

Principles of Programming Languages

Type Checking

We return in this Chapter to the issue of type safety that was presented informally in Chapter 1 when we introduced the type system of TypeScript. We investigate how we can analyze programs to verify they are type safe.

An attempt to apply a procedure to inappropriate data is a *type error*.

We develop techniques to analyze programs so that we can ensure they are *type safe* - that is, if we compute them, on any possible input values, we do not reach type errors.

In order to support type checking, we will extend our language with type annotations - in the same way as TypeScript extends JavaScript.

Given a program with annotations such as:

```
(define f  
  (lambda ((n : number)) : number  
    (+ n 3)))
```

```
(f 'x) ;; => Type error:  
      ;; 'x is not a number
```

We proceed in two stages:

- We first analyze programs that include full type annotations and verify that they satisfy their type declarations.

- We then analyze programs that include **partial** type annotations (and possibly no annotations at all) and infer the types of all variable declarations and functions, and check that the program is type safe.

Contracts of programs provide specification for their most important properties:

- Signature
- Type preconditions and postconditions

Contracts say nothing about the specifics of the implementation (such as performance, concrete data structures used in the implementation).

Proving program correctness consists in proving that a program implementation satisfies its contract. Type safety is one of the conditions we must check to prove correctness:

1. **Type correctness:** Check well-typing of all expressions, and possibly infer missing types.
2. **Program verification:** Show that if preconditions hold, then the program terminates, and the postconditions hold.

Program correctness can be checked either
statically or **dynamically**.

In static program correctness the program text is analyzed without running it. Static program analysis reveals problems that characterize the program independently of specific data.

Static type checking verifies that the program will not encounter runtime errors due to type mismatch problems. In dynamic program analysis, problems are detected by running the program on specific data.

Static correctness methods are strong because they analyze the program as a whole, and evaluate properties that hold for all possible applications on all possible data values. Dynamic correctness methods, like unit testing, are complementary to the static ones.

Let us recall the meaning of types we adopt:

The semantics of a programming language defines types as subsets of the domain of computed values. That is, values are split into subsets, termed types, that collect together values of a similar kind and which can be passed to similar functions.

In the Scheme subset that we have defined up to *L4* - computed values are the union of the disjoint types Numbers, Booleans, S-expressions, Closures, Primitive operators and Void.

The set of closures is defined inductively as mappings from tuples to values - where tuples are cartesian products of values - starting from the empty tuple, tuples of a single value, two values etc.

Most programming languages admit **fully typed semantics**, *i.e.*, every computed value belongs to a known type.

The semantics of a programming language defines a type system: it determines which types exist across the domain of computed values, how new types can be defined (through the usage of type constructors - such as List or Procedures or Union) and the possible relations among types (one type may be included in another, two types may be disjoint or overlap).

The basis of type systems is the **principle of substitutability**: two types A and B “match” when values of one can be used in place of values of the other. Therefore, the design of a type system determines when substitutions are safe.

The simplest form of substitutability is **identity**: a type can only be substituted with itself, and nothing else. For instance, if the declared type of a function's parameter is `Number`, then you can only call it with `Number`-typed values, nothing else. This is known as **invariance**: the set of values that can be passed into a function cannot vary from the set expected by that type.

This is so obvious that it might seem to hardly warrant a name - however, it is useful to name this because it sets up a contrast with more complex type systems when richer, non-trivial notions of substitutability exist (think of Subtyping in Object Oriented systems and the usage of Interfaces).

The key property of a type system is a set of rules which determine whether a given expression in the language is type safe - that is, whether the evaluation of this expression will never lead to type errors.

Type safety is achieved by defining an analysis method called **type checking**. The goal of type checking is to verify that if an expression E is assigned type T , then, whenever E is computed, its value will be of type T . If the type system has this property, we say that it is **sound**.

Note that type checking does not guarantee that the program will always terminate (that would be a strong guarantee equivalent to solving the Halting Problem) nor that it will not throw any exceptions, such as divide by 0. It only guarantees that the program when it is evaluated to a proper value will not throw type errors and will return a value in the predicted type.

