A simple calculator

Perhaps the simplest thing to start playing with is an integer calculator, i.e., integers and the four basic arithmetic operators, $+$, $-$, $\times$ and $/$ (integer division).

An **expression** may be built using these operations, integers and parentheses. For example $(2 + 3) \times (4 + 5)$ is an expression.

A **value** is an integer, therefore it is a special case of expression.

What a calculator does is to match an inputted expression with a value, usually called *the result*. For example, the above expression evaluates in 45.

This process is called **evaluation**.

Evaluation as rewriting expressions/Reductions

The evaluation of arithmetic expressions can be defined as a series of transformations leading to a value (the result). These transformations are called **rewrites** and a series of rewrites is a **reduction** or, more generally, a **computation**.

As we learnt in school long time ago, the way to perform these rewrites is to select a part of the current expression consisting of an operator and its arguments *as values*, then to replace this sub-expression by the operation result.

In the following reduction, let us underline the sub-expression to be rewritten and print in bold the result of each step:

$$(2+3)\times(\underline{4+5}) \rightarrow (\underline{2+3})\times \mathbf{9} \rightarrow \underline{\mathbf{5}\times 9} \rightarrow \mathbf{45}$$

Strategy of evaluation and rewrite systems

Note that there are different ways to choose the sub-expression. For example:

$$\underline{(2+3)} \times (4+5) \to \mathbf{5} \times \underline{(4+5)} \to \underline{5 \times \mathbf{9}} \to \mathbf{45}$$

It is usual to write, as a summary: $(2+3) \times (4+5) \xrightarrow{*} 45$ or $(2+3) \times (4+5) = 45$.

So, there are two concepts involved here:

1. the selection of a sub-expression made of an operator and its arguments as values,
2. the rewriting of such sub-expression into a value.

They are defined by a **rewrite system**.

Rewrite rules

A rewrite system is a set of **rewrite rules**.

Rewriting rules specify how to perform a rewrite.

For example, let us assume that we have an infinite set of rewriting rules, i.e., an infinite rewrite system, like:

$$
\begin{aligned}
0 + 0 &\to 0 \\
0 + 1 &\to 1 \\
\ldots &\to \ldots \\
124 - 57 &\to 67 \\
\ldots &\to \ldots
\end{aligned}
$$

Conditional rewrite rules and meta-variables

This rewrite system is not enough for the kind of expressions we want to evaluate, where it is possible to combine operations, like $(2 + 3) \times (4 + 5)$.

So let us extend our system with rules that specify the sub-expressions we can rewrite.

Consider the multiplication:

$$e_1 \times e_2 \to e_1' \times e_2 \quad \text{if } e_1 \to e_1'$$

where e_1, e_1' and e_2 denote any expression and are called **meta-variables**.

Instance of a rewrite rule

An **instance** of this rule is obtained when the meta-variables are replaced by expressions.

For example, $(2+3)\times(4+5)\to 5\times(4+5)$ is an instance since $2+3\to 5$. To understand why, replace e_1 by $2+3$, e_1' by 5 and e_2 by $4+5$.

Let us rewrite $((2+3)\times 6)\times(4+5)$. First, we see that the previous rule has the required shape, i.e., the left side of the arrow is $e_1\times e_2$ and if $e_1=(2+3)\times 6$ and $e_2=4+5$ then $e_1\times e_2$ is exactly the expression we wish to rewrite. So, we replace e_1 and e_2 by their associated expressions in the rule and we get an instance of it:

$$((2+3)\times 6)\times(4+5)\to e_1'\times(4+5)\quad \text{if } (2+3)\times 6\to e_1'$$

Therefore, we now know that we have to rewrite $(2+3)\times 6$.

