COMP520 - GoLite Type Checking Specification

Vincent Foley

April 8, 2018

1 Introduction

This document presents the typing rules for all the language constructs (i.e. declarations, statements, expressions) of GoLite. The rules are specified in two ways:

1. In prose
2. As inference rules (notes on notation can be found in the next section)

If a rule seems unclear, you should use the reference compiler's `golitec typecheck` command to see what happens.

1.1 Notation for inference rules

The context for GoLite will consist of three components:

- V: the set of variables and functions in scope
- T: the set of types in scope
- F: the return type of the current function or the special symbol $\circ$ if not inside a function. The type name *void* will be used for functions with no declared return type. (Note that *void* is **not** a valid GoLite type!)

1.2 Types in GoLite

In true Go, there are 3 concepts for types: defined types, base types, and type aliases. In GoLite, we focus on the first 2.

1.2.1 Defined types

A defined type is given by a type declaration (`type num int`) which defines a *new* type for the context which:

- Inherits from the underlying type
- Is distinct from all other types (including the underlying type)

The function RT (short for "resolve type") finds the underlying type a defined type τ given the set of types in scope T. For compound types (e.g. slices or structs), it does not resolve the types of the inner components.

$$RT(T, \tau) = \begin{cases} RT(T, \tau') & \text{if } \tau \text{ is a defined type for } \tau' \\ \tau & \text{otherwise} \end{cases}$$

```
type num int
type natural num

RT(T, int) = int
RT(T, num) = int
RT(T, natural) = int

type floats []float64
type naturals []natural

RT(T, floats) = []float64
RT(T, naturals) = []natural
```

1.2.2 Type equality

For GoLite we use the same notion of type equality as Go

- https://golang.org/ref/spec#Type_identity

In particular, note that defined types are distinct regardless of identifier and underlying type. For two defined types to be identical, they must point to the same type specification.

1.2.3 Base types

The full Go implementation has numerous base types – in GoLite we only support `int`, `float64`, `bool`, `rune`, and `string`.

We also make an important simplification regarding the base types. In Go, the identifier "int" refers to a *defined* type which resolves to the base type `int`. In GoLite, we ignore this indirection and let the "int" identifier point directly to the base type.

2 Declarations

Declarations are the primary means of introducing new identifiers in the symbol table. In Go, top-level declarations can come in any order; in GoLite, we will require that identifiers be declared before they are used. This will prevent mututally recursive functions, however it should make the type checker implementation easier.

The symbol table should start with a few pre-declared mappings; the boolean identifiers and the base types. These identifiers can be shadowed.

Identifier	Category	Type
true	constant*	bool
false	constant*	bool
int	type	int
float64	type	float64
rune	type	rune
bool	type	bool
string	type	string

*Constants belong to the variable category (can be used in expressions), but cannot be assigned without first being shadowed.

2.1 Variable declarations

`var x T`

Adds the mapping `x:T` to the symbol table.

$$\frac{V \cup \{x : \tau\}, T, F \vdash rest}{V, T, F \vdash var\ x\ \tau; rest}$$

`var x T = expr`

If `expr` is well-typed and its type is `T1`, and `T1=T`, the mapping `x:T` is added to the symbol table.

$$\frac{V, T, F \vdash e : \tau_1 \quad \tau_1 = \tau \quad V \cup \{x : \tau\}, T, F \vdash rest}{V, T, F \vdash var\ x\ \tau = e; rest}$$

```
var x = expr
```

If `expr` is well-typed and its type is `T`, the mapping `x:T` is added to the symbol table.

$$\frac{V, T, F \vdash e : \tau \quad V \cup \{x : \tau\}, T, F \vdash rest}{V, T, F \vdash var\ x = e; rest}$$

In all three cases, if `x` is already declared in the current scope, an error is raised. If `x` is already declared, but in an outer scope, the new `x:T` mapping will *shadow* the previous mapping.

Note: In Go, it is an error to declare a local variable and not use it. In GoLite, we will allow unused variables. (If you wanted to comply with the Go specification, how would you make sure that all locals are used?)

