A Theory of Programming

Christian Rinderknecht
`Christian.Rinderknecht@nomadic-labs.com`
Nomadic Labs (Paris)
Tezos Masterclass

20 October 2018

Introduction

- This series of lectures proposes a stepwise introduction to **functional programming** and the analyses to write reliable and efficient software.
- The theoretical background of functional languages is formal logic, where mathematical functions are the main building block.
- This close link with formal logic implies that the meaning of functional programs can often be made very precise.
- It is a good framework to get a sense of **static analyses**, and, as such, it stands a prominent place in the field of formal methods and **compiler construction**.

Introduction

- In a later lecture, we will learn a specific programming language called **OCaml**, enabling you to put in practice what you have learnt.
- OCaml is used for the core development of the **Tezos blockchain**.
- Most textbooks and courses on OCaml propose a traditional approach: basic concepts, data structures, algorithms.
- Here, we aim more at illustrating the kind of software **Tezos** uses OCaml for, in particular, compiler construction.
- This first lecture is not specific to OCaml and is theoretical because it introduces many concepts that are used in functional programming and compiler construction.

String rewriting

- Let us consider a string of white and black beads, like $\circ \bullet \bullet \bullet \circ \circ \bullet$, and a one-player game whose unique move consists in replacing two adjacent beads with only one according to the rules

$$\bullet\circ \xrightarrow{\alpha} \bullet \qquad \circ\ \bullet \xrightarrow{\beta} \bullet \qquad \bullet\ \bullet \xrightarrow{\gamma} \circ$$

- The rules α, β and γ make up a simple **string-rewriting system**.
- Rules α and β can be conceived as:
  A black bead absorbs the white bead next to it.
- The goal of this game is to end up with as few beads as possible, so our example may lead to the **rewrites**

$$\circ\bullet\bullet\boxed{\bullet\ \circ}\circ\bullet \xrightarrow{\alpha} \circ\bullet\bullet\boxed{\bullet\ \circ}\bullet \xrightarrow{\alpha} \boxed{\circ\ \bullet}\bullet\bullet\bullet \xrightarrow{\beta} \bullet\bullet\boxed{\bullet\ \bullet} \xrightarrow{\gamma} \bullet\boxed{\bullet\ \circ} \xrightarrow{\alpha} \boxed{\bullet\ \bullet} \xrightarrow{\gamma} \circ$$

where the part of the string to be rewritten next is framed.

Normal forms and termination

- Other compositions of the rules lead to the same result $\circ$ as well. Some others bring all-white or all-black strings, like $\circ\,\circ$ and $\bullet$.
- Strings that can not be further rewritten, or **reduced**, are called **normal forms**.
- We may wonder whether all strings have a normal form, and, if so, if it is unique and, furthermore, if it is either all-white or all-black.
- First, let us note that the system is **terminating**, that is, there is no infinite chain of rewrites, because the number of beads strictly decreases in all the rules.
- This is not a necessary condition in general, for instance, $\circ\,\bullet \xrightarrow{\beta} \bullet\,\circ\,\circ$ would preserve termination because the composition $\beta\alpha\alpha$ would be equivalent to the original rule β.
- In particular, this means that any string has a normal form.

Invariants

- Second, notice how the parity of the number of black beads is **invariant** through each rule and how there is no rewrite rule for two adjacent white beads.
- Therefore, if there are $2p$ initial black beads, then composing rules α and β lead to an all-black string, which can be reduced by applying rule γ to contiguous pairs of beads into an all-white string made of p beads.
- Otherwise, the same all-black string can be reduced by applying alternatively γ and β on the left end or γ and α on the right end, yielding $\circ$.
- Similarly, if there is an initial odd number of black beads, we always end up with one black bead.

Confluence

- Consequently, normal forms are not unique.
- A simple example is

$$\circ\circ \xleftarrow{\gamma} \bullet\bullet\circ \xrightarrow{\alpha} \bullet\bullet \xrightarrow{\gamma} \circ$$

- Systems where normal forms are unique are **confluent**.
- If we think of a rewrite step as an elementary computation, it becomes clear that confluence is a desirable property for carrying out computations, as we would expect, for instance, an arithmetic expression to have the same value no matter the order of the small steps.
- A systemic order of rule application is called a **reduction strategy**.

