

Barry Jay

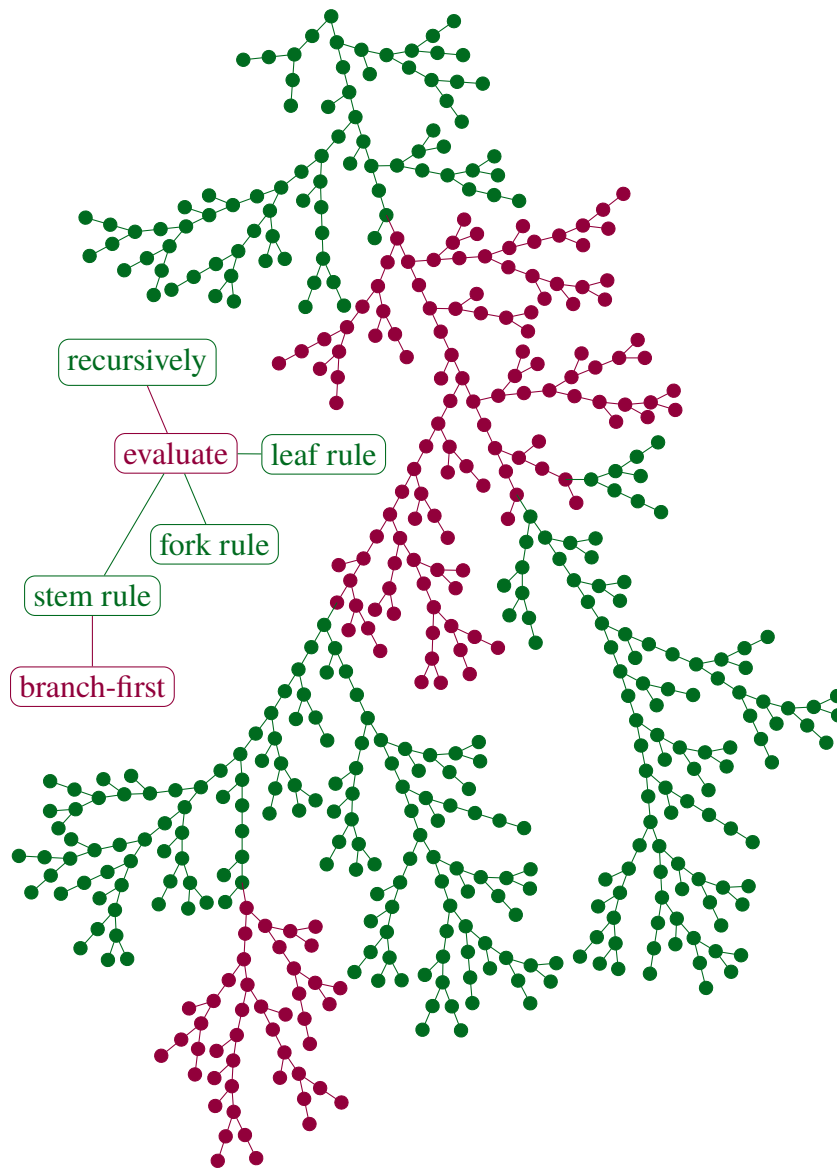
# Reflective Programs in Tree Calculus

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  
May 2020

*Barry Jay*



## **Acknowledgements**

To be written.



# Contents

## Part I Tree Calculus

<b>1</b>	<b>Introduction</b>	5
1.1	Reflective Programs	5
1.2	Tree Calculus	6
1.3	Other Models of Computation	7
1.4	How to Read this Book	9
<b>2</b>	<b>Equational Reasoning</b>	11
2.1	Addition	11
2.2	Inductive Definitions	14
2.3	Notational Conventions	15
2.4	Equivalence Relations	16
2.5	Congruences	16
2.6	Collecting Like-Terms	16
2.7	Rules for Operators	17
2.8	Arithmetic Expressions	17
2.9	Translation	21
<b>3</b>	<b>Tree Calculus</b>	23
3.1	Syntax Trees	23
3.2	Natural Trees	24
3.3	Tree Calculus	26
3.4	Programs	31
3.5	Propositional Logic	31
3.6	Pairs	33
3.7	Natural Numbers	33
3.8	Fundamental Queries	35

