Barry Jay

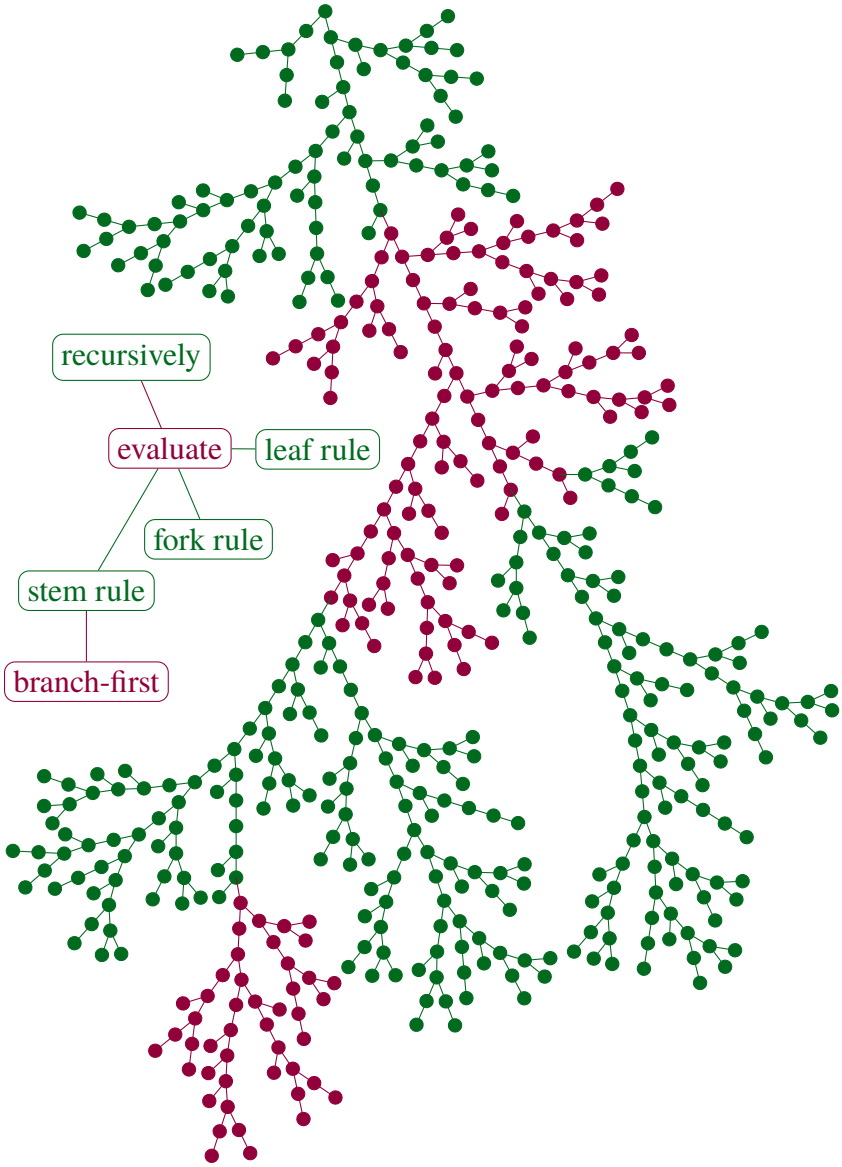
Reflective Programs in Tree Calculus

DRAFT

October 12, 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

Tree calculus emerged as a response to an anonymous negative review which asked, who cares about programs in your SF-calculus? This was the spur that pushed me to consider natural trees instead of artificial ones. More generally, every comment in every review has helped to sharpen these ideas.

Thanks go to all those who have read or commented on drafts of this work, including Justin Anderson, Thomas Given-Wilson and Alberto Vergara. Xuanyi Chew and Eric Torrebore read the second draft with great care, and made many valuable observations. I would especially like to thank my old friend Paul Pilipowski. An electrical engineer by profession, Paul tested my claim that high school mathematics was sufficient background, by patiently reading the whole of the first draft. Among other things, he pushed me to include a glossary. Adding illustrations, as a way of lightening the mood, was discussed with Benjamin Marks and Rodney Marks. Simultaneously, Cathy Lockhart assisted at the birth of Shasta and Fir, as I wibble-wobbled my way through the illustrations.

Aside from the appendix written with Alberto, all remaining errors are my sole responsibility.

Contents

List of Figures

Part I
Tree Calculus

Chapter 1
Introduction

1.1 Reflective Programs

Although the theory of computing began with the study of programs that act on numbers, the arrival of compilers and interpreters introduced programs that act on other programs, which is altogether more challenging. In particular, if an interpreter is written in the same language as that of its inputs, if the interpreter can be applied to itself, then the language supports *reflective programs* that are able to look in the mirror and see their own structure.

In such a language, programs have a dual nature, as both functions and data structures. This dual nature is hard to capture within the traditional models of computation, which emphasise one or the other. If programs are defined to be data structures, e.g. to be sequences of instructions, then some external machinery has been required to make them function. For example, Alan Turing showed how to represent a program as a sequence of symbols on a tape, but then some other machinery (a universal machine) is required to express the functionality. Alternatively, if programs were defined to be functions, as in the lambda-calculus of Alonzo Church, then external machinery has been required to convert functions into data structures, e.g. *serialisation* for their Gödel number or *quotation* for their syntax tree. In all cases, to support

reflection has required some additional machinery, functionality or theory that lies outside of the computational model in question.

In this sense, all of the traditional models of computation are lacking in some manner. No matter how good they are at numerical computation, or at manipulating functions, they have not been able to provide a satisfactory account of programs that act on programs, of reflective programming.

1.2 Tree Calculus

This book eliminates the need for additional machinery by introducing a new model of computation, tree calculus, in which computations are unlabelled, finitely-branching trees, among which the binary trees play all the other roles, of functions with their arguments and values, of programs with their inputs and results. Clearly, these trees are data structures. Indeed, since there are no labels, these *natural trees* consist of pure structure, just like the natural numbers. Their functionality arises by treating the node constructor $\triangle$ as a ternary operator with the three evaluation rules

$$\begin{aligned} \triangle\triangle yz &= y \\ \triangle(\triangle x)yz &= (yz)(xz) \\ \triangle(\triangle wx)yz &= zwx\ . \end{aligned}$$

Evaluation proceeds according to whether the first argument is a leaf $\triangle$ with no branches, a stem $\triangle x$ with one branch x or a fork with two branches w and x. We will explore these rules at length in Chapter 3. For now, it is enough to appreciate their simplicity.

Although the rules are simple, they are powerful. For example, the frontispiece shows a reflective program for self-interpretation in which function arguments are evaluated before the function itself. The main goal of Part I is to develop the machinery required for reflective programs like this. The development proceeds in five steps, in five chapters.

Chapter 2 introduces the equational reasoning tools required to define a self-interpreter. This amounts to little more than the ability to substitute equals for equals, much as in high school algebra, though the settings will be novel. It also introduces inductive definitions, which will be used passively throughout, though an inductive proof will feature at the end of Part I.

Chapter 3 introduces tree calculus and shows how to build tests for being a stem, leaf or fork. It defines the programs and normal forms to be the binary trees. When reduction rules are introduced in Part II, the normal forms will prove to be exactly the irreducible terms.

Chapter 4 considers functions that treat their argument t as a function whose internal structure is hidden. Thus, the only information which can be revealed about t concerns its input-output relation, i.e. its *extension*. Such extensional functions are the traditional preserve of functional programming so it is convenient to develop

some machinery for binding parameters (as in lambda-abstraction) and also for defining recursive functions as fixpoints. Unlike traditional accounts, the fixpoint construction here preserves normal forms, and so preserves programs in the sense above.

Chapter 5 considers functions that treat their argument t as a data structure, whose internal structure can be revealed by queries. In particular, queries can discover the algorithm that t implements, its *intension*. Examples include programs that compute the size of a program or test the equality of programs. It also shows how a program may be *tagged* with extra information that does not affect its extensional behaviour, its functionality, but can be recovered by intensional means, using queries. Of course, tags can be used to record the intentions of the programmer, too, including type information.

Chapter 6 produces some self-interpreters, or more precisely, *self-evaluators* for tree calculus. The first uses a *branch-first* evaluation strategy: the last uses a *node-first* evaluation strategy. These are reminiscent of eager and lazy evaluation respectively. Such *reflective* programs use all of the extensional and intensional machinery developed earlier. Previous accounts of self-interpretation had to work with programs that were not normal forms, that were unstable. Stability was imposed by first *quoting* the program to produce a data structure, by putting on some make-up. In tree calculus, the programs are already data structures, so that no pre-processing is required; both of the self-evaluators above act on the program and its input directly. In short, tree calculus supports honest reflection without make-up.

1.3 Other Models of Computation

The main goal of Part II is to compare tree calculus to three other models of computation: combinatory logic, λ-calculus and factorisation calculus. The translation of λ (or, more precisely of the abstraction operator A) is given by the tree at the beginning of Part II.

Since tree calculus supports the standard arithmetic functions, it is Turing complete in the traditional sense, and, in that sense, is equivalent to all the other models under consideration. However, the traditional analysis relies on translations between calculi that, while they respect the results of computation, need not respect the computation process itself. For example, the usual translation of λ-calculus (Chapter 9) to *combinatory logic* (Chapter 8) breaks some evaluations.

The computation process is respected by the *meaningful translations* introduced here. For each model, there is a meaningful translation to tree calculus but there are not always meaningful translations in other directions. For example, there is no meaningful translation of tree calculus to combinatory logic. Nor is there one from our λ-calculus, called *VA*-calculus, to combinatory logic.

The development proceeds in four steps, in four chapters, followed by a chapter of conclusions and an appendix.

Chapter 7 introduces *rewriting theory* and recasts tree calculus as a rewriting system. This allows us to prove some basic properties of tree calculus, especially its *soundness*, e.g. that forks are never leaves, and will facilitate the comparisons that follow.

Chapter 8 shows how to embed *combinatory logic* into tree calculus, which is easy. Much harder is to prove that there is *no* meaningful translation from tree calculus to combinatory logic. The key idea is to show that combinatory logic cannot separate identity programs, that it cannot distinguish programs M and N that act as the identity function. So, combinatory logic is, in this sense, incomplete. On the other hand, tree calculus can separate identity programs, and this property is preserved by meaningful translation, so there can be no such translation from tree calculus to combinatory logic.

Chapter 9 introduces *VA-calculus*, an equational variant of λ-calculus built from operators for variables and abstraction. It has evolved from my earlier creation, *closure calculus*, that has been further explored by Xuanyi Chew. There is a meaningful translation from *VA*-calculus to tree calculus. Interestingly, there is also a meaningful translation from tree calculus to *VA*-calculus that is made possible because *VA*-calculus supports tagging. The tags are stored in the environments of abstractions, a technique that is not available within traditional lambda-calculus. However, *VA*-calculus is not fully intensional as it does not support equality of programs, so that it is not well suited to reflective programming.

Chapter 10 recalls the factorisation calculus that is *SF-calculus*, developed by Thomas Given-Wilson and me. It has very similar expressive power to tree calculus, and there are meaningful translations in both directions. It gives a more direct account of divide-and-conquer algorithms, which simplifies the treatment of data structures. However, it is less attractive than tree calculus for two reasons. One is that it has more operators and evaluation rules. The other is that it uses trees with artificial labels S and F, and not natural trees.

Some other models of computation have *not* been considered for detailed treatment. For example, the μ-recursive functions act on numbers, not μ-recursive functions and so cannot support reflection. Again, Turing machines act on tapes, not Turing machines. Some elaborate encoding is required to patch over these difficulties, with the effect of pushing the computational effort out of sight. For example, Steven Cole Kleene introduced *partial combinatory algebras* to allow numbers to be functions, but their evaluation is not axiomatic, as it requires decoding of numbers to functions. To illustrate, Figure 1.1 shows the encoding of the branch-first self-evaluator of the frontispiece in Polish notation using ternary numerals (so that, for example, the ternary numeral 201 is the same as the decimal $2 * 3^2 + 1 = 19$). Reconstruction of the function from the numeral must recover all of the tree structure. Again, Turing machines can be encoded as tapes, but then the application of one tape to another requires Turing's *universal machine* to do the interpretation.

These properties of tree calculus show that it is distinct improvement over traditional models of computation because it is able to express computations that older models cannot express, without importing additional machinery. This may have profound implications for both the design of programming languages. It may even impact

```
212101021202120210102121212020112101021212121212110200200201020112121202121010
```

Fig. 1.1 Polish Notation for the Branch-First Evaluator in the Frontispiece

on the foundations of mathematics and logic. Certainly, it requires us to re-examine conventional beliefs about the nature of computation. Although it is tempting to discuss these possibilities immediately, it is better to leave future work to the concluding chapter, and the critique of the past ideas to the appendix. In this way, the formally verified theorems that are central to this work will not be overshadowed by speculation or controversy.

1.4 How to Read this Book

By and large, each chapter of the book builds on those before it so, from a logical point of view, there is not much opportunity to skip ahead. For example, reflective programs are built using a combination of extensional and intensional techniques. Also, the examples, proofs and exercises begin simply but complexity grows significantly. Fortunately, all the proofs of the named theorems have been verified in Coq and can be found among the materials accompanying the book at https://github.com/barry-jay-personal/tree-calculus/. These formal proofs are substantial, being larger than the book itself, so perhaps your first reading should focus on the ideas, leaving formal proofs for later.

This book is intended for all those who are interested in the theory of computation, from students to experts. Students do not need any particular exposure to computing theory or mathematics though some awareness of general principles will be useful; there is a glossary of all the main terms mentioned in this introduction. All ideas required for the technical development will be introduced as required but with minimal exposition. Each chapter after the introduction includes some exercises; where appropriate, solutions have been included with the proofs in Coq. Part I begins with an account of equational reasoning and inductive definitions. This is enough to follow all of Part I except the last proof, about the properties of a self-evaluator, which uses induction within a proof. Part II opens with a chapter on rewriting theory, that will be used to compare models of computation. By the end of the book, you will have learned about a wide variety of models of computation, and also some of the standard data structures, including booleans, natural numbers, tuples, lists and strings.

Experts in computing theory, especially in rewriting, may have two possible concerns. On the one hand, many constructions, e.g. of fixpoints, will be familiar, so there is the risk of boredom. On the other hand, many of the theorems conflict with conventional opinions about the nature of computation, so there is the fear of pollution. To counter boredom, I have included something new in every chapter. Here is a partial list.

Chapter 2 uses Roman numerals as a running example of a computing system. They will be familiar to many readers and yet provide a convenient setting in which to explore some of the hidden assumptions of elementary algebra. For example, when the numerals IV and VI are different numbers (four and six) then the commutativity of juxtaposition does not apply.

Chapter 3 introduces tree calculus, and defines the fundamental queries, for being a leaf, stem or fork.

Chapter 4 introduces the idea of *waiting*. An "application" $\mathsf{wait}\{M,N\}$ must wait for another argument P before applying M to N to get $\mathsf{wait}\{M,N\}P = MNP$. In turn, this is used to define fixpoint functions in normal form, that must wait for their argument before evaluation.

Chapter 5 shows how to tag a function with additional, recoverable information without changing its functionality.

Chapter 6 introduces a self-evaluator without the need for any of the usual outside machinery, such as quotation.

Chapter 7 shows that the halting problem is insoluble within tree calculus, without appealing to external machinery. Also, it shows that the self-evaluators of Chapter 6 do no more evaluation than allowed by the corresponding evaluation strategy.

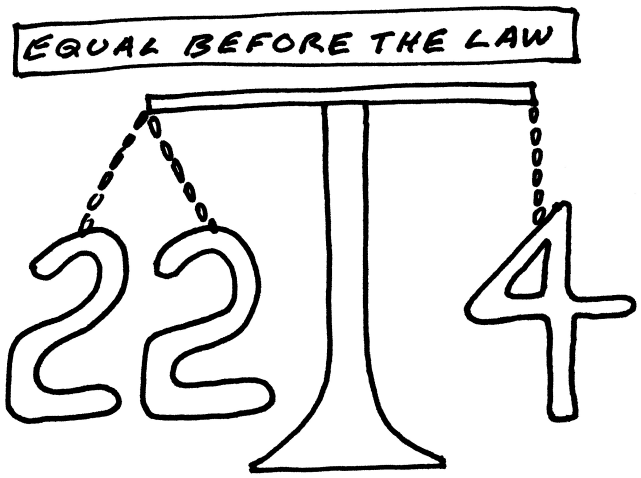
Chapter 8 shows that there is no meaningful translation of tree calculus to combinatory logic.

Chapter 9 introduces an equational variant of λ-calculus, in which abstraction is represented by the free-standing operator A.

Chapter 10 recalls SF-calculus, and shows how there is a meaningful translation of it into tree calculus, even though the translation of F is *not* a factorisation function in tree calculus.

The final concern is the fear of pollution. If this book conflicts with what you already know then it is natural to fear its corrupting influence. If your first goal is to consider the clash of ideas or if you want some historical context, then you could begin with the appendix. Written with Jose Vergara in 2015 but refused publication till now, it analyses the weaknesses inherent in the traditional, verbal, account of computation. However, I encourage you to save that till the end. After all, there is nothing to fear from a few simple axioms, some equational reasoning and some pictures. At worst, you will have some new tricks: at best, you will see computation in a new light.

Chapter 2
Equational Reasoning



2.1 Addition

Some of the core issues of computation can be seen in arithmetic. This chapter uses addition to illustrate the fundamentals of computation by equational reasoning, for Roman numerals and then for a small language of arithmetic expressions. The numerals are given by inductive definitions. Their equality will be defined by introducing a sequence of rules, from the general rules for equivalence relations and congruences, to the special rules that carry the meaning of these numerals. Proofs about the translation from Roman numerals to arithmetic expressions proceed by structural induction.

When children begin adding systematically, they commonly begin by memorising a table, as in Figure 2.1. Of course, memorisation can only take us so far, but the power of zero allows us to extend this to a general algorithm, for performing long addition, that uses positional notation and the principle of "carrying one" to add arbitrarily large numbers. For example, we have

+	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	10
2	2	3	4	5	6	7	8	9	10	11
3	3	4	5	6	7	8	9	10	11	12
4	4	5	6	7	8	9	10	11	12	13
5	5	6	7	8	9	10	11	12	13	14
6	6	7	8	9	10	11	12	13	14	15
7	7	8	9	10	11	12	13	14	15	16
8	8	9	10	11	12	13	14	15	16	17
9	9	10	11	12	13	14	15	16	17	18

Fig. 2.1 Table of Decimal Addition

$$\begin{array}{r} 2\ 0\ 4\ 7 \\ +\ 1\ 2\ 0_1\ 5 \\ \hline 3\ 2\ 5\ 2 \end{array}$$

in which the addition of 7 and 5 produces 12; put down the 2 and carry the 1. Note how the positioning of the zeroes is used to distinguish the 4 tens from the 2 thousands.

This approach works well enough but has two drawbacks. The first is the need to memorise the table. The second is the reliance on positional notation, which is a geometric, rather than a logical device.

The size of the table can be reduced from 10×10 to 2×2 by shifting from arithmetic in base 10 to base 2, or binary arithmetic, in which the only digits are 0 and 1. Now the table of addition is given in Figure 2.2 where 10 now indicates that there is one 2 and no 1's. Equally, we could use ternary arithmetic, as in Figure 1.1 or some other base.

+	0	1
0	0	1
1	1	10

Fig. 2.2 Table of Binary Addition

The tables can be eliminated altogether by using *tallies*. Now 7 + 4 is represented by

$$||||||| + |||| = ||||||||||$$

in which the lists of tallies are simply merged. Of course, this makes our eyes dance, and becomes completely impractical for long addition.

The Romans improved on the tallies by introducing some symbols for larger numbers, as in the following table

I	V	X	L	C	D	M
1	5	10	50	100	500	1000

Now 7 + 4 yields

$$VII + IIII = XI\ .$$

but we don't have to memorise this. Instead, we just need to memorise some definitions

$$\begin{aligned} V &= IIIII \\ X &= VV \\ L &= XXXXX \\ C &= LL \\ D &= CCCCC \\ M &= DD\ . \end{aligned}$$

Then we have

$$VII + IIII = VIIIIII = VVI = XI\ .$$

That is, addition is once again given by merging, but then followed by some tidying up. For the sake of simplicity, we will follow the earlier Roman convention in which IV is the same as VI so that

$$I + V = IV = VI\ .$$

Our long addition example works similarly, in that 2047 + 1205 becomes

$$\begin{aligned} MMXXXXVII + MCCV &= MMXXXXVIIMCCV \\ &= MMMCCXXXXVVII \\ &= MMMCCXXXXXII \\ &= MMMCCLII\ . \end{aligned} \tag{2.1}$$

The second equation above groups together all of the Ms, all of the Cs etc. That is, it collects *like-terms*. The third and fourth equations apply the definitions of X and L to contract the result. Of course, we could apply the equations differently, to produce 3252 copies of I or $IILCCMMM$ but $MMMCCLII$ is the traditional answer.

This system proved to be a good compromise that retains most of the simplicity of the tally system, and most of the power of the Arabic numerals in Figure 2.1 One still has to memorise a little, and it doesn't work for numbers over a few thousand, but overall it's pretty good.

2.2 Inductive Definitions

When studying computation, it is usual to define the set of terms inductively, by specifying some constants and some rules for building new terms from old.

For example, unary numerals are given by a single constant, zero and single rule: if n is a numeral then successor n is a numeral. This information can be expressed in tree form, by

$$\frac{}{\vdash \mathsf{zero}} \qquad \frac{\vdash n}{\vdash \mathsf{successor}\ n}\ .$$

The syntax $\vdash n$ is a *judgment* that n is a (well-formed) numeral. The horizontal lines are *inferences*, that the *conclusion* below the line follows from the *premises* above the line. The whole tree is called a *derivation*. In a more compressed form, we can express the constructions as a Backus-Naur Form, or BNF for short, as follows

$$n ::= \mathsf{zero} \mid \mathsf{successor}\ n\ .$$

It asserts that an n is either zero or the successor of some n. In a sense, unary arithmetic adds zero to the system of tallies.

Here is a BNF for the Roman numerals:

$$r, s ::= I \mid V \mid X \mid L \mid C \mid D \mid M \mid (r\ s)\ .$$

It is to be read as follows. The symbols r and s denote Roman numerals. The possible forms for Roman numerals come after the symbol ::= and are separated by vertical bars | . There are eight possible forms. Seven of them are the constants I, V etc. The final form is the *concatenation* $(r\ s)$ of two Roman numerals r and s. To support reasoning about Roman numerals, it is important to ensure that each numeral is constructed in a unique manner, so that the order in which concatenation is performed matters. For example, the numerals

$$((XV)I) \quad \text{and} \quad (X(VI))$$

are distinct representations of the number sixteen, just as 15 + 1 and 10 + 6 are. The differences can be highlighted by examining their derivations in Figure 2.3.

$$\dfrac{\dfrac{\dfrac{}{\vdash X} \quad \dfrac{}{\vdash V}}{\vdash (XV)} \quad \dfrac{}{\vdash I}}{\vdash ((XV)I)} \qquad \dfrac{\dfrac{}{\vdash X} \quad \dfrac{\dfrac{}{\vdash V} \quad \dfrac{}{\vdash I}}{\vdash (VI)}}{\vdash (X(VI))}$$

Fig. 2.3 Derivations of Two Roman Numerals

Of course, the profusion of brackets makes the notation heavy, so let us adopt the conventions that outermost brackets may be dropped and that concatenation

associates to the left, so that a numeral of the form

$$r\ s\ t$$

is actually

$$((r\ s)\ t)\ .$$

For example, $((XV)I)$ may be written XVI and $(X(VI))$ may be written $X(VI)$.

We can modularise the inductive definition of Roman numerals by separating the constants, or operators, from the constructed numerals by defining

$$\begin{aligned} O &::= I \mid V \mid X \mid L \mid C \mid D \mid M \\ r, s &::= O \mid r\ s\ . \end{aligned}$$

That is, O represents an arbitrary operator. Roman numerals are either operators or concatenations of Roman numerals. In this manner, we can add or remove operators without changing the general nature of the numerals. When two Roman numerals r and s are identical, and not merely equal according to our rules, we may write this as $r \equiv s$. The same convention applies for any other inductively-defined terms.

2.3 Notational Conventions

This seems like a convenient point at which to mention notational conventions. Upper-case letters such as O, I, C, M may be used to denote operators but we will also use $M, N, P, Q \ldots$ to denote combinations of operators. We rely on the context to make clear which meaning is intended. The lower case letters $m, n, p, q, r, s, t, u, v$ will be used to denote terms appearing in some BNF. Other lower case letters x, y, z may be used to denote variables in two distinct ways. For example, when upper case letters represent operators or particular combinations then lower case letters will be used to represent their arguments. For example, if K is an operator its evaluation rule may be given by

$$Kxy = x\ .$$

Alternatively, we may wish to bind a variable x in some arbitrary term t as in

$$\lambda^* x.t\ .$$

In this case, the syntactic class of terms will include a sub-class of variables such as x which are *not combinations*. Since these situations tend not to overlap, we will rely on the context to make clear which conventions are being used.

Words in sans-serif font, such as zero, successor, quote, depthfirst are used to represent programs. Further, curly braces may be used to indicate the parameters of a program, e.g. wait$\{M, N\}$. In general, spaces between terms represented by single letters may be elided if this eases legibility, so that $r\ s$ may be written rs. Of course, we may *not* write quote$s$ for quote s or unquote for $u\ n$ quote.

2.4 Equivalence Relations

Now let us justify the equational reasoning of (2.1) by defining the equality relation = on the inductively defined Roman numerals above. This will proceed in four steps, as we introduce the rules for: equivalence relations; congruences; collecting like-terms; and characterising the operators. We will quickly cover all of this material here, and use it to understand the Roman numerals, but if you haven't seen any of it before then you may want to consult a textbook on discrete mathematics.

Let T be a set, whose elements are to be thought of as terms. Let $\sim$ be a binary relation on T that is written *infix*, so that if s and t are elements of T then $s \sim t$ asserts that s and t are related by $\sim$. Then $\sim$ is an *equivalence relation* if it satisfies the following three rules:

(Reflexive) $t \sim t$ for all t in T.
(Symmetric) If $s \sim t$ then $t \sim s$ for all s and t in T.
(Transitive) If $r \sim s$ and $s \sim t$ then $r \sim t$ for all r and s and t in T.

In particular cases, we may use the symbol = instead of $\sim$. For example, the chain of equalities in (2.1) implies

$$MMXXXXVIIMCCV = MMMCCLII$$

by transitivity.

2.5 Congruences

Where T above is inductively defined, then an equivalence relation is a *congruence* if equals can be replaced by equals in all of the inductive constructions. For Roman numerals, there is only one (non-trivial) construction, so we make = a congruence by adding the following rule

(Congruence) If $r_1 = r_2$ and $s_1 = s_2$ then $r_1 s_1 = r_2 s_2$ for any Roman numerals r_1, r_2, s_1 and s_2.

For example,

$$LLVV = CX$$

since $LL = C$ and $VV = X$.

2.6 Collecting Like-Terms

Also, we need rules for collecting like terms. The rules for associativity and commutativity are

(i) (Associative) $(r\ s)\ t = r\ (s\ t)$.

(ii) (Commutative) $r\ s = s\ r$.

In our example, these rules are used as follows.

$$\begin{aligned}
&(MMXXXXVII)(MCCV)\\
&= (MXXXXVII)M(MCCV) \text{ (commutativity, congruence)}\\
&= (MXXXXVII)(MMCCV) \text{ (associativity)}\\
&= \ldots\\
&= MMMCCXXXXVVII\ .
\end{aligned} \tag{2.2}$$

2.7 Rules for Operators

Having collected the like-terms, we finish up with some rules for simplification, that characterise the operators in terms of I:

$$\begin{aligned}
V &= IIIII\\
X &= VV\\
L &= XXXXX\\
C &= LL\\
D &= CCCCC\\
M &= DD\ .
\end{aligned}$$

In our example, this supports

$$MMMCCXXXXVVII = MMMCCXXXXXII = MMMCCLII\ .$$

This completes our description of the equality of Roman numerals: three rules for the equivalence relation; one for the congruence; two for collecting like-terms; and six for characterising the operators.

2.8 Arithmetic Expressions

The equality of Roman numerals requires, as well as the rules for congruence, eight rules for collecting like terms and characterizing the operators. This adds eight cases to simple proofs about equality, and dozens of steps to proofs that consider pairs of equations.

We can simplify the system by representing the numbers as unary numerals, built from zero and successor, while the concatenation of Roman numerals will be represented by addition.

Define the *arithmetic expressions* by

$$m, n ::= \mathsf{zero} \mid \mathsf{successor}\ m \mid (m + n)$$

As before, we eliminate superfluous brackets, on the understanding that the infix operators + associates to the left, and that successor binds more tightly than +. For example,

$$\mathsf{successor}\ (\mathsf{successor}\ m) + n + p = (((\mathsf{successor}\ (\mathsf{successor}\ m)) + n) + p)\ .$$

The arithmetic expressions that are constructed from zero and successor only, without addition, are the *unary numerals*.

The equality relation $m = n$ is defined as follows. First, it is an equivalence relation (reflexive, symmetric and transitive). Second, it is a congruence. Since there are three non-trivial ways to build arithmetic expressions, this requires the following three rules.

(i) $\mathsf{zero} = \mathsf{zero}$.
(ii) If $m = n$ then $\mathsf{successor}\ m = \mathsf{successor}\ n$.
(iii) If $m_1 = n_1$ and $m_2 = n_2$ then $m_1 + m_2 = n_1 + n_2$.

Note that there are two rules asserting that zero equals itself, namely the reflexive rule and the congruence rule. Finally, we need rules that characterise addition.

$$\mathsf{zero} + n = n$$
$$\mathsf{successor}\ m + n = m + (\mathsf{successor}\ n)$$

These rules cover all the possible ways of adding numerals, but do not give rules for expressions in general. For example, there is no specific rule for simplifying $m + n + p$. Rather, we must first simplify $m + n$ to become a numeral, so that the rules above will apply.

To check that this is always possible requires some proofs. We begin by considering the addition of numerals.

Theorem 2.1 *If m and n are unary numerals then there is a unary numeral p such that $m + n = p$.*

Proof The proof is by *mathematical induction* on m. In order to show that a property is true of all unary numerals, it is enough to show two things:

(i) zero has the property.
(ii) If m has the property then so does successor m.

For this theorem, the property of m is that:

For all numerals n there is a numeral p such that $m + n = p$.

Suppose m is zero and let n be any numeral. Then choose p to be n. Then $m + n$ is $\mathsf{zero} + n$ which equals n by the first rule for addition.

Suppose m is of the form successor m_1 and let n be any numeral. Then m_1 has the property with respect to the numeral successor n. Hence, there is a numeral p_1 such that $m_1 + \mathsf{successor}\ n = p_1$. Choose p to be p_1. Then $m + n$ is $\mathsf{successor}\ m_1 + n$.

Now successor $m_1 + n = m_1 +$ successor n by the second rule for addition, and $m_1 +$ successor $n = p$ so, by transitivity, we have $m + n = p$ as required. □

Theorem 2.2 *Every arithmetic expression is equal to a numeral.*

Proof Let e be an arithmetic expression. The proof is by *structural induction*, i.e. by induction on the structure of the expression e. This is a generalization of mathematical induction. In order to show that a property is true of all arithmetic expressions, it is enough to show three things:

(i) zero has the property.
(ii) If m has the property then so does successor m.
(iii) If m and n have the property then so does $m + n$.

For this theorem, the property of e is that there is a numeral n such that $e = n$. Here are the three cases.

If e is zero then choose n to be zero. Then zero = zero by reflexivity.

If e is successor e_1 and e_1 has the property, then there is a numeral n_1 such that $e_1 = n_1$. Choose n to be successor n_1. Then $e = n$ by congruence.

If e is of the form $e_1 + e_2$ and e_1 and e_2 have the property then there are numerals n_1 and n_2 such that $e_1 = n_1$ and $e_2 = n_2$. By theorem 2.1 there is a numeral n such that $n_1 + n_2 = n$. Then

$$e = e_1 + e_2 = n_1 + n_2 = n$$

by reflexivity, congruence, and the equality above, so $e = n$ by transitivity. □

Here are some further results which strengthen our confidence in this account of addition.

Theorem 2.3 *For all numerals m and n we have*

$$m + \text{successor } n = \text{successor } (m + n) \ .$$

Proof The proof is by induction on m. If m is zero then we have zero + successor n = successor n = successor (zero + n) by two applications of the first rule for addition, and congruence.

If m is of the form successor m_1 then

$$\text{successor } m_1 + \text{successor } n = m_1 + \text{successor } (\text{successor } n)$$

by the first rule of addition. Now the induction principle for m_1 with respect to successor n yields

$$m_1 + \text{successor } (\text{successor } n) = \text{successor } (m_1 + \text{successor } n)$$

which is equal to successor (successor $m_1 + n$) or successor $(m + n)$ by congruence and the first rule for addition. □

Theorem 2.4 zero *is a unit for addition of unary numerals. That is, for all unary numerals n we have*

$$\text{zero} + n = n = n + \text{zero} \ .$$

Proof The first equation holds by the first law of addition. For the second equation, proceed by induction on the structure of n.

If n is zero then we have $n + \mathsf{zero} = \mathsf{zero} + \mathsf{zero} = \mathsf{zero}$ by the first rule for addition.

If n is some successor successor n_1 then

$$\begin{aligned} n + \mathsf{zero} &= \mathsf{successor}\ n_1 + \mathsf{zero} \\ &= n_1 + \mathsf{successor}\ \mathsf{zero} \\ &= \mathsf{successor}\ (n_1 + \mathsf{zero}) \\ &= \mathsf{successor}\ n_1 = n\ . \end{aligned}$$

by the first rule of addition, and Theorem 2.3 and the induction hypothesis. □

Theorem 2.5 *Addition is commutative. That is, for all arithmetic expressions e_1 and e_2 we have*

$$e_1 + e_2 = e_2 + e_1\ .$$

Proof By Theorem 2.2, we may assume that e_1 and e_2 are numerals n_1 and n_2 respectively. Now proceed by induction with respect to n_1, to establish the following property:

For all numerals n_2, we have $n_1 + n_2 = n_2 + n_1$.

If n_1 is zero then apply Theorem 2.4. If n_1 is some successor successor n_3 then we have

$$\begin{aligned} n_1 + n_2 &= \mathsf{successor}\ n_3 + n_2 \\ &= n_3 + \mathsf{successor}\ n_2 \\ &= \mathsf{successor}\ n_2 + n_3 \\ &= n_2 + \mathsf{successor}\ n_3 = n_2 + n_1\ . \end{aligned}$$

□

Theorem 2.6 *Addition is associative. That is, for all arithmetic expressions e_1, e_2 and e_3, we have*

$$(e_1 + e_2) + e_3 = e_1 + (e_2 + e_3)\ .$$

Proof Without loss of generality, e_1, e_2 and e_3 are unary numerals n_1, n_2 and n_3 respectively. The proof is by induction on the structure of n_1. If n_1 is zero then the result is straightforward. If n_1 is some successor successor n_4 then we have

$$\begin{aligned}(n_1 + n_2) + n_3 &= (\mathsf{successor}\ n_4 + n_2) + n_3\\ &= (n_4 + \mathsf{successor}\ n_2) + n_3\\ &= n_4 + (\mathsf{successor}\ n_2 + n_3)\\ &= n_4 + (n_2 + \mathsf{successor}\ n_3)\\ &= n_4 + \mathsf{successor}\ (n_2 + n_3)\\ &= \mathsf{successor}\ (n_4 + (n_2 + n_3))\\ &= ((\mathsf{successor}\ n_4) + (n_2 + n_3))\\ &= n_1 + (n_2 + n_3)\ .\end{aligned}$$

□

2.9 Translation

In what sense do the Roman numerals and the unary numerals represent the same things? They both represent whole numbers, but the Roman numerals do not represent zero, so there are some differences. The relationship can be expressed by providing a translation $[-]$ from the Roman numerals to the arithmetic expressions that preserves all of the important information. Deciding what is important will be discussed more thoroughly in Part II, where translations play a central role but as the Roman numerals will not feature there, it seems best to develop this example immediately.

In this setting, a translation should map $r\ s$ to $[r] + [s]$ and it should preserve equality. There is a strong argument for requiring that I is translated by $\mathsf{successor\ zero}$ too. Here is a translation which will prove to have these properties.

$$\begin{aligned}[I] &= \mathsf{successor\ zero}\\ [V] &= [I] + [I] + [I] + [I] + [I]\\ [X] &= [V] + [V]\\ [L] &= [X] + [X] + [X] + [X] + [X]\\ [C] &= [L] + [L]\\ [D] &= [C] + [C] + [C] + [C] + [C]\\ [M] &= [D] + [D]\\ [r\ s] &= [r] + [s]\ .\end{aligned}$$

The first and last of the three properties above are true by definition.

Theorem 2.7 *Let r and s be two Roman numerals. If $r = s$ then $[r] = [s]$.*

Proof Once again, the proof is by structural induction. However, the proof is *not* by induction on the structure of r or s. Rather, it is by induction on the structure of the

proof of $r = s$. It may help to think of this proof as a tree, so that it is the structure of the tree which underpins the induction.

Recall that there are twelve rules for equality of Roman numerals: three for equivalence; one for congruence; two for collecting like terms; and six for characterising the operators. However, the proof is not so long.

Equality of arithmetic expression is also an equivalence relation, so the first three cases are immediate.

For congruence, suppose that $r_1 = s_1$ and $r_2 = s_2$. Then we have

$$[r_1 r_2] = [r_1] + [r_2] = [s_1] + [s_2] = [s_1 s_2]$$

where the middle equation holds by induction.

The rules for associativity and commutativity are preserved by Theorems 2.6 and 2.5.

The rules for characterizing the operators are preserved by the way in which the translation was defined. For example,

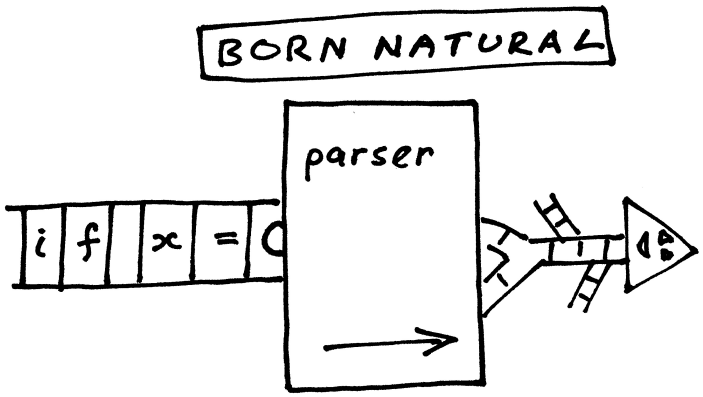
$$[X] = [V] + [V] = [VV] \,.$$

We do not expect there to be a translation in the opposite direction since there is no Roman numeral for zero. To prove this, it is enough to show that there is *no* Roman numeral r such that $rI = I$. However, such proofs are not easy: even proving that II is not equal to I is difficult. We shall return to this challenge after developing some more machinery, in Chapter 7.

