>
```

In this definition, f is a higher order function that take a parameter x and returns a function that takes a parameter y. Note that the form of the definition of f is essentially saying that f x y = (f x) y is the expression x (y +1). Let us examine the expression x (y +1):

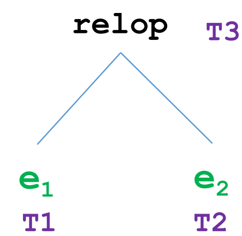
1. The value 1 is added to y, so the type of y must be int
2. x is applied to (y +1), so x must be a function that takes an int parameter and returns a value of some type T .
3. The type of f x y is the same as the return type of x because the value f x y is the value returned by x when it is applied to (y +1). In the OCaml editor, the type T is denoted 'a .

More Type Inference Constraints

7. <u>Relational operators</u>

e : e_1 relop e_2

relop : < | > | =



constraint T1 = T2 T3 = bool

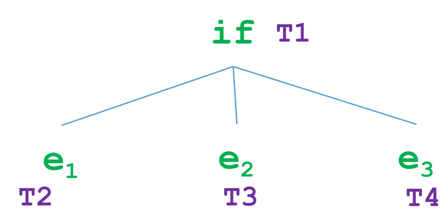
If the expression **e** is used and has the form e_1 **relop** e_2 , then the expressions e_1 and e_2 must have the same type **T** and the expression **e** has type **bool** (boolean).

Example

```
# let f x y = x < y;;
val f : 'a -> 'a -> bool = <fun>
```

Note. relop can apply to two operands that have the same type. I said in class that it can also be used to compare functions but that is no longer valid now. It was valid when I prepared the original notes (this is based on command line not specs).

8. <u>If expression</u> **if e_1 then e_2 else e_3**



constraints T1 = T3 = T4
T2 = bool

If the expression **if e_1 then e_2 else e_3** is used, then e_1 must have type **bool** and e_2 and e_3 must have the same type **T** which is also the type of the if expression

Example

```
# let f x y = if x then 1 else y;;
val f : bool -> int -> int = <fun>
```

In this example **x** is used as the condition, so its type should be **bool.** The true branch has type of integer constant 1, so the true branch has type **int**. It follows that the false branch and the whole if-expression must have type **int:** T_y = **int** and the type of the if-expression is also **int,** which is also the type of the return value of function **f**.

More Type Inference Constraints

9. **Function definition** let $\mathtt{f\ x_1\ x_2\ ...\ x_k\ =\ e}$

Here we are saying that $\mathtt{f\ x_1\ x_2\ ...\ x_k}$ evaluates to $\mathtt{e}$. So, $\mathtt{f}$ must be a function that takes an argument $\mathtt{x_1}$ and returns a function that takes an argument $\mathtt{x_2}$ and returns a function ..... that returns the expression $\mathtt{e}$.

If $\mathtt{e}$ has type $\mathtt{T}$ and $\mathtt{x_1\ x_2\ ...\ x_k}$ have types $\mathtt{T_1}$, $\mathtt{T_2}$, $\mathtt{T_3}$, ... , $\mathtt{T_k}$ respectively, then $\mathtt{f}$ is a function whose type is

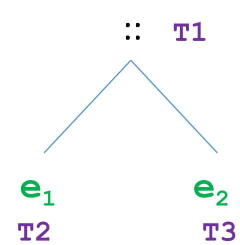
constraint $\mathtt{T_f = T_1 \text{ -> } T_2 \text{ -> } T_3 \text{ -> } \ ...\ \text{ -> } T_k \text{ -> } T}$

Note that the type of $\mathtt{f\ x_1\ x_2\ ...\ x_k}$ is the same as the type of $\mathtt{e}$

10. **Empty List** `[]`

The empty list has type `T List` where `T` is unconstrained.

11. **Prepend Operator** $\mathtt{e_1 :: e_2}$

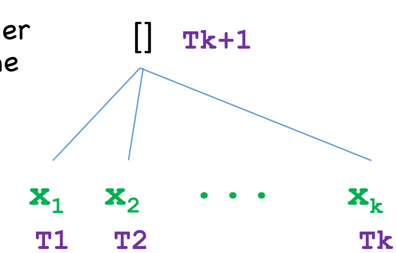


constraints `T1 = T3 = T2 List`

If the operator is applied to $\mathtt{e_1}$ and $\mathtt{e_2}$, then $\mathtt{e_2}$ must be a list and $\mathtt{e_1}$ must have the same type as the type of the elements of $\mathtt{e_2}$ and the result is a list of the same type as $\mathtt{e_2}$.

12. **List of many elements** $\mathtt{[\ x_1 ;\ x_2\ ;\ ...\ ;\ x_k\ ]}$

If the elements $\mathtt{x_1 ;\ x_2\ ;\ ...\ ;\ x_k}$ are put together in a list, they should all have the same type `T` and the type of the resulting list is `T List`.



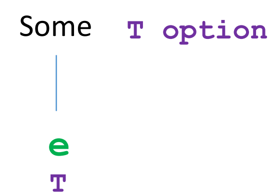
constraints `T1 = T3 = ... = Tk = T`
`Tk+1 = T List`

More Type Inference Constraints

12. Option Type `Some` and `None`

For every type `T`, Ocaml has an `T option` type whose set of values is the same as the set of values of `T` plus the additional undefined value `None`. To distinguish between values from type `T` and values from type `T option`, `Some` is used. For example

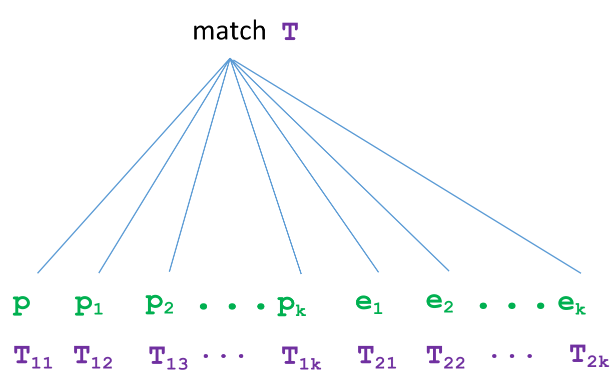
1	`int`
`Some 1`	`int option`
`Some e`	`T option`, where `T` is the type of `e`
`None`	`T option`, where `T` is unconstrained



13. Pattern Matching `match with`

The `match` expression of Ocaml can is a very powerful generalization of C switch statement (which matches a integer value to integer constants). The syntax of the `match` expression is

