Extending the Imperative Model

We have seen so far that the traditional imperative model of computation has as its computational elements "statements" or "commands" that alter the program *state*. As expected, these include an

- an identity command — `skip`,
- assignment as the basic operation — `x := e`,
- sequential composition — $c_1; c_2$,
- conditionals — if b then c_1 else c_2 fi — and
- repetition — while b do c_1 od

But what are the basic data on which these operations act? The "program state". Now so far we have treated the program state as a table, i.e., a mapping from variables to values (answers). However, the reality underlying this model is that the program state can be decomposed into two mappings, the first a finite domain function (called a "binding") $\ell : \mathcal{X} \rightharpoonup_{fin} Loc$ from variables to a set of *locations* Loc, and another finite domain function (called a "store") $\sigma : Loc \rightharpoonup_{fin} Ans$ from locations to values (answers). So what we saw as a table γ is actually their composition $\ell\sigma : \mathcal{X} \rightharpoonup_{fin} Ans$.

The reason to do so is that in the imperative model, during the course of execution of a command, the bindings *do not change*; it is only the store that changes. That is, the memory location with which a variable is associated remains the same; it's the *contents* of the locations that are changed by executing a command.

In forming the composition between the binding $\ell : \mathcal{X} \rightharpoonup_{fin} Loc$ and the store $\sigma : Loc \rightharpoonup_{fin} Ans$, there are several mismatches that can happen.

- One, it is possible that for some identifier (i.e., program variable) x, the expected location $\ell(x)$ does *not* exist. In this case we have a situation called a "dangling reference". That is x names a non-existent location.
- Two, it is possible that $\ell(x) = l \in Loc$, but the store at that location, i.e., $\sigma(l)$ is *not* defined. In this case, we have an uninitialised variable. Sometimes, there is an unknown value (answer) in that location, about which we know nothing.
- Three, there is a location $l \in Loc$ which is *not* in $range(\ell)$ but $\ell \in dom(\sigma)$. Such a location is not named by any program variable, and therefore cannot be accessed by the program commands; however, it contains a value (answer). Such a location is called "*garbage*".

All these situations can potentially occur in languages or language implementations where the allocation and deallocation of memory is not carefully managed.
- The presence of dangling references ("null pointer references") can be very dangerous and has resulted in several software errors, including some quite spectacular disasters.
- The presence of uninitialised variables is also a worry, since the location may contain an indeterminate value, and a calculation may be erroneous as a result.

- Garbage is never pleasant to have around, since it can slow down a computation, and is a waste of resources. However, it does not as such pose a threat to the correctness of a program, unless the program has to run within strictly bounded resources.

- "Allocation" involves creating a fresh memory location, and binding it to a variable. It is good practice to also initialise the newly created memory location with an appropriate value (answer).
- Allocation can be either on "stack" or on "heap". Stack allocation has the advantage of being automatically managed — we allocate the memory and create a binding, and when the binding is no longer needed, we move down the stack pointer. Thus the life of the location coincides with the duration of the binding. Heap allocation is preferred when the size of the data is not known (e.g., when we create a list of arbitrary length), and when the lifetime of the allocation need not coincide with that of the binding. Many languages have explicit operations to allocate and deallocate memory on heap. For example, C has the operations `malloc()` and `free()`

Rephrasing the operational rules, we can write:

- $\ell \vdash \sigma -[skip]\rightarrow \sigma$
- $$\frac{\ell\sigma \vdash e \Longrightarrow a}{\ell \vdash \sigma -[x := e]\rightarrow \sigma[\ell(x) \mapsto a]}$$
- $$\frac{\ell \vdash \sigma -[c_1]\rightarrow \sigma_1 \qquad \ell \vdash \sigma_1 -[c_2]\rightarrow \sigma_2}{\ell \vdash \sigma -[c_1; c_2]\rightarrow \sigma_2}$$
- $$\frac{\ell\sigma \vdash b \Longrightarrow T \qquad \ell \vdash \sigma -[c_1]\rightarrow \sigma_1}{\ell \vdash \sigma -[\text{if } b \text{ then } c_1 \text{ else } c_2 \text{ fi}]\rightarrow \sigma_1}$$
- $$\frac{\ell\sigma \vdash b \Longrightarrow F \qquad \ell \vdash \sigma -[c_2]\rightarrow \sigma_2}{\ell \vdash \sigma -[\text{if } b \text{ then } c_1 \text{ else } c_2 \text{ fi}]\rightarrow \sigma_2}$$
- $$\frac{\ell\sigma \vdash b \Longrightarrow F}{\ell \vdash \sigma -[\text{while } b \text{ do } c_1 \text{ od}]\rightarrow \sigma}$$
- $$\frac{\ell\sigma \vdash b \Longrightarrow T \qquad \ell \vdash \sigma -[c_1]\rightarrow \sigma_1 \qquad \ell \vdash \sigma_1 -[\text{while } b \text{ do } c_1 \text{ od}]\rightarrow \sigma_2}{\ell \vdash \sigma -[\text{while } b \text{ do } c_1 \text{ od}]\rightarrow \sigma_2}$$

Principle of Qualification in the Imperative paradigm

Recall that the Principle of Qualification stated that any syntactic category can be qualified by a definition.