Instance of a rewrite rule (cont)

We need another instance of the same rule. This time, let $e_1 = 2 + 3$ and $e_2 = 6$. We get

$$(2 + 3) \times 6 \to e_1' \times 6 \quad \text{if } 2 + 3 \to e_1'$$

It is important to understand that this latter e_1' is not the same as the former: they belong to two different instances of the same rule. The latter e_1' is actually easy to find, since we have all the rules for basic cases as $2 + 3 \to 5$. Hence, $e_1' = 5$ and the second instance is now complete:

$$(2 + 3) \times 6 \to 5 \times 6 \quad \text{if } 2 + 3 \to 5$$

Going backwards, this implies that the former e_1' is 5×6. So, the first instance is

$$((2 + 3) \times 6) \times (4 + 5) \to (5 \times 6) \times (4 + 5) \quad \text{if } 2 + 3 \to 5$$

Instance of a rewrite rule (cont)

Our rule for multiplication, $e_1 \times e_2 \to e_1' \times e_2$, actually depends on another, $e_1 \to e_1'$. It is usual to write them this way:

$$\frac{e_1 \to e_1'}{e_1 \times e_2 \to e_1' \times e_2} \langle \textsc{Mult}_1 \rangle$$

This is called an **inference rule**. Note that it has a name, $\langle \textsc{Mult}_1 \rangle$. The rewrite rule on the top, acting as a condition for the rule on the bottom, is called a **premise**. The rule at the bottom is called a **conclusion**.

A **substitution** is a mapping from meta-variables to expressions which applies to an inference rule or rewrite rule. For example, let σ be the following substitution: $\{e_1 \mapsto (2+3) \times 6; e_2 \mapsto 4+5\}$. This notation is equivalent to $\sigma(e_1) = (2+3) \times 6$ and $\sigma(e_2) = 4+5$.

Instance of a rewrite rule (cont)

Now we can make a clear definition of the notion of "instance of a rule":

> *An instance of a rule is a rule in which some meta-variables e_i are replaced by expressions $\sigma(e_i)$.*

For example, here is the instance of $\langle \textsc{Mult}_1 \rangle$ using previously defined substitution σ, and noted $\sigma\langle \textsc{Mult}_1 \rangle$:

$$\frac{(2+3) \times 6 \to e_1'}{((2+3) \times 6) \times (4+5) \to e_1' \times (4+5)}\ \sigma\langle \textsc{Mult}_1 \rangle$$

Instance of a rewrite rule (cont)

Now how do we find (a mapping for) e_1'?

Let us define another substitution $\sigma' = \{e_1 \mapsto 2 + 3; e_2 \mapsto 6\}$ and apply it:

$$\frac{2 + 3 \to e_1'}{(2 + 3) \times 6 \to e_1' \times 6}\ \sigma'\langle \textsc{Mult}_1 \rangle$$

Note that the meta-variable e_1' in $\sigma'\langle \textsc{Mult}_1 \rangle$ is not the same as in $\sigma\langle \textsc{Mult}_1 \rangle$: they should have a different mapping.

We have a basic rewrite rule $2 + 3 \to 5$, thus we extend σ' with $e_1' \mapsto 5$ and get

$$\frac{2 + 3 \to 5}{(2 + 3) \times 6 \to 5 \times 6}\ \sigma'\langle \textsc{Mult}_1 \rangle$$

Instance of a rewrite rule (cont)

Therefore we can extend σ with $e_1' \mapsto 5 \times 6$ and we get

$$\frac{(2+3) \times 6 \to 5 \times 6}{((2+3) \times 6) \times (4+5) \to (5 \times 6) \times (4+5)}\ \sigma\langle \textsc{Mult}_1 \rangle$$

We can summarise this rewrite using the two instances of $\langle \textsc{Mult}_1 \rangle$ as

$$\frac{\dfrac{2+3 \to 5}{(2+3) \times 6 \to 5 \times 6}\ \sigma'\langle \textsc{Mult}_1 \rangle}{((2+3) \times 6) \times (4+5) \to (5 \times 6) \times (4+5)}\ \sigma\langle \textsc{Mult}_1 \rangle$$

Instance of a rewrite rule (cont)

But what about the rewrite of an expression like $7 \times (4 + 5)$?

Let us define another substitution on $\langle \textsc{Mult}_1 \rangle$:
$\sigma' = \{e_1 \mapsto 7; e_2 \mapsto 4 + 5\}$.