```
match p with
   p1 -> e1
 | p2 -> e2
 | p3 -> e3
   ...
 | pk -> ek
```



The type constraints for this expression are

constraints

$$T_{11} = T_{12} = \ldots = T_{1k}$$

$$T_{21} = T_{22} = \ldots T_{2k} = T$$

Dynamic Semantics and Examples

Dynamic Semantics. The dynamic semantics of pattern matching tell us how to execute pattern matching. It is basically a sequence of if then else

```
     if p matches p1 then e1
else if p matches p2 then e2
else if p matches p3 then e3
else …
else if p matches pk then ek
```

Example 1

```
let f a b = match (a,b) with
              (true,true) -> true
            | _           -> false
```

In this example we are matching (a,b) which is a pair with the pair (true,true) in which case the expression evaluates to true

The alternative, is a wildcard match in which case the answer is false.

The function calculates the Boolean and function. Notice that the wild card need not be expressed as (_,_) because its type is deduced from the type of (true,true).

Example 2

```
let fl = match l with
              []          -> false
            | _::[]       -> false
            | _::_::[]    -> false
            | _           -> true
```

This function determines if the input l is a list of more than two elements. The first pattern [] is the empty list. The second pattern _::[] is a list of one element and the third pattern _::_::[] is a list of two elements. Finally, the wild card is matched if none of the other patterns are matched.

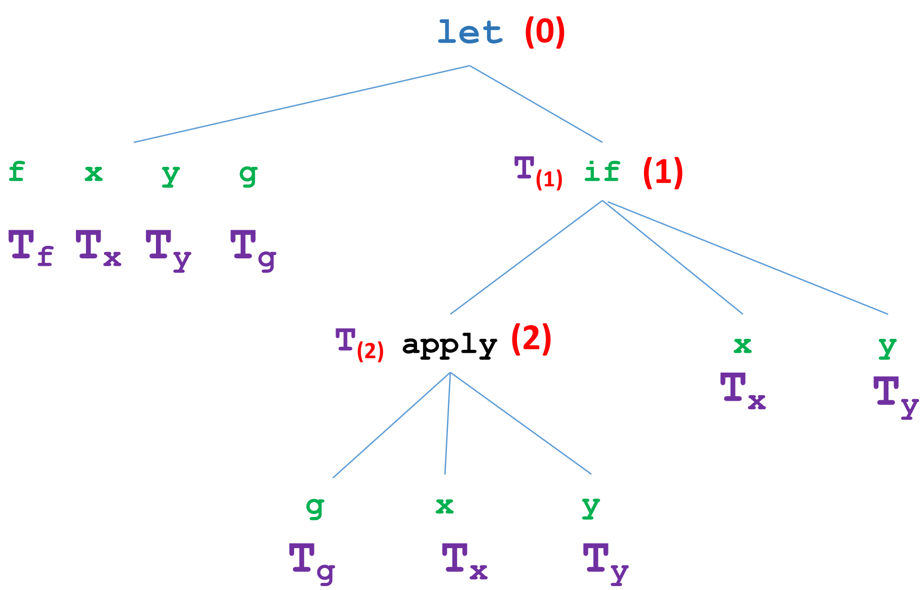
Hindley-Milner Type Checking

Here is the outline of Hindley-Milner type checking. The examples should make it clearer.

1. Build an abstract syntax tree (AST). An abstract syntax tree is similar to a parse tree but omits unnecessary syntax. For example, the AST for if expression need not show **then** and **else**.
2. Associate with each named variable a type
3. Associate with each node of the AST a type
4. Visit the nodes of the AST in any order
    1. for each visited node, apply the constraint of that node to get a relationship between the type of the node and the types of its children
    2. If a type is expressed in two different ways (is given as two expressions), you should unify the two expressions. This means that you should find the type for which the two expressions are equal. This is not unlike solving equations in algebra. For example, if we know that x = y - 2 and then later we determine that x = 2, then we know that y must be 4.

Another Example

```
# let f x y g = if g x y then x else y;;
```



Even though it is relatively straightforward to determine the types for this example, I will do it systematically, by visiting every node in the abstract syntax tree (AST) of the function definition and apply the constraint corresponding to the node.

Visit node (0): We apply the constraint for a "let" node and we conclude that
$T_f = T_x \to T_y \to T_g \to T_{(1)}$

Visit node (1): Node (1) is an if node. The constraint for an "if" node requires that the branch corresponding to the condition is bool and the other two branches have the same type as the if node itself. We get:
$T_{(1)} = T_x = T_y = T$ and $T_{(2)} = \text{bool}$

Visit node (2): Node (2) is an apply node. This means that g must be a function of x and and y that returns bool (the type of the "apply" node). We get:
$T_g = T \to T \to \text{bool}$

After visiting all the nodes, we have the answer

$T_x = T$

$T_y = T$

$T_g = T \to T \to \text{bool}$

$T_f = T \to T \to (T \to T \to \text{bool}) \to T$

```
val f : 'a -> 'a -> ('a -> 'a -> bool) -> 'a = <fun>
```

Note that I did not explicitly visit the leaf nodes. In this example, each leaf node just has the type of the named variable.

More Examples without step by step derivation

Example 1

```
# let f a i = a.(i) + i;;
```

Here f is a higher order function that takes a and i as parameters. Since a is used as an array in the expression a.(i), we conclude that a is an array of T , where T is the element type, and i is int. Since a.(i), is added to i we conclude that the element type is also int. The type of f is therefore the following:

```
val f : int array -> int -> int = <fun>
```

Example 2

```
# let max x y = if x > y then x else y;;
val max : 'a -> 'a -> 'a = <fun>
# max 1 2;;
- : int = 2
# max 1.0 2.2;;
- : float = 2.2
# max "abc" "def";;
- : string = "def"
# max 1.2 3;;
Line 1, characters 8-9:
Error: This expression has type int but an expression was expected of type float
  Hint: Did you mean `3.'?
# let f1 x = 1;;
val f1 : 'a -> int = <fun>
# let f2 x = 2.3;;
val f2 : 'a -> float = <fun>
# max f1 f2;;
Line 1, characters 7-9:
Error: This expression has type 'a -> float but an expression was expected of type
  'a -> int
Type float is not compatible with type int
```

Here max is a function that takes two parameters of the same type and returns the larger of the two according to a comparison operator >.

The example shows how max can be used with arguments of different types as long as the two arguments have the same type

More Examples without step by step derivation

Example 3

```
# let f a b = a b;;
val f : ('a -> 'b) -> 'a -> 'b = <fun>
# let a1 x = x+1;;
val a1 : int -> int = <fun>
# let a2 x = x+.1.0;;
val a2 : float -> float = <fun>
# f a1 2;;
- : int = 3
# f a2 2.3;;
- : float = 3.3
# f a1 1.0;;
Line 1, characters 5-8:
Error: This expression has type float but an expression was expected of type int
```

Here `f` is a higher order function that takes two parameters and applies the first parameter `a` to the second parameter `b` and returns the result of the application of `a` to `b`. Other than the fact that `a` must be a function and that the type of `b` must match the type of the argument of `a`, there are no other constraints on the types of `a` and `b`.

The example shows how `f` can be used with arguments of different types.

1. `a1` is a function that take an `int` argument `x` and returns `x + 1`
2. `a2` is a function that takes a `float` argument x and returns `x +. 1.0`
3. `f` can be applied to `a1` and an `int` value
4. `f` can be applied to `a2` and a `float` value
5. `f` cannot be applied to `a1` and a `float` value

More Examples without step by step derivation

Example 4

```
# let f1 x = x;;
val f1 : 'a -> 'a = <fun>
# let f2 x y = 1;;
val f2 : 'a -> 'b -> int = <fun>
# max f1 f2;;
- : ('_a -> int) -> '_a -> int = <fun>
#
```

In this example, **max** is the function from example 3. This example might be confusing at first sight. Here **f1** is function that takes **x** as argument and returns **x**. Since **x** is not constraint, the time of **f1** is **T -> T** where **T** is the type of **x**. **f2** is a higher order function that takes **x** as an argument and returns a function that takes **y** as argument and returns 1, so the type of **f2** is T_x **->** T_y **-> int** , where T_x is the type of **x** and T_y is the type of **y**.

What might be confusing is how can we compare **f1** and **f2** in the function **max**. In the context in which we are comparing the two functions in **max**, the two functions must have the same type. So, we need the following two type to be the same

1. **T -> T**
2. T_x **->** T_y **-> int**

For these two types to be equal, we need

1. The argument types to be the same: $T = T_x$
2. The returns types to be the same: $T = T_y$ **-> int**

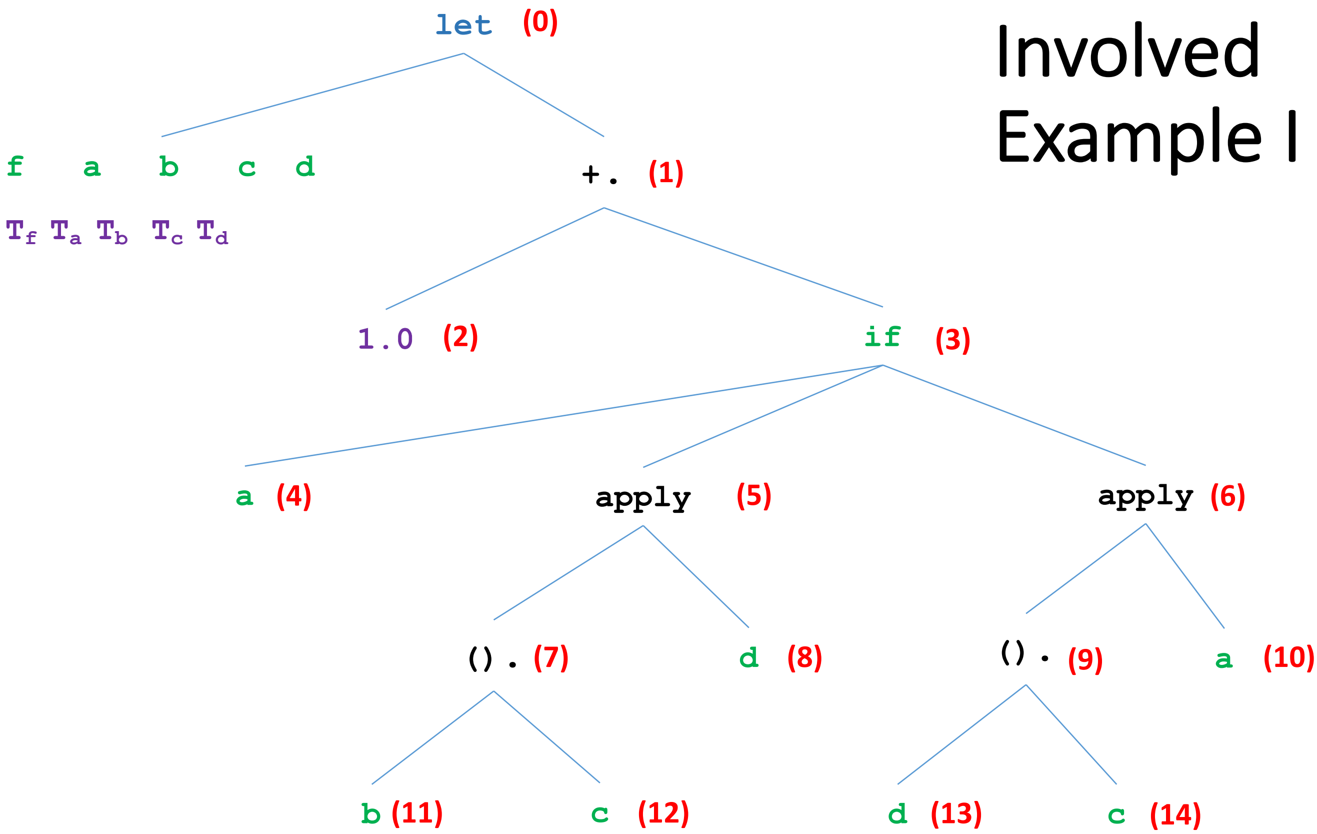
Both conditions can be satisfied if $T = T_x = T_y$ **-> int**

and max f1 f2 will have type (T_y **-> int**) **->** T_y **-> int**

Note This example is given only to illustrate how arguments with polymorphic types that are passed to a function can be further restricted based on how they are used in the function