Exercises

1. Convert the decimal numbers 2047 and 1205 to binary notation, add them together, and confirm that the result is the binary notation for the decimal numeral 3252.
2. Write out the 3×3 addition table for ternary arithmetic in which $2 + 2$ is 11.
3. Produce the tree derivation of the Roman numeral $CCLX$. Hint: add all the missing brackets to the numeral first.
4. Expand the equational reasoning in (2.1) so that each equation is justified according to the rules for equivalence, congruence etc. Don't forget the uses of reflexivity and symmetry. If you wish, use a tree of derivations.
5. Define one to be successor zero and two to be successor one and three to be successor two and four to be successor three. Now prove that two + two = four.
6. Using the theorems in the text, prove that $m + n + p = p + n + m$ for all arithmetic expressions m, n and p. Hint: add all the superfluous brackets first.
7. (Hard) In most presentations of Roman numerals, IV represents four, rather than six, so that there is a hint of positional notation involved. Describe how you would adapt the developments in this chapter to cope with this.

Chapter 3
Tree Calculus



3.1 Syntax Trees

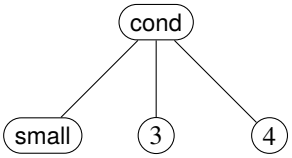
The standard life-cycle of a program begins with a sequence of characters which are chunked into a sequence of tokens, parsed to form a syntax tree, analysed, optimised and then converted to an executable for some machine. For example, consider the following program fragment. The sequence of characters

[i,f, ,s,m,a,l,l, ,t,h,e,n, ,3, ,e,l,s,e, ,4]

becomes the sequence of tokens

[if,small,then,3,else,4]

and then parsing produces the syntax tree



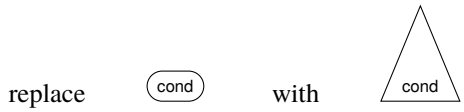
in which the if-then-else is represented by a node of the tree that has three branches, each of which has a label. If analysis shows that small is always true then this tree may be optimised to 3 but if small is a variable then the resulting executable will include the conditional.

Reflective programming allows this sort of program analysis to be done after the program has been converted to an executable, by recovering the syntax tree from the executable, optimising and then converting back to executable form. However, this is not well presented in traditional theories of computation. In the best case scenario, the language of the program being analysed is the same as the language in which the analysis is written but even then, the conversion functions lie outside of the language in some sort of meta-theory. For example, if the programs are terms of a traditional λ-calculus (see Chapter 9) then *quotation* is required to convert abstractions into trees.

Tree calculus will adopt a different approach, by avoiding the conversions from trees to executables altogether. Rather, the trees will be themselves executables, will be functions as well as data structures. Then the challenge is to represent function application as an operation on trees.

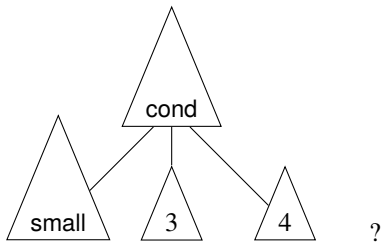
3.2 Natural Trees

The first issue then is to decide which tokens will be used to label the nodes of our trees. In the example of the previous section, the nodes were labelled by the keyword cond, the identifier small, and the numerals 3 and 4. In general, there will be many different syntactic constructions and so many different sorts of labels. Fortunately, we do not need to commit to some choice, since the labelled trees can be represented by unlabelled trees, here called *natural trees*, by analogy with the natural numbers. In pictures,



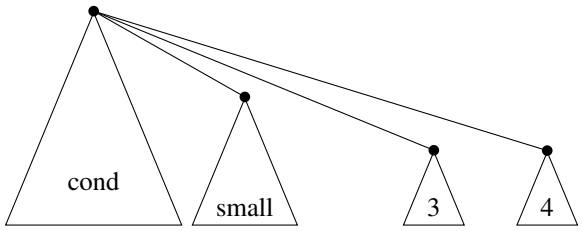
so that the label cond (represented by the rounded rectangle) is replaced by a tree (represented by the triangle) whose name is cond. In the same manner, we can replace the labels small and 3 and 4 with little trees, represented by named triangles.

That done, should the display of the original program fragment be



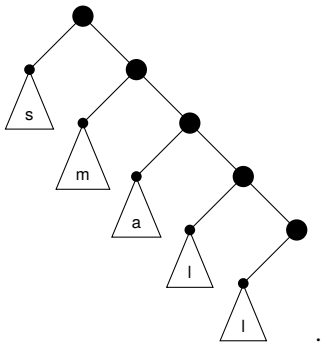
?

This would suggest that the subordinate trees have the status of leaves of cond when in fact they are branches, as was clear when cond was a mere node with a label. To avoid this suggestion, their status as branches will be emphasised by adding a dot to represent the root of each named tree, and drawing the edges between the roots of the trees. Thus, the program fragment becomes



so that we can see that the sub-trees have been grafted to the root of the main tree.

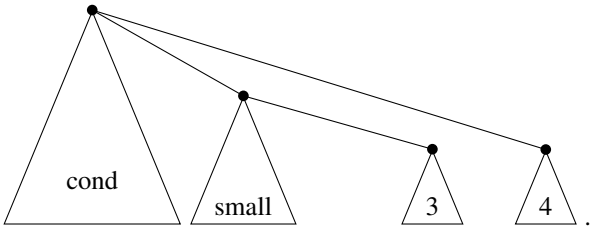
Since there are only finitely many keywords, it is easy to designate some particular, small trees to represent them. However, there is an unbounded collection of numbers and identifiers to be represented too. Happily, these can be represented using a finite alphabet. For example, the identifier small could be represented by



.

Note that the large dots in the picture are actual, free-standing nodes, while the small dots denote the roots of the various sub-trees.

All of the natural trees can be built from unlabelled nodes by grafting new branches in the style developed above. While the dot notation for nodes is good in graphics, it reads poorly in text, so we will there use a small triangle $\triangle$ to represent a node. Although its name is "node" you may think of this as the Greek letter delta (or Δ) on the understanding that, like a tree, the delta of the river Nile has branches. The grafting of a new branch N to an existing tree M is represented by the *application* MN. Application associates to the left, so that MNP is the same as $(MN)P$ but *not* the same as $M(NP)$. For example, the iterated application cond small 3 4 is pictured above, while cond (small 3) 4 is pictured by



In this manner, syntax trees can be represented by natural trees so the question becomes how to compute with them.

3.3 Tree Calculus

The natural trees are represented by the combinations of tree calculus, which are given by the BNF

$$M, N ::= \triangle \mid MN \ .$$

This asserts that every tree given by the BNF is either $\triangle$ or an application MN of one tree to another. Pictorially, each instance of $\triangle$ provides a node of the tree, while each application MN adds an edge from the root of M to the root of N.

One way of computing with natural trees is to treat them all as potential outputs, but then the functions which act on trees, i.e. the programs, and the machinery for evaluation must be elsewhere. This leads to a traditional separation of functions from data, which will not help to build reflective programs. Rather, let us consider how the trees themselves can be the programs, can be functions, so that the computation of program M on input N is given by the application MN of M to N. In this sense, the trees of tree calculus will be *computations* that require evaluation. The question is then to decide the nature of the results produced by evaluation and, indeed, what constitutes a program or it inputs.

Since tree calculus is to be reflective, the programs must also qualify as inputs. Since it important to be able to compose programs, the inputs should be the same as the outputs. So allow the programs, inputs and results form a single sub-class of trees that we may call the *values* or *programs*.

To discuss the choice of values it will be useful to introduce some terminology. A natural tree which is a node without any branches is a *leaf* or sometimes a *kernel*. A natural tree with one branch is a *stem*. A natural tree with two branches is a *fork*. Since the order of the branches is important, a fork has a *left branch* and a *right branch*.

If the only value is a leaf then no distinctions between values are possible. If the values are expanded to allow stems built from other stems then the values are *chains* of the form

$$\triangle(\triangle(\triangle \ldots (\triangle\triangle))) \ .$$

These correspond to the natural numbers. More conveniently, let us define

$$M^k(N) = M(M \ldots (MN) \ldots) \quad \text{where } M \text{ is applied } k \text{ times .}$$

Then we can identify the natural number k with $\triangle^k \triangle$.

This approach leads to a traditional account of computation built around natural numbers. The difficulty with this is that chains have no obvious internal structure, so that elaborate encoding and decoding is required to represent program structure. These operations are themselves computations which then require their own explanation, so this cannot be considered progress.

The next simplest option is for the values to be the *binary trees*: the kernels, stems and forks whose branches are also binary trees. As the forks are easily able to represent internal structure, this is the choice made for tree calculus. It follows that trees containing a node with three or more branches is a computation that is not a value, and so must be evaluated. Symbolically, this implies that any tree of the form $\triangle MNP$ requires evaluation. If M, N and P are values then $\triangle MN$ is a program which is applied to the input P. In other words, the node $\triangle$ is a ternary operator, i.e. one that requires three arguments to trigger evaluation whose goal is to produce a binary tree.

Now let us consider how we might formulate the computation rules. If the evaluation does not query the internal structure of its arguments x, y and z then we are led to produce a single rule of the form

$$\triangle xyz = t$$

for some tree t built from the trees x and y and z. However, this would limit the expressive power to that of combinatory logic, which cannot support reflection (see Chapter 8), so we must *query* the internal structure of the arguments. This opens up a world of possibilities. The first argument x could be a leaf, stem or fork, whose branches could also take one of three forms, etc. Similar remarks apply for y and z. Even if we limit the inspection to just the first layer of structure in each argument, there are $3 \times 3 \times 3 = 27$ possible left-hand sides for equations! Fortunately, there is no need to explore too far. Complex queries can be broken down into a sequence of simple queries, with each one considering just one level of structure of just one argument. Since English is written from left to right, it is customary to inspect arguments from left to right as well, so we will consider the internal structure of the first argument x above.

Thus, the left-hand sides of the three rules take the form

$$\begin{aligned} \triangle\triangle yz &= \ldots \\ \triangle(\triangle x)yz &= \ldots \\ \triangle(\triangle wx)yz &= \ldots \end{aligned}$$

The choice of right-hand sides for these rules are inspired by earlier calculi, specifically, the operators K and S of combinatory logic (Chapter 8) and the operator F of SF-calculus (Chapter 10). Happily, the operator K inspires the *kernel rule* (K), the operator S inspires the *stem rule* (S) and the operator F inspires the *fork rule* (F).

The rules are:

$$\triangle\triangle yz = y \qquad (K)$$
$$\triangle(\triangle x)yz = yz(xz) \qquad (S)$$
$$\triangle(\triangle wx)yz = zwx \qquad (F)$$

These rules can be expressed pictorially, using graphs, in Figure 3.1.

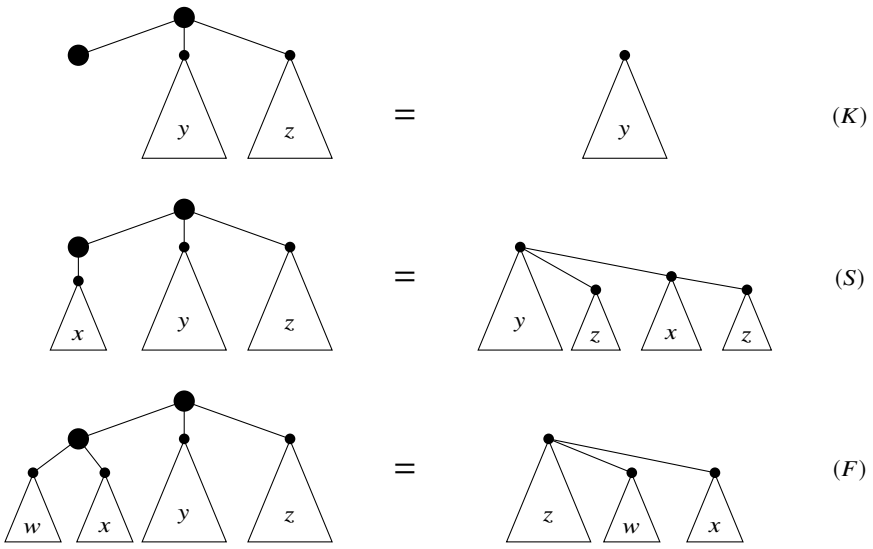


Fig. 3.1 Tree Equations

Note that (K) deletes the third argument, (S) duplicates the third argument, and (F) decomposes the first argument to expose its branches. These rules will prove to be enough to represent the calculi that inspired them. For example, we have the combinations

$$K = \triangle\triangle$$
$$I = \triangle(\triangle\triangle)(\triangle\triangle)$$
$$D = \triangle(\triangle\triangle)(\triangle\triangle\triangle)\ .$$

Their pictures are given in Figure 3.2. They satisfy the equations

$$\begin{aligned} Kyz &= \triangle\triangle yz = y \\ Ix &= \triangle(\triangle\triangle)(\triangle\triangle)x = \triangle\triangle x(\triangle x) = x \\ Dxyz &= \triangle(\triangle\triangle)(\triangle\triangle\triangle)xyz \\ &= \triangle\triangle\triangle x(\triangle x)yz \\ &= \triangle(\triangle x)yz \\ &= yz(xz)\ . \end{aligned}$$

Since Dx already simplifies to $\triangle(\triangle x)$ it is useful to define

$$d\{x\} = \triangle(\triangle x)$$

to make combinations easier to read.

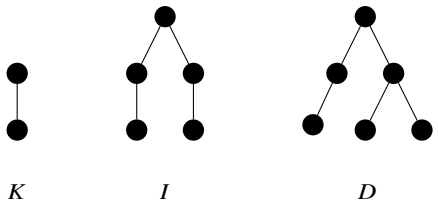


Fig. 3.2 The Combinations K, I and D

As well as duplicating its third argument, the stem rule (and D) permute the order in which the first two arguments occur, which is not the obvious choice. However, this approach has the advantage of being able to decide if a program is a stem or not, since $yz(xz)$ in the rule for stems allows us to use the behaviour of y to expose the fact that x appeared in a stem, while the behaviour of $xz(yz)$ is determined by that of the unknown x. This will be important for identifying the leaves, stems and forks.

Also, the permutation is easily programmed away. For example, there is a tree S such that

$$Sxyz = xz(yz)\ .$$

The equation can be solved by eliminating the variables one at a time, as follows.

$$\begin{aligned} Sxyz &= xz(yz) \\ &= d\{y\}xz \end{aligned}$$

so we must solve $Sxy = Dyx$ by

$$\begin{aligned} Sxy &= Dyx \\ &= Dy(Kxy) \\ &= D(Kx)Dy \end{aligned}$$

so it is enough to solve $Sx = D(Kx)D$. For the next step, we must represent $D(Kx)$ and D as functions of x by

$$\begin{aligned} Sx &= D(Kx)D \\ &= (KDx)(Kx)(KDx) \\ &= DK(KD)x(KDx) \\ &= D(KD)(DK(KD))x \end{aligned}$$

so we have S being $D(KD)(DK(KD))$ which evaluates to

$$S = d\{KD\}(d\{K\}(KD)) \; . \tag{3.1}$$

We can check the answer by

$$\begin{aligned} Sxyz &= d\{KD\}(d\{K\}(KD))xyz \\ &= d\{K\}(KD))x(KDx)yz \\ &= KDx(Kx)(KDx)yz \\ &= D(Kx)Dyz \\ &= Dy(Kxy)z \\ &= Dyxz \\ &= xz(yz) \; . \end{aligned}$$

In this manner, the operators S, K and I of combinatory logic (see Chapter 8) can be defined as trees.

Figure 3.3 shows two representations of the tree S. That on the left uses a mix of bare nodes and named branches. That on the right uses bare nodes only. Each has its advantages. Note that the precise angles of forks, and the relative weight given to nodes and edges may have a big impact on the look of a tree but they have no semantic significance.

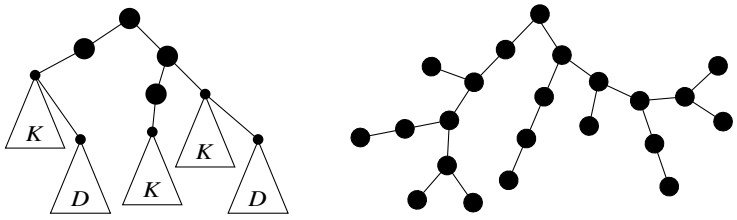


Fig. 3.3 Two Representations of S

3.4 Programs

We define the *programs* of tree calculus to be the binary trees. They can be given by the BNF

$$p, q ::= \triangle \mid \triangle p \mid \triangle p q \ .$$

More precisely, every program *is* a computation of tree calculus, so this BNF describes those computations which are programs.

It is not immediately obvious that this definition accords with the usual intuitions about programs. To the extent that programs are distinct from arbitrary terms of a calculus, they should be more easily analysed and optimised by, say, a compiler, while still maintaining the desired expressive power of the underlying calculus. Certainly, it is easy to work with binary trees, especially as they are stable; none of the evaluation rules can be applied from left to right. The important question is whether they can maintain expressive power. This will emerge throughout the book, as programs are defined for equality of arbitrary programs, for the operators of abstraction calculus, for pattern-matching, etc. These are all examples of *recursive programs*. Traditionally, recursive programs that act on other programs are *not* stable, so the identification of programs with binary trees may be surprising. Nevertheless, it will all work out. The definition of programs is given here as it will simplify the development. Its suitability will emerge as we go along.

3.5 Propositional Logic

Propositional logic determines the truth of compound propositions, built using conjunction, disjunction, implication, negation and equivalence, from the truth of their components. That is, it computes elementary functions of the boolean values true and false.

These boolean values are specified by the requirements

$$\mathsf{true}\ x\ y = x$$
$$\mathsf{false}\ x\ y = y \ .$$

Thus we can define true to be $K = \triangle\triangle$ above and false to be KI or $\triangle\triangle(\triangle(\triangle\triangle)(\triangle\triangle))$ since

$$KIxy = Iy = y$$

as required.

The basic boolean functions are defined by

$$\begin{aligned}
\mathsf{and} &= d\{K(KI)\} \\
\mathsf{or} &= d\{KK\}I \\
\mathsf{implies} &= d\{KK\} \\
\mathsf{not} &= d\{KK\}\ (d\{K(KI)\}I) \\
\mathsf{iff} &= \triangle(\triangle I\ \mathsf{not})\triangle\ .
\end{aligned}$$

Their graphs are given in Figure 3.4.

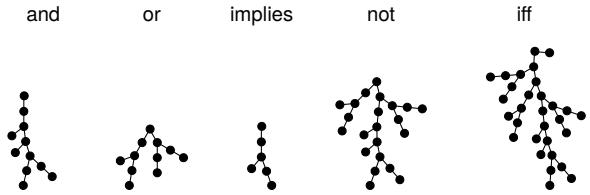


Fig. 3.4 Some Boolean Operations

Now

$$\mathsf{and}\ x\ y = d\{K(KI)\}xy = xy(K(KI)y) = xy(KI)\ .$$

There are four cases to check for conjunction, according to whether x and y are true or false. The first two are

$$\begin{aligned}
\mathsf{and\ true\ true} &= \mathsf{true\ true}\ (KI) \\
&= K\ \mathsf{true}\ (KI) \\
&= \mathsf{true}
\end{aligned}$$

and

$$\begin{aligned}
\mathsf{and\ true\ false} &= \mathsf{true\ false}\ (KI) \\
&= K\ \mathsf{false}\ (KI) \\
&= \mathsf{false}\ .
\end{aligned}$$

The other two cases are left as exercises.

Similarly, we have

$$\begin{aligned}
\mathsf{or}\ x\ y &= d\{KK\}Ixy \\
&= Ix(KKx)y \\
&= xKy\ .
\end{aligned}$$

If x is true then we get

$$\mathsf{or\ true}\ y = KKy = K = \mathsf{true}$$

and if x is false then we get

$$\mathsf{or\ false}\ y = KIKy = y$$

as required. The properties of implies, not and iff are left as exercises.

3.6 Pairs

The term $\triangle\ x\ y$ can be used to represent the pair of x and y so that we have

$$\begin{aligned}\mathsf{Pair} &= \triangle\\ \mathsf{first}\{p\} &= \triangle p \triangle K\\ \mathsf{second}\{p\} &= \triangle p \triangle (KI)\end{aligned}$$

which implies

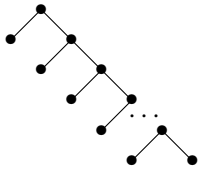
$$\begin{aligned}\mathsf{first}\{\mathsf{Pair}\ x\ y\} &= \triangle(\triangle xy)\triangle K = Kxy = x\\ \mathsf{second}\{\mathsf{Pair}\ x\ y\} &= \triangle(\triangle xy)\triangle(KI) = (KI)xy = y\ .\end{aligned}$$

3.7 Natural Numbers

In Section 3.3 the natural numbers were represented by chains of the form $\triangle^n\triangle$. Working with such chains makes constant reference to the rule (S) which is the most complex of the three rules. To avoid this, let us represent the number n by

$$K^n\triangle$$

or



so that positive numbers are always forks. Now zero is given by $\triangle$ and the *successor* of n is given by Kn so that we have $\mathsf{successor} = K$. Now the test for being zero is

$$\mathsf{isZero} = d\{K^4I\}\ (d\{KK\}\triangle)\ .$$

To see this, observe that

$$\begin{aligned}\mathsf{isZero}\ n &= d\{K^4I\}\ (d\{KK\}\triangle)\ n\\ &= d\{KK)\}kern((K^4I)n)\\ &= \triangle n(KKn)(K^3I)\\ &= \triangle nK(K^3I)\ .\end{aligned}$$

So we have

$$\begin{aligned}\mathsf{isZero}\ \triangle &= \triangle\triangle K(K^3I)\\ &= K = \mathsf{true}\end{aligned}$$

and

$$\begin{aligned}\mathsf{isZero}\ (Kn) &= \triangle(Kn)K(K^3I)\\ &= \triangle(\triangle\triangle n)K(K^3I)\\ &= K^3I\triangle n\\ &= KI = \mathsf{false}\end{aligned}$$

as required. Now the *predecessor* of a positive number $n_1 = \mathsf{successor}\ n$ is recovered by $n_1\triangle$ since

$$\mathsf{successor}\ n\ \triangle = Kn\triangle = n\ .$$

Unfortunately, this account doesn't work for the predecessor of zero, which should be zero, but $\triangle\triangle$ is not $\triangle$. The solution is to first test n by solving

$$\mathsf{predecessor}\ n = \triangle\ n\ \triangle\ (KI)$$

to get

$$\mathsf{predecessor} = d\{K^2I\}(d\{K\triangle\}\triangle). \tag{3.2}$$

since

$$\begin{aligned}\mathsf{predecessor}\ n &= d\{K^2I\}(d\{K\triangle\}\triangle)n\\ &= d\{K\triangle\}\triangle\ n\ (K^2In)\\ &= \triangle n(K\triangle n)(KI)\\ &= \triangle n\triangle(KI)\end{aligned}$$

as required. The graphs of the zero test and the predecessor are given in Figure 3.5.

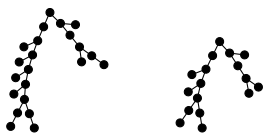


Fig. 3.5 The Zero Test and the Predecessor

Further arithmetic functions, such as addition and subtraction, will be introduced in Chapter 4 after more machinery has been produced.

3.8 Fundamental Queries

It is not quite trivial to distinguish the leaves, stems and forks, since a test $\triangle xMN$ of the structure of the argument x has only two terms M and N to separate the three possibilities. The fundamental queries provides the machinery to do this. Here are the details.

A test for being a leaf must map leaves to K and stems and forks to KI. There are two other possibilities for the other two tests. To solve all three problems together, let is0 be the result of testing a leaf, and is1 and is2 be the results of testing stems and forks, respectively. Now we must solve three equations:

$$\begin{aligned}\mathsf{query}\{\mathsf{is0},\mathsf{is1},\mathsf{is2}\}\triangle &= \mathsf{is0}\\ \mathsf{query}\{\mathsf{is0},\mathsf{is1},\mathsf{is2}\}(\triangle x) &= \mathsf{is1}\\ \mathsf{query}\{\mathsf{is0},\mathsf{is1},\mathsf{is2}\}(\triangle xy) &= \mathsf{is2}\ .\end{aligned}$$

Since the goal is to test x, we look for a solution of the form $\triangle xfg$ or $\triangle xfgh$ or $\triangle fghk$ or .... A first attempt might be something like $\triangle\ x$ is0 is1 is2 but I can't solve the problem using just three arguments, so consider solving by $\triangle xfghk$. The three possibilities for x yield

$$\begin{aligned}fhk &= \mathsf{is0}\\ fg\_hk &= \mathsf{is1}\\ g\_\ \_hk &= \mathsf{is2}\end{aligned}$$

where the wildcards _ represent arguments over which we have no control, or information. The naive solutions for f and g are K^2(is0) and K^4(is2). This leaves the middle equation, which simplifies to

$$\mathsf{is0}\ h\ k = \mathsf{is1}\ .$$

If looking for a leaf then we must solve $Khk = KI$ so that h is KI. The rest are solved by taking k to be is1. So we have

$$\mathsf{query}\{\mathsf{is0},\mathsf{is1},\mathsf{is2}\}\ x = \ \triangle x(K^2\mathsf{is0})(K^4\mathsf{is2})(KI)\ \mathsf{is1}$$

Finally, an equation of the form

$$qx = \triangle xfghk$$

can be solved by

$$\begin{aligned} qx &= \triangle x f g h k \\ &= D(Kf)\triangle x g h k \\ &= D(Kg)(D(Kf)\triangle)x h k \\ &= D(Kh)(D(Kg)(D(Kf)\triangle))x k \\ &= D(Kk)(D(Kh)(D(Kg)(D(Kf)\triangle)))x \end{aligned}$$

so, after simplifying, we have

$$\mathsf{query}\{\mathsf{is0}, \mathsf{is1}, \mathsf{is2}\} = d\{K\ \mathsf{is1}\}(d\{K^2 I\}(d\{K^5\ \mathsf{is2}\}(d\{K^3\ \mathsf{is0}\}\triangle)))$$

which is represented in Figure 3.6.

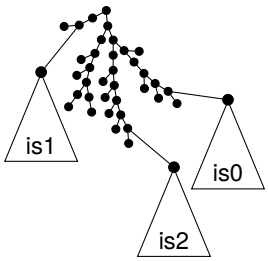


Fig. 3.6 Parametrising the Fundamental Queries

Hence, we can define the *fundamental queries*

$$\begin{aligned} \mathsf{isLeaf} &= \mathsf{query}\{K, KI, KI\} \\ \mathsf{isStem} &= \mathsf{query}\{KI, K, KI\} \\ \mathsf{isFork} &= \mathsf{query}\{KI, KI, K\} \end{aligned}$$

which are tests for being a leaf, a stem or a fork, respectively. They are represented in Figure 3.7.

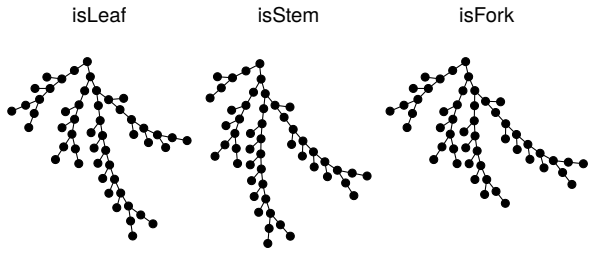
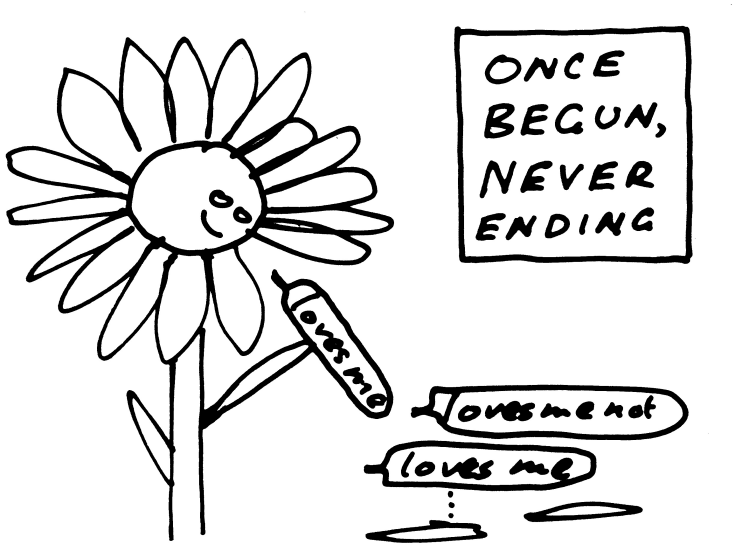


Fig. 3.7 The Fundamental Queries

Exercises

1. Define D in terms of K and S just as S can be defined in terms of K and D in (3.1).
2. Check the evaluation of and x y where x is false.
3. Check the properties of the boolean operators or, implies, not and iff.
4. Find trees fst and snd that represent the projections from pairs in general, instead requiring a specific pair. That is, solve fst p = first$\{p\}$ and snd p = second$\{p\}$. Draw their graphs.
5. Complete the proof that the definition of the predecessor in (3.2) meets its specification.
6. (Hard) Rebuild the arithmetic of this chapter where the number n is represented by $\triangle^n\triangle$ instead of $K^n\triangle$.
7. Write out the binary tree for isLeaf. Use it to evaluate isLeaf $\triangle$ and isLeaf K and isLeaf $(K\triangle)$ from first principals.
8. (Hard) The *depth* of a binary tree is the length of the longest chain within it that starts at the root. How many binary trees are there with depth at most three, or four, or five? How many binary trees are there with depth exactly three, or four, or five? Can you give a general formula for these?

Chapter 4
Extensional Programs



4.1 Combinators, Combinations and Terms

The *extensional* or *functional behaviour* of a program is the behaviour which is revealed by applying the program to various arguments. In particular, two programs which have the same input-output relation are extensionally equivalent. Haskell Curry characterised this by identifying the extensional functions with the combinators. A *combinator* M is a function of the form

$$Mx_1x_2\ldots x_n = t \tag{4.1}$$

where the x_i are *variables* and t is some *combination* of the x_i built by application. That is, M is a function which, when applied to some combinators $N_1, N_2, \ldots N_n$ produces the result of substituting N_i for x_i in t. More generally, we can denote the substitution of N for x in t by $\{N/x\}t$ so that

$$MN_1N_2\ldots N_n = \{N_1/x_1\}\{N_2/x_2\}\ldots\{N_n/x_n\}t$$

or, removing superfluous brackets, that

$$MN_1N_2 \ldots N_n = \{N_1/x_1, N_2/x_2, \ldots, N_n/x_n\}t \ .$$

For examples of combinators, we have

$$\begin{aligned} Ix &= x \\ Kxy &= x \\ Sxyz &= xz(yz) \\ Bfxy &= fyx \\ Cgfx &= g(fx) \\ Dxyz &= yz(xz) \ . \end{aligned} \tag{4.2}$$

Thus KMN is equal to the result of substituting M for x and N for y in x, i.e. M.

In turn, the combinators form a calculus that has no need of variables, as they can be applied to each other. For example, if M and N are combinators then

$$(KI)MN = (KIM)N = IN = N$$

shows that KI returns its second argument, just as K returns its first argument. In this manner, the combinators provide good support for computation. Actually, the first combinators were introduced by Moses Schönfinkel as a functional calculus ("funktion calcul") with the goal of eliminating variables and their binding from the logic of predicate calculus. From this point of view, Curry's definition of combinators is not quite satisfactory, as it references the variables that we are trying to eliminate. So let us start with the operators.

In developing a calculus of combinators, we have various choices for the primitive operators, as the collection S, K, I, B, C or S, K, I or S, K. Further, we have the primitive operator $\triangle$ of tree calculus. So let us develop a general account based on combinations of some given class O of operators. That is, the *O-combinations* are given by the Backus-Naur Form (or BNF, see Chapter 2.2)

$$M, N ::= O \mid MN$$

where MN is the *application* of M to N and application is left-associative. Usually, the operators come with some evaluation rules. Where the choice of rules is given, the collection of combinations and evaluation rules is called O-calculus. For example, we may call tree calculus $\triangle$-calculus, on the understanding that the evaluation rules are $(K), (S)$ and (F).

Although the goal of combinatory logic was to eliminate the variables, it is difficult, if not impossible to follow through, because variables M, N, P, Q are required to describe all of the evaluation rules considered so far. Instead, we must be content to eliminate variable binding, or at least to give an account of it in terms of more fundamental operations. Thus, to present the evaluation rules requires variables $x, y, z, \ldots$. We will not here concern ourselves with the exact nature of the variables. They could be given by strings of characters, by numbers or something else; it is enough that they be plentiful and that we are able to decide if two variables are the same or not. Then the *O-terms* are described by the BNF

$$t, u ::= v \mid O \mid tu$$

where v here stands for an arbitrary variable. The *closed terms* are the subset of the terms corresponding to the O-combinations.

For example, the combinations of variables t used to define the combinators in Equation (4.1) are O-terms where O is the empty collection of operators.

The *substitution* $\{u/x\}t$ of a term u for a variable x in a term t is defined by

$$\begin{aligned} \{u/x\}x &= u \\ \{u/x\}y &= y \quad (y \neq x) \\ \{u/x\}O &= O \\ \{u/x\}(s\ t) &= \{u/x\}s\ \{u/x\}t\ . \end{aligned}$$

4.2 Variable Binding

In tree calculus, every combinator M as in Equation (4.1) can be represented by a combination denoted by

$$M = [x_1][x_2] \ldots [x_n]t$$

where the *bracket abstraction* $[x]t$ represents the function that, when applied to x yields t. It is defined as follows:

$$\begin{aligned} [x]x &= I \\ [x]y &= Ky \quad (y \neq x) \\ [x]O &= KO \\ [x]uv &= d\{[x]v\}([x]u)\ . \end{aligned}$$

Theorem 4.1 (bracket_beta)

For all variables x and terms t and u we have

$$([x]t)u = \{u/x\}t\ . \qquad (\beta)$$

Hence the combinator M in Equation (4.1) is represented by the combination $[x_1][x_2] \ldots [x_n]t$.

Proof The proof of the β-rule is by induction on the structure of t. There are three cases, according to whether t is a variable, operator or application. The proof of the second statement is by induction on the number n of variables. Details are left as exercises, or can be found in the Coq proofs. □

Since all combinators can now be represented in tree calculus, we say that tree calculus is *combinatorially complete*. Note that this does not imply that $\triangle$ can be represented as a combinator, as will be proved in Chapter 8.

A curious property of bracket abstractions is that they are *always* stable, in the sense that none of the evaluation rules can be applied from left to right. For example, although Kxx reduces to x, we have

$$[x]Kxx = d\{[x]x\}([x]Kx) = d\{I\}(\triangle(\triangle I)(KK))$$

which does *not* reduce to $[x]x = I$. That is, bracket abstraction does *not* preserve the equality induced by the evaluation rules. In other words, it does not preserve the meaning of terms.

In this sense, bracket abstraction is not a "first-class" means of constructing terms, as these should preserve the equality based on the evaluation rules. This has long been seen as a weakness of combinatory logic. Could we add evaluation rules so that, for example, $[x]Kxx$ reduces to $[x]x$? Could we derive

$$d\{I\}(d\{I\}(KK)) = I\ ?$$

Despite various attempts, no set of additional rules has been found that will support arbitrary evaluation under bracket abstraction. This abstraction problem will be solved in *VA*-calculus in Chapter 9. For now, this "bug" is better regarded as a feature since the stability of bracket abstraction (and star abstraction below) provides fine control over reduction, as we will see in Section 4.5.

Unfortunately, the abstraction $[x]t$ is usually larger than necessary. For example, if M is a combinator then KM satisfies the β-rule just as well as $[x]M$ since KMu reduces to M in one step. However, $[x]M$ is much bigger than M. If M is a combinator of size k (e.g built from k copies of $\triangle$) then $[x]M$ has size $5k - 2$. For example, if M is I then we have

$$[x]I = [x](\triangle(\triangle\triangle)(\triangle\triangle))$$

which has size twenty-three instead of KI whose size is seven. Similarly, $[x](Mx) = d\{I\}([x]M)$ is bigger than M even though M also satisfies the β-rule. This five-fold expansion is especially problematic when abstractions are nested.

To solve this, it is necessary to keep track of variable occurrences in terms. Recall that we were trying to avoid variables, but this is easier said than done! Define $x \in t$ by induction on the structure of t as follows:

$$\begin{aligned} x \in y &= \text{if } x \text{ and } y \text{ are the same variable} \\ x \in \triangle &= \mathsf{false} \\ x \in tu &= x \in t \text{ or } x \in u\ . \end{aligned}$$

The solution is to optimise the bracket abstraction to exploit variable occurrence, by defining *star abstraction* as follows

$$\begin{array}{rll} \lambda^* x.t = Kt & & (x \notin t) \\ \lambda^* x.tx = t & & (x \notin t) \\ \lambda^* x.x = I & & \\ \lambda^* x.tu = d\{\lambda^* x.u\}(\lambda^* x.t) & & \text{(otherwise).} \end{array}$$

The reference to λ, the Greek letter lambda, comes from λ-calculus (see Chapter 9). If you are already familiar with λ-calculus then this notation will be suggestive but if not then it is safe to treat star abstraction as an independent concept.

Theorem 4.2 (star_beta)

For all variables x and terms t and u we have

$$(\lambda^* x.t)u = \{u/x\}t \ . \qquad (\beta)$$

Proof If t is of the form s or sx where $x \notin s$ then the result is immediate. Otherwise, the proof is as for bracket abstraction. □

Note that star abstraction still breaks some evaluations (since $\lambda^* x.Kxx = d\{I\}K$ which is not I) . Unlike bracket abstractions, star abstractions may require evaluation, since any combinator M is also $\lambda^* x.Mx$.

4.3 Fixpoints

One of the fundamental concepts in computation is *iteration*, in which a function f is applied a given number of times. For example, to iterate f three times over an argument x is to produce $f(f(fx))$. More generally, given a number n then we have

$$f^n(x) = f(f(\ldots(fx)\ldots)) \quad \text{(where } f \text{ is applied } n \text{ times.)}$$

Iteration is good enough when the number of iterations is known in advance, but there are many algorithms for searching or approximation where there is no prior bound on how long the search will take. These functions require the more powerful use of *recursion* which can be represented in *fixpoint functions*. This can be achieved by a combinator Y such that, for any f, we have

$$Yf = f(Yf) \ .$$

For example, if f is KI then

$$Y(KI)x = KI(Yf)x = x$$

so that $Y(KI)$ is an identity function. On the other hand, if f is I then we get

$$YIx = I(YI)x = YIx$$

which does *not* have a value. Some searches never find a solution: some approximations never converge: they go on forever. Here is a concrete example of a combinator whose evaluation never ends. For any x we have

$$(\lambda^* x.xx)(\lambda^* x.xx) = \{\lambda^* x.xx/x\}(xx) = (\lambda^* x.xx)(\lambda^* x.xx) \ .$$

Expressed using operators, this abstraction becomes

$$DII = \triangle(\triangle(\triangle(\triangle\triangle)(\triangle\triangle)))(\triangle(\triangle\triangle)(\triangle\triangle)) \ .$$

and we have

$$(DII)(DII) = I(DII)(I(DII)) = (DII)(DII) \ .$$

Since any evaluation of it can be further evaluated to recover the original itself, $(DII)(DII)$ can never be stabilised.

A variant of this can be used to define Y. Let

$$\omega = \lambda^* z.\lambda^* f.f(zzf)$$

and define $Y = \omega\omega$. Then we have

$$\begin{aligned} Yf &= \omega\omega f \\ &= (\lambda^* z.\lambda^* f.f(zzf))\omega f \\ &= \{\omega/z\}(\lambda^* f.f(zzf))f \\ &= (\lambda^* f.\{\omega/z\}(f(zzf)))f \\ &= (\lambda^* f.f(\omega\omega f))f \\ &= \{f/f\}(f(\omega\omega f)) \\ &= f(\omega\omega f) \\ &= f(Yf) \end{aligned}$$

as required. Note how the substitution of ω for z has been passed under the star abstraction with respect to f. On this occasion this is safe to do, but perhaps you would like to check by eliminating the star abstraction.