Completing a system

- We can obtain confluence by adding the rule δ:

$$\bullet\circ \xrightarrow{\alpha} \bullet \qquad \circ\ \bullet \xrightarrow{\beta} \bullet \qquad \bullet\ \bullet \xrightarrow{\gamma} \circ \qquad \circ\ \circ \xrightarrow{\delta} \circ$$

  the result of the game is always one bead, whose colour depends on the original parity of the black beads as before.
- To see why, let us consider first that two non-overlapping parts of a string can be rewritten in parallel, so they are not a concern.
- The interesting cases occur when two applications of rules (maybe of the same rule) lead to different strings because they do overlap. For instance,

$$\circ\circ \xleftarrow{\gamma} \bullet\bullet\circ \xrightarrow{\alpha} \bullet\bullet$$

- The important point is that $\circ\circ$ and $\bullet\bullet$ can be rewritten into $\circ$ at the next step by δ and γ, respectively.

Critical pairs

- In general, what matters is that all pairs of strings resulting from the application of overlapping rules, called **critical pairs**, can be rewritten to the same string: they are **joinable**.
- In our example, all interactions occur on substrings made of three beads, so we must examine eight cases:

$$\begin{array}{ccccccccc}
 & & \circ\circ & \xleftarrow{\delta} & \circ\circ\circ & \xrightarrow{\delta} & \circ\circ & & \\
 & & \circ\bullet & \xleftarrow{\delta} & \circ\circ\bullet & \xrightarrow{\beta} & \circ\bullet & & \\
\bullet & \xleftarrow{\alpha} & \bullet\circ & \xleftarrow{\beta} & \circ\bullet\circ & \xrightarrow{\alpha} & \circ\bullet & \xrightarrow{\beta} & \bullet \\
\circ & \xleftarrow{\gamma} & \bullet\bullet & \xleftarrow{\beta} & \circ\bullet\bullet & \xrightarrow{\gamma} & \circ\circ & \xrightarrow{\delta} & \circ \\
 & & \bullet\circ & \xleftarrow{\alpha} & \bullet\circ\circ & \xrightarrow{\delta} & \bullet\circ & & \\
 & & \bullet\bullet & \xleftarrow{\alpha} & \bullet\circ\bullet & \xrightarrow{\beta} & \bullet\bullet & & \\
\circ & \xleftarrow{\delta} & \circ\circ & \xleftarrow{\gamma} & \bullet\bullet\circ & \xrightarrow{\alpha} & \bullet\bullet & \xrightarrow{\gamma} & \circ \\
\bullet & \xleftarrow{\beta} & \circ\bullet & \xleftarrow{\gamma} & \bullet\bullet\bullet & \xrightarrow{\gamma} & \bullet\circ & \xrightarrow{\alpha} & \bullet
\end{array}$$

- In all the cases, the divergences are joinable in one step at most.

Local confluence

- If all critical pairs are joinable after a divergence in one ore more rewrites, the system is **locally confluent**.
- Moreover, if the system is also terminating, then **every** string has exactly **one** normal form (a strong property entailing confluence).
- Sometimes, as seen previously, a non-confluent system can be completed to become confluent, via a **completion procedure**.
- In the case of terminating systems, one of such semi-decision procedures is the Knuth-Bendix algorithm.

Ground and linear rewrite systems

- The system we defined is **ground**, that is, it involves no variables.
- These allow a finite system to denote an infinite number of ground rules if variables range over infinite sets, or simply, they reduce the size of rewrite systems.
- For instance, our continued example is equivalent to

$$\bullet \circ \xrightarrow{\alpha} \bullet \qquad \circ\ x \xrightarrow{\beta+\delta} x \qquad \bullet\ \bullet \xrightarrow{\gamma} \circ$$

- If we accept multiple occurrences of a variable on the left-hand side of a rule, a so-called **non left-linear rule**, we can further decrease the size of the system:

$$x\ x \xrightarrow{\gamma+\delta} \circ \qquad x\ y \xrightarrow{\alpha+\beta} \bullet$$

