

An Introduction to SpherePOP

Nested Circles, Typed Operators, and Reduction

Flyxion

Independent Researcher

October 2026

Contents

Preface	iv
1 Grouping and Evaluation	1
1.1 The problem with implicit grouping	1
1.2 Circles	1
1.3 Popping	2
1.4 What the notation leaves out	3
1.5 Reading a nested circle	3
2 Terms	5
2.1 Sorts, values, and operators	5
2.2 Terms	6
2.3 Positions and replacement	6
2.4 Structural induction	7
3 Popping	8
3.1 The reduction relation	8
3.2 Where redexes are	9
3.3 Normal forms and inert circles	9
3.4 Counting pops	9
4 Order, Termination, and Confluence	11
4.1 Termination	11
4.2 The diamond property	11
4.3 Unique normal forms	12
4.4 Strategies and parallel popping	13
4.5 Towers	13
5 Sorts and Admissibility	15
5.1 Sorting terms	15
5.2 Preservation	15
5.3 Progress and its limits	16
5.4 Making operators total	16

5.5	Ill-sorted terms in practice	17
6	Symbols and Substitution	18
6.1	Symbols in circles	18
6.2	Substitution	18
6.3	Substitution commutes with popping	19
6.4	Substitution does not preserve normal forms	20
6.5	Free symbols and what comes next	20
7	Colours and Images	21
7.1	The sorts	21
7.2	Three operators	21
7.3	Composing operators	22
7.4	Persistence	22
7.5	Equations between terms	22
7.6	Derived operators	23
7.7	Why the circle is not the picture	23
8	Rewriting	25
8.1	Rules	25
8.2	What changes	26
8.3	Termination of the arithmetic rules	26
8.4	Confluence by critical pairs	26
8.5	When rewriting goes wrong	27
8.6	Orthogonality and the earlier results	28
9	Neighbours and Extensions	29
9.1	Symbolic expressions	29
9.2	Many-sorted term rewriting	29
9.3	The lambda calculus	30
9.4	Evaluation order	30
9.5	Operators as commitments	31
10	A Reference Interpreter	32
10.1	Values and terms	32
10.2	Operators and admissibility	33
10.3	Popping	35
10.4	Substitution	36
10.5	Rewriting	36
10.6	A test run	37
A	Solutions to Selected Exercises	39
B	Summary of Notation and Rules	42

References and Further Reading	44
---------------------------------------	-----------

Preface

This book is an introduction to SpherePOP, a notation for writing computations as nested circles. Every circle names an operator and encloses its operands, and a computation proceeds by popping circles whose contents are ready, each popped circle being replaced by the value that its operator produces. The notation is small enough to be specified completely in a few pages, and yet it is rich enough to carry arithmetic, symbolic expressions, operations on pictures, and systems of rewriting rules. For that reason it makes a good setting in which to learn several ideas that recur throughout the study of programming languages: the difference between syntax and evaluation, the role of explicit grouping, the properties that make an evaluation order irrelevant, the way a system of sorts separates meaningful terms from meaningless ones, and the point at which rewriting becomes difficult.

The book is written for readers who know ordinary arithmetic notation, who have met proof by induction, and who are willing to follow a definition through a few examples before trusting it. No prior knowledge of lambda calculus, term rewriting, or type theory is assumed, although readers who have met these subjects will recognize familiar structures at several points and are told where to look for the standard treatments. Programming experience is helpful in the last chapter, where a complete reference interpreter of under three hundred lines is presented and its output is reproduced.

Organization. The first chapter motivates explicit grouping and introduces the circle drawings informally. Chapters 2 and 3 give the formal syntax and the single reduction rule. Chapter 4 proves the central structural results, namely that every computation terminates and that the order in which circles are popped does not affect the outcome. Chapter 5 introduces sorts and shows exactly where a well-sorted term can still fail to compute. Chapter 6 treats symbols and substitution, which allow a term to be partly evaluated before all of its inputs are known. Chapter 7 shows how the same machinery handles pictures and colours. Chapter 8 lets operators return circles instead of values, turning the notation into a rewriting system, and uses the classical criteria of termination and critical pairs to decide when the result is well behaved. Chapter 9 compares the notation with several neighbouring formalisms and indicates directions for extension. Chapter 10 presents the interpreter. The appendices contain solutions to selected exercises and a summary of notation.

Conventions. A circle is drawn as a closed curve, but in running text it is written $\langle o\ t_1 \cdots t_n \rangle$, with the operator name first and the operands following. The two forms are

interchangeable. Operator names are set in sans-serif type, for example `add` and `rotate`. Definitions, theorems, and examples are numbered within each chapter, and exercises are numbered by chapter. Exercises whose number is followed by a dagger have a solution in Appendix A. Each proof ends with a square, and each example ends with a small triangle.

Using the interpreter. All numerical examples in the text have been computed with the interpreter of Chapter 10, and the transcript of a complete test run is reproduced there. Readers are encouraged to run the program, to change the examples, and to attempt the extensions suggested in the exercises at the end of that chapter.

Chapter 1

Grouping and Evaluation

1.1 The problem with implicit grouping

Ordinary mathematical notation is compact because it leaves a great deal unsaid. The string $2 + 3 \times 4$ does not state which operation is to be performed first. A reader supplies the answer from a convention, according to which multiplication binds more tightly than addition, and obtains 14 and not 20. The convention is so thoroughly learned that it is rarely noticed, and yet it is part of the meaning of the expression. A reader who did not know it, or who applied a different one, would compute a different number from the same marks on the page.

Conventions of this kind multiply as notation grows. Subtraction and division associate to the left, so that $8 - 3 - 2$ is read as $(8 - 3) - 2 = 3$ and not as $8 - (3 - 2) = 7$. Exponentiation is conventionally read from the top down, so that a^{b^c} means $a^{(b^c)}$ and not $(a^b)^c$. The two readings are not the same in general. With $a = 2$, $b = 3$, $c = 2$ the first gives $2^9 = 512$ and the second gives $8^2 = 64$. Programming languages add further layers, with tables of operator precedence that may run to a dozen levels, and with different languages making different choices about the same symbols.

There are two standard responses to this situation. One is to fully parenthesize, writing $2 + (3 \times 4)$ and $a^{(b^c)}$, so that nothing is left to convention. The other is to change the order of writing, as in the prefix notation $(+ 2 (* 3 4))$ used by Lisp-family languages, where the operator comes first and the grouping is carried entirely by brackets. Both responses make the structure of the expression explicit. Both also make it harder to read at a glance, because a bracket is a small mark that must be matched against a distant partner.

1.2 Circles

SpherePOP takes the second response and replaces the brackets by a visual boundary. An expression is drawn as a closed curve with the name of an operator written on it, and the operands of the operator are drawn inside. An operand may itself be a circle. A circle that encloses nothing but plain values, such as numbers, is the simplest kind of computation, and a circle that encloses other circles describes a computation whose inner parts must be

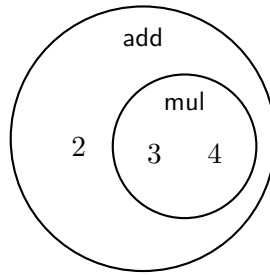


Figure 1.1: The expression $2 + 3 \times 4$ as nested circles. The inner circle labelled `mul` encloses the operands 3 and 4, and is itself an operand of the outer circle labelled `add`.

completed before the outer one can be.

Figure 1.1 shows $2 + 3 \times 4$ in this form. Nothing in the drawing refers to a precedence table. The multiplication is inside the addition because the drawing puts it there, and the drawing for $(2 + 3) \times 4$ would put the addition inside the multiplication. The two expressions are different pictures, and so the question of which one is meant never arises.

In running text a circle is written $\langle o \ t_1 \cdots t_n \rangle$, where o is the operator and $t_1, \dots, t_n$ are the enclosed operands, each of which is a value or another circle. The expression of Figure 1.1 is written $\langle \text{add } 2 \ \langle \text{mul } 3 \ 4 \rangle \rangle$. The angle brackets are a typographical stand-in for the drawn boundary and carry no meaning of their own beyond marking where a circle begins and ends.

1.3 Popping

A circle is a description of a computation, and it is carried out by popping. A circle may be popped when everything inside it is already a value. Popping replaces the entire circle by the value that its operator produces from the enclosed operands. For the circle of Figure 1.1 only the inner circle can be popped at first, since the outer circle encloses an operand that is not yet a value. Popping the inner circle replaces it by 12, which leaves the circle $\langle \text{add } 2 \ 12 \rangle$, and popping that circle gives 14. Written as a sequence,

$$\langle \text{add } 2 \ \langle \text{mul } 3 \ 4 \rangle \rangle \longrightarrow \langle \text{add } 2 \ 12 \rangle \longrightarrow 14.$$

Each arrow is one pop, and each pop removes exactly one circle from the expression. These two facts, that a pop needs all operands to be values and that a pop removes exactly one circle, are the foundation of the results proved in Chapter 4.

Several circles may be ready at the same time. In $\langle \text{add } \langle \text{mul } 2 \ 3 \rangle \ \langle \text{mul } 4 \ 5 \rangle \rangle$ both multiplications can be popped, and the notation does not say which one comes first. The choice turns out to be irrelevant: whichever is popped first, the other is still ready afterward, and the final value is 26 either way. Making this precise, and proving it, is the main business of Chapter 4.

1.4 What the notation leaves out

It is worth being explicit about what SpherePOP does not provide, because the omissions are deliberate. There are no precedence rules, since grouping is always drawn. There is no implicit conversion between kinds of value, since a circle either accepts the operands it is given or does not. There is no assignment, no loop, and no hidden state, since popping only replaces a circle by a value. And there is no fixed list of operators, since the notation does not care what an operator does, only that it takes some values in and gives a value back.

That last point is what makes the notation more than a curiosity of arithmetic. The operators in this book include the usual numerical ones, but also operators that rotate a picture, recolour it, and place two pictures side by side. The same circles, the same popping rule, and the same theorems apply to all of them. An expression such as $\langle \text{rotate } \langle \text{recolor } I \text{ red} \rangle 90 \rangle$ is read exactly as the arithmetic expression above: the inner circle is popped first, yielding a picture, and the outer circle then rotates that picture. The distinction between numbers and pictures is handled by sorts, which are introduced in Chapter 5, and not by changing the evaluation rule.

A second kind of generality concerns what is allowed inside a circle. A circle may enclose a symbol, a name standing for a value that is not yet known. Such a circle cannot be popped, because its operator has nothing to compute with, and so it simply stays in the expression. This is how the notation keeps an expression such as x^2 unevaluated until x is supplied, and it is the subject of Chapter 6.

1.5 Reading a nested circle

A useful habit is to read a nested expression from the inside out. The innermost circles, the ones that enclose no other circle, are the first candidates for popping. After they are replaced, a new set of innermost circles appears, and the process repeats. The number of times the process repeats is the greatest depth of nesting in the original expression, a quantity that reappears in Chapter 4 as the number of rounds needed if all ready circles are popped simultaneously.

Example 1.1. Consider $\langle \text{pow } \langle \text{add } 1 \ 1 \rangle \langle \text{mul } 2 \langle \text{sub } 5 \ 2 \rangle \rangle \rangle$. The innermost circles are $\langle \text{add } 1 \ 1 \rangle$ and $\langle \text{sub } 5 \ 2 \rangle$, and they pop to 2 and 3. The expression is now $\langle \text{pow } 2 \langle \text{mul } 2 \ 3 \rangle \rangle$, whose only innermost circle pops to 6, leaving $\langle \text{pow } 2 \ 6 \rangle$, which pops to 64. The expression contains four circles, and four pops were needed. $\triangleright$

Exercises

Exercise 1.1 ($\dagger$). Write $2 + 3 \times 4$ and $(2 + 3) \times 4$ as circle expressions in the linear notation, and compute both values.

Exercise 1.2 ($\dagger$). Write the two groupings of a^{b^c} as circle expressions using the operator **pow**. Compute both for $(a, b, c) = (2, 3, 2)$.

Exercise 1.3 ($\dagger$). Give every pop sequence for $\langle \text{add } \langle \text{mul } 2\ 3 \rangle \langle \text{sub } 9\ 4 \rangle \rangle$. How many are there, and do they reach the same value?

Exercise 1.4. Convert $((1 + 2) \times (3 + 4)) - 5$ to a circle expression, draw it as nested circles, and list the circles in an order in which they could be popped.

Exercise 1.5. Explain in a few sentences why the expression $8 - 3 - 2$ requires a convention in ordinary notation but has two distinct circle expressions. Write both.

