

A Simple Theory of Computation

Christian Rinderknecht

24 October 2008

Introduction

The purpose of this course is to give you some understanding on different concepts that underlie many programming languages, some of which you already are familiar with.

The point is that, despite you have a programming experience with some languages, as **Java**, **C**, **C++**, **C#** etc., you do not really know the **foundations of the features** of these languages, as recursion, variable scoping, assignments, loops, classes and objects, modules, typing, etc.

So the purpose of this course is to show you how some important features of programming languages you know are built one on top of the other, but also to introduce you to new concepts, that you can find in programming languages that you do not know (yet), as **functional programming languages**.

Introduction (cont)

Since we wish to describe programming concepts, we need a language to express ourselves. Two ways, at least, are possible:

- to use a new, artificial, programming language which is extremely simple (i.e. with only a very few features) and step by step show how to enrich it by adding new features on top of the previous ones or by using external concepts;
- to use an existing programming language which has a very simple subset allowing the same building procedure.

The second approach is not possible with the languages you already know: they are too specialised. But there is a family of programming languages that are suitable: functional programming languages.

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:

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

Strategy of evaluation and rewrite systems

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

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

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:

$$\begin{aligned}
 0 + 0 &\rightarrow 0 \\
 0 + 1 &\rightarrow 1 \\
 &\dots \rightarrow \dots \\
 124 - 57 &\rightarrow 67 \\
 &\dots \rightarrow \dots
 \end{aligned}$$

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:

$$e_1 \times e_2 \rightarrow e'_1 \times e_2 \quad \text{if } e_1 \rightarrow e'_1$$

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, $(2 + 3) \times (4 + 5) \rightarrow 5 \times (4 + 5)$ is an instance since $2 + 3 \rightarrow 5$. 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:

$$((2 + 3) \times 6) \times (4 + 5) \rightarrow e'_1 \times (4 + 5) \quad \text{if } (2 + 3) \times 6 \rightarrow e'_1$$

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

$$(2 + 3) \times 6 \rightarrow e'_1 \times 6 \quad \text{if } 2 + 3 \rightarrow e'_1$$

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 $2 + 3 \rightarrow 5$. Hence, $e'_1 = 5$ and the second instance is now complete:

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

Going backwards, this implies that the former e'_1 is 5×6 . So, the first instance is

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

Instance of a rewrite rule (cont)

Our rule for multiplication, $e_1 \times e_2 \rightarrow e'_1 \times e_2$, actually depends on another, $e_1 \rightarrow e'_1$. It is usual to write them this way:

$$\frac{e_1 \rightarrow e'_1}{e_1 \times e_2 \rightarrow e'_1 \times e_2} \langle \text{MULT}_1 \rangle$$

This is called an **inference rule**. Note that it has a name, $\langle \text{MULT}_1 \rangle$. 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 $\langle \text{MULT}_1 \rangle$ using previously defined substitution σ , and noted $\sigma\langle \text{MULT}_1 \rangle$:

$$\frac{(2 + 3) \times 6 \rightarrow e'_1}{((2 + 3) \times 6) \times (4 + 5) \rightarrow e'_1 \times (4 + 5)} \sigma\langle \text{MULT}_1 \rangle$$

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:

$$\frac{2 + 3 \rightarrow e'_1}{(2 + 3) \times 6 \rightarrow e'_1 \times 6} \sigma'\langle \text{MULT}_1 \rangle$$

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

We have a basic rewrite rule $2 + 3 \rightarrow 5$, thus we extend σ' with $e'_1 \mapsto 5$ and get

$$\frac{2 + 3 \rightarrow 5}{(2 + 3) \times 6 \rightarrow 5 \times 6} \sigma'\langle \text{MULT}_1 \rangle$$

Instance of a rewrite rule (cont)

Therefore we can extend σ with $e'_1 \mapsto 5 \times 6$ and we get

$$\frac{(2 + 3) \times 6 \rightarrow 5 \times 6}{((2 + 3) \times 6) \times (4 + 5) \rightarrow (5 \times 6) \times (4 + 5)} \sigma\langle \text{MULT}_1 \rangle$$

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

$$\frac{\frac{2 + 3 \rightarrow 5}{(2 + 3) \times 6 \rightarrow 5 \times 6} \sigma'\langle \text{MULT}_1 \rangle}{((2 + 3) \times 6) \times (4 + 5) \rightarrow (5 \times 6) \times (4 + 5)} \sigma\langle \text{MULT}_1 \rangle$$

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 $\langle \text{MULT}_1 \rangle$: $\sigma' = \{e_1 \mapsto 7; e_2 \mapsto 4+5\}$.