The type checker inspects every application node in the AST of a program E . Each operand in an application is an expression of some type (which is verified inductively). Therefore, we know that the value of the operand will be of that type.

If the operands are not of the type expected by the operator of the application, we say that this operator invocation (*i.e.*, this application expression node in the AST) is a **potential type error**.

If type errors are detected, the type checker can take some actions, which is also part of the language design. It can refuse to execute or compile the program, or it can take corrective measures (like type casting).

Type checking and type inference require associating program expressions with types.

In order to achieve this, we need to define two syntactic extensions to our language:

- Define a type language to specify type expressions.
- Define a way in the language to associate variables and procedures to type expressions.

The extension of a language with type expressions is exactly what we observed in the transition from JavaScript to TypeScript. TypeScript defines a way to specify type expressions (primitive types like Number, Boolean, String and compound types such as maps, arrays and functions, or type unions).

Similarly, we will define a new language, L_5 which extends L_4 by allowing the specification of type annotations.

We start with a definition of the type language.

We actually already used this type language when we introduced Scheme. In the code we wrote in Scheme, we added type annotations as part of the contract section of functions, under the type annotation.

Because we did not want to extend the language, we kept these annotations as comments in Scheme, as in the following example:

```
;; Purpose: Identity  
;; Signature: id(x)  
;; Type: [T -> T]  
(define id  
  (lambda (x) x))
```

Since we now know how to define our own language, we will add type annotations as part of the language *L5*.

The possible type expressions we will consider
are defined by the following syntax:

Type Language

```
<texp>          ::= <atomic-te> | <composite-te> | <tvar>
<atomic-te>      ::= <num-te> | <bool-te> | <void-te>
<num-te>         ::= number    // num-te()
<bool-te>        ::= boolean   // bool-te()
<str-te>         ::= string    // str-te()
<void-te>        ::= void      // void-te()
<composite-te>   ::= <proc-te> | <tuple-te>
<non-tuple-te>   ::= <atomic-te> | <proc-te> | <tvar>
<proc-te>        ::= [ <tuple-te> -> <non-tuple-te> ]
                  // proc-te(param-tes: list(te), return-te: te)
<tuple-te>       ::= <non-empty-tuple-te> | <empty-te>
<non-empty-tuple-te> ::= ( <non-tuple-te> *) * <non-tuple-te>
                  // tuple-te(tes: list(te))
<empty-te>       ::= Empty
<tvar>           ::= a symbol starting with T // tvar(id: Symbol)
```

The following are all examples of legal type expressions according to this syntax:

```
number
boolean
void
(number -> boolean)
(number * number -> boolean)
(number -> (number -> boolean))
(Empty -> number)
(Empty -> void)
(T1 -> T1)
```

We then define a way to add type annotations to expressions.

Where are type annotations needed within programs?

They can occur in only two specific places:

- As part of a variable declaration
- As part of a procedure expression to specify the expected return type

Accordingly, we extend the syntax of *L4* with type annotations in exactly those two expression types - these two changes are marked with **####** **L5** below:

Type Annotations

```
The only changes in the syntax of L5 are optional type annotations in var-decl and proc-exp
<program> ::= (L5 <exp>+) // program(exps:List(exp))
<exp> ::= <define-exp> | <cexp>
<define-exp> ::= (define <var-decl> <cexp>) // def-exp(var:var-decl, val:cexp)
<cexp> ::= <num-exp> // num-exp(val:Number)
          | <bool-exp> // bool-exp(val:Boolean)
          | <prim-op> // prim-op(op:Symbol)
          | <var-ref> // var-ref(var:Symbol)
          | (if <exp> <exp> <exp>) // if-exp(test,then,else)
          | (quote <sexp>) // lit-exp(val:Sexp)
          | (let (<binding>*) <cexp>+) // let-exp(bindings:List(binding), body:List(cexp))
          | (letrec (<binding>*) <cexp>+) // letrec-exp(bindings:List(binding), body:List(cexp))
          | (<cexp> <cexp>*) // app-exp(rator:cexp, rands:List(cexp))

          | (lambda (<var-decl>*) [: <texp>]? <cexp>+)
              // proc-exp(params:List(var-decl), body:List(cexp), return-te: Texp) ##### L5
<var-decl> ::= <symbol> | [<symbol> : <texp>] // var-decl(var:Symbol, type:Texp) ##### L5

;; Unchanged
<prim-op> ::= + | - | * | / | < | > | = | not | eq?
          | cons | car | cdr | pair? | list? | number? | boolean? | symbol? | display | newline
<num-exp> ::= a number token
<bool-exp> ::= #t | #f
<var-ref> ::= an identifier token
<sexp> ::= a symbol token | ( <sexp>* )
<binding> ::= ( <var-decl> <cexp> ) // Binding(var:var-decl, val:cexp)
```

