INCREMENTAL REDUCTION IN THE LAMBDA CALCULUS AND RELATED REDUCTION SYSTEMS

by

John Henry Field

This report is a revision of the author's dissertation, presented to the faculty of the Graduate School of Cornell University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

INCREMENTAL REDUCTION IN THE LAMBDA CALCULUS AND RELATED REDUCTION SYSTEMS

John Henry Field, Ph.D.

Cornell University 1991

An *incremental* algorithm is one that computes a function repeatedly on a series of inputs that differ only slightly from one another, yet avoids unnecessary duplication of computations from one input to the next.

This thesis is a study of incremental computation in a general class of *reduction systems*, focusing particularly on the untyped lambda calculus and term rewriting systems. We also study implementation techniques used for such reduction systems and the question of the existence of *optimal* reduction strategies.

Most of our results are based on a new class of reduction systems we define called *regular replacement systems*. These systems combine an abstract reduction relation with a notion of structure defined by a *Brouwerian algebra*. The reduction relation defines a computation system, while the algebra is used to define what constitutes an *incremental* change to some object. Numerous reduction systems, including many term and graph rewriting systems and a variant of the lambda calculus, constitute regular replacement systems. Among our results is a generalization of the lattice-theoretic reduction theory of Lévy to regular replacement systems.

This theory makes it possible to give a precise and intuitively appealing definition of what it means for both incremental and non-incremental computations to be *optimal*.

We then investigate a family of term rewriting systems centered around a system we call **ΛCCL**. Each of these systems is capable of simulating a normalizing reduction strategy in the lambda calculus. However, unlike the lambda calculus, the notion of *substitution* is an explicit part of **ΛCCL**'s semantics, rather than being relegated to the status of meta-theory.

Our study culminates with the definition of a **ΛCCL**-based incremental reduction algorithm. This algorithm is optimal, yet it is simple enough to allow a practical implementation. We believe that appropriate generalizations of the ideas embodied in the algorithm can be used in a variety of practical settings, particularly those in which an algorithm is expressed in a functional or applicative manner.

Biographical Sketch

John Henry Field was born in Northfield, Vermont on July 10, 1961. After a brief stint in New England, the remainder of his childhood was spent in North and South Carolina, beneath the sharpened needles of the southern pine and above the slippery slopes of Piedmont red clay.

He survived the upheavals of the 60's, young enough to enjoy that decade's peripheral oddities without being old enough to worry about the more fundamental social changes that produced them. During the tumultuous years of school desegregation in the South, he was shuttled around on school buses to a variety of perfectly decent secondary public schools attended by children from both sides of the tracks. Unaware of the prevailing wisdom that such an experience would induce traumatic culture shock and stunt his intellectual growth, he found it all great fun.

During his formative years, he professed an interest in science, engendered particularly by the then popular gee-whiz journalism that emphasized the superficially grandiose aspects of technology (house-sized computers with walls of twinkling lights, atom smashers emitting mysterious blue glows). His enthusiasm for computing was considerably diminished, however, after discovering that computers were neither encyclopedic question-and-answer machines (How long is the Nile, Mr. Computer?), nor did they possess resonant metallic voices.

He attended Princeton University under the guise of an electrical engineer, spending much of his spare time in musical pursuits. Although briefly tempted to make a permanent career out of music, he surprised no one more than himself by deciding that computing was rather interesting after all, and, still ignorant of the length of the Nile, decided to concentrate his studies in computer science. He received a Bachelor of Science in Engineering, *cum laude*, in 1983.

After an abortive job interview with a small company specializing in arcade video games—in which he was all but summarily dismissed after making the mistake of asking the company's vice president what the company would do if their latest product, a game called *Food Fight*, turned out to be but a passing fad—he decided instead to pursue graduate work. Upon overhearing the Ph.D. program in Computer Science at Cornell University derisively described as the "Department of Theoretical Computer Science," he decided that it would be just the place for a confirmed videophobe. He immediately applied, was accepted, and received a Master of Science degree in 1987.

In 1989, he had the extraordinary good fortune to marry Elise Romanik.

Food Fight flopped, and the company in question has since turned to making circuit boards.

To my teachers:

To Elizabeth Hinson, who taught me what I know about writing.

To Nell Scoggins, who taught me some degree of patience and perseverance.

To Jacobo Valdes, who taught me that there really is some science in Computer Science.

To Tim Teitelbaum, who taught me how the world of research works.

To my wife, Elise Romanik, who taught me about life and love.

To my parents, who taught me just about everything else.

Acknowledgements

In addition to those mentioned in the dedication, I am indebted to many others, both for technical advice and for friendship.

I wish to thank particularly Thérèse Hardin and Jean-Jacques Lévy for extremely fruitful discussions and for their careful reading of some of the work presented here. Tom Reps also kindly provided helpful comments. Prakash Panangaden offered invaluable advice on numerous occasions, and Keshav Pingali was helpful in pointing out related work. Micah Beck patiently answered numerous questions about TeX, and Paul Taylor provided a nifty set of LaTeX macros for producing commutative diagrams. Raimund Seidel corrected my misunderstanding of Latin declension. I thank Dexter Kozen and William Austin for serving on my Ph.D. committee; I also thank Edward Murray for graciously agreeing to serve as a proxy.

I am grateful to my officemates—Wilfred Chen, Carol Critchlow, and John Reppy—for their forbearance and invariable good cheer in the face of thesis-related grumpiness and my not-so-infrequent monopolization of our workstation. William Austin, Trevor Stephenson, and Edward Murray were inspiring mentors in my musical endeavors. I thank Phil Hren and Andy Shreve, with whom I braved so many of the vicissitudes of graduate life, for their companionship as housemates.

Roger Hoover has gone far beyond the call of duty in helping to make my transition to the corporate world as smooth as possible.

Finally, in what is inevitably a futile effort to adequately recognize everyone whose friendship has been inseparable from anything I may have accomplished as a graduate student, I give my heartfelt thanks to Chris Bischof, Doug Crisman, Kieren Herley, Sue Manning, B. Natarajan, Steve Peckham, Tim Peierls, Steve Rapkin, Pat Stephenson, and Brad Vander Zanden.

Funding for this research was provided by NSF and ONR under NSF Grant CCR-8514862, and by ONR under contract number N00014-88-K-0594. I gratefully acknowledge their assistance.

Table of Contents

List of Figures

List of Definitions

Chapter 1

Introduction

"...self-repetition [is] a feature frowned upon by theorists."

—Donald Francis Tovey, *A Companion to Beethoven's Pianoforte Sonatas*

1.1 Incremental Computation

Our concern here is *incremental computation.* An incremental algorithm is one that takes advantage of the fact that the function it computes is to be evaluated repeatedly on inputs that differ only slightly from one another, while at the same time avoiding unnecessary duplication of computational effort from one input to the next.

Indulging in just a bit of rhetorical excess, one might regard this sort of computation as a metaphor for the modern industrial world, combining as it does both monotonous repetition and the inevitable requirement for efficiency. Thus it should not be surprising that the need for incremental algorithms arises quite frequently.

Our goal in this thesis is to contribute to making the development of efficient incremental algorithms no more difficult than that of their non-incremental counterparts. In the process, we also hope to shed some light on more fundamental questions that arise in conjunction with the study of such computations.

Exactly what is incremental computation good for? The twin hallmarks of an incremental computation are *repetition* coupled with *minimal change* at each iteration. Let us examine some practical problems where such requirements indeed exist:

1.1.1 User Interfaces

Complex input and output devices are essential to the modern computer. Splendiferous graphical capabilities and mechanical pointing devices are quickly becoming minimum requirements for desktop workstations. At the extreme, one might even find full-fledged audio-video capabilities or the ability to perform three dimensional simulation. These facilities all require that the software used to control a so-called user interface be able both to maintain a complex notion of state (i.e., the output) and to respond rapidly to a variety of stimuli (the input). While externally quite impressive, the internal requirements of the software that must control a user interface are considerably more mundane: these programs must perform a myriad of housekeeping tasks *quickly*.

A typical task might be to provide a "windowing" capability for a graphical display. A program with graphical output would ideally produce it under the illusion of its appearance in a fixed size display. Depending on how the user arranges the location and appearance of the window in which the program's output is to appear, the output would be "clipped" to fit. Often the user only wants to make a small change in the window's configuration; it would thus be desirable for the algorithm that manages the window to require recomputation commensurate with the small magnitude of the change.

Here, then, we have the essential ingredients for incremental computation: A potentially time-consuming algorithm (redrawing a graphical display), executed repeatedly by a user (as a result of moving windows around), usually making only minimal changes to the current configuration. Ideally, those incremental algorithms that are used in a user interface could be implemented *as if they were non-incremental*. In this case, it would be desirable for graphical software to behave as if it only had to deal with a fixed window, rather than one whose shape and location could be changed.

1.1.2 Programming Environments

The software development process is an endeavor whose intricacies have yet to be tamed. Error-free software that is produced at reasonable cost and without prohibitive time devoted to testing is disappointingly rare.

One tool that has been touted as an aid to producing better software is the so-called *programming environment*: a computer system for developing computer software. The idea is that the computer can serve as an interactive assistant to the software developer, doing much more than merely translating programs to executable form and guarding against rudimentary errors. Instead (in conjunction with some of the useful developments in user interfaces), programming environments can notify the programmer of potential problems *as the program is being developed*, instead of repeatedly iterating the cycle of compiling, testing, and correcting errors.

A programming environment might provide interactive facilities for sophisticated error checking, for checking consistency between a program and a formal specification, for transforming selected bits of code to improve performance, for performing the myriad sorts of tedious clerical jobs necessary to keep track of the parts of a large set of interacting modules, to control interactions among teams of programmers, and to account for different versions of the same program for different uses or for different machines.

Since most of the benefits of machine assistance for the task of developing software are negated if the software-making software is too slow to keep up with the changing requirements of the programmer, it is of paramount importance that the algorithms used to perform the tasks outlined above are efficient enough to keep pace. Once again, incremental algorithms are called for: the effect of the typical change to a program will often be minimal, and an incremental algorithm that can rapidly perform computations necessary to determine whether such a change yields a potential error would be desirable.

1.1.3 Interactive theorem proving systems

Interactive theorem proving systems are similar in many respects to programming environments. Here the desired end is a true theorem, rather than a working program (although *proving* a program correct is clearly a closely related endeavor). As with programming environments, there are numerous clerical tasks to be performed in interactively verifying a large proof. In addition, automated assistance in proving certain simple lemmas is also desirable. Systems of this sort, e.g., NuPRL [Ce86] often have the capability to define and execute new proof strategies, or "tactics." Such strategies may be regarded as *functions* that take an incomplete proof as an argument and either produce a computed proof or fail. The system that carries out the strategy thus implements a *higher-order* function, whose input is a proof strategy that may be refined interactively.

Therefore, in the case of an interactive theorem proving system, we may have a function that is incrementally altered (the proof strategy) while its nominal input (the incomplete proof) remains unaltered. The capability to re-apply an altered proof strategy without performing redundant work would be valuable. Incremental computation of higher-order objects could thus reduce the computational overhead required to produce a new proof based on a modified strategy or altered premise.

1.1.4 Compilers

Many hardware designs, particular those with reduced instruction sets or vector operations, require an extremely sophisticated compiler to make the most of the hardware. Algorithms for instruction scheduling heuristics, code rearrangement, register allocation, and loop transformation are among those whose computational requirements are considerable. In all of these cases, incremental re-evaluation would clearly be useful for reducing the time required to compile a program after a small change. Even the more mundane tasks usually performed by a compiler, such as parsing and symbol table management, can benefit from incremental algorithms.

Even within a single compilation, incremental computations may prove valuable. Consider, for example, an optimization phase requiring computation of a data flow graph. Based on this information, the compiler may select a code-improving transformation, which may itself cause some modification to the control flow graph. In general, any change to the control flow graph invalidates the derived data flow information, which must then be updated before the next transformation can be selected and applied. It should thus be evident that any program that alternates data analysis with data transformation may be a candidate for incremental evaluation.

1.1.5 "Real-time" systems

Many of the settings in which incremental computation is desirable are interactive. However, there are numerous additional situations in which incremental computation would be valuable. For instance, in most so-called real-time applications, e.g., electromechanical control systems, quick response to asynchronous changes in individual sensor values is required: an aerodynamic control system for an airplane has to recompute outputs for control surfaces continuously, even though most of its sensor values will remain constant during any short interval of time. An incremental algorithm would make it possible to recompute only those outputs that depend on inputs that have changed, avoiding unnecessary computations pertaining only to unchanged values. Moreover, such an algorithm would be much easier to write if it were to take account of *all* sensor values, whether changed or not, and then transformed automatically to incremental form.

1.1.6 Program Optimizations

The work of Paige and Koenig on "finite differencing" in program optimization [PK82] can be viewed as an effort to use incremental techniques to improve the performance of algorithms not intended to process incremental changes to inputs at all. Their technique calls for replacing expensive calculations made inside iterative program constructs (such as set operations in "very high-level" programming languages) with semantically equivalent, though more efficient, incremental counterparts. These transformations take advantage of small changes made to large data structures inside program loops.

An earlier paper of Earley [Ear76], on which Paige and Koenig's work is partially based, observed that such optimization techniques could prove equally beneficial in implementing incremental algorithms, as opposed to using them only to improve non-incremental ones.

1.2 A Framework for Incremental Algorithms

To make a bit more precise what functional form an incremental algorithm should take, we define in this section a simple algebraic model of incremental computation that incorporates two distinct principles: taking advantage of information from previous computations in computing new results, and determining the effect on the output of a relatively small *change* to an input.

This model is intended as a minimal framework in which incremental algorithms can be related to their non-incremental counterparts. It is not intended as a tool for analysis of performance or for comparison of different incremental techniques.

1.2.1 Preliminaries

We begin with an algorithm, A_f, implementing a function

$$f : X \rightarrow Y$$

where X and Y are arbitrary sets.

In order to transform A_f into incremental form, we must first make precise what we mean by *increment.* We are concerned with expressing ways in which values in f's domain, X, may differ. In many mathematical contexts, the notion of difference is obtained "by default." For example, if g_1 and g_2 are elements of a group G, their difference is given by $g_1 + (-g_2) = g_3$, which is also an element of G. We wish to define a weaker notion of difference, applicable to a broader class of problems. Here, we will not assume the domain of f comes pre-equipped with an additive law of composition or an inverse.

Instead, we will use the notion of an *increment set*, Δ. An increment set can be thought of as a set of objects that encode allowable changes to X. For example, if X is the set of real numbers, we might let Δ be $[0 \ldots 1)$, a restricted interval. For a more concrete example, let X consist of the set of states of some electro-mechanical control system. Then Δ might consist of values encoding information of the form "sensor i has become {active, inactive}". Such information could then be used to update the system state accordingly.

Given sets X and Δ, we need a way to put them together. For this, we will use a partial composition operator

$$\oplus : (X \times \Delta) \rightarrow X$$

where for any $\langle x_1, \delta \rangle$ in the range of $\oplus$, there exists $x_2 \in X$ such that $x_1 \oplus \delta = x_2$. Informally, $\oplus$ applies one of the "editing operations" in Δ to an element of X, producing a new element in X. It will be useful to require that Δ contain an *identity element*, δ^0, such that for all $x \in X$, $x \oplus \delta^0 = x$.

In the most general case, the increment set might be a collection of editing operations mapping a previous input to a new one. Δ might then consist of pairs of the form $\langle EditOp,\ y\rangle$, where $EditOp$ is one of the set

$$\{add, delete\}$$

and y is any value from X. In this case, for all $op \in EditOp$ and $x, y \in X$,

$$x \oplus \langle op,\ y\rangle \triangleq x\ op\ y \tag{1.1}$$

In other words, $\oplus$ simply applies op to the given value (y) and the previous input (x).

Δ and $\oplus$ come as a package; for a given X, we may define any number of $\langle \Delta, \delta^0, \oplus\rangle$ triples. We will refer to such a triple as an *incremental adjunct* to X. The aim of this notation is to keep the mechanism used to represent increments to X quite distinct from X itself.

1.2.2 Incremental form of a function

Let us say we have algorithm A_f in hand, and have defined an incremental adjunct $\langle \Delta, \delta^0, \oplus\rangle$ applicable to inputs from X. Our task is now to produce an algorithm A_{f_Δ} implementing f_Δ, an incremental form of f. As alluded to earlier, such an *incremental implementation* will in general exhibit the following characteristics:

(1) use an implicit representation of input, expressed in terms of a *change* to a previous input.

(2) use information from previous computations in computing a new value.

We will refer to (2) as *history* information, although this term is perhaps not ideal, since the "history" from prior computations may consist simply of the value of the previous input or output. In general, however, the history will also include intermediate information used to produce the previous output or outputs, which can then be reused to enable a new answer to be computed more efficiently. We will thus in general augment f's range with history information drawn from a new set, Z. Z could contain not only information useful for improving the efficiency of subsequent computations, but will also generally contain sufficient information to allow the incremental function to operate directly on elements of Δ, rather than X.

Combining these ideas formally, A_{f_Δ} implements a function of the form

$$f_\Delta : (\Delta \times Z) \rightarrow (Y \times Z)$$

where we relate the behavior of f_Δ to that of f by the following consistency condition:

If $f(x) = y$, $\langle y,\ z\rangle$ is a pair in f_Δ's range, $x \oplus \delta = x'$, for some δ, and $f(x') = y'$, then

$$f_\Delta(\delta, z) = \langle y',\ z'\rangle$$

for some $z' \in Z$.

Since f_Δ uses previous outputs to compute new ones, we need a way to start off a sequence of incremental computations. Thus we will require that for all $x \in X$, there exist $z_x \in Z$ such that the following holds:

If $f(x) = y$, $x \oplus \delta = x'$ *for some* δ, *and* $f(x') = y'$, *then*

$$f_\Delta(\delta, z_x) = \langle y',\, z' \rangle$$

for some $z' \in Z$.

This allows us to feed the result of a non-incremental computation of f on x into a subsequent sequence of incremental computations of f_Δ, using z_x for the first incremental one.

The nature of the history information Z depends on the particular problem or incremental algorithm in question. Note in fact that in the degenerate case where Z is *empty* and Δ equals X, f_Δ has essentially the same form as non-incremental f.

1.2.3 Classifying Some Examples

Using the quasi-formalism given above, we can classify some of the previous examples of problems suitable for incremental computation (we avoid proposing a domain for the history information (Z), since here we wish to describe only the problems, not propose incremental methods for their solution).

User Interfaces

In the case of the window display example, we can view X as display information produced by a program with graphical output. f would implement a function producing a screen bitmap from the display information. Δ could encode moving and resizing information produced by the user, along with new display information produced by the program. $\oplus$ would then yield the bitmap produced by a resized window. A_{f_Δ} then produces a new bitmap (as quickly as possible) given the change information.

Programming Environments

We let X be the set of program syntax trees, and, f be, e.g., a type-checking algorithm. Δ would consist of syntax tree-pruning operations, and $\oplus$ would produce a new tree given the edit updates. The requirements for a compiler are similar, although the computations performed (e.g., register allocation) might be different.

Theorem Proving Systems

Here, we let X consist of a set of premises, lemmas, proof tactics, and a partial proof tree. Δ and $\oplus$ could encode and apply changes to any of the above. f would then determine the validity of the current partial proof.

Real-Time Systems

We can treat X as a vector consisting of the collection of sensor data. Δ would be a pair consisting of a vector index and a new value for the corresponding coordinate, with $\oplus$ replacing the old value at the given coordinate with the new one. f would then compute the settings of the control parameters.

Program Optimizations

In the Paige-Koenig system described above, X is a family of sets defined over a given domain of elements (specified in the language being compiled). f is a set-valued expression computed inside a program loop containing some set-valued variable (say w) whose value changes at each iteration, Δ is the set of increments by which w may change, and $\oplus$ is some set operation, e.g., $\cup$, which is used to alter w by Δ at each iteration. At each iteration of the loop, f must be incrementally updated given a change to w.

