Supervision 1

1. An author writes:

   "Most successful language design efforts share three important characteristics:

   - Motivating Application: The language was designed so that a specific kind of program could be written more easily.
   - ...
   - ...

   I discuss the merits and/or shortcomings of the statement that successful language design efforts share a Motivating Application.

   While many Theoreticians would like for programming languages to be adopted due to beauty and elegance, it's undeniable that any successful programming language must (by definition) have *people who use it.* Programmers will only adopt a programming language if there is a reason to do so. Why would I learn a new language and swap my codebase into a new language without a convincing reason. Any successful language must do something *so much better* than any existing language such that programmers are willing to put money into transferring.

   For example, Rust is an up-and-coming language which attempts to exploit a "gap in the market" of concurrent, high-performance, high-level programming languages. For the last 10 years, CPUs have had multicore and multithreaded parallelism which has been largely unexploited by modern programming languages. The mainstream languages which are able to properly exploit parallelism require explicit programmer input and management. This is analogous to manging which variables are stored in registers; highly impractical and it's impossible to determine the optimal choices without thorough investigation. Rust attempts to bridge this gap by providing memory safety, concurrency and performance by default. As a result, it has been gaining popularity despite having no strong theoretical underpinnings.

   Python is an example of a language with a strong motivating application; but no theoretical underpinnings or abstract machine. Python was designed as a general purpose programming language which was as logically close to human intuition as possible. It was initially targeted at mathematicians and data scientists. However, the simplicity and total abstraction from implementation (even at the cost of efficiency) was found to be popular so the target audience was broadened. Despite having no abstract machine and very few theoretical underpinnings, Python has become one of the most popular languages.

   Javascript is the classical example of a language with *only* a motivating application: writing web programs. Previously, programs on the web were only supported through Java applets. These had security holes and major browsers were hesitant to support them. Javascript was famously designed and written in 10 days; and its success has been to great frustration of theoretical computer scientists around the world. It has a weak type system and ugly syntax. However, at the time it was created there was a great requirement for programming on the web – a requirement which Javascript fit.

2. Give two operational semantics for the following simple language, one with dynamic scoping, and one with lexical scoping:

   $$E ::= x \mid \textbf{fn}\ x \to E \mid EE$$

   You may assume an encoding for numbers, booleans, and syntax for `let...in....end` and `if...then...else` without providing one.

For both, I define a value, v as follows:

$$v ::= \textbf{int} \mid \textbf{bool} \mid \textbf{fn}\ x \to E \text{ if } \textbf{fv}(E) \subseteq \{x\}$$

- Lexical (static) Scoping

$$\frac{E_1 \to E_1'}{E_1\ E_2 \to E_1'\ E_2} \qquad \frac{E_2 \to E_2'}{(\textbf{fn}\ x \to E)E_2 \to (\textbf{fn}\ x \to E)E_2'}$$

$$\frac{}{(\textbf{fn}\ x \to E)v \to \{v/x\}E}$$

$$\frac{}{\{v/x\}x = v} \qquad \frac{}{\{v/x\}y = y}$$

$$\frac{}{\{v/x\}(\textbf{fn}\ y \to E) = \textbf{fn}\ y \to \{v/x\}E}$$

$$\frac{}{\{v/x\}(\textbf{fn}\ x \to E) = \textbf{fn}\ x \to E}$$

$$\frac{}{\{v/x\}(E\ E') = \{v/x\}E\ \{v/x\}E'}$$

- Dynamic Scoping

$$\frac{}{\langle(\textbf{fn}\ x \to E)v, \text{Env}\rangle \to \langle E, (x, v) :: \text{Env}\rangle}$$

$$\frac{\text{Env} = (y_1, v_1) :: \ldots (y_n, v_n) :: (x, v) :: \text{Env}' \qquad x \notin \{y_1, \ldots, y_n\}}{\langle x, \text{Env}\rangle \to \langle v, \text{Env}\rangle}$$

$$\frac{\langle E_1, \text{Env}\rangle \to \langle E_1', \text{Env}'\rangle}{\langle E_1\ v, \text{Env}\rangle \to \langle E_1'\ v, \text{Env}'\rangle}$$

$$\frac{\langle E_2, \text{Env}\rangle \to^* \langle v, \text{Env}'\rangle}{\langle E_1\ E_2, \text{Env}\rangle \to^* \langle E_1\ v, \text{Env}\rangle}$$

How does typing work with dynamic scoping?