4.4 Waiting

The account of fixpoints above, where Yf evaluates to $f(Yf)$, means that there are always evaluation sequences that do not terminate, such as

$$Yf = f(Yf) = f(f(Yf)) = f(f(f(Yf))) = \ldots$$

where the next step is always to evaluate Yf to $f(Yf)$. Using this approach, our recursions are never stable, which makes reflective programming difficult, if not impossible. However, there is a better way.

Since the goal is to produce recursive *functions*, there is no prior requirement to evaluate Yf; it is enough to evaluate Yfx for any function argument x. All that is required is a mechanism to delay the application of Y to f until an argument x is given. The application of Y to f should *wait*.

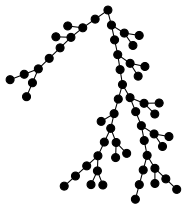


Fig. 4.1 The Program W

Consider the combinations $W_0 = \lambda^* x.\lambda^* y.\lambda^* z.xyz$ and

$$W = \lambda^* x.\lambda^* y.[z]xyz$$

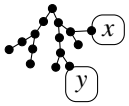
(see Figure 4.1). Both have the property that application to combinations M, N and P yields MNP but they differ when applied to M and N only. Since W_0 is actually I we have $W_0MN = IMN = MN$ in which MN can be reduced but

$$WMN = \{M/x, N/y\}[z]xyz = d\{I\}(d\{KN\}(KM))$$

in which M and N cannot yet interact. WMN must *wait* for another argument P before applying M to N. Note that, in general, WMN is *not* the same as $[z]MNz$ since the latter is always a program, even if M is not. Since it will be used often, let us define

$$\mathsf{wait}\{x, y\} = d\{I\}(d\{Ky\}(Kx)) \ .$$

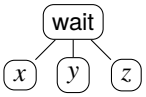
It is pictured by



More abstractly, it can be represented by using three labelled triangles for wait and x and y. Even more convenient is to use rounded rectangles instead of triangles, as in

This is a much cleaner approach, which will be iterated to construct much larger programs. However, it does require implicit knowledge that wait takes two parameters. Bt contrast, the diagram

represents Kxy since K takes no parameters. Indeed, these two interpretations can be combined in the diagram



which denotes the application $\mathsf{wait}\{x, y\}z$.

Here are some related forms of waiting. The value of Wx given by $\lambda^* y.\mathsf{wait}\{x, y\}$ on the assumption that the variable y does not occur in x. Calculating the star abstraction allows this assumption to be eliminated to produce

$$\mathsf{wait1}\{x\} = d(d\{K(K@x)\}(d(d\{K\}(K\triangle))(K\triangle))(K(d\{\triangle KK\})).$$

Waiting for two more arguments is given by analysis of $\lambda^* z.[w].xyzw$ where w, x, y and z are distinct variables. This yields

$$\mathsf{wait2}\{x, y\} = d\{d\{K(d\{Ky\}(Kx))\}\ d\{d\{K\}(K\triangle)\}(K\triangle)\}(K(d\{I\})).$$

Then

$$\mathsf{wait21}\{x\} = \lambda^* y.\mathsf{wait2}\{x, y\} \quad (\ y \text{ not in } x)$$

can be similarly expanded to avoid the side-condition.

Again, we can define $\mathsf{wait3}\{x, y\}$ by analysis of $\lambda^* z.\lambda^* t.[w]xyztw$ and then $\mathsf{wait31}\{x\}$ by abstracting with respect to y.

4.5 Fixpoint Functions

Now let us make fixpoint functions that wait for their argument. One solution begins by replacing $\omega\omega$ in Y by $\mathsf{wait}\{\omega, \omega\}$ which is now a normal form. However, there are still two difficulties: when applied to some program f it becomes $\omega\omega f$ which does not have a normal form; it evaluates to $f(\omega\omega f)$ which is *not* $\mathsf{wait}\{\omega, \omega\}f$. Continuing on this path will work, but there is a lot of waiting, and a lot of duplication. Instead, let us try a more direct approach.

To reduce duplication of combinations in the program, introduce

$$\mathsf{self\_apply} = \lambda^* w.ww = d\{I\}I$$

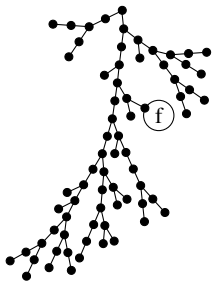


Fig. 4.2 The Program $Y_2\{f\}$

and replace $\omega\omega$ with self_apply ω. Then define

$$Y_2\{f\} = \mathsf{wait}\{\mathsf{self\_apply}, d\{\mathsf{wait1}\{\mathsf{self\_apply}\}(Kf)\}\ .$$

Theorem 4.3 (fixpoint_function) *For any function f the program $Y_2\{f\}$ is a fixpoint function for f.*

Proof

$$\begin{aligned} Y_2\{f\}x &= \mathsf{self\_apply}\ d\{\mathsf{wait1}\{\mathsf{self\_apply}\}(Kf)\ x \\ &= d\{\mathsf{wait1}\{\mathsf{self\_apply}\}(Kf)\ d\{\mathsf{wait1}\{\mathsf{self\_apply}\}(Kf)\ x \\ &= Kf(d\{\mathsf{wait1}\{\mathsf{self\_apply}\}(Kf)) \\ &\qquad (\mathsf{wait1}\{\mathsf{self\_apply}\}\ (d\{\mathsf{wait1}\{\mathsf{self\_apply}\}(Kf))))\ x \\ &= f\ Y_2\{f\}\ x\ . \end{aligned}$$

□

Two subtleties lie behind these calculations. First, even when $Y_2\{f\}$ is a program, this does *not* imply that $Y_2\{f\}x$ evaluates to a program. For example, $Y_2\{I\}$ is a program but $Y_2\{I\}x$ is not as it reduces to itself.

Second, it is tempting to replace $Y_2\{f\}$ by the application to f of the abstraction $\lambda^* f.Y_2\{f\}$. However, $(\lambda^* f.Y_2\{f\})f$ does not take the form of a program until simplified to $Y_2\{f\}$. Further, the proof of its properties becomes more complicated, since left-to-right application of the rules implies

$$(\lambda^* f.Y_2\{f\})fx = f(Y_2\{f\})x$$

but does not imply $(\lambda^* f.Y_2\{f\})fx = f((\lambda^* f.Y_2\{f\})f)fx$. Rather, the fixpoint property requires a second left-to-right application to show $f((\lambda^* f.Y_2\{f\})f)fx = f(Y_2\{f\})x$. This will become more troublesome in Part II when the rules will acquire an orientation. So we will stick with $Y_2\{f\}$.

Since recursive functions usually want to act on their argument directly, it is useful to have another fixpoint construction specified by

$$Y_{2s}\{f\}\ x = f\ x\ Y_{2s}\{f\}\ .$$

It is defined as

$$Y_{2s}\{f\} = Y_2\{\mathsf{swap}\{f\}\}$$

where $\mathsf{swap}\{f\} = d\{Kf\}(d\{d\{K\}(K\triangle)\}(K\triangle))$. This will be used in Chapter 6.

The same techniques can be used to define fixpoints for functions for any number of arguments. Here are fixpoints for functions of two and three arguments that will prove useful later:

$$Y_3\{f\} = \mathsf{wait2}\{\mathsf{self\_apply}, d\{\mathsf{wait21}\{\mathsf{self\_apply}\}(Kf)\}$$
$$Y_4\{f\} = \mathsf{wait3}\{\mathsf{self\_apply}, d\{\mathsf{wait31}\{\mathsf{self\_apply}\}(Kf)\}\ .$$

as expected.

4.6 Arithmetic

As a simple application of fixpoints, we have addition of natural numbers given by

$$\mathsf{plus} = Y_3\{\lambda^* p.\lambda^* m.\triangle mI(K(\lambda^* x.\lambda^* n.(K(pxn))))\}$$

which is presented in Figure 4.3.

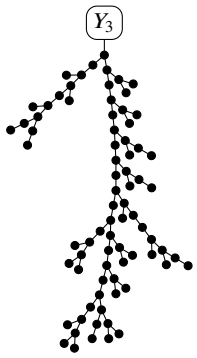


Fig. 4.3 The Program plus

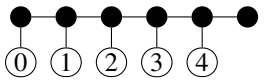
Similarly, subtraction can be defined by building on the predecessor instead of the successor. In turn, these can be used to define multiplication and division, exponentiation, etc.

The other main arithmetic task is to solve equations. In general, one can envisage all sorts of techniques for this but the brute force approach is to check each number in turn, starting with zero, to see if it is a solution. This process of *minimisation* can also be described using fixpoints, by fixing the function which maps n to n if it is a solution, and to the successor of n otherwise.

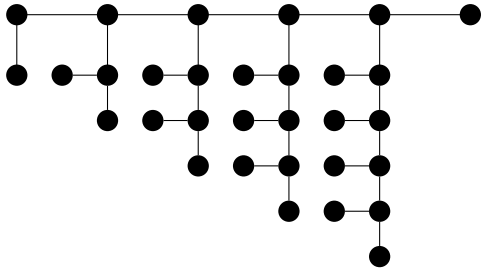
In this manner, Kleene specified the class of μ-recursive arithmetic functions, which arise in all the traditional models of computation. Indeed, they are all definable in tree calculus too. Since the details do not contribute much to the understanding of reflective programs, they are not given here in the text, but can be found in the formal development in Coq.

4.7 Lists and Strings

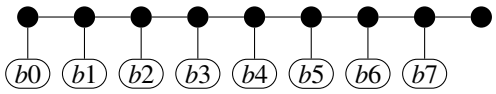
Lists can be represented by using a leaf to represent the empty or nil list, and a fork to combine the head and tail of a list. For example, the list with entries $0, 1, 2, 3, 4$ is given by



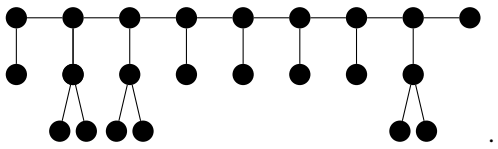
which expands to



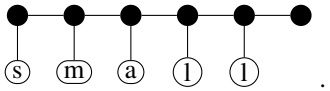
The *bits* 0 and 1 are represented by the natural numbers $\triangle$ and $K\triangle$ as before. The *bytes* are given by eight-tuples of bits $(b0, b1, b2, b3, b4, b5, b6, b7)$ which can be represented by



Hence, the ASCII characters can be represented by their bytecodes. For example, the character 'a' is given by the byte 01100001 which is given by



Then *strings* are given by lists of characters. For example, the string "small" is given by



.

To represent "small" as a winter tree without any labels, let us rotate the tree for each character clockwise by a quarter circle, to get the picture in Figure 4.4

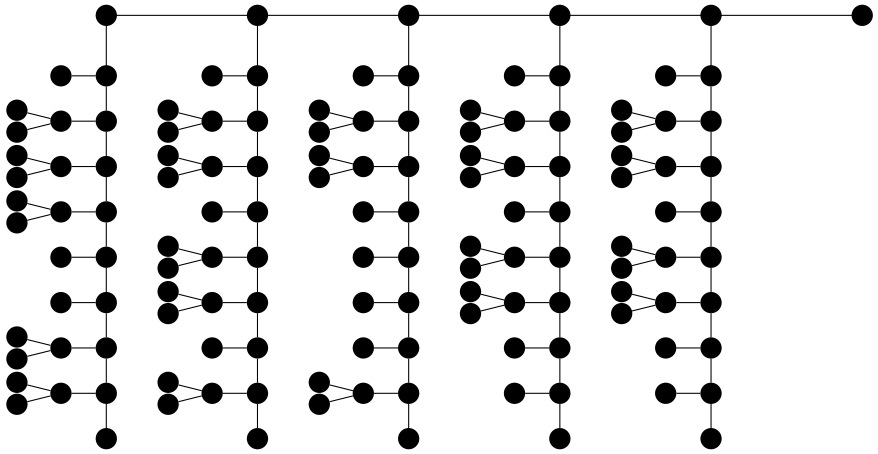


Fig. 4.4 The String "small"

4.8 Mapping and Folding

To *map* a function f over a list is to apply the function to each entry of the list. It is defined by

$$\mathsf{list\_map} = Y_3(\lambda^* m.\lambda^* f.\lambda^* x.\triangle x\ \mathsf{t\_nil}\ (\lambda^* h.\lambda^* t.\mathsf{t\_cons}\ (f\,h)\ (m\,f\,t)))\ .$$

Its graph is in Figure 4.5. It follows that we have

$$\begin{aligned}\mathsf{list\_map}\ f\ \mathsf{t\_nil} &= \mathsf{t\_nil}\\ \mathsf{list\_map}\ f\ (\mathsf{t\_cons}\ h\ t) &= \mathsf{t\_cons}\ (f\,h)\ (\mathsf{list\_map}\ f\ t)\end{aligned}$$

as required.

Similarly, *left folding* of a function over a list is specified by

$$\begin{aligned}\mathsf{list\_foldleft}\ f\ x\ \mathsf{t\_nil} &= x\\ \mathsf{list\_foldleft}\ f\ x\ (\mathsf{t\_cons}\ h\ t) &= \mathsf{list\_foldleft}\ f\ (f\,x\,h)\ t\end{aligned}$$

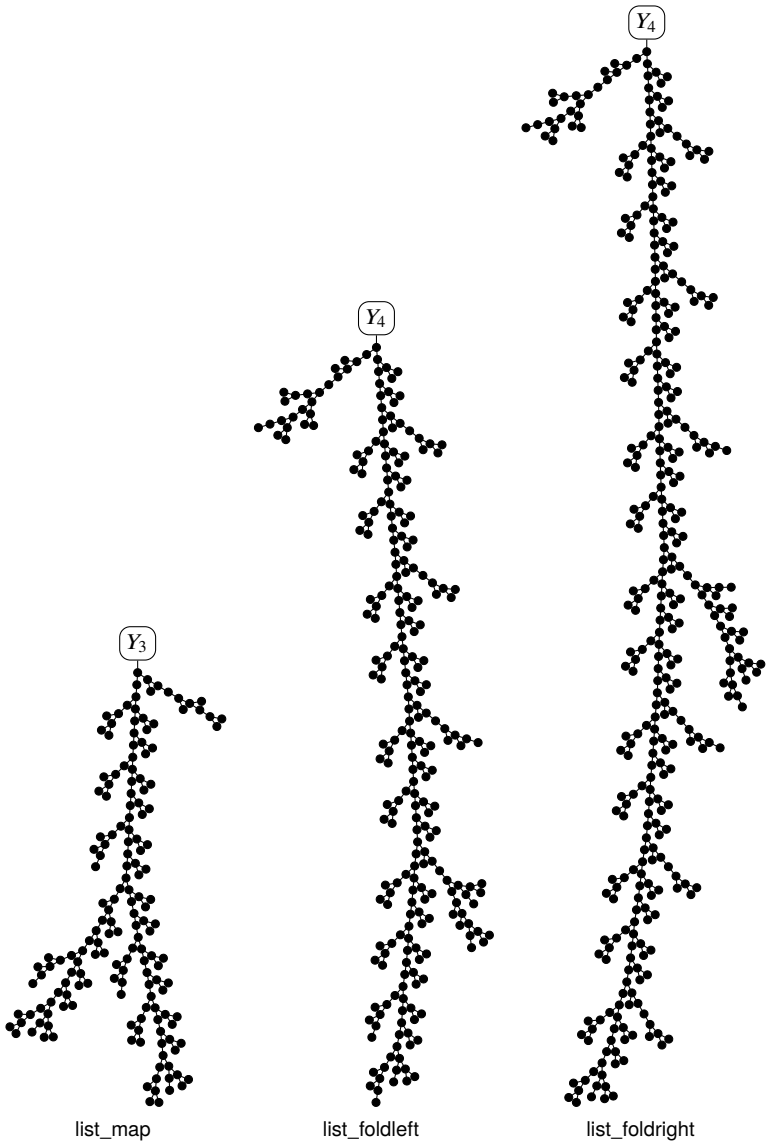


Fig. 4.5 Mapping and Folding over Lists

and realised in Figure 4.5 by

$$\mathsf{list\_foldleft} = Y_4(\lambda^* g.\lambda^* f.\lambda^* x.\lambda^* y.\triangle yx(D(fx)(K(gf)))) \ .$$

Again, *right folding* of a function over a list is specified by

$$\begin{aligned} \mathsf{list\_foldright}\ f\ x\ \mathsf{t\_nil} &= x \\ \mathsf{list\_foldright}\ f\ x\ (\mathsf{t\_cons}\ h\ t) &= f\ h\ (\mathsf{list\_foldright}\ f\ x\ t) \end{aligned}$$

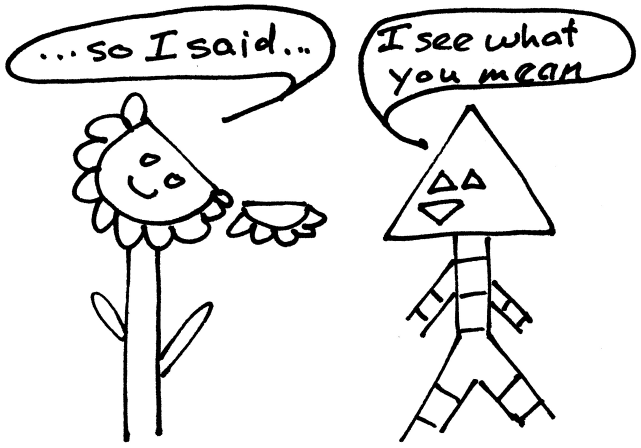
and realised in Figure 4.5 by

$$\mathsf{list\_foldright} = Y_4(\lambda^* g.\lambda^* f.\lambda^* x.\lambda^* y.\triangle yx(\lambda^* h.\lambda^* t.fh(gfxt))) \ ,$$

Exercises

1. Show that if M is a combination of size k then $[x]M$ has size $5k - 2$.
2. Show that $\{\omega/z\}(\lambda^* f.f(zzf)) = \lambda^* f.f(\omega\omega f)$.
3. What is the tree for W?
4. Define $\omega' = [x][f]f(xxf)$. Show that $\omega' xf$ has the same functional behaviour as ω.
5. Show that W_3 and Y_3 have the desired properties.
6. Show that W_4 and Y_4 have the desired properties.
7. Define multiplication of numerals and confirm its properties.
8. Define subtraction of numerals.

Chapter 5
Intensional Programs



5.1 Intensionality

Intensional programs query the internal structure of their arguments. Simple examples include programs for the size or equality of binary trees. More general is the ability to tag functions with additional information, such as comments or types.

The *intensional behaviour* of a program p is all behaviour which is revealed by making it the argument of some query, without trying to apply p to arguments. For example, any comments within a program reflect the intentions of the programmer. By design, they cannot impact on the program's functional behaviour but they may be revealed by program analysis. Similarly, the two programs which map a number x to $0 + x$ or to $x + 0$ have the same functional behaviour as the identity function, but may be distinguished if the programming language is sufficiently intensional.

Usually, calculi that allow functions to be applied to functions do not support intensional programming. For example, we will see in Chapter 8 that combinatory logic does not support comment extraction. By contrast, tree calculus supports intensional programming, in the sense that its programs can be tagged with comments which can be recovered during computation. These tags can be used to distinguish

terms which have different meanings even though they have the same behaviour as functions. In particular, tags may include *type information* which can be used to check for type errors.

This chapter will begin with a couple of simple examples of intensional computation, of the size of a program and of the equality of programs. Then we will introduce tagging, and use it to define a simply-typed language of booleans and natural numbers.

5.2 Size

The size of a program p is computed easily enough, given the machinery already developed. If it is a leaf then its size is 1. If it is a stem $\triangle x$ then its size is one more than that of x. If it is a fork $\triangle xy$ then its size is one more than the sum of the sizes of x and y. Since the functions size is a recursive function of one argument, it is given by an application of Y_2. The simplest way to separate the leaves, stems and forks is to first test the argument x for being a stem. If so, then take the successor of the size of its branch. Otherwise, x must be a kernel or fork, whose cases can be separated by an application of $\triangle\ x$ to suitable arguments. Hence, we have

$$\begin{aligned}\mathsf{size} = Y_2(\lambda^* s.\lambda^* x.&\mathsf{isStem}\ x\\ &(\triangle(x\ \triangle)\triangle(\lambda^* x_1.K(K(s\ x_1)))\\ &(\triangle\ x\ (K\triangle)\ (\lambda^* x_1.(\lambda^* x_2.K(\mathsf{plus}\ (s\ x_1)\ (s\ x_2)))))))\ .\end{aligned}$$

Although this works well enough, the use of plus introduces a second recursion, which is expensive. The representation of size is given in Figure 5.1.

5.3 Equality

Equality of programs is defined in the same manner as their size, except that now Y_3 is required to handle two arguments.

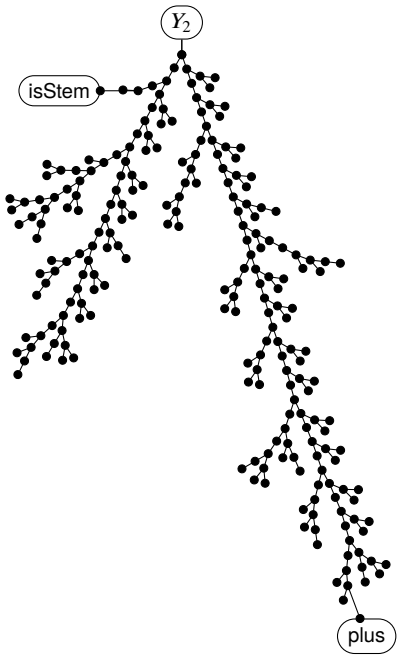


Fig. 5.1 The Size of Programs as a Function of plus

$$
\begin{array}{l}
\mathsf{equal} = Y_3(\lambda^* e.\lambda^* x.\lambda^* y. \\
\quad \mathsf{isStem}\ x \\
\quad\quad (\mathsf{isStem}\ y \\
\quad\quad\quad (e(x\triangle)\ (y\triangle)) \\
\quad\quad\quad (KI)) \\
\quad\quad (\triangle x \\
\quad\quad\quad (\mathsf{isLeaf}\ y) \\
\quad\quad\quad (\lambda^* x_1.\lambda^* x_2. \\
\quad\quad\quad\quad \mathsf{isFork}\ y \\
\quad\quad\quad\quad\quad (\triangle y\triangle(\lambda^* y_1.\lambda^* y_2.ex_1y_1(ex_2y_2)(KI))) \\
\quad\quad\quad\quad\quad (KI)))) \ .
\end{array}
$$

Theorem 5.1 (equal_programs)
For all programs M, equal $M\ M = K$

Proof The proof is by induction on the structure of M. □

Theorem 5.2 (unequal_programs)

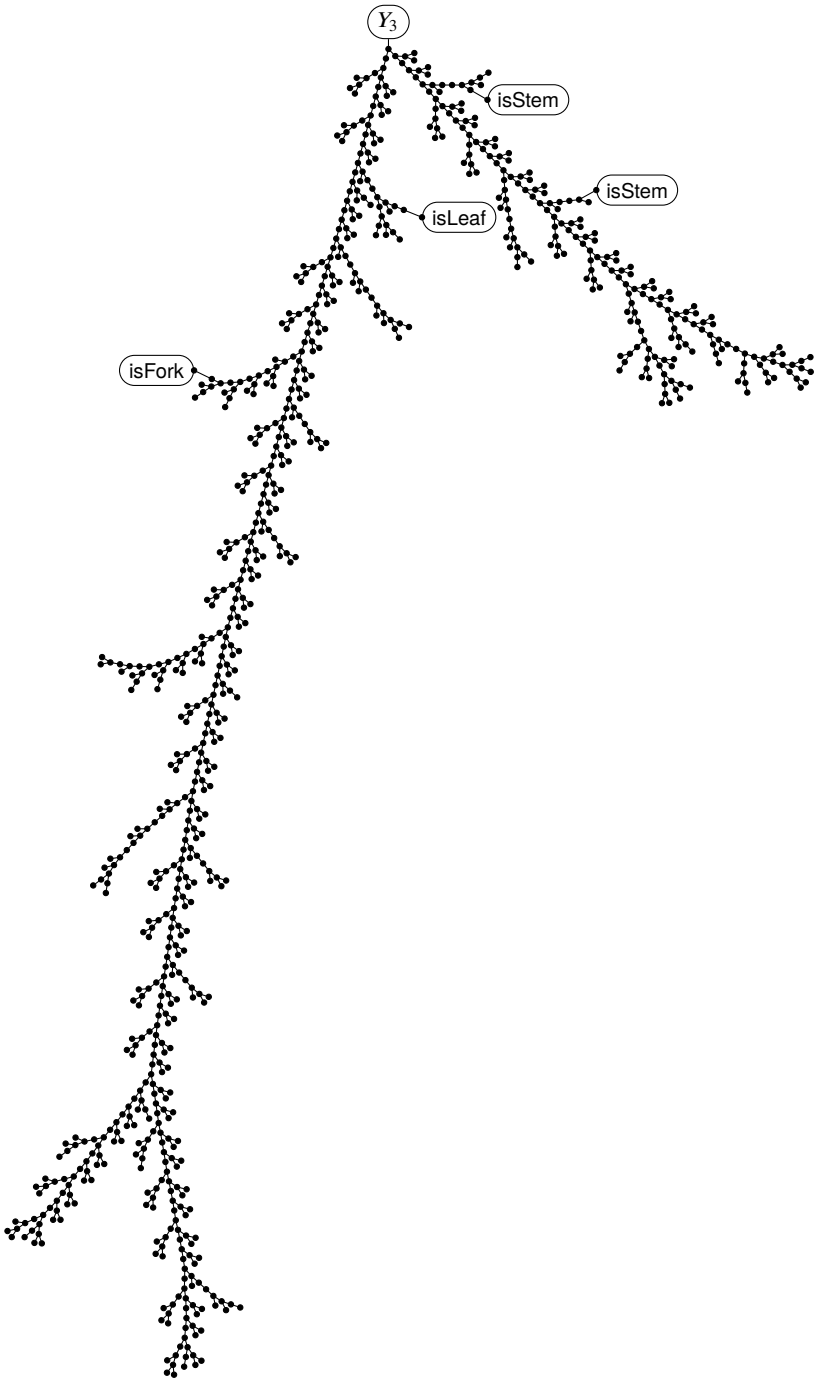


Fig. 5.2 The Program equal

For all programs M and N, if $M \neq N$ then $\mathsf{equal}\ M\ N = KI$

Proof The proof is by induction on the structure of M. □

5.4 Tagging

A calculus supports *tagging* if there are term forms tag and getTag such that

$$\begin{aligned} \mathsf{tag}\{t, f\}\ x &= f\ x \\ \mathsf{getTag}\ (\mathsf{tag}\{t, f\}) &= t\ . \end{aligned}$$

for all terms t and f. Thus, even though the tag t can be recovered by applying getTag, the tag has no effect on the functional behaviour, as the tag is discarded when the function is applied to some argument x. Note that getTag is defined to be a combination but tag is a binary function of combinations. The reasons for this are practical. In Part II we will define some programs using tag and getTag. That getTag is a combination simplifies the analysis, but if tag were a defined to be a combination then its applications would not be programs.

Theorem 5.3 (tree_calculus_supports_tagging)
Tree calculus supports tagging.

Proof Define

$$\begin{aligned} \mathsf{tag}\{t, f\} &= d\{t\}(d\{f\}(KK)) \\ \mathsf{getTag} &= \lambda^* p.\mathsf{first}\{\mathsf{first}\{p\}\triangle\}\ . \end{aligned}$$

Then

$$\begin{aligned} \mathsf{tag}\{t, f\}x &= d\{f\}(KK)x(tx) \\ &= KKx(fx)(tx) \\ &= K(fx)(tx) \\ &= fx\ . \end{aligned}$$

and

$$\begin{aligned} \mathsf{getTag}\ (\mathsf{tag}\{t, f\}) &= \mathsf{first}\{\mathsf{first}\{d\{t\}(d\{f\}(KK))\}\triangle\} \\ &= \mathsf{first}\{\triangle t\triangle\} = t \end{aligned}$$

as required. □

Further, we can recover the term f that has been tagged by

$$\mathsf{untag} = \lambda^* x.\mathsf{fst}\{\mathsf{fst}\{\mathsf{snd}\{x\}\}\ \triangle\}\ .$$

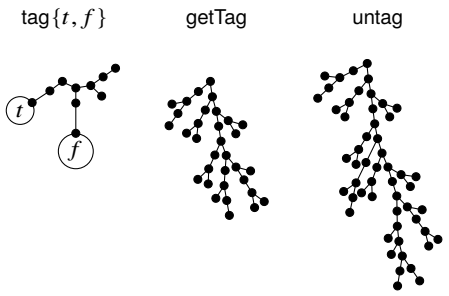


Fig. 5.3 Tagging

Trees for these constructions are in Figure 5.3

Tagging can be used to record the *intentions* of the programmer as executable comments. For example, it can be used to name things, so that the declaration let small = True can be replaced by tag{small, True}. We have seen how to define strings of characters in Chapter 4.7 so it is now easy to add comments to programs.

It will sometime be convenient to tag fixpoint functions. Such tags cannot be added after the fixpoint has been formed, as they must recur. Instead, define

$$Y_{2t}\{t, f\} = \mathsf{tag}\{\mathsf{wait}\{\mathsf{self\_apply}, d\{\mathsf{tag\_wait}\{t\}(Kf)\}\}\}$$

where tag_wait{t} is a tagged version of wait1{self_apply} that appeared in the definition of Y_2 that is given by

$$\mathsf{tag\_wait}\{t\} = \lambda^* w.\mathsf{tag}\{t, \mathsf{wait}\{\mathsf{self\_apply}, w\}\} \quad (w \text{ not in } t).$$

As usual, the side-condition can be eliminated by expanding out the star-abstraction.

Below is another application of tagging.

5.5 Simple Types

Types are used to separate programs into meaningful classes, so that, for example, functions and numbers are kept apart. There is a rich collection of type systems and of typed programming languages. Some languages require every term to have a type, which improves clarity and safety but excludes some interesting programs. In others, types are desirable, but optional. This increases expressive power at the cost of some safety. This section will introduce a simple type system, in the style of William Tait, that supports booleans, natural numbers and functions, and use it to type a few terms of tree calculus by tagging terms with their types. Much remains to be done, even for simple types. With little effort, the type system can be extended to support other type constructions, e.g. for list types. With a greater effort, it should be possible

to support more advanced type systems. However, these commonly require the full power of variable binding, which not be addressed until Part II. Furthermore, type systems deserve a comprehensive treatment that goes far beyond the scope of this book.

The *simple types* are given by the BNF

$$T ::= o \mid \iota \mid T \to T \ .$$

These are represented by the trees

$$\begin{aligned} [o] &= \triangle \text{ “Bool”} \\ [\iota] &= \triangle \text{ “Nat”} \\ [U \to T] &= \triangle\, [U]\, [T] \ . \end{aligned}$$

That is, the boolean and number types are represented by stems and function types are represented by forks.

A *typed term* t^T is a term t decorated with a type T. It is represented by

$$t^T = \mathsf{tag}\{T, t\} \ .$$

In this manner we can provide types for the booleans and some boolean operations, the natural numbers and some arithmetic operations, as follows.

$$\begin{aligned} \mathsf{true} &= K^o \\ \mathsf{false} &= (KI)^o \\ \mathsf{and} &= (\triangle^2(K^2I))^{o\to o\to o} \\ \mathsf{or} &= (\triangle(\triangle(KK))I)^{o\to o\to o} \\ \mathsf{implies} &= (\triangle^2(KK))^{o\to o\to o} \\ \mathsf{not} &= (\triangle(\triangle(KK))(\triangle(\triangle(K(KI)))I))^{o\to o} \\ \mathsf{iff} &= (\triangle(\triangle\mathsf{not}))^{o\to o\to o} \\ \mathsf{zero} &= \triangle^{\iota} \\ \mathsf{successor} &= K^{\iota\to\iota} \\ \mathsf{isZero} &= (\triangle(\triangle(K^4I))(\triangle(\triangle(KK))\triangle))^{\iota\to o} \\ \mathsf{plus} &= (Y_3\{\lambda^* p.\lambda^* x.\lambda^* y.\triangle yx(K(p(Kx)))\})^{\iota\to\iota\to\iota} \ . \end{aligned}$$

That done, we must show how to apply one typed term to another. The usual application will not do because the tags get in the way. Rather, we must tag the application of the untyped terms with its inferred type. Of course, if type inference fails then an error results. A type error is represented by $\mathsf{error} = \triangle$.

Type checking of applications is defined by

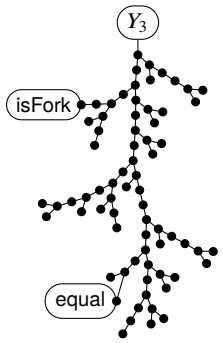


Fig. 5.4 The Program type_check

$$\mathsf{type\_check} = \lambda^* x.\lambda^* y.\mathsf{isFork}\ x\ (\triangle x \triangle(\lambda^* x_1.\lambda^* x_2.\mathsf{equal}\ x_1\ y\ x_2\ \mathsf{error}))\ \mathsf{error}\ .$$

Then *typed application* is defined by

$$\mathsf{typed\_app} = \lambda^* f.\lambda^* x.\mathsf{tag}\{\mathsf{type\_check}\ (\mathsf{getTag}\ f)\ (\mathsf{getTag}\ x), \mathsf{untag}\{f\}\ \mathsf{untag}\{x\}\}$$

The usual theorem to be proved of typed calculi is that evaluation preserves typing. This seems altogether plausible for our example, but we don't yet have the machinery required for the proof.

5.6 More Queries

The queries isLeaf and isStem and isFork of Chapter 3 produce the boolean values K or KI according to whether the argument is a leaf, stem or fork. However, this is not always the most efficient way of obtaining the information. An alternative is to use a leaf to represent falsehood and any fork to represent truth. Given such a boolean b then $\triangle bst$ is a form of conditional that applies t to the branches of a fork b and returns s if b is a leaf. Here are some examples.

$$\mathsf{isStem2} = \lambda^* z.\triangle z \triangle(K^2\triangle)\ .$$

It maps leaves and forks to a leaf (false) and a stem to a fork (true) namely $\triangle(K^2\triangle)(x(K^2\triangle))$. Of course, this fork is not very useful, since it does not support a uniform means of recovering x.

Similarly, define

$$\mathsf{isFork2} = \lambda^* z.\triangle z(KK)(K(K\triangle))\ .$$

It maps forks to $\triangle$ and maps leaves and stems to forks.

5.7 Triage

The programs size and equal both test their arguments, to test if they are leaves, stems or forks. This sort of testing can be packaged up using *triage*:

$$\mathsf{triage\_comb} = \lambda^* f_0.\lambda^* f_1.\lambda^* f_2.\lambda^* z.\mathsf{isStem2}\ z\ (\triangle z f_0 f_2)\ (\triangle(z\triangle)\triangle(\lambda^* x.K(f_1 x))\ .$$

If z is a stem $\triangle x$ then $z\triangle$ is a fork which can be used to apply f_1 to x. Otherwise, if z is a leaf or fork then $\triangle z f_0 f_2$ applies f_0 or f_2 respectively. It follows that we have

$$\begin{aligned}\mathsf{triage}\{f_0, f_1, f_2\}\ \triangle &= f_0\\ \mathsf{triage}\{f_0, f_1, f_2\}\ (\triangle x) &= f_1 x\\ \mathsf{triage}\{f_0, f_1, f_2\}\ (\triangle x y) &= f_2 x y\ .\end{aligned}$$

In this manner, we may re-express size as

$$\mathsf{size\_variant} = Y_2\{\lambda^* s.\mathsf{triage}\{K\triangle, \lambda^* y.K(sy), \lambda^* y.\lambda^* z.K(\mathsf{plus}\ (s\ y)\ (s\ z))\}$$

and re-express equal as

$$\begin{aligned}\mathsf{equal\_variant} = Y_3\{\lambda^* e.\mathsf{triage}\{&\\ &\mathsf{triage}\{K, KI, KI\},\\ &\lambda^* y.\mathsf{triage}\{KI, ey, K(K(KI))\},\\ &\lambda^* y_1.\lambda^* y_2.\mathsf{triage}\{KI, K(KI), \lambda^* z_1.\lambda^* z_2.ey_1 z_1(ey_2 z_2)(KI)\}\\ &\}\ .\end{aligned}$$

Note that equal_variant is not the same as equal since triage induces a slightly different flow of control during execution.

More sophisticated examples require the recursive use of triage, which can be expressed as pattern-matching.

5.8 Pattern Matching

Pattern-matching functions take the form

$$\begin{aligned}&p_1 \Rightarrow s_1\\ |\ &p_2 \Rightarrow s_2\\ &\ldots\\ |\ &p_k \Rightarrow s_k\\ |\ &r\end{aligned}$$

where each $p_i \Rightarrow s_i$ is a *case*. An argument u is compared to each pattern in turn until a match is found or all patterns have been attempted. If u matches with p_i then the resulting substitution is applied to s_i. If no match is found then ru results.

Such pattern-matching functions can be expressed as iterated *extensions* of the form

$$p \Rightarrow s \mid r$$

where p is the *pattern*, s is the *body* and r is the *default function*, since r may itself be an extension that captures all the other cases. In turn, the extension could be interpreted as an application of the case $p \Rightarrow s$ to r. However, this would imply that extensions are never programs, so we must use waiting, and define

$$p \Rightarrow s \mid r = \mathsf{wait}\{p \Rightarrow s, r\} \ .$$

The patterns for programs, or *program patterns* are given by the BNF

$$p, q ::= x \mid \triangle \mid \triangle p \mid \triangle pq \ .$$

That a variable occurs twice in a pattern is allowed but is not good style since only one occurrence will bind the body. A typical example of a pattern is $Y_2\{x\}$. Now define cases by induction on the structure of the pattern as shown in Figure 5.5.

$$\begin{aligned}
x \Rightarrow s &= K(\lambda^* x.s) \\
\triangle \Rightarrow s &= \{s/z\}(\lambda^* r.\lambda^* x.\mathsf{isLeaf}\ x\ z\ (r\ x)) \\
\triangle p \Rightarrow s &= \{p \Rightarrow s/z\}(\lambda^* r.\lambda^* x.\mathsf{isStem}\ x\ (\triangle(x\triangle)\triangle(\lambda^* y.K(f(\lambda^* z.r(\triangle z))y)))(r\ x)) \\
r_1 &= \lambda^* x_1.\lambda^* x_2.\lambda^* r.r(\triangle x_1 x_2) \\
r_2\{p\} &= \{p/z\}(\lambda^* x_2.(\lambda^* r.r(\triangle z x))) \\
\triangle pq \Rightarrow s &= \{p \Rightarrow \mathsf{wait}\{q \Rightarrow Ks, r_2\{p\}\}/z\} \\
&\qquad (\lambda^* r.(\lambda^* x.\mathsf{isFork}\ x\ (\triangle x\triangle(\lambda^* x_1.\lambda^* x_2.\mathsf{wait}\{z, r_1\}x_1 x_2 r))))
\end{aligned}$$

Fig. 5.5 Pattern Matching

Note that when the pattern is a variable then K is required to discard the default function. Also, when the pattern is a stem or fork then the default function r is used several times. Since this is commonly larger than the case itself, it is important to avoid its duplication. In turn, this requires auxiliary functions r_1 and r_2 to express the various uses of r when the pattern is a fork. This approach is able to show that, for example, we have $(Y_2\{x\} \Rightarrow x)\ I\ (Y_2\{K\}) = K$.

