Chapter 2

Types, Values, and Effects

2.1 Evaluation and Execution

Most familiar programming languages, such as C or Java, are based on an *imperative* model of computation. Programs are thought of as specifying a sequence of *commands* that modify the memory of the computer. Each step of execution examines the current contents of memory, performs a simple computation, modifies the memory, and continues with the next instruction. The individual commands are executed for their *effect* on the memory (which we may take to include both the internal memory and registers and the external input/output devices). The progress of the computation is controlled by evaluation of expressions, such as boolean tests or arithmetic operations, that are executed for their value. Conditional commands branch according to the value of some expression. Many languages maintain a distinction between expressions and commands, but often (in C, for example) expressions may also modify the memory, so that even expression evaluation has an effect.

Computation in ML is of a somewhat different nature. The emphasis in ML is on computation by *evaluation of expressions*, rather than *execution of commands*. The idea of computation is as a generalization of your experience from high school algebra in which you are given a polynomial in a variable x and are asked to calculate its value at a given value of x. We proceed by "plugging in" the given value for x, and then, using the rules of arithmetic, determine the value of the polynomial. The evaluation model of computation used in ML is based on the same idea, but rather than restrict ourselves to arithmetic operations on the reals, we admit a richer variety of values and a richer variety of primitive operations on them.

The evaluation model of computation enjoys several advantages over the more familiar imperative model. Because of its close relationship to mathematics, it is much easier to develop mathematical techniques for reasoning about the behavior of programs. These techniques are important tools for helping us to ensure that programs work properly without having to resort to tedious testing and debugging that can only show the presence of errors, never their absence. Moreover, they provide important tools for documenting the reasoning that went into the formulation of a program, making the code easier to understand and maintain.

What is more, the evaluation model subsumes the imperative model as a special case. Execution of commands for the effect on memory can be seen as a special case of evaluation of expressions by introducing primitive operations for allocating, accessing, and modifying memory. Rather than forcing all aspects of computation into the framework of memory modification, we instead take expression evaluation as the primary notion. Doing so allows us to support imperative programming without destroying the mathematical elegance of the evaluation model for programs that don't use memory. As we will see, it is quite remarkable how seldom memory modification is required. Nevertheless, the language provides for storage-based computation for those few times that it is actually necessary.

2.2 The ML Computation Model

Computation in ML consists of evaluation of expressions. Each expression has three important characteristics:

- It may or may not have a *type*.
- It may or may not have a *value*.
- It may or may not engender an *effect*.

These characteristics are all that you need to know to compute with an expression.

The type of an expression is a description of the value it yields, should it yield a value at all. For example, for an expression to have type `int` is to say that its value (should it have one) is a number, and for an expression to have type `real` is to say that its value (if any) is a floating point number. In general we can think of the type of an expression as a "prediction" of the form of the value that it has, should it have one. Every expression is required to have at least one type; those that do are said to be *well-typed*. Those without a type are said to be *ill-typed*; they are considered ineligible for evaluation. The *type checker* determines whether or not an expression is well-typed, rejecting with an error those that are not.

A well-typed expression is evaluated to determine its value, if indeed it has one. An expression can fail to have a value because its evaluation never terminates or because it raises an exception, either because of a run-time fault such as division by zero or because some programmer-defined condition is signalled during its evaluation. If an expression has a value, the form of that value is predicted by its type. For example, if an expression evaluates to a value v and its type is `bool`, then v must be either `true` or `false`; it cannot be, say, `17` or `3.14`. The *soundness* of the type system ensures the accuracy of the predictions made by the type checker.

Evaluation of an expression might also engender an *effect*. Effects include such phenomena as raising an exception, modifying memory, performing input or output, or sending a message on the network. It is important to note that the type of an expression says nothing about its possible effects! An expression of type `int` might well display a message on the screen before returning an integer value. This possibility is not accounted for in the type of the expression, which classifies only its value. For this reason effects are sometimes called *side effects*, to stress that they happen "off to the side" during evaluation, and are not part of the value of the expression. We will ignore effects until chapter 13. For the time being we will assume that all expressions are *effect-free*, or *pure*.

2.2.1 Type Checking

What is a type? What types are there? Generally speaking, a type is defined by specifying three things:

- a *name* for the type,
- the *values* of the type, and
- the *operations* that may be performed on values of the type.