```
let f a b c d = 1.0 +. if a then b.(c) d else d.(c) a
```

More Involved Example I



We will visit each node of the AST and apply the constraint of the node and unify types as needed

Visit node (0): We apply constraint of "let" node: $T_f = T_a \to T_b \to T_c \to T_d \to T_{(1)}$

Visit node (1): We apply constraint of "+." node: $T_{(1)} = T_{(2)} = T_{(3)} =$ `float`

Visit node (2): $T_{(2)} =$ `float` = type of `1.0` (consistent with value of $T_{(2)}$ determined above)

Visit node (3): We apply the constraint of "if" node:
$T_{(4)} =$ `bool` and $T_{(5)} = T_{(6)} = T_{(3)} =$ `float` (determined above)

Visit node (4): $\boxed{T_a = T_{(4)} = \texttt{bool}}$

Visit node (8): $T_{(8)} = T_d$

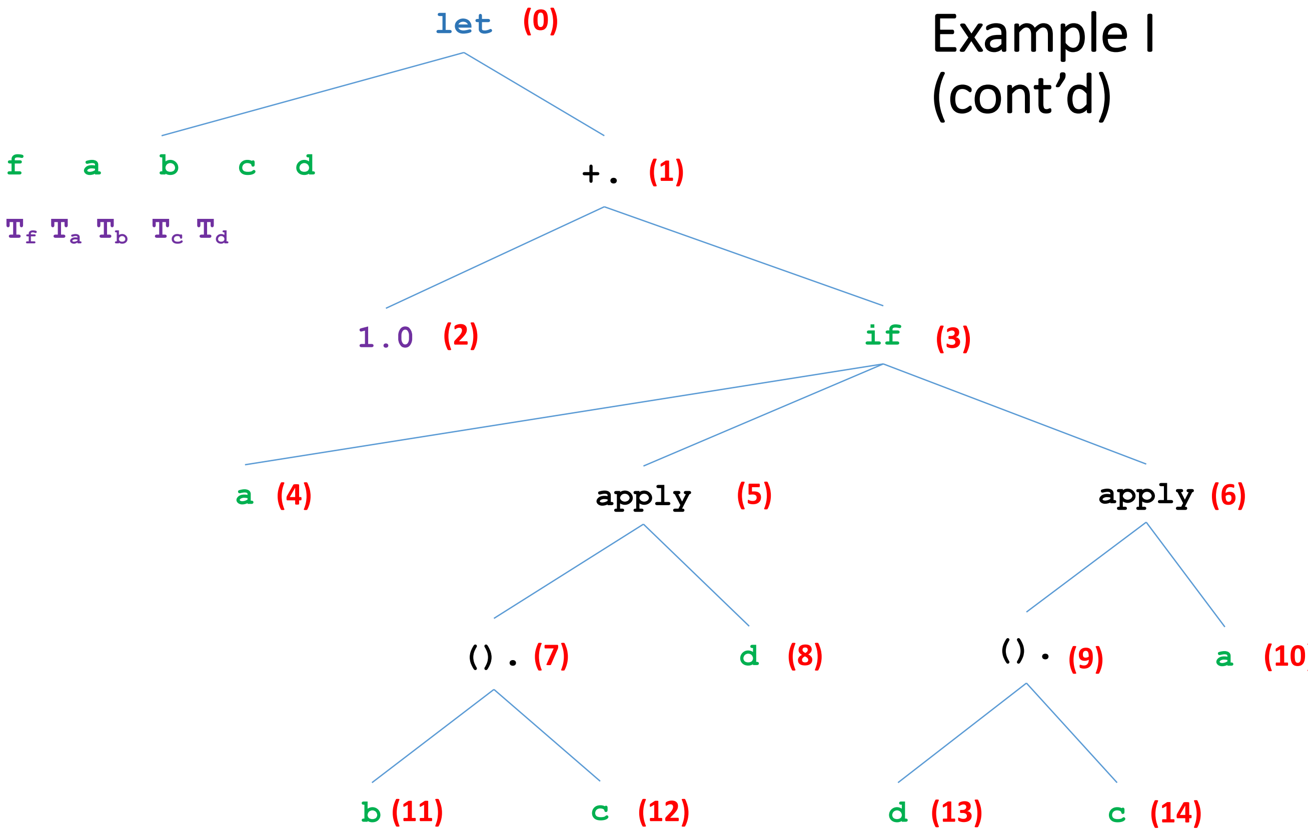
Visit node (10): $T_{(10)} = T_a =$ `bool`

Visit node (11): $T_{(11)} = T_b$

Visit node (12): $T_{(12)} = T_c$

Visit node (5): Node (5) is an "apply" node, we apply the constraint for "apply" node:
$T_{(7)} = T_{(8)} \to T_{(5)} = T_{(8)} \to$ `float` ($T_{(5)}$ was determined to `float` be above)

More Involved Example I (cont'd)



We continue from the previous page.

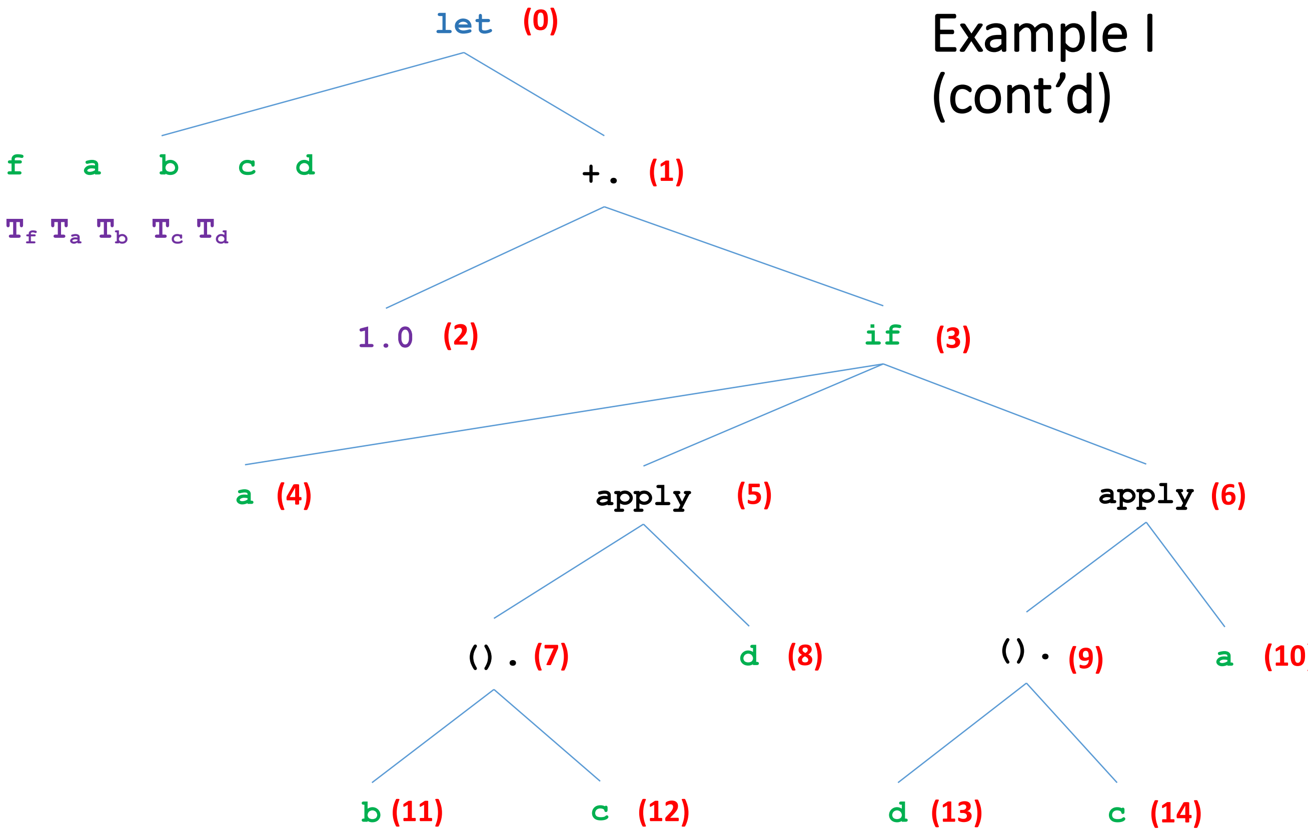
Visit node (7): Node (7) is an ".()" node (array access). Applying the constraint for array access, we get: $T_{(11)} = T_{(7)}$ array = ($T_{(8)}$ -> float) array
$T_{(12)}$ = int,

but $T_{(12)} = T_c$, $T_{(8)} = T_d$ and $T_{(11)} = T_b$ so, we get

$$\boxed{\begin{array}{l} T_b = (T_d \texttt{ -> float) array} \\ T_c = \texttt{int} \end{array}}$$

Visit node (6): Node (6) is an "apply" node, we apply the constraint for "apply" node:
$T_{(9)} = T_{(10)}$ -> $T_{(6)}$ = bool -> float ($T_{(6)}$ was determined to be float above)

More Involved Example I (cont'd)



Visit node (9): Node (9) is an ".()" node (array access). Applying the constraint for array access, we get: $T_{(13)} = T_{(9)}$ array = (bool -> float) array
$T_{(14)}$ = int,

but $T_{(13)} = T_d$, $T_{(14)} = T_c$ so, we get

T_c = int (consistent with previous determination)

T_d = (bool -> float) array

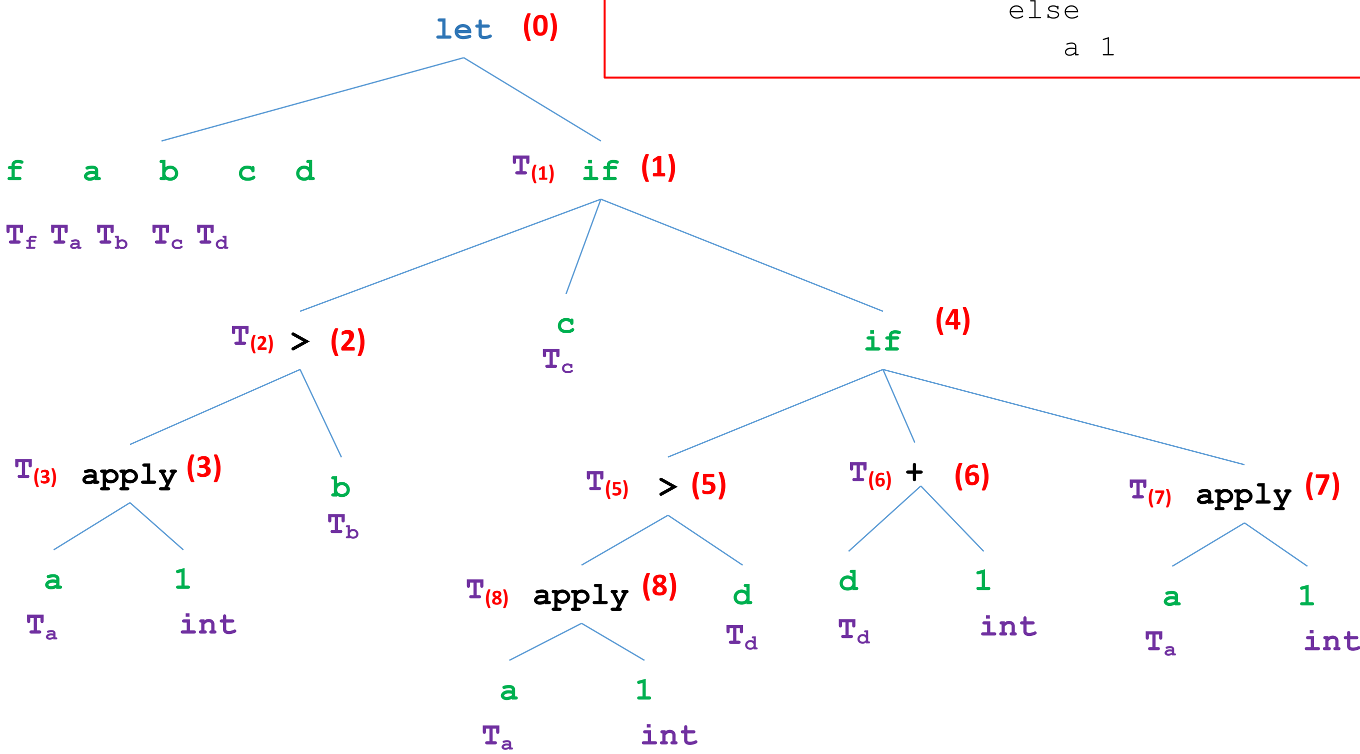
We conclude that T_b = (((bool -> float) array) -> float) array

To summarize

- T_a = bool
- T_b = (((bool -> float) array) -> float) array
- T_c = int
- T_d = (bool -> float) array

More Involved Example II