5.9 Eager Function Application

Sometimes, it is useful to be able to force the evaluation of an argument u before applying a function f to it. The precise definition of the *eager application* depends on the nature of the desired evaluation of u. Define a *factorable form* to be a leaf, stem or fork. Then eager evaluation to factorable form is given by

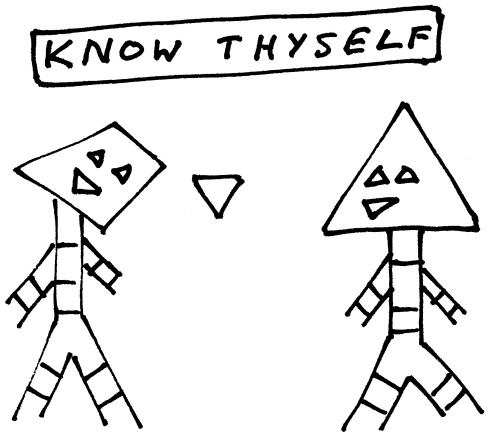
$$\mathsf{eager} = \lambda^* z.\lambda^* f.\triangle z I(K^2 I) I f z \ .$$

This works because $\triangle z I(K^2 I) I$ reduces to I whenever z is factorable.

Exercises

1. Compute size $(K^n \triangle)$.
2. Compute size W.
3. Compute size size.
4. Compute equal K $(K\triangle)$ by applying Theorem 5.2.
5. Compute equal equal equal by applying Theorem 5.1.
6. Confirm that tag and getTag meet their specifications.
7. Check the type of the program and true false.
8. Evaluate size_variant K.
9. Evaluate equal_variant K $(K\triangle)$.
10. Evaluate applications of eager to a leaf, a stem and a fork.
11. Produce a variant of eager that forces evaluation of the argument to be a program, and not just a leaf, stem or fork.
12. Redefine size as a pattern-matching function and use it to compute the size of K.
13. Define terms tag0 and getTag0 such that tag0 t f = tag$\{t, f\}$ and getTag0 p = getTag$\{p\}$.

Chapter 6
Reflective Programs



6.1 Reflection

Reflective programs are programs that can act on themselves to query their own structure. The querying is important: that the identity function can be applied to itself does not make it reflective. A simple example is the size function defined in Chapter 5. When applied to itself, the result is (the tree of) the number 1429. The canonical examples of reflective programs are self-evaluators, or self-interpreters, which express an evaluation strategy as a program which can then be self-applied. Self-evaluation in a calculus provides good evidence for the ability to perform program analysis and optimisation within the calculus itself. Traditionally, self-interpreters were allowed to act on the syntax tree of a program, i.e. its *quotation*. This is necessary if programs are inherently unstable, but quotation performs all of the work necessary to discover program structure, does all of the intensional computation. When quotation lies outside of the formal calculus then interpretation is separated from computation proper, so that some sort of staging is required.

6.2 Evaluation Strategies

So far, all of our calculations have been driven by sensible use of the equations of tree calculus. However, a computing machine needs a strategy to make the rules useful. For example, consider the evaluation of an application tu. Clearly, we need some information from t so suppose that it evaluates to some $\triangle ay$. In turn, we must evaluate a so suppose that it is of the form $\triangle x$. Should we evaluate x or y or u next, or should we evaluate $yu(xu)$? Traditionally, this question has been framed in terms of eager evaluation versus lazy evaluation, etc. but the alignment with concepts of tree calculus is not exact so new terminology will be introduced. We will consider four strategies that are definable within tree calculus.

Branch-first evaluation assumes that t and u above are programs and produces the program resulting from their application (if it has one). Each branch that is created is evaluated before evaluating the root. This is conceptually simple, since it is enough to manipulate programs, but will sometimes evaluate branches that do not contribute to the result. In the worst case, the evaluation of the unneeded branch fails to produce a result, so that evaluation fails to find the value.

Root evaluation does just enough work to determine whether y above is a leaf, stem or fork. More generally, it produces factorable forms whose branches are (unevaluated) computations. To control the evaluation of these branches, they will be *quoted*, so that computations are represented by values. Thus, root evaluation maps a quotation to a factorable form of quotations.

Root-and-branch evaluation first performs root evaluation and then recursively evaluates the branches of the resulting factorable form. Thus, root-and-branch evaluation maps a quotation to a program.

Of course, the use of quotation introduces meta-theory to the system, since quotation of computations is not definable within tree calculus. However, quotation of programs *is* definable by the program quote.

Root-first evaluation maps a pair of programs t and u to the program obtained by root-and-branch evaluation of $\triangle(\mathsf{quote}\ t)\ (\mathsf{quote}\ u)$. No meta-theory is required.

6.3 Branch-First Evaluation

In branch-first evaluation, all branches must be evaluated before trying to evaluate at the root. The values are the programs, here denoted by $v, v_1, v_2, \ldots$. The *evaluation*

$$v_1, v_2 \Rightarrow v$$

takes a pair of programs, the function v_1 and its argument v_2, and seeks a program v that is the result of the application, by applying the rules in Figure 6.1. Each rule has a conclusion below the line and premises above. Unlike tree calculus, the root of the tree is now at the bottom.

$$\frac{}{\triangle, z \Rightarrow \triangle z} \qquad \frac{}{\triangle y, z \Rightarrow \triangle yz} \qquad \frac{}{\triangle\triangle y, z \Rightarrow y}$$

$$\frac{y, z \Rightarrow v_1 \quad x, z \Rightarrow v_2 \quad v_1, v_2 \Rightarrow v}{\triangle(\triangle x)y, z \Rightarrow v} \qquad \frac{z, w \Rightarrow v_1 \quad v_1, x \Rightarrow v}{\triangle(\triangle wx)y, z \Rightarrow v}$$

Fig. 6.1 Branch-First Evaluation

For example, evaluation of $I\triangle$ proceeds as follows. First, place the pair to be evaluated below the line

$$\frac{?}{I, \triangle \Rightarrow ?}$$

where the question marks indicate missing parts of the derivation. Since I is $\triangle(\triangle\triangle)(\triangle\triangle)$ we can apply the stem rule to get

$$\frac{\triangle\triangle, \triangle \Rightarrow ? \quad \triangle, \triangle \Rightarrow ? \quad ?, ? \Rightarrow ?}{I, \triangle \Rightarrow ?}.$$

The first two premises follow from the first two rules, to get

$$\frac{\dfrac{}{\triangle\triangle, \triangle \Rightarrow \triangle\triangle\triangle} \quad \dfrac{}{\triangle, \triangle \Rightarrow \triangle\triangle} \quad \dfrac{?}{\triangle\triangle\triangle, \triangle\triangle \Rightarrow ?}}{I, \triangle \Rightarrow ?}.$$

Now apply the third to finish off with

$$\frac{\dfrac{}{\triangle\triangle, \triangle \Rightarrow \triangle\triangle\triangle} \quad \dfrac{}{\triangle, \triangle \Rightarrow \triangle\triangle} \quad \dfrac{}{\triangle\triangle\triangle, \triangle\triangle \Rightarrow \triangle}}{I, \triangle \Rightarrow \triangle}.$$

Note that for each function there is a unique rule that can be applied, so that the evaluation strategy is *deterministic*.

Theorem 6.1 (branch_first_eval_program)

If M and N are programs and there is a branch-first evaluation $M, N \Rightarrow P$ then P is a program.

Proof The proof is by induction on the structure of the evaluation. □

6.4 A Branch-First Self-Evaluator

A *self-evaluator* or, if you like, a *self-interpreter* for an evaluation strategy is a program that can represent the strategy. For branch-first evaluation, this would be a

program bf such that, for all programs M, N and P we have

$$M, N \Rightarrow P \quad \text{if and only if} \quad \mathsf{bf}\ M\ N = P\ .$$

In this chapter, we will show the more important half of this result: that if $M, N \Rightarrow P$ then $\mathsf{bf}\ M\ N = P$. That is, we can represent branch-first evaluation by a program. The proof of the converse, that bf does no more than the strategy allows, will require the rewriting theory of Chapter 7. Similar remarks apply to all the evaluators in this chapter.

The specification of bf is

$$Y_{2s}\{\mathsf{onFork}\{\mathsf{triage}\{\mathsf{bffl}, \mathsf{bffs}\{\mathsf{eager}\}, \mathsf{bfff}\}\}\}$$

as pictured in Figure 6.2. The term $\mathsf{onFork}\{f\}$ will be defined so that leaves and stems are left unchanged, while f is applied to the branches of a fork. Then triage on the left branch will determine which of the three evaluation rules to apply. The strategic choice is captured by the use of eager. Here are the details.

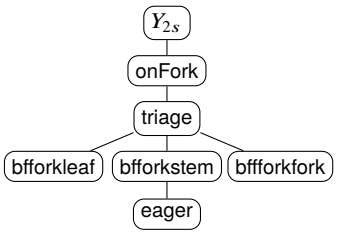


Fig. 6.2 A Branch-First Evaluator

Define

$$\mathsf{onFork}\{f\} = \lambda^* x.\triangle(\mathsf{isFork2}\ x)(\triangle x \triangle f)(K^3 x) \quad (x \text{ not in } f)\ .$$

If follows that $\mathsf{onFork}\{f\}\triangle\mathsf{bf} = K\triangle\mathsf{bf} = \triangle$ as required. As usual, the side condition can be eliminated by unfolding the definition of star-abstraction. One solution for the three cases is given by

$$\begin{aligned} \mathsf{bffl} &= \lambda^* y.\lambda^* b.\lambda^* z.y \\ \mathsf{bffs}\{e\} &= \lambda^* x.\lambda^* y.\lambda^* b.\lambda^* z.e(bxz)(b(byz)) \\ \mathsf{bfff} &= \lambda^* w.\lambda^* x.\lambda^* y.\lambda^* b.\lambda^* z.b(bzw)x \end{aligned}$$

Although this solution meets the specification, there are smaller solutions. And since bf appears in the frontispiece of the book, it is worth the effort to make it smaller. So define

$$\begin{aligned}
\mathsf{bfforkleaf} &= \lambda^* y.K(Ky)\\
D_b\{x\} &= D(D(Kx)I)(KD)\\
\mathsf{bfforkstem}\{e\} &= \lambda^* x.\lambda^* y.D(d\{K(Ke)\}(D_b\{x\}))(D(d\{K\}(D_b\{y\}))(KD))\\
\mathsf{bfforkfork} &= \lambda^* w.\lambda^* x.K(D(d\{K\}(D(D(Kw))(KD)))(K(D(Kx))))\ .
\end{aligned}$$

For example, if z is a program then

$$\begin{aligned}
\mathsf{bf}\ I\ z &= \mathsf{onFork}\{\mathsf{triage}\{\mathsf{bfforkleaf},\mathsf{bfforkstem}\{\mathsf{eager}\},\mathsf{bfforkfork}\}\}\ I\ \mathsf{bf}\ z\\
&= \mathsf{triage}\{\mathsf{bfforkleaf},\mathsf{bfforkstem}\{\mathsf{eager}\},\mathsf{bfforkfork}\}\ K\ K\ \mathsf{bf}\ z\\
&= \mathsf{bfforkstem}\{\mathsf{eager}\}\ \triangle\ K\ \mathsf{bf}\ z\\
&= \mathsf{eager}\ (\mathsf{bf}\ \triangle\ z)\ (\mathsf{bf}\ (\mathsf{bf}\ K\ z))\\
&= \mathsf{eager}\ (\triangle z)\ (\mathsf{bf}\ (\mathsf{bf}\ K\ z))\\
&= \mathsf{bf}\ (\mathsf{bf}\ K\ z)\ (\triangle z)\\
&= \mathsf{bf}\ (Kz)\ (\triangle z)\\
&= \mathsf{onFork}\{\mathsf{triage}\{\mathsf{bfforkleaf},\mathsf{bfforkstem}\{\mathsf{eager}\},\mathsf{bfforkfork}\}\}\ (Kz)\ \mathsf{bf}\ z\\
&= \mathsf{triage}\{\mathsf{bfforkleaf},\mathsf{bfforkstem}\{\mathsf{eager}\},\mathsf{bfforkfork}\}\ \triangle\ z\ \mathsf{bf}\ z\\
&= \mathsf{bfforkleaf}\ z\ \mathsf{bf}\ z\\
&= z\ .
\end{aligned}$$

Theorem 6.2 (branch_first_eval_to_bf)

If $M, N \Rightarrow P$ then $\mathsf{bf}\ M\ N = P$ *for all programs M and N and combinations P.*

Proof The proof is by induction on the structure of the derivation of $M, N \Rightarrow P$. There are five cases.

If M is a leaf then $P = \triangle N$ and $\mathsf{bf}\ \triangle N = \triangle N$.

If M is a stem $\triangle y$ then $P = \triangle yN$ and $\mathsf{bf}\ (\triangle y)\ N = \triangle y\ N$.

If M is a fork of a leaf $\triangle\triangle y$ then $P = y$ and $\mathsf{bf}\ (\triangle\triangle y)\ N = y$.

If M is a fork of a stem $\triangle(\triangle x)y$ then there are values v_1, v_2 and v_3 such that $y, N \Rightarrow v_1$ and $x, N \Rightarrow v_2$ and $v_1, v_2 \Rightarrow P$. Hence

$$\begin{aligned}
\mathsf{bf}\ (\triangle(\triangle x)y)\ N &= \mathsf{eager}\ (\mathsf{bf}\ x\ N)\ (\mathsf{bf}\ (\mathsf{bf}\ y\ N))\\
&= \mathsf{eager}\ v_2\ (\mathsf{bf}\ v_1)\\
&= \mathsf{bf}\ v_1\ v_2\\
&= P
\end{aligned}$$

by three applications of induction.

If M is a fork of a fork $\triangle(\triangle wx)y$ then there is a value v_1 such that $N, w \Rightarrow v_1$ and $v_1, x \Rightarrow P$. Hence

$$\begin{aligned}
\mathsf{bf}\ (\triangle(\triangle wx)y)\ z &= \mathsf{bf}\ (\mathsf{bf}\ N\ w)\ x\\
&= \mathsf{bf}\ v_1\ x\\
&= P
\end{aligned}$$

by two applications of induction. □

Hence, the self-evaluator can be applied to itself without requiring any additional machinery: if bf, $M \Rightarrow P$ then

$$\mathsf{bf}\ \mathsf{bf}\ M = P\ .$$

Although branch-first evaluation is quite efficient it does not find all results, since it may fail on an argument that is not required. Here is an example:

$$\begin{aligned}\triangle(\triangle\mathsf{self\_apply})(K(K\triangle))\ \mathsf{self\_apply} &= K(K\triangle)\ \mathsf{self\_apply}\ (\mathsf{self\_apply}\ \mathsf{self\_apply})\\ &= K\triangle\ (\mathsf{self\_apply}\ \mathsf{self\_apply})\ .\end{aligned}$$

The natural conclusion is to simplify the whole combination to $\triangle$. However, branch-first evaluation requires that the argument self_apply self_apply be evaluated first, but this term has no value! In more detail, bf $\triangle(\triangle\mathsf{self\_apply})(K(K\triangle))$ self_apply evaluates to

$$\mathsf{eager}\ (\mathsf{bf}\ \mathsf{self\_apply}\ \mathsf{self\_apply})\ (\mathsf{bf}\ (\mathsf{bf}\ (K\triangle)))\ .$$

Indeed, the role of eager in bf is precisely to ensure that all arguments are evaluated, as required by the branch-first evaluation strategy. It follows that this example requires a different strategy.

6.5 Root Evaluation

Root evaluation does just enough computation to determine the nature of the root of the tree, as a leaf, stem or fork, so that branches of a computation are evaluated only if needed. That is, the result of root evaluation must be able to represent unevaluated expressions even though these are not programs, are not values. This is achieved by *quotation*, which represents arbitrary computations by binary trees that have no stems.

The *quotation* of a combination is defined by

$$\begin{aligned}{'\triangle} &= \triangle\\ {'(MN)} &= \triangle('M)('N)\ .\end{aligned}$$

In tree calculus, quotation of arbitrary computations is not definable as a program since quotation does not preserve evaluation; that is, after all, the point. However, quotation of programs is definable, using

$$\begin{aligned}\mathsf{quote} = Y_2\{\lambda^* q.\lambda^* x.\mathsf{isStem}\ x\ (\triangle(x\triangle)\triangle(\lambda^* x_1.K(K(qx_1))))\\ (\triangle x\triangle(\lambda^* x_1.\lambda^* x_2.\triangle(K(qx_1))(qx_2)))\}\ .\end{aligned}$$

The graph of quote is given in Figure 6.3.

Root evaluation is a relation $M \Rightarrow P$ between combinations whose rules are given in Figure 6.4. If M is a quotation then P is a factorable form whose branches

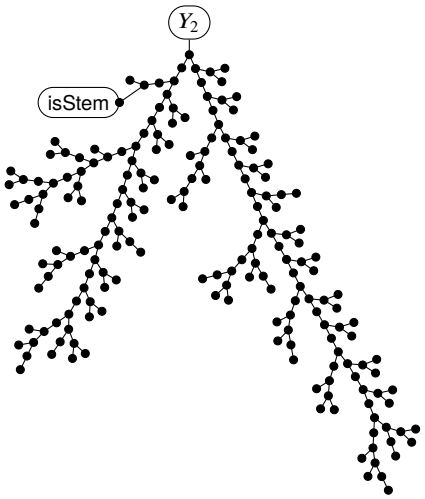


Fig. 6.3 Quotation

are quotations. However, these conditions are not enforced by the rules since, for example, $\triangle\triangle(\triangle\triangle) \Rightarrow \triangle\triangle$ even though $\triangle\triangle$ is not a quotation.

$$\frac{}{\triangle \Rightarrow \triangle} \qquad \frac{f \Rightarrow \triangle}{\triangle f z \Rightarrow \triangle z} \qquad \frac{f \Rightarrow \triangle y}{\triangle f z \Rightarrow \triangle y z}$$

$$\frac{f \Rightarrow \triangle t y \quad t \Rightarrow \triangle \quad y \Rightarrow v}{\triangle f z \Rightarrow v}$$

$$\frac{f \Rightarrow \triangle t y \quad t \Rightarrow \triangle x \quad \triangle(\triangle y z)(\triangle x z) \Rightarrow v}{\triangle f z \Rightarrow v}$$

$$\frac{f \Rightarrow \triangle t y \quad t \Rightarrow \triangle w x \quad \triangle(\triangle z w) x \Rightarrow v}{\triangle f z \Rightarrow v}$$

Fig. 6.4 Root Evaluation

The rules in Figure 6.4 are implemented by the program root given in Figure 6.5. Here is some explanation. It is easy to see that root $\triangle$ is an application of $\triangle\triangle\triangle$ which becomes $\triangle$. Otherwise, the application of root to some quote $\triangle f z$ becomes

$$\mathsf{onFork}\{\mathsf{rootn}\}(\mathsf{root}\ f)\ \mathsf{root}\ z\ .$$

Now consider root f. If this evaluates to a leaf or stem v_1 then the whole term evaluates to $v_1 z$ as required. If it evaluates to some fork $\triangle t y$ then the whole term evaluates to

$$\mathsf{rootn}\ t\ y\ \mathsf{root}\ z = \mathsf{triage}\{\mathsf{rootl}, \mathsf{roots}, \mathsf{rootf}\}\ (\mathsf{root}\ t)\ \mathsf{root}\ y\ z$$

Finally, triage on root t triggers evaluations that express the last three rules of the strategy.

$$\begin{aligned}
\mathsf{rootl} &= \lambda^* r.\lambda^* y.\lambda^* z.ry \\
\mathsf{roots} &= \lambda^* x.\lambda^* r.\lambda^* y.\lambda^* z.r(\triangle(\triangle yz)(\triangle xz)) \\
\mathsf{rootf} &= \lambda^* w.\lambda^* x.\lambda^* r.\lambda^* y\lambda^* z.r(\triangle(\triangle zw)x) \\
\mathsf{rootn} &= \lambda^* t.\lambda^* y.\lambda^* r.\mathsf{triage}\{\mathsf{rootl}, \mathsf{roots}, \mathsf{rootf}\}\ (rt)ry \\
\mathsf{root} &= Y_2\{\lambda^* r.\lambda^* a.\triangle a\triangle(\lambda^* f.\mathsf{onFork}\{\mathsf{rootn}\}(rf)r)\ .
\end{aligned}$$

Fig. 6.5 A Root Self-Evaluator

Theorem 6.3 (root_eval_to_root)

If $M \Rightarrow P$ *then* root $M = P$ *for all combinations* M.

Proof The proof is by induction on the structure of the derivation. □

A converse to this theorem will be proved in Chapter 7.

6.6 Root-and-Branch Evaluation

$$\frac{x \Rightarrow \triangle}{x \Downarrow \triangle} \qquad \frac{x \Rightarrow \triangle y \quad y \Downarrow v}{x \Downarrow \triangle v} \qquad \frac{x \Rightarrow \triangle yz \quad y \Downarrow v \quad z \Downarrow w}{x \Downarrow \triangle vw}$$

Fig. 6.6 Root-and-Branch Evaluation

Root-and-branch evaluation performs root evaluation and then recursively evaluates the branches to produce a normal form. The rules for this evaluation $M \Downarrow P$ are given in Figure 6.6. It is implemented by

$$\mathsf{rb} = Y_2\lambda^* r.\lambda^* x.\mathsf{triage}\{K\triangle, \lambda^* y.\lambda^* r.\triangle(ry), \lambda^* y.\lambda^* z.\triangle(ry)(rz)\}(\mathsf{root}\ x)\ r\ .$$

Theorem 6.4 (rb_eval_implies_rb)

If $'M \Downarrow P$ *then* rb $'M = P$ *for all combinations* M *and programs* P .

Proof The proof is by induction on the structure of the derivation. □

6.7 Root-First Evaluation

Finally, root-first evaluation of the application of the program M to the program N is given by root-and-branch evaluation of $'(MN)$. Since quotation of programs is definable, root-first evaluation is implemented by

$$\mathsf{rf} = \lambda^* f.\lambda^* z.\mathsf{rb}\ (\triangle\ (\mathsf{quote}\ f)(\mathsf{quote}\ z))\ .$$

Theorem 6.5 (root_first_eval_to_rf)

If $'(MN) \Downarrow P$ *then* $\mathsf{rf}\ M\ N = P$ *for all programs* M *and* N *and* P.

.

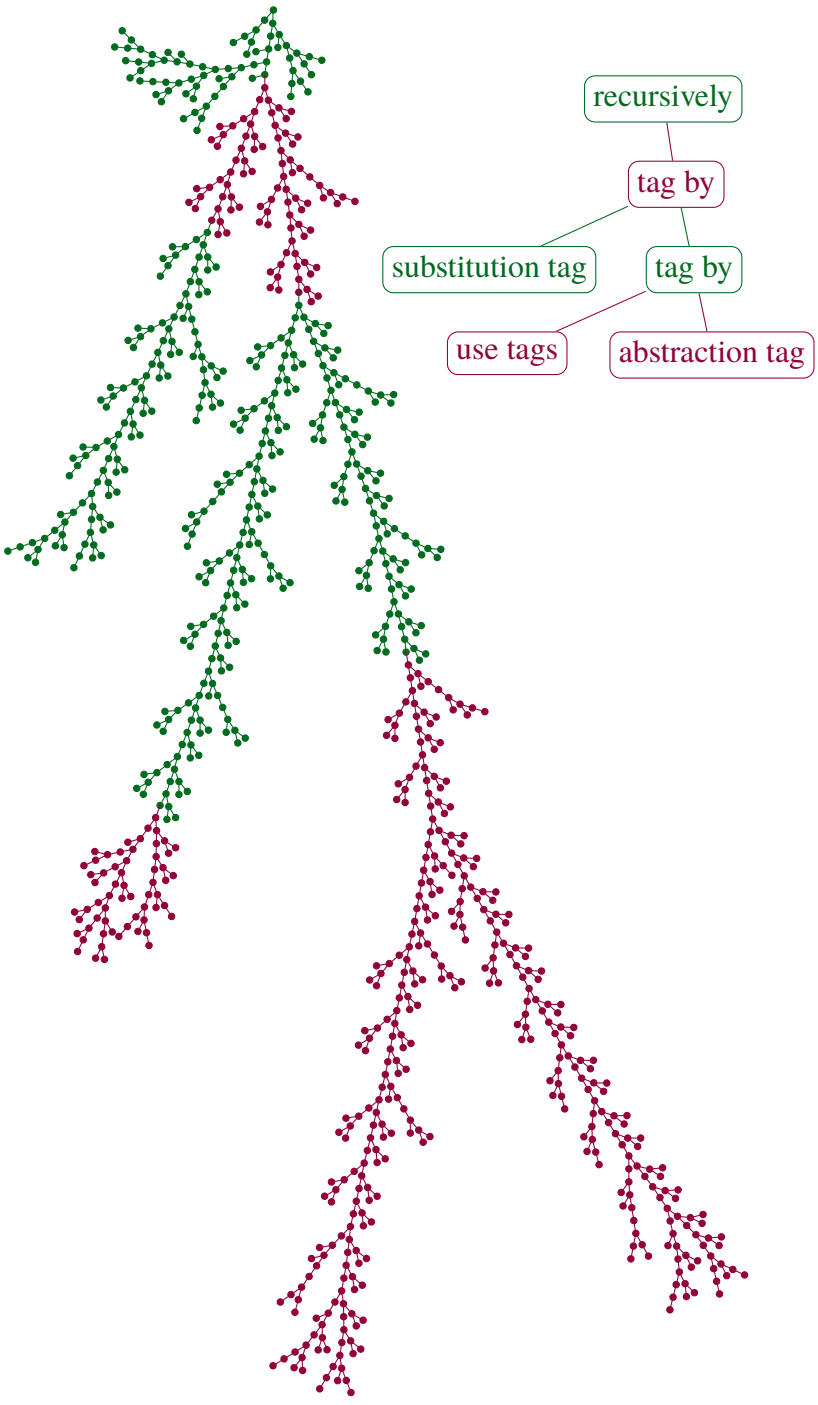
As noted earlier, each of the theorems in this chapter has a converse, showing that if the self-evaluator produces a result then so does the evaluation strategy. However, the proofs require the expressive power of rewriting theory.

Exercises

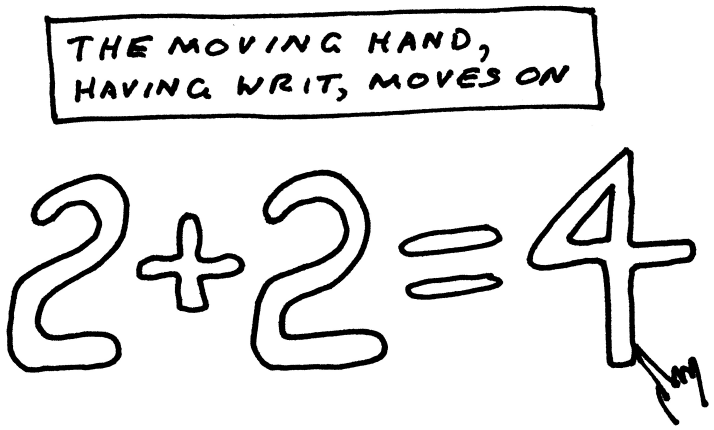
1. Try to evaluate the following applications of programs using branch-first evaluation:
$$K\triangle K$$
$$KI\triangle K$$
$$(\triangle(\triangle I)I)(\triangle(\triangle I)I)$$
$$K(\triangle(\triangle I)I)(\triangle(\triangle I)I)K$$
$$KI(\triangle(\triangle I)I)(\triangle(\triangle I)I)K\ .$$
Hint: evaluate them by hand first, to look for traps.
2. Repeat the previous exercise using root-first evaluation. What has changed?

Part II
Other Models of Computation

A Tree for λ



Chapter 7
Rewriting



7.1 Proving Negative Results

Equational reasoning is well able to show that things do happen, e.g that applying the program bf produces the same results as branch-first evaluation. However, it is ill-suited to showing that things don't happen, e.g. that bf does not produce any results other than those obtained by branch-first evaluation. More fundamentally, it cannot show that stems are never equal to forks. This would require an analysis of all possible equational calculations, which is altogether harder than merely producing a particular chain of equations. When the equations of tree calculus are only ever applied from left to right then it is easy to show that stems reduce to stems and forks to forks but this direction is not enforced, so that we might conceivably have

$$\triangle\triangle = \triangle\triangle\triangle\triangle = \ldots = \triangle\triangle\triangle?$$

That such a chain of equations cannot exist is completely dependent on the choice of the original evaluation rules of tree calculus. For example, if we were to add the equation $\triangle\triangle\triangle = \triangle\triangle$ then the result would be immediate. Further, from this we could show

$$\begin{aligned} \triangle x &= \triangle\triangle\triangle\triangle x \\ &=? \triangle\triangle\triangle x \\ &= \triangle \end{aligned}$$

which would make every combination equal to $\triangle$.

To show that this sort of derivation is impossible we will replace equations with *reduction rules* that are oriented from left to right.

Now there are two equality relations on terms. The *structural equality* $M \equiv N$ asserts that M and N have exactly the same structure. The *computational equality* $M = N$ is generated by the reduction rules by declaring that if $M \longrightarrow N$ then $M = N$. For example, $\triangle\triangle\triangle\triangle$ is not structurally equal to $\triangle\triangle\triangle(\triangle\triangle)$ but they are computationally equal, since both reduce to $\triangle$.

The reduction rules give less freedom than computational equality because the rules have a direction from left to right, but have more freedom than, say branch-first evaluation as the reduction rules can still be applied to any sub-term. This proves to be a convenient place to prove results by structural induction, which works with sub-terms.

7.2 Rewriting

A *reduction system* is given by some collection of terms and a binary relation on those terms. If M and N are so related then we may write

$$M > N \ .$$

The greater-than symbol $>$ is used to suggest that it imposes an order on the terms, in which the *redex* M may be replaced by its *reduct* N but not vice-versa. Closing these rules under all of the term formation rules produces the corresponding *rewriting relation* $r \longrightarrow s$. Since this book is never concerned with reduction systems that are not rewriting systems, we will write $r \longrightarrow s$ for $r > s$ without any risk of confusion. Just as we have chains of equations $M_1 = M_2 = \ldots = M_k$ we now have chains of reductions $M_1 \longrightarrow M_2 \longrightarrow \ldots \longrightarrow M_k$. When such as chain exists we may write $M_1 \longrightarrow^k M_k$ for the k-step reduction relation. When the choice of k is not important, we may write $M \longrightarrow^* N$. That is, $\longrightarrow^*$ is the reflexive, transitive closure of $\longrightarrow$. We may write $M \longrightarrow N$ for $M \longrightarrow^* N$ when the distinction between a *one-step reduction* and a *multi-step reduction* is not important. As noted above, every rewriting relation gives rise to a corresponding notion of *computational equality* which is obtained by adding the rule $N \longrightarrow M$ whenever we have $M \longrightarrow N$.

It is not always clear how to orient evaluation equations to form rewriting rules. For example, consider the rules for Roman numerals in Section 2.7. If they are all oriented from left to right, then canonical forms will be tallies, consisting of Is only. Alternatively, if the rules are oriented from right to left, then the traditional representations of Roman numerals will become canonical.

The general ideas introduced in the rest of the chapter will be illustrated using two examples derived from Part I, namely the arithmetic expressions of Chapter 2 and the tree calculus of Chapter 3.

Recall that the arithmetic expressions in Section 2.8 are given by the BNF

$$m, n ::= \mathsf{zero} \mid \mathsf{successor}\ m \mid (m + n)\ .$$

The rules for addition have a natural order, as

$$\mathsf{zero} + n > n$$
$$\mathsf{successor}\ m + n > m + \mathsf{successor}\ n\ .$$

Note that this order is different from the numerical order in which five is greater than three. The closure of the addition rules above under the term formation rules is achieved by

(i) If $m_1 \longrightarrow m_2$ then $\mathsf{successor}\ m_1 \longrightarrow \mathsf{successor}\ m_2$.
(ii) If $m_1 \longrightarrow m_2$ then $m_1 + n \longrightarrow m_2 + n$.
(iii) If $n_1 \longrightarrow n_2$ then $m_1 + n_1 \longrightarrow m_1 + n_2$.

Note that there are two rules for reducing additions; one for reducing on the left of the addition, and one for reducing on its right. This distinction is important when discussing the order in which reductions are made. It is routine to show that multi-step reduction preserves the structure.

Often properties of one-step reduction carry over to multi-step reduction. Here are some examples.

Theorem 7.1 *If* $e_1 \longrightarrow^* e_2$ *then* $\mathsf{successor}\ e_1 \longrightarrow^* \mathsf{successor}\ e_2$.

Proof The proof is by induction on the length of the reduction $e_1 \longrightarrow^* e_2$. If the reduction takes no steps then e_2 is e_1 and the results follow by reflexivity. Suppose that the reduction of e_1 is given by $e_1 \longrightarrow e_3 \longrightarrow^* e_2$ and the result holds for $e_3 \longrightarrow^* e_2$. Then we have

$$\mathsf{successor}\ e_1 \longrightarrow \mathsf{successor}\ e_3 \longrightarrow^* \mathsf{successor}\ e_2$$

by induction, and the result follows by transitivity of $\longrightarrow^*$. □

Theorem 7.2 *If* e_1, e_2 *and* e_3 *are arithmetic expressions such that* $e_1 \longrightarrow^* e_2$ *then* $e_1 + e_3 \longrightarrow^* e_2 + e_3$ *and* $e_3 + e_1 \longrightarrow^* e_3 + e_2$.

Proof The proof is by induction on the length of the reduction $e_1 \longrightarrow^* e_2$. If the reduction takes no steps then e_2 is e_1 and the results follow by reflexivity.

Suppose that the reduction of e_1 is given by $e_1 \longrightarrow e_4 \longrightarrow^* e_2$ and the result holds for $e_4 \longrightarrow^* e_2$. Then we have

$$e_1 + e_3 \longrightarrow e_4 + e_3 \longrightarrow^* e_2 + e_3$$

by induction. Similarly, we have $e_3 + e_1 \longrightarrow e_3 + e_4 \longrightarrow^* e_3 + e_2$. □

Corollary 7.1 *If e_1, e_2, e_3 and e_4 are arithmetic expressions such that $e_1 \longrightarrow^* e_2$ and $e_3 \longrightarrow^* e_4$ then $e_1 + e_3 \longrightarrow^* e_2 + e_4$.*

Proof By two applications of the previous theorem, we have

$$e_1 + e_3 \longrightarrow^* e_2 + e_3 \longrightarrow^* e_2 + e_4$$

and the result follows by transitivity. Note that we could have used a different reduction sequence, namely

$$e_1 + e_3 \longrightarrow^* e_1 + e_4 \longrightarrow^* e_2 + e_4 \ .$$

For tree calculus, the evaluation rules will be oriented from left to right, to get

$$\begin{aligned} \triangle\triangle yz &\longrightarrow y && (K)\\ \triangle(\triangle x)yz &\longrightarrow yz(xz) && (S)\\ \triangle(\triangle wx)yz &\longrightarrow zwx && (F) \end{aligned}$$

7.3 Normal Forms

From an abstract point of view, the goal of rewriting is to reduce a term until it can reduce no more. A term r is *irreducible* if it cannot be reduced, i.e. for all terms s there is no reduction $r \longrightarrow s$.

In practice, however, the goal is to produce a result or value that is in a particular class of sub-terms. For example, the arithmetic expressions are to produce numbers, represented by the *numerals*, given by the BNF

$$m, n ::= \mathsf{zero} \mid \mathsf{successor}\ m \ .$$

Again, the trees are to produce programs, represented by the binary trees.

When these values are exactly the irreducible terms then they may also be called *normal forms*.

As well as the normal forms, it is sometimes convenient to consider terms which reduce to a normal form. If t is a term that reduces to a normal form n by some reduction sequence $t \longrightarrow^* n$ then t is *normalisable* or *valuable*If, further, every reduction sequence starting at t stops at some normal form then t is *strongly normalisable*. If t is merely normalisable we may say it is *weakly normalisable* to emphasise that it is is not necessarily strongly normalisable.

For example, suppose that M and N are programs of tree calculus. Then $\mathsf{wait}\{M, N\}$ is also normal but WMN is not. Rather it is strongly normalisable, since all reduction sequences end with $\mathsf{wait}\{M, N\}$. Weak normality is common among applications of the fixpoint combination Y. For example, $Y(KI)\triangle$ is weakly normalisable since

$$Y(KI)\triangle \longrightarrow^* KI(Y(KI))\triangle \longrightarrow^* \triangle$$

but it is not strongly normalisable since $YK \longrightarrow^* K(YK)$ yields unbounded reduction sequences.

A reduction relation is *normalising* if each term has a normal form. That is, there is some *some* reduction sequence which reduces it to normal form. A reduction relation is *strongly normalising* if every reduction sequence must terminate. This distinction between "some" and "every" is important in practice, since if some reduction sequences do not terminate then either cleverness or strategy may be required to find the normal form.

Theorem 7.3 *Every arithmetic expression reduces to a numeral.*

Proof Let e be an arithmetic expression. The proof is by induction on the structure of e. If e is zero then the result is immediate. If e is successor e_1 and e_1 has the property, then there is a numeral n_1 such that $e_1 \longrightarrow^* n_1$. Choose n to be successor n_1. Then $e \longrightarrow^* n$ by Theorem 7.1.

If e is of the form $e_1 + e_2$ and e_1 and e_2 have the property then there are numerals n_1 and n_2 such that $e_1 \longrightarrow^* n_1$ and $e_2 \longrightarrow^* n_2$. By Corollary 7.1, we have $e_1 + e_2 \longrightarrow^* n_1 + n_2$. By Theorem 7.3 there is a numeral n such that $n_1 + n_2 \longrightarrow^* n$. Then

$$e = e_1 + e_2 \longrightarrow^* n_1 + n_2 \longrightarrow^* n$$

as required. □

Hence, reduction of arithmetic expressions is weakly normalising. Actually, it is strongly normalising, too.

In tree calculus, it is easy to see that the irreducible combinations are exactly the programs. However, it is tricky to characterise the irreducible terms, since variables may block reduction in various ways. For example,

$$x\triangle\triangle\triangle \quad \text{and} \quad \triangle(x\triangle\triangle\triangle)\triangle\triangle$$

are normal forms but

$$\triangle(\triangle(x\triangle\triangle\triangle))\triangle\triangle\triangle$$

reduces to $\triangle\triangle(x\triangle\triangle\triangle\triangle)\triangle$ and $x\triangle\triangle\triangle\triangle$. Happily, we will not need to characterise the terms that are normal forms, as our focus is on the combinations.

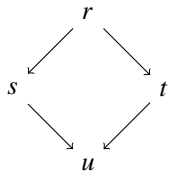
All of the equational reasoning about tree calculus in Part I carries over to the reduction system considered here. Indeed, all of the equational proofs in Coq for Part I have been reworked for the rewriting system, too.

7.4 The Diamond Property

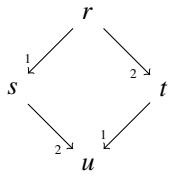
We have seen that every arithmetic expression can be reduced to a numeral. This section will show that the normal form of an arithmetic expression is unique. From this, it follows that numerals are computationally equal if and only if they are

structurally equal. There are several ways to do this, but we will adopt an approach which facilitates the later discussion of tree calculus, by introducing the diamond property.