3. Outline the key features that a language must have to be called object-oriented. Further, briefly discuss to what extent the programming languages Simula, Smalltalk, C++, and Java have them.

   The four main language concepts for object-oriented languages are:

   - Dynamic lookup

     Dynamic lookup means that which method to evaluate is selected dynamically at runtime based on the actual type of an object rather than its static compile-time type.

   - Abstraction

     Abstraction means that programs are written in such a way that the programmer interacts with an abstract interface whose *implementation details* are hidden from the programmer. This allows the user to use the interface without concern for the underlying implementation, and allows the creator of the class to change the implementation without affecting usage.

- Subtyping

  A subtype relation $T <: T'$ is a relation which specifies a hierarchy of types such that a subtype can be passed anywhere where a supertype is expected. This means the same code (i.e functions) can be reused with many different types.

- Inheritance

  There are two types of inheritance: type inheritance and code inheritance. Code inheritance is the ability for a new type to inherit methods and implementation details from another class. Type inheritance is the ability for a new class to be a subtype of an existing class.

I now discuss the support which each language provides for the various features of object-oriented languages:

- Simula

  As the original *object-oriented language*, Simula has support for most of the features. As is to be expected, not all the main features were deemed useful or thought of. Therefore, Simula does not support some critical features.

  Simula has dynamic lookup, subtyping and inheritance. However, abstraction is not supported. Any code can access all attributes and functions of any objects.

- Smalltalk

  Smalltalk was a language designed around Simula which attempted to overcome its shortcomings. Smalltalk had support for all four of the main features of object-oriented languages.

- C++

  C++ has optional dynamic lookup in the form of virtual functions. This gives the programmer a choice between highly efficient implementations and complicated language features (which require a runtime lookup in a vtable to establish which implementation should be called). In my personal opinion, this is the best of both worlds (although if I was designing a language, lookups would be dynamic by default).

  Like almost all modern programming languages, C++ does support abstraction – objects can have private attributes and present an interface which completely abstracts away from all implementation details. C++ has a unique feature which allows violation of abstraction – the `friend` keyword allows a specified class or function to see the private and protected variables.

  C++ has support for complex subtype relationships which can form a directed acyclic graph – in contrast to many other languages which restrict subtyping to a tree.

  C++ supports multiple inheritance – a powerful (if confusing) form of inheritance at its purest.

- Java

  Java has compulsory dynamic lookup. Java supports a restricted form of inheritance and subtyping. Java makes no distinction between code inheritance and type inheritance. Objects may only directly inherit from a single class. However, inheritance (both type and code) is transitive. This means java inheritance hierarchies form a tree. This is sufficient for most applications and makes code easier to understand.

Java has strong support for abstraction which is forced on the programmer – unlike C++ Java has no concept of a friend class.

4. Define the following parameter-passing mechanisms: pass-by-value, pass-by-reference, pass-by-value/result, and pass-by-name. Briefly comment on their merits and drawbacks.

   - Pass-by-value

     Parameters passed into functions are fresh copies of the parameter the callee passed. Any changes to the parameter inside the function will not be reflected outside the function.

     This is easy to reason about – although when passing large structures can become inefficient.

   - Pass-by-reference

     Aliases to parameters are passed into the function. Any changes to the parameter inside the function are immediately reflected outside the function.

     This means any variables and function calls only require copies of pointers – leading to greater efficiency. However, this can be illogical for the programmer. Call by reference requires a pointer indirection per parameter access and has a lower cache hit rate than other methods – since parameters are not all stored in the stack frame of the function being called.

   - Pass-by-value/result

     In pass-by value/result, a copy of the parameter is passed into the function. The function then copies this value and operates on it locally. When the function returns, this result is copied into the parameter.

     Pass-by-value/result will not affect variables if it crashes. This means erroneous programs will not corrupt parameters as happens in call-by-reference.

     However, pass-by-value requires two copies of every parameter. This makes it very inefficient to pass large objects.

   - Pass-by-name

     Parameters are passed into arguments as they appear at the calling site.

     Parameters are only evaluated if they are actually used.

     However, they may be evaluated many times. This can be inefficient and lead to confusing semantics if evaluation of parameters has side effects.

5. What is aliasing in the context of programming languages? Explain the contexts in which it arises and provide examples of the phenomenon. In what kinds of languages is it generally considered bad, and in what kinds of languages is it considered useful?