With this new syntax (heavily inspired by the TypeScript syntax), the following programs are fully annotated *L5* programs:

Type Annotations

```
(define (x : number) 5)
```

```
(define (f : (number -> number))  
  (lambda ((x : number)) : number  
    (* x x)))
```

```
(define (f : (number * number -> number))  
  (lambda ((x : number) (y : number)) : number  
    (* x y)))
```

Type Annotations

```
(let (((a : number) 1)
      ((b : boolean) #t))
  (if b a (+ a 1)))
```

```
(letrec (((a : (number -> number))
          (lambda ((x : number)) : number
            (* x x))))
  (a 3))
```

```
(define (id : (T1 -> T1))
  (lambda ((x : T1)) : T1
    x))
```

Type Annotations

We defined type annotations as optional - so that the following programs are also legal expressions in *L5*: no type annotation on `f` and on return value of `lambda`:

```
(define f  
  (lambda ((x : number))  
    (* x x)))
```

```
(let ((a 1)) (+ a a))
```

The implementation of this syntax definition is available on GitHub in the files:

- `src/L5/TExp.ts`
- `src/L5/L5-ast.ts`

It includes the following functions:

- **parseTE** takes a **string** representing a concrete type expression and returns a TExp AST.
- **unparseTExp** takes a TExp AST and returns the concrete string of the AST
- **parseL5** takes a **string** representing an L5 program and returns an L5 Program AST.
- **unparse** takes an **Exp | Program** AST and returns the concrete string of the AST.

Type expression ASTs look as shown in these examples:

```
it('parses atoms', () => {  
  expect(parseTE("number")).to.deep.equal(makeOk(makeNumTExp()));  
  expect(parseTE("boolean")).to.deep.equal(makeOk(makeBoolTExp()));  
});
```

```
it('parses type variables', () => {  
  expect(parseTE("T1")).to.deep.equal(makeOk(makeTVar("T1")));  
});
```

Type expression ASTs look as shown in these examples:

```
it('parses procedures', () => {  
  expect(parseTE("(T * T -> boolean)"))  
    .to.deep.equal(makeOk(  
      makeProcTExp([makeTVar("T"), makeTVar("T")], makeBoolTExp())));  
  
  expect(parseTE("(number -> (number -> number))"))  
    .to.deep.equal(makeOk(  
      makeProcTExp([makeNumTExp()],  
                    makeProcTExp([makeNumTExp()], makeNumTExp()))));  
});
```

```
it('parses "void" and "Empty"', () => {  
  expect(parseTE("void")).to.deep.equal(makeOk(makeVoidTExp()));  
  
  expect(parseTE("(Empty -> void)"))  
    .to.deep.equal(makeOk(makeProcTExp([], makeVoidTExp())));  
});
```

Note the following two points about the syntax of type expressions:

- At this point, we only support atomic type expressions (`number`, `boolean`, `void`) or procedure type expressions (`number -> number`) - and not composite types such as `List(number)`.
- Type variables are used to represent the case of polymorphic functions such as `[T -> T]` for the identity function.