The corresponding instance is

$$\frac{7 \to e_1'}{7 \times (4 + 5) \to e_1' \times (4 + 5)}\ \sigma' \langle \textsc{Mult}_1 \rangle$$

But there is no rule of shape "$v \to \ldots$", where v is *a value because a value is, by definition, an expression completely reduced.*

Completing the multiplication rule

This means that we need another rule for the evaluation of the *right* argument of $\times$:

$$\frac{e_2 \to e_2'}{e_1 \times e_2 \to e_1 \times e_2'} \langle \textsc{Mult}_2 \rangle$$

This inference rule says that we can rewrite a product by rewriting its right argument.

For example, here is the instance $\sigma' \langle \textsc{Mult}_2 \rangle$:

$$\frac{4 + 5 \to 9}{7 \times (4 + 5) \to 7 \times 9} \sigma' \langle \textsc{Mult}_2 \rangle$$

Strategy

But now, something interesting happens.

Consider again $(2 + 3) \times (4 + 5)$: both $\langle \textsc{Mult}_1 \rangle$ and $\langle \textsc{Mult}_2 \rangle$ can be instantiated by substitution $\sigma'' = \{e_1 \mapsto 2 + 3; e_2 \mapsto 4 + 5\}$:

$$\frac{2 + 3 \to 5}{(2 + 3) \times (4 + 5) \to 5 \times (4 + 5)}\ \sigma''\langle \textsc{Mult}_1 \rangle$$

$$\frac{4 + 5 \to 9}{(2 + 3) \times (4 + 5) \to (2 + 3) \times 9}\ \sigma''\langle \textsc{Mult}_2 \rangle$$

Strategy (cont)

This means that *our system does not specify the order of evaluation of* $\times$ *arguments.*

In other words, our rewrite system does not select only one sub-expression to rewrite at each step.

We need a **strategy** to complete it.

Finiteness and soundness of reductions

At this point, it is clear that these reductions satisfy the following properties:

- **Finiteness.** All the reductions are finite, i.e., all the computations terminate.
- **Soundness.** All the reductions of an expression end with the same value, if any.

About the second point, note indeed that *some expressions have no value*, as

$$\underline{(2+3)}/(4-4) \to \mathbf{5}/\underline{(4-4)} \to 5/\mathbf{0} \not\to$$

The reduction is finite but does not end with a value: this situation corresponds to an **error**, usually called **run-time error** in programming languages. Expression $5/0$ is **stuck**.

Another model of run-time errors

We can modify the calculator so that, in case of error (here, the only one is the division by zero), the result of a rewrite is a special value, called NaN (*Not a Number*). Then we add the following rules:

$$e/0 \to \mathsf{NaN} \quad \langle \textsc{DivZero} \rangle \qquad \mathsf{NaN}/e \to \mathsf{NaN} \quad \langle \textsc{Div-Err}_1 \rangle$$

$$e/\mathsf{NaN} \to \mathsf{NaN} \quad \langle \textsc{Div-Err}_2 \rangle \qquad \mathsf{NaN} \times e \to \mathsf{NaN} \quad \langle \textsc{Mult-Err}_1 \rangle$$

$$e \times \mathsf{NaN} \to \mathsf{NaN} \quad \langle \textsc{Mult-Err}_2 \rangle \qquad \mathsf{NaN} + e \to \mathsf{NaN} \quad \langle \textsc{Add-Err}_1 \rangle$$

$$e + \mathsf{NaN} \to \mathsf{NaN} \quad \langle \textsc{Add-Err}_2 \rangle \qquad \mathsf{NaN} - e \to \mathsf{NaN} \quad \langle \textsc{Sub-Err}_1 \rangle$$

$$e - \mathsf{NaN} \to \mathsf{NaN} \quad \langle \textsc{Sub-Err}_2 \rangle$$

Another model of run-time errors (cont)

The rationale of this system is that once an error occurs, no other computations (leading to integer values) are required and the reduction can end as soon as possible.

In this case:

$$(2+3)/(4-4) \to 5/(4-4) \to 5/0 \to \text{NaN}$$

or, even the shortest possible:

$$(2+3)/(4-4) \to (2+3)/0 \to \text{NaN}$$

Another model of run-time errors/Rephrasing soundness

The interesting phenomenon here is that, now, any reduction ends in a value. In the previous rewrite system, any reduction ends either in a value or a **stuck expression**, i.e., an expression that cannot be rewritten further.