A reduction relation $\longrightarrow$ satisfies the *diamond property* if, for each pair of reductions $r \longrightarrow s$ and $r \longrightarrow t$ there is a term u such that $s \longrightarrow u$ and $t \longrightarrow u$:



More generally, a *pair* of relations $\longrightarrow_1$ and $\longrightarrow_2$ satisify the *diamond property* if, for each pair of reductions $r \longrightarrow_1 s$ and $r \longrightarrow_2 t$ there is a term u such that $s \longrightarrow u$ and $t \longrightarrow u$:



Clearly, if $\longrightarrow_1$ and $\longrightarrow_2$ satisfy the diamond property, then so do $\longrightarrow_2$ and $\longrightarrow_1$.

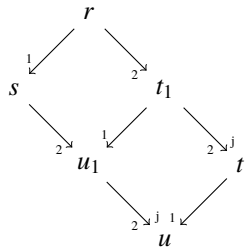
The diamond property extends to reflexive, transitive closures in two steps.

Lemma 7.1 *Whenever the diamond property holds for a pair of relations $\longrightarrow_1$ and $\longrightarrow_2$ then it holds for $\longrightarrow_1$ and $\longrightarrow_2^k$ for any natural number k. Hence it holds for $\longrightarrow_1$ and r_2^*.*

Proof The proof is by induction on k. Suppose that $r \longrightarrow_1 s$ and $r \longrightarrow_2^k t$.

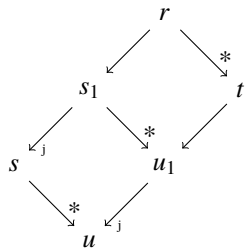
If $k = 0$ then $t = r$ and the diamond is completed by s.

If k is the successor of j then there is some t_1 such that $r \longrightarrow_2 t_1 \longrightarrow_2^j t$. By the diamond property for $\longrightarrow_1$ and $\longrightarrow_2$ there is some u_1 such that $s \longrightarrow_2 u_1$ and $t_1 \longrightarrow_1 u_1$. By induction, there is a u such that $u_1 \longrightarrow_2^j u$ and $t \longrightarrow_1 u$. Hence u is the desired term, as shown by the diagram below:



Lemma 7.2 *If $\longrightarrow$ has the diamond property then so does $\longrightarrow^*$.*

Proof It is enough to show that $\longrightarrow^k$ and $\longrightarrow^*$ have the diamond property for any k. Suppose that $r \longrightarrow^k s$ and $r \longrightarrow^* t$. If k is 0 then the result is immediate. If k is the successor of some j and we have $r \longrightarrow s_1 \longrightarrow^j s$ then we can apply Lemma 7.1 to r, s_1 and t to produce a common reduct u_1 of s_1 and t. Then induction yields a common reduct u of s and u_1 as in the following diagram



Now we can show that arithmetic expressions evaluate to at most one numeral.

Theorem 7.4 *Reduction of arithmetic expressions satisfies the diamond property.*

Proof Suppose that r, s and t are arithmetic expressions such that $r \longrightarrow s$ and $r \longrightarrow t$. The proof is by induction on the structure of r. If r is a numeral then it is irreducible, so choose u to be r. If r is an addition $r_1 + r_2$ then there are four possible ways that r can reduce to s.

(Case 1) Suppose s is $s_1 + r_2$ and $r_1 \longrightarrow s_1$. Now consider $r \longrightarrow t$. Again there are four possibilities.

Suppose that t is $t_1 + r_2$ and $r_1 \longrightarrow t_1$. Then by induction there is a term u_1 such that $s_1 \longrightarrow u_1$ and $t_1 \longrightarrow u_1$ so choose u to be $u_1 + r_2$.

Suppose that t is $r_1 + t_2$ and $r_2 \longrightarrow t_2$. Then choose $u = s_1 + t_2$.

Suppose that r_1 is zero. Then r_1 is both irreducible and reduces to s_1 which yields a contradiction.

Finally, suppose that r_1 is some successor successor r_3 and t is $r_3 +$ successor r_2. Then s_1 must be of the form successor s_3 where $r_3 \longrightarrow s_3$. So choose u to be $s_3 +$ successor r_2.

(Case 2) Suppose that s is $r_1 + s_2$ where $r_2 \longrightarrow s_2$. Then proceed as in Case 1.

(Case 3) Suppose that r_1 is zero and s is r_2. Then either t is r_2, in which case choose u to be r_2 or t is of the form zero $+ t_2$ where $r_2 \longrightarrow t_2$, in which case choose u to be t_2.

(Case 4) Suppose that r_1 is some successor successor r_3 and s is $r_3 +$ successor r_2. Then there are four possibilities for $r \longrightarrow t$ which are handled by variations on the arguments above.

Theorem 7.5 *If an arithmetic expression e reduces to two numerals m and n then m and n are structurally equal.*

Proof Since $\longrightarrow$ satisfies the diamond property, it follows that $\longrightarrow^*$ satisfies the diamond property, by Lemma 7.2. Hence, there is an expression u such that $m \longrightarrow^* u$

and $n \longrightarrow^* u$. However, numerals are irreducible by $\longrightarrow$ so it must be that $u \equiv m$ and $u \equiv n$. Hence $m \equiv n$ by transitivity. □

7.5 Confluence of Tree Calculus

Unfortunately, the diamond property fails for the one-step reduction of tree calculus because the stem rule may duplicate redexes, as in Figure 7.1.

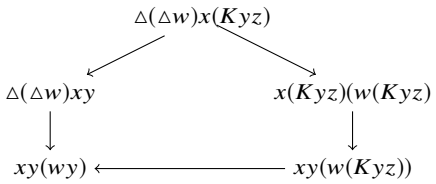
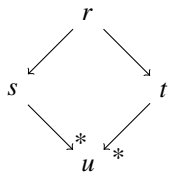


Fig. 7.1 Reduction may Duplicate Redexes

That is, it takes *two reduction steps* on the right to complete the diagram, when the diamond property insists on using one only.

In order to show that every combination reduces to at most one program, it is enough to show that reduction is confluent, in the following sense.

A relation $\longrightarrow$ is *confluent* if $r \longrightarrow s$ and $r \longrightarrow t$ implies there is a term u such that $s \longrightarrow^* u$ and $t \longrightarrow^* u$



Theorem 7.6 *If $\longrightarrow$ is a confluent reduction relation then each term has at most one normal form.*

Proof Suppose that $r \longrightarrow^* s$ and $r \longrightarrow^* t$ and that s and t are irreducible. Since $\longrightarrow$ is confluent, it follows that $\longrightarrow^*$ satisfies the diamond property, so that there is a term u such that $s \longrightarrow^* u$ and $t \longrightarrow^* u$. Since s and t are irreducible, it follows that s and t are both u. □

Corollary 7.2 *With respect to a confluent, weakly normalising reduction, every term has a unique normal form.*

Corollary 7.3 *Every arithmetic expression has a unique normal form.*

Actually, we will not require confluence in this general form. Instead, we will show the diamond property for $\longrightarrow^*$.

Lemma 7.3 *$\longrightarrow$ is confluent iff $\longrightarrow^*$ satisfies the diamond property.*

Proof Exercise. □

This may seem like a backward step, in that it is harder to reason about multi-step reductions than single steps. However, we will generalise the reduction relation to support *simultaneous reduction* $\longrightarrow_s$ so that both copies of the redex Kxy above can be reduced in a single step, as $x(Kyz)(w(Kyz)) \longrightarrow_s xy(wy)$. More generally, $\longrightarrow_s$ will prove to satisfy the diamond property, so that $\longrightarrow_s^*$ does too. Finally, $\longrightarrow_s^*$ will prove to be the same relation as $\longrightarrow^*$ so that the latter satisfies the diamond property and $\longrightarrow$ is confluent.

$$\frac{}{\triangle \longrightarrow_s \triangle} \qquad \frac{x \longrightarrow_s x' \quad y \longrightarrow_s y'}{xy \longrightarrow_s x'y'} \qquad \frac{y \longrightarrow_s y'}{\triangle\triangle yz \longrightarrow_s y'}$$

$$\frac{x \longrightarrow_s x' \quad y \longrightarrow_s y' \quad z \longrightarrow_s z'}{\triangle(\triangle x)yz \longrightarrow_s y'z'(x'z')} \qquad \frac{w \longrightarrow_s w' \quad x \longrightarrow_s x' \quad z \longrightarrow_s z'}{\triangle(\triangle wx)yz \longrightarrow_s z'w'x'}$$

Fig. 7.2 Simultaneous Reduction in Tree Calculus

The *simultaneous reduction* relation is given in Figure 7.2. Now the pentagon in Figure 7.1 can be collapsed to a diamond with respect to simultaneous reduction, as shown in Figure 7.3.

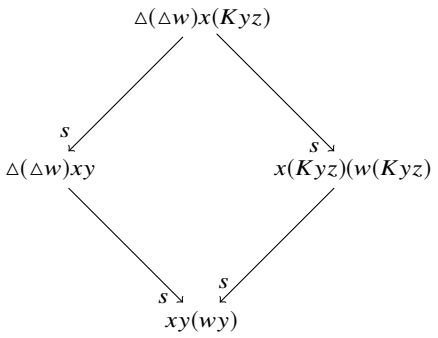


Fig. 7.3 Simultaneous Reduction can Handle Duplicate Redexes

Lemma 7.4 *For all combinations M and N of tree calculus, $M \longrightarrow^* N$ if and only if $M \longrightarrow_s^* N$.*

Theorem 7.7 (diamond_s_red1)

Simultaneous reduction has the diamond property.

Theorem 7.8 (confluence_tree_calculus)

Reduction of tree calculus is confluent.

Proof Since reduction has the same transitive closure as simultaneous reduction, it is enough to prove confluence of the latter, which follows from Theorem 7.7. □

Corollary 7.4 *Every tree has at most one normal form.*

Proof Apply Theorem 7.6. □

7.6 The Halting Problem

Some trees don't have any normal form. For example, the application of $\omega = (\triangle(\triangle I)I$ to itself reduces to itself by

$$\omega\omega = (\triangle(\triangle I)I\omega \longrightarrow I\omega(I\omega) \longrightarrow \omega\omega \ .$$

With a little effort one can prove that no other reduction sequence does any better.

Given that trees without normal forms exist, can the existence of a normal form be decided by a program? This is the adaptation to tree calculus of the Halting Problem posed by Alan Turing; the answer is no.

Define a *halting function* to be a combination h such that, for each combination x, either hx reduces to K and x has a normal form or hx reduces to KI and x does not have a normal form.

Theorem 7.9 (halting_problem_insoluble) *There is no halting function in tree calculus.*

Proof Suppose that h is a halting function. Define

$$h' = \lambda^* x.h(xx)(\omega\omega)K.$$

Then $h'h' = h(h'h')(\omega\omega)K$. Now, if $h(h'h') = K$ then $h'h'$ has a normal form but also $h'h' = K(\omega\omega)K = \omega\omega$ which does not have a normal form. Contradiction. On the other hand, if $h(h'h') = KI$ then $h'h'$ does not have a normal form but also $h'h' = KI(\omega\omega)K = K$ which does have a normal form. Contradiction. Since both alternatives lead to contradiction there can be no halting function h. □

7.7 Standard Reduction

The root-first evaluation of Chapter 6.7 has a useful property that is not shared by the other evaluation strategies of Chapter 6: if a combination M has a normal form N then root-and-branch evaluation will find it. This follows from a more general

result: if N contains k redexes then there is a standard reduction sequence from M to N with at most k violations of the root-and-branch strategy, in a sense that will be made precise. Thus, if k is zero then there are no violations, so that root-and-branch evaluation succeeds. An outline of the proof follows: details can be found in the Coq verification.

A combination x is *ready* if $x\triangle$ is a redex. That is, x is of the form $\triangle t\,y$ where t is factorable.

An *exact right reduction* $M \longrightarrow_n^1 N$ is a reduction $M \longrightarrow N$ in which the redex in M is to the right of exactly n other redexes in M. Call n the number of *violations* of root-first evaluation.

A *right reduction* $M \longrightarrow_n N$ is given by a sequence of exact right reductions, with some constraints on the violations, that are captured by the following derivation rules

$$\frac{}{M \longrightarrow^n M}\ n \leq \mathsf{redexes}(M) \qquad \frac{M \longrightarrow_1^m N \quad N \longrightarrow^n P}{M \longrightarrow_n P}\ m \leq n.$$

That is, when right reducing from M to P, the number of possible violations may grow with each step, provided that the final tally is no greater than the number of redexes in P.

If no violations are allowed, so that $M \longrightarrow^0 P$, then this is called *leftmost, outermost reduction*.

Head reduction is a restricted form of leftmost, outermost reduction which stops when a factorable form is found. One-step head reduction is given by

$$\frac{}{\triangle\triangle yz \longrightarrow_h y} \qquad \frac{}{\triangle(\triangle x)yz \longrightarrow_h yz(xz)} \qquad \frac{}{\triangle(\triangle wx)yz \longrightarrow_h zwx}$$

$$\frac{x \longrightarrow_h x'}{xz \longrightarrow_h x'z} \qquad \frac{x \longrightarrow_h x'}{\triangle xyz \longrightarrow_h \triangle x'yz}$$

A *staged reduction* $M \longrightarrow_s P$ begins with a sequence of head reductions $M \longrightarrow_h^* N$. Then, if N is some application N_1N_2 it must be that P is some application P_1P_2 with staged reductions from N_1 to P_1 and N_2 to P_2.

Theorem 7.10 (standardization)

If $M \longrightarrow^* N$ then $M \longrightarrow_n N$ for all combinations M and N where n is the number of redexes in N.

Proof If $M \longrightarrow^* N$ then lemmas show that there is a staged reduction from M to N and so $M \longrightarrow_{\mathsf{redexes}(N)} N$. □

Corollary 7.5 (leftmost_reduction) *If M is a combination and N is a program then $M \longrightarrow^* N$ implies there is a leftmost, outermost reduction from M to N.*

Proof If N is a program then it has no redexes. Now apply Theorem 7.10. □

Theorem 7.11 (head_reduction_to_factorable_form)

If M is a combination and N is a factorable form and $M \longrightarrow^* N$ then there is a factorable form Q such that $M \longrightarrow_h^* Q$ and $Q \longrightarrow^* N$.

Proof As above, there is a staged reduction from M to N. This is given by some head reduction $M \longrightarrow_h^* Q$ followed by some reduction of the branches of Q to produce N. Since N is a factorable form, it follows that Q is too. □

7.8 Converse Properties of the Self-Evaluators

Equipped with the machinery of rewriting, we can now prove that the self-evaluators of Chapter 6 do no more than is allowed by the corresponding strategies. All the proofs follows the same pattern. Let us consider branch-first evaluation and bf.

The key idea is that if bf M N has a factorable form then so does M. That is, bf M N needs M since the whole term reduces to something whose next evaluation step must involve M. The proof of the theorem proceeds by induction on the number of simultaneous reduction steps required to reduce bf M N and then case analysis on M. For example, if M is of the form $\triangle(\triangle wx)y$ then bf (bf N w) x also reduces to P, and this in fewer steps. Since P is factorable, and bf needs its first argument, it follows that bf N w reduces to some factorable form, N_1. Thus $N, w \Rightarrow N_1$ by induction. Further, bf N_1 x also reduces to P, so a second use of induction shows that $N_1, x \Rightarrow P$ and the result follows. Although the idea is not so difficult to grasp, the formal proof is quite dense. Here are the definitions required for the key lemma.

A term M is *next* in a term R if R is of the form $\triangle MNP$ or if R is of the form $R_1 P$ where M is next in R_1.

A term M is *active* in a term R if either M is next in R or there is some N such that M is next in N and N is active in R.

A term R *needs* a term M if R reduces to some P in which M is active.

It is routine to prove that bf M N needs M and eager M N needs M.

Lemma 7.5 (needy_s_red_factorable)

If R needs M and R reduces to a factorable form P in n simultaneous reduction steps then M reduces to some factorable form Q in n simultaneous steps.

Proof The proof is by induction on n. □

Theorem 7.12 (bf_to_branch_first_eval)

For all programs M and N and factorable forms P, if bf M $N \longrightarrow^* P$ *then there is a Q such that $M, N \Rightarrow Q$ and $P \longrightarrow^* Q$.*

Proof The proof is by induction on the number of simultaneous reductions to P and then case analysis of M. If M is a leaf or stem then bf M $N \longrightarrow^* MN$ and $M, N \Rightarrow MN$ so the result follows from confluence of simultaneous reduction. So

suppose that M is a fork $\triangle M_0 M_1$. If M_0 is a leafQ then bf $M\ N \longrightarrow^* M_1$ and $M, N \Rightarrow M_1$ and the result follows from confluence.

If M_1 is a fork $\triangle M_2 M_3$ then bf $M\ N \longrightarrow^*$ bf (bf $N\ M_2$) M_3. By confluence, there is some P_1 with simultaneous reductions from bf (bf $N\ M_2$) M_3 and P. It follows that P_1 is factorable. Further, the first of these two reductions takes the same number of steps as the original reduction to P. Actually, since there is a unique simultaneous reduction of bf $(\triangle(\triangle M_3 M_4)\ M_2)\ N$ to some P_0 we can rebuild the diamond to show that the simultaneous reduction from bf (bf $N\ M_3$) $M_4 \longrightarrow^* P_1$ takes one less step than the original. This is illustrated in the first square of Figure 7.4.

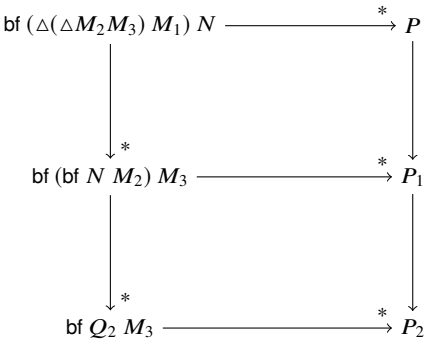


Fig. 7.4 Branch-First Evaluation Using the Fork Rule

Thus, by Lemma 7.5 there is a simultaneous reduction of bf $N\ M_2$ to some factorable form Q_0. Hence, by the induction hypothesis, there is a branch-first evaluation $N, M_2 \Rightarrow Q_2$ for some Q_2. Further, Q_2 is a program, by Theorem 6.1. So a second application of Lemma 7.5 yields a factorable form P_2 that completes the second square in Figure 7.4.

Finally, a second application of the induction hypothesis yields an evaluation $Q_2, M_3 \Rightarrow P_3$ for some factorable form P_3 where P_2 reduces to P_3. Now P_3 is the required factorable form. The reduction from P is by transitivity of reduction: the evaluation is given by

$$\frac{N, M_2 \Rightarrow Q_2 \quad Q_2, M_3 \Rightarrow P_3}{\triangle(\triangle M_2 M_3) M_1, N \Rightarrow P_3}$$

If M_0 is a stem $\triangle M_2$ then bf $M\ N \longrightarrow^*$ eager (bf $M_2\ N$) (bf (bf $M_1\ N$)) and the argument follows a similar pattern as above, as illustrated in Figure 7.5.

Corollary 7.6 (branch_first_eval_iff_bf)

For all programs M, N and P, the computation bf $M\ N$ *reduces to P if and only if $M, N \Rightarrow P$.*

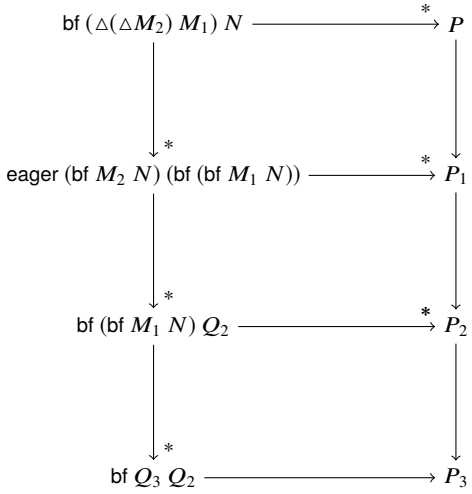


Fig. 7.5 Branch-First Evaluation Using the Stem Rule

Proof Apply Theorems 6.5 and 7.12. □

Happily, the same techniques apply to root evaluation, root-and-branch evaluation and root-first evaluation.

Theorem 7.13 (root_eval_iff_root)

For all combinations M and programs P, the computation root *'M reduces to P if and only if ' M ⇒ P.*

Proof In one direction, apply Theorem 6.3. In the other direction, apply the techniques of Theorem 7.12. □

Theorem 7.14 (rb_eval_iff_rb)

For all combinations M and programs P, the computation rb *'M reduces to P if and only if ' M ⇓ P.*

Proof In one direction, apply Theorem 6.4. In the other direction, use induction on the size of *P* and apply Theorem 7.13. □

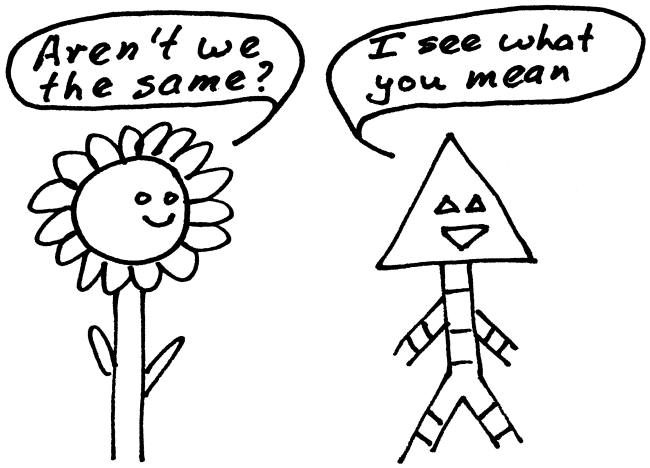
Theorem 7.15 (root_first_eval_iff_rf) *For all programs M, N and P, the computation* rf *M N reduces to P if and only if '(MN) ⇓ P .*

Proof Apply Theorem 7.14. □

Exercises

1. (Hard) Convert the equality of Roman numerals into a rewriting relation such that the normal forms consist of *I*s only, and reduction satisfies the Diamond Property.
2. (Very Hard) Convert the equality of Roman numerals into a rewriting relation such that the normal forms are the usual representations of numbers as Roman numerals only, and reduction satisfies the Diamond Property.
3. Verify all of the lemmas in this chapter, e.g. by running the proofs in Coq.
4. Reprise all of the results form Part I using rewriting instead of equational reasoning.

Chapter 8
Incompleteness of Combinatory Logic



8.1 Logic Without Variables

Combinatory logic emerged from a critique of predicate calculus by Moses Schönfinkel but was recast as a model of computation by Haskell Curry, who emphasised the correspondence with the λ-calculus of Alonzo Church. Although combinatory logic is both simple and expressive, it does not support intensionality, being unable to even decide equality of its programs. It is easy to embed combinatory logic into tree calculus but there is no meaningful translation in the other direction.

Propositional logic is concerned with the validity of arguments such as

$$\frac{p \;\&\; q}{p}$$

where p and q are variables that represent propositions, and the argument asserts that from the conjunction $p\&q$ we can infer the proposition p. Predicate logic is concerned with the validity of arguments such as

$$\frac{\forall x.P(x)\ \&\ Q(x)}{P(x)}$$

where P and Q are variables that represent predicates, so that $P(x)$ is a proposition for any x, and $\forall$ is the universal quantifier that binds x, so that $\forall x.P(x)$ is true if $P(a)$ is true for all values a that x might take.

Unfortunately, the naive use of quantifiers leads to paradoxes, notably Russell's Paradox. One solution is to use *types* to constrains the possible values a that x might take. Schönfinkel's solution was more radical. In the example above, treat P as a function (from x to $P(x)$) and replace the derivation above by

$$\frac{P \wedge Q}{P}$$

where $\wedge$ is a conjunction of functions. Now P and Q are functions (acting on a, b etc.) and $\wedge$ is a function of the functions P and Q that produces another function $P \wedge Q$. That is, Schönfinkel built the first *calculus of functions* in which functions can be used as arguments and returned as results of other functions, so that functions are *first-class citizens*. His goal was a calculus, built from some functions constants such as $\wedge$, that could capture the expressive power of predicate calculus without binding any variables.

8.2 SK-Calculus

As noted in Chapter 4.1, there are a few options when choosing the operators underpinning combinatory logic. Most common is to use S, K and I, as these appear in the definitions of bracket abstraction and star abstraction, but S and K will suffice. We will continue with all of the terminology of Section 4.1 for combinations and terms built from some class O of operators, and that for rewriting of Chapter 7.2.

The operators S and K of SK-calculus have reduction rules

$$\begin{aligned} Kxy &> x \\ Sxyz &> xz(yz)\ . \end{aligned}$$

Now I can be defined to be SKK since

$$SKKx \longrightarrow Kx(Kx) \longrightarrow x\ .$$

Indeed, any combinator of the form SKy is an identity function since $Kx(yx) \longrightarrow x$ for any y.

Theorem 8.1 (confluence_SK) *Reduction of SK-calculus is confluent.*

Proof The proof uses the same techniques as that for tree calculus in Theorem 7.8, by showing that simultaneous reduction satisfies the diamond property. □

Just as in tree calculus, we can define the programs in a rather simple manner, by the BNF

$$p, q ::= S \mid Sp \mid Spq \mid K \mid Kp \ .$$

That these are exactly the irreducible combinations, or normal forms follows from a simple proof by induction on the structure of the combinations.

The reduction rules show that K is used to delete arguments, and S is used to copy them. It turns out that deleting and copying are all that you need to represent any other combinator. In turn, this expressive power will prove to be enough to support all of the extensional functions in Chapter 4. Since the details are not quite the same in SK-calculus as in tree calculus, some redevelopment will be necessary.

8.3 Combinators in SK-Calculus

Every combinator M as in Equation (4.1) can be represented by an SK-combination denoted by

$$M = [x_1][x_2] \ldots [x_n]t$$

where the *bracket abstraction* $[x]t$ represents the function that, when applied to x yields t. It is defined as follows:

$$\begin{aligned} [x]x &= I \\ [x]y &= Ky \quad (y \neq x) \\ [x]O &= KO \quad (O \text{ is an operator}) \\ [x]uv &= S([x]u)([x]v) \ . \end{aligned}$$

The main difference from Chapter 4.2 is that $S([x]u)([x]v)$ has been used instead of $\triangle(\triangle([x]v)([x]u)$. Recall that $\{u/x\}t$ is the result of substituting the term u for the variable x in the term t.

Hence the combinator M in Equation (4.1) is represented by the SK-combination $[x_1][x_2] \ldots [x_n]t$.

Since all combinators can now be represented in SK-calculus, we say that SK-calculus is *combinatorially complete*. Note that this does not imply that $\triangle$ can be defined in SK-calculus, as it was not introduced as a combinator.

As before, we can optimise bracket abstraction by applying some lightweight analysis. Define $x \in t$ by induction on the structure of t just as in Chapter 4.2. Then define *star abstraction* as follows

$$\begin{aligned} \lambda^* x.t &= Kt && (x \notin t) \\ \lambda^* x.tx &= t && (x \notin t) \\ \lambda^* x.x &= I \\ \lambda^* x.y &= Ky && (y \neq x) \\ \lambda^* x.tu &= S(\lambda^* x.t)(\lambda^* x.u) && \text{(otherwise).} \end{aligned}$$

Once again, the only change is to use of S when abstracting from applications.

The differences between bracket abstraction and star abstraction can be highlighted by considering the combinators that correspond to S and K:

$$\begin{aligned} K_0 &= [x][y]x \\ &= [x]Kx \\ &= S(KK)I \\ S_0 &= [x][y][z]xz(yz) \\ &= [x][y]S([z]xz)([z]yz) \\ &= [x][y]S(S(Kx)I)(S(Ky)I) \\ &= \ldots \end{aligned}$$

By contrast, we have

$$\lambda^* x.\lambda^* y.x = \lambda^* x.Kx = K$$
$$\lambda^* x.\lambda^* y.\lambda^* z.xz(yz) = \lambda^* x.\lambda^* y.Sxy = \lambda^* x.Sx = S$$

by which the original operators are recovered.

8.4 Incompleteness of Combinatory Logic

Armed with the techniques of rewriting theory, we can now prove that equality of programs is not definable in combinatory logic. The standard definition of an equality term in SK-calculus is a term eq such that $\mathsf{eq}\ M\ N$ is true or false according to whether the programs M and N are equal or not. Further, let us try and capture the meaning of the booleans in terms of reduction rules, such as $\mathsf{true}\ x\ y \longrightarrow x$. Then preservation of the variables and reduction rules will ensure the preservation of truth, too.

Define an equality term in SK-calculus to a term eq such that, for all variables x and y we have:

- $\mathsf{eq}\ M\ M\ x\ y \longrightarrow^* x$ for all programs M
- $\mathsf{eq}\ M\ N\ x\ y \longrightarrow^* y$ for all programs M and N such that $M \not\equiv N$.

Note that we could have written $M \neq N$ above since structural equality and computational equality are the same for programs.

Our goal is to show that there is no equality term in SK-calculus. The basic idea is that SK-calculus cannot distinguish combinators that have the same functional behaviour, such as the combinators SKK and SKS that both represent the identity function. To formalise this requires some new rewriting machinery.

Define an *identity program* to be a program M such that $Mx \longrightarrow^* x$ for some variable x (the choice of x is immaterial).

Define a *separator* for programs M_1 and M_2 to be a program s such that there are distinct variables x and y for which

$$s\ M_1\ x\ y \longrightarrow x$$
$$s\ M_2\ x\ y \longrightarrow y\ .$$

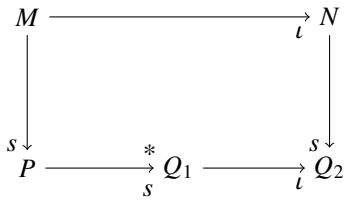
We will show that there are identity programs that cannot be separated in SK-calculus, and so program equality is not definable.

Define the *identity reduction* $\longrightarrow_\iota$ to be the congruence generated by the rule

$$M \longrightarrow_\iota I \quad \text{(if } M \text{ is an identity program).}$$

For our purposes, it is enough to show that identity reduction interacts well with the simultaneous reduction relation $\longrightarrow_s$ that was developed in the proof of confluence.

Lemma 8.1 (pentagon_id_red_s_red1) *For all terms M,N and P and reductions $M \longrightarrow_\iota N$ and $M \longrightarrow_s P$ there are terms Q_1 and Q_2 such that we have reductions $P \longrightarrow_s^* Q_1 \longrightarrow_\iota Q_2$ and $N \longrightarrow_s Q_2$ as illustrated in the following pentagon:*

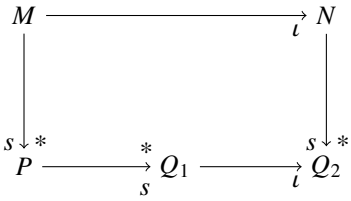


Proof The pentagon is like the diamond property for $\longrightarrow_s$ and $\longrightarrow_\iota$ except that it has beeen padded with some additional simultaneous reductions. The proof requires some careful case analysis of both sorts of reduction, but is not very illuminating. Details can be found in the Coq verification. □

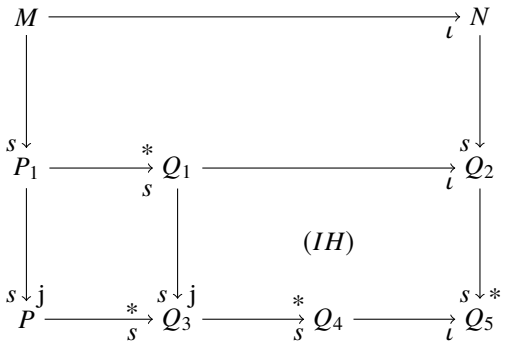
The pentagons can be generalised to handle multiple parallel reductions as follows.

Lemma 8.2 (pentagon_id_red_s_red)

For all terms M,N and P and reductions $M \longrightarrow_\iota N$ and $M \longrightarrow_s^* P$ there are terms Q_1 and Q_2 such that we have reductions $P \longrightarrow_s^* Q_1 \longrightarrow_\iota Q_2$ and $N \longrightarrow_s^* Q_2$ as illustrated in the following pentagon:



Proof The proof is by induction on the length k of the parallel reduction sequence $M \longrightarrow_s^k P$. If k is zero then the result is immediate. If k is $j + 1$ then apply the following diagram:

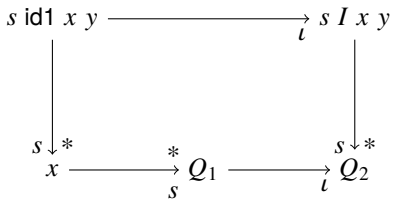


The upper pentagon exists by the previous lemma. The square exists by the confluence of simultaneous reduction. Further, the number of reduction steps from Q_1 to Q_3 is *the same* as the number of reductions from P_1 to P. Hence, the lower pentagon exists by the induction hypothesis. Thus Q_4 and Q_5 are the required terms, since there are simultaneous reductions from $P \longrightarrow_s^* Q_4$ and $N \longrightarrow_s^* Q_5$. □

Theorem 8.2 (no_separable_identities_in_SK)

Identity programs do not have separators in SK-calculus.

Proof Let id1 and id2 be distinct identity programs and assume that they have a separator s using distinct variables x and y. Apply the previous lemma to the identity reduction from id1 to I to get



Now Q_1 must be x since variables are irreducible, and Q_2 must be x since x is not a program. Thus $sIxy \longrightarrow_s^* x$. By a similar argument using id2, we have $sIxy \longrightarrow_s^* y$. Hence x and y share a reduct, by confluence. Hence they are equal, which contradicts the original assumption. □

Theorem 8.3 (equality_of_programs_is_not_definable_in_SK)
There is no equality term in SK-calculus.

Proof If eq were equality of programs then eq (SKK) would be a separator for SKK and SKS which contradicts Theorem 8.2. □

Since equality of programs is an effectively calculable function, it follows that SK-calculus is *not complete* for computation of its own programs. It is worth taking a moment to consider the scope and implications of this theorem. Although it does not limit the ability of SK-calculus to compute numerical functions, it does limit its ability to analyse them. More precisely, we have seen that all of the μ-recursive functions can be represented by extensional programs in tree calculus, and this representation can be repeated in SK-calculus, but there is little support for their optimisation. For example, suppose that program p can be replaced by program q whose computation takes fewer steps. In order to take advantage of this in a large program f one must search f for copies of p and replace them by q. However, there is no SK-combinator that can decide whether some sub-program p' is p or not. By contrast, within tree calculus there are combinations can decide equality of p and p', that can search f for occurrences of p and replace them by q. Thus, tree calculus is a far more promising setting for program analysis and optimisation than combinatory logic.

8.5 Meaningful Translation

Before considering translations between combinatory logic and tree calculus, let us consider what it means for a translation $[-]$ between such calculi to be meaningful, in the sense that the meaning of the terms is preserved.

First of all, computational equality must be preserved, so that if $s = t$ then $[s] = [t]$. Here and in all subsequent requirements, the equalities will be computational equalities.

Second, translation should preserve application, so that

$$[t\ u] = [t][u]\ .$$

Third, values must be preserved. That is, the translation of a program must reduce to a program. Otherwise, successful evaluations in one calculus may be translated to failed computations in the other. For example, suppose that tree calculus is augmented with a fixpoint function operator Z with the rule

$$Z\ f\ x = f(Zf)x\ .$$

and consider a translation back to tree calculus that preserves $\triangle$. If $[Z] = \lambda^* f.Y_2\{f\}$ then all is well but if $[Z] = Y$ then the value $Z\triangle$ will be translated by $Y\triangle$ which has no value.

Finally, variables of the term calculus must be preserved, so that $[x] = x$. This last point is subtle, since the variables are not programs. However, it is necessary to preserve the meaning of the reduction rules. For example, consider a variant of the arithmetic expressions that includes an error term error which blocks any further reduction. Its rules for adding zero become

$$\mathsf{zero} + \mathsf{zero} = \mathsf{zero}$$
$$\mathsf{zero} + \mathsf{successor}\ n = \mathsf{successor}\ n$$

but zero + error does *not* reduce to error. Is the embedding of the arithmetic expressions into this larger calculus meaningful? Certainly, it preserves all reductions of combinations but it does not preserve the reduction of terms, since $\mathsf{zero} + n$ does *not* reduce to n for any variable n.

Another way of looking at it is that the rule

$$\mathsf{zero} + n = n$$

is an assertion about all terms n no matter what their nature might be. So, to preserve the meaning requires that this quantification over all possible terms n should be preserved. By insisting that variables be preserved, we require that this equation hold in the new calculus, which in turn requires that $\mathsf{zero} + \mathsf{error} = \mathsf{error}$ hold for translation to be meaningful.

We can re-express the four requirements above in the language of rewriting as follows.

Definition 8.1 A *meaningful translation* between applicative rewriting systems is a function of their terms that satisfies the following criteria:

1. If $s = t$ then $[s] = [t]$.
2. $[tu] = [t][u]$ for all terms t and u.
3. If v is a value then $[v]$ is valuable.
4. $[x] = x$ for all variables x.

An example of a meaningful translation is the translation of Roman numerals to unary arithmetic in Chapter 2.9. Here is another.

The translation to tree calculus is deceptively simple, being determined by

$$[S] = S$$
$$[K] = K$$

with variables and applications being preserved.

Theorem 8.4 (meaningful_translation_from_sk_to_tree) *There is a meaningful translation from SK-calculus to tree calculus.*

Proof Variables, closed terms and applications are preserved by definition. That one-step reductions of SK-calculus are mapped to multi-step reductions of tree calculus was established in Chapter 3.3. Further, those proofs of preservation also show that partial applications of (the translations of) S and K have normal forms if their arguments do. In particular, values are translated to valuable terms. □

8.6 No Tree Calculus in Combinatory Logic

There is no meaningful translation from tree calculus to SK-calculus since tree calculus supports intensionality in ways that SK-calculus does not.

Lemma 8.3 (meaningful_translation_preserves_identity_programs)

If f is a meaningful translation from SK-calculus to tree calculus then f preserves identity programs.

Proof Routine calculation. □

Lemma 8.4 (meaningful_translation_preserves_separators)

If f is any meaningful translation from SK-calculus to tree calculus then f preserves separators.

Proof Routine calculation. □

Theorem 8.5 (no_translation_tree_to_sk)

There is no meaningful translation from tree calculus to SK-calculus.

Proof Tree calculus supports the program identities $\mathsf{tag}\{K, I\}$ and $\mathsf{tag}\{KI, I\}$ which are separated by getTag. If there were such a meaningful translation to SK-calculus then it would preserve the program identities by Lemma 8.3 and also their separator, by Lemma 8.4. Hence, SK-calculus would support separable identity programs but this contradicts Theorem 8.2 □

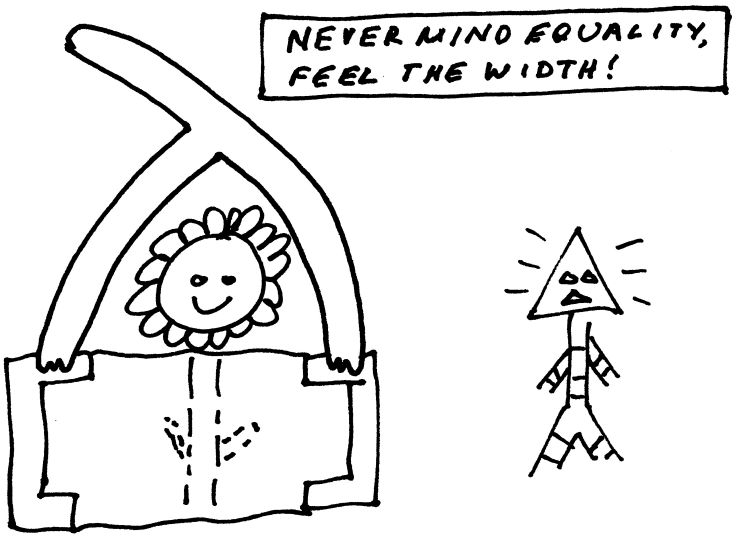
It follows that tree calculus is strictly more expressive than combinatory logic, since there is a meaningful translation from combinators to trees but not in the other direction. In particular, there are functions of programs that are not definable in combinatory logic but are definable in tree calculus, such as the separator of SKK and SKS.