The corresponding instance is

$$\frac{7 \rightarrow e'_1}{7 \times (4 + 5) \rightarrow e'_1 \times (4 + 5)} \sigma' \langle \text{MULT}_1 \rangle$$

But there is no rule of shape “ $v \rightarrow \dots$ ”, 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 :

$$\frac{e_2 \rightarrow e'_2}{e_1 \times e_2 \rightarrow e_1 \times e'_2} \langle \text{MULT}_2 \rangle$$

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

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

$$\frac{4 + 5 \rightarrow 9}{7 \times (4 + 5) \rightarrow 7 \times 9} \sigma' \langle \text{MULT}_2 \rangle$$

Strategy

But now, something interesting happens.

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

$$\frac{2 + 3 \rightarrow 5}{(2 + 3) \times (4 + 5) \rightarrow 5 \times (4 + 5)} \sigma'' \langle \text{MULT}_1 \rangle$$

$$\frac{4 + 5 \rightarrow 9}{(2 + 3) \times (4 + 5) \rightarrow (2 + 3) \times 9} \sigma'' \langle \text{MULT}_2 \rangle$$

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

$$(\underline{2+3})/(4-4) \rightarrow \mathbf{5}/(\underline{4-4}) \rightarrow 5/\mathbf{0} \nrightarrow$$

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:

$$\begin{array}{ll} e/0 \rightarrow \text{NaN} & \langle \text{DIV-ZERO} \rangle \qquad \text{NaN}/e \rightarrow \text{NaN} \quad \langle \text{DIV-ERR}_1 \rangle \\ e/\text{NaN} \rightarrow \text{NaN} & \langle \text{DIV-ERR}_2 \rangle \qquad \text{NaN} \times e \rightarrow \text{NaN} \quad \langle \text{MULT-ERR}_1 \rangle \\ e \times \text{NaN} \rightarrow \text{NaN} & \langle \text{MULT-ERR}_2 \rangle \qquad \text{NaN} + e \rightarrow \text{NaN} \quad \langle \text{ADD-ERR}_1 \rangle \\ e + \text{NaN} \rightarrow \text{NaN} & \langle \text{ADD-ERR}_2 \rangle \qquad \text{NaN} - e \rightarrow \text{NaN} \quad \langle \text{SUB-ERR}_1 \rangle \\ e - \text{NaN} \rightarrow \text{NaN} & \langle \text{SUB-ERR}_2 \rangle \end{array}$$

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:

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

or, even the shortest possible:

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

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 $\langle \text{MULT}_2 \rangle$ so that instead of e_1 , which stands for any expression, we specify a value v :

$$\frac{e_2 \rightarrow e'_2}{v \times e_2 \rightarrow v \times e'_2} \langle \text{NEW-MULT}_2 \rangle$$

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:

$$\frac{2 + 3 \rightarrow 5}{(2 + 3) \times (4 + 5) \rightarrow 5 \times (4 + 5)} \sigma_1 \langle \text{MULT}_1 \rangle$$

$$\frac{4 + 5 \rightarrow 9}{5 \times (4 + 5) \rightarrow 5 \times 9} \sigma_2 \langle \text{NEW-MULT}_2 \rangle$$

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

Determinism. If $e_1 \rightarrow e'_1$ and $e_2 \rightarrow e'_2$ 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 $\langle \text{MULT}_1 \rangle$, and $4+5$, with $\langle \text{MULT}_2 \rangle$, then 5×9 is finally rewritten in 45 (after **synchronisation** of the two reductions).

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:

$$\begin{array}{ll} \text{true} \wedge \text{true} \rightarrow \text{true} & \text{true} \wedge \text{false} \rightarrow \text{false} \\ \text{false} \wedge \text{true} \rightarrow \text{false} & \text{false} \wedge \text{false} \rightarrow \text{false} \end{array}$$

Boolean expressions made simpler

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

$$\begin{array}{l} \text{true} \wedge \text{true} \rightarrow \text{true} \quad \langle \text{TRUE} \rangle \\ \text{false} \wedge v \rightarrow \text{false} \quad \langle \text{FALSE}_1 \rangle \\ v \wedge \text{false} \rightarrow \text{false} \quad \langle \text{FALSE}_2 \rangle \end{array}$$

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

$$\frac{e_1 \rightarrow e'_1}{e_1 \wedge e_2 \rightarrow e'_1 \wedge e_2} \langle \wedge_1 \rangle \qquad \frac{e_2 \rightarrow e'_2}{e_1 \wedge e_2 \rightarrow e_1 \wedge e'_2} \langle \wedge_2 \rangle$$

then, for example,

$$\frac{\text{false} \wedge \text{true} \rightarrow \text{false} \quad \langle \text{FALSE}_1 \rangle}{(\text{false} \wedge \text{true}) \wedge (\text{true} \wedge \text{true}) \rightarrow \text{false} \wedge (\text{true} \wedge \text{true})} \sigma_1 \langle \wedge_1 \rangle$$