<b>4</b>	<b>Extensional Programs</b>	39
4.1	Combinators, Combinations and Terms	39
4.2	Variable Binding	41
4.3	Fixpoints	43
4.4	Waiting	44
4.5	Fixpoint Functions	46
4.6	Arithmetic	48
4.7	Lists and Strings	49
4.8	Mapping and Folding	50
<b>5</b>	<b>Intensional Programs</b>	53
5.1	Intensionality	53
5.2	Size	54
5.3	Equality	54
5.4	Tagging	57
5.5	Simple Types	58
5.6	More Queries	60
5.7	Triage	61
5.8	Pattern Matching	61
5.9	Eager Function Application	63
<b>6</b>	<b>Reflective Programs</b>	65
6.1	Reflection	65
6.2	Evaluation Strategies	66
6.3	Branch-First Evaluation	66
6.4	A Branch-First Self-Evaluator	67
6.5	Root Evaluation	70
6.6	Root-and-Branch Evaluation	72
6.7	Root-First Evaluation	73
 <b>Part II Other Models of Computation</b>		
<b>7</b>	<b>Rewriting</b>	79
7.1	Proving Negative Results	79
7.2	Rewriting	80
7.3	Normal Forms	82
7.4	The Diamond Property	83
7.5	Confluence of Tree Calculus	86
7.6	The Halting Problem	88
7.7	Standard Reduction	88
7.8	Converse Properties of the Self-Evaluators	90

<b>8</b>	<b>Incompleteness of Combinatory Logic</b>	95
8.1	Logic Without Variables	95
8.2	<i>SK</i> -Calculus	96
8.3	Combinators in <i>SK</i> -Calculus	97
8.4	Incompleteness of Combinatory Logic	98
8.5	Meaningful Translation	101
8.6	No Tree Calculus in Combinatory Logic	103
<b>9</b>	<b>Lambda-Abstraction in <i>VA</i>-Calculus</b>	105
9.1	First-Class Abstraction	105
9.2	<i>VA</i> -Calculus	108
9.3	Combinators	109
9.4	First-Class Substitutions	111
9.5	Incompleteness	111
9.6	Translation to Tree Calculus	112
9.7	Tagging	113
9.8	Translation from Tree Calculus	113
<b>10</b>	<b>Divide-and-Conquer in <i>SF</i>-Calculus</b>	119
10.1	Divide and Conquer	119
10.2	<i>SF</i> -Calculus	120
10.3	Intensional Examples	121
10.4	Translation from Tree Calculus	123
10.5	Translation to Tree Calculus	124
<b>11</b>	<b>Concluding Remarks</b>	129
11.1	Recapitulation	129
11.2	Program Completeness	130
11.3	Extensional, Intensional and Reflective Programs	131
11.4	Translations to Tree Calculus	132
11.5	Copying is Meta-Theoretic	132
11.6	Implications for Programming Languages	133
11.7	Implications for Logic	133
11.8	Implications for Mathematics	134
11.9	Implications for Traditional Computing Theory	135
<b>A</b>	<b>Confusion in the Church-Turing Thesis (w. Jose Vergara)</b>	137
A.1	Introduction	138
A.2	Confusion Leads to Error	141
A.3	Church's Thesis	142
A.4	Turing's Thesis	144
A.5	The Church-Turing Thesis	145
A.6	Models of Computability	146
A.7	Comparison of Models	147
A.8	Programming Language Design	150
A.9	Extensional and Intensional Computation	152

A.10 <i>SF</i> -Calculus .....	153
A.11 Conclusions .....	154
<b>B Illustrations</b> .....	157
<b>Glossary</b> .....	159