Exercises

1. Define B from Equation (4.2) as a tree.
2. Define C from Equation (4.2) as a tree.
3. Conjunction is given by
$$\mathsf{conj} = BB(KI) \ .$$
Check the truth table for conjunction.
4. The logical equivalence iff is given by
$$\mathsf{iff} = BS(B(BI(KI))K) \ .$$
Check its truth table.

5. Prove that bracket abstraction and star abstraction have the expected properties.
6. Compute the SK-combinator of $[f][x][y]fyx$ that corresponds to B in Equation (4.2). Do the same for C in Equation (4.2).
7. Confirm that getTag0 is a separator for tag$\{K, I\}$ and tag$\{KI, I\}$ in tree calculus.
8. Define a separator of SKK and SKS in tree calculus.
9. Verify Lemma 8.3.
10. Verify Lemma 8.4
11. Show that the translation from Roman numerals to unary arithmetic in Section 2.9 is meaningful.

Chapter 9
Lambda-Abstraction in *VA*-Calculus



9.1 First-Class Abstraction

We have seen the convenience of being able to convert a rule

$$x \mapsto M$$

into an abstraction with the property that applying it to some N results in the substitution of N for x in M. The fundamental idea of λ-calculus is to explore the power of abstraction as the primary means of term formation, as a first-class term construction, and not merely as a syntactic convenience, in the sense that abstraction preserves computational equality as well as syntactic equality. The various rules for operators are replaced by a single rule, the *β-rule*

$$(\lambda x.M)N = \{N/x\}M \ . \qquad (\beta)$$

which in turn requires rules for performing the substitution of N for x in M. Traditionally, the substitution rules are *not* reduction rules of the λ-calculus, but part of its *meta-theory* which must consider variables and substitutions, as well as terms. When

M is some combination of variables then substitution is straightforward but substitution into abstractions is not. This chapter will produce a new variant of λ-calculus called *VA-calculus* built using operators V and A for variables and abstractions, whose evaluation is given by reduction rules that do not require any meta-theory, much as in tree calculus. In particular, it follows that its abstractions are first-class.

Although *VA*-calculus meets the criterion for being a λ-calculus, in that it supports the abstractions that define the combinators, there cannot be meaningful translations between traditional λ-calculi and *VA*-calculus because they must take different approaches to substitution into redexes. When substitution is meta-theoretic, it must apply to any term, which implies

$$\{N/x\}(MP) = \{N/x\}M\ \{N/x\}P$$

even if MP is a redex. However, if substitution is part of the theory proper then such a reduction would overlap with other rules in a way that precludes any meaningful translation. In *VA*-calculus, substitution rules never apply to redexes, but to partially applied operators only. One might object that this change is too radical, but the precise mechanics of substitution is not part of the general understanding of abstraction, not part of its specification, so *VA*-calculus qualifies as a λ-calculus.

When considering the difficulties of λ-calculus, substitution into applications is not usually a concern. Rather, it is *scope management* which gets all of the attention, especially as it does not seem to support a combinatory representation. Significant difficulties arise when substituting into an abstraction, as in

$$(\lambda x.\lambda y.M)N = \{N/x\}(\lambda y.M)\ .$$

This requires a proper understanding of scope. In particular, what is the meaning of $\{N/x\}(\lambda x.M)$ when y is x? Again, what is the meaning of $\{y/x\}(\lambda y.M)$ when M is y (or contains y)? If this was not immediately obvious to you then you are in good company; the substitution rules are not obvious, are not axiomatic.

The traditional solution to such a name clash in $\{N/x\}(\lambda y.M)$ is to rename y by some variable z that is *fresh*, i.e. not already in use. For example, before substituting N for x in $\lambda x.x$ we should rename the bound name x by some fresh name z that does not appear in N so that we substitute of N for x in $\lambda z.z$ to get $\lambda z.z$. This solution is a little cumbersome: as well as adding equations for the renaming, there is the challenge of finding fresh variables in a systematic way.

Nicolaas Govert de Bruijn observed that the variables can be represented by indices $0, 1, 2, \ldots$ now called *de Bruijn indices*, where the index represents (one less than) the number of abstractions required to create a binding. For example, the term K given by $\lambda x.\lambda y.x$ becomes $\lambda(\lambda 1)$ since the x was bound by the second enclosing abstraction. When applied to K itself then β-reduction replaces 0 by K in $\lambda 1$. Then substitution under the λ causes 1 to be replaced by K in the body, to produce λK. Things are more complicated when the argument is a variable. For example, when K is applied to 3 then substitution of 3 for 0 in $\lambda 1$ replaces 1 by 4 to get $\lambda 4$. Of course, all this lifting and lowering of induces uses meta-theory, which we are trying to avoid.

In *VA*-calculus, substitution does not act on the body of an abstraction. Instead, abstractions carry an *environment* in which variable bindings can be recorded. The abstraction

$$AME$$

has body M and environment E. The reduction rules for A are guided by the intuition that reduction of $AMEN$ applies the "substitution" A_EN to M by analysing the structure of M. For example, if M is the zero index V then $AVEN = N$ or if M is a successor index Vk then $A(Vk)EN = Ek$ where the environment E is here thought of as a substitution. Most interesting is when M is itself an abstraction, in which case we have

$$A(APQ)EN = AP(AQEN).$$

That is, substitution is applied to the environment Q but not the body P of the abstraction APQ. In this manner, the various meta-rules for substitution are expressed as reduction rules for abstraction. Of course, there remains the challenge of creating suitable environments, of creating combinations that represent substitutions. This is achieved by adding the rule

$$A(AE)NM = AMEN$$

so that A_EN is represented by $A(AE)N$. When the rule is read from right to left, it expresses the β-rule, that applications of abstractions yield substitutions. However, as a reductiion rule, it will be read from left to right, so that substitutions reduce to abstractions.

Environments first appeared in the *explicit substitution calculus* of Martine Abadi, Luca Cardelli, Pierre-Louis Curien and Jean-Jacques Lèvy but they were not introduced to solve this problem, so that substitutions have continued to act on the bodies of abstractions, with no reduction in the amount of meta-theory. Also, when a substitution σ acts on environment that is a substitution ρ it is usual to compose them, as functions from variables to terms. In particular, if the environment ρ is empty then their composition is σ. Here, however, the application of a substitution σ to an empty environment yields an empty environment. While composition is more intuitive than application, it usually creates overlapping reduction rules which is unpleasant. This is doubly annoying when σ has no effect on M anyway. In practice, this constraint is often enforced by the use of *closures*. However, checking this property during computation would be awkward, and is not strictly necessary, so will not be enforced within the calculus itself.

Like *SK*-calculus, *VA*-calculus is combinatorially complete without being complete. That is, it supports all combinators and yet does not support equality of programs.

Also, there is a meaningful translation $[-]$ from *VA*-calculus to tree calculus. For example, it maps A to A_t which is pictured at the beginning of Part II. It is easy enough to define A_t to capture the desired behaviour when it is applied to three arguments. The challenge is to be able to substitute into its applications to zero, one

or two arguments. The solution is to use tags in tree calculus to record the information about the arguments of A_t.

Interestingly, even though *VA*-calculus is not complete, it does support tagging, since environments are able to store tags, which can be recovered by manipulating scopes in a non-standard manner. In turn, this can be used to define a translation from tree calculus to *VA*-calculus.

This shows that meaningful translation need not preserve completeness properties. In particular, the translations cannot be used to decide equality of programs. Even though equality of programs is not definable in *VA*-calculus, it is tempting to adopt the invalid argument in Figure 9.1.

The equality of programs in *VA*-calculus can be decided by:
translating the programs to tree calculus;
deciding equality of the translations using equal; and
translating the result back to *VA*-calculus.

Fig. 9.1 An Invalid Argument

The paradox is resolved as follows. Using translations $[-]$, the equality of programs p and q is specified by the computation of

$$[\mathsf{equal}]\ [[p]]\ [[q]]\ .$$

If the double translation from *VA*-calculus to tree calculus and back were definable in *VA*-calculus then this specification could be realised in *VA*-calculus. Conversely, since equality cannot be defined in *VA*-calculus, it follows that the double translation is *not* definable in *VA*-calculus. The double translation is performing computations which are beyond the expressive power of *VA*-calculus.

9.2 *VA*-Calculus

The term forms of *VA*-calculus are given by

$$M, N := V \mid A \mid MN\ .$$

Both V and A are ternary operators. V is used to build indices, with the kth index given by $V^k(V)$. We may write k for $V^k(V)$ when it is clear that k represents an index. A term of the form Vxy is a *suspended application*, which may also be written as $x@y$. They convert to ordinary applications under substitution. Some care is required here since applications of variables must be suspended. For example, to apply the first index VV to some argument x do not use VVx as this is the suspended application of $V@x$. Rather, use the suspended application $V(VV)x$ or $VV@x$. The *abstraction* Axy has *body* x and *environment* y. Note that x is *not* the index, but the

body. The operator A can be used as the empty environment, as it is stable under substitution. The partial application Ax will be used to support substitutions of the form $A(Ax)$.

As is clear from the descriptions above, the two operators are performing multiple roles. If preferred, one can introduce distinct operators for the zero index, for building successors, for suspended applications, for the empty environment and for substitutions. This would make the calculus easier to understand but lead to a dramatic increase in the number of reduction rules, as substitution requires a rule for each partially applied operator.

The reduction rules of VA-calculus are given in Figure 9.2. They can be understood as follows. An application of a delayed application Vxy to some z becomes a delayed application to z. Applications of abstractions substitute into the body of the abstraction, as determined by the form of the body.

$$\begin{aligned}
Vxyz &\longrightarrow V(Vxy)z \\
AVyz &\longrightarrow z \\
A(Vx)yz &\longrightarrow yx \\
A(Vwx)yz &\longrightarrow Awyz(Axyz) \\
AAyz &\longrightarrow A \\
A(Ax)yz &\longrightarrow Azxy \\
A(Awx)yz &\longrightarrow Aw(Axyz)
\end{aligned}$$

Fig. 9.2 Evaluation Rules of VA-Calculus

Theorem 9.1 (confluence_va_calculus) *VA-calculus is confluent.*

Proof The proof uses simultaneous reduction, just as for SK-calculus. None of the reduction rules overlap. □

9.3 Combinators

Bracket abstraction and star abstraction are defined by adapting techniques from tree calculus. It may seem strange to do this within a λ-calculus but it will prove convenient when the goal is to substitute into the body of an abstraction.

The identity combination is defined by

$$I = AVA$$

since $Ix = AVAx = x$. Also, we can define *bracket abstraction* by

$$
\begin{aligned}
[x]_b x &= V \\
[x]_b y &= Vy \quad (y \neq x) \\
[x]_b V &= VV \\
[x]_b A &= VA \\
[x]_b st &= ([x]_b s)@([x]_b t)
\end{aligned}
$$

$$[x]t = A([x]_b t)I \ .$$

Theorem 9.2 (bracket_beta) *For all terms t and u we have* $([x]t)u \longrightarrow \{u/x\}t$.

Proof We have $([x]t)u = A([x]_b t)Iu$ and now proceed by induction on the structure of t. □

As in tree calculus, bracket abstraction can be used to support waiting and fixpoint functions in normal form. Again, we can optimise bracket abstraction to produce *star abstraction* defined by

$$
\begin{aligned}
\lambda^*_b x.x &= V \\
\lambda^*_b x.y &= Vy \quad (y \neq x) \\
\lambda^*_b x.V &= VV \\
\lambda^*_b x.A &= VA \\
\lambda^*_b x.st &= V\ (s\ t) \quad (\text{if } x \notin s\ t) \\
\lambda^*_b x.st &= (\lambda^*_b x.s)@(\lambda^*_b x.t) \quad (\text{if } x \in s\ t)
\end{aligned}
$$

$$\lambda^* x.t = A(\lambda^*_b x.t)I \ .$$

Theorem 9.3 (star_beta) *For all terms t and u we have* $(\lambda^* x.t)u \longrightarrow \{u/x\}t$.

Proof Apply induction on the structure of t. □

Hence, we may define K and S in the usual manner by

$$
\begin{aligned}
K &= \lambda^* x.\lambda^* y.x \\
S &= \lambda^* x.\lambda^* y.\lambda^* z.xz(yz) \ .
\end{aligned}
$$

Theorem 9.4 (meaningful_translation_from_sk_to_va) *There is a meaningful translation from SK-calculus to VA-calculus.*

Proof With S and K defined as above, it remains to check that their partial applications to programs have normal forms. □

9.4 First-Class Substitutions

It follows that *VA*-calculus supports all the extensional functions introduced in Chapter 4. However, it often happens that there are more direct accounts that use non-empty environments to represent substitutions.

Given terms e and u define

$$[u, e] = A(V(V(VAV)(Ve))(Vu))I \ .$$

This is the same as $\lambda^* x.Axeu$ if x does not occur in e or u. It follows that

$$[u, e]t = Ateu$$

which reduces to u if t is V and to ek if t is Vk. That is $[u, e]$ imaps the index 0 to u and applies e to successors. When this construction is nested, we may suppress internal brackets, so that $[u_0, u_1, \ldots, u_k, e]$ denotes $[u_0, [u_1, [\ldots, [u_k, e] \ldots]]]$. Commonly, the environments of abstractions will take the form $[u_0, u_1, \ldots, u_k, I]$.

For example, *waiting* can be achieved by using an environment to delay application, in

$$\mathsf{wait}\{x, y\} = A(21@0)[y, x, I] \ .$$

Clearly, this term is a program if x and y are, while application to some z yields xyz.

Conversely, if a λ-calculus does not support waiting then there is no reason to expect a meaningful translation to it from SK-calculus. For example, if the SK-program

$$\mathsf{wait}\{SII, SII\}$$

has the same translation as $(SII)(SII)$ then the translation of the latter must be valuable, which is would be a surprise.

9.5 Incompleteness

VA-calculus is incomplete for similar reasons to SK-calculus.

Theorem 9.5 (equality_of_programs_is_not_definable_in_va) *The equality of normal forms is not definable in VA-calculus.*

Proof The proof is just as for SK-calculus but with the identity programs I and AVI instead of SKK and SKS. That is, one must define the identity-reduction relation, prove the pentagon lemmas, and then show that the existence of an equality term implies that distinct variables are equal. Details are in the formal proofs. □

9.6 Translation to Tree Calculus

Let us consider how to translate λ-calculus to tree calculus. In order to capture the intensionality of substitution into, say, an abstraction Azw it is necessary to be able to recover the translations of A and z and w from that of their combination, and this without interfering with the functionality of the abstraction. The solution is to tag the functional translation of the abstraction with the information required for substitution.

That is, A must be translated to a tree A_t such that $A_t xyz$ applies the tag of x to A_t and y and z. The delicate case arises when x is A_t itself since A_t will be a fixpoint function (and not a tagged term). Applying getTag to some $Y_2\{f\}$ reduces to I so the desired behaviour is achieved by ensuring that

$$A_t xyz = \mathsf{getTag}\ x\ (K(KA_t))\ y\ z$$

Further, the partial applications of A_t must be tagged appropriately. After some experimentation, the desired term is given by

$$\begin{aligned}
\mathsf{lamtag1} &= \lambda^* a.\lambda^* x.K(\lambda^* y.\lambda^* z.azxy)\\
\mathsf{lamtag2} &= \lambda^* a.\lambda^* x.\lambda^* y.K(\lambda^* y_2.\lambda^* z.\mathsf{wait}\{a,x\}\ (\mathsf{wait}\{a,y\}y_2 z))\\
\mathsf{useTag} &= \lambda^* a.\lambda^* x.\lambda^* y.\triangle(\triangle x\triangle K\triangle)\triangle K(K(Ka))y\\
\mathsf{lamtagged2} &= \lambda^* a.\lambda^* x.\lambda^* y.\mathsf{tag}\{\mathsf{lamtag2}\ a\ x\ y, \mathsf{useTag}\ a\ x\ y\}\\
\mathsf{lamtagged1} &= \lambda^* a.\lambda^* x.\mathsf{tag}\{\mathsf{lamtag1}\ a\ x, \mathsf{lamtagged2}\ a\ x\}\\
A_t &= Y_2\{\mathsf{lamtagged1}\}.
\end{aligned}$$

Note how waiting is used to ensure that partial applications of A_t can be normalised. A schematic representation of A_t is given in Figure 9.3. The picture of A_t is given by the tree for λ at the beginning of Part II.

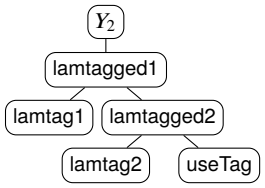


Fig. 9.3 A Schematic Translation of A

Similarly, define

$$
\begin{aligned}
V_t = \mathsf{tag}\{&K(KI),\\
&Y_2\{\lambda^* v.\lambda^* x.\mathsf{tag}\{K(\lambda^* e.\triangle(\triangle(Kx))(Ke)),\\
&\qquad \lambda^* y.\mathsf{tag}\{\lambda^* k.\lambda^* e.\triangle(\triangle(k\triangle\triangle ye))(k\triangle\triangle xe),\\
&\qquad\quad \triangle(\triangle I)(\triangle(\triangle(\triangle(\triangle Ky))(\triangle(\triangle(Kx))(K(\mathsf{tag}\{KK,f\}))))\\
&\qquad\qquad (K\ \mathsf{tag}\{K(KI),f\}))\}\}\ .
\end{aligned}
$$

The picture of V_t is given in Figure 9.4.

Theorem 9.6 (meaningful_translation_va_to_tree) *There is a meaningful translation from VA-calculus to tree calculus.*

Proof The translation is given above. The proof requires many lemmas, which can be found in the Coq implementation. □

9.7 Tagging

Even though *VA*-calculus is incomplete, it is still able to support tagging, which is a key aspect of intensional computation. Define

$$
\begin{aligned}
\mathsf{tag}\{t,f\} &= A(V(V(V21)(V(V^3(f))V))(V^3(t)))\ (A(A[[K,I],V,I]))\\
\mathsf{getTag} &= A(V(V(V(V(VA)V)(VI))(VKI))(VA))I.
\end{aligned}
$$

so that

$$
\begin{aligned}
\mathsf{tag}\{t,f\}\ x &= [K,I]0(fx)t = K(fx)t = fx\\
\mathsf{getTag}\ \mathsf{tag}\{t,f\} &= [K,I](KI)(fx)t = KI(fx)t = t
\end{aligned}
$$

as required.

Hidden in the blur of symbols is the following idea. By controlling the order in which substitutions are applied, the meaning of $2@1$ in the definition of $\mathsf{tag}\{t,f\}$ can become either K or KI to yield either the application of f or the term t. In turn, this is possible because of the flexibility of the term formation rules: substitutions are first-class terms that can be used to define other substitutions, and V has been applied to f and t without requiring them to be indices.

9.8 Translation from Tree Calculus

As with the translation of *VA*-calculus to tree calculus, tagging can be used to support a meaningful translation from tree calculus to *VA*-calculus, as follows. The action of $\triangle$ on a leaf is represented by K since $Kyz \longrightarrow y$. The actions of $\triangle$ on stems and forks requires combinations $\mathsf{stem}\{x\}$ and $\mathsf{fork}\{x,y\}$ such that

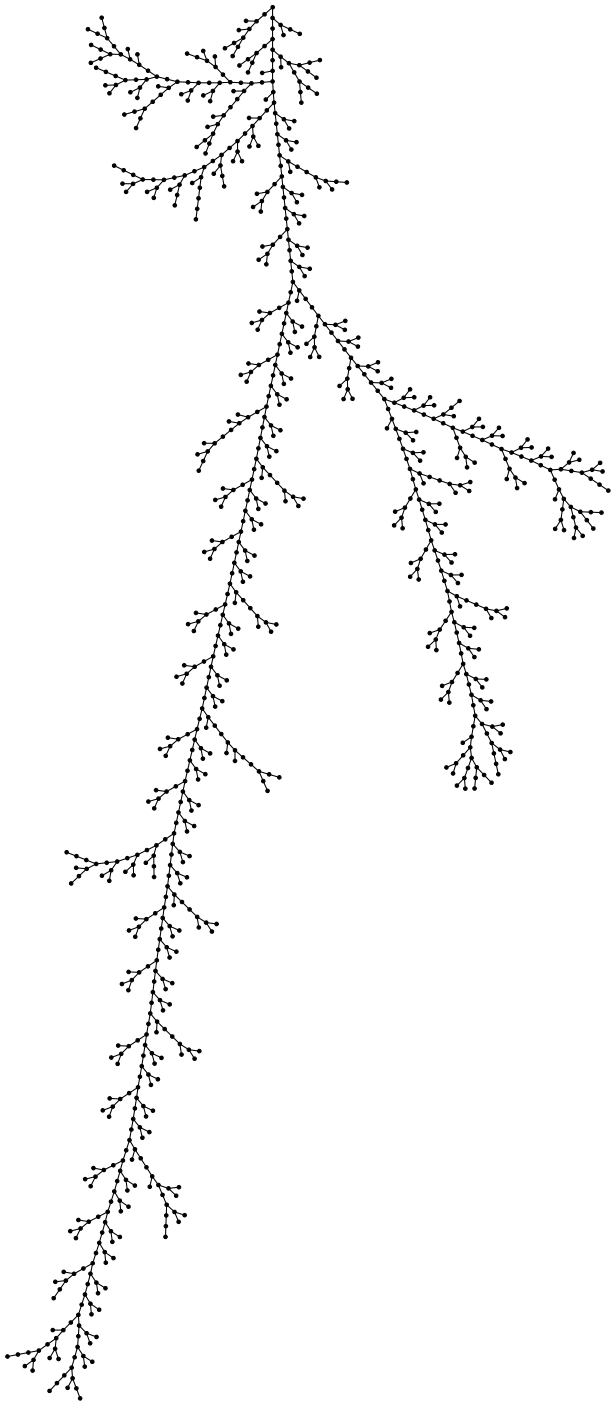


Fig. 9.4 The Translation of V to Tree Calculus

$$\begin{aligned}\mathsf{stem}\{x\}\ y\ z &\longrightarrow yz(xz)\\ \mathsf{fork}\{w,x\}\ y\ z &\longrightarrow zwx\ .\end{aligned}$$

These are given by star abstractions in the standard manner. Finally, we require an interpretation ker of the node, that satisfies

$$\begin{aligned}\mathsf{ker}\ x\ y\ z &= \mathsf{getTag}\ x\ y\ z\\ \mathsf{getTag}\ \mathsf{ker} &= K\\ \mathsf{getTag}\ (\mathsf{ker}\ x) &= \mathsf{stem}\{x\}\\ \mathsf{getTag}\ (\mathsf{ker}\ w\ x) &= \mathsf{fork}\{w,x\}\ .\end{aligned}$$

It is given by

$$\begin{aligned}&\mathsf{ker} = \mathsf{tag}\{K,\\ &\qquad \lambda^* x.\mathsf{tag}\{\mathsf{stem}\{x\},\\ &\qquad\qquad \lambda^* y.\mathsf{tag}\{\mathsf{fork}\{x,y\},\\ &\qquad\qquad\qquad (A(V(V(V(V\ \mathsf{getTag})(Vx))(Vy))V)I)\}\}\}\end{aligned}$$

and is pictured in Figure 9.5.

The *values* or *programs* of *VA*-calculus are given by the applications of V or A to zero, one or two programs. Once again, the *valuable* terms are those that reduce to values.

Theorem 9.7 (meaningful_translation_from_tree_to_va) *There is a meaningful translation from tree calculus to VA-calculus.*

Proof The proof requires many lemmas. See the Coq proofs for details. □

Corollary 9.1 *There is* no *meaningful translation from VA-calculus to combinatory logic.*

Proof If there were such a translation then it would compose with the translation above to produce a meaningful translation from tree calculus to combinatory logic, which is impossible by Theorem 8.5. □

Exercises

1. (Hard) List the key properties that you think that a lambda-calculus should satisfy, such as being Turing complete, being combinatorially complete, supporting the interpretation of functions as rules, supporting equational reasoning, etc. Try to avoid technical properties that are internal to the calculus. Determine if *VA*-calculus has these properties. How does *VA*-calculus compare to your favourite lambda-calculus?

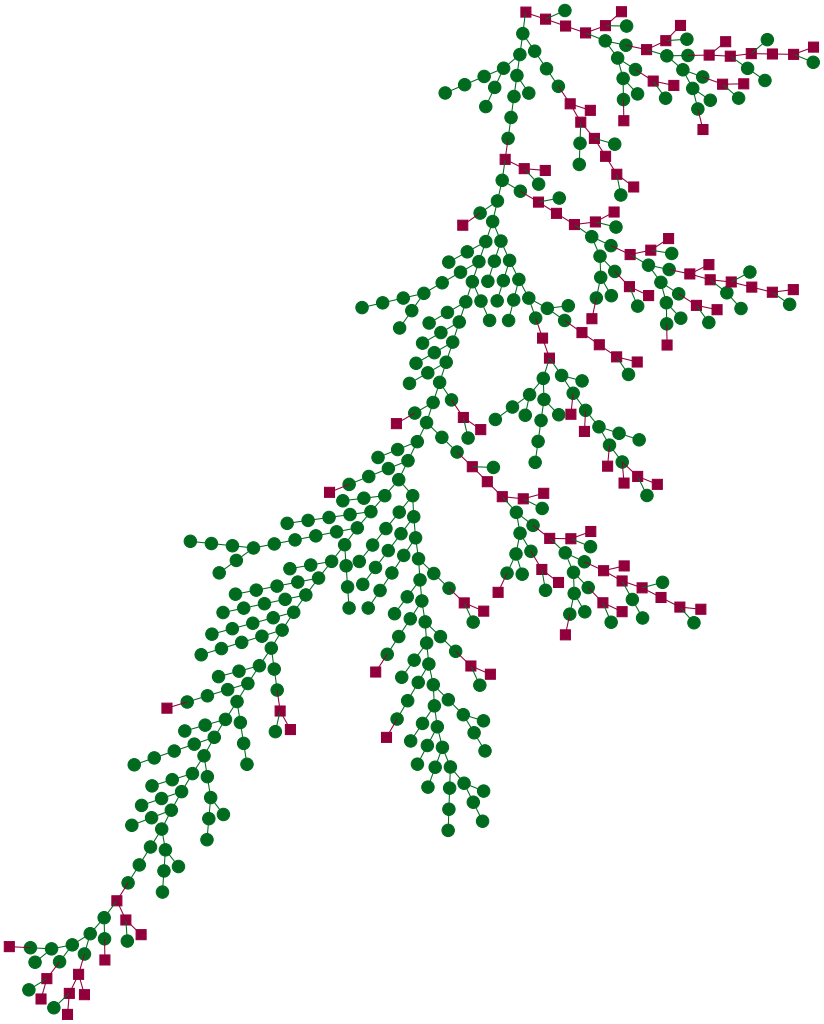
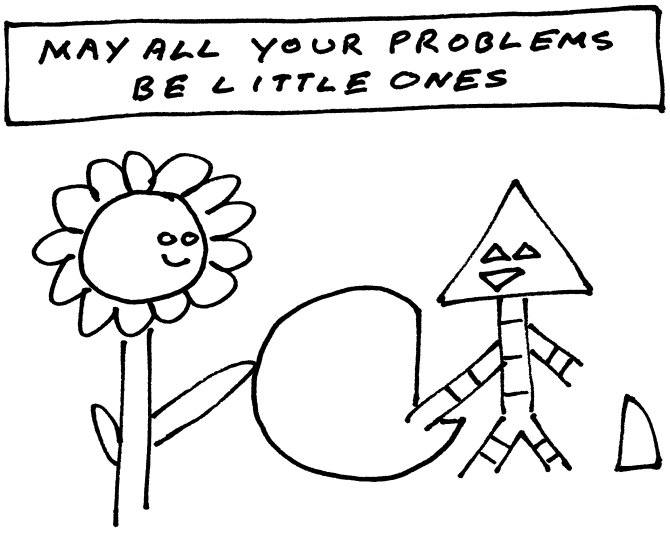


Fig. 9.5 The Translation of $\triangle$ to *VA*-Calculus: Circles are *V* and Squares are *A*

2. (Hard) Is the support for tagging in *VA*-calculus a bug or a feature? On the one hand, it exploits the dynamic approach to scoping which conflicts with traditional views. On the other hand, it is crucial to the translation from tree calculus.
3. (Hard) *VA*-calculus sits between combinatory logic and tree calculus: unlike combinatory logic it is intensional, to the extent that is implied by tagging; unlike tree calculus, it doe not support reflection as it does not support pattern-matching functions or even program equality. Is this intermediate level of expressiveness interesting in practice?

4. (Hard) Many powerful type systems use abstraction with respect to type variables to support polymorphism. So the *VA*-calculus can support such types, much as in Chapter 5.5. However, type inference requires the ability to decide type equality, which will not be possible in *VA*-calculus. How might tree calculus support polymorphic types?

Chapter 10
Divide-and-Conquer in SF-Calculus



10.1 Divide and Conquer

Data structures differ from functions in that the emphasis is on storing information rather than using it. This stored information is recovered by queries. More generally, we search data structures, or combine the data to produce bulk properties of the data structure. No matter the nature of the structure, these processes can be thought of as applications of a divide-and-conquer strategy, in which the structure is recursively divided into smaller and smaller pieces which are eventually conquered.

Although this is achievable in tree calculus, it is not especially convenient. When evaluating $\triangle xyz$, the reduction rules of tree calculus use the two arguments y and z to handle the three possibilities for x, as a leaf, stem or fork, so that y and z must perform double duty: y is used for leaves, z is used for forks, and both are used for stems. More to the point, this approach does not generalise to handle other constructions that use four or more arguments.

This chapter develops a cleaner approach, based on factorisation by an operator F. When evaluating $Fxyz$ there are only two possibilities: if x is an atom (i.e. an operator) then use y else if x is a compound (i.e. a partially applied operator) then

use z. That is, F supports divide-and-conquer strategies, in which compounds are divided and atoms are conquered. In a sense, the reduction rules for F correspond to the leaf and fork rules of tree calculus. However, reduction of F does not allow for duplication of any kind, as supported by the stem rule. This can be recovered by admitting the traditional S as a second operator, to get SF-calculus, developed with Thomas Given-Wilson. It turns out that SF-calculus is equivalent to tree calculus in expressive power. In particular, it supports equality of programs and tagging and pattern matching.

10.2 SF-Calculus

The *combinations* of SF-calculus are built using the operators S and F which are the *atoms*. The *compounds* are the applications of S and F to one or two arguments only, which can be thought of as their *partial applications*. Conceptually, there are two reduction rules for F, namely

$$\begin{aligned} Fxyz &\longrightarrow y \qquad \text{(if } x \text{ is an atom)} \\ F(wx)yz &\longrightarrow zwx \qquad \text{(if } wx \text{ is a compound).} \end{aligned}$$

In a more general setting, the atoms may include other operators or constructors for data types. Then the *compounds* are all the applications of atoms that are not redexes. Since S and F are both going to be ternary operators we can *define* the compounds to be combinations of the form Sx, Sxy, Fx and Fxy. Then the *reduction rules* of SF-calculus are formalised by expanding out the side-conditions to the rules above to get

$$\begin{aligned} Sxyz &\longrightarrow xz(yz) \\ FSyz &\longrightarrow y \\ FFyz &\longrightarrow y \\ F(Sx)yz &\longrightarrow zSx \\ F(Swx)yz &\longrightarrow z(Sw)x \\ F(Fx)yz &\longrightarrow zFx \\ F(Fwx)yz &\longrightarrow z(Fw)x \ . \end{aligned}$$

The *programs* of SF-calculus are its irreducible terms, which may be thought of as binary trees whose nodes are labelled by S or F.

Theorem 10.1 *Reduction of SF-calculus is confluent.*

Proof The proof uses the same techniques as that for tree calculus in Theorem 7.8, by showing that simultaneous reduction satisfies the diamond property. □

We can define K by FF since

$$Kyz = FFyz = y$$

as before. That is, SK-calculus is embedded within SF-calculus, so that all of the machinery of bracket abstraction, star abstraction, fixpoints, arithmetic, etc. carries over with almost no change.

Further, we can test for being a compound by the function

$$\mathsf{isComp} = \lambda^* x.Fx(KI)(K^3 I)$$

and define the component functions left and right by

$$\mathsf{left} = \lambda^* x.FxKK$$
$$\mathsf{right} = \lambda^* x.FxK(KI)$$

since if wx is a compound then

$$\mathsf{left}\ (wx) = F(wx)KK = Kwx = w$$
$$\mathsf{right}\ (wx) = F(wx)K(KI) = KIwx = x\ .$$

As these examples show, the first argument of F is often the last to be known, so that it was tempting to change the order of the arguments of F. However, this arrangement makes it easy to define K. Again, we could easily replace the operator S with D to obtain the DF-calculus, but let that pass.

As to the converse question, whether F can be defined in terms of S and K, the answer is no, for much the same reasons that one cannot define $\triangle$ in terms of S and K. For example, combinators cannot distinguish SKK from $SK(KI)$ as both represent the identity function, even though they are easily separated by right.

When factorisation is applied recursively, using fixpoints, it supports a general approach to divide-and-conquer algorithms: if the argument is a compound then divide it and recurse; if the argument is an operator then conquer it.

10.3 Intensional Examples

Here are some examples of intensional programs that have been adapted from Chapter 5. A simple example of this is the program size which determines the size of a normal form. It is given by

$$\mathsf{size} = Y_3(\lambda^* f.\lambda^* x.Fx1(\lambda^* y.\lambda^* z.(fy) + (fz)))\ .$$

Thus, the size of an atom is 1 and the size of a compound MN is the sum of the sizes of M and N.

To define the equality of programs requires a means of deciding if two operators are equal. Define isF by

$$\mathsf{isF} = \lambda^* x.x(KI)(K(KI))K\ .$$

It maps F to K and S to KI as desired since

$$\begin{aligned}\mathsf{isF}F &= F(KI)(K(KI))K = KKI = K\\ \mathsf{isF}S &= S(KI)(K(KI))K = KIK(K(KI)K) = I(KI) = KI\ .\end{aligned}$$

From this, we can define equality of operators by

$$\mathsf{eqatom} = \lambda^* x.\lambda^* y.\mathsf{iff}\ (\mathsf{isF}\ x)\ (\mathsf{isF}\ y)\ .$$

Structural equality of programs can now be given by

$$\begin{aligned}\mathsf{equal} = Y_3(\lambda^* e.\lambda^* x.\lambda^* y.&\\ Fx(Fy(\mathsf{eqatom}\ x\ y)(K^3 I))&\\ (\lambda^* x_1.\lambda^* x_2.Fy(KI)(\lambda^* y_1.\lambda^* y_2.&\\ \mathsf{and}\ (e\ x_1\ y_1)(e\ x_2\ y_2))))\ .&\end{aligned}$$

Theorem 10.2 (equal_sf_programs) equal $p\ p \longrightarrow K$ *for all programs* p .

Proof The proof is by induction on the structure of programs. □

Theorem 10.3 (unequal_sf_programs) equal $p\ q \longrightarrow KI$ *for all programs* p *and* q *that are not equal.*

Proof The proof is by induction on the structure of programs. □

Tagging is defined by

$$\mathsf{tag}\{t, f\} = S(S(KK)f)t\ .$$

Application removes the tag since

$$\begin{aligned}\mathsf{tag}\{t, f\}\ x &= S(KK)fx(tx)\\ &= KKx(fx)(tx)\\ &= fx\ .\end{aligned}$$

Also, the tag can be recovered by defining getTag to be right.

Pattern matching in SF-calculus is similar to that of tree calculus. The patterns are built from variables, operators and application. The corresponding *extensions* are defined as follows:

$$\begin{aligned}x \Rightarrow s \mid r &= \lambda^* x.s\\ S \Rightarrow s \mid r &= S(S\ \mathsf{isS}\ (Ks))r\\ F \Rightarrow s \mid r &= S(S\ \mathsf{isF}\ (Ks))r\\ p\ q \Rightarrow s \mid r &= S(SFr)(K(p \Rightarrow (q \Rightarrow s \mid (S(Kr)p)) \mid (S(Kr))))\end{aligned}$$

10.4 Translation from Tree Calculus

Since triage and factorisation are so similar, it will be no surprise that there are meaningful translations between tree calculus and SF-calculus. However, the translations are not trivial because leaves are not translated to atoms and F is not translated to a leaf. Here are some details.

In tree calculus, the atoms are the leaves and the compounds are the stems and forks. So factorisation is there given by

$$F_0 = \lambda^* x.\lambda^* y.\lambda^* z.\mathsf{triage}\{y, z\triangle, \triangle K(Kz)\}x \ .$$

For example, when applied to some fork $\triangle x_1 x_2$ and to y and z it reduces to

$$\triangle K(Kz)x_1x_2 = Kzx_1(\triangle x_1)x_2 = z(\triangle x_1)x_2$$

as desired. Unfortunately, we cannot translate the operator F of SF-calculus by F_0 since F_0 is a fork so F_0F_0yz reduces to an application of z.

Rather, the key to this, and the translations to follow, is to use tags to record the intensional information required for the reduction rules. Here, the translation of $\triangle$ applied to x, y and z will reduce to

$$\mathsf{getTag}\ x\ y\ z \ .$$

In turn, the required tags are revealed by the following reductions

$$\begin{aligned} Kyz &\longrightarrow y \\ SS(Kx)yz &\longrightarrow Sy(Kxy)z \longrightarrow yz(xz) \\ K(S(SI(Kw))(Kx))yz &\longrightarrow SI(Kw)z(Kxz) \longrightarrow zwx \ . \end{aligned}$$

Thus, the translation of $\triangle$ is given by adding these tags to getTag to produce

$$\begin{aligned} \triangle_f = \mathsf{tag}\{K,& \\ &\lambda^* x.\mathsf{tag}\{SS(Kx), \\ &\qquad \lambda^* y.\mathsf{tag}\{K(S(SIKx)(Ky)), \\ &\qquad\qquad \mathsf{getTag}\ x\ y\ z \\ &\}\}\} \ . \end{aligned}$$

The picture of $\triangle_f$ is in Figure 10.1

Theorem 10.4 (meaningful_translation_from_tree_to_sf)

There is a meaningful translation from tree calculus to SF-calculus.