Ordered rewrite systems

- Notice that there is now an implicit order over the rules: the rule $\gamma + \delta$ must be examined first for a **match** with a part of the current string, because it is included in the second (set $x = y$ in $\alpha + \beta$).
- Non-linear systems are equational theories due the implicit equality on substructures, which is complex: we will limit ourselves to **linear systems**.
- In general, the order in which the rules are written on the page is their **logical order**.
- Moreover, let us remark that the system does not specify that x must either be $\circ$ or $\bullet$: this should be stated nevertheless somewhere else.
- Generally speaking, this means that the **type** of the variables has to be defined elsewhere or inferred from the variable uses.
- We will return to that topic about OCaml.

Term rewriting

- More general are the **term-rewriting systems**, where a **term** is a mathematical object built on tuples, integers and variables.
- Let us consider the following totally ordered system:

$$\begin{aligned}(0, m) &\xrightarrow{1} m \\ (n, m) &\xrightarrow{2} (n-1, n \times m) \\ n &\xrightarrow{3} (n, 1)\end{aligned}$$

- Arithmetic operators $(-)$ and $(\cdot)$ are defined out of the system and m and n are variables denoting natural numbers.
- Would the rules not be ordered as they are actually laid out, the second rule would match any pair. Instead, it can be assumed that $n \neq 0$ when matching with it.
- The last rule is to be tried last, as it matches all terms.

Transitive closures

- We can easily see that all the compositions of rewrites starting with a natural number n end with the factorial of n:

$$n \xrightarrow{3} (n, 1) \xrightarrow{2} \ldots \xrightarrow{2} (0, n!) \xrightarrow{1} n!, \quad \text{for } n \in \mathbb{N}.$$

- Let us note $(\xrightarrow{n})$ the composition of $(\rightarrow)$ repeated $n - 1$ times:

$$\begin{aligned} (\xrightarrow{1}) &:= (\rightarrow) \\ (\xrightarrow{n+1}) &:= (\rightarrow) \circ (\xrightarrow{n}), \quad \text{with } n > 0. \end{aligned}$$

- The **transitive closure** of $(\rightarrow)$ is defined as

$$(\twoheadrightarrow) := \bigcup_{i>0} (\xrightarrow{i}).$$

- The factorial function coincides with the transitive closure of $(\rightarrow)$: $n \twoheadrightarrow n!$.
- Let $(\xrightarrow{*})$ be the reflexive-transitive closure of $(\rightarrow)$, that is,

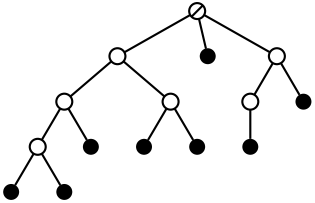
$$(\xrightarrow{*}) := (=) \cup (\twoheadrightarrow).$$

Functions

- A confluent system defines a **partial function**.
- If the system is also terminating, we get a **function**.
- We can name functions: for example, $\mathrm{c}(1, \mathrm{d}(n))$ is a term constructed with **function names** c and d, as well as variable n.
- A tuple tagged with a function name, like $\mathrm{f}(x, y)$, is called a **function call**.
- The components of the tuples are then called **arguments**, for example $\mathrm{d}(n)$ is the second argument of the call $\mathrm{c}(1, \mathrm{d}(n))$.
- It is possible for a function call to hold no arguments, like d().
- We restrict the left-hand sides of rules to be function calls.

Trees

- The topological understanding of a function call or a tuple is the finite **tree**.
- A tree is a hierarchical layout of information, for example:



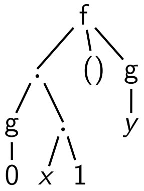
- The disks are called **nodes** and the segments which connect two nodes are called **edges**. The topmost node (with a diameter) is called the **root** and the bottommost ones (•) are called the **leaves**. All nodes except the leaves are seen to downwardly connect to some other nodes, called **children**. Upwardly, each node but the root is connected to another node, called its **parent**.