From the examples listed above, it should be clear that our framework is quite broad. In the sequel, we will restrict our attention to input, output, and Δ sets having the form of a *Brouwerian algebra*, and functions f whose algorithms are specified in a *functional* manner.

1.3 On-Line, Off-Line, and Incremental Algorithms

The distinction between incremental and non-incremental computation is sometimes confused with that made between *on-line* and *off-line* algorithms. (See, e.g., [AHU74, p. 109]). An on-line algorithm implements a function (say g) in the conventional way: given a single input, it immediately produces a single output. The off-line form of an algorithm is intended to compute g on each of a *set* of inputs supplied simultaneously. By grouping inputs together, it is frequently possible to compute the function on the elements of such sets of inputs more quickly than would be possible had each input and its corresponding output been computed separately. The problems for which this distinction is often made usually take the form of queries to some database.

By contrast, an incremental algorithm is still intended to compute a single output given a single input; however, each input is specified *implicitly* using some increment. Note that in a sense, the notion of an incremental algorithm is more general than that of an off-line algorithm, since one can always use an incremental algorithm to efficiently compute a set of related queries, but it is not always possible to twist an off-line algorithm into a form suitable for incremental input.

1.4 Approaches to Incrementality

We can distinguish between two general approaches to incremental computation: ad hoc algorithms, and general (or automatically-generated) techniques.

1.4.1 Ad Hoc Incremental Algorithms

Most previous research in incremental algorithms has concentrated on finding ad hoc incremental algorithms for specific problems (most of which already have well-known efficient non-incremental algorithms). Here are some examples:

Polynomial Evaluation

Perhaps one of the earliest problems for which specialized incremental techniques were found appropriate is the computation of polynomials by *finite differencing*. One can compute the value of a polynomial for values spaced at a fixed interval using *difference equations* (discrete analogues of derivatives) much more efficiently than is possible were each value to be computed from scratch. As Paige and Koenig [PK82] point out, this method can be attributed to the sixteenth century English mathematician Henry Briggs, who used it to compute mathematical tables. Charles Babbage's nineteenth century analytic difference engine was intended to compute such tables mechanically.

Data Flow Analysis

Rosen [Ros81], Zadeck [Zad84], Ryder and Carroll [RD87], and Marlowe [Mar89], among others, describe incremental approaches to regenerating program data flow graphs efficiently given small changes to the program being analyzed.

Graph Algorithms

Yellin [Yel88] has proposed an incremental algorithm to compute the transitive closure of a graph. Adding or deleting sets of edges need not require that the entire graph be traversed to recalculate the closure; instead, only those edges which impinge (either directly or transitively) need be considered. Spira and Pan [SP75] and Chin and Houck [CH78] have given algorithms for updating minimum cost spanning trees efficiently.

Matrix Computations

A number of matrix algorithms have been proposed that perform computations on matrices for which only a single element, row, or column has been changed. For instance, Daniel et al. [DGKS76] have proposed an efficient algorithm for computing the so-called QR-factorization given a rank-1 matrix update.

Constraint Solution

Van der Zanden [Van89] gives an algorithm for efficient solution of systems of constraints in which only some of the constraints change in between each solution. Such constraint solution turns out to be particularly useful in interactive graphical systems.

The list of ad hoc incremental algorithms given above is by no means intended to be exhaustive—there are countless other examples of such algorithms in problem domains ranging from computational geometry to searching and sorting (for a more comprehensive list, an excellent source is Marlowe [Mar89, pp.62–70]).

1.4.2 General Approaches to Incremental Computation

Each of the problem domains discussed above offers unique opportunities for exploiting incrementality. Unfortunately, considerable (and perhaps prohibitive) effort may be required to develop an incremental algorithm by hand each time a new problem arises, or to take advantage of improved non-incremental algorithms for existing problems. We will thus be concerned primarily with examining *algorithm-independent* approaches to incrementality, in which an incremental algorithm A_{f_Δ} is produced directly (and automatically) from non-incremental algorithm A_f. Given an arbitrary algorithm, the task of finding an equivalent incremental one automatically may initially appear daunting, perhaps even undecidable. Note, however, the crucial distinction between an *algorithm* and the *problem* that it solves: Though a particular algorithm may be amenable to all manner of productive analysis (for example, optimizing compilers use heuristic algorithmic transformations to produce implementations having improved performance), the corresponding problem is often difficult even to *express* formally, much less to analyze mechanically.

One should think of an "incremental analyzer" as a tool to alter the form, more than the content, of the original algorithm. The derived incremental algorithm should yield performance commensurate with the quality of the original. A particularly clever and crafty algorithm should result in an incremental equivalent that is at least as capable; a ponderous and plodding one cannot be expected to produce an incremental version that magically compensates for its deficiencies. While automatic transformation of algorithms into incremental form should not be seen as a panacea, it has the potential to greatly enhance the utility of the original.

Exactly what sort of performance can we reasonably expect? While we delve into this issue in somewhat greater detail in Section 9.6, we can sketch out some of the extremes: At a minimum, an incremental implementation, derived automatically, must perform no worse than the original algorithm. Indeed, certain classes of problems are likely to resist *any* attempt at providing incremental implementations that perform asymptotically better than their non-incremental antecedents. Algorithms for problems in the *NP-complete* class, for instance, seem to require exhaustive analysis of the space of feasible solutions. An incremental perturbation of the input to such algorithms apparently requires equally arduous re-analysis. At the other end of the spectrum, an incremental algorithm developed using *ad hoc* techniques may outperform one produced automatically.

The ultimate criterion for success of automatically-generated incremental algorithms is clearly that an incremental algorithm generated from the best non-incremental one for the problem be as efficient as any ad hoc incremental algorithm for the same problem. However, even if this goal cannot always be achieved, we believe that there is a broad middle ground in which automatic transformation of many algorithms into incremental form can produce significant and practical improvements in efficiency with minimal effort.

1.4.3 Characterizing Incremental Computation

Our goal in incremental algorithm generation can be viewed as the determination of a semantics-preserving program transformation that has the functional form given in Section 1.2. In achieving this end, there are a number of inextricably linked considerations to be balanced, many of which can be directly expressed in terms of some aspect of our formal notation. They include:

- Choice of domain and range sets used by the original (non-incremental) algorithms (allowable values of the sets X and Y).
- Definition of incremental adjunct ($\langle \Delta, \delta^0, \oplus \rangle$).
- Choice of notation for the original algorithm.
- Space and time required for the incremental algorithm (often related to the use of history information, Z).
- Domain of intended application (e.g., interactive, computation-intensive, real-time).

For some approaches to the problem, the value of making a particular set of choices from the range of possibilities listed above can be easily quantified. Other issues are more subtle; for instance, we would like the language used to express the original algorithm to be expressive and natural, and the choice of algorithmic domain and incremental adjunct to "fit" well with the intended application.

1.5 Existing General Systems

In comparison to the large number of ad hoc incremental algorithms that have been developed, only a small number of algorithm-independent approaches to incremental computation have been proposed. By "algorithm-independent" or "general," we mean systems for computation in which a *non-incremental* algorithm may be incrementally evaluated, or may be translated to some incremental form. Prior examples of such general systems include the following:

- Yellin and Strom [YS89] describe a special programming language where networks of functions on aggregate data structures can be defined. Small changes to such aggregates are reflected in efficient propagation of updates throughout the network.

- Demers, Reps, and Teitelbaum [RTD83,Rep82] have proposed algorithms to evaluate incrementally functions on syntax trees described by attribute grammars.

- Alpern, et al. [ACR+87] describe a generalization of incremental attribute grammar evaluation for arbitrary graphs.

- Pugh and Teitelbaum [PT89] discuss incremental evaluation of functions using "memoizing" techniques.

- Bengelloun [Ben82] describes an incremental evaluator for the language Scheme in which list structures manipulated by functions may be incrementally updated.

- Sundaresh and Hudak [SH90] have given a framework for incremental computation based on partial evaluation.

As interesting as many of the above approaches are, most have restrictions that limit their applicability. In some cases, they require that algorithms be specified over a restricted domain, e.g., a set of special data structures. In other cases, efficient incremental evaluation is possible only for a subset of the set of operations that may occur in an algorithm, or only for algorithms having a particular form. The work of Sundaresh and Hudak is perhaps closest in spirit to our work; however, it is described as primarily a "framework" for understanding incremental algorithms—its operational aspects are only sketched out. The systems listed above and their relation to the work in this thesis will be discussed in greater detail in Chapter 9.

Our goal here is to investigate both the theoretical and operational aspects of a very general, and in some respects, more fundamental model of incremental computation than those proposed heretofore: *incremental lambda reduction.*

1.6 Incremental Lambda Reduction

The untyped lambda calculus (henceforth simply referred to as "the lambda calculus") is an appealingly simple and much-studied formalism in which functions are defined as rules for generating new values from old ones. Despite this (somewhat deceptive) simplicity, the lambda calculus can be used to encode any computable function whatsoever.

The advantage of investigating incremental reduction in the lambda calculus is thus its extreme generality: one can use it to describe incremental changes both to functions (including higher-order functions) and to data structures. The lambda calculus also has a rich reduction theory that can be exploited to analyze the limits of incremental techniques.

The lambda calculus is the archetype for a variety of *functional*, or *applicative* programming languages, in which programs are composed entirely of side-effect free functions. Once viewed by efficiency fanatics as something of an eccentricity, functional programming has in recent years been held in higher esteem, particularly by those concerned with producing correct programs with notational economy and mathematical precision. (To be fair, however, functional computation is far from a panacea; though almost any problem can be solved functionally, many require rather unnatural contortions to fit within the functional framework).

Functional languages and the lambda calculus are well suited to computation by *reduction*, where a single computational step consists of the transformation of a program to an equivalent but (usually) simpler form. The reduction process starts with the original program and continues until no further simplifications are possible.

Our interest in using λ-reduction for *incremental* computation stems primarily from the following:

- Since computation in the lambda calculus proceeds by transforming programs to programs, intermediate computations have a natural representation as partially reduced subexpressions, which can be saved to avoid unnecessary recalculation.

- In the lambda calculus, the minimal distinction between program and input, function and higher-order function, means that the same incremental techniques are applicable to a variety of sorts of computation.

- Since "programs" in the lambda calculus are simply expression trees, there is a natural notion of *increment* corresponding to an update to the expression tree. Indeed, the distinction between tree transformations resulting from the reduction process and those resulting from an incremental change to the initial expression will turn out to be minimal.

1.7 Goals

While we argue that incremental lambda reduction has broad applicability, it is important to make clear what we will *not* try to do here: we do not propose a language or system into which one can drop one's favorite algorithm, out of which will miraculously pop an efficient incremental analogue.

Instead, we view our work in this thesis primarily as foundational: by outlining theoretically sound techniques for general incremental computation using lambda reduction, we wish to lay the groundwork for truly practical applications of these ideas. We have chosen the lambda calculus for this study precisely because it is the computational formalism closest to a usable programming language. While the lambda calculus has been the object of considerable theoretical study, there has also been a great deal of effort devoted to the practical end of making functional programming languages based on the lambda calculus as efficient as their more traditional imperative counterparts. These efforts have achieved a great deal of success, and we believe that the application of many of the implementation techniques that have been developed in non-incremental settings to the incremental algorithms developed here will be quite fruitful.

1.7.1 Results and Contributions

We outline below what we regard as the major technical contributions of the thesis:

Regular Abstract Replacement Systems

We present a new formalism based on *Brouwerian algebras* for the study of a broad class of reduction systems. We will call such a system a *Regular Abstract Replacement Systems*, or RRS. RRS's provide a unified treatment of the reduction theory (i.e., operational semantics) of certain graph rewriting systems, conditional and unconditional term rewriting systems, and the lambda calculus (the latter with a graph rewriting, rather than term rewriting semantics). The bulk of the reduction theory is a generalization in an abstract setting of work by Berry and Lévy [Lév78,BL79,Lév80]. Many existing reduction-theoretic results for term rewriting systems and the lambda calculus can be recast in this more general framework. Brouwerian algebras also provide a general "theory of editing" that will be used to provide a formal notion of *increment*.

ΛCCL

We define a new term rewriting system called **ΛCCL** that provides a formal account of the substitution process in the lambda calculus, as implemented using the common devices of *environment* and *closure*. **ΛCCL** is based on Curien's *Categorical Combinatory Logic*, but possesses many useful properties that do not hold for Curien's system. We show that the equational theory of **ΛCCL** is a *conservative extension* of the theory of β-equality in the lambda calculus. **ΛCCL** also extends the *β-reduction* relation (a stronger property than extension of the equality relation). We will take advantage of the close connection between λ-reduction and **ΛCCL** reduction by using **ΛCCL** as the basis for our incremental reduction algorithm. We also show that ΛCCL has a number of useful properties independent of its relation to the lambda calculus. Some of the proofs of these properties are interesting in their own right.

Optimality and Non-Optimality

We examine the problem of *optimal* reduction strategies for the lambda calculus, and show that the standard graph reduction method of implementing term rewriting in ΛCCL is insufficient to implement optimal λ-reduction strategies.

Forks

We introduce a simple new data structuring device called the *fork* node, which, in conjunction with graph rewriting techniques, makes it possible to maintain multiple λ-terms in a single graph. This device turns out to be the key to maintaining sets of incrementally evaluated subexpressions with reasonable efficiency. The formal behavior of the fork is embodied in $\triangle\Lambda$CCL, an extension of ΛCCL.

Projection and Incrementality

We define a new variant of Berry and Lévy's notion of reduction *equivalence* for term rewriting and λ-reduction, which provides the basis for a formal definition of "overlapping" reductions. Maximally incremental reductions are then defined as those that are non-overlapping. We define a new notion of *projection* of a reduction on substructures that is used to link editing operations to reductions on the structures thus edited.

ΛCCL$'$

We define a subsystem of ΛCCL, ΛCCL$'$, which (unlike ΛCCL itself) possesses an *optimal* graph rewriting implementation.[1] This system is then used to define a novel evaluation strategy for the lambda calculus that yields non-overlapping, and thus maximally incremental reductions. ΛCCL$'$ is *weak-head normalizing*, i.e., all λ-terms that possess a weak head normal form are reducible by ΛCCL$'$ to weak head normal form. This result extends to terms possessing *head normal forms* or *(strong) normal forms*. ΛCCL$'$ is thus sufficiently powerful to implement a practical reduction scheme.

Λ^{inc}

The ideas above are all embodied in an incremental reduction algorithm for ΛCCL called Λ^{inc}. Λ^{inc} reduces its initial term to a weak head normal form, if one exists. In the process, information about the reduction is retained for subsequent use. If any of a predefined subset of the set of initial terms is then

[1] Although ΛCCL$'$ has an optimal implementation for rewriting in its own equational theory, it still cannot be used to produce an optimal implementation of *β-reduction*.

replaced, the information from the previous reduction is used to avoid repeating any computations when reducing the term thus altered. The resulting incremental reduction is *optimal* in that the number of reduction steps required to reduce the edited term is minimal in a precise sense. Since **ΛCCL** effectively implements the λ-calculus, we regard Λ^{inc} as an incremental λ-reduction algorithm.

1.7.2 Grand Tour

The contents of the thesis are arranged as follows:

Chapter 2 is an introduction to computation by reduction and, in particular, to the lambda calculus and term rewriting systems. It also contains an introduction to *graph reduction* in the lambda calculus and an overview of incremental lambda reduction.

Chapter 3 contains most of the technical preliminaries required for the rest of the thesis, which are not inconsiderable, due to the complexity of the existing literature on reduction theory. We review the work of Huet on abstract properties of reduction relations, provide the basic properties of Brouwerian algebras, and introduce the new notion of regular abstract replacement system, or RRS. The chapter concludes with a study of the reduction theory of RRS's.

Chapter 4 covers the formal background required for the lambda calculus, the de Bruijn lambda calculus, Lévy's *labeled* lambda calculus, and term rewriting systems. We discuss the Brouwerian algebras of graphs and terms and their relationship to one another. We show that each of an important class of conditional and unconditional term rewriting systems is an RRS. Finally, we sketch a version of the lambda calculus with a graph-rewriting semantics that allows it to be treated as an RRS.

Chapter 5 introduces **ΛCCL** and contains an extensive discussion of its properties. In particular, we show the close connections between it and the lambda calculus. We give a simple interpreter for the lambda calculus using **ΛCCL** that illustrates both its practical and theoretical utility.

Chapter 6 discusses the problem of *optimal* implementations of RRS's and examines the role of graph reduction in producing optimal implementation of term rewriting systems. We then demonstrate that a graph rewriting implementation of **ΛCCL** cannot be used to implement optimal reduction schemes for the lambda calculus.

In Chapter 7, a theory of incremental reduction applicable to RRS's is developed. The basis for this theory is the notion of *projection* of a reduction on a subcomponent of its initial term. We also show the relationship between computation of projections and optimal computations.

Chapter 8 introduces several variants of **ΛCCL**. One of these variants, **ΛCCL′**, has particularly pleasing properties that make it a suitable basis for implementing incremental reduction. We show that $\mathbf{\Lambda CCL'}$ is weak head normalizing, and, unlike $\mathbf{\Lambda CCL}$, that it is an RRS. We next give a weak head normalizing graph reduction algorithm that makes use of $\mathbf{\Lambda CCL'}$. We extend $\mathbf{\Lambda CCL'}$ in such a way as to be able to preserve the results of intermediate computations on elements that might later be incrementally updated. Finally, we conclude the chapter by presenting $\Lambda^{\mathbf{inc}}$, our incremental reduction algorithm.

In Chapter 9, we discuss other general methods for incremental computation and compare them with incremental lambda reduction. We show that many of the drawbacks of these systems are remedied by the methods described here.

Finally, in Chapter 10, we discuss applications for which incremental lambda reduction or variants thereof are particularly well-suited. We outline a number of directions for additional research, and conclude with some final words about the implications of incremental techniques.

Chapter 2

Computation by Reduction: Formal Systems and their Implementation

"Entry of main theme."

"[He] is here biting with epigram and crowding as many different ideas into his kaleidoscope as will permit it to turn at all."

—Tovey

2.1 Computation by Rule

In this chapter, we will introduce two closely related formulations of functions as *rules* or *templates*: the lambda calculus and term rewriting systems. We will also give a brief introduction to the notion of *graph reduction* in the lambda calculus, a natural implementation technique that enables the full expressive capabilities of the lambda calculus to be used without an unreasonable penalty in efficiency. Graph reduction can also be used to implement term rewriting. In the sequel, we will discuss the tradeoffs present in various implementation techniques for the lambda calculus, including the question of the existence of *optimal* reduction techniques. Finally, given this background, we will outline our goals in producing an incremental evaluator for the lambda calculus.

We will be concerned primarily with the intentional behavior of a function's computation, rather than its extensional manifestation as a set of pairs. We are thus interested in the *operational* properties of computation in these formalisms, rather than their model-theoretic or *denotational* characteristics. While the latter theory is extraordinarily rich and extensive, the operational semantics of the lambda calculus is surprisingly complex as well, though perhaps less well-known. As we are interested in obtaining efficient incremental implementations of languages whose meanings are already well understood, our concentration on the operational aspects of the lambda calculus should not seem unreasonable. As Klop [Klo80] has pointed out,

> We do not feel...that the availability of the powerful model theoretic methods lessens the usefulness of...syntactical theorems. The reason is the well-known fact that the (sometimes tedious) work of syntactical investigations yields longer proofs, but also more information.

We will be able to exploit such information in obtaining the results we will present here.

2.1.1 The Lambda Calculus

The lambda calculus is both a language and a system for computation of the algorithms specified in that language. Since computation proceeds by transforming expressions in the language from one to another, the two aspects of the formalism are inalterably intertwined. In the lambda calculus, the specification and transformation of rules is the essence—in fact, the entirety—of computation. Data can be encoded by (unevaluated) rules, and rules exist only to manipulate other rules. The language of *λ-terms* is thus remarkably spare, its syntax consisting of the following three sorts of expressions:

(1) *Variables*, e.g.

$$x, y, z, \ldots$$

drawn from an infinite set.