Proof Translate $\triangle$ by $\triangle_f$. Details of the proof is routine. □

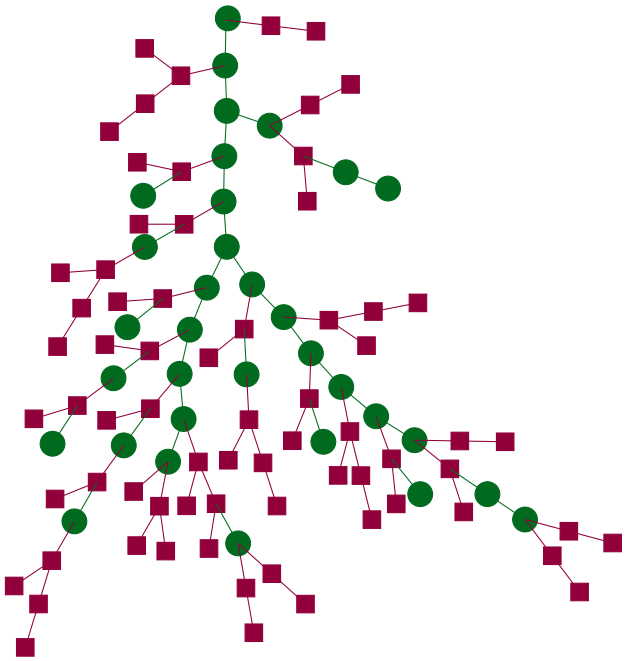


Fig. 10.1 The Translation of $\triangle$ to SF-Calculus: Circles are S and Squares are F

10.5 Translation to Tree Calculus

When translating SF-calculus to tree calculus, tags are used once again to keep track of intensional information. The tags that will be used to represent F are revealed by the computations

$$\begin{aligned} Kyz &\longrightarrow y \\ \triangle(\triangle Ox)yz &\longrightarrow zOx \\ K\ \mathsf{mythree}\{O,w,x\}\ y\ z &\longrightarrow z(Ow)x\ . \end{aligned}$$

where O is an operator, provided that $\mathsf{mythree}\{O,w,x\}\ z \longrightarrow z(Ow)x$. The obvious solution for $\mathsf{mythree}$ is $\lambda^* z.zOwx$ but this is not a program because Ow will reduce. So add some waiting to obtain

$$\mathsf{mythree}\{f,x,y\} = \triangle(\triangle(\triangle(\triangle I)(\triangle(\triangle(Kf))(Kx))))(Ky)\ .$$

The other point to note is that the translations of the operators O reference themselves in the tags, so that recursion is required. Putting this all together, we can translate F by

$$
\begin{array}{l}
\mathsf{tag}\{K,\\
\quad Y_2\{\lambda^* f.\lambda^* x.\\
\quad\quad \mathsf{tag}\{\triangle(\triangle\ \mathsf{tag}\{K,f\}\ x),\\
\quad\quad\quad \lambda^* y.\mathsf{tag}\{K(\mathsf{mythree}\{\mathsf{tag}\{K,f\},x,y\}),\\
\quad\quad\quad\quad \mathsf{getTag}\ x\ y\\
\quad \}\}\}\}\ .
\end{array}
$$

Further, S can be translated by modifying this program to use the combination S of tree calculus in place of getTag. So define

$$
\begin{array}{l}
\mathsf{ternary\_op}\{f\} = \mathsf{tag}\{K,\\
\quad\quad Y_2\{\lambda^* f.\lambda^* x.\\
\quad\quad\quad \mathsf{tag}\{\triangle(\triangle\ \mathsf{tag}\{K,f\}\ x),\\
\quad\quad\quad\quad \lambda^* y.\mathsf{tag}\{K(\mathsf{mythree}\{\mathsf{tag}\{K,f\},x,y\}),\\
\quad\quad\quad\quad\quad fxy\\
\quad\quad \}\}\}\}
\end{array}
$$

and then we have

$$
\begin{aligned}
S_t &= \mathsf{ternary\_op}\{S\}\\
F_t &= \mathsf{ternary\_op}\{\mathsf{getTag}\}\ .
\end{aligned}
$$

For any combination f and terms x, y and z we have

$$\mathsf{ternary\_op}\{f\}xyz \longrightarrow fxyz$$

so that

$$
\begin{aligned}
S_t xyz &\longrightarrow Sxyz \longrightarrow xz(yz)\\
F_t xyz &\longrightarrow \mathsf{getTag}\ x\ y\ z
\end{aligned}
$$

as intended.

The pictures of $\mathsf{ternary\_op}\{f\}$ is given in Figure 10.2 .

Theorem 10.5 (meaningful_translation_from_sf_to_tree) *There is a meaningful translation from SF-calculus to tree calculus.*

Proof The required translation maps S to S_t and F to F_t. The verification is routine.□

Exercises

1. Reduce the following combinations to normal form:

$$FFK(KI), FSK(KI), F(FF)K(KI), F(FS)K(KI), FKK(KI), FIK(KI)\ .$$

Take care to distinguish the operators from the compounds!

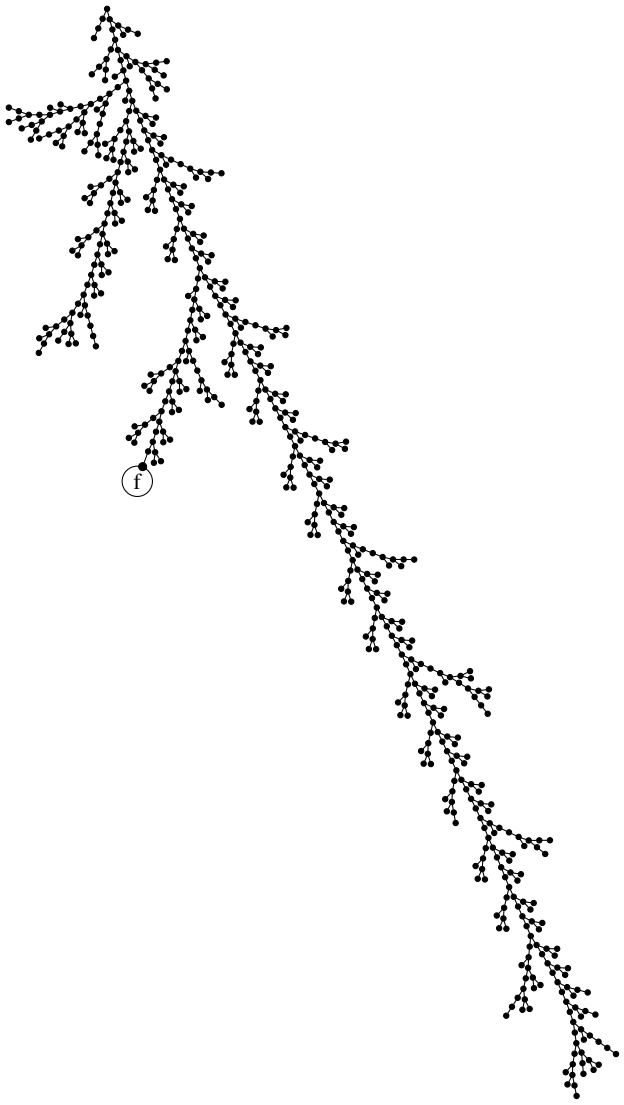


Fig. 10.2 ternary_op{f} is Used to Translate S and F to Tree Calculus

2. Show that if x is a compound then (left x)(right x) reduces to x for any compound x.
3. Define DF-calculus where $Dxyz \longrightarrow yz(xz)$. Can you define a meaningful translation from SF-calculus to DF-calculus? Hint: take care with the atoms.
4. Verify that if M is a normal form then Y_2M reduces to a normal form.
5. Verify the diamond property for simultaneous reduction in SF-calculus.
6. Check the properties of tagging in SF-calculus.

Chapter 11
Concluding Remarks

11.1 Recapitulation

Tree calculus is a better foundation for computation than any of the traditional models because of its simplicity, naturality, and expressive power. This may have implications for several disciplines.

Tree calculus is simple because it is built using combinations of a single operator, governed by three simple equations that do not require any side-conditions, or additional machinery of any kind. This compares well with traditional models. For example, Turing machines require a mechanism for converting the transition table into actions upon the tape. Again, lambda-calculus requires machinery for substitution that comes with side-conditions to maintain scoping. Combinatory logic is also equational, but not quite as simple, since it requires more than one operator.

Tree calculus is natural because it computes natural binary trees, whose nodes are unlabelled. More precisely, the functions, values, programs, inputs and outputs are all given by natural binary trees, while the computations are the finitely-branching natural trees. Again, this compares well with traditional models. Turing machines, lambda-abstractions and combinators are artificial. Although the μ-recursive func-

tions compute natural numbers, the functions themselves are artificial. Conversely, if the programs are defined to be the natural numbers then elaborate meta-theory is required to perform the computations. In tree calculus, every binary tree is simultaneously a data structure and a program, that can be the argument of a function or a function that takes an argument.

Tree calculus is powerful because it expresses more functions than the traditional models of computation. As well as the numerical computations expressed as μ-recursive functions, the combinators of combinatory logic, the abstractions of λ-calculus and the analyses of Turing machines, it can decide program equality self-evaluate. By contrast, traditional models must be augmented with meta-theory to supply the missing expressive power: the μ-recursive functions and the Turing machines maintain a formal separation of functions and arguments so that meta-theory is essential for reflection; combinatory logic and lambda-calculus are incomplete, since they cannot decide equality of values. All of these claims have been captured as theorems and verified using the Coq proof assistant.

The work is now done, except that there are some outstanding issues which deserve comment. They are raised here, rather than in the introduction or the body of the work, so that the status of the theorems is not mixed up with the opinions and speculations below.

11.2 Program Completeness

In what sense is tree calculus complete? Traditionally, a calculus was deemed complete if it was *Turing-complete*, in that it computed the same class of numerical functions as the Turing machines. The supposition that the Turing-computable numerical functions included all computable numerical functions is quite reasonable, but unproven. This became known as *Turing's Thesis* or the *Church-Turing Thesis*. We have seen that tree calculus is Turing-complete. Further, it supports a rich class of functions that act on programs, including decidable equality, pattern-matching functions, tagging and self-evaluators, that are not supported by the traditional models. It is not yet clear what the class of computable functions of programs should be, but here are some thoughts.

One option is to replicate the traditional approach but with binary natural trees instead of natural numbers. Now the tree calculus plays the role of the μ-recursive functions so that, by definition, tree calculus is tree-complete. However, all of the traditional models are complete in this sense, too. The problem is that their programs are not binary trees.

A second option is to require that the model supports an invertible function that converts programs to natural numbers. If the model is also Turing-complete then it supports a rich class of functions of programs. Tree calculus supports such a function but the traditional models of computation do not. The problem is that this privileges the natural numbers over the natural trees, even though the latter are proving to be more important.

The third option combines the previous two, to declare that a model is *program-complete* if it supports the same class of functions of binary trees as tree calculus, and it supports an invertible function from programs to binary natural trees. Now tree calculus is program-complete by definition but the traditional models of computation are not program-complete. Further, VA-calculus is not program complete, since it cannot define equality of programs, but SF-calculus is program-complete since the translation from programs to natural trees can be defined by pattern-matching.

11.3 Extensional, Intensional and Reflective Programs

One of the themes of this work has been the contrast between extensional, intensional and reflective programs. Although many examples have been given, these terms were never given formal definitions, as they were not required for the theorems. By contrast, it was essential to define the programs precisely in order to prove, for example, that equality of programs is decidable. Here, finally, is an approach.

The fundamental distinction is between extensional and intensional accounts of functions. An extensional description of a function is given by the relationship between inputs and outputs, i.e. a functional relation. In set theory, this becomes the definition of a function, as a set of ordered pairs. An intensional description of a function is an algorithm or construction that produces outputs from inputs, in the style of, say, the *intuitionism* of L.E.J. Brouwer. This is not to be confused with an intentional description, which includes the intent of the programmer, as captured by, say, the type of the program. For example, all tautologies are intensionally equal to the term for truth, even though they have different intentions. Conversely, two programmers who intend to produce the identity function may produce programs that differ intensionally, as we saw in combinatory logic.

On this understanding, an extensional program p should gain knowledge of an extensional function f only by applying f to arguments. Arguably, p should be an abstraction of some applications of variables, i.e. a combinator, in which case, a model is *extensionally complete* if and only if it is combinatorially complete. All of the models of computation in this book are extensionally complete in this sense.

Intensional completeness is perhaps the ability to perform arbitrary program analyses, as supported by pattern-matching functions. That is, the allowable patterns are given by terms with the property that if all variables in the pattern are replaced by programs then the result is a program. Then there is little or no choice for the matching algorithm. Of course, this must be formalised in a way that avoids circularity, e.g. by defining the programs without reference to pattern-matching. In this sense, tree calculus and SF-calculus are *intensionally complete* but the traditional models and VA-calculus are not.

Finally, the reflective programs can be viewed as intensional programs whose intent is to be self-applied, as when the size program is applied to itself, or a self-evaluator is self-evaluated. Other examples have arisen during translation, as when A_t is applied to itself.

11.4 Translations to Tree Calculus

Since all of the traditional models of computation have meaningful translations to tree calculus, that preserve computations, it is natural to ask if every model of computation has such a translation. This is a separate question to completeness since, for example, VA-calculus supports a meaningful translation from tree calculus but is not complete. The answer appears to be a qualified yes.

If the model is combinational, and its evaluation rules do not overlap then it seems that there is a translation to tree calculus, in which tagging is used to keep track of the internal structure, just as was done for VA-calculus and SF-calculus. However, if the rules do overlap then the situation does not look promising, since it seems unlikely that tree calculus can support the ambiguity of overlapping rules. In particular, it seems unlikely that one can define parallel-or in tree calculus, whose overlapping rules evaluate both or true x and or x true to true. This intuition is reinforced by the theorem that reduction to normal form can be achieved by leftmost, outermost reduction. The prospect of a translation is even more remote if the model is not confluent, or not combinational.

11.5 Copying is Meta-Theoretic

In criticising traditional models of computation, I have stressed their reliance on meta-theory for, say, substitution, or quotation. Can the same criticism be levelled at tree calculus? A notable feature of the Turing model is that it can be physically implemented in a way that each computation step takes the same fixed amount of time. In particular, to read or write a symbol from a finite alphabet is well controlled. By contrast, evaluation rules for combinations commonly take one step to copy or delete a combination. Since combinations may be arbitrarily large, a single computation step may take an arbitrarily long time. Perhaps a better calculus will keep all of the benefits of tree calculus while supporting the physicality of the Turing model.

The obvious approach would be to use pointers, or let-expressions, to allow sharing, so that trees are generalised to graphs. This would be a dramatic increase in the complexity of the model since, for example, equality of graphs is much harder to prove than that of trees. So this is best left to another time.

11.6 Implications for Programming Languages

The implementation of programming languages requires every program to play two roles, as both a function from inputs to outputs and as a data structure that can be manipulated by a compiler. Within compilers, the current emphasis is on the data structures. Programs are parsed into syntax trees which are then analysed and optimised before being converted to machine code. So, in practice, trees already play

a central role even though they have been almost ignored in the basic theory. One reason for this is that each analysis stage requires its own syntax but this is a burden that may now be avoided. For example, if types are represented as tags, they won't interfere with functionality, and so can be left in place for as long as convenient, even as typed assembly language.

Program optimisations change the program's algorithm while preserving its functional behaviour. A pre-requisite for supporting optimisation in general is the ability to interpret programs. So self-interpretation serves as a convenient proxy for the ability of a language to optimise its own programs and support a self-compiler.

A side effect of the current approach is that compilation tends to be staged as: analysis, optimisation, execution. It is difficult to perform analyses after optimisation, or to optimise during execution. Of course, these things are possible, but they are tricky, so best left to a few experts among those constructing compilers.

These remarks apply even more to those who write applications. Currently, they are unable to influence the compilation process, e.g. to add a new optimisation, without engaging a compilation expert, or becoming one themselves. This is the motivation for domain-specific languages, where users may access these design decisions. A language based on tree calculus may reduce or eliminate the barriers between developers and compilers. It would also be a broad-spectrum language, supporting both high-level abstractions and manipulation of bits and leaves. Such a language should support a wide variety of programming styles, including functional, imperative and object-oriented styles, and a wide variety of type systems, too.

The ability to represent types as tags, means that types and terms can belong to the same syntactic class. As well as being a conceptually simpler means of supporting simple types, or parametric polymorphism, there is potential to develop dependently-typed calculi in which dependent products and sums may be analysed as freely as any program of tree calculus.

11.7 Implications for Logic

The Curry-Howard Isomorphism from the 1960s recognises the deep correspondence between propositions and their proofs on the one hand, and type and their terms on the other. However, this is no accident, since the very first function calculus, by Moses Schönfinkel, was built to give an account of predicate logic. Now that we have a more expressive calculus of functions, perhaps we should look for a more expressive logic which will support proofs that correspond to intensional and reflective programs.

11.8 Implications for Mathematics

Various formalisms have been offered as a foundation for mathematics. The oldest uses numbers, the most popular is set theory, even logic and category theory have been proposed. Might trees be even better?

According to Pythagoras,

All is number

This idea is both simple and appealing but is not especially useful, as illustrated in Figure 1.1. However, it still has quite a hold on us. It re-appeared in Leopold Kronecker's dictum

God created the integers, all else is the work of man.

In computing theory tradition, the natural numbers are the common ground upon which comparisons of systems have been made. However, we have seen that the numbers are ill-suited to represent numerical functions.

In the set theory of Georg Cantor, functions are given by their extensions, as sets of ordered pairs. However, even the simplest numerical functions are then given by infinite sets, while the set of properties of natural numbers is even larger! In the standard terminology, the natural numbers are countably infinite, while the predicates on them are uncountably infinite. This is somewhat troubling when the predicates that can be described in any language are merely countable. Even worse is that naive set theory leads to paradoxes, most famously Russell's Paradox, which arises by considering the set of all sets. These infinities re-appear in Dana Scott's semantics of lambda-calculus, in which the denotation of any recursive function is the limit of its unfoldings.

The development of predicate calculus was introduced by Gottlieb Frege, and formalised by Bertrand Russell and Alfred North Whitehead. While it was the most significant development in logic since Aristotle, it led to Russell's Paradox mentioned above. Russell avoided the paradoxes by introducing a *type hierarchy*, in which higher-order functions act on lower-order functions, but this leads to restrictions on expressive power that re-appear in typed programming.

Category theory was developed by Samuel Eilenberg and Saunders Mac Lane as a means of formalising some relationships between geometry (topology) and algebra, and especially to define the *natural transformations*. Here, *arrows* represent extensional functions in a completely abstract manner, with no attempt to describe their internal structure, e.g. as sets or lambda-abstractions. That is, unlike lambda-calculus, the exploitation of extensionality avoids any use of algorithms, representations or intensionality. Surprisingly many properties of functions, data structures and logic can be developed in this way. William Lawvere considered category theory as a possible foundation for mathematics, but as soon as sets of objects and arrows appear, so do the set-theoretic difficulties.

By contrast, tree calculus is able to describe all of the predicates and functions of interest as finite trees. There is no need to consider the infinite, with its paradoxes,

as it is constructive. Schönfinkel's goal has been met by going beyond the expressive power of predicate calculus. Hence, we can improve on Pythagoras by declaring

All is tree

Let us see how far this can take us.

11.9 Implications for Traditional Computing Theory

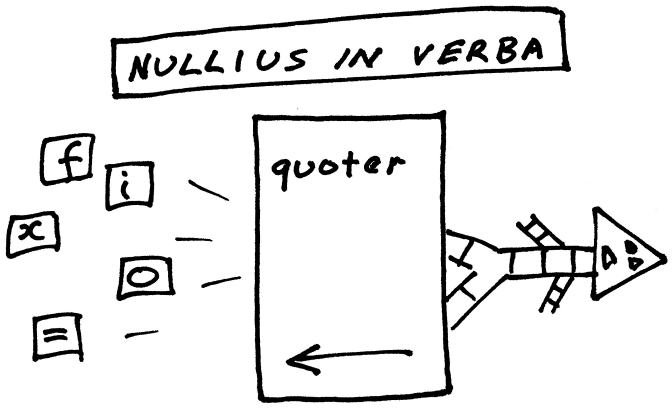
That tree calculus is a better foundation for computing than any previous model of computation is at odds with the traditional understanding, first articulated by Alonzo Church, that all of the traditional models of computation were equivalent and unlikely to be surpassed. After all, tree calculus can be implemented as a Turing machine, so does not appear to be adding anything. Taken at face value, the tension between this view and the theorems presented in this work appears to be a paradox. However, the paradox is resolved by observing that Church's notion of equivalence is very weak. First, it is limited to computation of numbers without any concern for computation of programs. Second, the translations between models are not meaningful, as they do not preserve the process of computation but merely preserve numerical results. Third, the double translations, from one model to another and then back again, are not usually definable in the original model. When comparisons use meaningful translations then the differences between models become much clearer. For example, there is no meaningful translation from *VA*-calculus to combinatory logic, or from combinatory logic to tree calculus.

Jose Vergara and I developed the critique above in a much refereed yet unpublished paper covering eighty years of computing theory. For those who are troubled by the paradox above, it is included, unchanged, in the appendix.

As the remarks above show, the simplicity, naturality and expressive power of trees calculus surpasses that of any previous model of computation, and suggests many avenues for further exploration, in both theory and practice.

Appendix A

Confusion in the Church-Turing Thesis (w. Jose Vergara)



Abstract

The Church-Turing Thesis confuses numerical computations with symbolic computations. In particular, any model of computability in which equality is not definable, such as the λ-models underpinning higher-order programming languages, is *not* equivalent to the Turing model. However, a modern combinatory calculus, the SF-calculus, can query its closed normal forms, and so yields a model of computability that is equivalent to the Turing model. This ability to support queries as well as the usual functions has profound implications for programming language design.

A.1 Introduction

The foundations of computing contain an error, the belief that the λ-calculus gives a complete account of computation, an error born of the confusion of numeric and symbolic computation in the Church-Turing Thesis (CTT). The confusion is resolved by: defining equivalence of models of computability that act on different symbolic domains; proving that the λ-calculus model of computability is *not* equivalent to the Turing model of computability; and proving that there are rewriting systems whose model of computability *is* equivalent to the Turing model. To the extent that rewriting systems like λ-calculus underpin the design of programming languages, this opens up significant new possibilities.

So far, responses to our claim of error have ranged from mild approval, through discouragement, to mockery and hostility. If you are feeling negative, then it is up to us to persuade you to engage in the painstaking effort of reconsidering some fundamental beliefs. If your reaction is mild approval then it is up to us to outline the implications, especially the impact on programming language design.

To be clear, the error is not in the work of Alan Turing, or Turing's Thesis (TT), and has no impact on the study of computing hardware or algorithmic complexity. Further, there is no logical error in the refereed papers of Alonzo Church, or in Church's Thesis (CT). Rather, the source of the conflict lies within the standard account of the expressive power of λ-calculus; as shown in out recent paper *Conflicting accounts of λ-definability* [42].

Corollary A.1 *The equality function of closed normal forms known as Church's δ is λ-definable.*

can be sourced to back to a statement in Kleene's book [46, p. 320] *"The equivalence of the computable to the λ-definable functions ... was proved by Turing 1937"*. This without any restriction on the domain of the computable functions; implies that Church's δ is λ-definable.

However, in Barendregt's book [1, p. 520, Corollary 20.3.3] we have its negation

Corollary A.2 *Church's δ is not λ-definable.*

This error is not of recent origin, but has been present from the beginning. Looking at the CTT historically does not helped us to solve the problem; according to Soare [69, page 11], the Church-Turing Thesis is first mentioned by name in Steven Cole Kleene's book *Introduction to Metamathematics* [46]. However, close examination shows that the thesis is never actually stated there, neither in the section of that name, nor in the preceding text. Since Kleene took some pains to define Turing's Thesis and Church's Thesis, this is a little surprising. In the decades since, one can find many statements of the thesis, e.g. [25, 68, 77, 22, 31, 5], but with a variety of distinct meanings, including TT, CT, or some claim of equivalence, but no clear winner. Hence, we must conclude that

> There is no canonical statement of the Church-Turing Thesis.

Further, there is no way to add a definition that makes Kleene's book whole, since it already contains an error due to the confusion of numeric and symbolic computation, similar to that identified above.

Although Kleene's book cannot supply a definition of CTT, there is one last source to consider. According to Solomon Feferman [22], the Church-Turing Thesis was born (but not named) in Alonzo Church's 1937 review of Alan Turing's paper on computability [75] which declared

> As a matter of fact, there is involved here the equivalence of three different notions: computability by a Turing machine, general recursiveness in the sense of Herbrand-Gödel-Kleene, and λ- definability in the sense of Kleene and the present reviewer.

However: computability by a Turing machine concerns words in some alphabet; general recursion concerns natural numbers; and λ-definability concerns λ-abstractions. So it is not clear whether computation is limited to numbers, or also includes symbolic computations.

If computation is limited to the domain of natural numbers then the statement is true, since all three notions describe the same set of numerical functions, but this version of CTT has no relevance to the design of higher-order languages that now dominate programming language design. As both Turing and Church understood, greater relevance requires consideration of arbitrary domains. However, it is not then clear, at least from the review, what "equivalence of notions" is to mean. The natural interpretation is that a notion is a predicate, so that equivalence of notions is equality of sets. This works for numeric computations but cannot work in general since, as noted above, there are Turing computable functions of λ-terms that are not λ-definable.

If a notion is not a predicate then there is no obvious alternative interpretation. However, by replacing "notion" with "model of computability" we are led to consider equivalence of models of computability. When these models use different domains, then the equivalence must employ simulations or encodings of one domain in another.

Certainly, Church used Gödelisation to simulate symbolic computation by numeric computation [16]. Conversely, λ-calculus can simulate numeric computation through Church encodings. However, the mere existence of simulations in both directions makes for a poor notion of equivalence.

To see this, consider the question of computational completeness, or adequacy. In the common understanding, as exemplified by the version by Barendregt et al given earlier, the Church-Turing Thesis includes the claim that anything that can be computed can be computed by a Turing machine, or using general recursion, or in λ-calculus, that these three models are computationally complete. In this context, it is to be expected that the simulations themselves are computable, since otherwise one could import, say, the solution to the Halting Problem as part of the simulation. While this expectation is natural, it cannot be enforced, since the simulations are meta-functions, outside of any particular model. That is probably why the CTT is called a thesis and not a theorem.

However, this is to give up too easily. When there are simulations in both directions then we can compose them, e.g. to obtain simulations of numbers by numbers, and λ-terms by λ-terms. That is, a natural number is mapped to the Gödel number of its Church encoding, while a λ-term is mapped to the Church encoding of its Gödel number. If the models are computationally complete, then both these re-codings should be computable in their respective models. Now the re-coding of natural numbers is general recursive, but the re-coding of λ-terms is *not* λ-definable, even for closed normal forms. Further, the re-coding of λ-terms in closed normal form is effectively calculable, so it follows that

> The λ-calculus and Turing machines yield models of computability that are **not** equivalent; λ-calculus is **not** computationally complete.

This, in outline, identifies the confusion, shows how it leads to error, and clarifies how to compare models of computability. What then, are the consequences of abandoning CTT? Since we still have CT and TT, perhaps it doesn't matter too much. Our response comes in two parts. First, we will show that there are new calculi that improve upon λ-calculus. Second, these new calculi suggest radical approaches to the design of programming languages. Both propositions will be incredible to adherents of CTT.

The great strength and weakness of λ-calculus is that there is only one way to obtain information, namely to evaluate functions; it is *extensional*. It follows that there is no uniform mechanism for performing analysis of functions or other structures, for writing queries; it is not *intensional*. Rather, there are ad hoc techniques for analysing (the representations of) natural numbers, etc. Hence, when general, symbolic computation is confused with numerical computation, the λ-calculus appears to be more powerful than it is.

However, once the general importance of intensionality is recognised, we can discover or build calculi that support it, as well as extensionality. Our latest published example is SF-calculus [38], a combinatory calculus which is more expressive than traditional combinatory logic [32] in that it can Gödelise its own closed normal forms, and so perform arbitrary analyses of closed normal forms. Thus,

The SF-calculus and Turing machines yield models of computability that **are** equivalent.

Now let us consider the implications for programming language design. The supposed pre-eminence of λ-calculus makes it impossible to comprehend why there are so many programming styles in use, or even why there are so many λ-calculi. It compares badly with the Turing model, which works so well for representing hardware. Further, λ-calculus has constrained the thinking of those who explore the possibilities for programming language design. Freed from these constraints, we offer an alternative view of the possibilities.

The existence of a calculus with generic support for both extensionality and intensionality opens up many possibilities for programming language design. It can be used to support generic queries of data structures, so that a program that searches for an item may be applied to a list or binary tree or any other sort of structure, as has been implemented in the programming language **bondi**[9]. More generally, SF-calculus is able to analyse programs, so that there is no need to separate program analysis from program execution. Further, if the equivalence of SF-calculus and Turing machines is strong enough, if all programming is built from extensional and intensional functions, then there may be a single programming language that is universal for software, in the sense that the Turing model is universal for hardware.

The structure of the paper is as follows. Section A.1 is the introduction. Section A.2 exposes an error in Kleene's book. Sections A.3–A.5 consider Church's Thesis, Turing's Thesis and the Church-Turing Thesis as described in Kleene's book, but drawing on other sources as appropriate. Sections A.6 and A.7 consider models of computability and their comparison. Section A.8 considers the impact of λ-calculus on programming language design. Section A.10 surveys some models of intensional computation. Section A.11 draws conclusions.

A.2 Confusion Leads to Error

(1)	"... every function which would naturally be regarded as computable is computable under his [Turing's] definition, i.e. by one of his machines ..."	(page 376)
(2)	"The equivalence of the computable to the λ-definable functions . . . was proved by Turing 1937."	(page 320)
(3)	Every function which would naturally be regarded as computable is λ-definable.	(equivalence)
(4)	The equality of λ-terms in closed normal form is λ-definable.	(specialise)

Fig. A.1 A faulty argument, with premises quoted from Kleene [46]

Figure A.1 contains a valid argument with a false conclusion. [A closed normal form is an irreducible term that has no free variables.] Hence, one of the premises

must be false, but these are direct quotations from Kleene's book. Hence, Kleene's book contains a false statement.

Now let us consider this in more detail. The premises (1) and (2) are direct quotes from Kleene's book. Then (3) follows from (1) by Leibnitz equality, by substituting equals for equals, in this case by substituting "λ-definable" for "computable" as per the equivalence in (2). Finally, (4) follows from (3) by specialisation, since equality of λ-terms in closed normal form is naturally regarded as computable. The conclusion asserts the λ-definability of equality of closed λ-terms in normal form. Since the conclusion is false [4, page 519] there must be a fault in the argument.

Two objections have been raised about this. First, it has been suggested that we are supporting a false conclusion. However, to assert that the argument is valid is not to support the truth of the conclusion. Rather, that the conclusion is false is essential to our purpose, of showing that one of the premises must be false. Second, it has been suggested that we must have taken the quotes out of context. However, it is hard to construct a context that would invalidate the argument, unless "every function which would naturally be regarded as computable" can be construed to exclude some functions which would naturally be regarded as computable, such as equality of λ-terms in closed normal form.

The source of the problem is that (1) only makes sense if the computable functions are taken to include symbolic functions, while in (2), Turing's proof was for *numerical* functions only, and does not include the function in (4). The source of the error is the confusion of symbolic and numeric computations. To clarify the issues, let us consider the theses of Church and Turing, etc. and their integration within the Church-Turing Thesis.

A.3 Church's Thesis

The background to Church's Thesis is Church's paper of 1936 *An unsolvable problem of number theory* [16] which, alongside Alan Turing's work discussed later, showed that some numerical problems do not have a computable solution, so that Hilbert's decision problem does not have a general solution. It is clear that Church's primary focus was on numerical functions, as all his formal definitions were expressed in numerical terms. For example, he writes, "A function F of one positive integer is said to be λ-definable if . . . " [16, page 349]. Again, in Theorem XVII he writes "Every λ-definable function of positive integers is recursive".

That said, he does broaden the discussion when considering *effective calculability*. On the one hand, an *effectively calculable* function of positive integers is defined to be "a recursive functions of positive integers" or "a λ-definable function of positive integers". On the other hand, he writes:

> . . . [in] some particular system of symbolic logic . . . it is necessary that each rule of procedure be an effectively calculable operation, . . . Suppose we interpret this to mean that, in terms of a system of Gödel representations for the expressions of the logic, each rule of procedure must be a recursive operation, . . .

That is to say, he supposes that a symbolic function is recursive if it can be simulated by a numerical function that is recursive, where simulation is defined using Gödel numbering. Later, this supposition hardens to become a premise when he writes, in a footnote:

> . . . in view of the Gödel representation and the ideas associated with it, symbolic logic can now be regarded, mathematically, as a branch of elementary number theory.

By this definition, the symbolic function that decides equality of closed normal λ-terms is effectively calculable, is recursive, since it is simulated by the equality of natural numbers. However, there is no sense in which equality of closed normal λ-terms is λ-definable. Although there is no logical error here, Church does not show any awareness of the tension between his accounts of λ-definability and effective calculability, which makes it easy for Kleene to fall into error.

In contrast to Church's narrow constructions, Kleene's definitions have broad scope. For example, he gives three definitions of λ-definability in his paper *λ-definability and recursiveness* [45], according to whether the domain of definition is the non-negative integers, the λ-terms themselves, or other mathematical entities for which a λ-representation has been given. It is the interplay between these definitions that is the primary cause of confusion.

Kleene introduces Church's Thesis as Thesis I in his 1952 book (Section 60, page 300) as follows:

(CT) Every effectively calculable function (effectively decidable predicate) is general recursive.

The crucial question is to determine the domain of the effectively calculable functions.

At the point where Church's Thesis is stated, in Section 60, the general recursive functions are all numerical, so it would seem that the effectively calculable functions must also be numerical. However, he does not include the phrase "of positive integers" in his statement of the thesis, in the careful manner of Church. We are required to add this rider in order to make sense of the thesis.

Later, in Section 62. Church's Thesis, Kleene presents seven pages of arguments for the thesis, which he groups under four headings A–D. In "(B) Equivalence of diverse formulations" he asserts that the set of λ-definable functions and the set of Turing computable functions are "co-extensive" with the set of general recursive functions. Again, this only makes sense if the functions are presumed to be numerical functions. This paragraph is also the source of statement (2) from Figure A.1.

If we were to stop at this point, then the explanation of the faulty argument would be quite simple: statement (2) should be read in a context where all functions are numerical, or be replaced by "The equivalence of the computable to the λ-definable **numerical** functions was proved by Turing 1937". The restriction to numerical functions propagates to statement (3) which cannot then be specialised to equality of λ-terms.

However, in "(D) Symbolic logics and symbolic algorithms", Kleene reprises Church's definition of symbolic functions that are recursive, so that, by the end

of Section 62, we have two definitions of effectively calculable functions and of recursive functions. So there are two versions of Church's Thesis, one for numerical functions (NCT) and one for symbolic functions (SCT). Unpacking the definition of a general recursive symbolic function to make the role of simulations explicit, the two versions become:

(NCT) Every effectively calculable numerical function (effectively decidable numerical predicate) is general recursive.

(SCT) Every effectively calculable function (effectively decidable predicate) can be simulated by a function which is general recursive.

Now consider the faulty argument in the context of SCT. We can infer that every λ-definable function is general recursive, but *not* that every general recursive function is λ-definable: these sets are not co-extensive; there is no equivalence in which (2) holds for symbolic computation.

Summarising, we see that Church was careful to separate theorems about numerical functions from the discussion of computation in symbolic logic. By contrast, Kleene presents a single statement of Church's Thesis with evidence that confuses the numerical with the symbolic. In turn, this confuses two different questions: whether two sets of numerical functions are the same; and whether there is an encoding that allows functions in a symbolic logic to be simulated by recursive functions on numbers. These confusions can be defused by isolating two versions of Church's Thesis which, from the viewpoint of Post [62], qualify as scientific laws. Then the faulty argument can be evaluated as follows. If (2) refers to numerical computation only then the specialisation that produces (4) is invalid. If (2) refers to symbolic computation then it is false, on the assumption that the equivalence is a Leibnitz equality, that supports substitution of equals for equals. We shall explore alternative notions of equivalence in Sections A.6 and A.7.

A.4 Turing's Thesis

Turing's paper of 1936 *On Computable Numbers, with an application to the Entscheidungsproblem* was, like Church's paper, concerned with numerical computation and Hilbert's decision problem. Like Church, Turing was careful to limit his definitions, e.g. of *computable functions*, to numerical functions while showing awareness of a broader scope. For example, in the first paragraph he writes:

> Although the subject of this paper is ostensibly the computable *numbers*, it is almost equally easy to define and investigate computable functions of an integral variable, or a real or computable variable, computable predicates, and so forth.

Similarly, the use of an unspecified alphabet of symbols on the tape of a Turing machine encourages us to consider computation over arbitrary symbolic domains.

Once again, Kleene confuses these two meanings in his third piece of evidence for Church's Thesis, headed (C): Turing's concept of a computing machine [46, page

320] where Kleene writes "Turing's computable functions (1936-7) are those which can be computed by a machine of a kind which is designed, according to his analysis, to reproduce all the sorts of operations which a human computer could perform, working according to preassigned instructions." As we have seen above "Turing's computable functions" are, by definition, numerical, while a human computer faces no such restriction.

In more compressed form, this confusion re-appears in Kleene's statement of Turing's Thesis. It is given in Section 70. Turing's thesis, as a sub-ordinate clause of the opening statement which, when elevated to an independent thesis, becomes:

(TT) Every function which would naturally be regarded as computable is computable under Turing's definition, i.e. by one of his machines.

Now the phrase "every function which would naturally be regarded as computable" surely includes all computations in formal systems such as λ-calculus or combinatory logic [32]. For example, it would be a distortion to assume that "naturally" here refers to the natural numbers. On the other hand, "Turing's definition" is certainly numerical. As with Church's Thesis, the solution is to create a numerical thesis (NTT) and a symbolic one (STT) as follows:

(NTT) Every numerical function which would naturally be regarded as computable is computable under Turing's definition.

(STT) Every function which would naturally be regarded as computable can be simulated by a function which is computable under Turing's definition.

From the viewpoint of Post [62], both versions of the thesis will qualify as scientific laws.

A.5 The Church-Turing Thesis

As noted in the introduction, the Church-Turing Thesis is mentioned in Kleene's book without being defined. Among the statements in Kleene's book, the closest candidate is the opening of Section 70:

> Turing's thesis that every function which would naturally be regarded as computable is computable under his definition, i.e. by one of his machines, is equivalent to Church's thesis by Theorem XXX.

On first reading, it is rather difficult to determine the nature of this declaration, as it contains the statement of Turing's thesis (TT) plus the statement of a theorem,

> Turing's Thesis is equivalent to Church's Thesis.

with its proof "by Theorem XXX". If this is the Church-Turing Thesis, then it is a theorem asserting logical equivalence of two theses. Further, the proof by Theorem XXX (on the same page) is numerical, which implies that the thesis is also

numerical. One must look elsewhere for a general, symbolic result, e.g. in Kleene's evidence for Church's Thesis mentioned earlier. In any event, the quote above cannot carry the burden that CTT is expected to bear.

The subsequent literature of the last fifty years has thrown up many versions of the theses, e.g. [25, 68, 22, 5]. Perhaps the best way to make sense of this variety is to view the Church-Turing Thesis as the class of all statements that are logically equivalent to any version of Church's Thesis or of Turing's Thesis. In this manner, all of the theses and proofs of logical equivalence are gathered under a single heading. This broad interpretation may explain why Kleene did not give a statement of it. In any event, this broad interpretation seems most appropriate when considering the impact of the Church-Turing Thesis on programming language design in Section A.8. Before leaving the founders of the theory of computation, let us consider how they might have compared models of computability, to decide relative expressive power, and completeness.

John Longley's Book *Higher order computability* [50] presents an impressive review of the las 50 years of the literature and looks at the question of what is the meaning of computability for domains beyond natural numbers.

> Whereas the work of Church and Turing from the 1930s provided a definitive concept of computability for natural numbers and similar discrete data, our present inquiry begins by asking what 'computability' might mean for data of more com- plex kinds. In particular, can one develop a good theory of computability in settings where 'computable operations' may themselves be passed as inputs to other com- putable operations? What does it mean to 'compute with' (for example) a function whose arguments are themselves functions?

However it leaves for future work the development of models that are applicable for the design and development of programming languages for higher order computation.