ASTs with type annotations look as in the following examples:

```
const p = (x: string): Result<Exp> =>
  bind(parseSexp(x), parseL5Exp);

expect(p("(define (a : number) 1)"))
.to.deep.equal(makeOk(
  makeDefineExp(makeVarDecl("a", makeNumTExp()),
    makeNumExp(1))));

expect(p("(lambda ((x : number)) : number x)"))
.to.deep.equal(makeOk(
  makeProcExp([makeVarDecl("x", makeNumTExp())],
    [makeVarRef("x")],
    makeNumTExp())));
```

We develop an algorithm which operates over an annotated AST expression of *L5* (the AST of the language with full type annotations for all **VarDecl** nodes and all procedures), and verifies that the expression is type safe and will return its verified type. This algorithm is a **type checker**.

The specific errors we will detect are the following:

- An attempt to apply a value which is neither a primitive nor a closure in an application expression.
- An attempt to apply a procedure or a primitive operator to the wrong number of arguments.

The specific errors we will detect are the following:

- An attempt to apply primitives to wrong type of arguments (for example, `+` to a non-number value)
- An attempt to use a non-boolean expression as the test in an `if`-expression.

Note that we do not try to check for other types of errors - such as divide by zero or, if we had lists or arrays, reference to an index of bounds in the list or the array.

We design this algorithm as a function `typeofExp` which given an expression will traverse the expression (the AST) and verify all the nodes in the AST for type correctness, and return the type the expression is expected to return. That is, we first expect the function `typeofExp` to have the following type:

$$\text{typeofExp}: \text{Exp} \rightarrow \text{TExp}$$

It is easy to think of how this function will work for simple expression types:

```
typeofExp(<NumExp val>) -> NumTExp
```

That is, the type of an AST node of the form
 <NumExp val> is Number. Similarly for
 booleans and strings.

We then must decide what should be the type of an expression which only consists of a **VarRef** - that is a reference to a variable.

Obviously, this depends on the context of the variable - since the same variable name in different contexts will yield different answers. We must, therefore, extend the signature of the **typeofExp** operation to accept as an additional parameter the assumptions we make about the type of variables in our program.

We had exactly the same issue when we defined the operational semantics in Chapter 2.

In order to compute the value of variables (even in the simplest model $L1$), we introduced an **environment** which keeps track of *what we know about variables*.

In a similar manner, we define **type environments** as a way to keep track of what we know about the **type of variables** in the program. As usual, we define the type environment in an inductive manner:

Definition: Type Environment

A type environment is a substitution of **language variables** to **type expressions**, *i.e.*, a mapping of a finite set of variables to type expressions. It is denoted as a set of variable type assumptions.

Type Environment

Definition: Type Environment

For example:

$$\{x: \text{number}, y: (\text{number} \rightarrow T1)\}$$

In this type environment, the variable **x** is mapped to the **Number** type, and the variable **y** is mapped to the polymorphic procedure type $(\text{number} \rightarrow T)$.

The type of a variable v with respect to a type environment TEnv is denoted $\text{TEnv}(v)$ (or $\text{applyTEnv}(\text{TEnv}, v)$).

1. The empty type environment, denoted $\{\}$, indicates that we make no assumptions about the types of variables.
2. Extending a type environment: we construct new type environments by combining new assumptions about variable-type mappings with another existing type environment. Formally, this is achieved by composing substitutions.

For example, if we combine the type assumption about the type of variable **z**:
{z: boolean} with the substitution above,
we obtain:

$$\{ x: \text{number}, y: (\text{number} \rightarrow T) \} \circ \{ z: \text{boolean} \} \\ = \{ x: \text{number}, y: (\text{number} \rightarrow T), z: \text{boolean} \}$$

The empty substitution is the neutral element of the substitution-composition operation:

$$\{\} \circ \{x_1: T_1, \dots, x_n: T_n\} = \{x_1: T_1, \dots, x_n: T_n\}$$

The `typeofExp` operation has thus the following signature and type definition:

`typeofExp: Exp * TEnv -> TExp`

We make efforts in this section to reuse the same mechanisms we used when describing the operational semantics of the language - environments and substitutions.

Equipped with type environments, we can define the type of variable expressions:

```
typeofExp(<VarRef var>, tenv) -> tenv(var)
```

What type should we return for a var in case we made no assumptions about its type in TEnv?