(2) *Abstractions*, expressions of the form

$$\lambda x.M$$

where x is an arbitrary variable, and M is an arbitrary λ-term (usually containing an instance of x).

(3) *Applications*, expressions of the form

$$(MN)$$

where M and N are arbitrary λ-terms.

Abstractions are the templates used to specify functions-as-rules. Variables are used in abstractions to specify the correspondence between a function's argument and the argument's appearance in an instance of the template specified by the abstraction. Applications are pairs consisting of a template (abstraction) and an arbitrary expression to be transformed. Consider the term Δ, defined as

$$\Delta \equiv \lambda x.(xx)$$

The x in λx is called the *bound variable* of the abstraction, and is a proxy for the yet-to-be-supplied argument to the template. The rest of the abstraction, (xx) is the *body* of the template, in which the appearance of two instances of x will cause an instance of the body with two copies of the argument to be created when the function is applied.

To supply an argument N to a function M, we simply juxtapose them, as

$$(MN)$$

(We will generally surround applications with enough parentheses to make clear which function is being applied to which argument.)

2.1.2 Computing with Lambda Terms

So far, we have specified only the expression language of the lambda calculus, not how to compute with it. For the latter, we first require the auxiliary notion of *substitution*. The expression

$$M[x := N]$$

is shorthand for an altered instance of M gotten by substituting N for all instances of x in M that are *not* parts of λ-abstractions whose binding variable is x. The latter subtlety is necessitated by that fact that various abstractions may contain the same bound variable, and we must have some rule to prevent their confusion. (Even with this proviso, substitution entails certain additional technical complications that we are ignoring here). Thus

$$(x\ \lambda y.(x\ \lambda x.x))[x := \lambda z.z]$$

is shorthand for

$$((\lambda z.z)\ \lambda y.((\lambda z.z)(\lambda x.x)))$$

Note that the x's *bound* by the inner abstraction remain after the substitution operation is carried out.

We can now define the computational operation of *β-contraction* by

$$((\lambda x.M)\, N) \quad \longrightarrow \quad M[x := N]$$

where M and N are arbitrary λ-expressions. This says that so-called *redex* terms having the form of the left expression above may be transformed into the term on the right side. This transformation may be carried out on any term or subterm having the form of a redex. An expression of the form

$$M \quad \longrightarrow \quad M'$$

will mean that M' results from M by β-contraction. We will use '$\twoheadrightarrow$' to denote *reduction*, the process of carrying out a *sequence* of contractions.

The intuition behind β-contraction is quite simple: When an abstraction (function) is applied to an argument, we create a new instance of the abstraction's template (body), with the argument substituted for all instances of the abstraction's bound variable in the template.

Thus, using the term $\Delta \equiv \lambda x.(xx)$, we have

$$(\Delta N) \quad \longrightarrow \quad (xx)[x := N] \ \equiv (NN)$$

The contraction of (ΔN) illustrates Δ's role as a a sort of "duplicator."

In general, algorithms require means to create new instances of data, to destroy data, and to move them around. The function Δ effectively creates copies of existing information. We can just as easily destroy information or move it around, using nested pairs of abstractions to manipulate pairs of arguments. For example

$$K \equiv \lambda x.(\lambda y.x)$$

can be regarded as a function that takes two arguments, throwing out the second and retaining only the first. For example, we have

$$((Ka)b) \quad \longrightarrow \quad a$$

Likewise

$$P \equiv \lambda x.(\lambda y.(yx))$$

can be regarded as taking two arguments and reversing them (technically, it applies one to the other in the opposite order), e.g.,

$$((Pa)b) \quad \longrightarrow \quad (b\,a)$$

We will generally omit parentheses and elide adjacent λs, when no confusion results, e.g.,

$$\lambda x.(\lambda y.((xx)y))$$

can be written more simply as

$$\lambda xy.xxy$$

One can use the infinite set of irreducible λ-terms (i.e., terms containing no subterms for which β-contraction can be applied) to *encode* any data type of interest: integers, pairs, lists, trees, etc. By combining such encodings with the data manipulations made possible by functions such as Δ, K, and P above, any computable function can be defined. Various extensions to the lambda calculus have been proposed in which certain constants, datatypes, and basic functions may be predefined without encoding, but it should be clear that the pure lambda calculus provides all the basic requirements for performing computation in a simple—even relatively readable—notation.

Much of the readability of λ-expressions comes from the fact that variables are used to indicate the correspondence between argument and result. However, this very readability is the source of hidden complexity. Variables take on a number of different roles in the lambda calculus. When bound in a λ-abstraction, they indicate the correspondence between argument and the argument's eventual place in the result. When unbound, however, their function is a bit more ambiguous: they may be treated as distinct "atoms," that is, uninterpretable base values, or they may take on a "logical" role as unknowns over which some expression is implicitly quantified.

It turns out that the technical difficulties we glossed over in our definition of substitution above are not trivial, and it is in the handling of variables and substitution that the real operational complexity of the lambda calculus lies. It is also why the connection between variables in the lambda calculus and their traditional use in logic is not at all straightforward. For instance, we need to have a means of creating *new variables* in the process of reduction, should clashes between instances of the same variable used in different abstractions occur. In Chapter 4, however, we will discuss a version of the lambda calculus that dispenses with the notion of variable altogether.

Even with the difficulties caused by variables, one can provide a formal account of λ-reduction in a "logical" fashion, using axioms and inference rules. However, we prefer to emphasize the computational view that β-contraction is a means for *transforming* terms to (an ultimately) simpler form, rather than a rule for proving equivalences. We will therefore not give more than passing mention to the so called η-rule, which can be used to show that two functions have the same extensional behavior, though their *computational* behavior may be different.

We will be concerned exclusively with the "pure" or untyped lambda calculus. This form of the lambda calculus, which was first studied by Church [Chu41], is pure in the *operational* sense, and thus ideal for our purposes. The considerable semantic difficulties it presents (principally caused by the possibility of performing self-application) has motivated investigation of more restrictive *typed* variants whose model-theoretic properties are less problematic, and which in many respects are more suitable bases for real programming languages. Nonetheless, since the operational aspects of typed calculi can all be embedded in an untyped world, our concentration on the untyped version of the language poses no difficulties.

2.1.3 Term Rewriting

As with the lambda calculus, a *term rewriting system* can be regarded as a means for specifying certain functions as rules or templates, in this case used to define transformations on structured expression trees, or *terms*.

Unlike the lambda calculus, however, the connections of term rewriting with logic are quite straightforward. Consider the following example of a set of axioms for group theory (taken from [HO80]):

Example 2.1

$$\begin{array}{lrcl} (1) & x+0 & = & x \\ (2) & x+(-x) & = & 0 \\ (3) & (x+y)+z & = & x+(y+z) \end{array}$$

We may naturally regard these equations as constituting an inference system for reasoning about the equational properties of groups. However, if we *orient* the equations above from left to right, we can regard each equation as a template. If a substitution instance of the template's left-hand side is found in an expression, we can transform it to an instance of the right-hand side template. The result is a *term rewriting system*. Thus, for example, while we can reason logically from the equations above that

$$(a+0)+(-a) \quad = \quad 0$$

we can also *compute* this fact, transforming the initial expression tree to its "canonical form," 0, proceeding as follows:

$$(a+0)+(-a) \xrightarrow{(1)} a+(-a) \xrightarrow{(2)} 0$$

Each reduction step above is labeled with the number of the oriented equation from Example 2.1 used.

The computational power of term rewriting is perhaps more evident if we consider a system such as the following set of equations, which define the function *Append* for appending two lists:

$$\begin{array}{lrcl} (1) & \mathit{Append}(\mathit{Null}, x) & = & x \\ (2) & \mathit{Append}(\mathit{Cons}(x,y), z) & = & \mathit{Cons}(x, \mathit{Append}(y,z)) \end{array}$$

In term rewriting systems, unlike the lambda calculus, we will not assume any fixed syntax or language: both the language and the set of templates are fixed in advance. Thus the computational properties and power of a given set of equations are entirely dependent on the definition at hand. Nonetheless, there are *classes* of rewrite systems whose computational properties are particularly easy to characterize. We will review these in chapter 4. We will see also that the presence of "fixed templates" in term rewriting systems, rather than the "user-definable" templates of the lambda calculus, need not restrict the computational capabilities of the system. *Combinatory logic* [HS86], whose development paralleled that of the lambda calculus, is a rewriting system which can be used to compute any function definable in the lambda calculus. We will define another such system in this thesis.

The fixed nature of the templates used in term rewriting systems have the advantage of eliminating a great deal of the complexity present in the lambda calculus. In particular, the problems caused by the treatment of variables under substitution do not arise in term rewriting systems. While variables are present in term rewriting systems, there is no notion of "bound" or "free," as in the lambda calculus. Our interest in term rewriting systems stems primarily from their simplicity in this respect. Therefore, while we regard the lambda calculus as our primary interest, we will use term rewriting systems of equivalent power, but more felicitous operational properties, as the basis for our algorithms.

2.2 Functional Programming

Why is the lambda calculus worth considering as a basis for the study of incremental computation? First of all, the lambda calculus can be regarded as the underpinning for almost all current sequential computer programming languages. The work of Scott and Strachey [SS71] showed that all the major features of imperative programming languages, including variable assignment and complex control flow, could be expressed entirely in terms of functions, and, fundamentally, the lambda calculus. It is rather remarkable that only relatively recently have practical *functional* programming languages been developed that incorporate most of the expressiveness and computational power of that deceptively simple formalism.

This class of programming languages has been designed with free computation on functional values as its central tenet [Hen80]. Researchers in this area might be seen as advocating a "back-to-basics" approach to programming—an effort to remove complexity and control structures that are difficult to reason about rigorously, and return to programming's mathematical roots. This effort at achieving greater purity and economy in algorithmic specification has sometimes been seen more as an interesting theoretical diversion than a serious effort to improve the quality of run-of-the-mill programs.

Hughes [Hug84a], however, argues that higher-order functions are a vital tool for program modularization, and thus of paramount *practical* importance. He notes that higher-order functions are really a sort of "glue" that allow algorithms to be factored into component functions that would otherwise be necessarily indivisible. Higher-order functions allow one to decompose an algorithm into a hierarchy of control structures of completely arbitrary form. These control structures can be re-used on a wide variety of problems having similar structure. Thus we would be within our rights to regard the control structures of a conventional imperative language as a poor-man's higher-order function (e.g., calling another function repetitively inside a loop), without flexibility or extensibility.

In addition to their lambda calculus inspired treatment of functions as values, many functional languages also provide facilities akin to a restricted form of term rewriting. These generally take the form of *pattern-matching* mechanisms by which functions can be defined equationally. However, with the exception of the work of O'Donnell [O'D77], most functional languages are best viewed as notational variants of the lambda calculus.

We can subdivide the set of functional languages into those *strict* languages whose computational strategy is closest to that used by traditional imperative programming languages (e.g., ML [HMT88]) and the more recently proposed *lazy* languages (e.g., HASKELL [HP90]), which permit a somewhat broader class of computations based on the most general operational and denotational semantics usually ascribed to the lambda calculus. Since only lazy languages take full advantage of the computational power of the lambda calculus, applications based on such languages are likely to be the primary beneficiaries of our results on incremental λ-reduction.

The spate of languages based on the lambda calculus has provided the impetus for practical implementations of various reduction strategies for the lambda calculus. If one can interpret (or compile) the pure lambda calculus efficiently, it is generally a simple task to produce a variant of the implementation for a specific functional language.

2.3 Implementation Issues in the Lambda Calculus

The goal of any system that purports to "implement the lambda calculus" is to reduce a λ-term to an equivalent term in one of several possible *normal forms.* The reduction is carried out by repeated application of the β-rule to redex subterms, producing new terms from old. Since a term may contain many subterms to which the β-rule may apply, any implementation must embody some *reduction strategy* to choose terms to be reduced.

Most work on implementations has focused on implementations of one of two reduction strategies: *leftmost* (or "normal") order, and *applicative* (or "call-by-value") order, whose semantics differ considerably. These strategies are used, respectively, in lazy and strict functional languages. The applicative strategy has the virtue of familiarity to programmers, and, more importantly, a fairly straightforward and efficient implementation. It may, however, not yield a normal form even though one may exist.

By contrast, leftmost-outermost reduction is guaranteed to yield a normal form if one exists. This has a number of desirable ramifications in the lazy languages that exploit this fact [Hug84a,Tur81]. However, until the work of Wadsworth [Wad71], leftmost-outermost reduction, though possessing pleasing semantic properties, was thought to be hopelessly inefficient.

Wadsworth noted that the apparent inefficiency of the leftmost-outermost reduction strategy stemmed from having to repeatedly evaluate identical (or similar) subexpressions that (to oversimplify matters a bit) applicative order reduction evaluated only once. He introduced a nonstandard representation of λ-terms as *graphs* rather than trees, allowing certain subterms to be effectively shared. He then developed a variant of the leftmost-outermost strategy, *graph reduction*, that exploited this sharing to avoid most redundant reductions. The result was that "lazy" languages, with their felicitous semantic properties, could now be implemented with efficiency comparable to their call-by-value cousins.

2.3.1 Graph Reduction

Consider the λ-term $M \equiv (\lambda y.(yy))(Iz)$, where $I \equiv \lambda x.x$. It may be reduced to a normal form in any one of three ways:

Example 2.2

$$\begin{aligned}
\sigma_1\colon M &\longrightarrow (Iz)(Iz) \longrightarrow z(Iz) \longrightarrow zz \\
\sigma_2\colon M &\longrightarrow (Iz)(Iz) \longrightarrow (Iz)z \longrightarrow zz \\
\sigma_3\colon M &\longrightarrow (\lambda y.(yy))z \longrightarrow zz
\end{aligned}$$

σ_1 is a *leftmost* reduction—one where the leftmost redex is contracted at each step. σ_3 is an *applicative order* reduction, where (informally) the argument part of a redex is reduced to a normal form before the redex is contracted. It is evident that σ_3 reaches the normal form (zz) in the fewest steps. It would clearly be desirable to have an *optimal* reduction strategy—one that always yields a normal form if one exists (i.e., is *normalizing*) and is also guaranteed to do so using the fewest possible redex contractions. Unfortunately, results of Barendregt, et al. [BBKV76], show that no such (recursive) strategy exists. However, we can improve matters considerably by extending the model of reduction a bit.

Note in the example above that the redex (Iz) of M is *copied* in reductions σ_1 and σ_2, since it is substituted for two instances of y. A natural alternative to copying expressions in arguments is to *share* them instead, using a graph-like data structure. The idea is illustrated below:

Example 2.3

$$\sigma_1'\colon (\lambda y.(yy))(Iz) \longrightarrow \underset{(Iz)}{(\bullet\bullet)} \longrightarrow \underset{z}{(\bullet\bullet)} \equiv zz$$

σ_1' proceeds from left to right, analogous to σ_1. In this case, however, the redex (Iz) is *shared*, rather than copied, as a result of its substitution for the two instances of variable y. The result of (Iz)'s reduction to z is shared as well. Using this method, the normal form's graph representation is reached after only two reduction steps.

Wadsworth's graph reduction algorithm [Wad71] formalizes the idea of Example 2.3. It combines a leftmost redex selection strategy with sharing of argument expressions. However, Wadsworth's algorithm is not *optimal*. If we contract non-leftmost redexes, shorter reductions (still using shared argument expressions) can be achieved, as the following example illustrates: Let $N \equiv (N_1 N_2)$, where $N_1 \equiv \lambda x.(xw)(xz)$ and $N_2 \equiv \lambda y.(Iy)$. Then the following are two graph reductions of N:

Example 2.4

$$
\begin{array}{llllll}
\rho_1 \colon N \longrightarrow & (\bullet w)(\bullet z) \longrightarrow & (Iw)(\bullet z) \longrightarrow & w(\bullet z) \longrightarrow & w(Iz) \longrightarrow & wz \\
& \quad\ \lambda y.(Iy) & \lambda y.(Iy) & \lambda y.(Iy) & &
\end{array}
$$

$$
\begin{array}{lllll}
\rho_2 \colon N \longrightarrow & (\bullet w)(\bullet z) \longrightarrow & (\bullet w)(\bullet z) \longrightarrow & w(\bullet z) \longrightarrow & wz \\
& \quad\ \lambda y.(Iy) & \quad\ \lambda y.y & \lambda y.y &
\end{array}
$$

Wadsworth's algorithm performs reduction ρ_1, while ρ_2 reaches the normal form in fewer steps by contracting the shared (Iy) redex inside N_2 before applying it to either w or z (a minimal length reduction can also be achieved without any sharing by contracting the (Iy) redex before N_1 is applied to N_2).

Reducing inner redexes, as in ρ_2, seems to bring about shorter reductions in many cases. Unfortunately, contraction of arbitrary inner redexes can sometimes lead to unnecessarily diverging reductions, as is the case with the applicative order strategy. Wadsworth's scheme reduces only leftmost redexes in order to ensure normalizability (although this is not by any means the only way to do so, see [BKKS87]).

There is evidently a subtle interplay among the issues of efficiency, normalizability, and redex sharing. The quandary is then to find a way to edge closer to the brink of optimality without plunging into the abyss of non-normalizability.

2.3.2 Graph Reduction for Terms

It is easy to define a natural notion of graph reduction for term rewriting similar to that defined for the lambda calculus. Suppose a rewriting rule has a right-hand side containing repeated instances of a variable found in the rule's left-side. Then when that rule is applied to an expression tree, we can transform that expression tree into a directed graph in which subtrees corresponding to repeated instances of the right-hand side variable are *shared*, just as we shared λ-subterms corresponding to multiple instances of a bound variable.

For instance, consider the aforementioned list of axioms for appending lists, augmented with the following equation:

$$Double(x) \;=\; Pair(x, x)$$

Then we could reduce the term

$$Double(Append(Null, a))$$

to

$$Pair(a, a)$$

using graphs as illustrated in Figure 2.1.

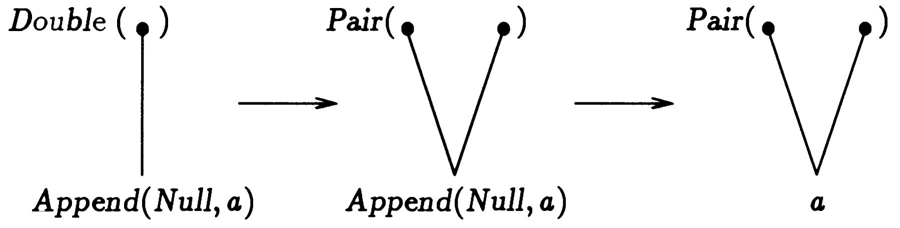


Figure 2.1: Graph Reduction of *Double*(*Append*(*Null*, *a*))

2.4 The Optimality Question

While graph reduction was a vast improvement on traditional leftmost-outermost reduction, it did not always yield a normal form with the fewest possible β-contractions. Nevertheless, Wadsworth's methods or variants thereof gained currency as reasonably efficient, though sub-optimal, implementation techniques for lazy languages.

The question of the existence of some notion of optimal reduction stood unresolved until 1978, when Lévy [Lév78,Lév80] pinned down precisely what was right and wrong with Wadsworth's method. Lévy pointed out that by contracting shared redexes in a λ-term represented by a graph, Wadsworth was effectively performing *parallel* reductions—contracting more than one redex (in the term represented by the graph) simultaneously. Lévy generalized notions of reduction on single redexes to simultaneous reduction of multiple redexes. This was done in conjunction with a careful study of the "fine structure" of reductions (aspects of which were also pursued by Klop [Klo80]).

