From specifications to algorithms

Now, how do we go from a specification to an **algorithm**?

By algorithm, we mean an **operational description** of the specification, i.e., a series of explicit computations that respect the equations and yield the expected result.

A specification describes abstract properties (using notations like $\forall$, (x, y), $\varnothing$, $\sum$ etc. and leaving out explicit data structure definitions) and the algorithm is a calculus schema (this introduces an abstract computer model, including time).

From specifications to algorithms (cont)

One says that an algorithm is **correct with respect to its specification** when all the results of the algorithm satisfy the specification, i.e., there is no contradiction when the results are substituted into the equations of the specification.

Correctness is always a relative concept (a property of the algorithm in regard to its specification).

From specifications to algorithms (cont)

So, an algorithm is more detailed than a specification. But how much more?

Contrary to algorithms, **programs** depend on programming languages, as we mentioned before. Thus algorithms can be considered as a useful step from specification to programs, as a refinement step.

From specifications to algorithms (cont)

Algorithms can be implemented using different programming languages (featuring objects, or pointers, or exceptions etc.). The more we want to be free of using our favorite programming language, the more the language for expressing algorithms should be abstract.

However, this language may assume an abstract model of the computer which can be quite different from the one assumed by the chosen programming language. On the contrary, sometimes they are the same (this is the case of Prolog).

From specifications to algorithms (cont)

Coming back to our question: how do we go from an algebraic specification (which, by definition, describes formally a concept) to an algorithm (which describes formally a computation)?

One way is to rely on the two kinds of equations we have, recursive and non-recursive. In mathematics, the integer sequence we give page 28 can be transformed into a functional definition, i.e., U_n is expressed only in terms of n, a and b, by forming $U_{n+1} - U_n = b$ and summing both sides: $\sum_{n=0}^{p} (U_{n+1} - U_n) = \sum_{n=0}^{p} b \Leftrightarrow U_{p+1} - U_0 = (p+1)b$ $\Leftrightarrow U_{p+1} - a = (p+1)b \Leftrightarrow U_p = a + pb \Leftrightarrow U_n = a + nb$.

But this is an *ad hoc* technique (e.g., we rely on properties of the integer numbers, as addition), which cannot be applied to our algebraic specifications.

From specifications to algorithms (cont)

The general approach consists in finding a **computation scheme** instead of relying on a **reasoning**, as we did for the sequence.

This is achieved by looking at the equations in the specification and **orienting them**.

This means that an equation is then considered as a **rewriting step**, from the left side to the right side, or the reverse way.

From specifications to algorithms (cont)

Assume we have an equation $A = B$, where A and B are expressions.

The way to pass from this equality to a computation is to orient the sides as a rewriting step, i.e., wherever we find an occurrence of the left side, we replace it by the right side.

For instance, if we set $A \to B$, then it means that wherever we find a A, we can write instead a B.

But, in theory, $A = B$ is equivalent to $A \to B$ and $B \to A$. But if we keep both (symmetric) rewriting steps, we get a **non-terminating rewriting system**, as demonstrated by the following infinite chain: $A \to B \to A \to B \to \ldots$

From specifications to algorithms (cont)

A rewriting system terminates if it stops on a value.

If we have a chain $X_1 \to X_2 \to \cdots \to X_n$ and there is no way to rewrite X_n, i.e., X_n does not appear in any part of a left-hand side, then if X_n is a value, the chain is terminating.

We want a rewriting system that terminates on a **value**.

A value is made of constructors only, whilst an **expression** is made of constructors and other functions. For example,

- $\textsc{True}$ is a value;
- $\textsc{Or}(\textsc{True}, \textsc{Not}(\textsc{False}))$ is an expression.

From specifications to algorithms (cont)

Now, how can we be sure we do not lose some property by just allowing one orientation for each equality, as just keeping $A \leftarrow B$ when $A = B$?

This problem is a **completeness problem** and is very difficult to tackle,[1] therefore we will not discuss it here.

So, how should we orient our equations?

[1] Refer to the Knuth-Bendix completion semi-algorithm.

Booleans/Orienting the equations

Since we want to use an equation to compute the function it characterises, if one side of an equation contains an occurrence of the function call and the other not, we orient from the former side to the latter:

$$\begin{aligned}
\textsc{Not}(\textsc{True}) &\to_1 \textsc{False} \\
\textsc{Not}(\textsc{False}) &\to_2 \textsc{True} \\
\textsc{And}(\textsc{True}, \textsc{True}) &\to_3 \textsc{True} \\
\textsc{And}(x, \textsc{False}) &\to_4 \textsc{False} \\
\textsc{And}(\textsc{False}, x) &\to_5 \textsc{False} \\
\textsc{Or}(b_1, b_2) &\to_6 \textsc{Not}(\textsc{And}(\textsc{Not}(b_1), \textsc{Not}(b_2)))
\end{aligned}$$

Booleans/Orienting the equations (cont)

For example, here is a terminating chain for the expression $\textsc{Or}(\textsc{True}, \textsc{True})$:

$$\begin{aligned}
\textsc{Or}(\textsc{True}, \underline{\textsc{And}(\textsc{True}, \textsc{False})}) &\to_4 \textsc{Or}(\textsc{True}, \underline{\textsc{False}}) \\
\underline{\textsc{Or}(\textsc{True}, \textsc{False})} &\to_6 \underline{\textsc{Not}(\textsc{And}(\textsc{Not}(\textsc{True}), \textsc{Not}(\textsc{False})))} \\
\textsc{Not}(\textsc{And}(\underline{\textsc{Not}(\textsc{True})}, \textsc{Not}(\textsc{False}))) &\to_1 \textsc{Not}(\textsc{And}(\underline{\textsc{False}}, \textsc{Not}(\textsc{False}))) \\
\textsc{Not}(\textsc{And}(\textsc{False}, \underline{\textsc{Not}(\textsc{False})})) &\to_2 \textsc{Not}(\textsc{And}(\textsc{False}, \underline{\textsc{True}})) \\
\textsc{Not}(\underline{\textsc{And}(\textsc{False}, \textsc{True})}) &\to_5 \textsc{Not}(\underline{\textsc{False}}) \\
\underline{\textsc{Not}(\textsc{False})} &\to_2 \underline{\textsc{True}}
\end{aligned}$$

An important property of our system is that it allows different chains starting from the same expression but they all end on the same value, e.g., we could have applied $\to_2$ before $\to_1$.

Booleans/Orienting the equations (cont)

Here, we will always use a **strategy** which rewrites the arguments before the function calls.

For instance, given

$$\textsc{Or}(\textsc{And}(\textsc{True}, \textsc{True}), \textsc{False})$$

we will rewrite first

$$\textsc{And}(\textsc{True}, \textsc{True}) \to_3 \textsc{True}$$

and then

$$\textsc{Or}(\textsc{True}, \textsc{False}) \to_6 \textsc{True}$$

This strategy is named **call-by-value** because we rewrite the arguments into their values first before rewriting the function call.