Chapter 2

Terms

This chapter gives the precise syntax of the notation. The definitions are short, but they are used throughout the rest of the book, and the habit of reasoning by structural induction, which is introduced here, is the main proof technique of the chapters that follow.

2.1 Sorts, values, and operators

Every value has a sort, which says what kind of thing it is. The core fragment of the notation uses four sorts: **Num** for integers, **Color** for colour names, **Image** for finite rectangular grids of colours, and **Sym** for symbols, which are names of values not yet known. The set of sorts is not closed. A different application may add sorts for strings, for booleans, or for any other kind of data, and nothing in the development below depends on the particular four chosen here. For each sort σ there is a set V_σ of values of that sort, and the set of all values is the union of these sets.

An *operator* o has a name, a *signature* $\sigma_1 \times \cdots \times \sigma_n \rightarrow \sigma$ giving the sorts of its operands and of its result, and a partial function δ_o from tuples of values to values. The number n is the arity of the operator. The function δ_o is the meaning of the operator. It is partial because some operand tuples with the right sorts are nonetheless unsuitable. Raising an integer to a negative power does not give an integer, for example, and placing two pictures of different heights side by side does not give a rectangle. The tuples on which δ_o is defined are called the *admissible* operands of o , and the set of admissible tuples is written $\text{dom}(\delta_o)$.

Two requirements are placed on operators throughout the book. First, δ_o is a function, so that the same operands always give the same result. This is the property that makes the notation deterministic. Second, if $(v_1, \dots, v_n) \in \text{dom}(\delta_o)$ then the sorts of $v_1, \dots, v_n$ are those of the signature and $\delta_o(v_1, \dots, v_n)$ is a value whose sort is the result sort of the signature. Chapter 8 relaxes the requirement that the result be a value.

Example 2.1. The operator **add** has signature $\text{Num} \times \text{Num} \rightarrow \text{Num}$, and $\delta_{\text{add}}(a, b) = a + b$ for all integers. The operator **pow** has the same signature, but $\delta_{\text{pow}}(a, b) = a^b$ is defined only when $b \geq 0$, so that the result is an integer. The operator **rotate** has signature $\text{Image} \times \text{Num} \rightarrow \text{Image}$, and it is admissible only when the number is a multiple of 90. ▷

2.2 Terms

Definition 2.2 (Terms). The set of *terms* is the smallest set such that

- (i) every value is a term, and
- (ii) if o is an operator of arity n and $t_1, \dots, t_n$ are terms, then $\langle o \ t_1 \cdots t_n \rangle$ is a term, called a *circle*.

A term that is not a circle is a value. A circle is drawn as a closed curve labelled o enclosing the drawings of $t_1, \dots, t_n$, and the linear notation $\langle o \ t_1 \cdots t_n \rangle$ is an abbreviation for that drawing. Because the brackets are explicit and the arity of every operator is fixed, every term has exactly one way of being read: there is no question of where a circle ends or which operands belong to it. This property, often called unique readability, is not a theorem to be proved but a consequence of the definition, and it is the reason that no precedence rules are needed.

Two quantities of a term are used repeatedly. The *size* of a term is the number of circles it contains, and the *depth* is the greatest number of circles that lie one inside another along any path from the outside of the term to a value. Formally,

$$\begin{aligned} \text{size}(v) &= 0, & \text{size}\langle o \ t_1 \cdots t_n \rangle &= 1 + \sum_{i=1}^n \text{size}(t_i), \\ \text{depth}(v) &= 0, & \text{depth}\langle o \ t_1 \cdots t_n \rangle &= 1 + \max_{1 \leq i \leq n} \text{depth}(t_i). \end{aligned}$$

For an operator of arity zero the maximum is taken over the empty set and is read as 0, so that a nullary circle has depth 1. For every term $\text{depth}(t) \leq \text{size}(t)$, because a chain of nested circles is a set of distinct circles.

Example 2.3. Let $t = \langle \text{add} \ \langle \text{mul} \ 2 \ 3 \rangle \ \langle \text{pow} \ \langle \text{sub} \ 5 \ 1 \rangle \ 2 \rangle \rangle$. The circles are **add**, **mul**, **pow**, and **sub**, so $\text{size}(t) = 4$. The circle **sub** has depth 1, the circle **pow** containing it has depth 2, the circle **mul** has depth 1, and the outer circle has depth $1 + \max(1, 2) = 3$. Hence $\text{depth}(t) = 3$.
▷

2.3 Positions and replacement

To talk about a particular circle inside a larger term it is convenient to give it an address. A *position* is a finite sequence of positive integers. The empty sequence ε is a position of every term and denotes the whole term. If $t = \langle o \ t_1 \cdots t_n \rangle$ and $p = i \cdot q$ with $1 \leq i \leq n$, then p is a position of t and the subterm of t at p is the subterm of t_i at q . Writing $t|_p$ for the subterm at p , and $t[p \leftarrow s]$ for the term obtained from t by replacing the subterm at p by s , one has the usual identities, for example $t[p \leftarrow t|_p] = t$.

Two positions are *disjoint* if neither is a prefix of the other. If p and q are disjoint positions of t , then replacements at p and at q do not interfere, and

$$(t[p \leftarrow s])[q \leftarrow s'] = (t[q \leftarrow s'])[p \leftarrow s].$$

This elementary fact is the heart of the argument for order independence in Chapter 4.

Example 2.4. In $t = \langle \text{add } 1 \langle \text{mul } 2 \ 3 \rangle \rangle$ the positions are ε , 1, 2, $2 \cdot 1$, and $2 \cdot 2$. The subterm at 2 is $\langle \text{mul } 2 \ 3 \rangle$, and $t[2 \leftarrow 6] = \langle \text{add } 1 \ 6 \rangle$. The positions 1 and 2 are disjoint, but 2 and $2 \cdot 1$ are not. $\triangleright$

2.4 Structural induction

Because terms are defined as the smallest set closed under two constructions, one may prove that every term has a property P by proving two things: that every value has P , and that a circle has P whenever all of its operands do. This is the principle of structural induction. A parallel principle, structural recursion, justifies definitions such as those of size and depth, in which the value for a circle is computed from the values for its operands.

Proposition 2.5. *In a term every circle of which has exactly two operands, the number of value occurrences is $\text{size}(t) + 1$.*

Proof. By structural induction. A value is a term with one value occurrence and size 0, and $1 = 0 + 1$. For a circle $\langle o \ t_1 \ t_2 \rangle$ the number of value occurrences is the sum of those in t_1 and t_2 , which by hypothesis is $(\text{size}(t_1) + 1) + (\text{size}(t_2) + 1)$. The size of the circle is $1 + \text{size}(t_1) + \text{size}(t_2)$, and the number of value occurrences is therefore $\text{size}(t) + 1$. $\square$

Exercises

Exercise 2.1 ($\dagger$). Compute the size and depth of $\langle \text{add } \langle \text{mul } 2 \ 3 \rangle \langle \text{pow } \langle \text{sub } 5 \ 1 \rangle \ 2 \rangle \rangle$ and of the term obtained by replacing 2 in the last operand by $\langle \text{add } 1 \ 1 \rangle$.

Exercise 2.2 ($\dagger$). Prove that in a term all of whose circles are binary, $\text{size}(t) \leq 2^{\text{depth}(t)} - 1$.

Exercise 2.3 ($\dagger$). List all positions of $\langle \text{add } \langle \text{mul } 2 \ 3 \rangle \ 4 \rangle$ and compute $t[1 \cdot 2 \leftarrow 7]$.

Exercise 2.4. Prove, using structural induction, that $\text{depth}(t) \leq \text{size}(t)$ for every term, and give a term for which equality holds and a term for which the inequality is strict.

Exercise 2.5. Show that the disjointness identity displayed above can fail when p is a prefix of q . Which side, if either, is defined?

Chapter 3

Popping

3.1 The reduction relation

A computation in SpherePOP is a sequence of pops. A single pop is described by a relation $t \longrightarrow t'$ between terms, read *t pops to t'*, which is defined by two rules. The first, called (Pop), states what happens to a circle that is ready. The second, called (Arg), lets a pop happen anywhere inside a term.

Definition 3.1 (Popping). A circle $\langle o \ v_1 \cdots v_n \rangle$ whose operands are all values and satisfy $(v_1, \dots, v_n) \in \text{dom}(\delta_o)$ is called a *redex*. The relation $\longrightarrow$ is the smallest relation satisfying

$$\begin{aligned} \langle o \ v_1 \cdots v_n \rangle &\longrightarrow \delta_o(v_1, \dots, v_n) && \text{if } (v_1, \dots, v_n) \in \text{dom}(\delta_o), && \text{(Pop)} \\ \langle o \ t_1 \cdots t_i \cdots t_n \rangle &\longrightarrow \langle o \ t_1 \cdots t'_i \cdots t_n \rangle && \text{if } t_i \longrightarrow t'_i. && \text{(Arg)} \end{aligned}$$

The reflexive and transitive closure of $\longrightarrow$ is written $\longrightarrow^*$.

The first rule needs no commentary beyond the observation that it applies only when every operand is already a value. The second rule states that popping is allowed at any depth. Taken together they say that a pop replaces, at some position p of the term, a redex $t|_p$ by the value $\delta_o(v_1, \dots, v_n)$, so that $t' = t[p \leftarrow \delta_o(v_1, \dots, v_n)]$. This description is often more convenient than the pair of rules, and the two are equivalent by a routine induction on the position p .

Example 3.2. Let $t = \langle \text{add } \langle \text{mul } 2 \ 3 \rangle \ \langle \text{mul } 4 \ 5 \rangle \rangle$. The redexes of t are the two multiplications, at positions 1 and 2. Popping the first gives $\langle \text{add } 6 \ \langle \text{mul } 4 \ 5 \rangle \rangle$ and popping the second gives $\langle \text{add } \langle \text{mul } 2 \ 3 \rangle \ 20 \rangle$. Each of these has exactly one redex left, and popping it in either case gives $\langle \text{add } 6 \ 20 \rangle$, which pops to 26. The two derivations are

$$\begin{aligned} t &\longrightarrow \langle \text{add } 6 \ \langle \text{mul } 4 \ 5 \rangle \rangle \longrightarrow \langle \text{add } 6 \ 20 \rangle \longrightarrow 26, \\ t &\longrightarrow \langle \text{add } \langle \text{mul } 2 \ 3 \rangle \ 20 \rangle \longrightarrow \langle \text{add } 6 \ 20 \rangle \longrightarrow 26. \end{aligned} \quad \triangleright$$

3.2 Where redexes are

A redex is a circle all of whose operands are values. Such a circle contains no other circle as a subterm, and this observation gives a clean description of where redexes can occur. Call a circle *innermost* if none of its operands is a circle.

Lemma 3.3. *The redexes of a term are exactly its innermost circles whose operands are admissible. Every term that is not a value has at least one innermost circle.*

Proof. A redex has only values as operands, so it is innermost, and by definition its operands are admissible. Conversely an innermost circle has only values as operands, and if they are admissible it is a redex. For the second claim, choose in a non-value term a circle of greatest depth along some path from the outside. If one of its operands were a circle, that circle would lie deeper still, contradicting the choice. Hence all operands are values. $\square$

3.3 Normal forms and inert circles

A term is in *normal form* if no pop applies to it. Every value is in normal form. A term that is in normal form but is not a value is called *blocked*. By Lemma 3.3, a term is in normal form exactly when every innermost circle in it has inadmissible operands, and a blocked term is one in which at least one circle is innermost and inadmissible, so that popping stops with a circle still present.

A circle with inadmissible operands is called *inert*. Inert circles are not errors. Nothing is signalled and nothing is discarded. The circle simply remains in the term, and every circle that encloses it remains too, since an enclosing circle cannot be popped until all of its operands are values. This is a feature of the design. An inert circle is visible, it can be inspected, and it can sometimes be repaired later, as when a symbol inside it is replaced by a number in Chapter 6.

Example 3.4. The circle $\langle \text{pow } 2 \text{ } -1 \rangle$ is inert, because δ_{pow} is not defined for a negative exponent. The term $\langle \text{add } 1 \text{ } \langle \text{pow } 2 \text{ } -1 \rangle \rangle$ is therefore blocked: its only innermost circle is inert, and the outer circle is not a redex because one of its operands is a circle and not a value. Both circles remain after popping stops. $\triangleright$

3.4 Counting pops

The most important feature of a pop is that it removes exactly one circle. A redex $\langle o \ v_1 \cdots v_n \rangle$ is one circle, and it is replaced by a value, which is no circle at all. No other part of the term changes.

