1.1.

1. Polymorphic type expression is a way to describe an infinite collection of types. For example, the type expression [T -> Number] is polymorphic, and represents a procedure from any type T to the type Number. This means its type is [Number -> Number] but also [Boolean -> Number], etc.

Polymorphic language expression is an expression that may yield different a value of a different type depends on the application. For example, the type of ((lambda (x) x) T) will change according to the type of T.

1. To specify a type, we need to provide a type constructor and a value constructor. Type constructor constructs types and value constructors create values of that type.

Type constructor in Java is a class declaration; For example, the following expression:

class Person {

String name;

Person(String name) {

this.name = name;

}

}

will construct the type "Person", and constructing such object, Person p = new Person("Obama");, will execute the value constructor for the type Person.

In Scheme, it is not possible to create types without structs, but we can call value constructors. For example, for the type Number, any actual number will call the value constructor for it and create it.

1. Under the assumptions that x is of type Number, and f is of type procedure whose domain is Number and range T, the application of proecdure f on x will be of type T.
2. Java is a language with a fully typed syntax, meaning it requires full specification of types for every language construct - constants, variables, functions, etc.

Partially typed syntax languages are languages that doesn't require type specification, but allows it. An example for such language is PHP, in which you may define a variable type, but it is not required.

1. Static typing is possible if the type checking algorithm for a language is based only on the program code, which means it can be ran without running the actual program. In Java, which is a fully typed language, this is possible. We can guarantee the type returned from Java expressions before run time.

Dynamic typing happens when in order to evaluate the types in a program we must run it. In Scheme, for example, we have no type information, so typing happens in run time - we have no way to know what would be the type of every expression before the actual run time.

1. The advantages for translating a recursive procedure into a CPS style is obvious - we get an iterative procedure, which won't use many stack frames and as so will have a smaller memory footprint.

CPS also allows to act differently upon success or failure and break early from the recursion.

CPS gives almost full control over the expression evaluation order, since we wrap calculations in closures and delay the evaluation. We can manipulate the evaluation order as we wish.

Another advantage is that we can return more than one value.

The disadvantages may be creating a lot closures and a code which is generally hard to understand, not intuitive and difficult to maintain.

1.2.

1. **False**. The expression:

(define f (lambda(x) (f x)))

(define g (lambda (x) 5))

(g (f 0))

Will result in the value 5 in normal-eval, but will never finish evaluating on applicative-eval, as it tries to evaluate (f 0) while normal-eval will never eval it, because the body of g does not contain x.

1. **True**. In Scheme, List is basically a Pair in which the second value is always a List, or null (Mind blowing recursive definition!).

So, every list can be constructed from chaining values and Pairs.

1. **False**. In Scheme, even though Pair is constructed of 2 values, it might be constructed with another Pair as a value, and so on, creating a pair that is virtually infinite.
2. **False**. (list 1 'a) creates an heterogeneous list, consisting of a Number type and a Symbol type.
3. **False**. The non-deterministic type inference algorithm will fail if there's no assumptions for a free variable in a given code snippet, but obviously Scheme can interpret it to an actual value that was defined out of that code snippet.
4. **False**. Rational numbers can be represented in infinite different ways (Consider 1/2 = 2/4 = 4/8, etc.), so if the implementation minimizes the rational number that is being constructed by make-rat, the given invariant won't hold.
5. **True**. Lazy procedural implementation of ADTs allows defining new operations without changing any previous implementation parts - a closure will be passed along to be executed on the data, rather than just a symbol defining the operation like in the eager implementation, which requires changing a few lines to support more operations.
6. **False**. They are just a way of abstracting implementation away and providing a cleaner method to perform such operations.

2.

1. (proc-return-te (make-proc-te (make-tuple-te (list ‘Number ‘Number)), ‘Boolean)) = ‘Boolean

The invariant makes sure the procedure type expression we created, Number\*Number -> Boolean will have a return type of Boolean.