Often the division of labor into values and operations is not completely clear-cut, but it nevertheless serves as a very useful guideline for describing types.

Let's consider first the type of integers. Its name is `int`. The values of type `int` are the numerals `0`, `1`, `~1`, `2`, `~2`, and so on. (Note that negative numbers are written with a prefix tilde, rather than a minus sign!) Operations on integers include addition, `+`, subtraction, `-`, multiplication, `*`, quotient, `div`, and remainder, `mod`. Arithmetic expressions are formed in the familiar manner, for example, `3*2+6`, governed by the usual rules of precedence. Parentheses may be used to override the precedence conventions, just as in ordinary mathematical practice. Thus the preceding expression may be equivalently written as `(3*2)+6`, but we may also write `3*(2+6)` to override the default precedences.

The formation of expressions is governed by a set of *typing rules* that define the types of expressions in terms of the types of their constituent expressions (if any). The typing rules are generally quite intuitive since they are consistent with our experience in mathematics and in other programming languages. In their full generality the rules are somewhat involved, but we will sneak up on them by first considering only a small fragment of the language, building up additional machinery as we go along.

Here are some simple arithmetic expressions, written using *infix* notation for the operations (meaning that the operator comes between the arguments, as is customary in mathematics):

```
3
3 + 4
4 div 3
4 mod 3
```

Each of these expressions is well-formed; in fact, they each have type `int`. This is indicated by a *typing assertion* of the form *exp* : *typ*, which states that the expression *exp* has the type *typ*. A typing assertion is said to be *valid* iff the expression *exp* does indeed have the type *typ*. The following are all valid typing assertions:

```
3 : int
3 + 4 : int
4 div 3 : int
4 mod 3 : int
```

Why are these typing assertions valid? In the case of the value `3`, it is an axiom that integer numerals have integer type. What about the expression `3+4`? The addition operation takes two arguments, each of which must have type `int`. Since both arguments in fact have type `int`, it follows that the entire expression is of type `int`. For more complex cases we reason analogously, for example, deducing that `(3+4) div (2+3)`: `int` by observing that `(3+4)`: `int` and `(2+3)`: `int`.

The reasoning involved in demonstrating the validity of a typing assertion may be summarized by a *typing derivation* consisting of a nested sequence of typing assertions, each justified either by an axiom, or a typing rule for an operation. For example, the validity of the typing assertion `(3+7) div 5` : `int` is justified by the following derivation:

1. `(3+7)`: `int`, because
   (a) `3` : `int` because it is an axiom
   (b) `7` : `int` because it is an axiom
   (c) the arguments of `+` must be integers, and the result of `+` is an integer
2. `5` : `int` because it is an axiom
3. the arguments of `div` must be integers, and the result is an integer

The outermost steps justify the assertion `(3+4) div 5` : `int` by demonstrating that the arguments each have type `int`. Recursively, the inner steps justify that `(3+4)`: `int`.

2.2.2 Evaluation

Evaluation of expressions is defined by a set of *evaluation rules* that determine how the value of a compound expression is determined as a function of the values of its constituent expressions (if any). Since the value of an operator is determined by the values of its arguments, ML is sometimes said to be a *call-by-value* language. While this may seem like the only sensible way to define evaluation, we will see in chapter 15 that this need not be the case — some operations may yield a value *without* evaluating their arguments. Such operations are sometimes said to be *lazy*, to distinguish them from *eager* operations that require their arguments to be evaluated before the operation is performed.

An *evaluation assertion* has the form *exp*$\Downarrow$*val*. This assertion states that the expression *exp* has value *val*. It should be intuitively clear that the following evaluation assertions are valid.

```
5 ⇓ 5
2+3 ⇓ 5
(2+3) div (1+4) ⇓ 1
```

An evaluation assertion may be justified by an *evaluation derivation*, which is similar to a typing derivation. For example, we may justify the assertion `(3+7) div 5` $\Downarrow$ `2` by the derivation

1. `(3+7)` $\Downarrow$ `10` because
   (a) `3` $\Downarrow$ `3` because it is an axiom
   (b) `7` $\Downarrow$ `7` because it is an axiom
   (c) Adding `3` to `7` yields `10`.
2. `5` $\Downarrow$ `5` because it is an axiom
3. Dividing `10` by `5` yields `2`.