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

$$\frac{\text{true} \wedge \text{true} \rightarrow \text{true} \quad \langle \text{TRUE} \rangle}{\text{false} \wedge (\text{true} \wedge \text{true}) \rightarrow \text{false} \wedge \text{true}} \sigma_2 \langle \wedge_2 \rangle$$

where $\sigma_2 = \{e_1 \mapsto \text{false}; e_2 \mapsto \text{true} \wedge \text{true}; e'_1 \mapsto \text{true}\}$.

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

$$\text{false} \wedge \text{true} \rightarrow \text{false} \quad \sigma_3 \langle \text{FALSE}_1 \rangle$$

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:

$$\begin{array}{ll} \text{false} \wedge e \rightarrow \text{false} & \langle \text{FALSE}_1 \rangle \quad e \wedge \text{false} \rightarrow \text{false} \quad \langle \text{FALSE}_2 \rangle \\ \text{true} \wedge e \rightarrow e & \langle \text{TRUE}_1 \rangle \quad e \wedge \text{true} \rightarrow e \quad \langle \text{TRUE}_2 \rangle \end{array}$$

where e represents any boolean expression. Then, now

$$\text{false} \wedge (\text{true} \wedge \text{true}) \rightarrow \text{false} \quad \sigma_3 \langle \text{FALSE}_1 \rangle$$

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

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:

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

$$\frac{e_1 \rightarrow e'_1}{e_1 \wedge e_2 \rightarrow e'_1 \wedge e_2} \quad \langle \wedge \rangle$$

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:

$$e \wedge e \rightarrow e \quad \langle \wedge_{\text{REFL}} \rangle$$

This rule explicitly specifies the **reflexivity** of the conjunction.

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

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 be 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 :

$$\begin{array}{ll} \text{true} \wedge e \rightarrow e & \langle \text{TRUE} \rangle \qquad \text{false} \wedge e \rightarrow e \quad \langle \text{FALSE} \rangle \\[1em] \frac{e_1 \rightarrow e'_1}{e_1 \wedge e_2 \rightarrow e'_1 \wedge e_2} & \langle \wedge \rangle \qquad e_1 \wedge e_2 \rightarrow e_2 \wedge e_1 \quad \langle \text{COMM} \rangle \end{array}$$

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

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

$$\begin{array}{c} \text{true} \wedge \text{false} \xrightarrow{\langle \text{COMM} \rangle} \text{false} \wedge \text{true} \xrightarrow{\langle \text{COMM} \rangle} \text{true} \wedge \text{false} \\ \xrightarrow{\langle \text{COMM} \rangle} \text{false} \wedge \text{true} \rightarrow \dots \end{array}$$

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:

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

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

$$\neg \neg e \rightarrow e$$

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

$$e \rightarrow \neg \neg e \quad \text{where } e \text{ is not a value.}$$

This allows infinite reductions: $e \rightarrow \neg \neg e \rightarrow e \rightarrow \neg \neg e \rightarrow \dots$

Summary

There are three kinds of reductions:

1. Finite and ending in a value

$$e_1 \rightarrow e_2 \rightarrow \dots \rightarrow v$$

2. Finite but ending in a stuck expression

$$e_1 \rightarrow e_2 \rightarrow \dots \rightarrow e_n \nrightarrow$$

3. Infinite

$$e_1 \rightarrow e_2 \rightarrow \dots$$

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:

$$\mathbf{if\ true\ then\ } e_2 \mathbf{\ else\ } e_3 \rightarrow e_2 \quad \langle \text{IF-TRUE} \rangle$$

$$\mathbf{if\ false\ then\ } e_2 \mathbf{\ else\ } e_3 \rightarrow e_3 \quad \langle \text{IF-FALSE} \rangle$$

$$\frac{e_1 \rightarrow e'_1}{\mathbf{if\ } e_1 \mathbf{\ then\ } e_2 \mathbf{\ else\ } e_3 \rightarrow \mathbf{if\ } e'_1 \mathbf{\ then\ } e_2 \mathbf{\ else\ } e_3} \quad \langle \text{IF} \rangle$$

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).

Stacks

Until now, we studied basic arithmetic and boolean expressions by means of rewrite systems, which provided the formal definition of their evaluation.

