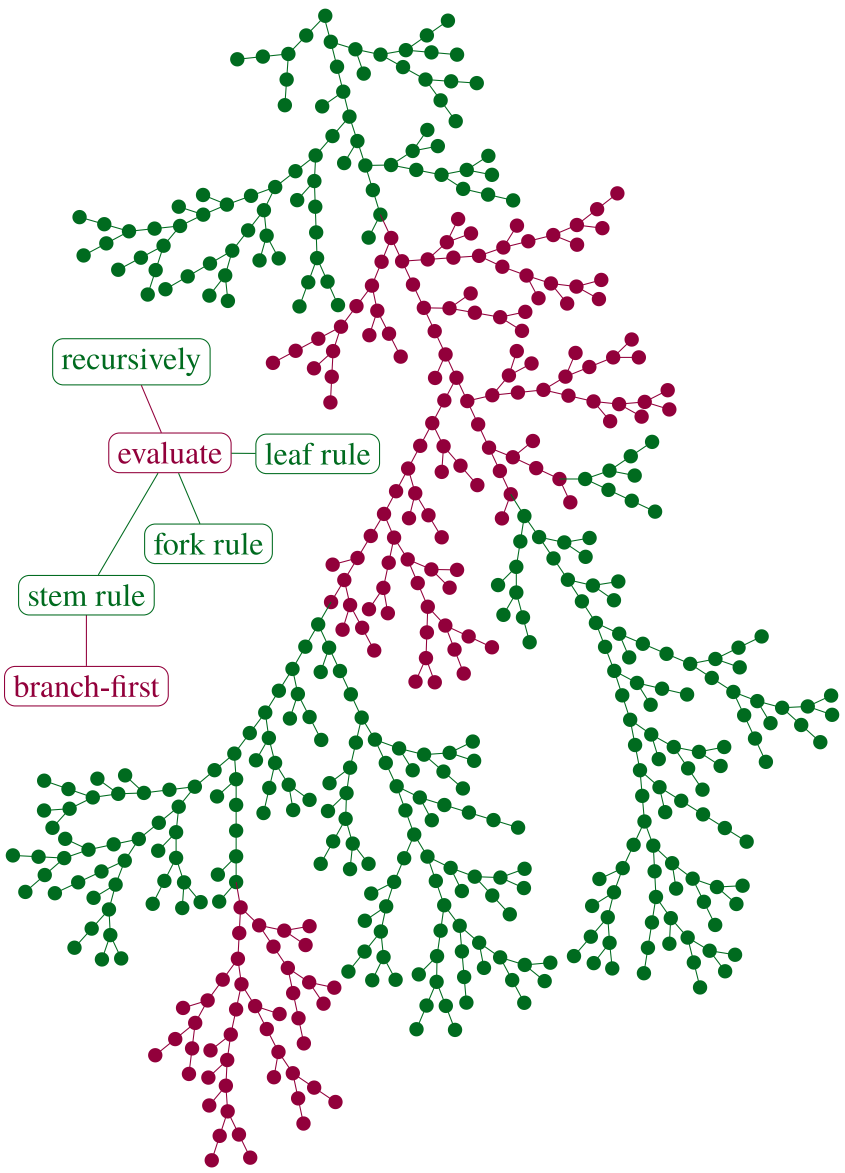
Barry Jay

Reflective Programs in Tree Calculus

September 25, 2020

Springer Nature

A Branch-First Evaluator



To Cathy
who is always there for me

Preface

Tree calculus is seeded by a single operator, whose self-application gives rise to powerful reflective programs including a size program that can compute its own size, an equality program that can decide its own equality, and a self-interpreter that can interpret itself. This is achieved without any of the usual outside machinery, such as the Gödel numbering of Turing machines, the quotation of lambda-abstractions or serialisation of programs. The resulting theory of computation is both simple and powerful: simple because it is based on three equations only; powerful because one program can query the internal structure of another, using a mix of extensional and intensional techniques.