Note that is an axiom that a numeral evaluates to itself; numerals are fully-evaluated expressions, or *values*. Second, the rules of arithmetic are used to determine that adding `3` and `7` yields `10`.

Not every expression has a value. A simple example is the expression `5 div 0`, which is undefined. If you attempt to evaluate this expression it will incur a run-time error, reflecting the erroneous attempt to find the number n that, when multiplied by 0, yields 5. The error is expressed in ML by raising an *exception*; we will have more to say about exceptions in chapter 12. Another reason that a well-typed expression might not have a value is that the attempt to evaluate it leads to an infinite loop. We don't yet have the machinery in place to define such expressions, but we will soon see that it is possible for an expression to *diverge*, or run forever, when evaluated.

2.3 Types, Types, Types

What types are there besides the integers? Here are a few useful *base* types of ML:

- *Type name*: `real`
  - *Values*: `3.14`, `2.17`, `0.1E6`, ...
  - *Operations*: `+`, `-`, `*`, `/`, `=`, `<`, ...
- *Type name*: `char`
  - *Values*: `#"a"`, `#"b"`, ...
  - *Operations*: `ord`,`chr`,`=`, `<`, ...
- *Type name*: `string`
  - *Values*: `"abc"`, `"1234"`, ...
  - *Operations*: `^`, `size`, `=`, `<`, ...
- *Type name*: `bool`
  - *Values*: `true`, `false`
  - *Operations*: `if` *exp* `then` exp_1 `else` exp_2

There are many, many (in fact, infinitely many!) others, but these are enough to get us started. (See V for a complete description of the primitive types of ML, including the ones given above.)

Notice that some of the arithmetic operations for real numbers are written the same way as for the corresponding operation on integers. For example, we may write `3.1+2.7` to perform a floating point addition of two floating point numbers. This is called *overloading*; the addition operation is said to be *overloaded* at the types `int` and `real`. In an expression involving addition the type checker tries to resolve which form of addition (fixed point or floating point) you mean. If the arguments are `int`'s, then fixed point addition is used; if the arguments are `real`'s, then floating addition is used; otherwise an error is reported.[1] Note that ML does *not* perform any implicit conversions between types! For example, the expression

[1]If the type of the arguments cannot be determined, the type defaults to `int`.

`3+3.14` is rejected as ill-formed! If you intend floating point addition, you must write instead `real(3)+3.14`, which converts the integer `3` to its floating point representation before performing the addition. If, on the other hand, you intend integer addition, you must write `3+round(3.14)`, which converts `3.14` to an integer by rounding before performing the addition.

Finally, note that floating point division is a *different* operation from integer quotient! Thus we write `3.1/2.7` for the result of dividing `3.1` by `2.7`, which results in a floating point number. We reserve the operator `div` for integers, and use `/` for floating point division.

The *conditional expression*

`if` *exp* `then` *exp*$_1$ `else` *exp*$_2$

is used to discriminate on a Boolean value. It has type *typ* if *exp* has type `bool` and both *exp*$_1$ and *exp*$_2$ have type *typ*. Notice that both “arms” of the conditional must have the same type! It is evaluated by first evaluating *exp*, then proceeding to evaluate either *exp*$_1$ or *exp*$_2$, according to whether the value of *exp* is `true` or `false`. For example,

```
if 1<2 then "less" else "greater"
```

evaluates to `"less"` since the value of the expression `1<2` is true.

Note that the expression

```
if 1<2 then 0 else (1 div 0)
```

evaluates to `0`, even though `1 div 0` incurs a run-time error. This is because evaluation of the conditional proceeds *either* to the `then` clause *or* to the `else` clause, depending on the outcome of the boolean test. Whichever clause is evaluated, the other is simply discarded without further consideration.

Although we may, in fact, test equality of two boolean expressions, it is rarely useful to do so. Beginners often writen conditionals of the form

`if` *exp* `= true then` *exp*$_1$ `else` *exp*$_2$`.`

But this is equivalent to the simpler expression

`if` *exp* `then` *exp*$_1$ `else` *exp*$_2$`.`

Similarly, rather than write

`if` *exp* `= false then` *exp₁* `else` *exp₂*,

it is better to write

`if not` *exp* `then` *exp₁* `else` *exp₂*

or, better yet, just

`if` *exp* `then` *exp₂* `else` *exp₁*.