Proposition 3.5. *If $t \longrightarrow t'$ then $\text{size}(t') = \text{size}(t) - 1$.*

Proof. Write $t' = t[p \leftarrow v]$ where $t|_p$ is a redex and v is a value. The redex contributes $1 + \sum \text{size}(v_i) = 1$ to the size of t , since its operands are values, and v contributes 0 to the size of t' . The remaining circles of t and t' are the same. $\square$

Corollary 3.6. *If $t \longrightarrow^* t'$ in k steps then $\text{size}(t') = \text{size}(t) - k$. In particular a term that pops to a value does so in exactly $\text{size}(t)$ steps.*

The corollary has a pleasant practical reading. The cost of evaluating a term to a value, counted in pops, is its size and nothing else. It does not depend on the order of popping, on which of several ready circles is chosen first, or on the values involved. A term whose operators are expensive may of course be slow to evaluate, but the number of pops is fixed in advance by the shape of the term.

Exercises

Exercise 3.1 (†). List the redexes of $\langle \text{add } \langle \text{mul } 2 \ 3 \rangle \ \langle \text{mul } 4 \ \langle \text{sub } 9 \ 4 \rangle \rangle \rangle$ and give all pop sequences that reach a value.

Exercise 3.2 (†). Evaluate $\langle \text{pow } \langle \text{add } 1 \ 1 \rangle \ \langle \text{mul } 2 \ \langle \text{sub } 5 \ 2 \rangle \rangle \rangle$ to a value, state the number of pops, and compare it with the size of the term.

Exercise 3.3 (†). Is $\langle \text{mul } 3 \ \langle \text{pow } 0 \ \langle \text{sub } 2 \ 3 \rangle \rangle \rangle$ a value, a blocked term, or a term with a redex? Evaluate as far as possible and list the circles that remain.

Exercise 3.4. Show that if t is blocked then t contains an innermost circle c whose operands are inadmissible, and that every circle containing c is also present in every term to which t pops.

Exercise 3.5. Prove that the (Arg) rule is redundant if the relation is instead defined by the statement “ $t \longrightarrow t[p \leftarrow \delta_o(v_1, \dots, v_n)]$ whenever $t|_p = \langle o \ v_1 \cdots v_n \rangle$ is a redex.” That is, show that the two definitions give the same relation.

Chapter 4

Order, Termination, and Confluence

A definition of evaluation is only trustworthy if it answers three questions. Does evaluation always stop? If it stops, does it give one answer, or can the answer depend on arbitrary choices made along the way? And how much work does it take? For the notation of the preceding chapters all three questions have complete and simple answers, and this chapter proves them. The proofs are short, but they are worth reading closely, because the same three questions are asked of every system in later chapters and the answers there are less comfortable.

4.1 Termination

Theorem 4.1 (Termination). *There is no infinite sequence $t_0 \longrightarrow t_1 \longrightarrow t_2 \longrightarrow \dots$. Every sequence of pops starting from a term t has at most $\text{size}(t)$ steps.*

Proof. By Proposition 3.5, each pop reduces the size by exactly one. Sizes are non-negative integers, so a sequence starting from size n cannot have more than n steps. $\square$

The argument uses nothing about the operators except that popping replaces a circle by a value. It is therefore insensitive to what the operators compute, and it would survive the addition of any number of new sorts and operators, provided each new operator returns a value. In Chapter 8 operators are allowed to return circles, and this is exactly the point at which the argument breaks.

4.2 The diamond property

Suppose two different circles of a term are both ready. Popping one leaves the other ready, and the order of the two pops cannot be detected afterward. The following lemma states this precisely.

Lemma 4.2 (Diamond property). *If $t \longrightarrow t_1$ and $t \longrightarrow t_2$ then either $t_1 = t_2$, or there is a term t_3 with $t_1 \longrightarrow t_3$ and $t_2 \longrightarrow t_3$.*

Proof. Let the first pop replace the redex at position p and the second the redex at position q . If $p = q$ then $t_1 = t_2$, because δ_o is a function. Otherwise suppose, for a contradiction, that p is a proper prefix of q . Then the redex at p contains the redex at q as a proper subterm. But a redex has only values as operands, and a circle is not a value, so a redex contains no circle as a proper subterm. The same argument excludes the case that q is a proper prefix of p . Hence p and q are disjoint. Write v_p and v_q for the values that the two redexes pop to, so that $t_1 = t[p \leftarrow v_p]$ and $t_2 = t[q \leftarrow v_q]$. Because the positions are disjoint, the redex at q is still present in t_1 and the redex at p is still present in t_2 , and popping them gives

$$t_1[q \leftarrow v_q] = t[p \leftarrow v_p][q \leftarrow v_q] = t[q \leftarrow v_q][p \leftarrow v_p] = t_2[p \leftarrow v_p],$$

the middle equality being the disjointness identity of Chapter 2. Taking t_3 to be this common term completes the proof. $\square$

4.3 Unique normal forms

Theorem 4.3 (Unique normal form). *Every term t has exactly one normal form $\text{nf}(t)$, and every maximal sequence of pops starting from t ends in $\text{nf}(t)$.*

Proof. By Theorem 4.1 every sequence of pops from t can be extended until no pop applies, so a normal form exists. For uniqueness, argue by strong induction on $\text{size}(t)$ that all maximal sequences from t end in the same term. If t is in normal form, the only maximal sequence is the empty one. Otherwise take two maximal sequences, the first beginning $t \longrightarrow a$ and the second beginning $t \longrightarrow b$. If $a = b$ the induction hypothesis applies to a , since $\text{size}(a) < \text{size}(t)$, and the two sequences end in the same term. If $a \neq b$ then Lemma 4.2 provides c with $a \longrightarrow c$ and $b \longrightarrow c$. Let n be the end of some maximal sequence from c . Then $a \longrightarrow c \longrightarrow^* n$ is a maximal sequence from a , so by the induction hypothesis the first sequence ends in n , and in the same way $b \longrightarrow c \longrightarrow^* n$ shows that the second sequence ends in n as well. $\square$

Corollary 4.4 (Order independence). *All maximal sequences of pops from t have the same length, namely $\text{size}(t) - \text{size}(\text{nf}(t))$.*

Proof. By Theorem 4.3 all of them end in $\text{nf}(t)$, and by Corollary 3.6 the number of steps is determined by the sizes of the first and last terms. $\square$

The practical meaning is considerable. A program written in this notation has no evaluation order to get wrong. An implementation may choose the leftmost ready circle, the rightmost, a random one, or all of them at once, and it will obtain the same normal form in the same number of pops. A reader, human or mechanical, who works through the term in a different order from the author's will not disagree about the result. If two readers do disagree, one of them has made an arithmetic mistake.

4.4 Strategies and parallel popping

A *strategy* is a rule for choosing which ready circle to pop. Theorem 4.3 says that every strategy is correct. Strategies differ only in how they use time. The simplest is the *leftmost* strategy, which pops the first redex found in a left-to-right scan, and it is the one used by the interpreter of Chapter 10. A more interesting one is *parallel popping*, which pops every redex of the current term in a single round. Lemma 4.2 guarantees that the redexes do not interfere, so a round is well defined and gives a term that is reachable by ordinary pops.

Proposition 4.5 (Parallel rounds). *If every circle that arises during evaluation is admissible, then parallel popping evaluates t to a value in exactly $\text{depth}(t)$ rounds.*

Proof. Let t be a term that is not a value, and let t' be the term obtained by one round of parallel popping, in which every innermost circle is replaced by a value. It suffices to show $\text{depth}(t') = \text{depth}(t) - 1$, since a term of depth 0 is a value. The claim is proved by structural induction on t . For a circle all of whose operands are values the depth is 1 and the circle is replaced by a value of depth 0. For a circle $\langle o \ t_1 \cdots t_n \rangle$ with some operand not a value, each operand t_i of depth $d_i \geq 1$ becomes a term of depth $d_i - 1$ by the induction hypothesis, while operands that are values stay as they are with depth 0. The greatest operand depth $m = \max d_i \geq 1$ therefore becomes $m - 1$, and the depth of the circle changes from $1 + m$ to $1 + (m - 1) = m$, a decrease of exactly one. $\square$

The proposition shows that the number of rounds, the parallel time, is governed by depth and not by size. A balanced expression of depth d has $2^d - 1$ binary circles, and parallel popping needs only d rounds, whereas sequential popping needs $2^d - 1$ pops. A tall tower of nested circles, by contrast, has depth equal to its size, and parallelism gives no benefit at all.

Example 4.6. Let E_3 be the balanced expression

$$\langle \text{add} \langle \text{add} \langle \text{add} \ 1 \ 1 \rangle \langle \text{add} \ 1 \ 1 \rangle \rangle \langle \text{add} \langle \text{add} \ 1 \ 1 \rangle \langle \text{add} \ 1 \ 1 \rangle \rangle \rangle.$$

It has seven circles and depth three. Sequential popping takes seven pops, and parallel popping takes three rounds: the four innermost circles at once, then the two above them, then the outermost. The value is 8. $\triangleright$

4.5 Towers

A particularly instructive family of terms is the family of right-nested towers of exponentiation. Writing x for a fixed operand, let

$$T_1 = x, \quad T_{n+1} = \langle \text{pow } x \ T_n \rangle.$$

Then $T_2 = \langle \text{pow } x \ x \rangle$, $T_3 = \langle \text{pow } x \ \langle \text{pow } x \ x \rangle \rangle$, and in general T_n has $n - 1$ circles, each enclosing the next, so $\text{size}(T_n) = \text{depth}(T_n) = n - 1$. If x is the number 2 the sequence of values is 2, 4, 16, 65536, and then 2^{65536} , and each is obtained by $n - 1$ pops. The towers grow

in value with extraordinary speed, while they grow in size, and therefore in cost, only linearly. They also show that the depth bound of Proposition 4.5 cannot be improved: nothing can be popped in parallel in a tower, because at every stage there is only one ready circle. By Corollary 3.6, the number of pops needed to reach the value is always $n - 1$, no matter how large the value turns out to be.

The same operands arranged the other way, as the left-nested term $\langle \text{pow } \langle \text{pow } \langle \text{pow } 2 \ 2 \rangle \ 2 \rangle \ 2 \rangle$, give a very different value, namely 256 and not 65536. The notation makes the two groupings different terms from the outset, and so there is nothing to disambiguate.

Exercises

Exercise 4.1 ($\dagger$). For $t = \langle \text{add } \langle \text{mul } 2 \ 3 \rangle \ \langle \text{mul } 4 \ 5 \rangle \rangle$ write out the two terms t_1, t_2 reached by popping each redex and exhibit the common term t_3 required by Lemma 4.2.

Exercise 4.2 ($\dagger$). A term has size 15 and is a balanced binary expression. How many sequential pops and how many parallel rounds does it need?

Exercise 4.3 ($\dagger$). Compute the value of $\langle \text{pow } 3 \ \langle \text{pow } 3 \ 3 \rangle \rangle$ and of $\langle \text{pow } \langle \text{pow } 3 \ 3 \rangle \ 3 \rangle$, and state the number of pops for each.

Exercise 4.4. The proof of Theorem 4.3 uses induction on size. Where does it use that δ_o is a function? Where would it fail if δ_o were allowed to return different values on different occasions?

Exercise 4.5. Prove that any set of redexes in a term may be popped in any order, by induction on the number of redexes, using Lemma 4.2. What is the relation to Proposition 4.5?

Exercise 4.6. Find a term t with $\text{size}(t) = 5$ for which parallel popping takes 5 rounds and a term with $\text{size}(t) = 5$ for which it takes 3.

Chapter 5

Sorts and Admissibility

The previous chapters allowed an operator to refuse a tuple of operands without saying why. In practice the refusals fall into two quite different kinds. Some are failures of kind: a colour has been supplied where a number is wanted. Others are failures of detail: the operands are numbers, as required, but one of them is a negative exponent. This chapter separates the two. Sorts catch the first kind before any popping begins, and a precise accounting of the second kind shows exactly what a sort system does and does not guarantee.

5.1 Sorting terms

Each operator has a signature $\sigma_1 \times \cdots \times \sigma_n \rightarrow \sigma$. A term is *well-sorted* with sort τ , written $\vdash t : \tau$, according to the following rules.

$$\vdash v : \text{sort}(v) \quad (\text{Val})$$

$$\frac{\vdash t_1 : \sigma_1 \quad \cdots \quad \vdash t_n : \sigma_n \quad o : \sigma_1 \times \cdots \times \sigma_n \rightarrow \sigma}{\vdash \langle o \ t_1 \cdots t_n \rangle : \sigma} \quad (\text{Circ})$$

A term is *well-sorted* if $\vdash t : \tau$ for some τ , and the sort, when it exists, is unique, since each rule has only one conclusion for a given term. Sorting is checked from the inside out and takes a number of steps proportional to the size of the term. It does not require popping anything.