   Aliasing is where multiple variables refer to the same object. Thus editing one of them will affect the value of the other. It's considered very useful in pure functional programming languages – objects are immutable so if multiple variables refer to the same object then memory is saved.

   However, aliasing is often bad in imperative programming languages since it leads to confusing semantics.

6. Recall that Algol 60 has a primitive static type system. In particular, in Algol 60, the type of a procedure parameter does not include the types of its parameters. Thus, for instance, in the Algol 60 code

```
procedure P(procedure F)
begin F(true) end ;
```

the types of the parameters of the procedure parameter `F` are not specified. Explain why this piece of code is statically type correct. Explain also why a call `P(Q)` may produce a run-time type error, and exemplify your answer by exhibiting a declaration for `Q` with this effect. Why does this problem not arise in Standard ML?

In Algol 60, the type of a parameter is dependent only on its return type. This means that the arguments to parameters are not statically type-checked. As a result, many programs which statically type-check will fail at runtime. For example, it would pass type-checking to pass a function `Q` with type `int` $\to$ `int` to P.

7. Consider the Simula declarations

```
CLASS A
A CLASS B;
```

which has the effect of producing the subtype relation `B<:A`, and

```
REF(A) a
REF(B) b;
```

Recall that Simula uses the semantically incorrect principle that if `B<:A` then `REF(B)<:REF(A)` and consider now the following Simula code

```
PROCEDURE ASSIGNa(REF(A) x)
BEGIN x :- a END ;
ASSIGNa(b);
```

Does it statically type check? If so, will it cause a run-time type error? Justify your answers.

This program **does** statically type-check. However, it **will** cause a runtime type error.

The program determines that `REF(B) <: REF(A)` – therefore passing an object of type `REF(B)` to the procedure `ASSIGNa` is legal and passes static type-checking.

At runtime, `ASSIGNa` takes `b` as argument then tries to assign `a` in a location expecting type `REF(B)`. This will (ideally) fail dynamic type-checking or cause an error when `b` is read from.

8. What is the problem with making the tag optional in Pascal's variant records?

Variant types are disjoint sums. However, the tag is optional. If the tag is not present then runtime checks aren't possible – the record has no type information so the program cannot check whether the value being extracted is of the same type as the value which was inserted.

9. What is the type of the expression

```
fn a => fn b => fn c => (a b) (b c)
```

given by the SML interpreter? Explain how this is inferred.

$$((\alpha \to \beta) \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$$

The SML interpreter will unify types, inferring function types however keeping everything as general as possible.

Initially, it will infer the types:

$$\alpha \to \beta \to \gamma \to \delta$$

Next, the α is unified with $\beta \to \epsilon$

$$(\beta \to \epsilon) \to \beta \to \gamma \to \delta$$

Next, β is unified with $\gamma \rightarrow \zeta$:

$$((\gamma \rightarrow \zeta) \rightarrow \epsilon) \rightarrow (\gamma \rightarrow \zeta) \rightarrow \gamma \rightarrow \delta$$

Next, ϵ is unified with $\zeta \rightarrow \eta$

$$((\gamma \rightarrow \zeta) \rightarrow (\zeta \rightarrow \eta)) \rightarrow (\gamma \rightarrow \zeta) \rightarrow \gamma \rightarrow \delta$$

Next η is unified with δ:

$$((\gamma \rightarrow \zeta) \rightarrow (\zeta \rightarrow \eta)) \rightarrow (\gamma \rightarrow \zeta) \rightarrow \gamma \rightarrow \eta$$

Now, I rewrite to α-equivalence for clarity:

$$((\alpha \rightarrow \beta) \rightarrow (\beta \rightarrow \gamma)) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma$$

Because of the order of precedence, we can remove some bracketing:

$$((\alpha \rightarrow \beta) \rightarrow \beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma$$

10. For each of the following ML declarations, either justify their ability to be soundly used or give a program using them which would violate type safety:

    -
      ```
      exception poly of 'a;
      ```

      This cannot be soundly used. For example, the following program would break type safety:

      ```
      try (
              raise poly true
      ) with poly x -> x + 1;
      ```

    -
      ```
      val ml = ref [];
      ```

      This cannot be soundly used. It's type is not known when it is instantiated. Furthermore, it cannot be used polymorphically since after the first usage we may be inserting multiple types into the same list. OCaml bypasses this by "weak" types – types which are not yet known but will become bound after their first usage.