```
let f a b c d = if a 1 > b then
                    c
                else
                    if a 1 > d then
                        d + 1
                    else
                        a 1
```



We will visit each node of the AST and apply the constraint of the node and unify types as needed

Visit node (0): We apply constraint of "let" node: $T_f = T_a \to T_b \to T_c \to T_d \to T_{(1)}$

Visit node (1): We apply constraint of "if" node: $T_{(1)} = T_c = T_{(4)} = T$ and $T_{(2)} = \mathtt{bool}$

Visit node (2): We apply the constraint of "relational operator" node:
$T_{(3)} = T_b$ and $T_{(2)} = \mathtt{bool}$ (consistent with value of $T_{(2)}$ above)

Visit node (3): We apply the constraint of "apply" node:
$T_a = \mathtt{int} \to T_{(3)} = \mathtt{int} \to T_b$ (remember $T_{(3)} = T_b$)

Visit node (4): We apply constraint of "if" node:
$T_{(6)} = T_{(7)} = T_{(4)} = T$ and $T_{(5)} = \mathtt{bool}$

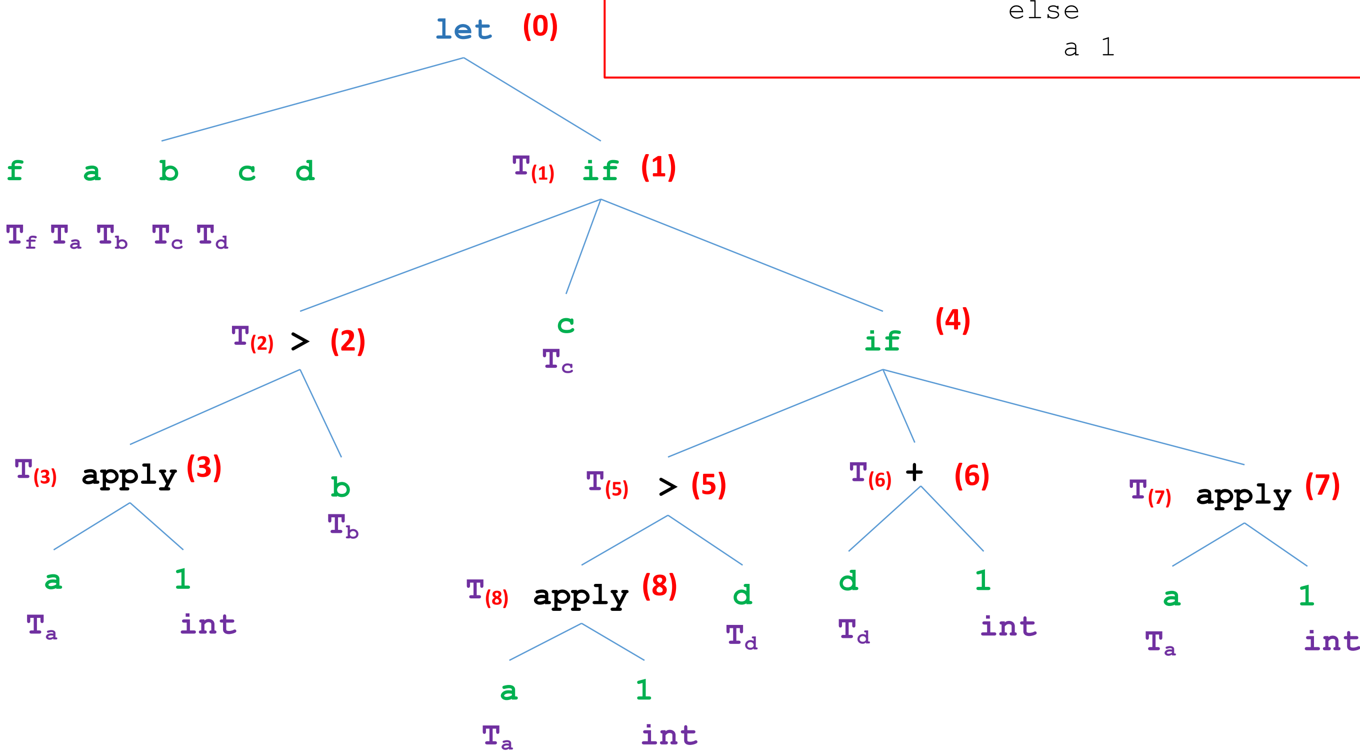
Visit node (5): We apply the constraint of "relational operator" node:
$T_{(8)} = T_d$ and $T_{(5)} = \mathtt{bool}$ (consistent with value of $T_{(5)}$ above)

Visit node (6): We apply the constraint of a "+" node:
$T_{(6)} = T_d = \mathtt{int}$ so we conclude that $T_c = \mathtt{int}$, $T_{(1)} = \mathtt{int}$,
$T_{(4)} = \mathtt{int}$, and $T_{(7)} = \mathtt{int}$

Visit node (7): We apply the constraint of "apply" node"
$T_a = \mathtt{int} \to T_{(7)}$ so $T_a = \mathtt{int} \to \mathtt{int}$ and $T_b = \mathtt{int}$

More Involved Example II