Example 5.1. The term $\langle \text{rotate } \langle \text{recolor } I \text{ red} \rangle 90 \rangle$ is well-sorted with sort `Image`, if I is an image: the inner circle has sort `Image` because $I : \text{Image}$ and $\text{red} : \text{Color}$, and the outer circle has sort `Image` because its operands have sorts `Image` and `Num`. The term $\langle \text{rotate } \langle \text{add } 1 \ 2 \rangle I \rangle$ is not well-sorted. Its first operand has sort `Num` where `Image` is required, and its second has sort `Image` where `Num` is required. $\triangleright$

5.2 Preservation

Popping must not change the sort of a term, or the sort system would tell the reader nothing about what to expect from evaluation. The requirement is stated for operators in the following

form: whenever $(v_1, \dots, v_n) \in \text{dom}(\delta_o)$, the value $\delta_o(v_1, \dots, v_n)$ has the result sort of the signature of o . This is part of the definition of an operator given in Chapter 2.

Theorem 5.2 (Preservation). *If $\vdash t : \tau$ and $t \longrightarrow t'$ then $\vdash t' : \tau$.*

Proof. By induction on the derivation of $t \longrightarrow t'$. If the last rule is (Pop) then $t = \langle o \ v_1 \cdots v_n \rangle$ with admissible operands, and by the sorting rule the sort τ is the result sort of o . The term $t' = \delta_o(v_1, \dots, v_n)$ is a value of exactly that sort. If the last rule is (Arg), then $t = \langle o \ t_1 \cdots t_i \cdots t_n \rangle$ and t' replaces t_i by a term t'_i with $t_i \longrightarrow t'_i$. The sort of t_i is the i -th operand sort of o because t is well-sorted, and by the induction hypothesis t'_i has the same sort. The sorting rule therefore applies to t' with the same result. $\square$

5.3 Progress and its limits

A well-sorted term may still fail to evaluate. To find out how, one needs the converse of preservation, which says that a well-sorted term is either a value or can be popped. That statement is false in general, and the way in which it fails is informative.

Theorem 5.3 (Progress, in two parts). *Let t be well-sorted and not a value.*

- (a) *If every operator is total on its signature, meaning that δ_o is defined on every tuple of values of the right sorts, then t can be popped.*
- (b) *In general, if t is in normal form, then t contains an innermost circle whose operands have the right sorts but are not in $\text{dom}(\delta_o)$.*

Proof. By Lemma 3.3 the term t contains an innermost circle $c = \langle o \ v_1 \cdots v_n \rangle$. Since t is well-sorted, so is c , and therefore each v_i has the i -th operand sort of o . If δ_o is total on its signature the tuple is admissible, so c is a redex and t can be popped, which gives (a). If t is in normal form then no innermost circle is a redex, so each has operands of the right sorts that are not admissible, which gives (b). $\square$

Part (b) is the precise statement of what a sort system buys. In a well-sorted term, evaluation can stop short of a value for exactly one reason: some operator was offered operands of the correct sorts that fall outside its domain of definition. Failures of kind do not occur after sorting succeeds. The remaining failures are failures of detail, and they are visible as inert circles that name the operator at fault.

Example 5.4. The term $\langle \text{pow } 2 \ \langle \text{sub } 0 \ 1 \rangle \rangle$ is well-sorted with sort Num. Popping the inner circle gives $\langle \text{pow } 2 \ -1 \rangle$, which is inert, since pow is not defined for negative exponents. The term is blocked although it was well-sorted, in accordance with part (b) of the theorem. $\triangleright$

5.4 Making operators total

Part (a) of the theorem suggests a design goal: choose signatures so that every operator is total on its signature, and then every well-sorted term evaluates to a value. This can be

achieved in some cases by refining the sorts. Adding a sort **Nat** of non-negative integers and giving **pow** the signature $\text{Num} \times \text{Nat} \rightarrow \text{Num}$ makes **pow** total, since a^b is an integer for every integer a and every non-negative b . Likewise, **rotate** becomes total if its second operand has a sort whose values are exactly the multiples of 90.

The price is that the refined sorts must be tracked through the rest of the system. The result of **sub** cannot be given sort **Nat**, because $0-1$ is negative, and so the term $\langle \text{pow } 2 \langle \text{sub } 5 \ 2 \rangle \rangle$ becomes ill-sorted, even though it evaluates without any difficulty. A sort system is conservative in this sense: it rejects some terms that would have evaluated, in order to guarantee that all of the terms it accepts do. The conservatism can be reduced by supplying more operators, for instance a truncated subtraction with result sort **Nat**, but it cannot in general be removed. Whether to enforce totality by refined sorts or to accept inert circles for the leftover cases is a design decision, and different applications settle it differently.

5.5 Ill-sorted terms in practice

An ill-sorted term is still a term, and the reduction relation applies to it unchanged. The term $\langle \text{rotate } \langle \text{add } 1 \ 2 \rangle \ I \rangle$ of the example above pops its inner circle to give $\langle \text{rotate } 3 \ I \rangle$, which is inert, because the number 3 is not an image. The failure shows up late, after some work has already been done, and the inert circle that results mixes the original operands with the partial result. A static sort check catches the same fault before any pop occurs, and for this reason sorting is normally run first. The inert-circle convention then acts as a second line of defence, covering the failures that sorting is not designed to exclude.

Exercises

Exercise 5.1 (†). Decide whether $\langle \text{rotate } \langle \text{add } 1 \ 2 \rangle \ I \rangle$ is well-sorted, and trace its evaluation. Explain which operand sorts are wrong.

Exercise 5.2 (†). Give a well-sorted term that is blocked, and say which operator fails and why.

Exercise 5.3 (†). With the sort **Nat** and the signature $\text{pow} : \text{Num} \times \text{Nat} \rightarrow \text{Num}$, is $\langle \text{pow } 2 \langle \text{sub } 5 \ 2 \rangle \rangle$ well-sorted if **sub** has signature $\text{Num} \times \text{Num} \rightarrow \text{Num}$? What operator could be added to make it well-sorted again?

Exercise 5.4. Prove that sorting is decidable and give an algorithm whose running time is linear in the size of the term.

Exercise 5.5. Let t be well-sorted and let every operator be total on its signature. Show that $\text{nf}(t)$ is a value, and that it has the same sort as t . Which earlier results do you need?

Chapter 6

Symbols and Substitution

All the terms considered so far have used concrete numbers, colours, and images. A computation is more useful if some of its inputs are left open and supplied later, and the notation provides for this with the sort **Sym**. A value of sort **Sym** is a symbol, a name that stands for a value not yet known. This chapter shows how symbols interact with popping, how they are replaced by values, and why evaluating a term in stages gives the same result as evaluating it all at once.

6.1 Symbols in circles

A symbol is a value, so a circle containing symbols is a legitimate term. What a symbol cannot do is take part in any numerical or pictorial operation, because no operator in the core fragment has **Sym** in its signature. The operators **add**, **mul**, **pow**, and the rest expect numbers or images, and a symbol is neither. A circle with a symbol as an operand is therefore inert in the sense of Chapter 3, and every circle enclosing it stays unpopped as well.

Example 6.1. The term $\langle \text{add } 1 \ x \rangle$, in which x is a symbol, is in normal form. So is every tower $T_n = \langle \text{pow } \text{Meta } T_{n-1} \rangle$ built on a symbol *Meta*, as in Chapter 4. The term $\langle \text{pow } \text{Meta } \langle \text{pow } \text{Meta } \text{Meta} \rangle \rangle$ has two circles, both inert, and the structure of the term is available for inspection even though no number has been computed. $\triangleright$

The observation is worth stating separately, because it contrasts with the way many programming languages treat an unknown. There the unknown is typically an error, or a null, or a value that must be supplied before evaluation can begin. Here the term simply stays as it is, and nothing is lost. The term is a perfectly good description of a computation that has not yet been carried out.

6.2 Substitution

A *substitution* is a finite map σ from symbols to values. Applying σ to a term t gives the term $\sigma(t)$ obtained by replacing each occurrence of a symbol x in the domain of σ by the

value $\sigma(x)$. Formally,

$$\sigma(v) = \begin{cases} \sigma(x) & \text{if } v \text{ is a symbol } x \text{ in the domain of } \sigma, \\ v & \text{otherwise,} \end{cases} \quad \sigma\langle o \ t_1 \cdots t_n \rangle = \langle o \ \sigma(t_1) \cdots \sigma(t_n) \rangle.$$

The values assigned by σ are values, so substitution never introduces a circle, and $\text{size}(\sigma(t)) = \text{size}(t)$ for every t .

Example 6.2. Let σ send *Meta* to 2. Then $\sigma(T_4) = \langle \text{pow } 2 \ \langle \text{pow } 2 \ \langle \text{pow } 2 \ 2 \rangle \rangle \rangle$, which pops in three steps to 65536. With a different substitution, sending *Meta* to 3, the tower T_3 becomes $\langle \text{pow } 3 \ \langle \text{pow } 3 \ 3 \rangle \rangle$, which pops to $3^{27} = 7625597484987$. A single symbolic tower thus describes a family of numerical computations, one for each value assigned to the symbol.

▷

6.3 Substitution commutes with popping

Substituting before or after popping should lead to the same place, at least where popping is possible both ways. The following lemma makes that precise. It requires that no operator has **Sym** in its signature and that no operator returns a symbol, so that symbols are always passive.

Lemma 6.3 (Substitution commutes with popping). *If $t \longrightarrow t'$ then $\sigma(t) \longrightarrow \sigma(t')$, for every substitution σ .*

Proof. Let the pop replace the redex $\langle o \ v_1 \cdots v_n \rangle$ at position p by the value $w = \delta_o(v_1, \dots, v_n)$. Since the operands are admissible they have the sorts required by the signature, none of which is **Sym**, so none of $v_1, \dots, v_n$ is a symbol and σ leaves them unchanged. Hence $\sigma(t)|_p = \langle o \ v_1 \cdots v_n \rangle$ is still a redex of $\sigma(t)$ and pops to w . The value w is not a symbol, since no operator returns one, and so $\sigma(w) = w$. Finally, the parts of t outside position p are acted on by σ in the same way in t and in t' . Therefore $\sigma(t) \longrightarrow \sigma(t[p \leftarrow w]) = \sigma(t')$. $\square$

Theorem 6.4 (Staged evaluation). *For every term t and substitution σ , $\sigma(\text{nf}(t)) \longrightarrow^* \text{nf}(\sigma(t))$.*

Proof. Since $t \longrightarrow^* \text{nf}(t)$, repeated use of Lemma 6.3 gives $\sigma(t) \longrightarrow^* \sigma(\text{nf}(t))$. Continuing to pop $\sigma(\text{nf}(t))$ until no pop applies yields a normal form of $\sigma(t)$, and by Theorem 4.3 this normal form is $\text{nf}(\sigma(t))$. $\square$

In practical terms: evaluating as far as possible with symbols present, then supplying values and finishing, always gives the same result as supplying the values first. The first stage does work that need not be repeated when the same term is used with many different substitutions. This is the idea of partial evaluation, in its simplest form.

Example 6.5. Let $t = \langle \text{add} \ \langle \text{mul } 2 \ 3 \rangle \ \langle \text{mul } x \ 4 \rangle \rangle$. The first stage pops the circle $\langle \text{mul } 2 \ 3 \rangle$ and stops, since the remaining circle $\langle \text{mul } x \ 4 \rangle$ is inert. The normal form is $\langle \text{add } 6 \ \langle \text{mul } x \ 4 \rangle \rangle$ after one pop. If x is later given the value 5, two more pops give 20 and then 26. The total number of pops is $3 = \text{size}(t)$, as it must be by Corollary 3.6, and for each further value of x only two pops are needed, not three. $\triangleright$

6.4 Substitution does not preserve normal forms

The inclusion in Theorem 6.4 cannot be strengthened to an equality. In general $\sigma(\text{nf}(t)) \neq \text{nf}(\sigma(t))$, because substitution can make inert circles ready. If $t = \langle \text{pow } x \ 2 \rangle$ then t is in normal form, and $\sigma(\text{nf}(t)) = \langle \text{pow } 3 \ 2 \rangle$ for the substitution sending x to 3, while $\text{nf}(\sigma(t)) = 9$. The two differ by one pop. The theorem says that the difference can always be closed by further popping, in some number of steps, and never leaves a gap that cannot be closed.