Wadsworth's technique allowed redexes that are normally *copied* by β-reduction (so-called *residuals*) to be shared. While sharing many redexes that would normally be copied, he also copies some other redexes, losing (perhaps) opportunities for further sharing. Lévy's analysis of the reduction process identified all those redexes in a given reduction sequence that could have arisen as a result of copying. The equivalence class of redexes that could arise from copying was deemed a *family*. Lévy showed that any reduction mechanism that reduced all members of a redex family simultaneously, and furthermore never reduced a family where one of its members was not "needed" (in a technical sense) in a terminating reduction, would be optimal for any reduction that allowed sharing of redexes. In other words, any reduction technique that was normalizing and performed parallel reductions by shared redexes was optimal if every family class had exactly one shared representative.

We will examine the issue of optimality in Chapter 6, showing among other things what means *cannot* be used to implement optimal λ-reduction. However, we will not propose a means for performing Lévy's parallel reductions in the lambda calculus. Optimal λ-reduction algorithms have recently been proposed by Lamping [Lam90] and Kathail [Kat90]. However, there is as yet no consensus on whether their methods, which are extremely complicated, may constitute truly practical implementation techniques for programming languages.

Nonetheless, we will make use of the optimality properties of certain term rewriting systems that are essentially equivalent to the lambda calculus in terms of computational power, but for which the number of reductions required to produce an answer is greater in general than would be possible in an optimal reduction sequence in the lambda calculus itself. Such system-specific optimality properties will in fact be the key to implementing incremental λ-reduction.

2.5 Informal Introduction to Incremental Lambda Reduction

We are now in a position to take an informal first look at what *incremental* might mean in the context of λ-reduction.

2.5.1 Reduction of Similar Terms

Consider the following λ-terms (where $I \equiv \lambda x.x$):

$$\begin{aligned}
M_1 &\equiv (\boxed{\lambda\text{xy.xy}}\,(II)\,(II))\\
M_2 &\equiv (\boxed{\lambda\text{xy.x}}\,(II)\,(II))\\
M_3 &\equiv (\boxed{\lambda\text{xy.y}}\,(II)\,(II))
\end{aligned}$$

M_1, M_2, and M_3 are identical except for the boxed terms. The following set of (outermost) reductions suffice to reduce these terms to normal form:

$$\begin{array}{lcll}
\rho_1: & & (\boxed{\lambda\text{xy.xy}}\,(I^aI)\,(I^bI)) & \equiv M_1\\
& \twoheadrightarrow & (I^aI)\,(I^bI) & \\
& \xrightarrow{a} & I\,(I^bI) & \\
& \longrightarrow & I^bI & \\
& \xrightarrow{b} & I & \equiv M_1'\\
\rho_2: & & (\boxed{\lambda\text{xy.x}}\,(I^aI)\,(I^bI)) & \equiv M_2\\
& \twoheadrightarrow & I^aI & \\
& \xrightarrow{a} & I & \equiv M_2'\\
\rho_3: & & (\boxed{\lambda\text{xy.y}}\,(I^aI)\,(I^bI)) & \equiv M_3\\
& \twoheadrightarrow & I^bI & \\
& \xrightarrow{b} & I & \equiv M_3'
\end{array}$$

Certain redexes above have been labeled by attachment of a superscript letter to their abstraction term. Contraction of such a redex is indicated by the correspondingly labeled '$\longrightarrow$'.

The redexes contracted in ρ_1, ρ_2, and ρ_3 *overlap*, in the sense that ρ_1 and ρ_2 both contract the redex labeled a, and ρ_1 and ρ_3 both contract the redex labeled b.

If we reduce M_1 and subsequently are faced with the task of reducing M_2 and M_3, it would be useful to take advantage of the fact that certain redexes in M_1 have already been contracted, avoiding contracting the same redexes in the reductions of M_2 and M_3.

The similarity among M_1, M_2, and M_3 is made more apparent if we note that they can be represented as substitution instances of the same term:

$$\begin{aligned} M_1 &\equiv N[z := \lambda xy.xy] \\ M_2 &\equiv N[z := \lambda xy.x] \\ M_3 &\equiv N[z := \lambda xy.y] \end{aligned}$$

where $N \equiv (z\,(II)\,(II))$.

We will refer to such terms as *similar by substitution* (or, in the context of this thesis, simply "similar"), since they are the same modulo differing substitutions for some free variable in a term. We will use the notation $M_1 \overset{N}{\sim} M_2$ if M_1 and M_2 can be rewritten as $N[z := P_1]$ and $N[z := P_2]$ for some z, P_1, and P_2. The terms P_1 and P_2 are deemed the *substituends* of the similar terms, and N will be called the *common term* (of M_1 and M_2). A *set of similar terms*, e.g., S^N will refer to a set of pairwise similar terms all having the same common term (e.g., N).

Note that the common term is *not* necessarily a subterm of any of the set of similar terms—the appearance of N in $N[z := P_i]$ is merely a notational convenience. For instance, $(z\,(II)\,(II))$ is not a subterm of M_1. However, the common term of a set of similar terms always can be explicitly "factored out" using β-expansion:

$$\begin{array}{rcl} ((\lambda z.N)\,\lambda xy.xy) & \longrightarrow_\beta & M_1 \\ ((\lambda z.N)\,\lambda xy.x) & \longrightarrow_\beta & M_2 \\ ((\lambda z.N)\,\lambda xy.y) & \longrightarrow_\beta & M_3 \end{array}$$

2.5.2 Reduction in Common Subterms

Our goal is to reduce a sequence of similar terms in such a way as to avoid the contraction of common redexes, with the aim of enabling *shorter* (thus faster) reductions of later terms in the sequence than would be possible were each term reduced *ab initio*. We defer a more formal definition of what constitutes a "common contraction" or "common computation" to Chapter 7, appealing here to intuition.

Our approach to the problem is as follows: each member of a set of similar terms is reduced in such a way as to preserve in the *common* term the relevant computations performed in the reduction of the entire term. The idea is illustrated in the following reduction of M_1, where we let

$$M_1 \;\equiv\; N[z := \lambda xy.xy]$$

and

$$N \;\equiv\; (z\,(I^a I)\,(I^b I))$$

We then have:

$$\begin{array}{lll}
\rho_1'\!: & N[z := \lambda xy.xy] & \equiv M_1 \\
 & \xrightarrow{a}\; (z\,I\,(I^b I))[z := \lambda xy.xy] & \equiv (\lambda xy.xy)\,I\,(I^b I) \\
 & \xrightarrow{b}\; \underbrace{(z\,I\,I)}_{N'}[z := \lambda xy.xy] & \equiv (\lambda xy.xy)\,I\,I \\
 & \twoheadrightarrow\; I\,I & \\
 & \longrightarrow\; I & \equiv M_1'
\end{array}$$

While ρ_1' is not an outermost reduction, it nonetheless reaches a normal form, and has the pleasing property of producing a very useful intermediate subterm, $N'[z := \lambda xy.xy]$. This term results from the contraction of redexes labeled a and b, which are in a sense relevant only to the common term N. We can summarize ρ_1' as follows:

$$\rho_1'\!: N[z := \lambda xy.xy] \twoheadrightarrow N'[z := \lambda xy.xy] \twoheadrightarrow I$$

If we were able to somehow extract the reduced common term N' for future use as a side-effect of reducing M_1, we could use it to reduce M_2 and M_3 as follows: Let $N' \equiv (zII)$. We then have:

$$\begin{array}{lll}
\rho_2'\!: & N'[z := \lambda xy.x] & \equiv \lambda xy.x\,I\,I \\
 & \quad\twoheadrightarrow\; I & \equiv M_2'
\end{array}$$

$$\begin{array}{lll}
\rho_3'\!: & N'[z := \lambda xy.y] & \equiv \lambda xy.y\,I\,I \\
 & \quad\twoheadrightarrow\; I & \equiv M_3'
\end{array}$$

We also know that since $N \twoheadrightarrow N'$, and $M_i \twoheadrightarrow N'[z := P_i]$, ρ_2' and ρ_3' yield correct normal forms for M_2 and M_3, respectively, though their initial terms are not M_2 and M_3.

Unlike their counterparts ρ_2 and ρ_3, ρ_2' and ρ_3' avoid contracting redexes labeled a and b, since these computations were previously carried out in reduction ρ_1', and are embodied in N'. In the sequel, we will demonstrate that the common term and its reduction can indeed be effectively preserved in the course of reducing one of its substitution instances.

Given a set of similar terms, it should be clear that in general it is not possible simply to extract and reduce the common term to a normal form (as we did with

N'), as none may exist. Let $R \equiv (z\,(II)\,\Omega)$, where $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$. We then have the following outermost reductions:

$$\begin{aligned}
\tau_1\colon\ & R[z := \lambda xy.x] \equiv Q_1 \longrightarrow \lambda xy.x\,(II)\,\Omega \longrightarrow II \longrightarrow I \\
\tau_2\colon\ & R[z := \lambda xy.a] \equiv Q_2 \longrightarrow \lambda xy.a\,(II)\,\Omega \longrightarrow a \\
\tau_3\colon\ & R[z := \lambda xy.y] \equiv Q_3 \longrightarrow \lambda xy.y\,(II)\,\Omega \longrightarrow \Omega \longrightarrow \cdots
\end{aligned}$$

It should be evident from the example above that the redexes contracted in R depend on R's substituend. In τ_1, $\lambda xy.x$ "caused" redex (II) in R to be contracted. In τ_2, however, the (II) redex is not contracted at all. τ_3 never reaches a normal form (and thus none exists for Q_3). In short, whether or not a redex in common term R is *needed* in order to yield a normal form depends on the value of the substituend. The notion of "neededness" is formalized and discussed in some detail in [BKKS87].

The possibility of contracting redexes in the common term in advance of the reduction of the entire term is limited by the fact that knowing which of the redexes in the common term will be subsequently needed is undecidable. At best, the common term can only be reduced to some "safe" form (such as head normal form). However, this would not necessarily capture all the contractions performed in the common term during the reduction of one of its substitution instances.

Such safe pre-computation falls under the general rubric of *partial evaluation*, a technique that has been proposed for the implementation of compiler construction systems, among other things (see, e.g., [JSS89]). The goal here is rather different: to preserve computations in the common term that are *dependent* on the substitutable part of an expression *in addition to* those that are independent.

We are thus led to investigate techniques that allow the preservation of the result of contractions in the common term as a *side-effect* of reducing one of the members of a set of similar terms.

2.5.3 Incremental Lambda Reduction

In general, we assume we are given terms M_1 and M_2 such that $M_1 \overset{N}{\sim} M_2$, that is, terms having the forms $N[z := P_1]$ and $N[z := P_2]$. We would then like to perform reduction

$$\tau_1\colon\ N[z := P_1] \twoheadrightarrow M_1'$$

while also computing a term N' such that $\tau_1^N\colon\ N \twoheadrightarrow N'$, where (informally) N' embodies all of the contractions performed in N during τ_1. τ_1^N will be called a *projection* of τ_1 on (common) term N. We can then use N' in a reduction of M_2:

$$\tau_2\colon\ N'[z := P_2] \twoheadrightarrow M_2'$$

This process can easily be extended to a sequence of reductions of an arbitrarily large set of similar terms; e.g., from τ_2, we can compute an N'', the final term of a projection $\tau_2^{N'}\colon\ N' \twoheadrightarrow N''$ of τ_2 on N', etc.

Note that depending on N', it is generally *not* the case that

$$N[z := P_2] \quad \twoheadrightarrow \quad Q$$

implies

$$N'[z := P_2] \quad \twoheadrightarrow \quad Q$$

(although $N[z := P_2] =_\beta N'[z := P_2]$). We will, however, be concerned here with applications in which the desired final term of a reduction is some sort of normal form (e.g., weak head normal form), where *any* final term of the desired form is acceptable. Our goal will be to minimize as much as possible the number of β-contractions required to yield such a normal form by taking advantage of previous computations. The process described above constitutes *incremental reduction* in the lambda calculus.

We have heretofore described only terms that differ in a single subterm. However, the notion of "similarity" can be extended easily to terms that may differ in arbitrarily many subterms (e.g., let $M_1 \equiv N[z_1 := P_{11}, z_2 := P_{12}]$ and $M_2 \equiv N[z_1 := P_{21}, z_2 := P_{22}]$). We will also consider terms whose substituends may be *nested*, i.e., terms in which substituends contain other substituends. The language denotation discussed in Section 2.6 has substituends of this form induced by the term structure of its grammar. However, we will defer discussion of these generalizations, (which are of far greater practical value than the case of single substituends), to Section 8.1.3. The special case of similar terms with single substituends suffices here for the purposes of exposition.

2.6 An Example

We now give a small example of an application that illustrates how incremental λ-reduction may be used in practice. Consider the small expression language in Figure 2.2, which we will call L.

A program of L consists of a list of declarations and an expression. The expressions of L are built from natural numbers, identifiers, and sums. A declaration is a binding of an identifier to a number. Though patently useless, the language serves to illustrate some important issues. A typical expression of L is the following:

$$\textbf{let}\; \boldsymbol{a} = 2; \boldsymbol{b} = 5\,\textbf{in}\; ((3 + 4) + a)$$

(where we add appropriate keywords for the sake of readability).

Along with each grammar rule in L, we define a *denotation* for the rule's left-hand nonterminal (e.g., $[\![Prog]\!]$), using a notational variant of the lambda calculus in which integer operations, conditionals, etc., are predefined. By reducing the λ-expression denoting a program of L to weak head normal form, we obtain its

Grammar				*Denotation*
$Prog ::= Decls\ Expr$	(program)		$[\![Prog]\!]$	$\equiv ([\![Expr]\!][\![Decls]\!])$
$Expr ::= Nat$	(constant)		$[\![Expr]\!]$	$\equiv \lambda\, env\, .\, Nat$
$\mid\ Id$	(variable)			$\equiv \lambda\, env\, .(env\ Id)$
$\mid\ Expr_1\ Expr_2$	(sum)			$\equiv \lambda\, env\, .(([\![Expr_1]\!]\ env) +([\![Expr_2]\!]\ env))$
$Decls ::= Nulldecls$	(empty decl.)		$[\![Decls]\!]$	$\equiv \lambda\, i\, .(-1)$
$\mid\ Bind\ Decls$	(decl. list)			$\equiv \lambda\, i\, .\mathrm{if}(([\![Bind]\!]\ i) \neq (-1))$ $\mathbf{then}\,([\![Bind]\!]\ i)$ $\mathrm{else}\,([\![Decls]\!]\ i)$
$Bind ::= Id\ Nat$	(binding)		$[\![Bind]\!]$	$\equiv \lambda\, i\, .\mathrm{if}(i = Id)$ $\mathbf{then}\, Nat$ $\mathrm{else}\,(-1)$

Figure 2.2: The Language L

meaning. Thus the meaning of the expression above is 9, obtained by reduction of the λ-expression that denotes it. Most of the complications in the denotational definitions of L arise from the "lookup" mechanism used for determining values to which variables are bound. The abstract syntax of L may be viewed as defining a term algebra of the "parse trees" for the language in the usual way. Editing an expression of L thus amounts to performing a subtree replacement in a term of its algebra, for which there is a corresponding subtree replacement in the λ-expression denoting the term.

Consider the possible consequences of changing part of the program above (by making a subtree replacement in its corresponding term), e.g.,

(1) Changing the binding for b to 7.

(2) Changing the binding for a to 3.

(3) Adding another binding for a.

If we perform (1), there is no need to re-evaluate the sum since it contains no instance of variable b. (2) requires re-evaluating the outer sum to determine its new meaning, but not the inner expression $(3 + 4)$. (3) may invalidate the use of the previous binding for a, as well as require re-evaluation of the outer sum. Multiple changes would have further-reaching consequences.

We would like it to be the case that the new denotational λ-expression can be re-evaluated without repeating any of the work required to evaluate the denotation of the old term—in other words, we want to reduce the new λ-expression *incrementally*. Note that the λ-expression defining a denotation is not subject to completely arbitrary changes after a subtree replacement; only λ-terms denoting corresponding subterms in the language's term algebra are subject to alteration. For instance, in the λ-term denoting a sum,

$$\lambda\, env\, .(([\![Expr_1]\!]\; env) + ([\![Expr_2]\!]\; env))$$

the only λ-subterms that can be replaced are $[\![Expr_1]\!]$ and $[\![Expr_2]\!]$.

Chapter 3

Reduction Systems

> "We know nothing of form until we begin to study proportions and details."
>
> "The method proceeds from point to point...and when several facts overlap they are stated in the order of their function in the design."
>
> "A little slower than the theme."
>
> —Tovey

In order to study incremental reduction, we must first make precise what we mean by *increment* and what we mean by *reduction*. This chapter contains the technical preliminaries required to do both. The principal fruit of our labors will be the axiomatic definition of *regular replacement systems*, a new class of reduction systems defined on structured objects. A large class of term and graph rewriting systems, including the lambda calculus (using a slightly nonstandard operational semantics), form regular replacement systems. The primary technical contribution of this chapter is a generalization to regular replacement systems of the operational semantics of reduction pioneered by Lévy [Lév78,BL79,HL79,Lév80]. These results are used to define a general theory of optimality for regular replacement systems. In Chapter 7, we will extend the results further to define a theory of incremental reduction.

The material here is, of necessity, somewhat dry. However, to ameliorate the difficulty of digesting this essential but fibrous material, we first outline the progression of subjects covered:

To start, we motivate our interest in computation via *reduction*.

We then review Huet's definition of an *abstract reduction relation*. We treat abstract reduction relations as defining the atomic operations that comprise a computation. Iteration of such single-step operations in *confluent* abstract reduction relations can be used to compute functions.

Next, we discuss *Brouwerian Algebras*, which we regard as a means of imparting *structure* to objects. This notion of structure will, in the sequel, be used to define the increment in incremental computations.

An abstract reduction relation defined on objects whose structure is given by Brouwerian algebras will yield an abstract *replacement* system. An important aspect of such a reduction relation is that it extends straightforwardly to a *parallel* reduction relation (which is analogous to concurrent computation).

We next define an abstraction of the concept of *residual* found in the reduction theory of the lambda calculus and term rewriting systems. Residuals describe how various single-step reductions interact. Abstract replacement systems possessing an appropriately defined notion of residual are deemed *regular* replacement systems.

We then show that the reduction theory of Lévy extends to *any* regular replacement system, resulting in a general lattice-theoretic operational model of computation.

3.1 Relations and Computation

In the sequel, we will be concerned with notions of computation rooted in relations rather than functions. Given our purported interest in functional programming and the lambda calculus, this may seem surprising. However, we are only interested in relations that ultimately *behave* like functions; this behavior will be embodied formally in the notion of *confluence*.

The principal advantage of the relational view of computation is *choice*: there will generally be more than one way to compute the same result. One implication of this fact is that we can consider *parallel* computational strategies, safe in the knowledge that the outcome is unaffected by our choice of independent subproblems, or the fact that we choose to tackle them simultaneously. Of more immediate importance is the fact that we can choose one of a variety of computational strategies to produce a result depending on the nature of the problem at hand. We will exploit such alternative evaluation strategies in investigating incremental reduction.

Fundamental to our notion of computation will be the single-step transformation, or *reduction*, of a program into an equivalent, though "more computed" form. By iteration of single-step reductions, a program can be transformed into a final result, or *normal form* (assuming one exists). We can loosely refer to the intermediate forms in the reduction process between program and result as *states* of the computation. A *reduction relation* contains all possible single-step reductions. Since each step in the computation comes from a relation rather than a function, there may be more than one transformation applicable to any intermediate state. However, if the relation is confluent, divergent computational paths can always be made to converge, guaranteeing that normal forms are unique.