```
let f a b c d = if a 1 > b then
                   c
                else
                   if a 1 > d then
                      d + 1
                   else
                      a 1
```



Visit node (8): We apply the constraint of "apply" node"

T_a = int -> $T_{(8)}$, but we know that T_a = int -> int

we conclude that $T_{(8)}$ = int.

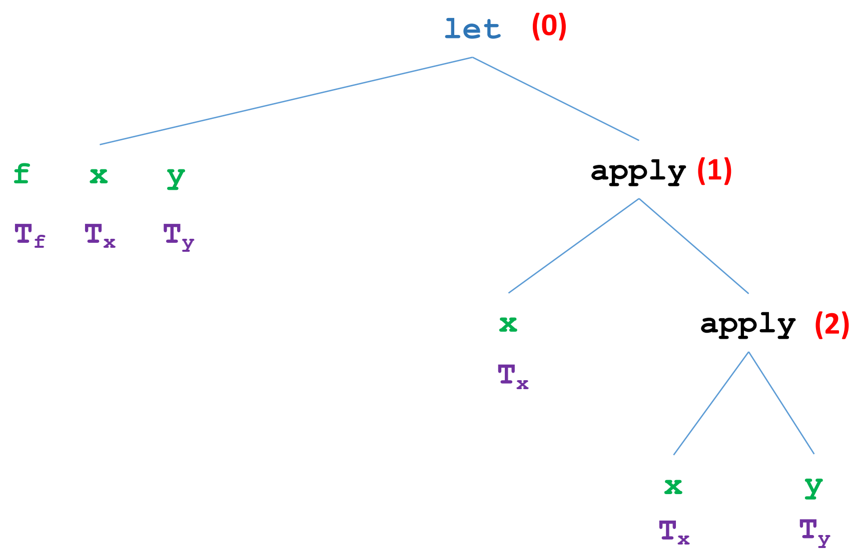
We also know that $T_{(8)}$ = T_d and we conclude that T_d = int

The types are T_a = int -> int , T_b = int , T_c = int , T_d = int

T_f = (int -> int) -> int -> int -> int -> int

Example III