At present, we will trigger this as an error - as we only consider the case of fully typed programs, *i.e.*, we require the programmer to declare the type of all the variables in the program. Variables cannot be referenced if they are not declared beforehand.

We will revisit this decision later when we consider the task of **type inference** as opposed to **type checking**.

In the same way as we defined evaluation rules for each type of expression, we define type analysis rules.

To describe the typing rules, we first define a useful device we call a **typing statement**:

Type Statements

Definition: Typing Statement

A typing statement is a true/false formula that states a judgment about the **type of an expression, given a type environment.**

Notation: $TEnv \vdash e : T$

This statement means that if the type of variables in a language expression e is as specified in $TEnv$, then the type of e is T .

For example, the typing statement:

$$\{x: \text{Number}\} \vdash (+\ x\ 5): \text{Number}$$

states that under the assumption that the type of x is **Number**, the type of $(+\ x\ 5)$ is **Number**.

The typing statement:

$$\{f: [T1 \rightarrow T2], g: T1\} \vdash (f \ g): T2$$

states that for every consistent replacement of $T1$, $T2$, under the assumption that the type of f is $[T1 \rightarrow T2]$, and the type of g is $T1$, the type of $(f \ g)$ is $T2$.

The following typing statements are false:

$$\{f: [T1 \rightarrow T2]\} \vdash (f\ g): T2$$

This is false because having no type assumption on g , $(f\ g)$ might not satisfy the well-typing rules of Scheme, and create a runtime error.

$\{f: [\text{Empty} \rightarrow T_2], g: T_2\} \vdash (f\ g): T_2$

is false because based on the operational semantics of Scheme, if f is a parameter-less procedure, the expression $(f\ g)$ does not satisfy the well-typing rules of Scheme.

Let us enumerate typing rules for each type of expression in the language, starting with simple expression types. These typing rules define the type system of our programming language.

Typing rule Number:

For every type environment `_TEnv` and number `_n`:
`_TEnv |- (NumExp _n): Number`

Typing rule Boolean:

For every type environment `_TEnv` and boolean `_b`:
`_TEnv |- (BoolExp _b): Boolean`

Typing rule Variable:

For every type environment $_TEnv$ and variable $_v$:

$$_TEnv \vdash (VarRef _v): _TEnv(_v)$$

For primitive operators, we use the type definition of each primitive operator.

We know for example that `+` is a procedure with
type

`(Number * ... * Number -> Number)`.

We express this in a single typing rule for each
primitive procedure:

For every type environment `_TEnv`:

`_TEnv |- +: (Number * ... * Number -> Number)`

In the implementation of the type checker, to simplify the code, we ignore variadic primitives
- and consider $+$, $-$, $*$ and $/$ to be binary operators only.

Similarly, for other primitives:

For every type environment $_TEnv$:
 $_TEnv \vdash \text{not} : [_S \rightarrow \text{Boolean}]$

`_S` is a type variable. That is, `not` is a polymorphic primitive procedure - it applies to any type and returns a boolean value.

The `display` procedure has the typing rule:

For every type environment `_TEnv`:
`_TEnv |- display: (_S -> Void)`

`display` is also a polymorphic primitive procedure.

The expressions which include variable declarations are more complex and they involve multiple type environments. Let us review the rule for typing procedure expressions.

A procedure has the structure
`(lambda (x1 ... xn) body).`

With type annotations, we have:

`(lambda (x1:t1 ... xn:tn) : t body).`

What should be the type of this expression?

If we trust the annotations, the answer is
simple:

```
typeofExp((lambda (x1:t1 ... xn:tn) : t body))  
  = [t1 * ... * tn -> t]
```

But can we trust the annotations in a specific expression? This is exactly what we want to check, by traversing the body and type checking it under specific typing assumptions.