6.5 Free symbols and what comes next

A symbol that occurs in a term is *free*, since nothing in the term binds it. The core notation has no binding construct. If the notation is extended with a form that introduces a name, as in the circle $\langle \text{let } x \ t \ s \rangle$ meaning that x is to stand for t inside s , then questions of scope arise, and the ordinary difficulties of renaming and capture appear. They are well understood, and Chapter 9 indicates where to read about them. Everything in this book can be done with free symbols and substitution alone.

Exercises

Exercise 6.1 (†). Evaluate $\sigma(T_3)$ for $\sigma(\text{Meta}) = 3$, and state the number of pops.

Exercise 6.2 (†). Evaluate $\langle \text{add } \langle \text{mul } 2 \ 3 \rangle \ \langle \text{mul } x \ 4 \rangle \rangle$ in two stages for $x = 5$, giving the term after each stage and the pop counts.

Exercise 6.3 (†). Find a term t and substitution σ with $\sigma(\text{nf}(t)) \neq \text{nf}(\sigma(t))$ other than $\langle \text{pow } x \ 2 \rangle$, and verify that $\sigma(\text{nf}(t)) \longrightarrow^* \text{nf}(\sigma(t))$.

Exercise 6.4. Show that Lemma 6.3 fails if some operator is allowed to accept symbols. Construct an operator **show** of signature $\text{Sym} \rightarrow \text{Num}$, giving the number of letters in a name, and a term and substitution for which the lemma breaks.

Exercise 6.5. Compose two substitutions, σ followed by τ . Prove that $\tau(\sigma(t))$ is a substitution instance of t and describe it.

Chapter 7

Colours and Images

Nothing in the development so far depends on the values being numbers. A sort is a collection of values together with whatever operators are defined on it, and the notation, the popping rule, and the theorems of Chapters 2 to 6 apply unchanged to any collection of sorts and operators that meets the two requirements stated in Chapter 2. This chapter shows the claim at work on a second family of operators, which act on colours and on pictures. The point of the chapter is not the picture operations themselves, which are deliberately simple, but the fact that mixing sorts costs nothing in the theory.

7.1 The sorts

A *colour* is a name such as red or blue. An *image* is a finite rectangular grid of cells, each cell holding either a colour or the blank marker, written $\cdot$. An image with h rows and w columns is said to have height h and width w . Rows are listed from top to bottom and cells within a row from left to right. In running text an image is written as its rows separated by slashes, so that $\mathbf{x} \cdot \mathbf{x} / \cdot \mathbf{x}$ is the image of height 2 and width 3 whose top row is a filled cell, a blank, and another filled cell, and whose bottom row has the filled cell in the middle. Cells marked $\mathbf{x}$ in a drawing are understood to hold some colour whose name is not important for the example.

7.2 Three operators

The operators of the picture fragment have the following signatures and meanings.

Operator	Signature	Meaning
rotate	$\text{Image} \times \text{Num} \rightarrow \text{Image}$	rotate clockwise by the given number of degrees
recolour	$\text{Image} \times \text{Color} \rightarrow \text{Image}$	give every non-blank cell the given colour
beside	$\text{Image} \times \text{Image} \rightarrow \text{Image}$	place the second image to the right of the first

The operator **rotate** is admissible only when the angle is a multiple of 90, and the operator **beside** only when the two images have the same height. Negative angles are allowed and mean

anticlockwise rotation, and an angle of 360 gives back the original. A rotation by 90 clockwise turns the leftmost column, read from bottom to top, into the top row, and so on across the columns.

Example 7.1. Let $I = \text{abc/def}$, with six distinct colours. Then

$$\langle \text{rotate } I \ 90 \rangle = \text{da/eb/fc}, \quad \langle \text{rotate } I \ 180 \rangle = \text{fed/cba}, \quad \langle \text{rotate } I \ 270 \rangle = \text{cf/be/ad}.$$

Rotation by 90 changes the height and the width, exchanging them, while rotation by 180 does not. ▷

7.3 Composing operators

Nested circles of different sorts compose in exactly the way that nested numerical circles do. The term

$$\langle \text{rotate } \langle \text{recolor } I \ \text{red} \rangle \ 90 \rangle$$

is well-sorted of sort `Image` for any image I . Popping the inner circle gives an image of the same shape as I with every non-blank cell coloured red, and popping the outer circle rotates that image. If $I = \mathbf{x} \cdot \mathbf{x} / \cdot \mathbf{x}$ then the result is the image of height 3 and width 2 whose rows are $\cdot \mathbf{r}$, $\mathbf{r} \cdot$, $\cdot \mathbf{r}$, with $\mathbf{r}$ standing for a red cell.

A term with two independent inner circles shows the diamond property in a pictorial setting. In

$$\langle \text{beside } \langle \text{rotate } J \ 90 \rangle \ \langle \text{recolor } J \ \text{blue} \rangle \rangle$$

the rotation and the recolouring are both ready. By Lemma 4.2 they may be popped in either order, or together, and the final circle then places the rotated copy of J next to the blue copy. For this to be admissible the two images must have the same height, which holds, for instance, when J is square.

7.4 Persistence

Popping a circle never changes the image it was given. A circle such as $\langle \text{recolor } I \ \text{red} \rangle$ is replaced by a *new* image, and any other part of the term that mentions I continues to mention the original. The values of the notation are persistent in this sense, and the property is not an extra feature but a consequence of the single rule: popping replaces a circle by a value, and nothing else is touched. A practical consequence is that every intermediate term in a sequence of pops is still available. A user who has popped too far can return to an earlier term, and nothing needs to be undone.

7.5 Equations between terms

The confluence results of Chapter 4 give a sensible meaning to equality between terms. Say that terms s and t are *equivalent*, written $s \equiv t$, when they have the same normal form.

By Theorem 4.3 the relation is well defined, and it is an equivalence relation. Equivalences between picture terms can then be proved by calculation.

Proposition 7.2. *For every image I , colours c and d , and angles a and b that are multiples of 90:*

- (a) $\langle \text{rotate } \langle \text{rotate } I \ a \rangle \ b \rangle \equiv \langle \text{rotate } I \ (a + b) \rangle$;
- (b) $\langle \text{rotate } I \ 360 \rangle \equiv I$;
- (c) $\langle \text{rotate } \langle \text{recolor } I \ c \rangle \ a \rangle \equiv \langle \text{recolor } \langle \text{rotate } I \ a \rangle \ c \rangle$;
- (d) $\langle \text{recolor } \langle \text{recolor } I \ c \rangle \ d \rangle \equiv \langle \text{recolor } I \ d \rangle$.

Proof. Part (a): rotation by a degrees is a power of the quarter-turn Q , namely $Q^{a/90}$, and the powers of Q satisfy $Q^m Q^n = Q^{m+n}$ and $Q^4 = \text{id}$. Part (b) is the case $Q^4 = \text{id}$, which is checked directly: four quarter-turns return every cell to its starting position. Part (c): rotation moves cells without altering their contents, and recolouring alters the contents of non-blank cells without moving them, so the two operations act on independent aspects of the image and commute. Part (d): the second recolouring overwrites the colour of every non-blank cell with d , and the cells that are non-blank are the same before and after the first recolouring, so the colour c leaves no trace. $\square$

7.6 Derived operators

A circle expression can be given a name and used as an operator, if the name is understood as an abbreviation. Stacking one image above another is not among the three operators listed, but it can be defined from them. Place A above B by rotating each anticlockwise by a quarter-turn, putting them side by side, and rotating back:

$$\langle \text{above } A \ B \rangle := \langle \text{rotate } \langle \text{beside } \langle \text{rotate } A \ 270 \rangle \ \langle \text{rotate } B \ 270 \rangle \rangle \ 90 \rangle.$$

For $A = \text{ab/cd}$ and $B = \text{ef}$, the anticlockwise quarter-turns give bd/ac and f/e , which are placed side by side as bdf/ace , and the final clockwise turn gives ab/cd/ef , as desired. The definition is admissible exactly when A and B have the same width, because the widths become the heights after the first rotation. A derived operator introduces no new reduction behaviour, since every use of it can be replaced by its definition, and the theorems of the earlier chapters apply to terms that use it.

7.7 Why the circle is not the picture

A circle in a term is a computational boundary. It records which operands belong to which operator, and it plays no part in the appearance of the result. A circle containing a rotation has no visible counterpart in the rotated image, and an image that happens to be shaped like a ring has no particular relation to any circle in a term that produced it. The two uses of the word should be kept apart, even though the notation can be drawn with circles around

pictures, so that the structure of a computation sits visually on top of the objects it operates on. The separation is what lets the notation stay general, and the same circles that enclose picture operands in this chapter enclosed numbers in the preceding ones.

Exercises

Exercise 7.1 (†). Let $I = \mathbf{abc/def}$. Compute $\langle \text{rotate } I \ 90 \rangle$, $\langle \text{rotate } I \ 180 \rangle$, and $\langle \text{rotate } I \ 270 \rangle$, and check that three quarter-turns agree with $\langle \text{rotate } I \ 270 \rangle$.

Exercise 7.2 (†). Prove part (c) of Proposition 7.2 by describing the effect of each operator on a single cell.

Exercise 7.3 (†). Let I have height 2 and width 3 and let K have height 3 and width 3. Is $\langle \text{beside } I \ K \rangle$ a redex? What happens if it is placed inside a larger term?

Exercise 7.4 (†). Compute $\langle \text{above } A \ B \rangle$ for $A = \mathbf{ab/cd}$ and $B = \mathbf{ef}$ by popping the defining term step by step, and count the pops.

Exercise 7.5. Define an operator **flip** that reflects an image left to right. Express a clockwise rotation by 90 as a composition of **flip** and another operation of your choosing, and decide which further operators, if any, are needed.

Chapter 8

Rewriting

Up to this point every operator has returned a value. An operator may instead be allowed to return a circle, so that popping transforms one expression into another expression and does not necessarily bring the computation closer to a number. This is the step from evaluation to rewriting, and it is the step at which the clean results of Chapter 4 stop being automatic. This chapter introduces rewriting in the setting of the notation, shows a system of rules for which the good properties can be proved, and exhibits three small variations that destroy them.

8.1 Rules

A *rewriting rule* has the form $\ell \rightarrow r$, where ℓ and r are terms that may contain *variables*, written x , y , and standing for arbitrary terms. The left-hand side ℓ is the pattern. A term t *matches* the rule at a position p if $t|_p$ is obtained from ℓ by replacing each variable by some term, and the rule then allows t to be rewritten by replacing $t|_p$ by r with the same replacements made. The variables of r are required to occur in ℓ . Values are written in rules as themselves and are matched only by themselves.

Consider the following eight rules, which describe the usual identities of arithmetic and are added to the (Pop) rule of Chapter 3.

(R1) $\langle \text{pow } x \ 1 \rangle \rightarrow x$	(R2) $\langle \text{pow } x \ 0 \rangle \rightarrow 1$
(R3) $\langle \text{add } x \ 0 \rangle \rightarrow x$	(R4) $\langle \text{add } 0 \ x \rangle \rightarrow x$
(R5) $\langle \text{mul } x \ 1 \rangle \rightarrow x$	(R6) $\langle \text{mul } 1 \ x \rangle \rightarrow x$
(R7) $\langle \text{mul } x \ 0 \rangle \rightarrow 0$	(R8) $\langle \text{mul } 0 \ x \rangle \rightarrow 0$

Because the variable x may stand for any term, the rules apply to circles that (Pop) cannot touch. The term $\langle \text{pow } y \ 1 \rangle$, with y a symbol, is inert under (Pop), but rule (R1) rewrites it to y . Similarly $\langle \text{mul } \langle \text{pow } y \ 2 \rangle \ 0 \rangle$ is rewritten by (R7) to 0, discarding the inert circle inside it.

Example 8.1. The term $\langle \text{add } \langle \text{mul } x \ 1 \rangle \ 0 \rangle$, with x a symbol, rewrites by (R5) to $\langle \text{add } x \ 0 \rangle$ and then by (R3) to x . The expression has two circles and the symbolic computation removes

both, without ever assigning x a value. The term $\langle \text{pow } \langle \text{pow } x \ 1 \rangle \ 0 \rangle$ rewrites to $\langle \text{pow } x \ 0 \rangle$ by (R1) and to 1 by (R2). $\triangleright$

8.2 What changes

Two of the properties established in Chapter 4 no longer follow from the same arguments. A rule need not replace a circle by a value, so the reduction of a term can leave the size of the term fixed or even increase it, and Theorem 4.1 has nothing to say. Rules also may overlap, in the sense that two different rules, or one rule at two different positions of the same term, may apply to the same circle, and the redexes then no longer sit at disjoint positions, so Lemma 4.2 cannot be applied directly. Order independence of length, finally, is lost, even where the normal form is still unique.