Trees have long been fundamental to computing practice. Compilers parse strings of characters to syntax trees which are analysed and transformed in various ways before being converted into bits or numbers that can be executed by appropriate hardware. However, syntax trees are quite artificial, with nodes labelled by keywords that vary from one language to another, which makes them unsuitable for a general theory. When the same issue arises with the choice of characters to be parsed, it is resolved by replacing the characters with natural numbers. So now we avoid artifice by using *natural trees*, without labels, which are as natural as the numbers but whose branching provides support for program structure, so that extra machinery such as Gödel numbers or quotation is no longer necessary.

Comparison with tree calculus reveals new insights about traditional models of computation. For example, combinatory logic supports fixpoint functions that are in normal form, that do not reduce until given an argument. Again, there is a new variant of lambda-calculus, called *VAR*-calculus, whose three operators represent variables, abstraction and replacement (or substitution), without any explicit mention of scoping rules.

The relative power of tree calculus is shown by giving *meaningful translations* to it from traditional models of computation. Conversely, there is no such translation from tree calculus to combinatory logic. In this sense, combinatory logic is weaker than tree calculus, which conflicts with some versions of the Church-Turing Thesis. The conflict is resolved in the appendix, written a few years ago with my student Jose Vergara.

By eliminating such outside machinery, tree calculus eliminates the boundary between programs and their interpreters, so that compilation techniques may be freely used during program execution without serialisation or other artifice. It will be interesting to see how this develops. More broadly, tree calculus may provide a better foundation for mathematics, in which data structures and functions are uniformly represented by trees.

This book is intended for anyone who is interested in the theory of computation, from students to experts. It is in two parts. Part I introduces tree calculus and explores its expressive power, culminating in the production of self interpreters, using no more than equational reasoning and a little structural induction. Part II considers its relationship to other models of computation, using some basic rewriting theory.

None of the key results in this book have been previously published. Those of Part I and the appendix have been submitted to multiple conferences and journals but spirited debate has never quite resulted in a decision to publish, even when the theorems have been formally verified. So I am particularly grateful to those who have read early drafts.