```
let f x y = x ( x y )
```



We will visit each node of the AST and apply the constraint of the node and unify types as needed

Visit node (0): We apply constraint of "let" node: $T_f = T_x \to T_y \to T_{(1)}$

Visit node (1): We apply constraint of "apply" node: $T_x = T_{(2)} \to T_{(1)}$

Visit node (2): We apply constraint of "apply" node: $T_x = T_y \to T_{(2)}$

We have two expressions for T_x : $T_x = T_{(2)} \to T_{(1)}$

$T_x = T_y \to T_{(2)}$

The two expressions should be equal, so we should have $(T_{(2)} \to T_{(1)}) = (T_y \to T_{(2)})$
Both are functions, so we need to have identical argument types and identical return types:

- $T_{(2)} = T_y$ arguments have the same type
- $T_{(1)} = T_{(2)}$ return types are the same

We conclude that $T_{(1)} = T_{(2)} = T_y = T$ (some type T)

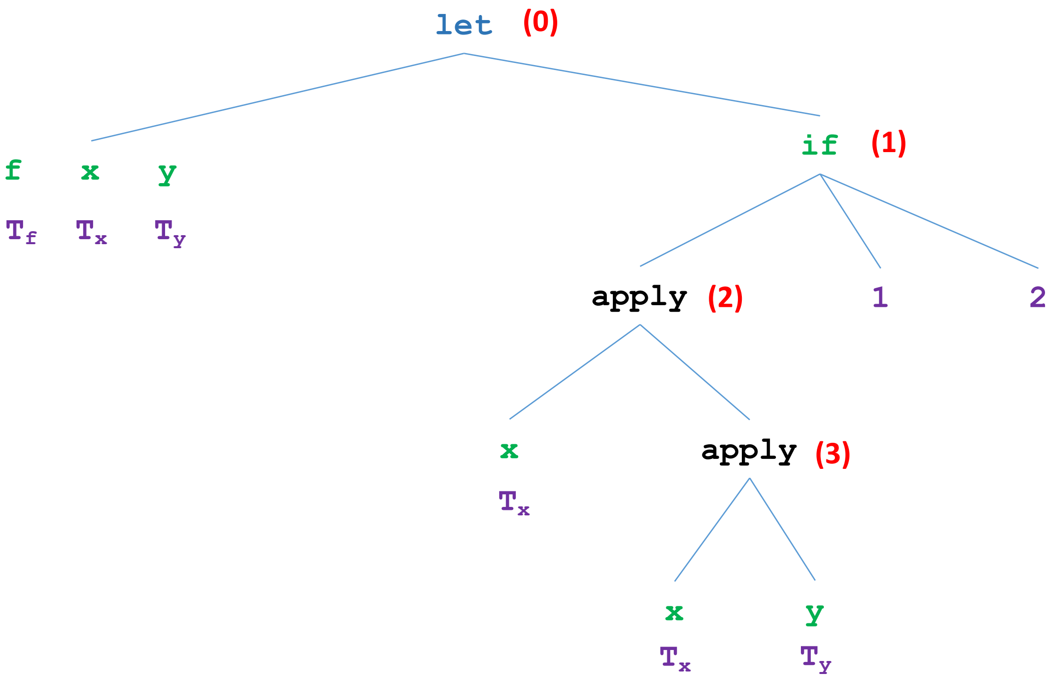
$T_x = T \to T$

$T_y = T$

$$T_f = \underbrace{(T \to T)}_{T_x} \to \underbrace{T}_{T_y} \to T$$

Example IV