We have so far seen definitions of the form **def** $x = e$, where an identifier (variable) x is bound to a canonical answer obtained as the result of evaluation (calculation) of e with respect to an environment (table). In a language such as Pascal or Algol60, such a definition is called a "constant definition".

During the course, we have also made type definitions in languages such as OCaml.

Both constant definitions and type definitions are present in languages like Pascal or C, usually in that order (since definitions of array types often required a size, which is specified as a constant).

In OCaml and in our toy languages we have encountered function definitions.

However in an imperative language — whether the toy language above or widely used languages such as Algol, Pascal, C, C++, C#, Java, etc. — we encounter a very common form of definition, namely the *declaration of (local) variables*, prior to a command. For example, in Pascal, we have program *blocks* such as

```
var
       x, y: integer
begin
     readLn(input, x);
     y := x*x;
     writeLn(output, y)
end
```

or in C,

```
{
      int x, y;
      scanf("%d", &x);
      y = x*x;
      printf("%d\n", y);
}
```

These *blocks* in so-called block-structured languages are a consequence of the Principle of Qualification — a variable declaration such as "`x, y: integer`" in Pascal or "`int x, y;`" in C is a form of definition, where the variable identifiers are being bound to newly created *locations,* and these are local declarations that qualify the subsequent commands delimited between the **`begin-end`** markers or **`{-}`** braces.

The types mentioned in these variable declarations are important in these languages for another reason — they indicated the amount of memory to allocate for each variable.

The listing of multiple variables having the same type is merely a convenience for readability. In a simplified form, we could have written "`x: integer; y: integer`". It is a matter of taste whether you prefer the Pascal style (where the "<*variable*> : <*type*>" can be read analogous to set membership) or the C style (where the type is written first, which can be seen similar to a predicate notation).

<u>The Principle of Qualification allows us to extend the syntax of commands to include blocks.</u>

Note that new bindings are being created in these declarations. Thus we can present an (incremental) elaboration rule for variable declarations as follows:

$$\ell, \sigma = [x_1 : \tau_1, \ldots, x_n : \tau_n] \Rightarrow \{x_1 \mapsto l_1, \ldots x_n \mapsto l_n\}, \{l_1 \mapsto \perp, \ldots, l_n \mapsto \perp\}$$

where $l_1, \ldots, l_n$ are all distinct locations and fresh with respect to $range(\ell) \cup dom(\sigma)$.

Now we can state the operational rule for a block (i.e., a command qualified with local variable definitions

$$\frac{\ell, \sigma \; =[x_1 : \tau_1, \ldots, x_n : \tau_n]\Rightarrow \; \ell^+, \sigma^+ \qquad \ell[\ell^+] \vdash \sigma[\sigma^+] \; -[c]\rightarrow \; \sigma'}{\ell \vdash \sigma \; -[\textbf{var } x_i : \tau_i \textbf{ begin } c \textbf{ end}]\rightarrow \; \sigma'|_{dom(\sigma)}}$$

Here, the variable declarations create new bindings (written as ℓ^+) and the corresponding store for the newly allocated locations (written as σ^+). Then we execute the command in the block to obtain a resulting store σ', which is then *pruned* removing all the newly allocated locations. (This is specified by writing $\sigma'|_{dom(\sigma)}$, the *restriction* of σ' to the domain of σ). Note that on exiting the block the bindings are restored to the original.

Principle of Abstraction in the Imperative paradigm

The abstractions of commands are called *procedures*. (In C, these are loosely called "functions", but these are really value-returning procedures, since they can have side-effects on global variables.) The Principle says that *procedure calls* can be included in the syntax of commands.

The syntax of a procedure definition may be presented as:

```
procedure p (x1 : τ1, ..., xn : τn) ;
    var y1 : τ'1, ..., ym : τ'm;
    begin c end
```

This is a definitional form where the name p is being given to the command (block) abstract. Here $x_1 : \tau_1, \ldots, x_n : \tau_n$ constitutes the list of *formal parameters* of the procedure, and $y_1 : \tau'_1, \ldots, y_m : \tau'_m$ the list of *local variables* of the procedure. The body of the procedure is the command c, which can operate over the all the variables $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$ as well as any "global" variables declared outside the procedure.