In order to check the expressive power of rewrite systems, let us show how we can use them to define precisely a **data structure** which is very useful in programming: the **stack**.

This approach allows us to define and understand the concept of stack without referring to features like **pointers**, which depend on specific **computer architectures** or **operating system libraries** etc.

For instance, in the case of C, it is not possible to study stacks without the notions of **memory**, **array** or **memory allocation and deallocation** (**malloc** and **free**), integer or **pointer arithmetic** etc.

Stacks/From the analogy to the formal notation

A stack is similar to a box which is open on the top and that contains a pile of paper sheets: we can only add a new sheet on its top (this is called **to push**) and remove one on its top (this is called **to pop**).

How do we model the fact that the stack has changed after a pop or a push? The simplest is to imagine that we rewrite the operation on a given stack into a (modified) stack.

Also, we do not want to specify actually the nature of the elements in the stack, in order to be general.

Items in stacks can be of any kind, so we do not define them: *items are independent from the concept of stack*.

Stacks/Examples

Let us define the “stack values”:

- let us note **EMPTY** the empty stack;
- let us note **PUSH(i, s)** the stack resulting of pushing item i onto the stack (value) s .

For example:

- EMPTY is the empty stack;
- PUSH(i , EMPTY) is the stack containing only item i ;
- PUSH(i_1 , PUSH(i_2 , EMPTY)) is the stack which only contains item i_1 on the top and item i_2 below (the bottom).

Stacks/Comparison with C

In C, we need the standard library. Then we need to define the stack data structure using pointers. Since it is recursive, we need a trick to declare it:

```
#include<stdlib.h>
typedef struct stack_ stack;
struct stack_ { int item; stack* next; };
```

Then, we need to take care of deallocating a stack:

```
void free_stack (stack* s) {
    if (s != NULL) { stack* sub_stack = (*s).next;
                    free(s);
                    free_stack(sub_stack);
    }
}
```

Stacks/Comparison with C (cont)

The definition of PUSH in C must take care of allocating memory and checking whether this implies a memory overflow. If so, we must exit but not forget to deallocate the previous stack, if not EMPTY. Note the type of push: parameter s must be a pointer to a pointer because traditional C does not include reference passing. Also, the type of the items must be given (int).

```
void push (stack** s, int item) {
    stack* new_stack = malloc(sizeof(stack));
    if (new_stack == NULL) { if (s != NULL) free_stack(*s);
                           exit (2); }

    (*new_stack).item = item;
    (*new_stack).next = *s;
    *s = new_stack;
}
```

Stacks/Comparison with C (cont)

On this simple example, it is easy to see how the concept of stack is buried in many important details in C: memory management, address of pointers as arguments, tricky recursive type definition, fixed type for the items in the stack.

The approach with rewrite systems makes clear what the concept of stack is and what it is not necessary. In particular, our formalism does not make room for the concept of in-place modification: there is not even the concept of memory. Contrast this to the C implementation of `push`, which modifies the *given* stack in memory.

Stacks/Comparison with C (cont)

Also, in C we must give the type of the elements in the stack, even if the function does not use them: this is because the pointer arithmetic needs to know the size of each cell.

Then, once we understand what it is, we can more safely translate our understanding into programs because we already have in mind what we want to implement, and we can focus on the specific features of the language.

Stacks/Discussion

Until now, there are only stacks (or “stack values”) but no *computable* operation on them.

Operation PUSH is not computable, i.e., there is no rewrite rule for it: PUSH is only used to define what a new stack is, given another and an item to push on top of it.

This is different from the operator \times on integers, because there were rewrite rules like $2 \times 5 \rightarrow 10$. There are no such rules for PUSH.

Stacks/Discussion (cont)

In other words, EMPTY and PUSH define the concept of stack, they are *values*.

But how about popping an item from the stack? This should be part of the concept too, because an item can only be popped from the top of a stack. Should not this be part of the definition of the concept of stack?

Stacks/Popping

It is true that the way items are taken from a stack is part of the definition of the concept of stack, but in an operational way, not ontological.

In other words, popping, while participating to the definition of what a stack is, actually de-structures a stack. Therefore, popping can be defined in terms of a sub-stack, i.e., it can be defined as a computable operation.

Stacks/Popping (cont)

Actually, this is very easy: let us take POP as the exact inverse function of PUSH:

$$\text{POP}(\text{PUSH}(i, s)) \rightarrow s \quad \langle \text{POP} \rangle$$

Note that there is no rewrite rule for expression POP(EMPTY) because there is no way to pop an item from an empty stack.