Trees

- **Trees are pervasive** in informatics, especially in functional programming, but not only.
- Structured documents, like books, articles, and web pages, are composed of chapters, sections, paragraphs, figures, appendices, indices, etc.
- The occurrences of these components are mutually constrained; for instance, it is understood that a section is part of a chapter and that appendices are located at the end of a document.
- This hierarchical layout is meant to facilitate reading, and it supports the search for specific items of information.
- When considering computer systems, these data must be uniformly encoded by means of a formal language.

Terms as trees

- Trees can be used to depict terms as follows. A function call is a tree whose root is the function name and the children are the trees denoting the arguments.
- A tuple can be considered as having an invisible function name represented by a node with a period (.) in the tree, in which case the components of the tuple are its children. For example,



  is the tree corresponding to the tern $\mathrm{f}((\mathrm{g}(0),(x,1)),(),\mathrm{g}(y))$.
- Note that tuples, except the empty tuple, are not represented in the tree, since they encode the structure itself, which is already laid out. The number of arguments of a function is called **arity**.

Functional languages

- For programming, we only consider confluent systems because they define partial functions.
- To have only one reduction chain per term to the normal form, we may also enforce that rules are ordered.
- We also enforce that normal forms must be **values**, that is, terms without function calls or only with calls to functions not occurring on the left-hand sides.
- In other words, irreducible calls of defined functions are not allowed in values.
- Systems with these constraints make up a **functional language**.
- A program here is a term, and evaluating it means to rewrite the term
    1. to a value,
    2. to a normal form that is not a value (that is, an error),
    3. or to never terminate.

Termination

- We do not require by construction that all functional programs terminate, although that is a desirable property, and we speak nevertheless of “functions” instead of “partial functions” (which would be technically correct).
- If we enforce termination, we loose **Turing-completeness**: we cannot program all computable functions.
- The termination of rewrite systems and, in particular, functional programs, is an **undecidable problem** in general.
- Nevertheless, there exists a set of well-known **decision criteria**, which we will use when programming recursive functions in OCaml.

Call-by-value

- If a function call contains at least one other function call, there is a choice as to which one to reduce first.
- One reduction strategy, named **call-by-value**, consists in always reducing the arguments before the call itself.
- Unfortunately, call-by-value enables otherwise terminating rewrites to not terminate. For instance, let us consider

  $$\mathsf{f}(x) \xrightarrow{\alpha} 0 \qquad \mathsf{g}() \xrightarrow{\beta} \mathsf{g}()$$

  We have $\mathsf{f}(\mathsf{g}()) \xrightarrow{\alpha} 0$ but $\mathsf{f}(\mathsf{g}()) \xrightarrow{\beta} \mathsf{f}(\mathsf{g}()) \xrightarrow{\beta} \ldots$
- Despite the loss of expressiveness, we shall retain in this lecture the call by value, as it is the strategy of OCaml and it is easy to follow.
- In general, the order in which the arguments of a function call are evaluated is not specified.

Call-by-value

- Another reason to choose call-by-value is that it allows us to restrict the shape of the left-hand sides, called **patterns**, to one, outermost function call.
- For instance, we can then disallow, for being useless, a rule like

  $$\mathsf{plus}(x, \mathsf{plus}(y, z)) \rightarrow \mathsf{plus}(\mathsf{plus}(x, y), z).$$

  because, following call-by-value, the argument $\mathsf{plus}(y, z)$ must be evaluated before the outermost call, therefore it cannot be part of any pattern.

Higher-order programs, recursion, λ-calculus

- Most functional languages allow **higher-order function** definitions, whereas standard term-rewriting systems do not. For example:

  $$\mathsf{f}(g, 0) \rightarrow 1 \qquad \mathsf{f}(g, n) \rightarrow n \times g(g, n-1) \qquad \mathsf{fact}(n) \rightarrow \mathsf{f}(\mathsf{f}, n)$$

  where $n \in \mathbb{N}$. Note that these two definitions are **not** recursive, yet the function fact is the factorial function.
- A function definition is **recursive** if the name of the definition occurs on at least one right-hand side.
- An adequate theoretical framework to understand higher-order functions is λ-**calculus**.
- In fact, λ-calculus features prominently in the semantics of programming languages, even not functional ones.
- For didactic purposes, we began with rewrite systems because they offer implicit pattern matching (that is, rule selection).