2.2 Type declarations

```
type T1 T2
```

Adds the type mapping `T1 -> def(T2)` to the type symbol table (i.e., `T1` is a defined type inheriting from `T2`). If `T1` is already declared in the current scope, an error is raised. If `T1` is already declared, but in an outer scope, the new `T1 -> def(T2)` type mapping will *shadow* the previous mapping.

$$\frac{V, T \cup \{\tau_1 \to def(\tau_2)\}, F \vdash rest}{V, T, F \vdash type\ \tau_1\ \tau_2; rest}$$

2.3 Function declarations

```
func f(p1 T1, p2 T2, ..., pn Tn) Tr {
        // statements
}
```

Given the declaration for `f` above, the mapping `f:(T1 * T2 * ...  * Tn -> Tr)` is added to the symbol table. If `f` is already declared in the current scope (i.e. the global scope since we don't have nested functions), an error is raised.

For each formal parameter `pi`, the mapping `pi:Ti` is added to the symbol table. If two parameters have the same name, an error is raised. A formal parameter or a variable or type declared in the body of the function may have the same name as the function.

```
// Valid
func f(f int) {
        ...
}

// Invalid
func f(f int) {
        var f float64  // Redeclares f (the formal parameter)
        ...
}
```

A function declaration type checks if the statements of its body type check.

$$\frac{\begin{array}{c} V \cup \{f : \tau_1 \times ... \times \tau_k \to \tau, x_1 : \tau_1, ..., x_k : \tau_k\}, T, \tau \vdash body \\ V \cup \{f : \tau_1 \times ... \times \tau_k \to \tau\}, T, F \vdash rest \end{array}}{V, T, F \vdash func\ f(x_1\ \tau_1, ..., x_k\ \tau_k)\ \tau\ \{\ body\ \}; rest}$$

$$\frac{\begin{array}{c} V \cup \{f : \tau_1 \times ... \times \tau_k \to void, x_1 : \tau_1, ..., x_k : \tau_k\}, T, void \vdash body \\ V \cup \{f : \tau_1 \times ... \times \tau_k \to void\}, T, F \vdash rest \end{array}}{V, T, F \vdash func\ f(x_1\ \tau_1, ..., x_k\ \tau_k)\ \{\ body\ \}; rest}$$

Additionally, for functions that return a value, the statements list should end in a terminating statement (weeding pass).

- https://golang.org/ref/spec#Terminating_statements

2.4 Special functions

GoLite contains 2 special functions to conform with the official Go compiler: `init` and `main`.

For these special functions to type check they must follow the rules in the previous section but have no parameters or return type.

Additionally, `init` and `main` may only be declared as `functions` at the top-level scope. Note that `init` does not introduce a binding and thus may be declared multiple times.

3 Statements

Type checking of a statement involves making sure that all its children are well-typed. A statement does **not** have a type.

3.1 Empty statement

The empty statement is trivially well-typed.

$$\frac{V, T, F \vdash rest}{V, T, F \vdash \langle \text{empty} \rangle; rest}$$

3.2 `break` and `continue`

The `break` and `continue` statements are trivially well-typed.

$$\frac{V, T, F \vdash rest}{V, T, F \vdash break; rest} \quad \frac{V, T, F \vdash rest}{V, T, F \vdash continue; rest}$$

3.3 Expression statement

`expr`

An expression statement is well-typed if its expression child is well-typed. In GoLite, only function call expressions are allowed to be used as statements, i.e. `foo(x, y)` can be used as a statement, but `x-1` cannot.

$$\frac{V, T, F \vdash e : \tau \quad V, T, F \vdash rest}{V, T, F \vdash e; rest}$$

3.4 `return`

`return`

A `return` statement with no expression is well-typed if the enclosing function has no return type.

$$\frac{F = void \quad V, T, F \vdash rest}{V, T, F \vdash return; rest}$$

`return expr`

A `return` statement with an expression is well-typed if its expression is well-typed and the type of this expression is the same as the return type of the enclosing function.

$$\frac{V, T, F \vdash e : \tau \quad F = \tau \quad V, T, F \vdash rest}{V, T, F \vdash return\ e; rest}$$

Note: although the statements after a `return` can never actually be executed, we need to type check them nonetheless.

3.5 Short declaration

`x1, x2, ..., xk := e1, e2, ..., ek`

A short declaration type checks if:

1. All the expressions on the right-hand side are well-typed;
2. At least one variable on the left-hand side is not declared in the current scope;
3. The variables already declared in the current scope are assigned expressions of the same type. E.g. if the symbol table contains the mapping `x1 -> T1`, then it must be the case that `typeof(e1) = T1`.

