Order of evaluation in C

In case of C, the order of evaluation arguments of arithmetic operators and functions is not specified, it actually depends on the compiler and its version. Therefore it would be dangerous to rely on the order of evaluation in C.

Why is the order left unspecified? Simply because of possible **side-effects**. Imagine the following situation:

```
int n = 0;
int f () { n++; return n; };
int g () { n = 2; return n; };
int main () { return f() + g() + n; }
```

What is the result?

Boolean expressions

Let $\wedge$ represent the conjunction ("and"), $\vee$ the disjunction ("or"), $\neg$ the negation ("not"), and let the two values true and false.

The definition of the conjunction of values is very simple:

$$\text{true} \wedge \text{true} \to \text{true} \qquad \text{true} \wedge \text{false} \to \text{false}$$

$$\text{false} \wedge \text{true} \to \text{false} \qquad \text{false} \wedge \text{false} \to \text{false}$$

Boolean expressions made simpler

This system can even be made simpler, by means of meta-variables:

$$\text{true} \wedge \text{true} \to \text{true} \quad \langle \textsc{True} \rangle$$

$$\text{false} \wedge v \to \text{false} \quad \langle \textsc{False}_1 \rangle$$

$$v \wedge \text{false} \to \text{false} \quad \langle \textsc{False}_2 \rangle$$

where v denotes any value, i.e., either true or false.

Boolean expressions (cont)

So, if we define the rewrite of boolean expressions (not just the values true and false) as

$$\frac{e_1 \to e_1'}{e_1 \wedge\ e_2 \to e_1' \wedge\ e_2}\ \langle \wedge_1 \rangle \qquad \frac{e_2 \to e_2'}{e_1 \wedge\ e_2 \to e_1 \wedge\ e_2'}\ \langle \wedge_2 \rangle$$

then, for example,

$$\frac{\text{false} \wedge\ \text{true} \to \text{false} \quad \langle \textsc{False}_1 \rangle}{(\text{false} \wedge\ \text{true}) \wedge\ (\text{true} \wedge\ \text{true}) \to \text{false} \wedge\ (\text{true} \wedge\ \text{true})}\ \sigma_1 \langle \wedge_1 \rangle$$

where $\sigma_1 = \{e_1 \mapsto \text{false} \wedge \text{true};\ e_2 \mapsto \text{true} \wedge \text{true};\ e_1' \mapsto \text{false}\}$.

Boolean expressions (cont)

Then

$$\frac{\mathsf{true} \wedge \mathsf{true} \to \mathsf{true} \quad \langle \textsc{True} \rangle}{\mathsf{false} \wedge (\mathsf{true} \wedge \mathsf{true}) \to \mathsf{false} \wedge \mathsf{true}}\ \sigma_2\langle \wedge_2 \rangle$$

where $\sigma_2 = \{e_1 \mapsto \mathsf{false}; e_2 \mapsto \mathsf{true} \wedge \mathsf{true}; e_1' \mapsto \mathsf{true}\}$.

and, finally, we can instantiate $\langle \textsc{False}_1 \rangle$ with $\sigma_3 = \{v \mapsto \mathsf{true}\}$:

$$\mathsf{false} \wedge \mathsf{true} \to \mathsf{false} \quad \sigma_3\langle \textsc{False}_1 \rangle$$

In this example, we computed both arguments of $\wedge$ to reach the value.

Boolean expressions (cont)

But the inspection of the basic rules shows that if one of the arguments of $\wedge$ reduces to false, then there is no need to reduce the other argument: the rewrite will always be false. Therefore, it is more efficient to extend the rules on values to the expressions:

$$\mathsf{false} \wedge e \to \mathsf{false} \quad \langle \textsc{False}_1 \rangle \qquad e \wedge \mathsf{false} \to \mathsf{false} \quad \langle \textsc{False}_2 \rangle$$

$$\mathsf{true} \wedge e \to e \quad \langle \textsc{True}_1 \rangle \qquad e \wedge \mathsf{true} \to e \quad \langle \textsc{True}_2 \rangle$$

where e represents any boolean expression. Then, now

$$\mathsf{false} \wedge (\mathsf{true} \wedge \mathsf{true}) \to \mathsf{false} \quad \sigma_3 \langle \textsc{False}_1 \rangle$$

where $\sigma_3 = \{e \mapsto \mathsf{true} \wedge \mathsf{true}\}$.

Boolean expressions (cont)

Try to reduce

$$(\text{true} \wedge (\text{false} \wedge \text{true})) \wedge (\text{true} \wedge (\text{true} \wedge \text{true}))$$

Note that this rewrite system is not deterministic. If we enforce that the left argument of $\wedge$ is completely reduced first, then we can check if it is true or false: in the latter case, we do not need to reduce the right argument.