Stacks in a functional setting

- We show how to express linear data structures in a purely functional language and how to compute with them.
- The simplest of those structures is the **stack**.
- A stack can be thought of as a finite series of items that can only be accessed sequentially from one end, called the **top**, following the analogy with a stack of material objects.

Stacks in a functional setting (continued)

- To define a stack in a functional language, one uses calls to undefined functions to model all the possible shapes a stack can take.
- Here, a stack can be either empty or not.
- Let nil() be the irreducible call that denotes an empty stack (the function nil is undefined).
- If not empty, the stack then must contain a topmost (first) item, and a **substack**, that is, the stack made of all the items but the first.
- Let $\text{cons}(x, s)$ denote the non-empty stack with the item x on top and the **immediate substack** s.

Inductive definition of stacks

- The previous definition is intuitive, but not formal yet.
- Data structures are usually defined **inductively**.
- Let $\mathcal{T}$ be the set of all possible terms and $\mathcal{S} \subset \mathcal{T}$ be the set of all stacks.
- Formally, $\mathcal{S}$ can by defined by **induction** as the smallest set such that
  1. $\mathsf{nil}() \in \mathcal{S}$;
  2. if $x \in \mathcal{T}$ and $s \in \mathcal{S}$, then $\mathsf{cons}(x, s) \in \mathcal{S}$.
- We take the smallest set because we do not want in $\mathcal{S}$ terms that were not constructed by the inductive rules (this is called the **smallest fixed point** of the inductive relation).

Appending a stack

- For all $s, t \in \mathcal{S}$, let us consider the functional program

$$\begin{aligned} \mathsf{cat}(\mathsf{nil}(), t) &\xrightarrow{\alpha} t \\ \mathsf{cat}(\mathsf{cons}(x, s), t) &\xrightarrow{\beta} \mathsf{cons}(x, \mathsf{cat}(s, t)) \end{aligned}$$

- This is a functional program suitable for call-by-value reduction strategy, because the function calls in the patterns (i.e., left-hand sides) are irreducible.
- The right-hand sides are stacks, so a call to "cat" is a stack.
- Rule α says that, if the first stack is empty, the value is the second.
- In rule β, the first stack is not empty, and its top item is the top of the final stack (x), the immediate substack (s) being used in a call to "cat" with the second stack (t) unchanged.

Appending a stack

- This function appends the second stack at the bottom of a **copy** of the first. (The function name “cat” stands for “catenation”, and is slightly better than “app”).
- We could also say that this function appends a copy of the first stack on top of the other. But this is not the same as pushing the first stack on top of the other, like

$$\mathrm{cat}(s, t) \rightarrow \mathrm{cons}(s, t) \qquad \text{Wrong!}$$

- This is **not** appending: here, the stack s becomes an item of the resulting stack, but we want only the **contents** of s to be pushed on top of t, not their container.
- Analogy: You may think of an empty stack as a empty cardboard box, open topside, and every item pushed in it is like a book that you put on the topmost book, or at the bottom of the box if this is the first book.

Appending a stack

- Let us set the abbreviations
  - $[\,] := \mathsf{nil}()$,
  - $x :: s := \mathsf{cons}(x, s)$,

  after the convention of the programming language OCaml.
- We can further abbreviate
  - $x_1 :: x_2 :: \ldots :: x_n :: s := \mathsf{cons}(x_1, \mathsf{cons}(x_2, \ldots \mathsf{cons}(x_n, s)))$
  - $[x] := x :: [\,]$
  - $[x_1; x_2, \ldots; x_n] := x_1 :: x_2 :: x_3 :: \ldots :: x_n :: [\,]$.
- Our system now becomes a bit more legible:

$$\mathsf{cat}([\,], t) \xrightarrow{\alpha} t$$
$$\mathsf{cat}(x :: s, t) \xrightarrow{\beta} x :: \mathsf{cat}(s, t)$$