# List of Figures

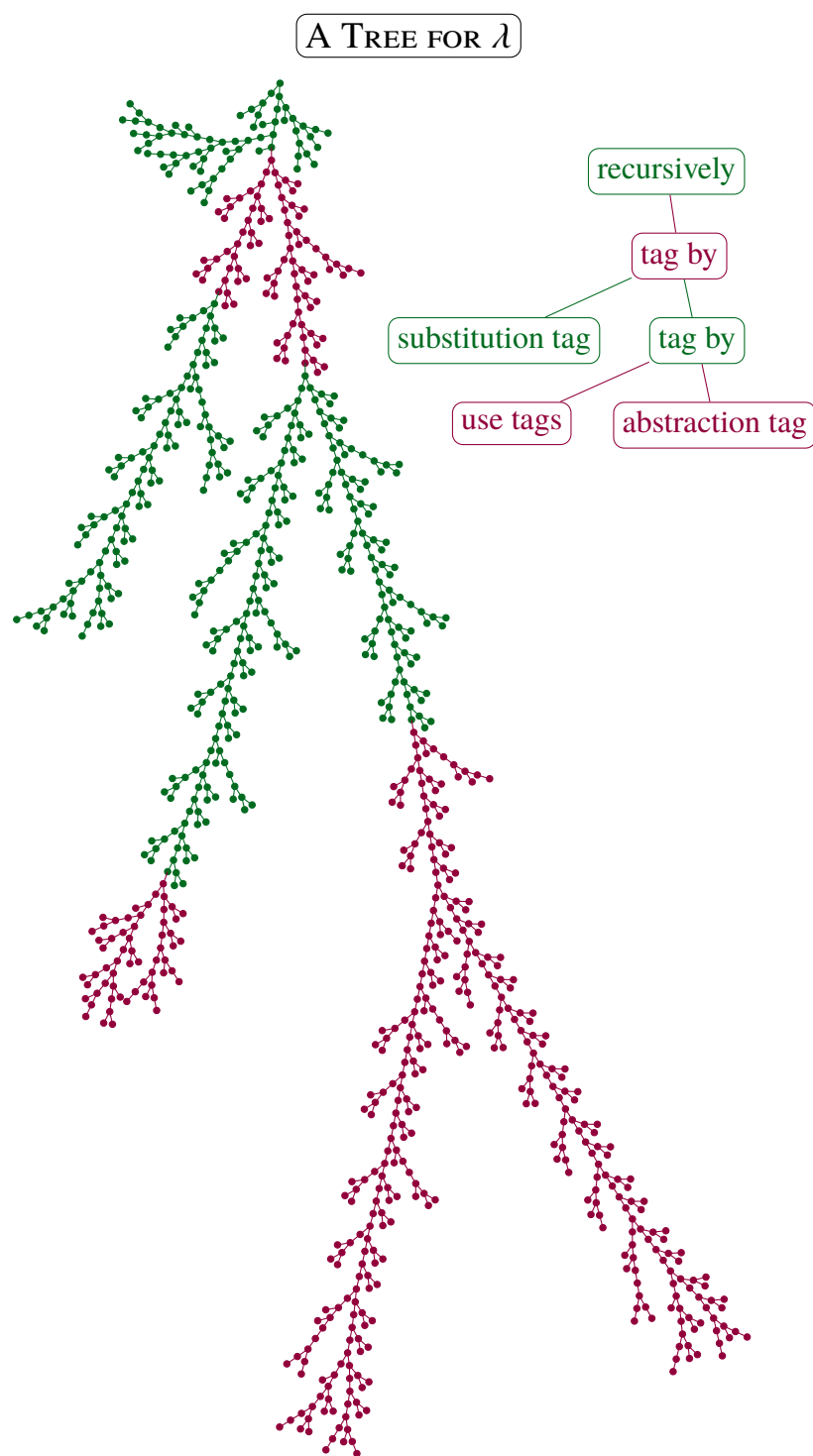
1.1	Polish Notation for the Branch-First Evaluator in the Frontispiece . . .	9
2.1	Table of Decimal Addition . . . . .	12
2.2	Table of Binary Addition . . . . .	12
2.3	Derivations of Two Roman Numerals . . . . .	14
3.1	Tree Equations . . . . .	28
3.2	The Combinations $K, I$ and $D$ . . . . .	29
3.3	Two Representations of $S$ . . . . .	30
3.4	Some Boolean Operations . . . . .	32
3.5	The Zero Test and the Predecessor . . . . .	34
3.6	Parametrising the Fundamental Queries . . . . .	36
3.7	The Fundamental Queries . . . . .	36
4.1	The Program $W$ . . . . .	45
4.2	The Program $Y_2\{f\}$ . . . . .	47
4.3	The Program <code>plus</code> . . . . .	48
4.4	The String “small” . . . . .	50
4.5	Mapping and Folding over Lists . . . . .	51
5.1	The Size of Programs as a Function of <code>plus</code> . . . . .	55
5.2	The Program <code>equal</code> . . . . .	56
5.3	Tagging . . . . .	58
5.4	The Program <code>type_check</code> . . . . .	60
5.5	Pattern Matching . . . . .	62
6.1	Branch-First Evaluation . . . . .	67
6.2	A Branch-First Evaluator . . . . .	68
6.3	Quotation . . . . .	71
6.4	Root Evaluation . . . . .	71
6.5	A Root Self-Evaluator . . . . .	72
6.6	Root-and-Branch Evaluation . . . . .	72