The rule for our language reads as follows:

Typing Rule for Procedures

Typing rule Procedure:

For every: type environment $_Tenv$,
variables $_x1, \dots, _xn$, $n \geq 0$
expressions $_e1, \dots, _em$, $m \geq 1$, and
type expressions $_S1, \dots, _Sn, _U1, \dots, _Um$:

Procedure with parameters ($n > 0$):

If $_Tenv \vdash \{ _x1 : _S1, \dots, _xn : _Sn \}$
| $_ei : _Ui$ for all $i = 1..m$,
Then $_Tenv \vdash (\lambda (_x1 \dots _xn) _e1 \dots _em) :$
 $[_S1 * \dots * _Sn \rightarrow _Um]$

Parameter-less Procedure ($n = 0$):

If $_Tenv \vdash _ei : _Ui$ for all $i=1..m$,
Then $_Tenv \vdash (\lambda () _e1 \dots _em) : [\text{Empty} \rightarrow _Um]$

Note how the type of the body is the type of the last expression in the body (the body is a list of expressions meant to be evaluated in sequence - the value of the body is the value of the last expression, hence the type of the body is the type of the last expression).

Still, we apply the rule to all the expressions in the body, to actually type check them.

Note next that the rule indicates that we type check the body in a TEnv where we assume that the parameters have the declared types.

The typing rules include meta-variables for language expressions, type expressions and type environments. When rules are instantiated, the meta-variables are replaced by real expressions of the same kind.

The meta-variables should not be confused with language or type variables. Therefore, they deliberately are preceded with an underscore to distinguish them from non-meta-variables.

Each typing rule specifies an independent (standalone), universally quantified typing statement. The meta-variables used in different rules are not related, and can be consistently renamed.

Every typing rule requires typing statements for all sub-expressions of the expression for which a typing statement is derived. This property guarantees type safety – the typing algorithm assigns a type to every sub-expression which is evaluated at run-time.

We will need to specify rules for **if** expressions, application expressions, **let** and **letrec** expressions as we go to complete the specification of the type system of the language.

We now have the tools to specify the type checking algorithm:

We assume here that **all** variable declarations and procedures are fully type annotated.

The algorithm traverses the AST of the expression, as we have learned to do when writing interpreters:

Type Checking Algorithm

```
// Purpose: Compute the type of an expression  
// Traverse the AST and check the type according to the exp type.  
// We assume that all variables and procedures  
// have been explicitly typed in the program.  
export const typeofExp = (exp: Parsed, tenv: TEnv): Result<TExp> =>  
  isNumExp(exp) ? makeOk(typeofNum(exp)) :  
  isBoolExp(exp) ? makeOk(typeofBool(exp)) :  
  isStrExp(exp) ? makeOk(typeofStr(exp)) :  
  isPrimOp(exp) ? typeofPrim(exp) :  
  isVarRef(exp) ? applyTEnv(tenv, exp.var) :  
  isIfExp(exp) ? typeofIf(exp, tenv) :  
  isProcExp(exp) ? typeofProc(exp, tenv) :  
  isAppExp(exp) ? typeofApp(exp, tenv) :  
  isLetExp(exp) ? typeofLet(exp, tenv) :  
  isLetrecExp(exp) ? typeofLetrec(exp, tenv) :  
  isDefineExp(exp) ? typeofDefine(exp, tenv) :  
  isProgram(exp) ? typeofProgram(exp, tenv) :  
  makeFailure("Unknown type");
```

Each rule is implemented in a dedicated procedure which traverses inductively its parameter.

The first few types of simple expressions are
simple procedures:

```
// a number literal has type num-te  
const typeofNum = (n: NumExp): NumTExp => makeNumTExp();  
  
// a boolean literal has type bool-te  
const typeofBool = (b: BoolExp): BoolTExp => makeBoolTExp();  
  
// a string literal has type str-te  
const typeofStr = (s: StrExp): StrTExp => makeStrTExp();
```

These procedures do not take **TEnv** as a parameter because they are true regardless of the **TEnv** state.