If these conditions are met, the mappings `x1 -> typeof(e1)`, `x2 -> typeof(e2)`, ..., `xk -> typeof(ek)` are added to symbol table.

(Take a deep breath right now, scary math just ahead!)

$$\frac{\begin{array}{ll} V, T, F \vdash e_1 : \tau_1 \quad ... \quad V, T, F \vdash e_k : \tau_k & (1) \\ \exists i \in \{1..k\} : x_i \notin V & (2) \\ \forall i \in \{1..k\} : x_i \in V \implies V(x_i) = \tau_i & (3) \\ V \cup \{x_1 : \tau_1, ..., x_k : \tau_k\}, T, F \vdash rest & \end{array}}{V, T, F \vdash x_1, ..., x_k \ := \ e_1, ..., e_k; rest}$$

Hint: short declarations are hard to get right, make sure you write a bunch of tests and compare against the Go compiler and the reference GoLite compiler.

3.6 Declarations

Declaration statements obey the rules described in the previous section.

3.7 Assignment

```
v1, v2, ..., vk = e1, e2, ..., en
```

An assignment statement type checks if:

- All the expressions on the left-hand side are well-typed;
- All the expressions on the right-hand side are well-typed;
- For every pair of lvalue/expression, `typeof(vi) = typeof(ei)` (no resolving).

$$\frac{\begin{array}{ccc} V,T,F \vdash v_1 : \tau_1 & ... & V,T,F \vdash v_k : \tau_k \\ V,T,F \vdash e_1 : \tau_1 & ... & V,T,F \vdash e_k : \tau_k \\ & V,T,F \vdash rest & \end{array}}{V,T,F \vdash v_1, v_2, ..., v_k = e_1, e_2, ..., e_k; rest}$$

Additionally, the expressions on the left-hand side must be lvalues (addressable):

- Variables (non-constant)
- Slice indexing
- Array indexing of an addressable array
- Field selection of an addressable struct
- https://golang.org/ref/spec#Address_operators

3.8 Op-assignment

```
v op= expr
```

An op-assignment statement type checks if:

- The expression on the left-hand side is well-typed;
- The expression on the right-hand side is well-typed;
- The operator `op` accepts two arguments of types `typeof(v)` and `typeof(expr)` and return a value of type `typeof(v)`.

$$\frac{V,T,F \vdash v : \tau \quad V,T,F \vdash e : \tau \quad V,T,F \vdash rest}{V,T,F \vdash v \text{ op=} \ e; rest}$$

The expressions on the left-hand size must also be lvalues.

3.9 Block

```
{
        // statements
}
```

A block type checks if its statements type check. A block opens a new scope in the symbol table.

$$\frac{\begin{array}{l}\forall i \in \{1..k\} : V,T,F \vdash stmt_i \\ V,T,F \vdash rest\end{array}}{V,T,F \vdash \{stmt_1; ...; stmt_k\}; rest}$$

3.10 `print` and `println`

```
print(e1, ..., ek)
println(e1, ..., ek)
```

A print statement type checks if all its expressions are well-typed and resolve to a base type (int, float64, bool, string, rune).

$$\frac{\begin{array}{l}\forall i \in \{1..k\} : V,T,F \vdash e_i : \tau_i \\ RT(T,\tau_i) \in \{int, float64, rune, string, bool\} \\ V,T,F \vdash rest\end{array}}{V,T,F \vdash println(e_1, ..., e_k); rest}$$

3.11 For loop

```
for {
        // statements
}
```

An infinite for loop type checks if its body type checks. The body opens a new scope in the symbol table.

$$\frac{\forall i \in \{1..k\} : V, T, F \vdash stmt_i}{V, T, F \vdash for\ \{stmt_1; ...; stmt_k\}; rest}$$

```
for expr {
        // statements
}
```

A "while" loop type checks if:

- Its expression is well-typed and resolves to type `bool`;
- The statements type check.

The body opens a new scope in the symbol table.

$$\frac{V, T, F \vdash e : \tau \quad RT(T, \tau) = bool \quad \forall i \in \{1..k\} : V, T, F \vdash stmt_i}{V, T, F \vdash for\ e\ \{stmt_1; ...; stmt_k\}; rest}$$

```
for init; expr; post {
        // statements
}
```

A three-part for loop type checks if:

1. `init` type check;
2. `expr` is well-typed and resolves to type `bool`;
3. `post` type checks;
4. the statements type check.