3.2 Abstract Reduction Relations

Since we have chosen to emphasize computation from a relational point of view, we review here some basic properties of the sort of relations we have in mind: Huet's *abstract reduction relations* [Hue80]. Huet's goal is to separate those properties of reduction systems that are solely relation-theoretic from those that are consequences of the term structure or semantics of a particular system. Thus Huet's results apply, e.g., to both the lambda calculus and term rewriting systems. We begin with the fundamental idea of a *reduction relation*, on which the remainder of our study is based:

Definition 3.1 (Reduction Relation)
Given an arbitrary set $\mathcal{E}$, any binary relation $\longrightarrow_R$ on $\mathcal{E}$ is a reduction relation. *The set $\mathcal{E}$ is called the* domain *of $\longrightarrow_R$, and is denoted by*

$$\mathrm{dom}(\longrightarrow_R)$$

We will generally use capital letters to refer to elements of the domain of a reduction relation. The notation

$$A \longrightarrow_R A'$$

will be shorthand for $\langle A,\ A' \rangle \in \longrightarrow_R$. We will use various subscripts to differentiate among reduction relations, e.g., $\longrightarrow_R$, $\longrightarrow_S$, ... Unless explicitly noted, however, when we consider more than one reduction relation, we will always assume that the underlying set is the same. We will also feel free to drop the subscripts identifying a particular relation when it is evident from context.

If we need to be more precise, we will refer to an *abstract reduction system*:

Definition 3.2 (Abstract Reduction System)
An abstract reduction system $\langle \mathcal{E},\ \mathcal{R} \rangle$ *is a pair consisting of a set $\mathcal{E}$ and a set $\mathcal{R}$ of reduction relations on $\mathcal{E}$.*

In the (most frequently encountered) case of an abstract reduction system with only one reduction relation, we will take $\langle \mathcal{E},\ \longrightarrow_R \rangle$ to mean $\langle \mathcal{E},\ \{\longrightarrow_R\} \rangle$.

In the systems with which we will be concerned, our intent is to treat $\longrightarrow$ as defining the notion of atomic or "single-step" computation for some system. That is, we will take

$$A \longrightarrow A'$$

to mean that there is an single-step computation which yields A' given A.

3.2.1 Derived Relations

We will use "$\cdot$" to represent relational composition, i.e.,

Definition 3.3 (Relational Composition)

$$(\longrightarrow_R \cdot \longrightarrow_S) \triangleq \{\langle X,\, Z\rangle \mid \exists Y\ (X \longrightarrow_R Y \text{ and } Y \longrightarrow_S Z)\}$$

We can now define a number of other relations derived from $\longrightarrow_R$:

Definition 3.4 (Derived Relations)
Given a reduction relation $\longrightarrow_R$ on $\mathcal{E}$,

(1) $\longrightarrow_R^0$ *denotes the identity relation on $\mathcal{E}$, i.e.,*

$$\longrightarrow_R^0 \triangleq \{\langle X,\, X\rangle \mid X \in \mathcal{E}\}$$

(2) $\longrightarrow_{R^{-1}}$ *denotes the inverse of $\longrightarrow_R$, i.e.,*

$$\longrightarrow_{R^{-1}} \triangleq \{\langle X,\, Y\rangle \mid Y \longrightarrow_R X\}$$

($\longrightarrow_{R^{-1}}$ will occasionally be notated $\longleftarrow_R$).

(3) $\longrightarrow_R^{\equiv}$ *denotes the reflexive closure of $\longrightarrow_R$, i.e.,*

$$\longrightarrow_R^{\equiv} \triangleq \longrightarrow_R \cup \longrightarrow_R^0$$

(4) $\longrightarrow_R^i$ *denotes the i-fold composition of $\longrightarrow_R$, i.e.,*

$$\longrightarrow_R^i \triangleq \longrightarrow_R \cdot \longrightarrow_R^{i-1} \quad (i > 0)$$

(5) $\longrightarrow_R^+$ *denotes the transitive closure of $\longrightarrow_R$, i.e.,*

$$\longrightarrow_R^+ \triangleq \bigcup_{i>0} \longrightarrow_R^i$$

(6) $\twoheadrightarrow_R$ *denotes the reflexive, transitive closure of $\longrightarrow_R$, i.e.,*

$$\twoheadrightarrow_R \triangleq \longrightarrow_R^0 \cup \longrightarrow_R^+$$

The last relation is of sufficient importance to merit a distinctive double arrow.

If $X \longrightarrow_R Y$, we will say there is a *one-step R-reduction* from X to Y. However, if $X \twoheadrightarrow_R Y$, we will simply say there is an *R-reduction* from X to Y. This terminology emphasizes the primacy of $\twoheadrightarrow_R$ in the ensuing discussion, even though it is technically derived from $\longrightarrow_R$.

We now define the equivalence relation induced by $\longrightarrow_R$:

Definition 3.5 (Induced Equivalence)
Given reduction relation $\longrightarrow_R$, the associated equivalence $=_R$ is the least relation satisfying the following:

$$\begin{aligned} M \twoheadrightarrow_R N &\quad \textit{implies} \quad M =_R N \\ M =_R N &\quad \textit{implies} \quad N =_R M \\ M =_R N,\, N =_R P &\quad \textit{implies} \quad M =_R P \end{aligned}$$

$=_R$ is thus the symmetric, transitive closure of $\twoheadrightarrow_R$.

We define the elements of $\mathcal{E}$ reachable in 1 step, 1 or more steps, and 0 or more steps, respectively, as follows:

Definition 3.6 (Successors)

$$
\begin{array}{lrcl}
(1) & succ_R(X) & \triangleq & \{Y \mid X \longrightarrow_R Y\} \\
(2) & succ_R^+(X) & \triangleq & \{Y \mid X \longrightarrow_R^+ Y\} \\
(3) & succ_R^*(X) & \triangleq & \{Y \mid X \twoheadrightarrow_R Y\}
\end{array}
$$

We define predicates that indicate whether two elements diverge from a single source or converge to a common sink as follows:

Definition 3.7 (Convergence, Divergence)

$$
\begin{array}{lrcl}
(1) & X \downarrow_R Y & \iff & \exists\, Z\ (X \twoheadrightarrow_R Z \text{ and } Y \twoheadrightarrow_R Z) \\
(2) & X \uparrow_R Y & \iff & \exists\, Z\ (Z \twoheadrightarrow_R X \text{ and } Z \twoheadrightarrow_R Y)
\end{array}
$$

Normal forms are sets of elements of an abstract reduction system which are closed under the reduction relation:

Definition 3.8 (Normal Form)
Let $\langle \mathcal{E}, \longrightarrow_R \rangle$ be an abstract reduction system. Then $\mathcal{N} \subseteq \mathcal{E}$ is a set of normal forms *iff for all $X \in \mathcal{N}$,*

$$succ_R^*(X) \in \mathcal{N}$$

If $\mathcal{N}$ is a set of normal forms, we say that X *has* an $\mathcal{N}$-normal form Y iff $X \twoheadrightarrow_R Y$ and $Y \in \mathcal{N}$. If X has an $\mathcal{N}$-normal form, then we will say that it is $\mathcal{N}$-*normalizable*. The *canonical* set of normal forms is the set of elements from which there is no one-step reduction:

Definition 3.9 (Canonical Normal Forms)
Let $\langle \mathcal{E}, \longrightarrow_R \rangle$ be an abstract reduction system. The canonical set of normal forms, *notated NF_R, is defined by*

$$NF_R \triangleq \{X \mid \nexists Y (X \longrightarrow_R Y)\}$$

It will be useful to be able to speak of the restriction of a relation to some subset of its underlying set, as follows:

Definition 3.10 (Restriction of Reduction Relation)
If $\longrightarrow_R$ is a reduction relation on some set $\mathcal{E}$ and $\mathcal{F}$ is a subset of $\mathcal{E}$, we will denote the restriction of $\longrightarrow_R$ to $\mathcal{F}$ by $\longrightarrow_{R(\mathcal{F})}$.

3.2.2 Properties of Reduction Relations

We now examine the properties of sets of interacting reduction relations, particularly the consequences of a reduction relation interacting with itself. The following sequence of definitions will be useful:

Definition 3.11 (Permutative Relations)
$\longrightarrow_R$ *and* $\longrightarrow_S$ permute *iff for all* X, Y, *and* Z *such that*

$$X \longrightarrow_R Y \quad \text{and} \quad Y \longrightarrow_S Z$$

there exists W *such that*

$$X \longrightarrow_S W \quad \text{and} \quad W \longrightarrow_R Z$$

The definition above is more perspicuously communicated by the following diagram:

$$\begin{tikzcd} X \arrow[r, dotted, "S"] \arrow[d, "R"'] & W \arrow[d, dotted, "R"] \\ Y \arrow[r, "S"'] & Z \end{tikzcd}$$

We will consistently adopt the convention in commutative diagrams that dotted arrows represent reductions that are existentially dependent on the universally quantified reductions represented by solid arrows. Following convention in this area, we will often leave out the elements on which the relations are defined; e.g., the diagram above could have been drawn as:

$$\begin{tikzcd} \cdot \arrow[r, dotted, "S"] \arrow[d, "R"'] & \cdot \arrow[d, dotted, "R"] \\ \cdot \arrow[r, "S"'] & \cdot \end{tikzcd}$$

Definition 3.12 (Subcommutative Relations)
$\longrightarrow_R$ *and* $\longrightarrow_S$ subcommute *iff for all* X, Y, *and* Z *such that*

$$X \longrightarrow_R Y \quad \text{and} \quad X \longrightarrow_S Z$$

there exists W *such that*

$$Y \longrightarrow^{\equiv}_{S} W \quad \text{and} \quad Z \longrightarrow^{\equiv}_{R} W$$

i.e., the following diagram commutes:

$$\begin{tikzcd} X \arrow[r, "S"] \arrow[d, "R"'] & Z \arrow[d, dotted, "R", "\equiv"'] \\ Y \arrow[r, dotted, "\equiv", "S"'] & W \end{tikzcd}$$

Definition 3.13 (Weakly Commutative Relations)
$\longrightarrow_R$ *and* $\longrightarrow_S$ commute weakly *if for all* X, Y, *and* Z *such that*

$$X \longrightarrow_R Y \quad \text{and} \quad X \longrightarrow_S Z$$

there exists W *such that*

$$Y \twoheadrightarrow_S W \quad \text{and} \quad Z \twoheadrightarrow_R W$$

i.e., the following diagram commutes:

$$\begin{tikzcd} X \arrow[r, "S"] \arrow[d, "R"'] & Z \arrow[d, dotted, twoheadrightarrow, "R"] \\ Y \arrow[r, dotted, twoheadrightarrow, "S"'] & W \end{tikzcd}$$

Definition 3.14 (Commutative Relations)
$\longrightarrow_R$ *and* $\longrightarrow_S$ commute *if for all* X, Y, *and* Z *such that*

$$X \twoheadrightarrow_R Y \quad \text{and} \quad X \twoheadrightarrow_S Z$$

there exists W *such that*

$$Y \twoheadrightarrow_S W \quad \text{and} \quad Z \twoheadrightarrow_R W$$

i.e., the following diagram commutes:

$$\begin{tikzcd} X \arrow[r, twoheadrightarrow, "S"] \arrow[d, twoheadrightarrow, "R"'] & Z \arrow[d, dotted, twoheadrightarrow, "R"] \\ Y \arrow[r, dotted, twoheadrightarrow, "S"'] & W \end{tikzcd}$$

When examining the consequences of a single reduction relation interacting with itself, we will use the following terminology:

Definition 3.15 (Subcommutativity)
$\longrightarrow_R$ *is* subcommutative *iff* $\longrightarrow_R$ *subcommutes with itself.*

Definition 3.16 (Weak Confluence)
$\longrightarrow_R$ *is* weakly confluent *iff* $\longrightarrow_R$ *commutes weakly with itself.*

Definition 3.17 (Confluence)
$\longrightarrow_R$ *is* confluent *iff* $\longrightarrow_R$ *commutes with itself; i.e., for all* X *and* Y*, if* $X \uparrow_R Y$ *then* $X \downarrow_R Y$.

The following theorem follows immediately from the definition of confluence:

Theorem 3.1 ([Hue80]) *If* $\longrightarrow_R$ *is confluent, then for all* X *and* Y*,* $X =_R Y$ *iff* $X \downarrow_R Y$.

This theorem says that in a confluent system, we can reason equationally using only reductions. The property given by Theorem 3.1 is often called the "Church-Rosser" property, although that appellation is more often given to what we have here called confluence.

In the case of subcommutative relations, it is particularly easy to show confluence. We have the following theorem from Staples:

Theorem 3.2 ([Sta80b]) *Every subcommutative relation is confluent.*

Proof Let $X \twoheadrightarrow Y$ and $X \twoheadrightarrow Z$. Then we can fill in the following commutative diagram with subcommutative squares to prove the theorem (which is more formally an induction on $\langle m, n\rangle$, where $X \longrightarrow^m Y$ and $X \longrightarrow^n Z$):

$$\begin{tikzcd}
X \arrow[r] \arrow[d] & \cdot \arrow[r] \arrow[d, dotted, "\equiv"] & \cdot \arrow[r, phantom, "\cdots"] \arrow[d, dotted, "\equiv"] & \cdot \arrow[r] & Z \\
\cdot \arrow[r, dotted, "\equiv"'] \arrow[d] & \cdot \arrow[r, dotted, "\equiv"'] \arrow[d, dotted, "\equiv"] & \cdot \arrow[r, phantom, "\cdots"] \arrow[d, dotted, no head] & {} & \\
\cdot \arrow[r, dotted, "\equiv"'] \arrow[d, phantom, "\vdots"] & \cdot \arrow[r, dotted, no head] \arrow[d, phantom, "\vdots"] & \cdot & & \\
\cdot \arrow[d] & \cdot & & & \\
Y & & & &
\end{tikzcd}$$

□

We will frequently resort to the technique of "proof by diagram-chasing."

3.2.3 Noetherian Relations

More powerful tools are usually required to show confluence for relations that are not subcommutative. One quite useful property of a reduction system is that it have no infinite reductions. Such a relation is deemed *noetherian*:

Definition 3.18 (Noetherian Relation)
$\longrightarrow_R$ *is* noetherian (or strongly normalizing) *if there is no infinite sequence of the form*

$$X_1 \longrightarrow_R X_2 \longrightarrow_R \cdots \longrightarrow_R X_n \longrightarrow_R \cdots$$

To prove properties about noetherian relations, we can use a powerful induction principle, based on the following definition of Huet:

Definition 3.19 ($\longrightarrow_R$-Completeness [Hue80])
Let P be any predicate on $\mathcal{E}$. Then P is $\longrightarrow_R$-complete *iff for all $X \in \mathcal{E}$,*

$$(\forall Y \in succ_R^+(X)\,(P(Y))) \implies P(X)$$

In other words, P is $\longrightarrow_R$ *– complete* if and only if for any X, we can infer that P holds for X from the assumption that P holds for all *R-successors* of X.

The induction principle we will use is as follows:

Definition 3.20 (Principle of Noetherian Induction [Hue80])
Let $\langle\mathcal{E}, \longrightarrow_R\rangle$ be an abstract reduction system where $\longrightarrow_R$ is a noetherian relation, and let P be a $\longrightarrow_R$-complete predicate. Then $P(X)$ is true for all $X \in \mathcal{E}$.

As an example of the use of noetherian induction, we can now prove the following theorem (a trivial generalization of a result in [Hue80]):

Theorem 3.3 *Let $\longrightarrow_R$ and $\longrightarrow_S$ be weakly commuting reduction relations. Then if either of $\longrightarrow_R$ or $\longrightarrow_S$ are noetherian, $\longrightarrow_R$ commutes with $\longrightarrow_S$.*

Proof Assume without loss of generality that $\longrightarrow_R$ is noetherian. Then we let $P(X)$ be a predicate stating the result we wish to prove, namely:

$$\begin{aligned} P(X) \;\triangleq\; & \forall Y, Z\,((X \twoheadrightarrow_R Y \text{ and } X \twoheadrightarrow_S Z) \\ & \implies \exists W\,(Y \twoheadrightarrow_S W \text{ and } Z \twoheadrightarrow_R W)) \end{aligned}$$

We then must have $X \longrightarrow_R^m Y$ and $X \longrightarrow_S^n Z$ for some values of m and n. There are two cases:

$$
\begin{tikzcd}
X \arrow[r, "S"] \arrow[d, "R"'] \arrow[dr, phantom, "\textbf{Wk. Confl.}"] & Z_1 \arrow[r, two heads, "S"] \arrow[d, dotted, two heads, "R"] \arrow[ddr, phantom, "\textbf{Induction}"] & Z \arrow[dd, dotted, two heads, "R"] \\
Y_1 \arrow[r, dotted, two heads, "S"'] \arrow[d, two heads, "R"'] \arrow[dr, phantom, "\textbf{Induction}"] & U \arrow[d, dotted, two heads, "R"] & \\
Y \arrow[r, dotted, two heads, "S"'] & V \arrow[r, dotted, two heads, "S"'] & W
\end{tikzcd}
$$

Figure 3.1: Proof of Theorem 3.3

(1) If $m = 0$, simply let $W = Z$; likewise, if $n = 0$, let $W = Y$.

(2) Otherwise, there exist Y_1 and Z_1 such that

$$X \longrightarrow_R Y_1 \twoheadrightarrow_R Y \text{ and } X \longrightarrow_S Z_1 \twoheadrightarrow_S Z$$

Since both $\twoheadrightarrow_R$ and $\twoheadrightarrow_S$ commute weakly, we know that there exists U such that

$$Z_1 \twoheadrightarrow_R U \text{ and } Y_1 \twoheadrightarrow_S U$$

By the induction hypothesis, we can assume $P(Y_1)$ and $P(Z_1)$, namely that that there exists V and W such that

$$U \twoheadrightarrow_R V \text{ and } Y \twoheadrightarrow_S V$$

and

$$Z \twoheadrightarrow_R W \text{ and } V \twoheadrightarrow_S W$$

respectively. Collecting the steps above, we use the diagram in Figure 3.1 to show that W is a common sink for Y and Z.

□

In proving some theorems in the sequel using noetherian induction, we will generally avoid supplying all the details of the induction, instead appealing to diagrams such as that above.

3.2.4 Relations among Abstract Reduction Systems

We will often want to examine the relation between various reduction systems. The familiar notion of *extension* from mathematical logic can be defined in the context of reductions using the following definition, a slight alteration of that of Hardin:

Definition 3.21 (Monomorphic Extension [Har87])
Let $\langle \mathcal{E}, \longrightarrow_R \rangle$ and $\langle \mathcal{F}, \longrightarrow_S \rangle$ be abstract reduction systems. Let h be an injective mapping from $\mathcal{E}$ to $\mathcal{F}$. Then $\langle \mathcal{F}, \longrightarrow_S \rangle$ is a monomorphic extension *of $\langle \mathcal{E}, \longrightarrow_R \rangle$ iff the following hold:*

(1) *For all $E_1, E_2 \in \mathcal{E}$, $E_1 \twoheadrightarrow_R E_2$ iff $h(E_1) \twoheadrightarrow_S h(E_2)$.*

(2) *For all $E_1 \in \mathcal{E}, F \in \mathcal{F}$, if $h(E_1) \twoheadrightarrow_S F$, then there exists an $E_2 \in \mathcal{E}$ such that $F \twoheadrightarrow_S h(E_2)$.*

If for all $E_1 \in \mathcal{E}, F \in \mathcal{F}$ such that $h(E_1) \twoheadrightarrow_S F$ there exists $E_2 \in \mathcal{E}$ such that $F = h(E_2)$, then we say that $\langle \mathcal{F}, \longrightarrow_S \rangle$ is a strong *monomorphic extension.*

Definition 3.21 says that h must not only translate $\twoheadrightarrow_R$ reductions to $\twoheadrightarrow_S$ reductions, it must also ensure that any $\twoheadrightarrow_S$ reduction can be *extended* to a reduction that can be translated back.