```
let f x y = if  x ( x y ) then 1 else 2
```



We will visit each node of the AST and apply the constraint of the node and unify types as needed

Visit node (0): We apply constraint of "let" node: $T_f = T_x \to T_y \to T_{(1)}$

Visit node (1): We apply constraint of "if" node

$T_{(2)}$ = bool $T_{(1)}$ = type of 1 = type of 2 = int

Visit node (2): We apply constraint of "apply" node: $T_x = T_{(3)} \to \text{bool}$

Visit node (3): We apply constraint of "apply" node: $T_x = T_y \to T_{(3)}$

We have two expressions for T_x : $T_x = T_{(3)} \to \text{bool}$

$T_x = T_y \to T_{(3)}$

The two expressions should be equal, so we should have $(T_{(3)} \to \text{bool}) = (T_y \to T_{(3)})$
Both are functions, so we need to have identical argument types and identical return types:

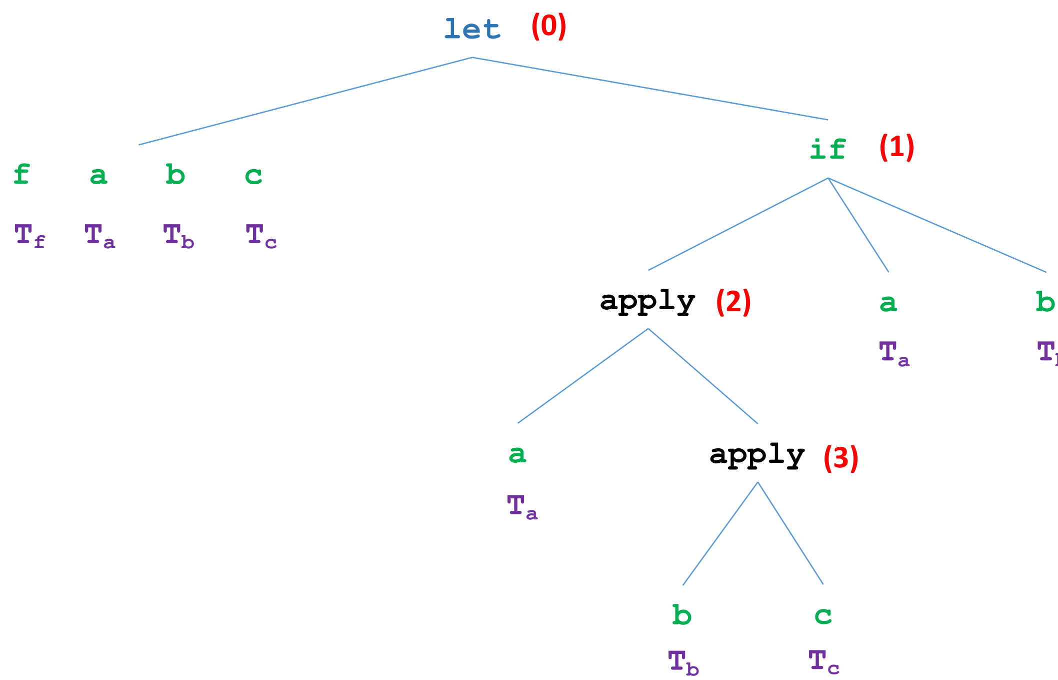
- $T_{(3)} = T_y$ arguments have the same type
- $\text{bool} = T_{(3)}$ return types are the same

We conclude that $T_{(3)} = T_y = \text{bool}$

$T_y = \text{bool}$
$T_x = \text{bool} \to \text{bool}$
$T_f = (\text{bool} \to \text{bool}) \to \text{bool} \to \text{int}$

Example V

```
let f a b c = if  a ( b c ) then a else b
```



We will visit each node of the AST and apply the constraint of the node and unify types as needed

Visit node (0): We apply constraint of "let" node: $T_f = T_a \to T_b \to T_{(1)}$

Visit node (1): We apply constraint of "if" node

$T_{(2)} = bool \quad T_{(1)} = T_a = T_b$

Visit node (2): We apply constraint of "apply" node: $T_a = T_{(3)} \to T_{(2)}$

$= T_{(3)} \to bool$

Visit node (3): We apply constraint of "apply" node: $T_b = T_c \to T_{(3)}$

but we know that $T_a = T_b$

so $\underbrace{T_{(3)} \to bool}_{T_a} = \underbrace{T_c \to T_{(3)}}_{T_b}$

to make the two expressions equal, we need $T_{(3)} = T_c$ and $bool = T_{(3)}$

so, we conclude

$T_a = T_b = bool \to bool$
$T_c = bool$
$T_f = \underbrace{(bool \to bool)}_{T_a} \to \underbrace{(bool \to bool)}_{T_b} \to \underbrace{bool}_{T_c} \to \underbrace{(bool \to bool)}_{T_{(1)}}$