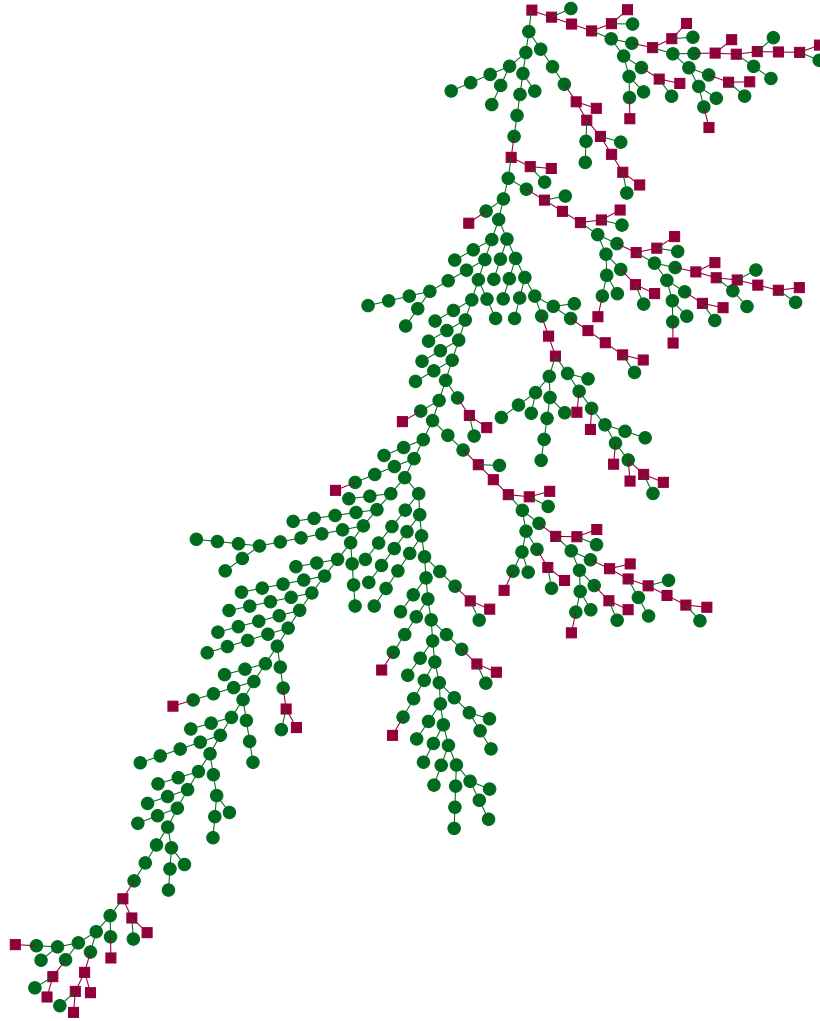
7.1	Reduction may Duplicate Redexes .....	86
7.2	Simultaneous Reduction in Tree Calculus .....	87
7.3	Simultaneous Reduction can Handle Duplicate Redexes .....	87
7.4	Branch-First Evaluation Using the Fork Rule.....	91
7.5	Branch-First Evaluation Using the Stem Rule .....	92
9.1	An Invalid Argument .....	108
9.2	Evaluation Rules of $VA$ -Calculus .....	109
9.3	A Schematic Translation of $A$ .....	112
9.4	The Translation of $V$ to Tree Calculus .....	114
9.5	The Translation of $\Delta$ to $VA$ -Calculus: Circles are $V$ and Squares are $A$ 116	
10.1	The Translation of $\Delta$ to $SF$ -Calculus: Circles are $S$ and Squares are $F$ 124	
10.2	The Translation of $S$ and $F$ to Tree Calculus Using <code>ternary_op</code> .....	126
A.1	A faulty argument, with premises quoted from Kleene [?] .....	141
A.2	Recoding is not $\lambda$ -definable .....	150
A.3	Recoding is $SF$ -definable but not $\lambda$ -definable .....	154