With the new system, that explicitly specifies the reduction of the NaN error, the soundness property can be rephrased as

All the reductions of an expression end with the same value.

Obviously, the drawback of the new system is that it requires a lot of additional rules, and, as the system is augmented with new rules, new rules for the errors have to be added... or are forgotten by mistake.

Specifying the strategy

It is possible to force an order of evaluation on the arguments of $\times$. Assume we want to reduce the left argument first.

The idea is to change $\langle \textsc{Mult}_2 \rangle$ so that instead of e_1, which stands for any expression, we specify a value v:

$$\frac{e_2 \to e_2'}{v \times e_2 \to v \times e_2'} \langle \textsc{New-Mult}_2 \rangle$$

This way, it is not possible anymore to instantiate this rule with expression $(2 + 3) \times (4 + 5)$ because $2 + 3$ is not a value.

Specifying the strategy (cont)

The only possible reduction is:

$$\frac{2 + 3 \to 5}{(2 + 3) \times (4 + 5) \to 5 \times (4 + 5)}\ \sigma_1\langle \textsc{Mult}_1 \rangle$$

$$\frac{4 + 5 \to 9}{5 \times (4 + 5) \to 5 \times 9}\ \sigma_2\langle \textsc{New-Mult}_2 \rangle$$

where $\sigma_1 = \{e_1 \mapsto 2 + 3; e_2 \mapsto 4 + 5\}$ and $\sigma_2 = \{v \mapsto 5; e_2 \mapsto 4 + 5\}$.

Determinism

We have specified the strategy **in** the rewrite system.

Also, with $\langle \textsc{New-Mult}_2 \rangle$, given an expression, there is only one rule that can be applied (or none): this is sometimes called **determinism**, or **determinacy**:

Determinism. If $e_1 \to e_1'$ and $e_2 \to e_2'$ then $e_1' = e_2'$.

This is a stronger property than soundness, i.e., it implies soundness. Indeed, determinism implies that there is only one reduction from an expression to its value (if any).

Pragmatics of programming languages

In many programming languages, the order of evaluation of operator arguments is not specified, which usually means that there is a sentence in the official document which defines the language like: "The order of evaluation of the arguments of operators and functions is not specified."

Formally, this means that the underlying rewrite system is **non-deterministic**.

One notable exception is Java, which specifies that the arguments are always evaluated from left to right, i.e., following the order of writing.

Of course, if an order is specified, as in Java, the compilers of these languages must implement it. Otherwise one is chosen arbitrarily and may change from one release to another.

Rewrite systems as models of languages

By studying rewrite systems, we can learn the basic concepts involved in programming languages, because rewrite systems are extremely simple.

There are some languages, like Maude, whose programs actually are rewrite systems.

Some compilers transform the input program (called **source**) and output a program, called **byte-code**, which can be reduced by a rewrite system.

Rewrite systems as models of languages (cont)

This is the case of Java. This is why, in general, we need a **Java virtual machine** to run the program: this JVM performs the reductions according to a rewrite system.

Even if there is no byte-code, we always can interpret what the program does by studying a rewrite system which models it.

Sequentiality and parallelism

Some programming languages, like Ada, allow the programmer to specify that different parts of its program can actually be executed separately: this is called, in general, **parallelism**. This parallelism is possible only if the underlying rewrite system is non-deterministic.

Indeed, if all the strategies lead to the same value (soundness property), it is possible to reduce different parts of an expression in parallel if the rewrite system allows it.

Sequentiality and parallelism (cont)

For example, the first argument of $\times$ can be computed *at the same time* as its second argument. In the case of $(2+3)\times(4+5)$, this means that we can reduce in parallel $2+3$, with $\langle \textsc{Mult}_1 \rangle$, and $4+5$, with $\langle \textsc{Mult}_2 \rangle$, then 5×9 is finally rewritten in 45 (after **synchronisation** of the two reductions).