Example 8.2. Let $t = \langle \text{mul } \langle \text{add } x \ 0 \rangle \ 0 \rangle$. Rewriting the inner circle first, by (R3), gives $\langle \text{mul } x \ 0 \rangle$, and (R7) then gives 0, for a total of two steps. Rewriting the outer circle first, by (R7), gives 0 immediately, in one step. The normal form is the same, but the lengths of the two sequences differ, and so Corollary 4.4 does not extend to rewriting. $\triangleright$

8.3 Termination of the arithmetic rules

For the eight rules above, termination is easy to prove, and the proof shows the general method: find a quantity that decreases with every step.

Theorem 8.3. *Every sequence of rewrites using (Pop) and (R1)–(R8) is finite.*

Proof. The size of a term decreases at every step. For (Pop) this is Proposition 3.5. For a rule among (R1)–(R8), the left-hand side has size $1 + \text{size}(x)$ and the right-hand side is x , of size $\text{size}(x)$, or a numeral, of size 0, so the replacement lowers the size of the subterm by at least one. The size of the whole term is then lowered by the same amount, since the context is unchanged. $\square$

8.4 Confluence by critical pairs

Showing that the normal form is unique requires a different argument. Two rules overlap when a single term can be rewritten by either. The *critical pairs* of a system are the pairs of terms obtained by applying each of two overlapping rules to the overlap. If both terms of every critical pair can be rewritten to a common term, the pair is said to be *joinable*. A theorem of Knuth and Bendix states that a terminating rewriting system is locally confluent if and only if all of its critical pairs are joinable, and a lemma of Newman states that a terminating, locally confluent system is confluent, and therefore has unique normal forms. The two results together give a practical test for uniqueness of normal forms: check the critical pairs.

Theorem 8.4. *The system consisting of (Pop) and (R1)–(R8) is confluent, and every term has a unique normal form.*

Proof. The system terminates by Theorem 8.3, so by the two results quoted it suffices to check that every critical pair is joinable. The left-hand sides of (R1)–(R8) each have one circle, with operands that are variables or numerals, so there is no overlap below the root of a left-hand side, and every critical pair comes from two rules applying to the same circle. Rules with different operators cannot overlap. For the same operator there are the following cases.

Addition. Rules (R3) and (R4) both apply to $\langle \text{add } 0 \ 0 \rangle$ and give 0 in both cases.

Multiplication. Rules (R5) and (R6) both apply to $\langle \text{mul } 1 \ 1 \rangle$ and give 1. Rules (R5) and (R8) both apply to $\langle \text{mul } 0 \ 1 \rangle$ and give 0. Rules (R6) and (R7) both apply to $\langle \text{mul } 1 \ 0 \rangle$ and give 0. Rules (R7) and (R8) both apply to $\langle \text{mul } 0 \ 0 \rangle$ and give 0. The other pairs cannot overlap, since they require the same operand to be both 1 and 0.

Powers. Rules (R1) and (R2) require the exponent to be 1 and 0 respectively and so cannot overlap.

Overlaps with (Pop). The rule (Pop) applies to a circle whose operands are all numbers, and the rules (R1)–(R8) may apply to the same circle when an operand is 0 or 1. The two results agree in every such case, by the identities $a + 0 = 0 + a = a$, $a \cdot 1 = 1 \cdot a = a$, $a \cdot 0 = 0 \cdot a = 0$, $a^1 = a$, and $a^0 = 1$.

All critical pairs are therefore joinable. $\square$

The theorem justifies using any strategy at all for simplification. The interpreter of Chapter 10 uses the leftmost innermost strategy, and by the theorem it reaches the same normal form as any other, although, as Example 8.2 shows, possibly in a different number of steps.

8.5 When rewriting goes wrong

Three small variations show how fragile the conclusions of the previous section are.

Non-termination. Add the rule $\langle \text{add } x \ y \rangle \rightarrow \langle \text{add } y \ x \rangle$, which expresses commutativity of addition. Then $\langle \text{add } a \ b \rangle \rightarrow \langle \text{add } b \ a \rangle \rightarrow \langle \text{add } a \ b \rangle \rightarrow \dots$ is an infinite sequence, for symbols a and b , and no term of this shape has a normal form. The identity is true and a perfectly good equation, but as a rule applied in one direction it makes the system loop. Systems that must use commutative operators handle the difficulty by rewriting modulo the commutativity, an extension not pursued here.

Growth without non-termination. Add the distributive rule $\langle \text{mul } a \ \langle \text{add } b \ c \rangle \rangle \rightarrow \langle \text{add } \langle \text{mul } a \ b \rangle \ \langle \text{mul } a \ c \rangle \rangle$. The right-hand side has three circles where the left has two, so the size of a term can increase at a step, and the argument by size no longer applies. The system restricted to **add** and **mul** nonetheless terminates. Interpret each value as the number 2, each circle $\langle \text{add } s \ t \rangle$ as $[s] + [t] + 1$, and each circle $\langle \text{mul } s \ t \rangle$ as $[s] \cdot [t]$, where $[s]$ denotes the interpretation of s . All interpretations are at least 2, and both operations are strictly increasing

in each argument at such values. For the distributive rule with $[a]$, $[b]$, $[c]$ the interpretations of the three subterms, the left-hand side has interpretation $[a]([b] + [c] + 1) = [a][b] + [a][c] + [a]$, and the right-hand side has $[a][b] + [a][c] + 1$, which is smaller by $[a] - 1 \geq 1$. Every use of the rule, in any context, therefore lowers the interpretation of the whole term. The rule (Pop) lowers it as well, since a circle whose two operands are values has interpretation at least 4 and is replaced by a value, whose interpretation is 2. A strictly decreasing sequence of integers that are at least 2 must be finite.

Loss of confluence. Replace (R2) by the pair of rules $\langle \text{pow } x \ 0 \rangle \rightarrow 1$ and $\langle \text{pow } 0 \ x \rangle \rightarrow 0$. Both rules apply to $\langle \text{pow } 0 \ 0 \rangle$ and give 1 and 0 respectively. These are different values, and so both are normal forms. The critical pair is not joinable, the system is not confluent, and the term $\langle \text{pow } 0 \ 0 \rangle$ has two normal forms. The conflict is a genuine disagreement between two reasonable-looking identities, and no choice of evaluation strategy can repair it.

8.6 Orthogonality and the earlier results

The results of Chapter 4 can be recovered in the more general setting under stronger hypotheses. A rewriting system is *orthogonal* if its left-hand sides are linear, meaning that no variable occurs twice, and no two of its rules overlap. For orthogonal systems confluence holds without any termination assumption, a result going back to Rosen and to Huet. The core fragment of the notation is orthogonal, since (Pop) rules for distinct operators or distinct operand tuples cannot overlap, which is the reason that confluence there came so cheaply. The rules (R1)–(R8) are not orthogonal, because they overlap, and their confluence rests on the combination of termination and joinable critical pairs.

Exercises

Exercise 8.1 ($\dagger$). Normalize $\langle \text{mul } \langle \text{add } x \ 0 \rangle \ 0 \rangle$ using the leftmost innermost strategy and using the outermost strategy. Compare the lengths and the results.

Exercise 8.2. Show that the commutativity rule produces an infinite sequence from $\langle \text{add } a \ b \rangle$, and explain why the rule (Pop) does not prevent this when a and b are symbols.

Exercise 8.3 ($\dagger$). Verify that the modified rule pair $\langle \text{pow } x \ 0 \rangle \rightarrow 1$, $\langle \text{pow } 0 \ x \rangle \rightarrow 0$ has a critical pair that is not joinable, and give the two normal forms of $\langle \text{pow } 0 \ 0 \rangle$.

Exercise 8.4 ($\dagger$). Carry out the interpretation argument for the distributive rule on the term $\langle \text{mul } 2 \ \langle \text{add } 3 \ 4 \rangle \rangle$, treating the numerals as values of interpretation 2, and show that the interpretation of the rewritten term is smaller.

Exercise 8.5. Find all critical pairs of the system obtained by adding the rule $\langle \text{add } x \ x \rangle \rightarrow \langle \text{mul } 2 \ x \rangle$ to (R1)–(R8), and decide which are joinable.

Chapter 9

Neighbours and Extensions

The notation developed in the preceding chapters is a small member of a large family of formalisms for expressions, and it is useful to see where it sits. This chapter compares it briefly with several neighbours, indicates what must be added to reach them, and records the results from the literature that the reader will need. It is a map and not a survey, and the references at the end of the book are the places to go for the details.

9.1 Symbolic expressions

A circle $\langle o\ t_1 \cdots t_n \rangle$ corresponds exactly to the parenthesized prefix form $(o\ t_1\ \cdots\ t_n)$ of Lisp-family languages, and a term of the notation can be read as a symbolic expression in which the first element of every list is an operator name drawn from a fixed table. The correspondence is useful in practice, because it means that the terms of the notation can be stored, printed, and exchanged in an existing and well-supported syntax, and the interpreter of Chapter 10 could equally well have been written in a Lisp. The differences are in what is allowed. In a Lisp-family language the head of a list may itself be a computed expression, a function may be passed as an argument and returned as a result, and names may be bound by the program. None of these is present in the core fragment. Operators here are constants of the signature, not values, and the operators cannot be built by the program that uses them.

9.2 Many-sorted term rewriting

Viewed abstractly, the sorts, operators, and terms of Chapter 2 form a many-sorted signature, and the terms are the ground terms over that signature extended with a constant for every value. The (Pop) rule is a family of ground rewriting rules, one for each admissible tuple of values. In this light the notation is a specific term rewriting system, and the results of Chapters 4 and 8 are instances of the standard theory of such systems: termination by a measure that decreases at every step, confluence from orthogonality or from joinable critical pairs, and the rest. Readers who wish to follow these ideas further will find the subject

treated systematically in the book of Baader and Nipkow, and the original results of Knuth and Bendix, Newman, Rosen, and Huet are cited at the end of this book.

9.3 The lambda calculus

The most important neighbour is the lambda calculus, in which every computation is built from variables, abstractions $\lambda x. M$ that introduce a name, and applications $M N$ that supply an argument. To reach it from the present notation one adds two kinds of term, an abstraction form that binds a symbol and an application form that substitutes for it, with one new reduction rule,

$$\langle \text{app } \langle \text{lam } x M \rangle N \rangle \longrightarrow M[x := N].$$

Substitution now has to be *capture-avoiding*: when N contains a symbol that is bound inside M , the bound symbol must be renamed first, or the substitution changes the meaning of the term. The details are standard, and they are treated in the books of Barendregt and of Pierce.

Two of the results of this book survive the addition and one does not. The diamond property in its present form fails, because a single substituted argument may be duplicated, so that the redexes it contains are copied and a single pop on one side requires several on the other. A weaker form, confluence, remains true, and is the Church–Rosser theorem, so normal forms are still unique when they exist. Termination, however, is lost. The term $(\lambda x. x x)(\lambda x. x x)$ pops to itself forever. Termination can be restored by a sort discipline: the simply typed lambda calculus, in which every abstraction and application is given a sort of function type, is strongly normalizing, a theorem usually attributed to Tait. The comparison with Chapter 5 is instructive. There, sorts served to separate failures of kind from failures of detail. In the lambda calculus, sorts are the very thing that tames self-application.

9.4 Evaluation order

The core fragment is pure, and every circle is evaluated before the circle enclosing it. A different design, used in some languages for conditional expressions, evaluates an operand only when it is needed. In the present setting an operator if of three operands, a test and two branches, would pop only when all three operands were values, so that both branches would be evaluated whichever one was selected. For a pure notation this has no effect on the value, by the results of Chapter 4, but it can have an effect on the cost, and on whether the evaluation blocks. If the discarded branch contains an inert circle, then the whole conditional is blocked even though its value is perfectly well determined by the selected branch. A lazy variant that pops the conditional as soon as the test is a value would avoid this, at the price of a slightly more complicated reduction relation, and the diamond property must be rechecked for it.

9.5 Operators as commitments

A last direction concerns the status of a circle that has been popped. In the present notation a pop is final, in that the circle disappears and the value takes its place, and nothing records that the circle was ever there. For some purposes this is too little. A circle might stand for a commitment to carry out an operation, whose lifecycle involves several stages: the commitment is opened, its operands are bound, the operation is applied, the result is verified against some condition, and only then is the commitment collapsed into its value. A commitment may also be refused at any of the earlier stages, and a failed verification is a state of its own and does not by itself discharge the commitment. A development of this kind turns the notation into a calculus of staged and checked computation, with the (Pop) rule of the present book corresponding to the final collapse in the case where nothing can go wrong. It requires a notion of history to be added to the semantics, and it requires types that record the stage of each commitment. These extensions are beyond the scope of an introduction, and they are mentioned here only to indicate that the core fragment is the first layer of a larger structure and not the whole of it.