For primitive operators, we map the operator to its type expression:

Type Checking Algorithm

```
// primitive ops have known proc-te types
const numOpTEExp = parseTE('(number * number -> number)');
const numCompTEExp = parseTE('(number * number -> boolean)');
const boolOpTEExp = parseTE('(boolean * boolean -> boolean)');

// Todo: cons, car, cdr, list
export const typeofPrim = (p: PrimOp): Result<TEExp> =>
  (p.op === '+' ? numOpTEExp :
   p.op === '-' ? numOpTEExp :
   p.op === '*' ? numOpTEExp :
   p.op === '/' ? numOpTEExp :
   p.op === 'and' ? boolOpTEExp :
   p.op === 'or' ? boolOpTEExp :
   p.op === '>' ? numCompTEExp :
   p.op === '<' ? numCompTEExp :
   p.op === '=' ? numCompTEExp :
   // Important to use a different signature for each op with a TVar to avoid capture
   p.op === 'number?' ? parseTE('(T -> boolean)') :
   p.op === 'boolean?' ? parseTE('(T -> boolean)') :
   p.op === 'string?' ? parseTE('(T -> boolean)') :
   p.op === 'list?' ? parseTE('(T -> boolean)') :
   p.op === 'pair?' ? parseTE('(T -> boolean)') :
   p.op === 'symbol?' ? parseTE('(T -> boolean)') :
   p.op === 'not' ? parseTE('(boolean -> boolean)') :
   p.op === 'eq?' ? parseTE('(T1 * T2 -> boolean)') :
   p.op === 'string=?' ? parseTE('(T1 * T2 -> boolean)') :
   p.op === 'display' ? parseTE('(T -> void)') :
   p.op === 'newline' ? parseTE('(Empty -> void)') :
   makeFailure(`Primitive not yet implemented: ${p.op}`);
```

Let us now consider a case of a compound expression without variable declarations: what should be the type of an `if` expression?

Type Checking Compound Expressions

```
const typeofIf = (ifExp: IfExp, tenv: TEnv): Result<TExp> => {  
  const testTE = typeofExp(ifExp.test, tenv);  
  const thenTE = typeofExp(ifExp.then, tenv);  
  const altTE = typeofExp(ifExp.alt, tenv);  
  const constraint1 = bind(testTE, (testTE: TExp) =>  
    checkEqualType(testTE, makeBoolTEExp(), ifExp));  
  const constraint2 = safe2((thenTE: TExp, altTE: TExp) =>  
    checkEqualType(thenTE, altTE, ifExp))(thenTE, altTE);  
  return safe2((_c1: true, _c2: true) => thenTE)  
    (constraint1, constraint2);  
};
```

We check that the components of the expression are well typed, by invoking recursively **typeofExp** of each of the three components of the **if** expression.

We then compare the type computed by `typeofExp` with our expectations:

- The `test` component must be a boolean
- The `then` and `alt` components must be of the same type
- The type of the whole expression is that of the `then` component (same as the `alt` component)

The `if` expression typing rule is thus specified as follows:

Type Checking Compound Expressions

```
For every type environment _TEnv,  
    expressions _test, _then, _alt  
    type expression _S:  
If    _TEnv |- _test : Boolean  
    _TEnv |- _then : _S  
    _TEnv |- _alt  : _S  
Then _TEnv |- (if _test _then _alt) : _S
```

In this rule, the constraint that the type of the **then** component and **alt** component are compatible is captured by the fact that the same meta-variable appears (**_S**).

In the type checking algorithm, we enforce this by invoking the function **checkEqualType**.

At this stage, this constraint checking is implemented as a simple equality test:

Type Checking Compound Expressions

```
// Purpose: Check that type expressions are equivalent  
// as part of a fully-annotated type check process of exp.  
// Return an error if the types are different - true otherwise.  
// Exp is only passed for documentation purposes.  
const checkEqualType =  
  (te1: TExp, te2: TExp, exp: Exp): Result<true> =>  
    equals(te1, te2) ? makeOk(true) :  
    safe3((te1: string, te2: string, exp: string) =>  
      makeFailure<true>(`Incompatible types:  
        ${te1} and ${te2} in ${exp}`))  
      (unparseTExp(te1), unparseTExp(te2), unparse(exp));
```

This type equality test is appropriate in the case of type checking, we will change this to a more complex mechanism when we turn to the type inference algorithm.

This procedure implements the **invariant** type system we discussed above - types are compatible if they are identical.

Recall that in TypeScript we have a richer type system with subtyping (for example between map types and disjoint union types), and to type check such relations a more complex version of `checkEqualType` would be necessary.