Stacks/Popping in C

In contrast, in C, the POP function would be implemented as follows. Again, the programmer must take care of passing the address of the pointer to the top of the stack, `s`. In case of error, we return an error code, so the popped element, if any, must be an argument whose address is passed (`top`). Some errors are improbable, like `s` being NULL, but mandatory in general.

```
int pop (stack** s, int* top) {
    if (s != NULL)
        if (*s != NULL) { *top = (**s).item;
                           *s = (**s).next;
                           return 0; }
    return 2;
    return 1;
}
```

Stacks/Popping in C (cont)

Again, the concept of popping is buried in too many low-level details, like handling three error cases, even if improbable. It is not obvious at all, just by looking at the code, that POP is actually exactly the inverse function of PUSH.

Note also that the C implementation `pop` modifies the given stack, because it would be expensive to make a copy of it. On the other hand, modifying an argument, called **side-effect**, is always risky because the caller must be aware of it.

There is no such notion of side-effect in our rewrite systems: it is not necessary in order to understand what a stack actually is.

Stacks/Top

Now let us consider a operator which is similar to popping. The function POP returns the stack that remains when we ignore the top element. The function TOP instead returns this top element and ignores the remaining stack. This is fairly simple to define:

$$\text{TOP}(\text{PUSH}(x, s)) \rightarrow x \quad \langle \text{TOP} \rangle$$

Just like POP, it does not apply to empty stacks. In C, we do not need a side-effect on the stack (only on the item):

```
int top (stack* s, int* item) {
    if (s == NULL) return 1;
    *item=(*s).item;
    return 0;}

```

Stacks/Appending

Let expression APPEND(s_1, s_2) represent the stack made of stack s_1 on top of stack s_2 .

Let us express exactly what we mean using a rewrite system defined by inference rules.

$$\begin{aligned} \text{APPEND}(\text{EMPTY}, \text{EMPTY}) &\rightarrow \text{EMPTY} \\ \text{APPEND}(\text{EMPTY}, \text{PUSH}(e, s)) &\rightarrow \text{PUSH}(e, s) \\ \text{APPEND}(\text{PUSH}(e, s), \text{EMPTY}) &\rightarrow \text{PUSH}(e, s) \\ \text{APPEND}(\text{PUSH}(e_1, s_1), \text{PUSH}(e_2, s_2)) &\rightarrow \\ &\text{PUSH}(e_1, \text{APPEND}(s_1, \text{PUSH}(e_2, s_2))) \end{aligned}$$

Let us explain how we find theses rewrite rules.

Stacks/Appending (cont)

The first step consists in understanding that we must find rules of the shape

$$\text{APPEND}(s_1, s_2) \rightarrow \dots$$

Second, we understand that s_1 and s_2 are stacks. We already know that stacks are built by EMPTY or PUSH. Thus, we replace each stack by all

of its possible basic values: `EMPTY` and `PUSH(..., ...)`. This leads to four cases:

$$\begin{aligned} \text{APPEND}(\text{EMPTY}, \text{EMPTY}) &\rightarrow \dots \\ \text{APPEND}(\text{EMPTY}, \text{PUSH}(e, s)) &\rightarrow \dots \\ \text{APPEND}(\text{PUSH}(e, s), \text{EMPTY}) &\rightarrow \dots \\ \text{APPEND}(\text{PUSH}(e_1, s_1), \text{PUSH}(e_2, s_2)) &\rightarrow \dots \end{aligned}$$

Stacks/Appending (cont)

Now we need to think about the interpretation of each case. An analogy and a picture is helpful. In the first case, we want append two empty stacks. It is obvious that the resulting stack will be empty too:

$$\text{APPEND}(\text{EMPTY}, \text{EMPTY}) \rightarrow \text{EMPTY}$$

In the second and third case, we append an empty stack and a non-empty one. The order in which we append them is irrelevant and the result is always the non-empty one:

$$\begin{aligned} \text{APPEND}(\text{EMPTY}, \text{PUSH}(e, s)) &\rightarrow \text{PUSH}(e, s) \\ \text{APPEND}(\text{PUSH}(e, s), \text{EMPTY}) &\rightarrow \text{PUSH}(e, s) \end{aligned}$$

Stacks/Appending (cont)

The last case is the more difficult one: we append two non-empty stacks. The top item of the stack to be appended on top of the other is e_1 . This item will also be on the top of the resulting stack. That is why we expect something like

$$\text{APPEND}(\text{PUSH}(e_1, s_1), \text{PUSH}(e_2, s_2)) \rightarrow \text{PUSH}(e_1, \dots)$$

Now, the stack below e_1 in the result can be computed by appending s_1 to the second stack, $\text{PUSH}(e_2, s_2)$:

$$\begin{aligned} \text{APPEND}(\text{PUSH}(e_1, s_1), \text{PUSH}(e_2, s_2)) &\rightarrow \\ &\text{PUSH}(e_1, \overline{\text{APPEND}(s_1, \text{PUSH}(e_2, s_2))}) \end{aligned}$$

Stacks/Appending (cont)

By looking carefully to the rewrite rules, we can devise a simpler one.