The `init` statement can shadow variables declared in the same scope as the `for` statement. The body opens a new scope in the symbol table and can redeclare variables declared in the `init` statement.

$$\frac{\begin{array}{ll} V, T, F \vdash init & (1) \\ V, T, F \vdash expr : \tau & (2) \\ V, T, F \vdash RT(R, \tau) = bool & (2) \\ V, T, F \vdash post & (3) \\ \forall i \in \{1..k\} : V, T, F \vdash stmt_i & (4) \end{array}}{V, T, F \vdash for\ init; expr; poststmt_1; ...; stmt_k}$$

3.12 If statement

```
if init; expr {
        // then statements
} else {
        // else statements
}
```

An if statement type checks if:

1. `init` type checks;
2. `expr` is well-typed and resolves to type `bool`;
3. The statements in the first block type check;
4. The statements in the second block type check.

The `init` statement can shadow variables declared in the same scope as the `for` statement. The bodies both open a new scope in the symbol table and can redeclare variables declared in the `init` statement.

$$\frac{\begin{array}{ll} V,T,F \vdash init & (1) \\ V,T,F \vdash expr : \tau & (2) \\ V,T,F \vdash RT(T,\tau) = bool & (2) \\ V,T,F \vdash then\_stmts & (3) \\ V,T,F \vdash else\_stmts & (4) \end{array}}{V,T,F \vdash if\ init; expr\ \{then\_stmts\}\ else\ \{else\_stmts\}}$$

3.13 Switch statement

```
switch init; expr {
case e1, e2, ..., en:
        // statements
default:
        // statements
}
```

A switch statement with an expression type checks if:

- `init` type checks;
- `expr` is well-typed and is a comparable type;

- The expressions `e1`, `e2`, ..., `en` are well-typed and have the same type as `expr`;
- The statements under the different alternatives type check.

```
switch init; {
case e1, e2, ..., en:
        // statements
default:
        // statements
}
```

A switch statement without an expression type checks if:

- `init` type checks;
- The expressions `e1`, `e2`, ..., `en` are well-typed and have type `bool`;
- The statements under the different alternatives type check.

3.14 Increment/decrement statements

```
expr++
expr--
```

An increment/decrement statement type checks if its expression is well-typed and resolves to a numeric base type (int, float64, rune).

$$\frac{V,T,F \vdash e : \tau \quad RT(T,\tau) \in \{int, float64, rune\} \quad V,T,F \vdash rest}{V,T,F \vdash e\text{<op>}; rest}$$

4 Expressions

Type checking of an expression involves making sure that all its children are well-typed **and also** giving a type to the expression itself. This type can should be stored (either in the AST itself or in an auxiliary data structure) as it will be queried by the expression's parent.

4.1 Literals

In Go, literals are *untyped* and have complex rules; in GoLite, literals are *typed* and we've deliberately simplified the rules to make your type checker easier to implement.

```
42              // int
1.62            // float64
'X'             // rune
"comp520"       // string
```

The different literals have obvious types:

- Integer literals have type `int`
- Float literals have type `float64`
- Rune literals have type `rune`
- String literals have type `string`

$$\frac{n \text{ is an integer literal}}{V, T, F \vdash n : int} \quad \frac{n \text{ is a float literal}}{V, T, F \vdash f : float64} \quad \frac{n \text{ is a rune literal}}{V, T, F \vdash r : rune} \quad \frac{n \text{ is a string literal}}{V, T, F \vdash s : string}$$

4.2 Identifiers

The type of an identifier is obtained by querying the symbol table. If the identifier cannot be found in the symbol table, an error is raised.

$$\frac{V(x) = \tau}{V, T, F \vdash x : \tau}$$

4.3 Unary exression

```
unop expr
```

A unary expression is well-typed if its sub-expression is well-typed and has the appropriate type for the operation. In GoLite, the type of a unary expression is always the same as its child.