The syntax for an invocation of this procedure of the form $p(e_1, \ldots, e_n)$, and this can appear wherever any command can appear. We refer to the tuple of expressions $(e_1, \ldots, e_n)$ as the *actual arguments* to the procedure in that call.

Note that most imperative languages do not consider procedural abstracts as first-class citizens (so the space of defined procedures is distinct from the syntactic space of commands, though commands which now include procedure calls).

For a procedure definition as given above, the operational semantics for a "call-by-value" procedural call is as follows:

$$\frac{\ell, \sigma \vdash e_1 \Longrightarrow a_1 \ \ldots \ \ell, \sigma \vdash e_n \Longrightarrow a_n \qquad \ell[\{x_i \mapsto l_i\}_{i=1}^{n}] \vdash \sigma[\{l_i \mapsto a_i\}_{i=1}^{n}] \dashv[\textbf{var } y_j : \tau_j' \textbf{ begin } c \textbf{ end}]\rightarrow \sigma'}{\ell \vdash \sigma \dashv[p(e_1, \ldots, e_n)]\rightarrow \sigma'|_{dom(\sigma)}}$$

where $l_1, \ldots, l_n$ are all distinct locations and fresh with respect to $range(\ell) \cup dom(\sigma)$.
Note that we execute the body of the procedure as a block, and restore the bindings to those that prevailed before the procedure call, but retains the effects on the store on the original locations, pruning away only the newly allocated locations, whether for the actual arguments or those for local variables in the body of the procedure.

Principle of Correspondence in the Imperative paradigm

The Principle of Correspondence in the imperative paradigm suggests that any *mechanism* we use for implementing parameter passing should be the same as that for local variables in the block that is the body of a procedure.

Call by Reference Parameters

Consider the Pascal procedure, intended to swap the contents of two variables.

```
procedure swap(x, y: integer);
    var
     temp: integer;
    begin
     temp := x;
     x := y;
     y := temp
    end;
```

written in C as

```
void swap(int x, y);
{
     int temp;

     temp = x;
     x = y;
     y = temp;
}
```

If we call this procedure on two integer variables, say `a` and `b`, by writing `swap(a, b)`, we find to our dismay that the contents of the two variables do not get interchanged.

This is because these variables are being *passed by value.* That is, the arguments `a` and `b` are evaluated, and their results are *copied* into fresh locations (on the stack). A third location is allocated for `temp`. Note that by the Principle of Correspondence, the same allocation mechanism is used for both the formal parameters (`x, y`) and the local variable (`temp`) — in this case, allocation of space on the stack is planned in the machine code generated for the procedure `swap`.

Now, within the procedure call, the contents of these three locations are changed as specified. However, all these three locations are deallocated (by popping the stack). However, the contents of the argument variables `a` and `b` are unaffected.

Note however, if we had inlined the body of the function, substituting the actual argument names for the formal parameters, then execution would have indeed swapped their contents:

```
temp := a;
a := b;
b := temp
```

So one has to device a *different* parameter-passing mechanism for arguments whose contents we wish to change. Such a parameter-passing mechanism is already suggestive when one wants to pass large data structures such as arrays as arguments to a function or procedure, or if the size of the argument is indeterminate, for example a list of variable length.

So instead of passing the results of evaluating the arguments, i.e., the contents of variables `a` and `b`, we pass the *locations* to which variables `a` and `b` refer. This difference is indicated in the header of the procedure declaration in Pascal:

```
procedure swap(var x, y: integer);
```

The **`var`** in the parameter declaration indicates to the compiler that the arguments' locations have to be passed. Now, the compiler will be informed that the arguments

In C, a slightly different approach is taken: *parameter-passing is always call-by-value.* So if a reference (address/location/pointer) is to be passed — as is necessary in this case — we pass the address (location) to which a variable refers. That is we call `swap(&a, &b)`, i.e., with the addresses `&a` and `&b` of the variables `a` and `b`. The procedure definition needs to be changed:

```
void swap((int *) x, y);
{
      int temp;

      temp = *x;
      *x = *y;
      *y = temp;
}
```

Note that as the procedure is executing, the contents of the actual arguments change as the procedure executes. So there isn't a pleasant modular aspect to procedure execution.

(This also has security implications — if for some reason, control has to abruptly return from the procedure call to the calling routine, there may be inconsistent changes to the "global variables". Even with normal return, if the stack can be corrupted somehow, by writing into parts of it below the logical boundary of the procedure, one can cause induce bugs by "stack smashing", where by over-writing the return address, control "returns" to a location where a malicious program begins to run.)