Pattern matching

- Let us compute $\mathsf{cat}([1;2],[3])$.
- We need to find the first rule that is matched by this call.
- Remember that the arguments must be values before you do that.
- Intuitively, this means that we compare the call with the patterns, that is, the left-hand sides of the rules, in order.
- Here, rule α is not matched because $[1;2] \neq []$.
- Rule β is matched because $[1;2] = 1 :: [2]$ and equals the sub-pattern $x :: s$ if we assign $x := 1$ and $s := [2]$. Also, trivially, $t := [3]$ works.
- After the rule β is selected, the assignments are used to replace the variables in the right-hand sides.
- The whole evaluation is

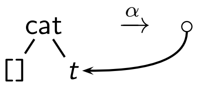
$$\mathsf{cat}([1;2],[3]) \xrightarrow{\beta} 1 :: \mathsf{cat}([2],[3]) \xrightarrow{\beta} 1 :: 2 :: \mathsf{cat}([\,],[3]) \xrightarrow{\alpha} [1;2;3].$$

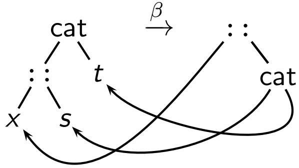
Abstract syntax trees

- Depending on the context, we may use the arborescent depiction of terms to bring to the fore certain aspects of a computation.
- For example, it may be interesting to show how parts of the output (the right-hand side) are actually **shared** with the input (the left-hand side).
- In other words, how much (new) data is created by a given rule.
- This concept supposes that terms reside in some sort of space and that they can be referred to from different other terms.
- This abstract space serves as a model of an interpreter's memory, called the **heap**.

Data sharing

- Consider for instance





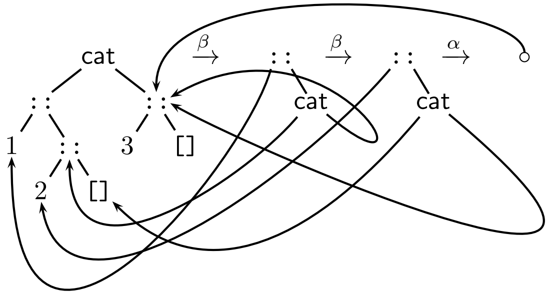
which is the same definition of "cat" as given above:

$$\mathrm{cat}([\,], t) \xrightarrow{\alpha} t \qquad\qquad \mathrm{cat}(x :: s, t) \xrightarrow{\beta} x :: \mathrm{cat}(s, t)$$

- The arrows on certain edges denote data sharing.
- When trees are used to visualise terms, they are called **abstract syntax trees** (AST).
- When some trees share subtrees, the whole **forest** (set of trees) is called a **directed acyclic graph** (DAG).

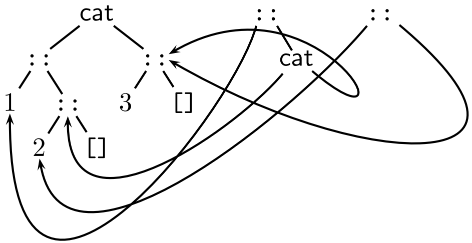
Data sharing and evaluation

- The evaluation of $\mathsf{cat}([1;2],[3])$ is:



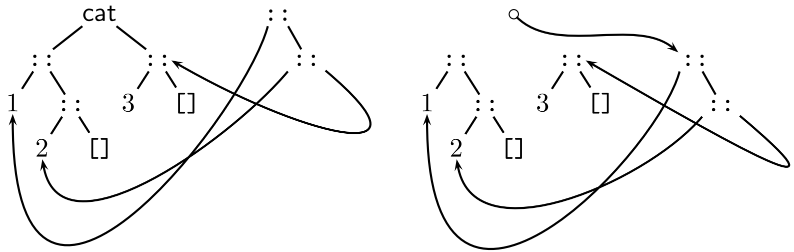
Data sharing and evaluation

- Note that each right-hand side shows only the rewritten call, not the whole term.
- Therefore, to reconstruct the value, we need to work leftwards and replace the rewritten call by the right-hand side. First:



Data sharing and evaluation

- Then



Call stack

- We can see that the value of the second argument, [3], is shared with the value of the call, as well as the integers 1 and 2.
- We saw how rewrites accumulated rightwards until the result had to be reconstructed by replacing leftwards the right-hand sides.
- This dynamic is that of a stack, but a stack holding suspended function calls (that is, awaiting their values) and references to ASTs in the heap.
- This is the **call stack**.
- The top of the stack (here, the rightmost abstract syntax tree) is the next term to be rewritten, unless the call on the left-hand side is the original call, in which case the top is its value.