So we want to prove that:

applicative-eval[(proc-return-te (make-proc-te (make-tuple-te (list ‘Number ‘Number)), ‘Boolean))] 🡺 ‘Boolean

applicative-eval[(proc-return-te (make-proc-te (\* . (‘Number . ‘Number)), ‘Boolean))] 🡺

applicative-eval[(proc-return-te (-> (\* . (‘Number . ‘Number)) . ‘Boolean))] 🡺

applicative-eval[(if (procedure? (-> (\* . (‘Number . ‘Number)) . ‘Boolean))

(caddr (-> (\* . (‘Number . ‘Number)) . ‘Boolean))

#f

)] 🡺

applicative-eval[if] 🡺 <if:primitive-procedure>

applicative-eval[(procedure? (-> (\* . (‘Number . ‘Number)) . ‘Boolean))]

applicative-eval[(-> (\* . (‘Number . ‘Number)) . ‘Boolean)] 🡺 (-> (\* . (‘Number . ‘Number)) . ‘Boolean)

applicative-eval[procedure?] 🡺 <Closure: (lambda (te) (and (list? te) (not (null? te)) (eq? (car te) '->)) ))>

…. 🡺#t

applicative-eval[(caddr (-> (\* . (‘Number . ‘Number)) . ‘Boolean))] 🡺

applicative-eval[caddr] 🡺 <caddr:primitive-procedure>

applicative-eval[(-> (\* . (‘Number . ‘Number)) . ‘Boolean)] 🡺 (-> (\* . (‘Number . ‘Number)) . ‘Boolean)

🡺’Boolean

🡺’Boolean

We got ‘Boolean, as expected.

1. (get-constructor (make-proc-te (make-tuple-te (list ‘Number ‘Number)), ‘Boolean)) = ‘->

The invariant makes sure the procedure parameter tuple is indeed a procedure type expression.

So, we want to prove that:

applicative-eval[(get-constructor (make-proc-te (make-tuple-te (list ‘Number ‘Number)), ‘Boolean))] 🡺 ‘->

applicative-eval[(get-constructor (make-proc-te (\* . (‘Number ‘Number)), ‘Boolean))] 🡺

applicative-eval[(get-constructor (-> (\* ‘Number ‘Number) . ‘Boolean))] 🡺

applicative-eval[(-> (\* ‘Number ‘Number) . ‘Boolean)] 🡺(-> (\* ‘Number ‘Number) . ‘Boolean)

applicative-eval[get-constructor] 🡺<Closure:(lambda (te)(if (composite? te) (car te) te)))

applicative-eval[(if (composite? [(-> (\* ‘Number ‘Number) . ‘Boolean)) (car [(-> (\* ‘Number ‘Number) . ‘Boolean)) [(-> (\* ‘Number ‘Number) . ‘Boolean))] 🡺

applicative-eval[if] 🡺 <if:primitive-procedure>

applicative-eval[(composite? [(-> (\* ‘Number ‘Number) . ‘Boolean))] 🡺

applicative-eval[[(-> (\* ‘Number ‘Number) . ‘Boolean)] 🡺 [(-> (\* ‘Number ‘Number) . ‘Boolean)

applicative-eval[composite?] 🡺 <Closure: (lambda (te) (or (procedure? te) (tuple? te)))>

applicative-eval[(or (procedure? [(-> (\* ‘Number ‘Number) . ‘Boolean)) (tuple? [(-> (\* ‘Number ‘Number) . ‘Boolean)))] 🡺

… 🡺#t

applicative-eval[(car [(-> (\* ‘Number ‘Number) . ‘Boolean))]

applicative-eval[car] 🡺<car:primitive-procedure>

applicative-eval[[(-> (\* ‘Number ‘Number) . ‘Boolean)] 🡺 [(-> (\* ‘Number ‘Number) . ‘Boolean)

🡺’->

🡺’->

🡺’->

We got ‘->, as expected.

1. Type: [T -> Boolean]

4.a.

1. get-variables(make-sub((list ‘T1 ‘T2) (list ‘Number ‘Boolean))) = (list ‘T1 ‘T2)
2. get-tes(make-sub((list ‘T1 ‘T2) (list ‘Number ‘Boolean))) = (list ‘Number ‘Boolean)
3. get-expression-of-variable(make-sub((list ‘T1 ‘T2) (list ‘Number ‘Boolean)) ‘T2) = ‘Boolean
4. get-expression-of-variable(

make-sub((list ‘T1 ‘T2) (list ‘Number ‘Boolean)) ‘T3) = ‘Empty

1. substitution-application(

make-sub((list ‘T1 ‘T2) (list ‘Number ‘make-tuple-te(list ‘Number ‘Boolean))) make-proc(make-tuple-te(list ‘T2) ‘T1)) = (-> (\* Number Boolean) Number)