- Unary plus: `expr` must resolve to a numeric type (int, float64, rune)
- Negation: `expr` must resolve to a numeric type (int, float64, rune)
- Logical negation: `expr` must resolve to a bool
- Bitwise negation: `expr` must resolve to an integer type (int, rune)

$$\frac{V,T,F \vdash e : \tau \qquad V,T,F \vdash RT(T,\tau) \in \{int, float64, rune\}}{V,T,F \vdash +e : \tau} \qquad \frac{V,T,F \vdash e : \tau \qquad V,T,F \vdash RT(T,\tau) \in \{int, float64, rune\}}{V,T,F \vdash -e : \tau}$$

$$\frac{V,T,F \vdash e : \tau \qquad V,T,F \vdash RT(T,\tau) = bool}{V,T,F \vdash !e : \tau} \qquad \frac{V,T,F \vdash e : \tau \qquad V,T,F \vdash RT(T,\tau) \in \{int, rune\}}{V,T,F \vdash \hat{}\,e : \tau}$$

4.4 Binary expressions

`expr binop expr`

A binary expression is well-typed if its sub-expressions are well-typed, are of the same type and that type resolves to a type appropriate for the operation. The type of the binary operation is detailed in the table below. The Go specification (links below) explains which types are ordered, comparable, numeric, integer, etc.

<table>
<tr><th>arg1</th><th>op</th><th>arg2</th><th>result</th></tr>
<tr><td>bool</td><td>||</td><td>bool</td><td>bool</td></tr>
<tr><td>bool</td><td>&&</td><td>bool</td><td>bool</td></tr>
<tr><td>comparable</td><td>==</td><td>comparable</td><td>bool</td></tr>
<tr><td>comparable</td><td>!=</td><td>comparable</td><td>bool</td></tr>
<tr><td>ordered</td><td><</td><td>ordered</td><td>bool</td></tr>
<tr><td>ordered</td><td><=</td><td>ordered</td><td>bool</td></tr>
<tr><td>ordered</td><td>></td><td>ordered</td><td>bool</td></tr>
<tr><td>ordered</td><td>>=</td><td>ordered</td><td>bool</td></tr>
<tr><td>numeric or string</td><td>+</td><td>numeric or string</td><td>numeric or string</td></tr>
<tr><td>numeric</td><td>-</td><td>numeric</td><td>numeric</td></tr>
<tr><td>numeric</td><td>*</td><td>numeric</td><td>numeric</td></tr>
<tr><td>numeric</td><td>/</td><td>numeric</td><td>numeric</td></tr>
<tr><td>integer</td><td>%</td><td>integer</td><td>integer</td></tr>
<tr><td>integer</td><td>|</td><td>integer</td><td>integer</td></tr>
<tr><td>integer</td><td>&</td><td>integer</td><td>integer</td></tr>
<tr><td>integer</td><td><<</td><td>integer*</td><td>integer</td></tr>
<tr><td>integer</td><td>>></td><td>integer*</td><td>integer</td></tr>
<tr><td>integer</td><td>&^</td><td>integer</td><td>integer</td></tr>
<tr><td>integer</td><td>^</td><td>integer</td><td>integer</td></tr>
</table>

Note: The Go specification states that if the divisor of a division is zero, the compiler should report an error. In GoLite, we allow such expressions and let the executable program throw the appropriate error.

*Shift operations in Go require unsigned integers on the left-hand side. Since GoLite does not support such types, we will simply allow signed types to be used.

$$\frac{V,T,F \vdash e_1 : \tau_1 \quad V,T,F \vdash e_2 : \tau_2 \quad \tau_1 = \tau_2 \quad RT(T,\tau_1) = bool \quad op \in \{||, \&\&\}}{V,T,F \vdash e_1 \; op \; e_2 : \tau_1}$$

$$\frac{V,T,F \vdash e_1 : \tau_1 \quad V,T,F \vdash e_2 : \tau_2 \quad \tau_1 = \tau_2 \quad RT(T,\tau_1) \text{ is comparable} \quad op \in \{==, !=\}}{V,T,F \vdash e_1 \; op \; e_2 : bool}$$

$$\frac{V,T,F \vdash e_1 : \tau_1 \quad V,T,F \vdash e_2 : \tau_2 \quad \tau_1 = \tau_2 \quad RT(T,\tau_1) \text{ is ordered} \quad op \in \{<=, <, >, >=\}}{V,T,F \vdash e_1 \; op \; e_2 : bool}$$