Exercises

Exercise 9.1. Write the term $\langle \text{add } 2 \langle \text{mul } 3 \ 4 \rangle \rangle$ as a symbolic expression in prefix form and as a function-application expression of the form $\text{add}(2, \text{mul}(3, 4))$.

Exercise 9.2. Explain why the diamond property of Lemma 4.2 fails for the **app-lam** rule in this chapter. Give a term in which a single pop on one side requires two pops to join with the other.

Exercise 9.3. Suppose an operator **if** of signature $\text{Bool} \times \text{Num} \times \text{Num} \rightarrow \text{Num}$ is added, with $\delta_{\text{if}}(\text{true}, a, b) = a$ and $\delta_{\text{if}}(\text{false}, a, b) = b$. Give a well-sorted term that is blocked although the selected branch has a value.

Chapter 10

A Reference Interpreter

This chapter presents a complete interpreter for the core fragment of the notation, together with the symbolic simplifier of Chapter 8. The program is written in Python, uses only the standard library, and runs in well under a second. It is intended to be read alongside the definitions: each function is a few lines long, and each corresponds closely to a definition or a lemma in the text. The file is `spherepop.py`, and the transcript at the end of the chapter is the output of running it.

10.1 Values and terms

A value is an integer, a colour, a symbol, or an image. A circle is a pair of an operator name and a tuple of operand terms, and a term is either a value or a circle. The classes are declared *frozen*, which makes instances immutable and hashable, so that the persistence property discussed in Chapter 7 holds by construction: no operation can change a value that another part of a term is holding.

```
# ----- values

@dataclass(frozen=True)
class Color:
    name: str

@dataclass(frozen=True)
class Sym:
    name: str

@dataclass(frozen=True)
class Image:
    rows: Tuple[Tuple[str, ...], ...]    # a rectangular grid of colour names
```

```
Value = Union[int, Color, Sym, Image]

@dataclass(frozen=True)
class Circle:
    op: str
    args: Tuple["Term", ...]

Term = Union[Value, Circle]

def sort_of(v: Value) -> str:
    if isinstance(v, bool):
        raise TypeError("booleans are not values of the core fragment")
    if isinstance(v, int):
        return "Num"
    if isinstance(v, Color):
        return "Color"
    if isinstance(v, Sym):
        return "Sym"
    if isinstance(v, Image):
        return "Image"
    raise TypeError(f"not a value: {v!r}")

def is_value(t: Term) -> bool:
    return not isinstance(t, Circle)
```

The function `sort_of` reports the sort of a value, and `is_value` distinguishes values from circles. Python's booleans are excluded explicitly, because they are a subclass of the integers and would otherwise be accepted as numbers.

10.2 Operators and admissibility

An operator is a record containing its name, its signature, its result sort, and a function `delta` that computes δ_o . The table `OPERATORS` holds the seven operators used in the text. The function `admissible` decides whether an operand tuple lies in $\text{dom}(\delta_o)$, by checking the arity and sorts against the signature and then the side conditions of the particular operator: a non-negative exponent for `pow`, an angle divisible by 90 for `rotate`, and equal heights for `beside`.

```
# ----- operators

@dataclass(frozen=True)
class Operator:
```

```
name: str
signature: Tuple[str, ...]
result: str
delta: Callable[..., Value]

def _rotate90(img: Image, quarter_turns: int) -> Image:
    rows = img.rows
    for _ in range(quarter_turns % 4):
        rows = tuple(zip(*rows[::-1]))          # clockwise quarter turn
    return Image(tuple(tuple(r) for r in rows))

def _rotate(img: Image, degrees: int) -> Image:
    if degrees % 90 != 0:
        raise ValueError("rotation is defined for multiples of 90 only")
    return _rotate90(img, degrees // 90)

def _recolor(img: Image, c: Color) -> Image:
    return Image(tuple(tuple(c.name if p != "." else "." for p in r)
                        for r in img.rows))

def _beside(a: Image, b: Image) -> Image:
    if len(a.rows) != len(b.rows):
        raise ValueError("images must have equal height")
    return Image(tuple(ra + rb for ra, rb in zip(a.rows, b.rows)))

OPERATORS: Dict[str, Operator] = {}

def define(name, signature, result, delta):
    OPERATORS[name] = Operator(name, tuple(signature), result, delta)

define("add", ("Num", "Num"), "Num", lambda a, b: a + b)
define("sub", ("Num", "Num"), "Num", lambda a, b: a - b)
define("mul", ("Num", "Num"), "Num", lambda a, b: a * b)
define("pow", ("Num", "Num"), "Num", lambda a, b: a ** b)
define("rotate", ("Image", "Num"), "Image", _rotate)
define("recolor", ("Image", "Color"), "Image", _recolor)
define("beside", ("Image", "Image"), "Image", _beside)

def admissible(op: Operator, args: Tuple[Value, ...]) -> bool:
    """Sort check plus the operator's own side conditions."""
```

```

if len(args) != len(op.signature):
    return False
if any(sort_of(a) != s for a, s in zip(args, op.signature)):
    return False
if op.name == "pow" and args[1] < 0:
    return False                                # keep the carrier integral
if op.name == "rotate" and args[1] % 90 != 0:
    return False
if op.name == "beside" and len(args[0].rows) != len(args[1].rows):
    return False
return True

```

The rotation is implemented by reversing the rows and transposing, which is the clockwise quarter-turn; repeating it gives the other angles, and Python's treatment of negative remainders makes negative angles behave as anticlockwise rotations, in accordance with the convention of Chapter 7.

10.3 Popping

The function `step` performs one pop. It searches the operands from left to right for a circle that can be popped, and if it finds one it rebuilds the enclosing circle around the result. If none of the operands can be popped, and all of them are values, it tries rule (Pop) on the circle itself. The function returns `None` when no pop applies, which is the test for a normal form. The function `normalize` repeats `step` until it returns `None`, counting the pops, and the function `size` counts circles. The step limit guards against runaway computation, although by Theorem 4.1 it can never be reached by a term of fewer circles than the limit.

```

# ----- reduction

def step(t: Term) -> Optional[Term]:
    """One leftmost pop, or None if the term is in normal form."""
    if not isinstance(t, Circle):
        return None
    for i, a in enumerate(t.args):
        r = step(a)
        if r is not None:
            return Circle(t.op, t.args[:i] + (r,) + t.args[i + 1:])
    if all(is_value(a) for a in t.args):
        op = OPERATORS.get(t.op)
        if op is not None and admissible(op, t.args):
            return op.delta(*t.args)
    return None

def normalize(t: Term, limit: int = 10_000) -> Tuple[Term, int]:
    n = 0

```

```
while n < limit:
    r = step(t)
    if r is None:
        return t, n
    t, n = r, n + 1
raise RuntimeError("step limit reached")

def size(t: Term) -> int:
    """Number of circles in a term."""
    if not isinstance(t, Circle):
        return 0
    return 1 + sum(size(a) for a in t.args)
```

The strategy is leftmost innermost. By Theorem 4.3 it makes no difference to the normal form, and by Corollary 4.4 none to the number of pops.

10.4 Substitution

Substitution is a direct transcription of the definition in Chapter 6. A substitution is a dictionary from symbol names to values.

```
def substitute(t: Term, env: Dict[str, Value]) -> Term:
    if isinstance(t, Sym):
        return env.get(t.name, t)
    if isinstance(t, Circle):
        return Circle(t.op, tuple(substitute(a, env) for a in t.args))
    return t
```

10.5 Rewriting

The function `rewrite_step` extends `step` with the rules (R1)–(R8) of Chapter 8. As before it first looks for a step inside the operands, and only then tries the rules on the circle itself, so the strategy remains leftmost innermost. Comparing it with `step` shows exactly what rewriting adds: the rules fire on circles whose operands are not all values, which is what allows inert circles to be simplified.

```
def rewrite_step(t: Term) -> Optional[Term]:
    """Symbolic simplification: operators may now return circles."""
    if not isinstance(t, Circle):
        return None
    for i, a in enumerate(t.args):
        r = rewrite_step(a)
        if r is not None:
            return Circle(t.op, t.args[:i] + (r,) + t.args[i + 1:])
    a = t.args
```

```
    if t.op == "pow" and len(a) == 2:
        if a[1] == 1:
            return a[0]
        if a[1] == 0:
            return 1
    if t.op == "add" and len(a) == 2:
        if a[1] == 0:
            return a[0]
        if a[0] == 0:
            return a[1]
    if t.op == "mul" and len(a) == 2:
        if a[1] == 1:
            return a[0]
        if a[0] == 1:
            return a[1]
        if a[1] == 0 or a[0] == 0:
            return 0
    if all(is_value(x) for x in a):
        op = OPERATORS.get(t.op)
        if op is not None and admissible(op, a):
            return op.delta(*a)
    return None

def simplify(t: Term, limit: int = 10_000) -> Tuple[Term, int]:
    n = 0
    while n < limit:
        r = rewrite_step(t)
        if r is None:
            return t, n
        t, n = r, n + 1
    raise RuntimeError("step limit reached")
```

10.6 A test run

The remaining part of the file builds the examples of the text and prints the results. Running it with `python3 spherepop.py` produces the following transcript. The first line shows the evaluation of $2 + 3 \times 4$ in two pops. The tower section confirms that each T_n over a symbol is already in normal form, with $n - 1$ circles, and the next section shows the same towers with *Meta* replaced by 2, giving 2, 4, 16, 65536 in 0, 1, 2, 3 pops, and checking that the fifth tower is 2^{65536} , a number of 65537 bits. The left-nested term with four twos gives 256. The two picture terms of Chapter 7 follow, and then two blocked terms, which stop at once with zero pops. The last section shows the simplifier of Chapter 8.

```
-- arithmetic
<add 2 <mul 3 4>> => 14 in 2 steps
```

```

-- towers over the symbol Meta
1 Meta circles: 0 normal form unchanged: True
2 <pow Meta Meta> circles: 1 normal form unchanged: True
3 <pow Meta <pow Meta Meta>> circles: 2 normal form unchanged: True
4 <pow Meta <pow Meta <pow Meta Meta>>> circles: 3 normal form unchanged: True
-- towers with Meta := 2
1 2 => 2 in 0 steps
2 <pow 2 2> => 4 in 1 steps
3 <pow 2 <pow 2 2>> => 16 in 2 steps
4 <pow 2 <pow 2 <pow 2 2>>> => 65536 in 3 steps
5 value is 2**65536: True bit length: 65537
-- left nesting of four twos
<pow <pow <pow 2 2> 2> 2> => 256
-- images
<rotate <recolor x.x/.x. red> 90> => .red/red./.red
<beside <rotate xx/xx 90> <recolor xx/xx blue>> => xxblueblue/xxblueblue
-- inert circles
<rotate red x.x/.x.> => <rotate red x.x/.x.> after 0 steps
<add 1 <pow 2 -1>> => <add 1 <pow 2 -1>> after 0 steps
-- order independence on a small term
<add <mul 2 3> <mul 4 5>> => 26
-- rewriting
<pow x 1> => x in 1 steps
<add <mul x 1> 0> => x in 2 steps
<mul <add x 0> 0> => 0 in 2 steps
<pow <pow x 1> 0> => 1 in 2 steps

```

Exercises

Exercise 10.1. Add an operator `min` of signature $\text{Num} \times \text{Num} \rightarrow \text{Num}$ to the table and check that terms using it normalize as expected. Which parts of the proofs in Chapter 4 need to be revisited? (None should.)

Exercise 10.2. Implement the operator `flip` of Exercise 7.5 and test the identity you found there on several images.

Exercise 10.3. Write a function that performs one round of parallel popping, and use it to confirm Proposition 4.5 on balanced and unbalanced terms.

Exercise 10.4. Write a version of `normalize` that, at each step, chooses a random redex instead of the leftmost one. Run it many times on a term of your choice and check that the result and the number of pops never vary, as Theorem 4.3 and Corollary 4.4 predict. Then do the same for the simplifier, using the term of Example 8.2, and observe that the number of steps does vary.

Appendix A

Solutions to Selected Exercises

Solutions are given for the exercises marked with a dagger.

Exercise 1.1. $2 + 3 \times 4$ is $\langle \text{add } 2 \langle \text{mul } 3 \ 4 \rangle \rangle$, with value 14. $(2 + 3) \times 4$ is $\langle \text{mul } \langle \text{add } 2 \ 3 \rangle \ 4 \rangle$, with value 20.