Indeed, in the case of one of the stacks is empty, the result will always be the other stack. Thus, we only need to check if the first or the second stack is empty:

$$\begin{aligned}\text{APPEND}(\text{EMPTY}, s) &\rightarrow s \\ \text{APPEND}(s, \text{EMPTY}) &\rightarrow s\end{aligned}$$

How can we check that this is correct?

Stacks/Appending (cont)

Remember that s can be replaced by any stack, so let us try with the two basic stacks:

$$\begin{aligned}\text{APPEND}(\text{EMPTY}, \text{EMPTY}) &\rightarrow s \\ \text{APPEND}(\text{EMPTY}, \text{PUSH}(e, s)) &\rightarrow s \\ \text{APPEND}(\text{EMPTY}, \text{EMPTY}) &\rightarrow s \\ \text{APPEND}(\text{PUSH}(e, s), \text{EMPTY}) &\rightarrow s\end{aligned}$$

Stacks/Appending (cont)

The third rule is redundant. If we remove it, we find the original system. Thus both systems are equivalent.

There is a different, though, but not with respect of the rewrites themselves (both systems are sound) but about the *strategy*: the original system was deterministic while the second is non-deterministic: in the case when the two stacks are empty, two rules are possible.

Stacks/Appending (cont)

We can simplify further. The last rule shows that during the rewrite, the sub-expression $\text{PUSH}(e_2, s_2)$ is invariant, i.e., that it is simply copied, without none of its sub-expressions (e_2 and s_2) being used some place else in the result.

This shows that the last rule does not care whether the second stack, i.e., the one that will be below the first one, is empty or not: it is just copied. Therefore we can simply write

$$\text{APPEND}(\text{PUSH}(e_1, s_1), s) \rightarrow \text{PUSH}(e_1, \text{APPEND}(s_1, s))$$

We can rename s into s_2 , to make it more regular:

$$\text{APPEND}(\text{PUSH}(e_1, s_1), s_2) \rightarrow \text{PUSH}(e_1, \text{APPEND}(s_1, s_2))$$

Stacks/Appending (cont)

Finally:

$$\begin{array}{ll}
\text{APPEND}(\text{EMPTY}, s) \rightarrow s & \langle \text{APPEND}_1 \rangle \\
\text{APPEND}(s, \text{EMPTY}) \rightarrow s & \langle \text{APPEND}_2 \rangle \\
\text{APPEND}(\text{PUSH}(e_1, s_1), s_2) \rightarrow \text{PUSH}(e_1, \text{APPEND}(s_1, s_2)) & \langle \text{APPEND}_3 \rangle
\end{array}$$

The new last rule adds to the non-determinism. Let $\sigma_1(s) = \text{PUSH}(e_1, s_1)$ and $\sigma_2(s_2) = \text{EMPTY}$:

$$\text{APPEND}(\underline{\text{PUSH}(e_1, s_1)}, \text{EMPTY}) \rightarrow \overline{\text{PUSH}(e_1, s_1)} \quad \sigma_1 \langle \text{APPEND}_2 \rangle$$

$$\begin{array}{l}
\text{APPEND}(\text{PUSH}(e_1, s_1), \underline{\text{EMPTY}}) \rightarrow \\
\text{PUSH}(e_1, \underline{\text{APPEND}(s_1, \text{EMPTY})}) \sigma_2 \langle \text{APPEND}_3 \rangle
\end{array}$$

Stacks/Appending (cont)

The only difference is the **complexity**, that is, in this framework of rewriting systems, the number of steps needed to reach the result.

With rule APPEND_2 , if the second stack is empty, we can conclude in one step.

If rule APPEND_3 is successively preferred, we have to traverse all the elements of the first stack before terminating.

Exercise. Implement in C the APPEND function

1. using loops;
2. without loops.

Stacks/Queues

There is another common and useful linear data structure call **queue**.

As the stack, it is fairly intuitive, since we experience the concept when we are waiting at some place to get some goods or service.

Let us note EMPTY the empty queue. This is the same name as for the empty stacks, because it is a convenient choice. When the context is clear, there is no need to be more precise, otherwise we can write STACK.EMPTY for noting empty stacks and QUEUE.EMPTY for empty queues.