2.4 Type Errors

Now that we have more than one type, we have enough rope to hang ourselves by forming *ill-typed* expressions. For example, the following expressions are not well-typed:

```
size 45
#"1" + 1
#"2" ^ "1"
3.14 + 2
```

In each case we are "misusing" an operator with arguments of the wrong type.

This raises a natural question: is the following expression well-typed or not?

```
if 1<2 then 0 else ("abc"+4)
```

Since the boolean test will come out `true`, the `else` clause will never be executed, and hence need not be constrained to be well-typed. While this reasoning is sensible for such a simple example, in general it is impossible for the type checker to determine the outcome of the boolean test during type checking. To be safe the type checker "assumes the worst" and insists that both clauses of the conditional be well-typed, and in fact have the *same* type, to ensure that the conditional expression can be given a type, namely that of both of its clauses.

2.5 Sample Code

Here is the complete code for this chapter.

Chapter 3

Declarations

3.1 Variables

Just as in any other programming language, values may be assigned to variables, which may then be used in expressions to stand for that value. However, in sharp contrast to most familiar languages, *variables in ML do not vary*! A value may be *bound* to a variable using a construct called a *value binding*. Once a variable is bound to a value, it is bound to it for life; there is no possibility of changing the binding of a variable once it has been bound. In this respect variables in ML are more akin to variables in mathematics than to variables in languages such as C.

A type may also be bound to a *type constructor* using a *type binding*. A bound type constructor stands for the type bound to it, and can never stand for any other type. For this reason a type binding is sometimes called a *type abbreviation* — the type constructor stands for the type to which it is bound.[1]

A value or type binding introduces a "new" variable or type constructor, distinct from all others of that class, for use within its range of significance, or *scope*. Scoping in ML is *static*, or *lexical*, meaning that the range of significance of a variable or type constructor is determined by the program text, not by the order of evaluation of its constituent expressions. (Languages with *dynamic* scope adopt the opposite convention.) For the time being variables and type constructors have *global scope*, meaning that

[1]By the same token a value binding might also be called a *value abbreviation*, but for some reason it never is.

the range of significance of the variable or type constructor is the "rest" of the program — the part that lexically follows the binding. However, we will soon introduce mechanisms for limiting the scopes of variables or type constructors to a given expression.

3.2 Basic Bindings

3.2.1 Type Bindings

Any type may be given a name using a *type binding*. At this stage we have so few types that it is hard to justify binding type names to identifiers, but we'll do it anyway because we'll need it later. Here are some examples of type bindings:

```
type float = real
type count = int and average = real
```

The first type binding introduces the type constructor `float`, which subsequently is synonymous with `real`. The second introduces *two* type constructors, `count` and `average`, which stand for `int` and `real`, respectively.

In general a type binding introduces one or more new type constructors *simultaneously* in the sense that the definitions of the type constructors may not involve any of the type constructors being defined. Thus a binding such as

```
type float = real and average = float
```

is nonsensical (in isolation) since the type constructors `float` and `average` are introduced simultaneously, and hence cannot refer to one another.

The syntax for type bindings is

```
type $tycon_1$ = $typ_1$
and ...
and $tycon_n$ = $typ_n$
```

where each $tycon_i$ is a type constructor and each typ_i is a type expression.

3.2.2 Value Bindings

A value may be given a name using a *value binding*. Here are some examples:

```
val m : int = 3+2
val pi : real = 3.14 and e : real = 2.17
```

The first binding introduces the variable `m`, specifying its type to be `int` and its value to be `5`. The second introduces two variables, `pi` and `e`, simultaneously, both having type `real`, and with `pi` having value `3.14` and `e` having value `2.17`. Notice that a value binding specifies both the type and the value of a variable.

The syntax of value bindings is

val var_1 : typ_1 = exp_1
and ...
and var_n : typ_n = exp_n,

where each var_i is a variable, each typ_i is a type expression, and each exp_i is an expression.