Garbage collection

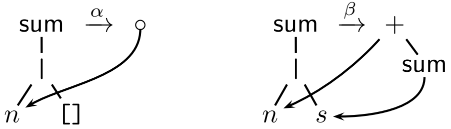
- In the abstract syntax trees (AST), the nodes with the calls that are replaced by their values have been immediately removed.
- In fact, they could have been removed later, in any order.
- The process which identifies and discards useless ASTs is called the **garbage collector** (GC).
- A term is useless if it cannot be accessed from anywhere in the call stack.
- Contrary to the heap, which can be thought of as an unordered set of trees (a forest of DAGs), the call stack is an ordered set that can be efficiently extended or reduced.
- The garbage collector can be conceived as a process interleaved with the process of evaluation.
- Like evaluators of functional programs, garbage collectors may also have different strategies (as to when dispose of useless data).

Tail call optimisation

- Let us define a function that sums integers in a non-empty list:

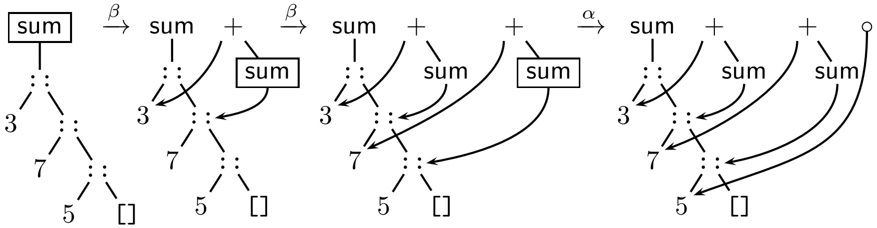
$$\mathsf{sum}([n]) \xrightarrow{\alpha} n \qquad \mathsf{sum}(n :: s) \xrightarrow{\beta} n + \mathsf{sum}(s)$$

- The view of the system with maximum sharing is:



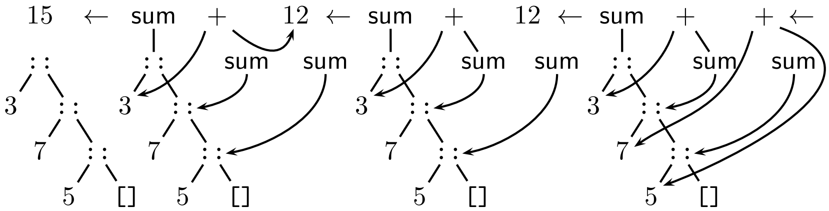
Tail call optimisation

The first phase of the evaluation of sum([3; 7; 5]) is:



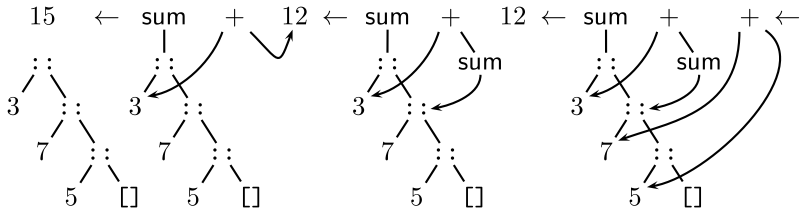
Tail call optimisation

The second phase (replacing the pending calls with their values) is



Tail call optimisation

The same, with a garbage collector that collects after each replacement:

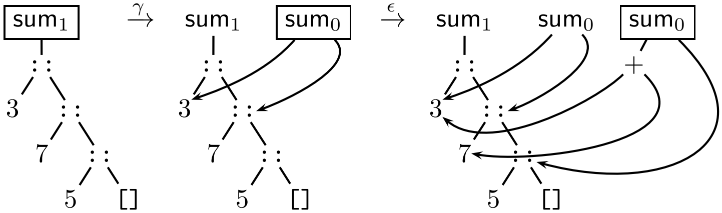


Tail call optimisation

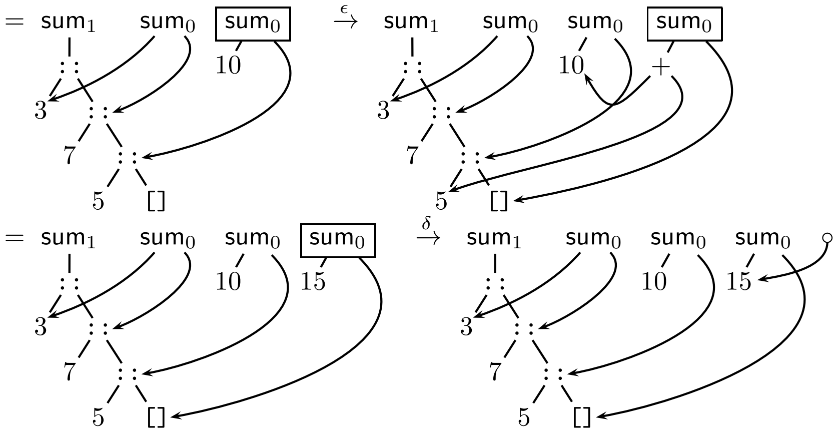
- Let us consider an alternative version of sum:

$$\begin{aligned}\mathsf{sum}_1(n :: t) &\xrightarrow{\gamma} \mathsf{sum}_0(n, t).\\ \mathsf{sum}_0(n, [\,]) &\xrightarrow{\delta} n\\ \mathsf{sum}_0(n, m :: s) &\xrightarrow{\epsilon} \mathsf{sum}_0(n + m, s).\end{aligned}$$

- For $\mathsf{sum}_1([3; 7; 5])$ we have the evaluation

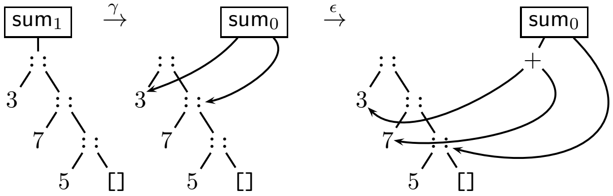


Tail call optimisation

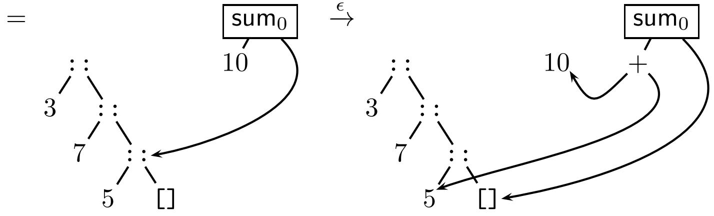


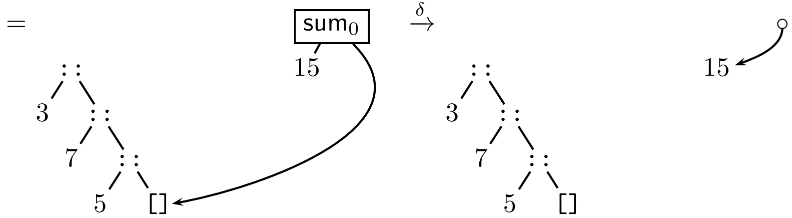
Tail call optimisation

- Notice that, in the case of sum_0, we found the result (15) at the end of the first phase (of growing the call stack).
- Therefore, there is no need for a second phase, which replaces the pending calls by their values: we can discard all those calls at once after the first phase.
- Or, to see it even more clearly, let us rewrite the same first phase and assume that the collector reclaims immediately the useless pending calls:



Tail call optimisation





Tail call optimisation

- As we see now, we can keep the size of the call stack constant, if we reclaim the top at each new rewrite.
- This optimisation is called **tail call optimisation**.
- Intuitively, a call in **tail position** is a call whose value is the value of the previously rewritten call.
- For example, the call $f(x)$ in $1 + f(x)$ is not in tail position, because its value needs to be incremented.
- Most compilers of functional languages detect and implement that optimisation that saves memory.
- Even the C compiler of GCC.