Let us note $\text{PUT}(i, q)$ the queue made by adding item i at the end of queue q .

Stacks/Queues (cont)

Let us define an operation on queues, named dequeuing, which consists in returning the next available item in the queue and a new queue without this item.

That is to say, $\text{GET}(q)$ is a pair (q', i) .

$$\text{GET}(\text{PUT}(i, \text{EMPTY})) \rightarrow (\text{EMPTY}, i) \quad \frac{\text{GET}(q) \rightarrow (q_1, i_1)}{\text{GET}(\text{PUT}(i, q)) \rightarrow (\text{PUT}(i, q_1), i_1)}$$

Trees

Let us now present the concept of **tree**. A tree is either

- the empty set
- or a tuple made of a **root** and other trees, called **subtrees**.

This is a **recursive definition** because the object (here, the tree) is defined by case and by grouping objects of the same kind (here, the subtrees).

A root could be further refined as containing some specific information.

It is usual to call **nodes** the root of a given tree and the roots of all its subtrees, *transitively*.

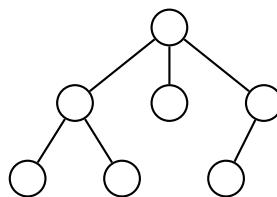
A node without non-empty subtrees is called a **leaf**.

Trees (cont)

If we consider trees as relationships between nodes, it is usual to call a root the **parent** of the roots of its direct subtrees (i.e., the ones immediately in the tuple). Conversely, these roots are **sons** of their parent (they are ordered).

It is also common to call subtree any tree included in it according to the subset relationship (otherwise we speak of *direct* subtrees).

Trees are often represented in a top-down way, the root being at the top of the page, nodes as circles and the relationship between nodes as **edges**. For instance:



Trees (cont)

The **depth** of a node is the length of the path from the root to it (note this path is unique). Thus the depth of the root is 0 and the depth of the empty tree is undefined.

The **height** of a tree is the maximal depth of its nodes. For example, the height of the tree in the previous page is 2.

A **level** in a tree is the set of all nodes with a given depth. Hence it is possible to define level 0, level 1 etc. (may be empty).

Trees/Traversals

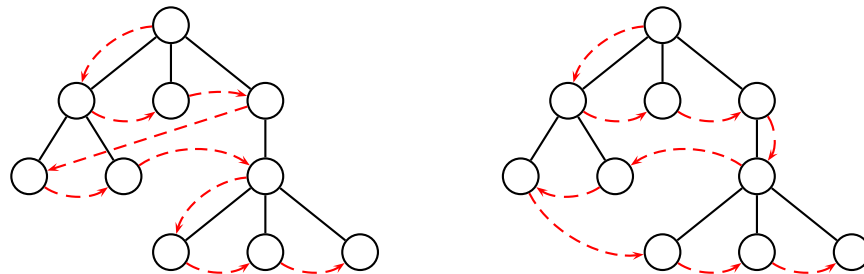
Given a tree, we can traverse it in many ways, but we must start from the root since we do not have any other node at this level.

There are two main kind of traversals:

- **Breadth-first** traversal consists in walking the tree by increasing levels: first the root (level 0), the sons of the root (level 1), then the sons of the sons (level 2) etc.
- **Depth-first** traversal consists in walking the tree by reaching the leaves as soon as possible.

In both cases, we are finished when all the leaves have been encountered.

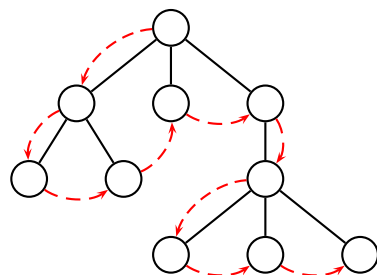
Trees/Traversals/Breadth-first



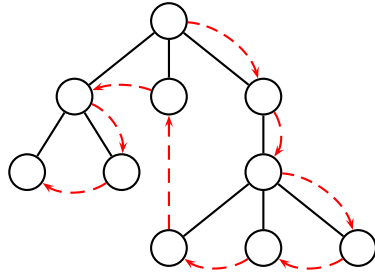
This is a **left to right** traversal.

Many others are possible, like choosing randomly the next node of the next level.