Exercise 1.2. The groupings are $\langle \text{pow } a \langle \text{pow } b \ c \rangle \rangle$ and $\langle \text{pow } \langle \text{pow } a \ b \rangle \ c \rangle$. For $(a, b, c) = (2, 3, 2)$ the first is $2^{(3^2)} = 2^9 = 512$ and the second is $(2^3)^2 = 8^2 = 64$.

Exercise 1.3. There are two sequences, according to whether $\langle \text{mul } 2 \ 3 \rangle$ or $\langle \text{sub } 9 \ 4 \rangle$ is popped first. Both pass through the term $\langle \text{add } 6 \ 5 \rangle$ after two pops and reach 11 after three.

Exercise 2.1. The first term has four circles and depth 3, as computed in the text. In the second term, the exponent 2 is replaced by $\langle \text{add } 1 \ 1 \rangle$, which adds one circle at depth 1 beneath **pow**. The size is 5. The depth is still 3, because **pow** has operands of depths 1 and 1, so its depth is 2, and the outer circle has depth $1 + \max(1, 2) = 3$.

Exercise 2.2. By induction on the depth d . If $d = 0$ the term is a value of size $0 = 2^0 - 1$. If $d \geq 1$ the term is $\langle o \ t_1 \ t_2 \rangle$ and each operand has depth at most $d - 1$, hence by the induction hypothesis size at most $2^{d-1} - 1$. The size of the term is therefore at most $1 + 2(2^{d-1} - 1) = 2^d - 1$.

Exercise 2.3. The positions are ε , 1, 2, $1 \cdot 1$, and $1 \cdot 2$. The subterm at $1 \cdot 2$ is 3, and $t[1 \cdot 2 \leftarrow 7] = \langle \text{add } \langle \text{mul } 2 \ 7 \rangle \ 4 \rangle$.

Exercise 3.1. The innermost circles are $\langle \text{mul } 2 \ 3 \rangle$ at position 1 and $\langle \text{sub } 9 \ 4 \rangle$ at position $2 \cdot 2$, and both are redexes. The circle $\langle \text{mul } 4 \langle \text{sub } 9 \ 4 \rangle \rangle$ is not a redex until the subtraction has been popped. The term has four circles. Denote the pops by m_1 (for $\text{mul } 2 \ 3$, value 6), s (for $\text{sub } 9 \ 4$, value 5), m_2 (for $\text{mul } 4 \ 5$, value 20), and a (the final addition). The constraints are that s precedes m_2 and that a comes last, so there are three sequences: m_1, s, m_2, a ; s, m_1, m_2, a ; and s, m_2, m_1, a . All reach 26.

Exercise 3.2. $\langle \text{sub } 5 \ 2 \rangle \rightarrow 3$, then $\langle \text{mul } 2 \ 3 \rangle \rightarrow 6$, $\langle \text{add } 1 \ 1 \rangle \rightarrow 2$, and finally $\langle \text{pow } 2 \ 6 \rangle \rightarrow 64$. There are four pops, equal to the size of the term, as required by Corollary 3.6.

Exercise 3.3. The term is neither a value nor a term with a pending redex after the first pop. It starts with one redex, $\langle \text{sub } 2 \ 3 \rangle$, which pops to -1 . The circle $\langle \text{pow } 0 \ -1 \rangle$ is then innermost and inadmissible, since the exponent is negative, and so the term is blocked. The remaining term is $\langle \text{mul } 3 \ \langle \text{pow } 0 \ -1 \rangle \rangle$, with two circles.

Exercise 4.1. $t_1 = \langle \text{add } 6 \ \langle \text{mul } 4 \ 5 \rangle \rangle$, $t_2 = \langle \text{add } \langle \text{mul } 2 \ 3 \rangle \ 20 \rangle$, and $t_3 = \langle \text{add } 6 \ 20 \rangle$, reached from t_1 by popping the remaining multiplication and from t_2 by popping the other.

Exercise 4.2. A balanced binary term of size $15 = 2^4 - 1$ has depth 4. Sequential popping takes 15 pops, and parallel popping takes 4 rounds, by Proposition 4.5.

Exercise 4.3. $\langle \text{pow } 3 \ \langle \text{pow } 3 \ 3 \rangle \rangle$ pops to $\langle \text{pow } 3 \ 27 \rangle$ and then to $3^{27} = 7625597484987$, in two pops. $\langle \text{pow } \langle \text{pow } 3 \ 3 \rangle \ 3 \rangle$ pops to $\langle \text{pow } 27 \ 3 \rangle$ and then to $27^3 = 19683$, also in two pops.

Exercise 5.1. The term is not well-sorted: the first operand has sort **Num** where **rotate** requires **Image**, and the second operand has sort **Image** where it requires **Num**. Evaluation pops $\langle \text{add } 1 \ 2 \rangle$ to 3 and stops at the inert circle $\langle \text{rotate } 3 \ I \rangle$, after one pop.

Exercise 5.2. The term $\langle \text{pow } 2 \ \langle \text{sub } 0 \ 1 \rangle \rangle$ is well-sorted, with sort **Num**. It pops to $\langle \text{pow } 2 \ -1 \rangle$, which is blocked because **pow** is not defined for negative exponents.

Exercise 5.3. No. The second operand of **pow** is $\langle \text{sub } 5 \ 2 \rangle$, whose sort is **Num**, but the signature requires **Nat**. An operator of truncated subtraction, **monus** : $\text{Nat} \times \text{Nat} \rightarrow \text{Nat}$, with $\delta(a, b) = \max(a - b, 0)$, would restore well-sortedness, provided the numerals 5 and 2 are accepted as values of sort **Nat**.

Exercise 6.1. $\sigma(T_3) = \langle \text{pow } 3 \ \langle \text{pow } 3 \ 3 \rangle \rangle$, which pops to $\langle \text{pow } 3 \ 27 \rangle$ and then to 7625597484987, in two pops.

Exercise 6.2. The first stage pops $\langle \text{mul } 2 \ 3 \rangle$ and gives $\langle \text{add } 6 \ \langle \text{mul } x \ 4 \rangle \rangle$, after one pop. For $x = 5$, the second stage gives $\langle \text{add } 6 \ \langle \text{mul } 5 \ 4 \rangle \rangle$, then $\langle \text{add } 6 \ 20 \rangle$, then 26, in two pops. The total is three pops, equal to the size of the original term.

Exercise 6.3. Take $t = \langle \text{add } 1 \ x \rangle$ and $\sigma(x) = 2$. Then $\text{nf}(t) = t$ and $\sigma(\text{nf}(t)) = \langle \text{add } 1 \ 2 \rangle$, whereas $\text{nf}(\sigma(t)) = 3$. They differ, but $\langle \text{add } 1 \ 2 \rangle \rightarrow 3$.

Exercise 7.1. The results are **da/eb/fc**, **fed/cba**, and **cf/be/ad**. Three successive quarter-turns are the same as a rotation by 270.

Exercise 7.2. A rotation sends the cell at position (i, j) to a new position $\pi(i, j)$ determined by the rotation alone, and leaves its contents unchanged. A recolouring replaces the contents k of a cell by c if k is a colour and leaves a blank unchanged, and does not depend on the position. So both orders send the contents k at (i, j) to c or blank at $\pi(i, j)$, according to whether k was a colour or blank.

Exercise 7.3. No. The heights are 2 and 3, so the tuple is not admissible and the circle is inert. Inside a larger term the circle remains, and so do all enclosing circles, so the larger term is blocked.

Exercise 7.4. The defining term has four circles. Popping the two inner rotations gives bd/ac and f/e , and the **beside** circle then gives bdf/ace . The final rotation by 90 gives ab/cd/ef . There are four pops.

Exercise 8.1. Leftmost innermost: $\langle \text{add } x \ 0 \rangle \rightarrow x$ by (R3), then $\langle \text{mul } x \ 0 \rangle \rightarrow 0$ by (R7), two steps. Outermost: (R7) applies at the root at once and gives 0, in one step. The normal form is 0 in both cases and the lengths are 2 and 1.

Exercise 8.3. Both rules apply to $\langle \text{pow } 0 \ 0 \rangle$, the first giving 1 and the second giving 0. Neither 1 nor 0 can be rewritten further, so the critical pair is not joinable, and the term has the two distinct normal forms 1 and 0.

Exercise 8.4. With each numeral interpreted as 2, the term $\langle \text{mul } 2 \ \langle \text{add } 3 \ 4 \rangle \rangle$ has interpretation $2 \cdot (2 + 2 + 1) = 10$. The rewritten term $\langle \text{add } \langle \text{mul } 2 \ 3 \rangle \ \langle \text{mul } 2 \ 4 \rangle \rangle$ has interpretation $(2 \cdot 2) + (2 \cdot 2) + 1 = 9$. The decrease is $[a] - 1 = 1$, as predicted.

Appendix B

Summary of Notation and Rules

Syntax

Symbol	Meaning
$\langle o \ t_1 \cdots t_n \rangle$	circle with operator o and operands $t_1, \dots, t_n$
Num, Color, Image, Sym	sorts of the core fragment; Nat is introduced in Chapter 5
δ_o	the partial function giving the meaning of operator o
$\text{dom}(\delta_o)$	the admissible operand tuples of o
$\text{size}(t)$	number of circles in t
$\text{depth}(t)$	greatest nesting of circles in t
$t _p$	subterm of t at position p
$t[p \leftarrow s]$	t with the subterm at p replaced by s
$\sigma(t)$	substitution σ applied to t
$t \longrightarrow t'$	t pops to t' in one step
$t \longrightarrow^* t'$	t pops to t' in zero or more steps
$\text{nf}(t)$	the normal form of t
$s \equiv t$	s and t have the same normal form

Reduction

$$\begin{array}{lll}
 \langle o \ v_1 \cdots v_n \rangle \longrightarrow \delta_o(v_1, \dots, v_n) & \text{if } (v_1, \dots, v_n) \in \text{dom}(\delta_o) & (\text{Pop}) \\
 \langle o \ \dots t_i \dots \rangle \longrightarrow \langle o \ \dots t'_i \dots \rangle & \text{if } t_i \longrightarrow t'_i & (\text{Arg})
 \end{array}$$

Operators used in the text

Operator	Signature	Admissible when
add, sub, mul	$\text{Num} \times \text{Num} \rightarrow \text{Num}$	always
pow	$\text{Num} \times \text{Num} \rightarrow \text{Num}$	exponent is non-negative
rotate	$\text{Image} \times \text{Num} \rightarrow \text{Image}$	angle is a multiple of 90
recolor	$\text{Image} \times \text{Color} \rightarrow \text{Image}$	always
beside	$\text{Image} \times \text{Image} \rightarrow \text{Image}$	images have equal height

Main results

Result	Statement
Proposition 3.5	Each pop lowers the size by exactly one.
Theorem 4.1	Every sequence of pops is finite, of length at most the size.
Lemma 4.2	Two different pops from the same term join in one step each.
Theorem 4.3	Every term has a unique normal form.
Corollary 4.4	All maximal pop sequences have the same length.
Proposition 4.5	Parallel popping needs exactly as many rounds as the depth.
Theorem 5.2	Popping preserves sorts.
Theorem 5.3	A well-sorted blocked term has an inadmissible innermost circle.
Theorem 6.4	Evaluating with symbols first, then substituting, agrees with substituting first.
Theorem 8.4	The arithmetic simplification rules are terminating and confluent.

References and Further Reading

Bibliography

- [1] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998. The standard textbook on termination, confluence, and critical pairs, and the source for the criteria used in Chapter 8.
- [2] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, revised edition, 1984. The reference for the Church–Rosser theorem and for the substitution lemmas mentioned in Chapter 9.
- [3] G. Huet. Confluent reductions: abstract properties and applications to term rewriting systems. *Journal of the ACM* 27(4):797–821, 1980. Source of the critical-pair lemma and of confluence for orthogonal systems in their modern form.
- [4] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon, 1970. The original source of critical pairs.
- [5] M. H. A. Newman. On theories with a combinatorial definition of “equivalence”. *Annals of Mathematics* 43(2):223–243, 1942. The lemma that a terminating and locally confluent system is confluent.
- [6] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002. An introduction to sorts and types, preservation and progress, and the simply typed lambda calculus.
- [7] B. K. Rosen. Tree-manipulating systems and Church–Rosser theorems. *Journal of the ACM* 20(1):160–187, 1973. Confluence for orthogonal rewriting systems.
- [8] W. W. Tait. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic* 32(2):198–212, 1967. The proof that the simply typed lambda calculus is strongly normalizing.