Boolean expressions (cont)

This is easy to specify:

$$\text{true} \wedge e \to e \quad \langle \wedge_{\text{TRUE}} \rangle \qquad \text{false} \wedge e \to \text{false} \quad \langle \wedge_{\text{FALSE}} \rangle$$

$$\frac{e_1 \to e_1'}{e_1 \wedge e_2 \to e_1' \wedge e_2} \langle \wedge \rangle$$

Many programming languages, as C, use this strategy. It allows the programmers to write handy code like

```
while (index <= max && tab[index] > 0) ...
```

Boolean expressions (cont)

Note that this strategy (left argument before the right one) is not optimal in general: the first argument may reduce to true. *Finding an optimal strategy is not always possible.* For example, we could add another rule that shortens sometimes the reductions:

$$e \wedge e \to e \ \langle \wedge_{\mathrm{REFL}} \rangle$$

This rule explicitly specifies the **reflexivity** of the conjunction.
But the system becomes non-deterministic because there is now an overlap between rule $\langle \wedge_{\mathrm{REFL}} \rangle$ and both $\langle \wedge_{\mathrm{TRUE}} \rangle$ and $\langle \wedge_{\mathrm{FALSE}} \rangle$ (consider for example true $\wedge$ false).

Determinism versus non-determinism

Let us summarise the concepts of determinism and non-determinism.

- The determinism means that, for any expression, there is only one rule to rewrite it. This implies that, every time a program is run with the same input, the output will the same. This is a useful property, called soundness, in particular for debugging.
- Non-determinism means that, for any expression, there may be several rules to rewrite it. Moreover,
  - if the rewrite system is sound, the output will always be the same for any run on the same input, even if the trace may be different each time, which makes the debugging more difficult.
  - the rewrite system may be unsound.

Determinism versus non-determinism (cont)

Some programming languages have features that may be both non-deterministic and unsound.

Perhaps the most infamous case is in C, where variables do not need to be initialised at their declaration:

```
int a;
```

If the programmer forgets to give a value to the variable, any access to it is unspecified.

Practically, this means that whatever is located in the memory at the location of the variable will be used at each read access. Of course, this value may change from one run of the program to another, even on the same input.

Maybe not.

Commutativity and its consequences

We may try another rewrite system for $\wedge$:

$$\mathsf{true} \wedge e \to e \quad \langle \textsc{True} \rangle \qquad \mathsf{false} \wedge e \to e \quad \langle \textsc{False} \rangle$$

$$\frac{e_1 \to e_1'}{e_1 \wedge e_2 \to e_1' \wedge e_2} \langle \wedge \rangle \qquad e_1 \wedge e_2 \to e_2 \wedge e_1 \ \langle \textsc{Comm} \rangle$$

While this system is in theory fine, there are two practical problems:

1. it is **not deterministic** due to $\langle \textsc{Comm} \rangle$ that overlaps with all the other rules;
2. it may **not terminate** due to rule $\langle \textsc{Comm} \rangle$:

$$\mathsf{true} \wedge \mathsf{false} \xrightarrow{\langle \textsc{Comm} \rangle} \mathsf{false} \wedge \mathsf{true} \xrightarrow{\langle \textsc{Comm} \rangle} \mathsf{true} \wedge \mathsf{false} \xrightarrow{\langle \textsc{Comm} \rangle} \mathsf{false} \wedge \mathsf{true} \to \cdots$$

Non-termination

Now some reductions may not be finite, which models **non-termination**.

This particular kind of non-termination, due to commutativity, is the model in programming languages of so-called "**infinite loops**" that do not allocate memory, i.e., that do not require additional memory at each loop step. Equivalently, it corresponds to a *recursive call* like

```
bool and (bool e1, bool e2) { ... return and (e2, e1); ... }
```

Anyway, in presence of infinite reductions, the soundness property must be weakened as follows:

> **Weak soundness.** *All finite reductions of an expression end with the same value, if any.*

Non-termination (cont)

Let us define the negation:

$$\neg\text{true} \to \text{false} \qquad\qquad \neg\text{false} \to \text{true}$$

It is also well known that the negation satisfies $e = \neg\neg e$. So we could imagine another rewrite rule

$$\neg\neg e \to e$$

This is fine, this rule simplifies double negations. But what if we had the converse rule?

$$e \to \neg\neg e \qquad \text{where } e \text{ is not a value.}$$

This allows infinite reductions: $e \to \neg\neg e \to e \to \neg\neg e \to \ldots$

Summary

There are three kinds of reductions:

1. Finite and ending in a value

$$e_1 \to e_2 \to \cdots \to v$$

2. Finite but ending in a stuck expression

$$e_1 \to e_2 \to \cdots \to e_n \not\to$$

3. Infinite

$$e_1 \to e_2 \to \cdots$$

Conditioned expressions

At this point, boolean expressions can be

- **true** (a value)
- **false** (a value)
- $e_1 \wedge e_2$
- $e_1 \vee e_2$
- $\neg e_1$

where e_1 and e_2 are boolean expressions. Let us extend our boolean expressions with a **conditioned expression**:

- **if** e_1 **then** e_2 **else** e_3

Conditioned expressions (cont)

It is easy to extend the rewrite system to cope with conditioned:

$$\textbf{if true then } e_2 \textbf{ else } e_3 \to e_2 \quad \langle \textsc{If-True} \rangle$$

$$\textbf{if false then } e_2 \textbf{ else } e_3 \to e_3 \quad \langle \textsc{If-False} \rangle$$

$$\frac{e_1 \to e_1'}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \to \textbf{if } e_1' \textbf{ then } e_2 \textbf{ else } e_3} \langle \textsc{If} \rangle$$

This means that the **conditional expression** e_1 must be completely reduced *before* the others. This is the usual way in programming languages, the rationale being to avoid unnecessary computations (evaluating e_2 and e_3 is not necessary).