Trees/Traversals/Depth-first



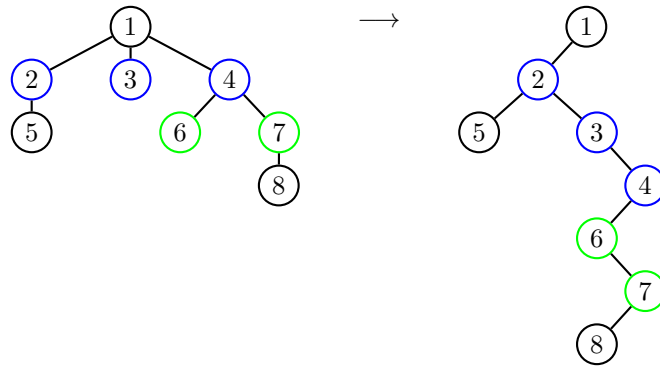
This is a **left to right** traversal.



This is a **right to left** traversal.

Trees/Binary trees

Let us consider for a while binary trees only. Indeed, we do not lose any generality with this apparent restriction because it is always possible to map any tree to a binary tree in a unique way. One method is the **left child/right sibling**:




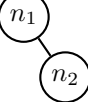
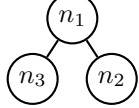
Trees/Formal definition

Let us formalise the concept of tree.

- Let us note **EMPTY** the empty tree. If the context is ambiguous, one can qualify the notation with the name of the data structure, for example, write **TREE.EMPTY** in order to make clear that we do not refer to empty stacks or empty queues.
- Let us note $\text{JOIN}(r, t_1, t_2)$ the tree whose root is r , left subtree is t_1 and right subtree is t_2 .

Trees/Examples

Let us show some examples of tree construction:

EMPTY	\emptyset
$\text{JOIN}(n_1, \text{EMPTY}, \text{EMPTY})$	
$\text{JOIN}(n_1, \text{EMPTY}, \text{JOIN}(n_2, \text{EMPTY}, \text{EMPTY}))$	
$\text{JOIN}(n_1, \text{JOIN}(n_3, \text{EMPTY}, \text{EMPTY}), \text{JOIN}(n_2, \text{EMPTY}, \text{EMPTY}))$	

Trees/Basic projections

Let us define the basic operations consisting in extracting the root, the left subtree and the right subtree of a binary tree.

$$\text{ROOT}(\text{JOIN}(r, t_1, t_2)) \rightarrow r$$

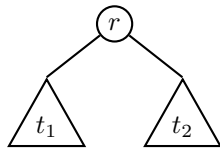
$$\text{LEFT}(\text{JOIN}(r, t_1, t_2)) \rightarrow t_1$$

$$\text{RIGHT}(\text{JOIN}(r, t_1, t_2)) \rightarrow t_2$$

Note that these three operations are not defined on empty trees. In a programming language, the case of the empty tree would be handled explicitly, either by returning an error code (C-like convention) or throwing an exception (C++ like).

We also could specify the error situation in the rewrite system, as we did for the division by zero, but we prefer not, in order to focus on the concept, not the errors.

Trees/Left to right traversals



A depth-first traversal from left to right first visits node r , then the left subtree t_1 and finally the right subtree t_2 . But if we want to keep track of the visited nodes, we have several ways.

- We record r , then nodes of t_1 and nodes of t_2 : this is **left prefix traversal**;

- we record nodes of t_1 , then r and nodes of t_2 : this is a **left infix traversal**;
- we record nodes of t_1 , then nodes of t_2 and r : this is a **left postfix traversal**.

Trees/Left-to-right prefix traversal

In order to record the traversed nodes, we need an additional structure: a stack.

The rewrite system corresponding to a left-to-right prefix traversal is

$$\begin{aligned} \text{LPREF}(\text{EMPTY}) &\rightarrow \text{EMPTY} \\ \text{LPREF}(\overline{\text{JOIN}(e, t_1, t_2)}) &\rightarrow \text{PUSH}(\bar{e}, \text{APPEND}(\text{LPREF}(\overline{t_1}), \text{LPREF}(\overline{t_2}))) \end{aligned}$$

Trees/Left-to-right postfix traversal

The rewrite system corresponding to a left-to-right postfix traversal is

$$\begin{aligned} \text{LPOST}(\overline{\text{JOIN}(e, t_1, t_2)}) &\rightarrow \\ &\text{APPEND}(\text{LPOST}(\overline{t_1}), \text{APPEND}(\text{LPOST}(\overline{t_2}), \text{PUSH}(\bar{e}, \text{EMPTY}))) \end{aligned}$$

Trees/Left-to-right infix traversal

The rewrite system corresponding to a left-to-right infix traversal is

$$\begin{aligned} \text{LINF}(\text{EMPTY}) &\rightarrow \text{EMPTY} \\ \text{LINF}(\text{JOIN}(e, t_1, t_2)) &\rightarrow \text{APPEND}(\text{LINF}(t_1), \text{PUSH}(e, \text{LINF}(t_2))) \end{aligned}$$