In some cases, a weaker notion of extension will be useful:

Definition 3.22 (Weak Monomorphic Extension)
Let $\langle \mathcal{E}, \longrightarrow_R \rangle$ and $\langle \mathcal{F}, \longrightarrow_S \rangle$ be abstract reduction systems. Let h be an injective mapping from $\mathcal{E}$ to $\mathcal{F}$. Then $\langle \mathcal{F}, \longrightarrow_S \rangle$ is a weak monomorphic extension *of $\langle \mathcal{E}, \longrightarrow_R \rangle$ iff the following hold:*

(1) *For all $E_1, E_2 \in \mathcal{E}$, $h(E_1) \twoheadrightarrow_S h(E_2)$ implies that $E_1 \twoheadrightarrow_R E_2$.*

(2) *For all $E_1, E_2 \in \mathcal{E}$ such that $E_1 \twoheadrightarrow_R E_2$, there exists $E_3 \in \mathcal{E}$ such that*

$$h(E_1) \quad \twoheadrightarrow_S \quad h(E_3) \quad \twoheadleftarrow_S \quad h(E_2)$$

(3) *For all $E_1 \in \mathcal{E}, F \in \mathcal{F}$, if $h(E_1) \twoheadrightarrow_S F$, then there exists $E_2 \in \mathcal{E}$ such that $F \twoheadrightarrow_S h(E_2)$.*

Note that a weak monomorphic extension is also a monomorphic extension.

We have the following lemma:

Lemma 3.4 *Let $\langle \mathcal{E}, \longrightarrow_R \rangle$ and $\langle \mathcal{F}, \longrightarrow_S \rangle$ be abstract reduction systems such that $\langle \mathcal{F}, \longrightarrow_S \rangle$ is a confluent weak monomorphic extension of $\langle \mathcal{E}, \longrightarrow_R \rangle$ by injection*

$$h : \mathcal{E} \to \mathcal{F}$$

Then $\longrightarrow_R$ is confluent.

Proof Let E_0, E_1, and E_2 be elements of $\mathcal{E}$ such that

$$E_1 \twoheadleftarrow_R E_0 \twoheadrightarrow_R E_2$$

Then by definition of weak monomorphic extension, we know that there exist $E'_1, E'_2 \in \mathcal{E}$ such that:

$$\begin{array}{ccccc} h(E_1) & \twoheadrightarrow_S & h(E'_1) & \twoheadleftarrow_S & h(E_0) \\ h(E_2) & \twoheadrightarrow_S & h(E'_2) & \twoheadleftarrow_S & h(E_0) \end{array}$$

Note that this also implies that

$$h(E'_1) \twoheadleftarrow_S h(E_0) \twoheadrightarrow_S h(E'_2)$$

and

$$E'_1 \twoheadleftarrow_R E_0 \twoheadrightarrow_R E'_2$$

Since $\longrightarrow_S$ is confluent, there must also exist $F \in \mathcal{F}$ such that

$$h(E'_1) \twoheadrightarrow_S F \twoheadleftarrow_S h(E'_2)$$

and by definition, there must also exist $E_3 \in \mathcal{E}$ such that

$$F \twoheadrightarrow_S h(E_3)$$

and thus we get transitively

$$h(E'_1) \twoheadrightarrow_S h(E_3) \twoheadleftarrow_S h(E'_2)$$

But then by definition, we must also have

$$E'_1 \twoheadrightarrow_R E_3 \twoheadleftarrow_R E'_2$$

and we are done. □

If we wish to reason equationally, we can define the logical notion of *conservative extension* using the following definition:

Definition 3.23 (Conservative Extension [Har87])
Let $\langle \mathcal{E}, \longrightarrow_R \rangle$ and $\langle \mathcal{F}, \longrightarrow_S \rangle$ be abstract reduction systems. Let h be an injective mapping from $\mathcal{E}$ to $\mathcal{F}$. Then $\langle \mathcal{F}, \longrightarrow_S \rangle$ is a conservative extension *of $\langle \mathcal{E}, \longrightarrow_R \rangle$ iff for all $E_1, E_2 \in \mathcal{E}$*

$$E_1 =_R E_2$$

implies

$$h(E_1) =_S h(E_2)$$

and conversely.

A confluent weakly monomorphic extension is sufficient to imply conservative extension, as the following theorem shows:

Theorem 3.5 *Let $\langle \mathcal{E}, \longrightarrow_R \rangle$ and $\langle \mathcal{F}, \longrightarrow_S \rangle$ be abstract reduction systems such that $\langle \mathcal{F}, \longrightarrow_S \rangle$ is a confluent weak monomorphic extension of $\langle \mathcal{E}, \longrightarrow_R \rangle$ by injection*

$$h: \mathcal{E} \to \mathcal{F}$$

Then $\langle \mathcal{F}, \longrightarrow_S \rangle$ is a conservative extension of $\langle \mathcal{E}, \longrightarrow_R \rangle$.

Proof Let E_1 and E_2 be elements of $\mathcal{E}$ such that

$$E_1 \;=_R\; E_2$$

We wish to show that

$$h(E_1) \;=_S\; h(E_2)$$

By Theorem 3.1, we know that there must exist $E_3 \in \mathcal{E}$ such that

$$E_1 \;\twoheadrightarrow_R\; E_3 \;\twoheadleftarrow_R\; E_2$$

By definition of weak monomorphic extension, we know there must exist $F_1, F_2 \in \mathcal{F}$ such that the following hold:

$$\begin{array}{lllll} h(E_1) & \twoheadrightarrow_S & F_1 & \twoheadleftarrow_S & h(E_3) \\ h(E_2) & \twoheadrightarrow_S & F_2 & \twoheadleftarrow_S & h(E_3) \end{array}$$

Thus we also have

$$F_1 \;\twoheadleftarrow_S\; h(E_3) \;\twoheadrightarrow_S\; F_2$$

But since $\longrightarrow_S$ is confluent, there must exist $F' \in \mathcal{F}$ such that

$$F_1 \;\twoheadrightarrow_S\; F' \;\twoheadleftarrow_S\; F_2$$

and thus by transitivity that

$$h(E_1) \;\twoheadrightarrow_S\; F' \;\twoheadleftarrow_S\; h(E_2)$$

from which we conclude that

$$h(E_1) \;=_S\; h(E_2)$$

The other direction of the proof is similar, once we observe that by Lemma 3.4, $\longrightarrow_R$ must be confluent. □

For (non-weak) monomorphic extensions, we have the following similar theorem of Hardin:

Theorem 3.6 ([Har87]) *Let $\langle \mathcal{E}, \longrightarrow_R \rangle$ and $\langle \mathcal{F}, \longrightarrow_S \rangle$ be abstract reduction systems such that $\langle \mathcal{F}, \longrightarrow_S \rangle$ is a confluent weak monomorphic extension of $\langle \mathcal{E}, \longrightarrow_R \rangle$. Then $\langle \mathcal{F}, \longrightarrow_S \rangle$ is a conservative extension of $\langle \mathcal{E}, \longrightarrow_R \rangle$.*

In the sequel, we will examine the feasibility of implementing certain classes of reductions in one abstract reduction system (say Φ) by means of a different system (say Ψ). We do this in order to take advantages of desirable *practical* properties of Ψ (e.g., existence of efficient reduction algorithms) while preserving the *formal* properties of Φ. We therefore define the notion formal notion of *implementation* as follows:

Definition 3.24 (Implementation)
Let

$$\begin{aligned} \Phi &\equiv \langle \mathcal{E}, \longrightarrow_R \rangle \\ \Psi &\equiv \langle \mathcal{F}, \longrightarrow_S \rangle \end{aligned}$$

be abstract reduction systems. Let g and h be mappings such that

$$\begin{aligned} g\colon \mathcal{F} &\rightarrow \mathcal{E} \\ h\colon \mathcal{E} &\rightarrow \mathcal{F} \end{aligned}$$

Then

$$\langle \Psi, g, h \rangle$$

is an implementation *of Φ iff the following conditions hold:*

(1) *For all $E \in \mathcal{E}$,*

$$g(h(E)) = E$$

(i.e., $\langle g, h \rangle$ is a retraction*).*

(2) *For all $E \in \mathcal{E}$, if*

$$h(E) \twoheadrightarrow_S F'$$

then

$$E \twoheadrightarrow_R g(F')$$

If $\mathcal{N}$ is a subset of $\mathcal{E}$, we say that an implementation $\langle \Psi, g, h \rangle$ of Φ is complete with respect to *$\mathcal{N}$ iff for all $E, E' \in \mathcal{E}$ such that*

$$E \twoheadrightarrow_R E'$$

and $E' \in \mathcal{N}$, *there exists* $F \in \mathcal{F}$ *such that*

$$h(E) \twoheadrightarrow_S F$$

and $g(F) \in \mathcal{N}$.

Finally, we say that $\langle \Psi,\ g,\ h \rangle$ *is simply* complete *iff for all* $E, E' \in \mathcal{E}$,

$$E \twoheadrightarrow_R E'$$

implies that there exists $E'' \in \mathcal{E}$ *such that*

$$E' \twoheadrightarrow_R E''$$

and

$$h(E) \twoheadrightarrow_S h(E')$$

From Definition 3.24, we see that an incomplete implementation need not be capable of simulating all reductions in the system being implemented, it need only correctly simulate *some* class of reductions. Note also that our definition of complete implementation allows a simulation to do "more" than the reduction it is simulating.

3.3 Brouwerian Algebras

We now edge slowly closer to the world of concrete reduction systems by restricting our focus to a useful subclass of abstract reduction systems that we will call *abstract replacement systems*. In such systems, the set of values on which reduction takes place has *structure* that can be exploited for a number of purposes. In particular, we can perform reductions on bits and pieces of an element of the set on which an abstract reduction relation is defined, safe in the knowledge that the reduction is valid for the element as a whole. We can define a coherent notion of parallel reduction. Finally, and most importantly for our interests here, we can introduce a notion of *editing*, exploiting the interaction between structural changes and a reduction relation to define incremental reduction.

The closest prior work in this area is apparently that of Ehrig, Kreowski, Maggiolo-Schettini, Rosen, and Winkowski [EKMS+81], in which notions from category theory are used to define structures and transformations thereto. Their work is concerned primarily with the properties of domain transformations that yield reduction relations, rather than the underlying structure of the domain being transformed. By contrast, we are concerned more with the structure of the domain: how it is preserved under editing operations and how it can be exploited in carrying out incremental computation. We start by introducing a class of structured objects that is particularly appropriate to our interests: *Brouwerian algebras*.

Brouwerian algebras were introduced by McKinsey and Tarski [MT46] in their study of algebraic topology (they have also been called "subtractive lattices with unit" [Cur77]). In [Rep89], Reps makes extensive use of the properties of Brouwerian algebras as a tool for investigating semantics-preserving program integration algorithms. His work argues eloquently for the value of a "theory of editing" that is sufficiently rich to facilitate manipulation of complex representations of programs such as program dependence graphs. Brouwerian algebras have also been adopted by Sundaresh and Hudak [SH90] in their study of incremental computation using partial evaluation. We will review their work in section 9.5.

We begin with the well-known concept of the *lattice*:

Definition 3.25 (Lattice)
A lattice *is a structure* $\langle \mathcal{L}, \sqcup, \sqcap \rangle$ *consisting of a set* $\mathcal{L}$ *closed under binary operations* $\sqcup$ (join) *and* $\sqcap$ (meet)*, such that the following axioms hold:*

$$\begin{array}{rclrcl} a \sqcup a & = & a & a \sqcap a & = & a \\ a \sqcup b & = & b \sqcup a & a \sqcap b & = & b \sqcap a \\ a \sqcup (b \sqcup c) & = & (a \sqcup b) \sqcup c & a \sqcap (b \sqcap c) & = & (a \sqcap b) \sqcap c \\ a \sqcup (a \sqcap b) & = & a & a \sqcap (a \sqcup b) & = & a \end{array}$$

In an *upper semilattice*, only the join operation is defined; in a *lower semilattice*, only the meet operation is defined. We will usually identify a lattice with the name of its underlying set when no confusion would result.

We define the sub-element relation $\sqsubseteq$ as follows:

Definition 3.26 (Sub-Element Relation)
$a \sqsubseteq b$ *iff there exists* c *such that* $a \sqcup c = b$.

The intended reading of '$\sqsubseteq$' in the sequel is 'is a part of.' We also define $\sqsubset$ to have its usual meaning: $a \sqsubset b$ iff $a \sqsubseteq b$ and $a \neq b$.

Definition 3.27 (Least and Greatest Elements)

(1) a *is a* least element *(or zero) of lattice* $\mathcal{L}$ *iff for all* $b \in \mathcal{L}$, $a \sqsubseteq b$.

(2) a *is a* greatest element *(or unit) of lattice* $\mathcal{L}$ *iff for all* $b \in \mathcal{L}$, $b \sqsubseteq a$.

By adding an additional operation to those of a lattice, we will have a sufficient theory to define our notion of editing:

Definition 3.28 (Brouwerian Algebra)
A Brouwerian Algebra *is a structure* $\langle \mathcal{L}, \sqcup, \sqcap, \dot{-}, \top \rangle$ *for which the following axioms hold:*

(1) $\mathcal{L}$ *is closed under* $\sqcup$, $\sqcap$, *and* $\dot{-}$.

(2) $\langle \mathcal{L}, \sqcup, \sqcap \rangle$ *is a lattice with greatest element* $\top$.

(3) *For all* a, b, *and* c *in* $\mathcal{L}$, $a \dot{-} b \sqsubseteq c$ *iff* $a \sqsubseteq b \sqcup c$.

The operator '$\dot{-}$' is called *pseudo-difference*. '$\top$' is read "top."

From McKinsey and Tarski, we know that every Brouwerian algebra has an element dual to $\top$:

Theorem 3.7 (**[MT46]**) *Let* $\langle \mathcal{L}, \sqcup, \sqcap, \dot{-}, \top \rangle$ *be a Brouwerian algebra. Then* $\mathcal{L}$ *has a least element* $\perp$ *(read "bottom") defined by*

$$\perp \triangleq \top \dot{-} \top$$

(The reader is cautioned not to regard '$\perp$' as having any connection with the semantic notion of divergence.)

We will often use the following notation for the join of all the elements of a finite set of elements from some Brouwerian algebra.

Definition 3.29 (Set Join)
Let $\mathcal{S} \subseteq \mathcal{L}$ *be a finite set of elements of a Brouwerian algebra* $\langle \mathcal{L}, \sqcup, \sqcap, \dot{-}, \top \rangle$. *Then* $\bigsqcup \mathcal{S}$ *denotes the join of all the elements of* $\mathcal{S}$ *(which is uniquely defined since '*$\sqcup$*' is commutative and associative). By convention, we define* $\bigsqcup \emptyset$ *to be* $\perp$.

As with lattices, we will generally identify a Brouwerian algebra with the name of the set of its elements.

Appendix B contains a number of elementary identities and properties of Brouwerian algebras from the literature. We will refer to such properties by number (e.g., B.1) without further commentary.

A Brouwerian algebra is a slightly weakened version of the more familiar notion of a *Boolean algebra*. Most of the Brouwerian algebras with which we will be concerned here are in fact also Boolean algebras. Some of the theorems or definitions in the sequel would be simplified through the use of Boolean algebras. However, there are examples of structures for which our theory may be useful that are *not* Boolean algebras, e.g., the program dependence graphs studied in [Rep89]. The following theorem from McKinsey and Tarski gives a concise condition which must hold for a Brouwerian algebra to be a Boolean algebra:

Theorem 3.8 (**[MT46]**) *A Brouwerian algebra* $\langle \mathcal{L}, \sqcup, \sqcap, \dot{-}, \top \rangle$ *is a Boolean algebra iff for all* $a \in \mathcal{L}$,

$$\top \dot{-} (\top \dot{-} a) = a$$

We can generate a Brouwerian algebra from one of its elements as follows:

Theorem 3.9 *Let*

$$\mathcal{F} \equiv \langle \mathcal{L}, \sqcup, \sqcap, \dot{-}, \top \rangle$$

be a Brouwerian algebra and let a *be an arbitrary element of* $\mathcal{L}$. *Define the set* $\mathcal{L}_a$ *as follows:*

$$\mathcal{L}_a \triangleq \{ b \mid b \in \mathcal{L} \quad \text{and} \quad b \sqsubseteq a \}$$

Let $\mathcal{F}_a$ *be the structure* $\langle \mathcal{L}_a, \sqcup, \sqcap, \dot{-}, a \rangle$. *Then* $\mathcal{F}_a$ *is a Brouwerian algebra.*

Proof By definition, a is a greatest element for $\mathcal{L}_a$. Since we have by hypothesis that $\mathcal{F}$ is a Brouwerian algebra, it remains only to show that $\mathcal{F}_a$ is closed under $\sqcup$, $\sqcap$, and $\dot{-}$, i.e., that for all $b, c \sqsubseteq a$,

$$b \sqcup c \;\sqsubseteq\; a \tag{3.9.1}$$
$$b \sqcap c \;\sqsubseteq\; a \tag{3.9.2}$$
$$b \dot{-} c \;\sqsubseteq\; a \tag{3.9.3}$$

(3.9.1) follows from Brouwerian algebra property B.36. From property B.35 we have $b \sqcap c \sqsubseteq b$, and since $b \sqsubseteq a$, we have shown (3.9.2). Similarly, from property B.11, we have that $b \dot{-} c \sqsubseteq b$, yielding (3.9.3). □

We will refer to $\mathcal{F}_a$ as the algebra *generated by* element a[1]. This theorem says that each element of a Brouwerian algebra may itself be treated as an algebra. In particular, we can treat mappings on elements of an algebra $\mathcal{F}$ as morphisms on algebras generated by elements of $\mathcal{F}$.

In the sequel, the following notion of *disjointness* of elements will be very important:

Definition 3.30 (Disjointness)
Let $\langle \mathcal{L}, \sqcup, \sqcap, \dot{-}, \top \rangle$ be a Brouwerian algebra. Then $a, b \in \mathcal{L}$ are disjoint, *notation $a \parallel b$, iff $a \sqcap b = \bot$.*

If $a \nparallel b$, we say that a and b *overlap*. If $\mathcal{S}$ is a set, we will use the notation $\parallel(\mathcal{S})$ to indicate that the elements of $\mathcal{S}$ are mutually disjoint. If $\mathcal{S}$ and $\mathcal{T}$ are sets, we interpret $\mathcal{S} \parallel \mathcal{T}$ to mean that for all $s \in \mathcal{S}$ and $t \in \mathcal{T}$, $s \parallel t$. In particular, this implies that for all $\mathcal{S}$, $\mathcal{S} \parallel \emptyset$.

Other authors (e.g. [O'D77]), have, by analogy with orthogonality in linear algebra, used the symbol '$\perp$' to denote concepts similar to our notion of disjointness. Thus our use of '$\parallel$' for the same ideas may initially seem a bit confusing. However, in addition to wanting to avoid confusion with zero elements of lattices, we wish here to emphasize the important connection between disjointness and *parallel reduction*; when one sees '$\parallel$', one should have a mental image of parallelism in computation, rather than some notion of colinearity.

The following lemma gives some properties of disjoint elements that do not hold for arbitrary elements of Brouwerian algebras. Its proof provides some useful examples of relatively simple manipulation properties of Brouwerian algebras.

Lemma 3.10 (Properties of Disjoint Elements) *Let $\langle \mathcal{L}, \sqcup, \sqcap, \dot{-}, \top \rangle$ be a Brouwerian algebra with $a, b, c, d \in \mathcal{L}$ and a, b, and c be mutually disjoint, i.e., $a \parallel b$, $a \parallel c$, and $b \parallel c$. Then the following hold:*

[1]Technically, $\mathcal{F}_a$ is not a *subalgebra* of $\mathcal{F}$, since their top elements differ.

$$
\begin{aligned}
a \dot{-} \boldsymbol{b} &= a & (1)\\
(a \sqcup \boldsymbol{b}) \dot{-} a &= b & (2)\\
a &\parallel (\boldsymbol{b} \sqcap c) & (3)\\
a &\parallel (b \sqcup c) & (4)\\
(a \sqcup b) \sqcap (b \sqcup c) &= b & (5)\\
(a \dot{-} d) &\parallel b & (6)
\end{aligned}
$$

Proof See Appendix A.2. □

Lemma 3.10 is summarized as properties B.42 through B.46 in appendix B.