Sydney, Australia *Barry Jay*
May 2020

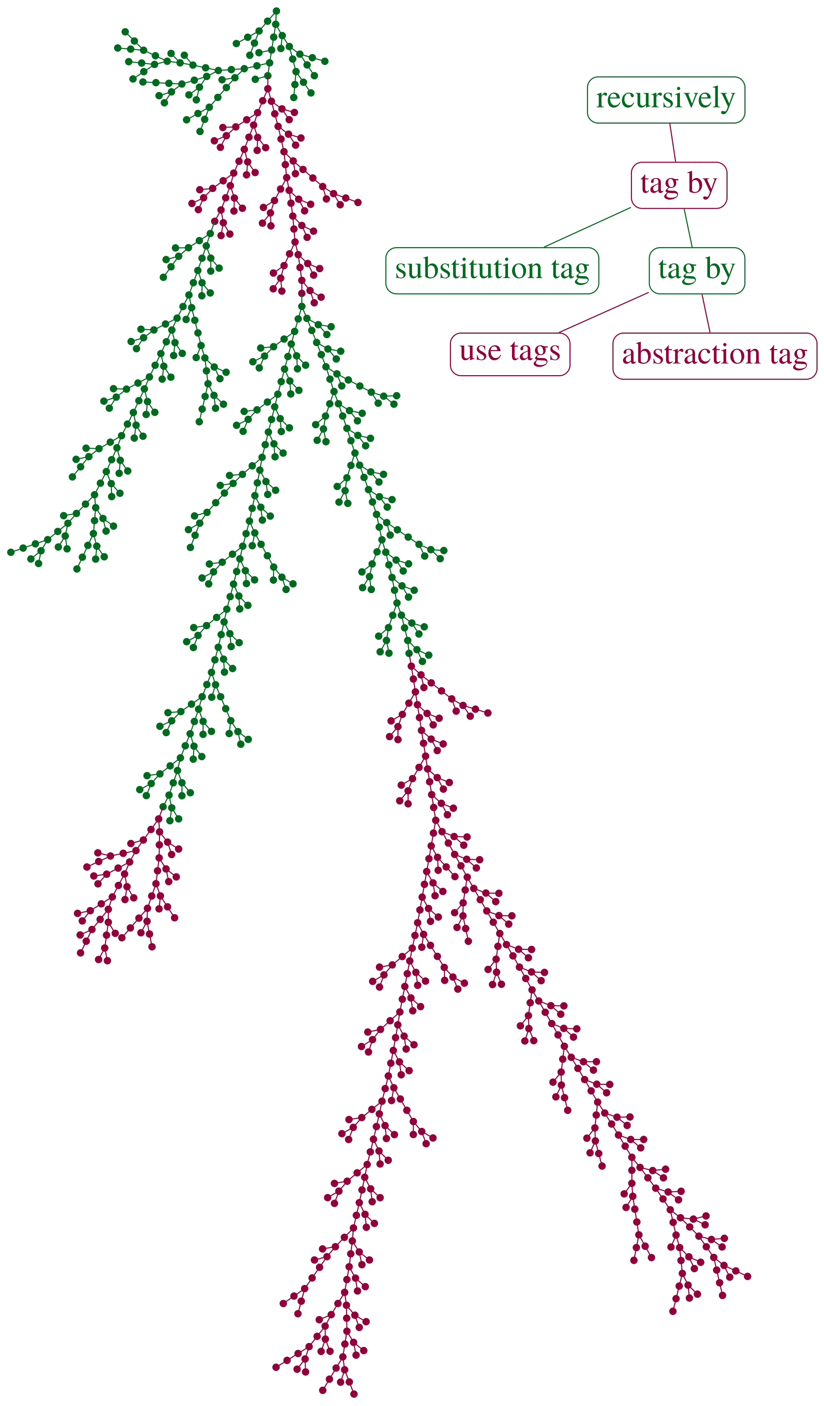
Acknowledgements

To be written.