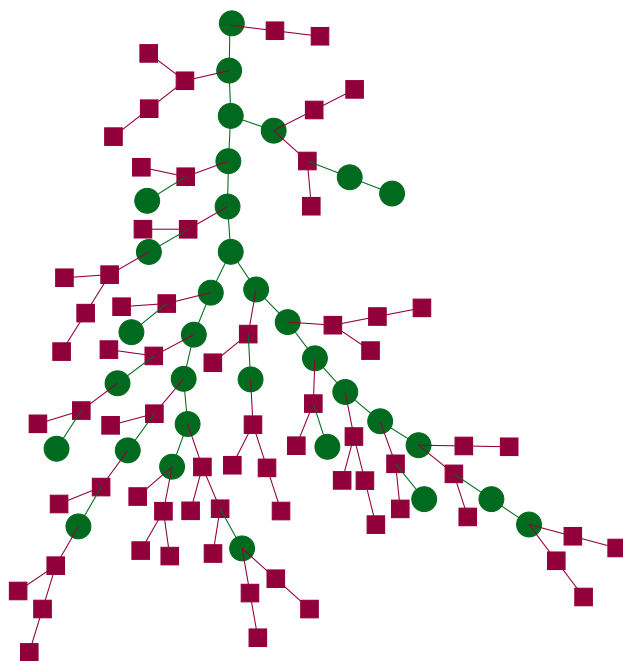
1. I've just finished drafting my new book "Reflective Programs in Tree Calculus". Here is the frontispiece.
2. Good computer science needs theorems (to show that bad things don't happen) an implementation (to show that good things do happen) and a story (that ties it all together).
3. At heart, computer science is the study of programs. An implementation is an interpreter or compiler, i.e a program that acts on programs. In a good story, all these programs have equal status, so interpreters can be self-applied. This requires a theory of reflective programs.
4. Turing machines do reflection by encoding machines as numbers on a tape. But this computation is outside of the model, so Turing machines don't make a good theory of computation! Nor do natural numbers.
5. Lambda calculi do most reflection by encoding terms as syntax trees. But this computation is outside of the model, so lambda calculus doesn't make a good theory of computation! Nor does combinatory logic.
6. So reflective programming is hard, but no harder than operating on your own brain.
7. Syntax trees support reflection but are artificial, so use natural trees, i.e. without labels, as in the frontispiece shown before.
8. Let parsing convert syntax to natural trees. Meet Fir (a tree made of blank tape).
9. In tree calculus, the programs, functions and data structures are exactly the binary trees, which are leaves, stems or forks. Other trees are computations. All these can be written as combinations of a single operator called Node, drawn as a small triangle. Here are its rules.
10. The rules K and S delete or copy the third argument, so tree calculus: supports all combinators; has macros for lambda-abstraction; and is Turing-complete. The rule (F) adds factorisation, as in SF-calculus.
11. When recursion is expressed by fixpoints, the functions are traditionally not stable, without normal form, but in tree calculus they are. Any instability is caused by function application. Meet Shasta, the flower of recursion.
12. Tree calculus also supports program analysis, such as a test for program equality.
13. Such program analysis is not possible in traditional models, as Fir understands.
14. A tree calculus program  $f$  can be tagged with a comment, type or other guidance  $t$  that can be recovered by applying a query to it, but behaves like  $f$  when it is applied:  $\text{tag}\{t, f\} = \Delta(\Delta t)(\Delta(\Delta f)(\Delta\Delta(\Delta\Delta)))$ . Tags could support just-in-time program analysis.
15. Self-interpretation for branch-first (eager) evaluation and root-first (lazy) evaluation uses trees that are very small (<1000 nodes). Maybe self-awareness is not so complicated :)
16. Combinatory logic is incomplete since equality is not definable. Also, it has no \*meaningful translation\* of tree calculus.
17. VA-calculus supports variables and also lambda-abstractions that have environments, much as closures do. This replaces all the meta-theory, for substitution, variable renaming etc, with 7 simple equations.

18. Although VA-calculus is incomplete (equality is not definable) there is a \*meaningful translation\* to it from tree calculus. Here is the translation of Node to a combination of V and A, given as a labelled tree.
19. The coq proofs for "Reflective Programs in Tree Calculus" are now available at <https://github.com/barry-jay-personal/tree-calculus/> and as the opam package `coq-tree-calculus` (logpath Reflective).
20. There is also a meaningful translation from VA-calculus to tree calculus. Here is the tree for A.
21. SF-calculus supports divide-and-conquer algorithms, such as equality or pattern-matching, using combinations built from operators S and F.
22. There are meaningful translations between SF-calculus and tree calculus.





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



**Fig. 0.2** The Translation of  $\Delta$  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  $\lambda$ -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 that have exactly two branches.

**general recursion** describes a large class of recursive functions; see also  $\mu$ -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  $V$ .

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