A value binding of the form

val *var* : *typ* = *exp*

is type-checked by ensuring that the expression *exp* has type *typ*. If not, the binding is rejected as ill-formed. If so, the binding is evaluated using the *bind-by-value* rule: first *exp* is evaluated to obtain its value *val*, then *val* is bound to *var*. If *exp* does not have a value, then the declaration does not bind anything to the variable *var*.

The purpose of a binding is to make a variable available for use within its scope. In the case of a type binding we may use the type variable introduced by that binding in type expressions occurring within its scope. For example, in the presence of the type bindings above, we may write

```
val pi : float = 3.14
```

since the type constructor `float` is bound to the type `real`, the type of the expression `3.14`. Similarly, we may make use of the variable introduced by a value binding in value expressions occurring within its scope.

Continuing from the preceding binding, we may use the expression

```
sin pi
```

to stand for `0.0` (approximately), and we may bind this value to a variable by writing

```
val x : float = sin pi
```

As these examples illustrate, type checking and evaluation are *context dependent* in the presence of type and value bindings since we must refer to these bindings to determine the types and values of expressions. For example, to determine that the above binding for `x` is well-formed, we must consult the binding for `pi` to determine that it has type `float`, consult the binding for `float` to determine that it is synonymous with `real`, which is necessary for the binding of `x` to have type `float`.

The rough-and-ready rule for both type-checking and evaluation is that a bound variable or type constructor is implicitly *replaced* by its binding prior to type checking and evaluation. This is sometimes called the *substitution principle* for bindings. For example, to evaluate the expression `cos x` in the scope of the above declarations, we first replace the occurrence of `x` by its value (approximately `0.0`), then compute as before, yielding (approximately) `1.0`. Later on we will have to refine this simple principle to take account of more sophisticated language features, but it is useful nonetheless to keep this simple idea in mind.

3.3 Compound Declarations

Bindings may be combined to form *declarations*. A binding is an atomic declaration, even though it may introduce many variables simultaneously. Two declarations may be combined by *sequential composition* by simply writing them one after the other, optionally separated by a semicolon. Thus we may write the declaration

```
val m : int = 3+2
val n : int = m*m
```

which binds `m` to `5` and `n` to `25`. Subsequently, we may evaluate `m+n` to obtain the value `30`. In general a sequential composition of declarations has the form $dec_1 \ldots dec_n$, where n is at least 2. The scopes of these declarations are *nested* within one another: the scope of dec_1 includes $dec_2, \ldots, dec_n$, the scope of dec_2 includes $dec_3, \ldots, dec_n$, and so on.

One thing to keep in mind is that *binding is not assignment*. The binding of a variable never changes; once bound to a value, it is always bound to that value (within the scope of the binding). However, we may *shadow* a binding by introducing a second binding for a variable within the scope of the first binding. Continuing the above example, we may write

```
val n : real = 2.17
```

to introduce a new variable `n` with both a different type and a different value than the earlier binding. The new binding eclipses the old one, which may then be discarded since it is no longer accessible. (Later on, we will see that in the presence of higher-order functions shadowed bindings are not always discarded, but are preserved as private data in a closure. One might say that old bindings never die, they just fade away.)

3.4 Limiting Scope

The scope of a variable or type constructor may be delimited by using `let` expressions and `local` declarations. A `let` expression has the form

`let` *dec* `in` *exp* `end`

where *dec* is any declaration and *exp* is any expression. The scope of the declaration *dec* is limited to the expression *exp*. The bindings introduced by *dec* are discarded upon completion of evaluation of *exp*.

Similarly, we may limit the scope of one declaration to another declaration by writing

`local` *dec* `in` *dec′* `end`

The scope of the bindings in *dec* is limited to the declaration *dec′*. After processing *dec′*, the bindings in *dec* may be discarded.

The value of a `let` expression is determined by evaluating the declaration part, then evaluating the expression relative to the bindings introduced by the declaration, yielding this value as the overall value of the `let` expression. An example will help clarify the idea:

```
let
    val m : int = 3
    val n : int = m*m
in
    m*n
end
```

This expression has type `int` and value `27`, as you can readily verify by first calculating the bindings for `m` and `n`, then computing the value of `m*n` relative to these bindings. The bindings for `m` and `n` are local to the expression `m*n`, and are not accessible from outside the expression.