The following definitions will also be useful:

Definition 3.31 (Separability)
Let $\langle \mathcal{L}, \sqcup, \sqcap, \dot{-}, \top \rangle$ be a Brouwerian algebra, with $a, b \in \mathcal{L}$. Then $\boldsymbol{b}$ is separable *from a iff*

$$\boldsymbol{b} \parallel (a \dot{-} b)$$

Note that from disjoint element property B.43, we know that $a \parallel \boldsymbol{b}$ implies a is separable from $a \sqcup \boldsymbol{b}$.

Definition 3.32 (Similar Elements)
Let $\langle \mathcal{L}, \sqcup, \sqcap, \dot{-}, \top \rangle$ be a Brouwerian algebra with $a, b, c \in \mathcal{L}$. Then a and $\boldsymbol{b}$ are similar modulo c *(or just* similar*), notated*

$$a \overset{c}{\sim} b$$

iff c is separable from a and b, and $a \sqcap \boldsymbol{b} = c$.

Thus a and $\boldsymbol{b}$ are similar if they overlap, and if the overlapping part can be "cleanly" removed from each.

It will be useful to be able to divide an element of a Brouwerian algebra into a set of disjoint pieces. Such a set is deemed a *partition*:

Definition 3.33 (Partition)
Let $\langle \mathcal{L}, \sqcup, \sqcap, \dot{-}, \top \rangle$ be a Brouwerian algebra. Let a be an element of $\mathcal{L}$. Then a set $\mathcal{P}_a \subseteq \mathcal{L}$ is a partition *of a iff $\parallel(\mathcal{P}_a)$ and $a = (\bigsqcup \mathcal{P}_a)$.*

The following theorem shows that the sets of mutually disjoint elements constituting a partition form a Boolean algebra:

Theorem 3.11 *Let*

$$\mathcal{F} \equiv \langle \mathcal{L}, \sqcup, \sqcap, \dot{-}, \top \rangle$$

be a Brouwerian algebra. Let $\mathcal{P}_a \subseteq \mathcal{L}$ be a finite partition of some element $a \in \mathcal{L}$. Let $\mathcal{S}$ be the set defined as follows:

$$\mathcal{S} \triangleq \{\bigsqcup T \mid T \subseteq \mathcal{P}_a\}$$

(i.e., $\mathcal{S}$ is the set consisting of arbitrary joins of each subset of the partition $\mathcal{P}_a$). Finally, let $\mathcal{F}_{\mathcal{P}_a}$ be the structure

$$\mathcal{F}_{\mathcal{P}_a} \equiv \langle \mathcal{S}, \sqcup, \sqcap, \dot{-}, a \rangle$$

Then $\mathcal{F}_{\mathcal{P}_a}$ is a Boolean algebra.

Proof We first show that $\langle \mathcal{S}, \sqcup, \sqcap, \dot{-}, \top \rangle$ is a *Brouwerian* algebra. Since every element of $\mathcal{S}$ is constructed from finite joins of elements $s \in \mathcal{P}_a$ such that (by definition of partition) $s \sqsubseteq a$, we conclude that a must be a greatest element for $\mathcal{S}$. It remains to be shown that $\mathcal{S}$ is closed under the operations of the algebra. We define for each element $s \in \mathcal{S}$ the set $\mathcal{P}_s \subseteq \mathcal{P}_a$ which is the unique set of disjoint elements of $\mathcal{P}_a$ whose join yields s. We consider first the join operator, noting that for all $s, t \in \mathcal{S}$,

$$s \sqcup t = \bigsqcup(\mathcal{P}_s \cup \mathcal{P}_t)$$

Thus $s \sqcup t \in \mathcal{S}$. (Note that '$\cup$' is *set* union, not join).

Turning now to the meet operator, we consider $s \sqcap t$. Define u, $\hat{s}$, and $\hat{t}$ as follows:

$$\begin{aligned} u &\triangleq \bigsqcup(\mathcal{P}_s \cap \mathcal{P}_t) \\ \hat{s} &\triangleq \bigsqcup(\mathcal{P}_s - (\mathcal{P}_s \cap \mathcal{P}_t)) \\ \hat{t} &\triangleq \bigsqcup(\mathcal{P}_t - (\mathcal{P}_s \cap \mathcal{P}_t)) \end{aligned}$$

u, $\hat{s}$, and $\hat{t}$ are each elements of $\mathcal{S}$, since each consists of the join of a subset of the elements comprising $\mathcal{P}_s$. We then have

$$s = \hat{s} \sqcup u \tag{3.11.1}$$

$$t = u \sqcup \hat{t} \tag{3.11.2}$$

Since by construction, $\hat{s}$ and u are joins of mutually disjoint elements, we have by Brouwerian algebra property B.45 that $\hat{s} \parallel u$. Likewise, $\hat{t} \parallel u$ and $\hat{s} \parallel \hat{t}$. Thus from property B.46 we have that

$$(\hat{s} \sqcup u) \sqcap (u \sqcup \hat{t}) = u$$

and thus that $s \sqcap t = u \in \mathcal{S}$, as desired.

Finally, we consider the pseudo-difference operator. Once again defining u, $\hat{s}$, and $\hat{t}$ as above, we have the following sequence of equations:

$$\begin{aligned} s \dot{-} t &= (\hat{s} \sqcup u) \dot{-} (u \sqcup \hat{t}) && (3.11.3) \\ &= ((\hat{s} \sqcup u) \dot{-} u) \dot{-} \hat{t} && (3.11.4) \\ &= \hat{s} \dot{-} \hat{t} && (3.11.5) \\ &= \hat{s} && (3.11.6) \end{aligned}$$

(3.11.4) follows from property B.16. We get (3.11.5) from disjoint element property B.43. Finally, from B.42, we get (3.11.6), and since $\hat{s} \in \mathcal{S}$, we are done.

We have thus far shown that $\mathcal{F}_{\mathcal{P}_a}$ is a Brouwerian algebra. To show that it is also Boolean, we know from Theorem 3.8 that we need only show for arbitrary $s \in \mathcal{S}$

$$a \dot{-} (a \dot{-} s) \;=\; s \qquad (3.11.7)$$

Define the element $\hat{a}$ by

$$\hat{a} \;\triangleq\; \bigsqcup(\mathcal{P}_a - \mathcal{P}_s) \qquad (3.11.8)$$

Then, as with the construction of $\hat{s}$ and $\hat{t}$ above, we have $\hat{a} \parallel s$. We then get the following sequence of equations:

$$\begin{aligned} a \dot{-} (a \dot{-} s) &= (\hat{a} \sqcup s) \dot{-} ((\hat{a} \sqcup s) \dot{-} s) && (3.11.9) \\ &= (\hat{a} \sqcup s) \dot{-} \hat{a} && (3.11.10) \\ &= s && (3.11.11) \end{aligned}$$

(3.11.10) and (3.11.11) are consequences of two applications of property B.43. 3.11.11 is then the result we seek. □

A simpler way of forming a Boolean Algebra is the well-known construction using the powerset of a given set: (see, e.g., [RS63, p. 68]) construction using all subsets of a given set, as follows:

Theorem 3.12 ([RS63]) *Let $\mathcal{S}$ be a set. Then*

$$\langle P(\mathcal{S}), \cup, \cap, -, \mathcal{S} \rangle$$

is a Boolean algebra.

3.3.1 Algebra Homomorphisms

It will be useful to define structure-preserving mappings between various Brouwerian algebras. For this, we use the standard definition of algebra *homomorphism*:

Definition 3.34 (Brouwerian Algebra Homomorphism)
Let

$$\mathcal{B} \equiv \langle \mathcal{L}, \sqcup_{\mathcal{B}}, \sqcap_{\mathcal{B}}, \dot{-}_{\mathcal{B}}, \top_{\mathcal{B}} \rangle$$

and

$$\mathcal{C} \equiv \langle \mathcal{K}, \sqcup_{\mathcal{C}}, \sqcap_{\mathcal{C}}, \dot{-}_{\mathcal{C}}, \top_{\mathcal{C}} \rangle$$

be Brouwerian algebras, and let h be a mapping from elements of $\mathcal{L}$ to elements of $\mathcal{K}$. Then h is a Brouwerian algebra homomorphism *(from $\mathcal{B}$ to $\mathcal{C}$) if for all $a, b \in \mathcal{L}$, the following hold:*

$$\begin{aligned} h(a \sqcup_{\mathcal{B}} b) &= h(a) \sqcup_{\mathcal{C}} h(b) \\ h(a \sqcap_{\mathcal{B}} b) &= h(a) \sqcap_{\mathcal{C}} h(b) \\ h(a \dot{-}_{\mathcal{B}} b) &= h(a) \dot{-}_{\mathcal{C}} h(b) \\ h(\top_{\mathcal{B}}) &= \top_{\mathcal{C}} \end{aligned}$$

We will refer to homomorphism h as a monomorphism *iff h is an injection.*

Brouwerian algebra homomorphisms map elements from one algebra to another while preserving their structure. One important aspect of such homomorphisms is that they preserve disjointness:

Theorem 3.13 *Let*

$$\mathcal{B} \equiv \langle \mathcal{L}, \sqcup_{\mathcal{B}}, \sqcap_{\mathcal{B}}, \dot{-}_{\mathcal{B}}, \top_{\mathcal{B}} \rangle$$

and

$$\mathcal{C} \equiv \langle \mathcal{K}, \sqcup_{\mathcal{C}}, \sqcap_{\mathcal{C}}, \dot{-}_{\mathcal{C}}, \top_{\mathcal{C}} \rangle$$

be Brouwerian algebras, and let h be a homomorphism from $\mathcal{B}$ to $\mathcal{C}$. Then for all $B_1, B_2 \in \mathcal{L}$, $B_1 \parallel B_2$ implies $h(B_1) \parallel h(B_2)$.

Proof Since $B_1 \parallel B_2$, we have

$$B_1 \sqcap_{\mathcal{B}} B_2 = \bot_{\mathcal{B}}$$

But h is a homomorphism, so we must also have

$$h(B_1) \sqcap_{\mathcal{C}} h(B_2) = h(\bot_{\mathcal{B}}) = \bot_{\mathcal{C}}$$

from which we conclude $h(B_1) \parallel h(B_2)$. $\square$

We will refer to the identity homomorphism for Brouwerian algebra $\mathcal{B}$ by $\mathbf{id}_{\mathcal{B}}$.

3.3.2 Forests and Trees

In order to give the reader a feel for structures that are naturally represented by Brouwerian algebras, we consider here two examples: *Trees* and *Context Forests*. The set of trees will form a Brouwerian algebra. Generalizations of operations on trees to allow, for instance, the removal of subtrees, will yield context forests. For the sake of simplicity, we will consider here only *unlabeled* structures. When we consider actual rewriting systems in Section 4.8, we will deal with labeled structures.

Following Brainerd, Rosen, and O'Donnell [Bra69,Ros73,O'D77], our definitions for trees and context forests will represent nodes by sequences of integers[2]. The sequences are intended to represent the path from the tree root to the node in question, with the root of a tree represented by the empty sequence, $\langle\!\langle\rangle\!\rangle$, the leftmost child of the root by $\langle\!\langle 1\rangle\!\rangle$, the latter's leftmost child by $\langle\!\langle 1,1\rangle\!\rangle$, the second child of the root by $\langle\!\langle 2\rangle\!\rangle$, etc. More formally, we make the following definitions:

Definition 3.35 (Positive Integer Sequences)
Let $\mathbb{P}$ be the set of positive integers. Then $\mathbb{P}^*$ is the set of finite sequences of positive integers, e.g., $\langle\!\langle 1,3,4,3\rangle\!\rangle$. The empty sequences is represented by $\langle\!\langle\rangle\!\rangle$. Sequence concatenation is represented by $\circ$, e.g.,

$$\langle\!\langle 1,3,4\rangle\!\rangle \circ \langle\!\langle 2,2\rangle\!\rangle = \langle\!\langle 1,3,4,2,2\rangle\!\rangle$$

We will frequently use i as a shorthand for $\langle\!\langle i\rangle\!\rangle$.

We now define context forests:

Definition 3.36 (Context Forest)
A context forest *f is a set of elements of $\mathbb{P}^*$. The set of all context forests is represented by $\mathcal{F}$.*

As is so often the case, the trees are rather more interesting than the forests. Trees are defined as follows:

Definition 3.37 (Tree)
Let t be an element of $\mathcal{F}$. Then t is a tree *iff the following hold:*

$$(1) \qquad s \circ i \in t \ \textit{ implies } \ s \in t$$

$$(2) \qquad s \circ (i+1) \in t \ \textit{ implies } \ s \circ i \in t$$

where $i \in \mathbb{P}$. The set of all trees is denoted by $\mathcal{T}_0$.

[2] What we are referring to as trees, Brainerd, Rosen, and O'Donnell call *tree domains*.

The first clause of Definition 3.37 says that trees are upward closed: every child must have a parent. The second clause says that every there can be no gaps in the progeny of any parent.

We can also define *subtrees* by similar means:

Definition 3.38 (Subtree)
Let t be an element of $\mathcal{F}$. Then t is a subtree *iff the following hold:*

(1) *There is a unique element $r \in t$ such that for all $s \in t$,*

$$s = r \circ u$$

for some $u \in \mathbb{P}^*$.

(2)

$$s_1,\ s_1 \circ s_2 \circ s_3 \in t \quad \textit{implies} \quad s_1 \circ s_2 \in t$$

(3) *For all $i \in \mathbb{P}^*$,*

$$s \circ (i+1) \in t \quad \textit{implies} \quad s \circ i \in t$$

r is the root *of t, and is denoted by* root(t)*. The set of all subtrees is denoted by $\mathcal{T}$.*

The first clause of Definition 3.38 says that subtrees have a unique root. The second says that there are no vertical gaps in the node structure. Finally, as with trees, the third clause says that a parent's progeny have no *horizontal* gaps. Note that any tree is also a subtree.

Figure 3.2 depicts a tree t_1, whose nodes are labeled by corresponding sequences from $\mathbb{P}^*$. Figure 3.3 is a subtree s of t_1. Figure 3.4 depicts a context forest f consisting of a subset of the nodes of tree t_1.

Since the set $\mathcal{F}$ is composed of subsets of $\mathbb{P}^*$, we have by Theorem 3.12 that

$$\langle \mathcal{F}, \cup, \cap, -, \mathbb{P}^* \rangle$$

is a trivial Brouwerian algebra. However, we can construct a more interesting algebra of trees as follows:

Proposition 3.14 *For any two trees $f, g \in \mathcal{T}_0$, define the operations $\sqcup$, $\sqcap$, and $\dot{-}$ as follows:*

$$\begin{aligned} f \sqcup g &\triangleq f \cup g \\ f \sqcap g &\triangleq f \cap g \\ f \dot{-} g &\triangleq \{s \mid s \circ i \in (f - g)\} \cup \{(s \circ i) \mid s \circ (i+1) \in (f - g)\} \end{aligned}$$

Then $\langle \mathcal{T}_0, \sqcup, \sqcap, \dot{-}, \mathbb{P}^* \rangle$ is a Brouwerian algebra.

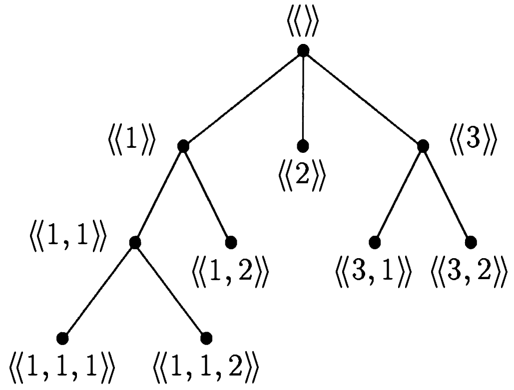


Figure 3.2: t_1: A Tree

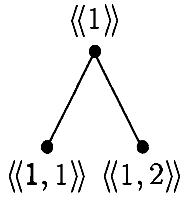


Figure 3.3: s: A Subtree of t_1

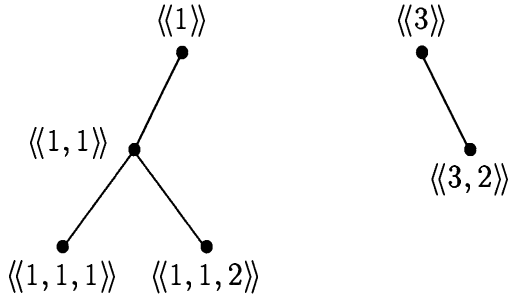


Figure 3.4: f: A Context Forest

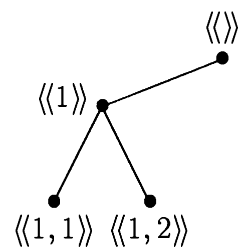


Figure 3.5: t_2: A Tree

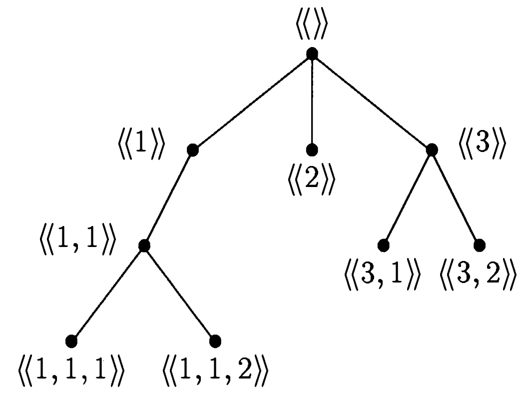


Figure 3.6: Tree Difference $t' = t_1 \dot{-} t_2$

Proposition 3.14 is easily verified after noting that the $\mathcal{T}_0$ is closed under the operations of the algebra (forcibly so in the case of the operation '$\dot{-}$').

Figure 3.5 depicts another tree t_2 (which happens to be a subtree of t_1). Figure 3.6 depicts $t' = t_1 \dot{-} t_2$, the tree resulting from the pseudo-difference of t_1 and t_2. Note that if we were to use the normal set difference operation '$-$' instead of the pseudo-difference operation '$\dot{-}$' defined in Proposition 3.14, we would get the set of nodes u depicted in Figure 3.7, which is not a tree.

Having performed a few manipulations with Brouwerian algebras, it should be clear that one can indeed regard such an algebra as defining a "theory of editing." We can thus treat the operations '$\sqcup$' and '$\dot{-}$' of a Brouwerian algebra as representing the operations *add* and *delete*, mentioned as an example of an increment set in section 1.2. We could regard '$\sqcap$' as a sort of *find* operator, since given two elements, it returns the element they have in common, if any. Not only do Brouwerian algebras form a general model of structure modification for editing, but we will also see in the next section that they are also extremely useful for describing reduction relations that represent *local* modifications to structures.