Contents

List of Figures

A TREE FOR λ



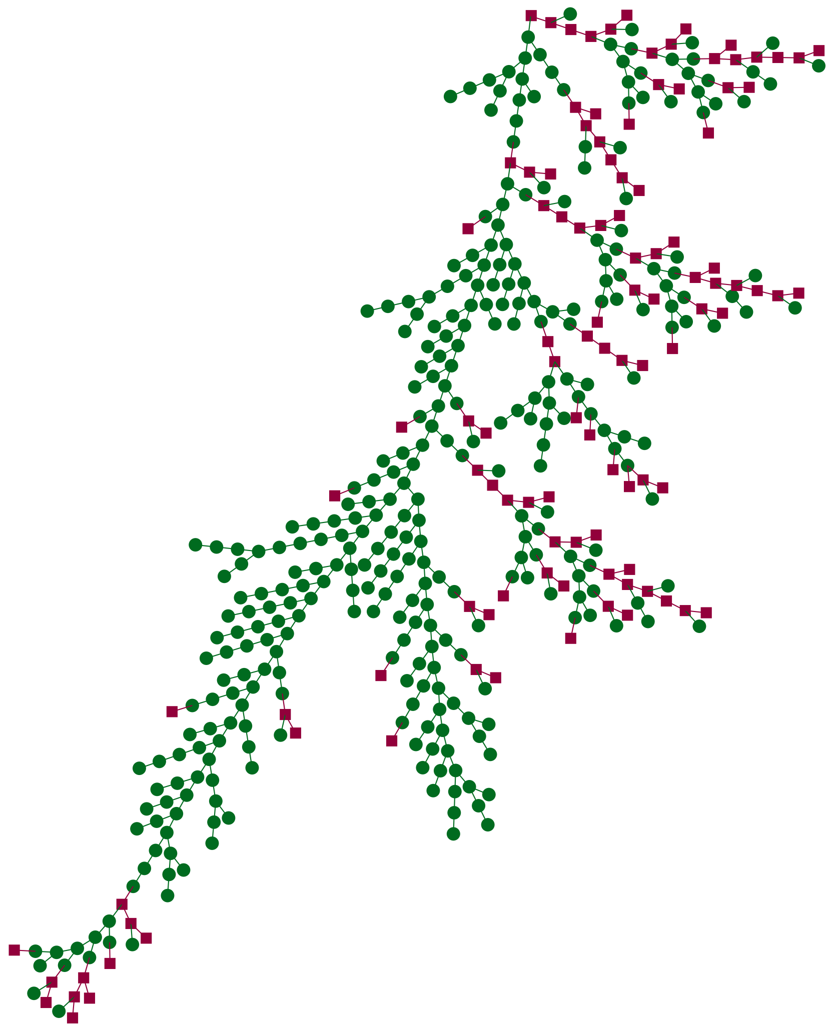


Fig. 0.1 The Translation of $\triangle$ to VA-Calculus: Circles are V and Squares are A

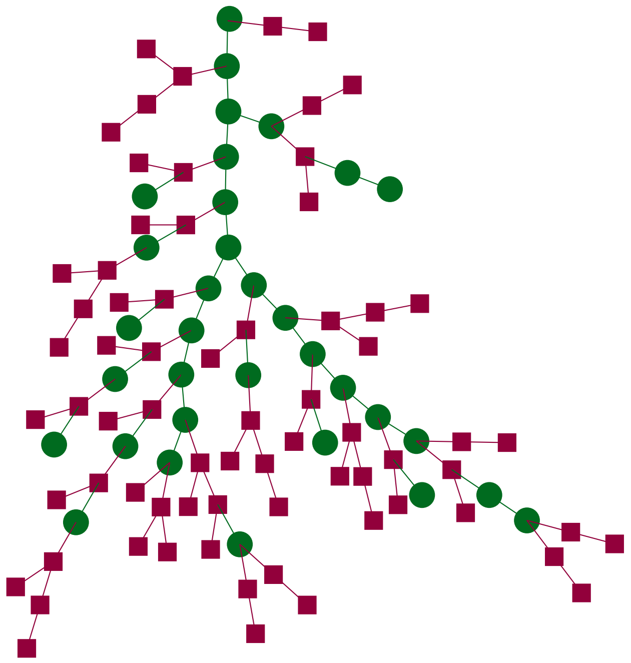


Fig. 0.2 The Translation of $\triangle$ to SF-Calculus: Circles are S and Squares are F

Glossary

abstraction with respect to some variable x converts a term t into a function of x. Although the most common realisation of this is in lambda calculus, the term form for λ-abstraction does not appear in any of the BNFs in this work. Either the constructions $[x]t$ or $\lambda^* x.t$ are used to specify combinations, or the operator A is used to build abstractions in VAR-calculus.

atoms are the indivisible combinations, here synonymous with operators.

Backus-Naur Form (BNF) is a notation for describing inductively-defined classes of terms or expressions, such as trees, numbers, combinations, programs or types.

binary trees are trees in which each node has zero, one or two branches. In tree calculus these are the values.

breadth-first evaluation (similar to lazy evaluation) requires that evaluation at a node proceed as soon as possible, so that its branches are evaluated only if necessary.

calculus (plural "calculi") is any system for doing calculations.

canonical forms are a well-specified sub-class of expressions, e.g. given by a BNF, that are the intended results of computation. When these are the irreducible terms of a rewriting system then they are also known as the normal forms.

confluence is a property of rewriting systems which ensures that each expression has at most one normal form.

combinations are expressions built from operators by application, such as combinators.

combinators are functions obtained by abstracting with respect to all variables in expressions built from variables by application. This class of functions can be built as combinations of a small set of operators, such as S and K.

compilers convert a program text or description into executable instructions.

compounds are combinations in which the leading operator does not have enough arguments to be evaluated, e.g. SMN where S requires three arguments.

computation combines a program with its inputs to calculate the output. In the calculi of this work, all combinations are computations.

dependent types blur the distinction between types and terms. For example, a term may be of array type which depends on, which is built using, a term that represents the length of the array.

depth-first evaluation (similar to eager evaluation) requires that all branches of a node be evaluated before any evaluation of the node itself.

equational reasoning is driven by the ability to replace equals by equals in any context, a property which fails in some models of computation.

extensional programs do not query the internal structure of their arguments. Extensional functions can be identified with the combinators.

extensions are pattern-matching functions that extend a default function with a new case.

evaluation strategies constrain the application of rewriting rules, usually to ensure that evaluation becomes deterministic. Examples include depth-first evaluation and breadth-first evaluation.

factorisation is used to support divide-and-conquer strategies, where the compounds are divided and the atoms are conquered.

forks are nodes hat have exactly two branches.

general recursion describes a large class of recursive functions; see also μ-recursion.