If the declaration part of a `let` expression eclipses earlier bindings, the ambient bindings are restored upon completion of evaluation of the `let` expression. Thus the following expression evaluates to `54`:

```
val m : int = 2
val r : int =
    let
        val m : int = 3
        val n : int = m*m
    in
        m*n
    end * m
```

The binding of `m` is temporarily overridden during the evaluation of the `let` expression, then restored upon completion of this evaluation.

3.5 Typing and Evaluation

To complete this chapter, let's consider in more detail the context-sensitivity of type checking and evaluation in the presence of bindings. The key ideas are:

- Type checking must take account of the declared type of a variable.
- Evaluation must take account of the declared value of a variable.

This is achieved by maintaining *environments* for type checking and evaluation. The *type environment* records the types of variables; the *value* *environment* records their values. For example, after processing the compound declaration

```
val m : int = 0
val x : real = Math.sqrt(2.0)
val c : char = #"a"
```

the type environment contains the information

```
val m : int
val x : real
val c : char
```

and the value environment contains the information

```
val m = 0
val x = 1.414
val c = #"a"
```

In a sense the value declarations have been divided in "half", separating the type from the value information.

Thus we see that value bindings have significance for both type checking and evaluation. In contrast type bindings have significance only for type checking, and hence contribute only to the type environment. A type binding such as

```
type float = real
```

is recorded in its entirety in the type environment, and no change is made to the value environment. Subsequently, whenever we encounter the type constructor `float` in a type expression, it is replaced by `real` in accordance with the type binding above.

In chapter 2 we said that a typing assertion has the form *exp* : *typ*, and that an evaluation assertion has the form *exp* ⇓ *val*. While two-place typing and evaluation assertions are sufficient for *closed* expressions (those without variables), we must extend these relations to account for *open* expressions (those with variables). Each must be equipped with an *environment* recording information about type constructors and variables introduced by declarations.

Typing assertions are generalized to have the form

$typenv \vdash exp : typ$

where *typenv* is a *type environment* that records the bindings of type constructors and the types of variables that may occur in *exp*.[2] We may think of *typenv* as a sequence of specifications of one of the following two forms:

1. `type` *typvar* `=` *typ*
2. `val` *var* `:` *typ*

Note that the second form does *not* include the binding for *var*, only its type!

Evaluation assertions are generalized to have the form

$valenv \vdash exp \Downarrow val$

where *valenv* is a *value environment* that records the bindings of the variables that may occur in *exp*. We may think of *valenv* as a sequence of specifications of the form

`val` *var* `=` *val*

that bind the value *val* to the variable *var*.

Finally, we also need a new assertion, called *type equivalence*, that determines when two types are equivalent, relative to a type environment. This is written

$typenv \vdash typ_1 \equiv typ_2$

Two types are equivalent iff they are the same when the type constructors defined in *typenv* are replaced by their bindings.

The primary use of a type environment is to record the types of the value variables that are available for use in a given expression. This is expressed by the following axiom:

$\ldots \texttt{val}\ var : typ \ldots \vdash var : typ$

[2]The *turnstile* symbol, "$\vdash$", is simply a punctuation mark separating the type environment from the expression and its type.

In words, if the specification `val` *var* `:` *typ* occurs in the type environment, then we may conclude that the variable *var* has type *typ*. This rule glosses over an important point. In order to account for shadowing we require that the *rightmost* specification govern the type of a variable. That way re-binding of variables with the same name but different types behaves as expected.

Similarly, the evaluation relation must take account of the value environment. Evaluation of variables is governed by the following axiom:

$$\ldots\texttt{val}\ \mathit{var}\ \texttt{=}\ \mathit{val}\ldots \vdash \mathit{var} \Downarrow \mathit{val}$$

Here again we assume that the `val` specification is the rightmost one governing the variable *var* to ensure that the scoping rules are respected.

The role of the type equivalence assertion is to ensure that type constructors always stand for their bindings. This is expressed by the following axiom:

$$\ldots\texttt{type}\ \mathit{typvar}\ \texttt{=}\ \mathit{typ}\ldots \vdash \mathit{typvar} \equiv \mathit{typ}$$

Once again, the rightmost specification for *typvar* governs the assertion.

3.6 Sample Code

Here is the complete code for this chapter.