Let us analyze how the type checker implements the typing rule for procedure expressions: We specified this rule as follows:

Typing Expressions with Variable Declarations

Typing rule Procedure:

For every: type environment $_Tenv$,
variables $_x1, \dots, _xn$, $n \geq 0$
expressions $_e1, \dots, _em$, $m \geq 1$, and
type expressions $_S1, \dots, _Sn, _U1, \dots, _Um$:

Procedure with parameters ($n > 0$):

If $_TEnv \vdash \{ _x1 : _S1, \dots, _xn : _Sn \}$
|- $_ei : _Ui$ for all $i = 1..m$,
Then $_TEnv \vdash (\lambda (_x1 \dots _xn) _e1 \dots _em) :$
 $[_S1 * \dots * _Sn \rightarrow _Um]$

Parameter-less Procedure ($n = 0$):

If $_TEnv \vdash _ei : _Ui$ for all $i=1..m$,
Then $_TEnv \vdash (\lambda () _e1 \dots _em) : [Empty \rightarrow _Um]$

The corresponding code in the type checker includes the recursive traversal of the sub-components in the body of the procedure:

Typing Expressions with Variable Declarations

```
// Purpose: compute the type of a proc-exp
// Typing rule:
// If   type<body>(extend-tenv(x1=t1,...,xn=tn; tenv)) = t
// then type<lambda (x1:t1,...,xn:tn) : t exp>(tenv) =
//      (t1 * ... * tn -> t)
const typeofProc = (proc: ProcExp, tenv: TEnv): Result<TExp> => {
  const argsTEs = map((vd) => vd.texp, proc.args);
  const extTEnv =
    makeExtendTEnv(map((vd) => vd.var, proc.args), argsTEs, tenv);
  const constraint1 =
    bind(typeofExps(proc.body, extTEnv),
        (body: TExp) => checkEqualType(body, proc.returnTE, proc));
  return bind(constraint1,
    _ => makeOk(makeProcTExp(argsTEs, proc.returnTE)));
};
```

Finally, let us consider the typing rule for application expression:

Typing Expressions with Variable Declarations

Typing rule Application:

For every: type environment $_TEnv$,
expressions $_f, _e1, \dots, _en$, $n \geq 0$, and
type expressions $_S1, \dots, _Sn, _S$:

Procedure with parameters ($n > 0$):

If $_TEnv \vdash _f : [_S1 * \dots * _Sn \rightarrow _S]$,
 $_TEnv \vdash _e1 : _S1, \dots, _TEnv \vdash _en : _Sn$
Then $_TEnv \vdash (_f _e1 \dots _en) : _S$

Parameter-less Procedure ($n = 0$):

If $_TEnv \vdash _f : [\text{Empty} \rightarrow _S]$
Then $_TEnv \vdash (_f) : _S$

The implementation in the type checker of this rule is [here](#).

Observe how the implementation verifies additional semantic errors:

- Invocation of a non-procedure type
- Invocation of a procedure with the wrong number of parameters

Let us observe the structure of the type checker: it is a typical syntax-driven traversal of the expression AST.

- All the nodes in the AST are exhaustively traversed.
- On each node, we apply a typing rule and compute a type value.

- When we traverse expressions which traverse a scope contour (bind variable declarations to values - such as application expressions, `let` or `letrec` expressions) or a new scope (such as procedure expressions), we maintain an environment to reflect the structure of the accessible variables.

This structure is parallel to the structure of the interpreters we analyzed in Chapter 2.

There are, however, important differences between the type checker and the interpreter:

Static vs. Dynamic Analysis

- The type checker sees only program text, whereas the interpreter runs over actual data.
- The type environment binds identifiers to types, whereas the interpreter's environment binds identifiers to values or locations (boxes).

- The type checker compresses (even infinite) sets of values into types, whereas the interpreter treats the elements of these sets distinctly.
- The type checker always terminates, whereas the interpreter might not.

- The type checker passes over the body of each expression only once, whereas the interpreter might pass over each body anywhere from zero to infinite times.