$$\frac{V,T,F \vdash e_1 : \tau_1 \quad V,T,F \vdash e_2 : \tau_2 \quad \tau_1 = \tau_2 \quad RT(T,\tau_1) \in \{numeric, string\} \quad op = +}{V,T,F \vdash e_1 \; op \; e_2 : \tau_1}$$

$$\frac{V,T,F \vdash e_1 : \tau_1 \quad V,T,F \vdash e_2 : \tau_2 \quad \tau_1 = \tau_2 \quad RT(T,\tau_1) \text{ is numeric} \quad op \in \{-, *, /\}}{V,T,F \vdash e_1 \; op \; e_2 : \tau_1}$$

$$\frac{V,T,F \vdash e_1 : \tau_1 \quad V,T,F \vdash e_2 : \tau_2 \quad \tau_1 = \tau_2 \quad RT(T,\tau_1) \text{ is integer} \quad op \in \{\%, |, \&, <<, >>, \&\hat{}, \hat{}\}}{V,T,F \vdash e_1 \; op \; e_2 : \tau_1}$$

- http://golang.org/ref/spec#Arithmetic_operators
- http://golang.org/ref/spec#Comparison_operators
- http://golang.org/ref/spec#Logical_operators

4.5 Function call

`expr(arg1, arg2, ..., argk)`

A function call is well-typed if:

- `arg1`, `arg2`, ..., `argk` are well-typed and have types `T1`, `T2`, ..., `Tk` respectively;
- `expr` is well-typed and has function type `(T1 * T2 * ... * Tk) -> Tr`.

The type of a function call is `Tr`.

$$\frac{V,T,F \vdash e_1 : \tau_1 \quad ... \quad V,T,F \vdash e_k : \tau_k \qquad V,T,F \vdash e : \tau_1 \times ... \times \tau_k \rightarrow \tau_r}{V,T,F \vdash e(e_1,...,e_k) : \tau_r}$$

Note that the special function `init` may *not* be called.

4.6 Indexing

`expr[index]`

Indexing into a slice or an array is well-typed if:

- `expr` is well-typed and resolves to `[]T` or `[N]T`;
- `index` is well-typed and resolves to `int`.

The result of the indexing expression is `T`.

Note: The Go specification states that the compiler should report an error if the index of an array (not of a slice) evaluates to a statically-known constant that is outsides the bounds of the array. You do not have to implement this at compile-time in GoLite, instead we'll do the check at runtime.

$$\frac{V,T,F \vdash e : \tau_1 \quad RT(T,\tau_1) \in \{[]\tau, [N]\tau\} \qquad V,T,F \vdash i : \tau_2 \quad RT(T,\tau_2) = int}{V,T,F \vdash e[i] : \tau}$$

4.7 Field selection

`expr.id`

Selecting a field in a struct is well-typed if:

- `expr` is well-typed and has type `S`;
- `S` resolves to a struct type that has a field named `id`.

The type of a field selection expression is the type associated with `id` in the struct definition.

$$\frac{V, T, F \vdash e : S \quad RT(T, S) = struct\{..., id : \tau, ...\}}{V, T, F \vdash e.id : \tau}$$

4.8 `append`

`append(e1, e2)`

An `append` expression is well-typed if:

- `e1` is well-typed, has type `S` and `S` resolves to a `[]T`;
- `e2` is well-typed and has type `T`.

The type of `append` is `S`

$$\frac{V, T, F \vdash e_1 : S \quad RT(T, S) = []\tau \quad V, T, F \vdash e_2 : \tau}{V, T, F \vdash append(e_1, e_2) : S}$$

4.9 Type cast

`type(expr)`

A type cast expression is well-typed if:

- `type` resolves to a base type `int`, `float64`, `bool`, `rune` or `string`;
- `expr` is well-typed and has a type that can be be cast to `type`:
    1. `type` and `expr` resolve to identical underlying types;
    2. `type` and `expr` both resolve to numeric types;
    3. `type` resolves to a `string` type and `expr` resolves to an integer type (`rune` or `int`)

The type of a type cast expression is `type`.

$$\frac{\begin{array}{l} V,T,F \vdash RT(T,\tau) \in \{int, float64, bool, rune, string\} \\ V,T,F \vdash e : \tau_2 \\ \tau_2 \text{ can be cast to } \tau \end{array}}{V,T,F \vdash \tau(e) : \tau}$$