Gödel numbering is used to convert any symbolic expression into a natural number.

inputs ; see **values**.

intensional programs may query the internal structure of their arguments, e.g. to determine if they are atoms or compounds, or to perform triage.

interpreters act on a program paired with its inputs to produce some output. The program may be a term in a calculus or some representation of it obtained by quotation. Often the output is the same as that obtained by evaluating the program directly but if the program has been quoted then the output may be no more than the result of unquoting, to recover the original term.

intuitionism expects all mathematics to be constructive so that, for example, a proof of existence must include the ability to construct a witness that has the desired property.

irreducible terms cannot be reduced, as no sub-term instantiates the left-hand side of a rewriting rule.

judgments are the premises and conclusions of formal systems for, say, performing evaluation according to some strategy.

kernel ; see leaves.

leaves are nodes that have no branches.

meaningful translations preserve the process of evaluation as well as the results.

normal forms are canonical forms that are irreducible.

operators are indivisible constants used to build up combinations.

outputs ; see **values**

pattern calculus is a family of calculi that base computation on pattern matching. This supports generic queries of data structures but not of pattern-matching functions and so cannot support reflection.

programs have been variously identified with their text or syntax, the executable code produced by a compiler, a function from input to outputs, or a term in some calculus. In the technical chapters of this work, a program is identified with a value in some calculus, e.g. a binary tree.

quotation is used to represent functions and computations as data structures.

recursion is the ability of a function or program to call itself during evaluation with inputs that have been freshly calculated; see also general recursion.

reflective programs are intensional programs that are designed to query the internal structure of other programs, including themselves.

rewriting systems perform computation by applying their given rewriting rules to terms, in the hope of simplifying them to a canonical form.

self-evaluators are a particular kind of self-interpreter that may exist when programs and their inputs are irreducible. They evaluate the program according to some strategy, such as depth-first or breadth-first evaluation, so that evaluation is deterministic, even though evaluation in the calculus itself is non-deterministic.

self-interpreters are interpreters in which the language or calculus of the interpreter is the same as that of the program.

serialisation converts computations into natural numbers (or strings of bits) that can be communicated to another computer.

stems are nodes that have exactly one branch.

syntax trees commonly arise after parsing of an input string to determine the intended structure.

tagging of a function with a tag allows the tag to change the intensional behaviour of the term without changing the extensional behaviour of the function.

terms are expressions built from variables and operators by applications, and so include the combinations. The variables serve as placeholders for unknown terms.

triage performs three-way case analysis on binary trees.

Turing complete systems are able to compute the same class of functions as the Turing machines, as discussed in the appendix.

Turing machines compute by using a transition table to determine the next action as a function of the current state and the current symbol on the tape. Since the transition table is not a tape, Turing machines cannot be applied to each other without elaborate encoding and decoding.

types are formal systems that are used to classify programs.

values are the permissible results of computations. In rewriting systems, they are usually the irreducible terms or normal forms. In the calculi of this work, the values are identified with the programs, their inputs and their outputs. In tree calculus, the values are the binary trees; trees with three or more branches are computations.

variables are used in two closely related ways: as placeholders for unknown terms, say in evaluation rules; or to support abstraction. In the terms of tree calculus, the variables are drawn from some unspecified, infinite class, the only obligation being that their equality should be decidable. In VAR-calculus, the variables are represented as indices built using the operator V.

μ-recursion is a class of general recursive functions that act on natural numbers. Since the functions are not numbers μ-recursive functions cannot be applied to other such without elaborate encoding and decoding.