In addition; The paper *The representational foundations of computation* [63] Explores the role of representations (encodings) for computability theory for a philosophical point of view. The paper points out the importance of being aware of the gap between the domain of a model of computation and a desired higher order domain. However the paper does not identify any problem with the literature in particular the CTT.

A.6 Models of Computability

We adopt the simplest definition of model of computability in which the discussion of simulation makes sense. This was introduced by Boker and Dershowitz as a *model of computation* [8], but since the focus is upon functions for which a computation is possible, rather than the actual mechanics of computation, it seems more accurate to call them models of computability. Note, too, that the domain of the computable functions is not actually required to be symbolic in any way, or even to be enumerable, however natural this may be. This makes the notion too weak for many purposes, but here it emphasises the generality of our results.

A *model of computability* $(D, \mathcal{F})$ is given by a domain D which is a set of *values* that provides arguments and results of computations, and a collection $\mathcal{F}$ of partial functions from powers of D to D. Here are some examples.

The partial recursive functions on natural numbers form a model with domain given by the natural numbers and functions given by the partial recursive functions. Call this the *recursive model of computability*.

Recall that an injective function is a total function that does not identify distinct arguments. For any finite alphabet Σ, and any domain D equipped with an injective function from D to the words of Σ, the *Turing model of computability on D* has domain D and partial functions given by those which can be computed by a Turing machine with alphabet Σ. When the choice of domain is understood from the context, or unimportant, then it may be called the *Turing model of computability*.

Any *applicative rewriting system* [73] has a *normal model* whose values are the closed terms in normal form, and whose partial functions are those representable by closed terms in normal form. Further, any subset D of values determines a model, where the partial functions are now defined on a value in D only if the result is also in D.

For example, classical combinatory logic [32] has terms built from applications of the operators S and K and variables. As well as its normal model, there is a *numerical model* whose domain is restricted to be the Church numerals. Also, one can use Polish notation to encode combinators as words using the alphabet $\Sigma = \{A, S, K\}$, where A is for application. For example, $S(KK)$ is mapped to $ASAKK$. This yields a Turing model of computability for SK-normal forms.

Similarly, λ-calculus has *normal models* and *numerical models*, once the terms and reduction rules have been specified. First, a λ-term is unchanged by renaming of bound variables, i.e. is an α-equivalence class in the syntax [4]. Since this equivalence is a distraction, we will work with λ-terms using deBruijn notation [12] so that, for example, $\lambda x.x$ becomes $\lambda 0$ and $\lambda x.\lambda y.xy$ becomes $\lambda\lambda 10$. Second, there are various choices of reduction rules possible, with each choice producing a normal λ-model. Define a *λ-model of higher-order programming* to be any normal λ-model in which equality of closed normal forms is not definable. This excludes any model whose domain is numerical, and would seem to include any models that are relevant to higher-order programming. Certainly the λ-models of higher-order programming include those given by β-reduction alone, or $\beta\eta$-reduction.

A.7 Comparison of Models

Now consider what it means for one model of computability to be more expressive than another. The simple interpretation requires that their computable functions have the same domain and then compares sets of computable functions by subset inclusion, as is done by John Mitchell [54] and Neil Jones [43]. The choice of domain is important here. For example, if the domain consists of natural numbers then the λ-model and the recursive model are indeed equivalent. However, this restriction

is unreasonable for modeling higher-order programming since functions must be among the values. Now it is an easy matter to see that any normal λ-model of computability has fewer computable functions than the Turing model. For example, no λ-term can decide equality of values but this function is in the Turing model.

The complex interpretation of relative expressive power allows the domains to vary, but now the comparison of computability over domains D_1 and D_2 must be mediated by simulations, which are given by encodings. Note that it makes no sense to consider a simulation in one direction, and compare sets of functions in the other direction, since, as observed by Boker and Dershowitz [8], this can lead to paradoxes. Rather, there should be encodings of each domain in the other that are, in some sense, inverse to each other. Various choices are possible here but two requirements seem to be essential. First, the encodings should be injective functions, since distinct values should not be identified. Second, the encodings should be *passive*, in the sense that they are not adding expressive power beyond that of their target model. In particular, they should be effectively calculable. This requirement can only be verified informally, on a case by case basis, since functions from D_1 to D_2 are not in the scope of either model.

A *simulation* of a model of computability $(D_1, \mathcal{F}_1)$ in another such $(D_2, \mathcal{F}_2)$ is given by an injective *encoding* $\rho : D_1 \to D_2$ such that every function f_1 in $\mathcal{F}_1$ can be *simulated by* a function f_2 in $\mathcal{F}_2$ in the following sense: for all $x_1, \ldots, x_n \in D_1$ such that $f_1(x_1, \ldots, x_n)$ is defined, then $f_2(\rho(x_1), \ldots, \rho(x_n))$ is defined and

$$\rho(f_1(x_1, \ldots, x_n)) = f_2(\rho(x_1), \ldots, \rho(x_n)) \; .$$

For example, Gödelisation provides a simulation of the normal SK-model into the recursive model. More generally, Church and Kleene both use this notion of simulation to define recursive symbolic functions. Gödelisation seems to be passive.

Further, the encoding of the natural numbers using Church numerals provides a simulation of the recursive model into the normal SK-model or any normal λ-model.

Other related notions of simulation can be found in the literature, e.g. [43, 8]. For example, Richard Montague [57, page 430] considers, and Hartley Rogers [65, page 28] adopts, a slightly different approach, in which the encoding of numbers is achieved by reversing a bijective Gödelisation. However, this inverse encoding may not be a simulation. For example, the equality of numbers cannot be simulated by a λ-term over the domain of closed λ-terms in normal form. Rogers, like Kleene, ensures a simulation by *defining* the computable functions in the symbolic domain to be all simulations of partial recursive functions, but this says nothing about λ-definability.

Given the formal definition of simulations, it may appear that the strongest possible notion of equivalence is that each model simulates the other. However, if the encodings are passive then so are the *recodings* from D_1 to D_1 and from D_2 to D_2 obtained by encoding twice. Since these are in the scope of the two models, we can require that recodings be computable.

Let $(D_1, \mathcal{F}_1)$ and $(D_2, \mathcal{F}_2)$ be two models of computability with simulations $\rho_2 : D_1 \to D_2$ and $\rho_1 : D_2 \to D_1$. Then $(D_2, \mathcal{F}_2)$ is *at least as expressive* as $(D_1, \mathcal{F}_1)$ if the recoding $\rho_2 \circ \rho_1 : D_2 \to D_2$ is computable in $\mathcal{F}_2$. If, in addition, $(D_1, \mathcal{F}_1)$ is more

expressive than $(D_2, \mathcal{F}_2)$ then the two models are *weakly equivalent*. If the recodings both have computable inverses then the weak equivalence is an *equivalence*. Note that these notions of equivalence is indeed an equivalence relation on models of computability.

It is interesting to compare this definition with those for equivalence of *partial combinatory algebras* by Cockett and Hofstra [19] and John Longley [49]. They would not require the encodings to be injective, but Longley would require that the recodings be invertible. Adding the latter requirement is perfectly reasonable but is immaterial in the current setting.

It is easy to prove that the recursive model is at least as expressive as any λ-model. Our focus will be on the converse.

Theorem A.1 *Any model of computability that is at least as expressive as the recursive function model can define equality of values.*

Proof The recursive model is presumed to use 0 and 1 for booleans. In the other model, identify the booleans with the encodings of 0 and 1 so that the equality function is given by recoding its arguments and then applying the simulation of the equality of numbers. □

Corollary A.3 *The normal model of computability for SK-calculus is not weakly equivalent to the recursive function model.*

Proof If the normal SK-model could define equality then it could distinguish the values SKK and SKS but the standard translation from combinatory logic to λ-calculus identifies them (both reduce to the identity), and so they cannot be distinguished by any SK-combinator. □

Corollary A.4 *No λ-model of higher-order programming is weakly equivalent to the recursive model of computability.*

Proof Since normal λ-models do not define equality, the result is immediate. Note that Longley has proved the analogous result for his definition of equivalence [49].□

Now we can express some classical results in the new terminology.

Theorem A.2 *Turing's Numerical Thesis is logically equivalent to Church's Numerical Thesis.*

Proof Apply Kleene's 30th theorem, i.e. Theorem XXX [46, page 376]. □

Theorem A.3 *The recursive model of computability is weakly equivalent to any Turing model of computability.*

Proof The traditional simulations yield encodings that are computable. □

Corollary A.5 *Turing's Symbolic Thesis is logically equivalent to Church's Symbolic Thesis.*

Proof Any simulation into a Turing model yields a simulation into the recursive model by composing with the simulation given by weak equivalence. The converse is similar. □

Corollary A.6 *No λ-model of higher-order programming is weakly equivalent to the Turing model.*

Proof Since weak equivalence is transitive, the result follows from Corollaries A.4 and A.3. □

Now any reasonable notion of equivalence must imply weak equivalence. So it follows that the Turing model of computability is strictly more expressive than any λ-model of computability suitable for modelling higher-order programming. The relationships are summarised in Figure A.2, where **N** denotes the natural numbers, λ denotes the λ-terms in closed normal form, and the arrows denote the various codings and re-codings.

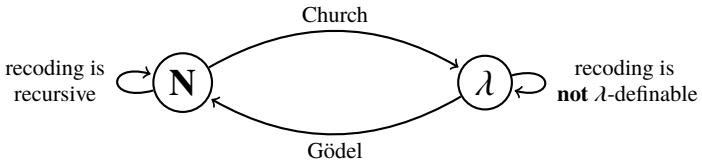


Fig. A.2 Recoding is not λ-definable

A.8 Programming Language Design

Here are three examples, from the last fifty years, of how confusion in the Church-Turing Thesis has limited the design space for programming languages.

Peter Landin's seminal paper of 1966 *The Next 700 Programming Languages* [48] proposes a powerful model of programming language development in which λ-calculus is the universal intermediate language. That is: create a source language with various additional features such as types, or let-declarations; transform this into λ-calculus; then implement an evaluation strategy for λ-calculus (e.g. lazy or eager) as a Turing machine. His main comment about the suitability of λ-calculus for this role is:

> A possible first step in the research program is 1700 doctoral theses called "A Correspondence between x and Church's λ-notation."

So, Landin sees a central role for λ-calculus, with a research program that would occupy a generation of computer scientists. This research program influenced many

languages designs, including Algol68 [64], Scheme [71], and ML [53]. This approach continues to hold sway.

Matthias Felleisen, in his paper of 1990 on the expressive power of programming languages [23], comments:

> Comparing the set of computable functions that a language can represent is useless because the languages in question are usually universal; other measures do not exist.

After this appeal to the Church-Turing Thesis, the paper goes on to consider various λ-calculi, in the belief that nothing has been left out. Although Felleisen makes some useful distinctions, his paper excludes the possibility of going beyond λ-calculus, which limits its scope.

Indeed the development of Embedded Domain Specific Languages (DSL) using Higher Order Syntax [20]; can be complex due to the need to reflect and reify the terms of a language between two levels. A deep level where lambdas are encoded as lambdas in the host language, and a shallow level where lambdas and other terms are encoded as data structures. An understanding of programming language implementation beyond the CTT has the potential to simplify the development of embedded DSL's by using a uniform approach to functions and data structures.

Robert Harper's book *Practical Foundations for Programming Languages* [30] contains a version of Church's Thesis (given as Church's Law) which is careful to limit its scope to natural numbers. It also emphasises that equality is not λ-definable (given as Scott's Theorem). However, while the title proclaims the subject matter to be the foundation for programming languages in general, it is solely focused on the λ-calculus, with no allowance made for other possibilities. If there remains any doubt about the author's views about foundations, consider the following slogan, attributed to him by Dana Scott in the year of the book's publication. In a talk during the Turing Centenary celebrations of 2012, he asserts [67]:

λ conquers all!

There is no explicit justification given for this focus, which we can only assume is based upon Landin's research program, and the Church-Turing Thesis.

Here are some examples of calculi and languages that don't easily fit into Landin's program, since they may exceed the expressive power of λ-calculus. Candidates include: first-order languages such as SQL [15]; languages without an underlying calculus, such as Lisp [52] with its operators `car` and `cdr`; and the intensional programming language Rum [72]. Richer examples include the *self-calculus* of Abadi and Cardelli for object-orientation [1], the *pattern calculus* for structure polymorphism [35], the *pure pattern calculus* for generic queries of data structures [39, 36]. Again, the **bondi** programming language [9] uses pure pattern calculus to support both generic forms of the usual database queries, and a pattern-matching account of object-orientation, including method specialisation, sub-typing, etc.

The richer calculi above have *not* been shown equivalent to λ-calculus. Rather, all evidence points the other way, since, for example, no-one has defined generic queries in pure λ-calculus.

Summarising, while the ability to simulate λ-calculus as a Turing machine has been enormously fruitful as an aid to implementation, the larger claims of the Church-Turing Thesis have been suggesting unnecessary limits on programming language design for almost fifty years.

A.9 Extensional and Intensional Computation

An account of functions is intensional if it allows access to the underlying algorithm of a function and not just its input-output relation. In this sense, the Turing computable functions, being mere relations, are extensional, while the Turing model of computation, in which we have access to the machine that performs the computation, is intensional. In this sense, the lambda-calculus is also intensional [76]. However, this ability to access the algorithm, to analyse a lambda-abstraction, cannot be represented within the lambda-calculus itself. According to this stronger standard, the lambda-calculus is *extensional* in the sense of Church [17, page 2]. By contrast, an *intensional* theory of computation is able to examine the algorithms, to analyse or query them, just as we query data structures for pairs, lists or binary trees. In this sense, the Turing model of functions is intensional, since functions are represented as symbols on a tape, all of which are readable.

Even within λ-calculus, intensionality proves to be quite desirable, as a means of recording type information or naming constructors, etc. Since such information, such intensions, cannot be properly represented within the pure λ-calculus, the community is continually developing new variants of λ-calculus in which the additional information is provided to the program analyst. Somehow, the existence of hundreds, if not thousands of different calculi does not seem to disturb our foundational beliefs to any extent. So resigned are we to the situation that the reference work *Term Rewriting Systems* defines the λ-calculus to be a family of calculi!

Intensionality is so central to partial evaluation and decompilation that various efforts have been made to to extend λ-calculus with Gödelisation. This has been done for simply typed λ-calculus [6], a combinatory calculus [29], and untyped λ-calculus augmented with some labels [56]. However, some of the attractive features of pure λ-calculus, such as being typable, confluent, and a rewriting system have been compromised.

Others have tried to develop a direct understanding of intensionality. It has been the subject of much research by philosophers [57, 21, 14, 44], logicians [24, 74, 11], type theorists [51, 58, 10] and computer scientists [33, 3, 13], so before proceeding, let us determine what it will mean for us. In the concrete setting of λ-calculus, when should two λ-terms be considered intensionally equal? Should this be limited to closed normal forms or are arbitrary terms to be included? In part, the answer depends upon whether your semantics is denotational or operational.

Denotational semantics constructs the meaning of a program from that of its fragments, whose contexts may supply values to free variables, or determine whether or not the evaluation of the fragment terminates. Examples may be found in *domain*

theory [47, 66], *abstract and algebraic data types* [28, 55], the *effective topos* [34], and *partial combinatory algebras* [7, 18, 49]. This suggests that all terms are included but equality of arbitrary lambda terms is not computable [4, page 519].

By contrast, other semantics do not account for terms without normal form or for open terms, and so avoid the need to assign them values. For example, *axiomatic recursion theory* [65, 43] uses Kleene equality [46, page 327]. Again, *operational semantics* in the style of Gordon Plotkin's *structured operational semantics* [60] can limit its values to be closed terms that are, in some sense, normal, e.g. are irreducible, or in head-normal form [4], etc. Thereby, various problems caused by non-termination, such as the difficulty of defining the parallel-or function [2], do not arise. In particular, it is easy to represent equality.

Thus, we see the challenge is to extend standard calculi, such as λ-calculus, with the general ability to query internal structure, and this while retaining many of the attractive features of λ-calculus, such as being a rewriting system, especially one that is confluent or typable. The resulting calculi are expressed, not as a variant of λ-calculus, but as an improvement upon it. As a result, we are routinely criticised for violating the Church-Turing Thesis, which motivated the production of this paper. Pure pattern calculus is an example of this approach; two more follow below.

A.10 SF-Calculus

Like λ-calculus, the traditional combinatory logic, the SK-calculus, is extensional. The SF-calculus [38] supports intensionality by replacing the operator K of SK-calculus with a *factorisation operator* F that is able to query the internal structure of terms in closed normal form. Such terms are either operators, also called atoms, or irreducible applications, called compounds. Being irreducible, it is perfectly safe to split such a compound PQ into its components P and Q, which reveals their internal structure. It cannot be stressed too much that not every application is a compound. For example, $FFFF$ is not a compound as it reduces to the atom F. In this manner, closed normal forms can be completely analysed into their constituent operators, whose equality can be tested by extensional means. Details of the calculus can be found in the original paper [38].

The approach supports definable equality of closed normal forms, typed self-interpreters [40] (see also [61]), and Guy Steele's approach [70] to growing a language [41]. It can also be extended to a concurrent setting [27, 26]. Since the factorisation operator F cannot be defined in terms of S and K, the SF-calculus is strictly more expressive than SK-calculus. Indeed, SF-calculus is complete in the following sense.

Theorem A.4 *The normal model of computability for SF-calculus is equivalent to the recursive model.*

Proof That Gödelisation and Church encoding are both simulations follows from the work of Church [16] and Kleene [45], so that it is enough to show that both

re-codings are computable. It is easy to see that the recoding of numbers to numbers is recursive. In the other direction, the recoding of SF-combinators can be described by a pattern-matching function that acts on the combinators in normal form. Such pattern-matching functions are represented by SF-combinators because SF-calculus is *structure complete* [38]. Note that this proof does not apply for SK-calculus as this is merely *combinatorially complete* [73]. Also, since the recodings are computably invertible, this is an equivalence. □

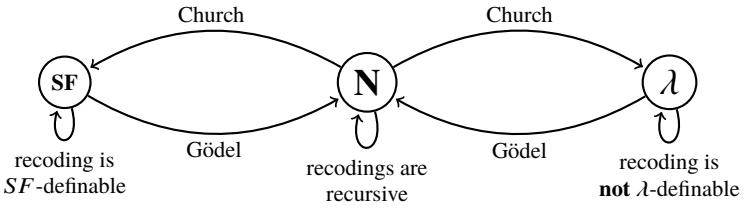


Fig. A.3 Recoding is SF-definable but not λ-definable

The situation can be summarised in Figure A.3, where SF denotes the closed normal forms of SF-calculus.

This makes SF-calculus a superior foundation for higher-order programming languages compared to, say,SK-calculus. The main thing that is missing is native support for λ-abstraction, which is the subject of current research.

Ongoing work has developed λSF-calculus [37], which combines λ-calculus and SF-calculus, so that one may query λ-abstractions. Indeed, Gödelisation of arbitrary closed normal forms, including λ-abstractions, is definable within the calculus.

This calculus, or something like it, may well provide a universal calculus in which to support all of the variants of λ-calculus, the different programming styles, and all the machinery necessary for their implementation. However, we are getting ahead of ourselves. For now, the point is that such a calculus may well exist, but we will never look for it if we believe that λ-calculus gives a complete account of computation.

A.11 Conclusions

The Church-Turing Thesis has been a confusion since it was first named, but not defined, by Kleene in 1952. The numerical results of Church and Turing support numerical versions of their eponymous theses, in which sets of numerical functions are co-extensive. Further, there are separate theses for symbolic computation, involving simulations of one model of computability in another. Kleene confused these two settings, with a little encouragement from Church.

Once the role of simulations is made explicit, it is easier to see that mutual simulation yields an equivalence of models only if both re-codings are computable,

each in its respective model. This requirement exposes the limitations of λ-calculus, since Gödelisation is not λ-definable, even for closed λ-terms in normal form. λ-definability is not, in this sense, equivalent to Turing computability.

These limitations are, in some sense, well known within the λ-calculus community, in that λ-calculus cannot define equality, even of closed normal forms. Indeed, those working with categorical models of computability, or analysing programs defined as λ-terms, are acutely aware of these limitations. However, the community as a whole manages to work around these limitations without giving up on the Church-Turing Thesis. Overall, this is easier to do when the thesis has no canonical form, and is labeled a thesis, and not a theorem. Students who ask awkward questions may be told "Beware the Turing tarpit!" [59] or "Don't look under the lambda!" or "You're asking the wrong question!" which closes off discussion without clarifying anything.

The limitations of λ-calculus are essential to its nature, since λ-terms cannot directly query the internal structure of their arguments; the expressive power of λ-calculus is extensional. This support for abstraction barriers is often, and rightly, held up as one of its key strengths. Equally, its ability to query the internal structure of natural numbers, e.g. to compute predecessors, is also celebrated. However, its inability to query internal structure in a general way goes unremarked.

Rather, intensional computation requires a fresh outlook. The simplest illustration of this is the SF-calculus whose factorisation operator F is able to uniformly decompose normal forms to their constituent operators. Since SF-calculus also has all of the expressive power of SK-calculus, its normal model of computability is equivalent to the Turing model or the recursive function model. Current work on λSF-calculus suggests we may be able to define a calculus that, unlike λ-calculus, is universal, in the same way that there is a universal Turing machine.

The implications of this for programming language design are profound. The **bondi** programming language has already shown how the usual database queries can be made polymorphic, and that object-orientation can be defined in terms of pattern-matching. Now the factorisation operator paves the way for program analysis to be conducted in the source language, so that growing a language can become easier than ever.

In short, confusion in the Church-Turing Thesis has obscured the fundamental limitations of λ-calculus as a foundation for programming languages. It is time to wind up Landin's research program, and pursue the development of intensional calculi and programming languages.

Acknowledgments We thank Jacques Carette, Robin Cockett, John Crossley, Nachum Dershowitz, Thomas Given-Wilson, Neil Jones, Jens Palsberg, Reuben Rowe, Peter Selinger, and Eric Torreborre for comments on drafts of this paper.

References

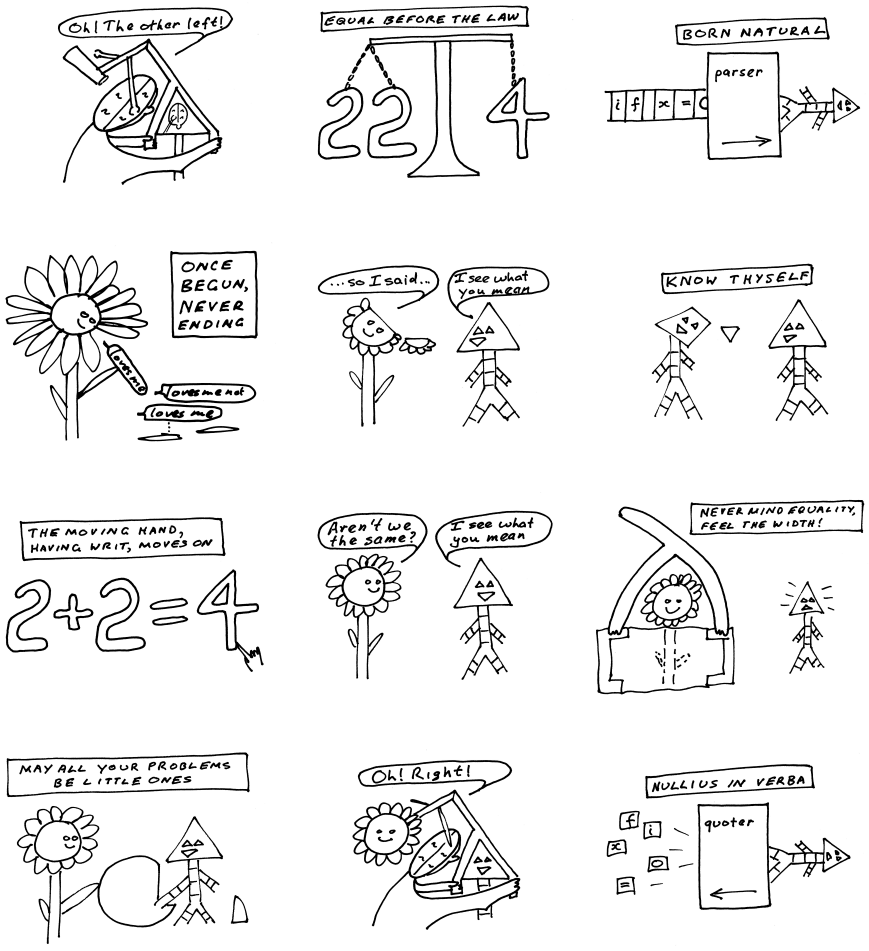
1. Abadi, M., Cardelli, L.: A Theory of Objects, 1st edn. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1996)
2. Abramsky, S., Jagadeesan, R., Malacaria, P.: Full abstraction for PCF. Information and Computation **163**(2), 409–470 (2000). DOI 10.1006/inco.2000.2930. URL http://dx.doi.org/10.1006/inco.2000.2930
3. Artemov, S., Bonelli, E.: The intensional lambda calculus. In: S.N. Artemov, A. Nerode (eds.) Logical Foundations of Computer Science, *Lecture Notes in Computer Science*, vol. 4514, pp. 12–25. Springer Berlin Heidelberg (2007). DOI 10.1007/978-3-540-72734-7-2. URL http://dx.doi.org/10.1007/978-3-540-72734-7-2
4. Barendregt, H.P.: The Lambda Calculus — Its Syntax and Semantics, *Studies in Logic and the Foundations of Mathematics*, vol. 103, revised edn. North-Holland, Amsterdam (1984)
5. Barendregt, H.P., Manzonetto, G.: Turing's contributions to lambda calculus. In: B. Cooper, J. van Leeuwen (eds.) Alan Turing — His Work and Impact, pp. 139–143. Elsevier, Boston (2013)
6. Berger, U., Schwichtenberg, H.: An inverse of the evaluation functional for typed λ-calculus. In: G. Kahn (ed.) IEEE Symposium on Logic in Computer Science (LICS), pp. 203–211. IEEE Computer Society Press (1991)
7. Bethke, I.: Notes on partial combinatory algebras. Ph.D. thesis, Universiteit van Amsterdam (1988)
8. Boker, U., Dershowitz, N.: Comparing computational power. Logic Journal of the IGPL **4**(5), 633–647 (2006)
9. bondi programming language. bondi.it.uts.edu.au/ (2014). URL bondi.it.uts.edu.au/
10. Bove, A., Dybjer, P., Norell, U.: A brief overview of Agda — a functional language with dependent types. In: Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, TPHOLs '09, pp. 73–78. Springer-Verlag, Berlin, Heidelberg (2009). DOI 10.1007/978-3-642-03359-9-6. URL http://dx.doi.org/10.1007/978-3-642-03359-9-6
11. Brouwer, L.J.: Brouwer's Cambridge Lectures on Intuitionism. Cambridge University Press, Cambridge ; New York (1981). Edited by Dirk van Dalen
12. de Bruijn, N.: A survey of the project AUTOMATH. In: J.P. Seldin, J.R. Hindley (eds.) To H.B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism, pp. 579–606. Academic Press, Amsterdam (1980)
13. Carette, J., Stump, A.: Towards typing for small-step direct reflection. In: Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation, PEPM '12, pp. 93–96. ACM, New York, NY, USA (2012). DOI 10.1145/2103746.2103765. URL http://doi.acm.org/10.1145/2103746.2103765
14. Cellucci, C.: Proof theory and theory of meaning. In: M. Dalla Chiara (ed.) Italian Studies in the Philosophy of Science, *Boston Studies in the Philosophy of Science*, vol. 47, pp. 13–29. Springer Netherlands, Amsterdam (1981). DOI 10.1007/978-94-009-8937-5-2. URL http://dx.doi.org/10.1007/978-94-009-8937-5-2
15. Chamberlin, D.D., Boyce, R.F.: Sequel: A structured English query language. In: Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control, SIGFIDET '74, pp. 249–264. ACM, New York, NY, USA (1974). DOI 10.1145/800296.811515. URL http://doi.acm.org/10.1145/800296.811515
16. Church, A.: An unsolvable problem of elementary number theory. American Journal of Mathematics **58**(2), pp. 345–363 (1936). URL http://www.jstor.org/stable/2371045
17. Church, A.: The calculi of lambda conversion. No. 6 in Annals of Mathematics Studies. Princeton University Press, Princeton, NJ, USA (1985)
18. Cockett, J.R.B., Hofstra, P.J.W.: Introduction to Turing categories. Annals of Pure and Applied Logic **156**(2-3), 183–209 (2008)
19. Cockett, J.R.B., Hofstra, P.J.W.: Categorical simulations. Journal of Pure and Applied Algebra **214**(10), 1835–1853 (2010). DOI 10.1016/j.jpaa.2009.12.028

20. Danvy, O.: Type-directed partial evaluation. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '96, pp. 242–257. ACM, New York, NY, USA (1996). DOI 10.1145/237721.237784. URL http://doi.acm.org/10.1145/237721.237784
21. Feferman, S.: Arithmetization of metamathematics in a general setting. Fundamenta Mathematicae **49**(1), 35–92 (1960)
22. Feferman, S.: Turing's thesis. Notices of the American Mathematical Society **53**, 1200–1206 (2006)
23. Felleisen, M.: On the expressive power of programming languages. In: N. Jones (ed.) ESOP '90, *Lecture Notes in Computer Science*, vol. 432, pp. 134–151. Springer Berlin, Heidelberg (1990). DOI 10.1007/3-540-52592-0-60. URL http://dx.doi.org/10.1007/3-540-52592-0-60
24. Frege, G.: On sense and reference. In: P. Geach, M. Black (eds.) Translations from the Philosophical Writings of Gottlob Frege, pp. 56–78. Basil Blackwell, Oxford (1960). Originally published in 1892 as "Über Sinn und Bedeutung." Translated by Max Black
25. Gandy, R.: Church's thesis and principles for mechanisms. In: K.J. Barwise, H.J. Keisler, K. Kunen (eds.) The Kleene Symposium, pp. 123–148. North-Holland, Amsterdam (1980)
26. Given-Wilson, T.: An intensional concurrent faithful encoding of Turing machines. In: I. Lanese, A.L. Lafuente, A. Sokolova, H.T. Vieira (eds.) Proceedings 7th Interaction and Concurrency Experience, Berlin, Germany, 6th June 2014, *Electronic Proceedings in Theoretical Computer Science*, vol. 132, pp. 19–35. Open Publishing Association (2014)
27. Given-Wilson, T., Gorla, D., Jay, B.: A Concurrent Pattern Calculus. Logical Methods in Computer Science **10**(3) (2014). URL https://hal.inria.fr/hal-00987578
28. Goguen, J.A., Thatcher, J.W., Wagner, E.G., Wright, J.B.: Initial algebra semantics and continuous algebras. Journal of the ACM **24**(1), 68–95 (1977). DOI 10.1145/321992.321997. URL http://doi.acm.org/10.1145/321992.321997
29. Goldberg, M.: Gödelization in the lambda calculus. Information Processing Letters **75**(1), 13–16 (2000)
30. Harper, R.: Practical Foundations for Programming Languages. Cambridge University Press, New York, NY, USA (2012)
31. Hein, J.L.: Discrete Structures, Logic, and Computability, 3rd edn. Jones and Bartlett Publishers, Inc., USA (2010)
32. Hindley, J.R., Seldin, J.P.: Introduction to Combinators and [lambda]-calculus, vol. 1. CUP Archive (1986)
33. Hobbs, J.R., Rosenschein, S.J.: Making computational sense of Montague's intensional logic. Artificial Intelligence **9**, 287–306 (1978)
34. Hyland, J.M.: The effective topos. In: A. Troelstra, D. van Dalen (eds.) The L. E. J. Brouwer Centenary Symposium Proceedings of the Conference held in Noordwijkerhout, *Studies in Logic and the Foundations of Mathematics*, vol. 110, pp. 165–216. Elsevier, Amsterdam (1982). DOI http://dx.doi.org/10.1016/S0049-237X(09)70129-6. URL http://www.sciencedirect.com/science/article/pii/S0049237X09701296
35. Jay, B.: The pattern calculus. ACM Transactions on Programming Languages and Systems (TOPLAS) **26**(6), 911–937 (2004)
36. Jay, B.: Pattern Calculus: Computing with Functions and Structures. Springer (2009)
37. Jay, B.: Programs as data structures in lambda-SF-calculus. Proceedings of MFPS XXXII (2016). To appear
38. Jay, B., Given-Wilson, T.: A combinatory account of internal structure. Journal of Symbolic Logic **76**(3), 807–826 (2011)
39. Jay, B., Kesner, D.: First-class patterns. Journal of Functional Programming **19**(2), 191–225 (2009)
40. Jay, B., Palsberg, J.: Typed self-interpretation by pattern matching. In: Proceedings of the 2011 ACM Sigplan International Conference on Functional Programming, pp. 247–58 (2011)
41. Jay, B., Vergara, J.: Growing a language in pattern calculus. In: Theoretical Aspects of Software Engineering (TASE), 2013 International Symposium on, pp. 233–240. IEEE (2013)
42. Jay, B., Vergara, J.: Conflicting accounts of λ-definability. Journal of Logical and Algebraic Methods in Programming **87**, 1 – 3 (2017)

43. Jones, N.D.: Computability and Complexity: From a Programming Perspective. MIT Press, Cambridge, MA, USA (1997)
44. Katz, J.J.: The end of Milleanism: Multiple bearers, improper names, and compositional meaning. The Journal of Philosophy **98**(3), 137–166 (2000)
45. Kleene, S.C.: λ-definability and recursiveness. Duke Mathematical Journal **2**, 340–353 (1936)
46. Kleene, S.C.: Introduction to Metamathematics. Bibliotheca Mathematica. North-Holland Publishing Company, Amsterdam (1952). Co-publisher: Wolters–Noordhoff; 8th revised ed.1980.
47. Kreisel, G.: Some reasons for generalizing recursion theory. In: Logic Colloquium '69, pp. 139–198. North–Holland Publishing Co., Amsterdam (1969)
48. Landin, P.J.: The Next 700 Programming Languages. Communications of the ACM **9**, 157–166 (1966)
49. Longley, J.: Computability structures, simulations and realizability. Mathematical Structures in Computer Science **24**(02), e240201 (2014)
50. Longley, J., Normann, D.: Higher-Order Computability. Theory and Applications of Computability. Springer (2015)
51. Martin-Löf, P.: Intuitionistic Type Theory. Bibliopolis, Napoli (1984)
52. McCarthy, J.: History of Lisp. SIGPLAN Notices **13**(8), 217–223 (1978). DOI 10.1145/960118.808387. URL http://doi.acm.org/10.1145/960118.808387
53. Milner, R., Tofte, M., MacQueen, D.: The Definition of Standard ML. MIT Press, Cambridge, MA, USA (1997)
54. Mitchell, J.C.: On abstraction and the expressive power of programming languages. Science of Computer Programming **21**(2), 141–163 (1993)
55. Mitchell, J.C., Plotkin, G.D.: Abstract types have existential type. ACM Transactions on Programming Languages and Systems (TOPLAS) **10**(3), 470–502 (1988). DOI 10.1145/44501.45065. URL http://doi.acm.org/10.1145/44501.45065
56. Mogensen, T.Æ.: Gödelization in the untyped lambda-calculus. In: O. Danvy (ed.) Proceedings of the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, San Antonio, Texas, USA, January 22-23, 1999. Technical report BRICS-NS-99-1, pp. 19–24. University of Aarhus (1999)
57. Montague, R.: Towards a general theory of computability. Synthese **12**(4), 429–438 (1960)
58. Muskens, R.: Intensional models for the theory of types. Journal of Symbolic Logic **72**(1), 98–118 (2007)
59. Perlis, A.J.: Special feature: Epigrams on programming. SIGPLAN Notices **17**(9), 7–13 (1982). DOI 10.1145/947955.1083808. URL http://doi.acm.org/10.1145/947955.1083808
60. Plotkin, G.D.: The origins of structural operational semantics. Journal of Logic and Algebraic Programming **60**, 3–15 (2004)
61. Polonsky, A.: Axiomatizing the quote. In: M. Bezem (ed.) Computer Science Logic (CSL'11) — 25th International Workshop/20th Annual Conference of the EACSL, *Leibniz International Proceedings in Informatics (LIPIcs)*, vol. 12, pp. 458–469. Schloss Dagstuhl—Leibniz-Zentrum füer Informatik, Dagstuhl, Germany (2011). DOI http://dx.doi.org/10.4230/LIPIcs.CSL.2011.458. URL http://drops.dagstuhl.de/opus/volltexte/2011/3249
62. Post, E.L.: Formal reductions of the general combinatorial decision problem. American Journal of Mathematics **65**(2), pp. 197–215 (1943). URL http://www.jstor.org/stable/2371809
63. Rescorla, M.: The representational foundations of computation. Philosophia Mathematica pp. 338–366 (2015)
64. Reynolds, J.C.: The essence of ALGOL. In: P.W. O'Hearn, R.D. Tennent (eds.) ALGOL-like Languages, Volume 1, pp. 67–88. Birkhauser Boston Inc., Cambridge, MA, USA (1997). URL http://dl.acm.org/citation.cfm?id=251167.251168
65. Rogers, H.J.: Theory of Recursive Functions and Effective Computability. MIT Press, Cambridge, MA, USA (1987)
66. Scott, D.S.: Data types as lattices. SIAM Journal on Computing **5**, 522–587 (1976)
67. Scott, D.S.: λ-calculus: Then & now. In: ACM Turing Centenary Celebration, p. 9. ACM (2012)

68. Soare, R.I.: Computability and recursion. The Bulletin of Symbolic Logic **2**(3), pp. 284–321 (1996). URL http://www.jstor.org/stable/420992
69. Soare, R.I.: Chapter 1 The history and concept of computability. In: E.R. Griffor (ed.) Handbook of Computability Theory, *Studies in Logic and the Foundations of Mathematics*, vol. 140, pp. 3–36. Elsevier (1999). DOI http://dx.doi.org/10.1016/S0049-237X(99)80017-2. URL http://www.sciencedirect.com/science/article/pii/S0049237X99800172
70. Steele, G.: Growing a language. Higher-Order and Symbolic Computation **12**(3), 221–236 (1999)
71. Sussman, G.J., Steele, G.L.: SCHEME: An interpreter for extended lambda calculus. AI Memo 349, MIT Artificial Intelligence Laboratory (1975)
72. Talcott, C.: The essence of rum — A theory of intensional and extensional aspects of lisp computation. Ph.D. thesis, Stanford University (1985)
73. Terese: Term Rewriting Systems, *Cambridge Tracts in Theoretical Computer Science*, vol. 55. Cambridge University Press, Cambridge U.K (2003)
74. Tichỳ, P.: Intension in terms of Turing machines. Studia Logica **24**(1), 7–21 (1969)
75. Turing, A.M.: On computable numbers, with an application to the Entscheidungsproblem. Procedings of the London Mathematical Society **42**(2), 230–265 (1936)
76. White, G.G.: Pluralism ignored: The Church-Turing thesis and philosophical practice. In: A. Beckmann, E. Csuhaj-Varjú, K. Meer (eds.) Language, Life, Limits: 10th Conference on Computability in Europe, CiE 2014, Budapest, Hungary, June 23-27, 2014. Proceedings, pp. 373–382. Springer International Publishing, Cham (2014). DOI 10.1007/978-3-319-08019-2-39. URL http://dx.doi.org/10.1007/978-3-319-08019-2-39
77. Yan, S.Y.: An Introduction to Formal Languages and Machine Computation. World Scientific Publishing Co., Inc., River Edge, NJ, USA (2000)

Appendix B
Shasta and Fir



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.