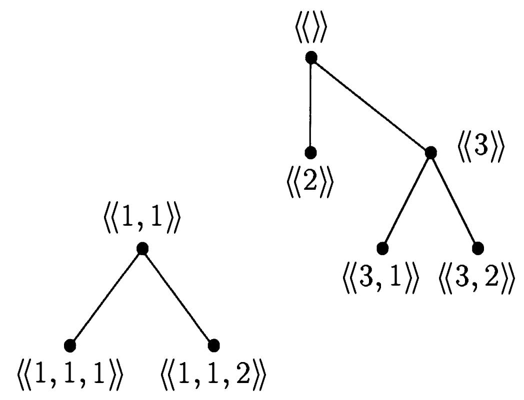


Figure 3.7: $u = t_1 - t_2$: Not a Tree

3.4 Abstract Replacement Systems

3.4.1 Compatibility

Having slogged through some rather glutinous preliminaries, we now get to the heart of the matter: combining abstract reduction relations with the structure imposed by Brouwerian algebras. The result is an abstract *replacement* system, and is based on the following idea of *compatibility*:

Definition 3.39 (Compatible Relation)
Let

$$\mathcal{F} \equiv \langle \mathcal{L}, \sqcup, \sqcap, \dot{-}, \top \rangle$$

be a Brouwerian algebra. Then a reduction relation $\longrightarrow_R$ *is* compatible *with* $\mathcal{F}$ *iff*

$$dom(\longrightarrow_R) \;\subseteq\; \mathcal{F}$$

and for all $A, B \in \mathcal{L}$ *such that* $A \parallel B$*,*

$$A \sqcup B \;\in\; dom(\longrightarrow_R)$$

and

$$A \longrightarrow_R A'$$

imply

$$(A \sqcup B) \longrightarrow_R (A' \sqcup B)$$

and

$$A' \parallel B$$

Note that a compatible relation does not necessarily form a congruence for the algebra, since we only require that the relation extend to *joins* of disjoint elements, not to arbitrary operations on arbitrary elements.

Given an arbitrary reduction relation, we can always extend it to a compatible one as follows:

Definition 3.40 (Compatible Closure)
Let $\mathcal{F}$ be the Brouwerian algebra $\langle \mathcal{L}, \sqcup, \sqcap, \dot{-}, \top \rangle$ and $\longrightarrow_R$ be a reduction relation on $\mathcal{L}$. Then the compatible closure *of $\mathcal{L}$, $\longrightarrow_{\mathcal{F}(R)}$, is the least relation satisfying the following:*

$$A, B \in \mathcal{L},\ A \longrightarrow_R A',\ \text{and}\ A \parallel B \quad \textit{implies} \quad (A \sqcup B) \longrightarrow_{\mathcal{F}(R)} (A' \sqcup B)$$

Note that if $A \longrightarrow_R A'$, then we also have $A \longrightarrow_{\mathcal{F}(R)} A'$ by letting B in Definition 3.40 equal $\bot$.

We now formally define the notion of abstract replacement system:

Definition 3.41 (Abstract Replacement System)
An abstract replacement system *$\langle \mathcal{F},\ \mathcal{R} \rangle$ is a pair consisting of a Brouwerian algebra*

$$\mathcal{F} \equiv \langle \mathcal{L}, \sqcup, \sqcap, \dot{-}, \top \rangle$$

and a set $\mathcal{R}$ of reduction relations such that for all $R \in \mathcal{R}$, R is compatible with $\mathcal{F}$.

We will as usual adopt the convention that

$$\langle \mathcal{F},\ \longrightarrow_R \rangle$$

is shorthand for

$$\langle \mathcal{F},\ \{ \longrightarrow_R \} \rangle$$

The use of the term "replacement" is due to O'Donnell [O'D77], who used it in his study of *subtree* replacement systems. The terminology is reasonably apt, since we can think of the application of a compatible relation to a sub-element in a larger context as constituting a replacement of the sub-element.

The following lemma will be handy:

Lemma 3.15 *Let $\langle \mathcal{F},\ \mathcal{R} \rangle$ be an abstract replacement system, such that $A, X \in \mathcal{F}$ and A is separable from X. Then*

$$A \longrightarrow_R A'$$

implies

$$X \sqcup A \quad \longrightarrow_R \quad (X \dot{-} A) \sqcup A'$$

Proof By Brouwerian algebra property B.12, we have

$$X \sqcup A \;=\; (X \dot{-} A) \sqcup A$$

but since A is separable from X, we have by definition that

$$(X \dot{-} A) \parallel A$$

Therefore we must have

$$(X \dot{-} A) \sqcup A \quad \longrightarrow_R \quad (X \dot{-} A) \sqcup A'$$

by compatibility, and thus

$$X \sqcup A \quad \longrightarrow_R \quad (X \dot{-} A) \sqcup A'$$

$\square$

Note that if in addition to the conditions on X and A imposed by Lemma 3.15 we also have $A \sqsubseteq X$, then we get

$$X \quad \longrightarrow_R \quad (X \dot{-} A) \sqcup A'$$

We now define a *disjoint replacement system*, whose properties are particularly simple:

Definition 3.42 (Disjoint Replacement Systems)
Let $\langle \mathcal{F},\ \{\longrightarrow_R, \longrightarrow_S\}\rangle$ be an abstract replacement system. Then reduction relations $\longrightarrow_R$ and $\longrightarrow_S$ are disjoint, *notation $\longrightarrow_R \parallel \longrightarrow_S$ if and only if for all X, Y, Z such that $Y \neq Z$, $X \longrightarrow_R Y$ and $X \longrightarrow_S Z$, there exist A, B such that $A \parallel B$, $X = A \sqcup B$, $A \longrightarrow_R A'$, $B \longrightarrow_S B'$, $Y = A' \sqcup B$, and $Z = A \sqcup B'$. The situation is given by the following diagram:*

$$\begin{tikzcd}
A \sqcup B \arrow[rr, dotted, "S"] \arrow[dd, dotted, "R"'] & \equiv & A \sqcup B' \arrow[d, phantom, "\equiv"] \\
 & X \arrow[r, "S"] \arrow[d, "R"'] & Z \\
A' \sqcup B \arrow[r, phantom, "\equiv"] & Y &
\end{tikzcd}$$

We can then extend the definition of disjointness to a single relation in the obvious way:

Definition 3.43 *An abstract replacement system $\langle \mathcal{F},\ \longrightarrow_R\rangle$ is* disjoint *iff $\longrightarrow_R$ is disjoint with itself.*

We can then show the following:

Theorem 3.16 *If* $\longrightarrow_R \parallel \longrightarrow_S$, $\longrightarrow_R$ *and* $\longrightarrow_S$ *subcommute (and thus commute).*

Proof Since $\longrightarrow_R$ and $\longrightarrow_S$ are disjoint and thus also compatible, we can always find a convergent single-step reduction as shown in the following diagram:

$$\begin{tikzcd} X \equiv A \sqcup B \arrow[r, "S"] \arrow[d, "R"'] & A \sqcup B' \equiv Z \arrow[d, dotted, "R"] \\ Y \equiv A' \sqcup B \arrow[r, dotted, "S"'] & A' \sqcup B' \equiv W \end{tikzcd}$$

□

3.4.2 Redexes and Contraction

Since we require that abstract replacement systems be compatible with the structure of a Brouwerian algebra, we can view an arbitrary pair $\langle A, A' \rangle$ of the system's reduction relation (i.e., such that $A \longrightarrow A'$) as a sort of "operator" that can be applied to any element X which contains A and from which A is separable. "Applying" the pair to X thus yields a new element X' such that $X \longrightarrow X'$. In a sense, we *replace* A in X with A', thus justifying our use of the term "replacement system".

To make this point of view more precise, we start by defining the notions of *redex* and *contractum* for elements of a reduction relation:

Definition 3.44 (Contraction, Redex, Contractum)
Let $\langle \mathcal{F}, \longrightarrow_R \rangle$ *be an abstract replacement system with*

$$\mathcal{A} = \langle A, A' \rangle \in \longrightarrow_R$$

Then we say that $\mathcal{A}$ *is a* contraction *(of* $\longrightarrow_R$*),* A *is its* redex *and* A' *is its* contractum. *We use the following notation:*

$$\begin{aligned} \mathrm{red}(\mathcal{A}) &\triangleq A \\ \mathrm{con}(\mathcal{A}) &\triangleq A' \end{aligned}$$

We say that $A \in \mathcal{F}$ is *a redex iff there exists* $\mathcal{A} \in \longrightarrow_R$ *such that* $\mathrm{red}(\mathcal{A}) = A$.

Extending the notation above to sets, given

$$\mathcal{P} \subseteq \longrightarrow_R$$

we define

$$\begin{aligned} \mathit{red}(\mathcal{P}) &\triangleq \bigcup\{\mathit{red}(\mathcal{A}) \mid \mathcal{A} \in \mathcal{P}\} \\ \mathit{con}(\mathcal{P}) &\triangleq \bigcup\{\mathit{con}(\mathcal{A}) \mid \mathcal{A} \in \mathcal{P}\} \end{aligned}$$

Finally, given $\mathcal{A}, \mathcal{B} \in \ \longrightarrow_R$ *, we will use*

$$\mathcal{A} \parallel \mathcal{B}$$

as shorthand for

$$\mathrm{red}(\mathcal{A}) \parallel \mathrm{red}(\mathcal{B})$$

Similarly, given $\mathcal{P} \subseteq \ \longrightarrow_R$ *, we define* $\parallel(\mathcal{P})$ *to hold iff* $\parallel(\mathrm{red}(\mathcal{P}))$.

(Note that if $\langle A,\ A' \rangle \parallel \langle B,\ B' \rangle$ then by compatibility we know that $A' \parallel B'$). In many systems, redexes are considered elements of the domain of some "fundamental" reduction relation (e.g., β in the lambda calculus) which is then compatibly closed over the structure of terms. Our notion of redex is by contrast somewhat simpler in that we do not require distinguishing the fundamental relation.

We now define what it means for a contraction to be *applicable* to an element:

Definition 3.45 (Contraction Applicability)
Let $\langle \mathcal{F},\ \longrightarrow_R \rangle$ *be an abstract replacement system and* X *be an element of* $\mathcal{F}$*. Then* $\mathcal{A} \in \ \longrightarrow_R$ *is* applicable *to* X *iff* $\mathrm{red}(\mathcal{A}) \sqsubseteq X$ *and* $\mathrm{red}(\mathcal{A})$ *is separable from* X*. If* $\mathcal{A}$ *is applicable to* X*, we say that* $\mathrm{red}(\mathcal{A})$ *is a redex* of X.

Thus far, we have viewed "contraction" as a noun—an element of a reduction relation. We now treat it as a verb, defining the notion of contraction *of* an element, thus formalizing the idea of replacement:

Definition 3.46 (Contraction of an Element)
Let

$$\langle\langle \mathcal{L}, \sqcup, \sqcap, \dot{-}, \top \rangle,\ \longrightarrow_R \rangle$$

be an abstract replacement system with $X, X' \in \mathcal{F}$ *and* $\mathcal{A}$ *be a contraction of* $\longrightarrow_R$ *applicable to* X*. Then* X reduces to X' by *contraction* $\mathcal{A}$ *iff*

$$X' = (X \dot{-} \mathrm{red}(\mathcal{A})) \sqcup \mathit{con}(\mathcal{A})$$

We also say that X *reduces to* X' *by contraction* of *redex* $\mathrm{red}(\mathcal{A})$*. We refer to* X' *as the* reduct *of* X.

We will use either of the following to denote the fact that X *reduces to* X'*:*

$$\begin{aligned} X &\xrightarrow{\mathcal{A}}_R X' \\ \mathcal{A}{:}X &= X' \end{aligned}$$

The first notation in Definition 3.46 is intended to emphasize the involvement of a contraction in *replacement*. The second notation highlights its interpretation as an operation that can be *applied* to an element to produce a new element. Note in general that there may be several contractions applicable to an element X, all of which may yield the same element X' (in particular, if X contains a subelement that is a redex, then X is also a redex of X). Conversely, many systems will contain redexes that have a unique contractum, i.e., if $\langle A,\ A'\rangle$ is a contraction, there may be no contraction $\mathcal{B}$ such that $red(\mathcal{B}) = A$ and $con(\mathcal{B}) \neq A'$. When this is the case, we will feel free to speak of a contraction and its redex interchangeably.

It will occasionally be convenient to use the following abbreviated notation for contraction of an element:

$$X \xrightarrow{A}_R X'$$

where there exists $\mathcal{A}$ such that $A = red(\mathcal{A})$ and

$$X \xrightarrow{\mathcal{A}}_R X'$$

since the contractum part of the contraction is implied by the reduct X'.

3.4.3 Reductions

We define *reductions*, sequences of contractions, as follows:

Definition 3.47 (Reductions)
*Let $\langle \mathcal{F},\ \longrightarrow_R \rangle$ be an abstract replacement system. An R-*reduction *ρ is a (possibly infinite) list of contractions, i.e.,*

$$\langle\!\langle \mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3, \ldots \rangle\!\rangle$$

such that for all i, $\mathcal{A}_i \in\ \longrightarrow_R$.

In general, we will denote a reduction

$$\langle\!\langle \mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n \rangle\!\rangle$$

more compactly by *juxtaposition* of its elements:

$$\mathcal{A}_1\mathcal{A}_2 \ldots \mathcal{A}_n$$

Similarly, the singleton reduction $\langle\!\langle \mathcal{A} \rangle\!\rangle$ will be denoted simply by $\mathcal{A}$. The concatenation of two reductions ρ and σ will also be denoted by juxtaposition, i.e., if ρ and σ are as follows:

$$\begin{aligned} \rho &\equiv \mathcal{A}_1\mathcal{A}_2 \ldots \mathcal{A}_m \\ \sigma &\equiv \mathcal{B}_1\mathcal{B}_2 \ldots \mathcal{B}_n \end{aligned}$$

then

$$\rho\sigma$$

denotes the sequence

$$\mathcal{A}_1\mathcal{A}_2\ldots\mathcal{A}_m\mathcal{B}_1\mathcal{B}_2\ldots\mathcal{B}_n$$

If $\mathcal{A} = \langle A,\ A'\rangle$ and ρ, υ, and γ are reductions such that

$$\rho = \upsilon\mathcal{A}\gamma$$

then we say that $\mathcal{A}$ is an *element* of ρ, notation $\mathcal{A} \in \rho$. The *length* of a reduction ρ, i.e., the number of redexes comprising it, will be denoted by $|\rho|$. The empty sequence, i.e., the contraction of *no* redexes, will be denoted by ϵ, and behaves as a left and right identity under concatenation.

We now define what it means for a reduction to be applicable to a particular element:

Definition 3.48 (Reduction of an Element)
Let $\langle \mathcal{F},\ \longrightarrow_R\rangle$ be an abstract replacement system, X_0 be an element of $\mathcal{F}$, and ρ be an R-reduction. Then ρ is a reduction *of X_0 iff $\rho = \epsilon$ or ρ is of the form*

$$\rho = \mathcal{A}_1\mathcal{A}_2\ldots\mathcal{A}_n$$

and there exist elements

$$X_1, X_2, \ldots, X_n$$

such that

$$\begin{array}{ccc} X_0 & \xrightarrow{\mathcal{A}_1}_R & X_1 \\ X_1 & \xrightarrow{\mathcal{A}_2}_R & X_2 \\ & \vdots & \\ X_{n-1} & \xrightarrow{\mathcal{A}_n}_R & X_n \end{array}$$

We say that the reduction of X_0 by ρ yields *X_n. We will refer to the set of all reductions of element X by $R(X)$.*

It is immediate that if ρ is a reduction of X yielding X', then

$$X \twoheadrightarrow_R X'$$

When we are referring to a reduction ρ of an element X yielding X', we will use whichever of the following notations is more convenient:

(1)

$$X \overset{\rho}{\twoheadrightarrow}_R X'$$

(2)

$$\rho\colon X \xrightarrow{\mathcal{A}_1}_R X_1 \xrightarrow{\mathcal{A}_2}_R \cdots \xrightarrow{\mathcal{A}_n}_R X'$$

(3)

$$\rho\colon X = X'$$

Note that $(\rho\colon X) = (\sigma\colon X)$ means that ρ and σ are two reductions of X that yield the same final element. Also note that with respect to the null reduction ϵ, we have for all X

$$\begin{array}{ccc} X & \overset{\epsilon}{\twoheadrightarrow}_R & X \\ \epsilon\colon X & = & X \end{array}$$

In the sequel, we will be concerned exclusively with reductions that are *realizable*:

Definition 3.49 (Realizability)
Let $\langle \mathcal{F}, \longrightarrow_R \rangle$ be an abstract replacement system. Then a reduction ρ is realizable *iff there exists $X \in \mathcal{F}$ such that ρ is a reduction of X. The set of all realizable **R**-reductions is denoted by R^*.*

Whenever we perform operations on reductions, e.g., concatenation, we will ensure that the resulting reduction is realizable.

We remark that if ρ is a reduction such that $\rho = \mathcal{A}\sigma$ for some reduction $\sigma = \mathcal{B}\sigma'$, then it is easy to show that ρ is realizable iff $red(\mathcal{B})$ is separable from $con(\mathcal{A}) \sqcup red(\mathcal{B})$ and σ is realizable.

Two reductions that can be realized by a single element are *coinitial*:

Definition 3.50 (Coinitiality)
Let $\langle \mathcal{F}, \longrightarrow_R \rangle$ be an abstract replacement system. Then $\rho, \sigma \in R^*$ are coinitial *iff there exists $X \in \mathcal{F}$ such that ρ and σ are both reductions of X.*

The notion dual to coinitiality is *cofinality*:

Definition 3.51 (Cofinality)
Let $\langle \mathcal{F}, \longrightarrow_R \rangle$ be an abstract replacement system. Then $\rho, \sigma \in R^*$ are cofinal *iff ρ and σ are coinitial, and for all $X \in \mathcal{F}$ such that ρ and σ are reductions of X,*

$$\rho\colon X \;=\; \sigma\colon X$$

Cofinal reductions thus yield the same "result" when applied to the same initial element.

3.4.4 Parallel Reduction

We are now in a position to extend our notion of reduction to allow *parallel* reduction of disjoint sets of redexes. We begin by defining the notion of parallel redex:

Definition 3.52 (Parallel Contraction, Redex, Contractum)
Let $\langle \mathcal{F}, \longrightarrow_R \rangle$ *be an abstract replacement system. Then* $\mathcal{P} \subseteq \longrightarrow_R$ *is a* parallel contraction *iff* $\mathcal{P}$ *is finite and* $\|(\mathcal{P})$. *The set of all parallel contractions is denoted by* $\|(\longrightarrow_R)$. *If* $\mathcal{P}$ *is a parallel contraction,* red$(\mathcal{P})$ *is its parallel redex,* con$(\mathcal{P})$ *is its* parallel contractum, *and* $|\mathcal{P}|$ *denotes the cardinality of* $\mathcal{P}$. *Parallel contraction* $\mathcal{P}$ *is* applicable *to* $X \in \mathcal{F}$ *iff for all* $\mathcal{A} \in \mathcal{P}$, $\mathcal{A}$ *is applicable to* X. *If* $\mathcal{P}$ *is applicable to* X, *we say that* red$(\mathcal{P})$ *is a redex* of X.

Extending the notion of contraction to parallel redexes, we have the following:

Definition 3.53 (Parallel Contraction of an Element)
Let

$$\langle\langle \mathcal{L}, \sqcup, \sqcap, \dot{-}, \top\rangle, \longrightarrow_R \rangle$$

be an abstract replacement system with $X, X' \in \mathcal{L}$ *and* $\mathcal{P} \in \|(\longrightarrow_R)$ *be a parallel contraction applicable to* X. *Then* X reduces to X' by *parallel contraction* $\mathcal{P}$ *iff*

$$X' = (X \dot{-} \bigsqcup red(\mathcal{P})) \sqcup \bigsqcup con(\mathcal{P})$$

We also say that X *reduces to* X' *by contraction* of *redex* red$(\mathcal{P})$. *We refer to* X' *as the* reduct *of* X.

As before, the reduction will be notated in one of the following ways:

$$\begin{array}{ccc} X & \overset{\mathcal{P}}{\longrightarrow}{}^{\|}_{R} & X' \\ \mathcal{P}{:}X & = & X' \end{array}$$

Note that by Definition 3.53, we have for all X

$$X \overset{\emptyset}{\longrightarrow}{}^{\|}_{R} X$$

Thus $\emptyset$ functions as a sort of "universal" parallel contraction, and we will thus refer to it as the *trivial* contraction.

Given Definition 3.53, we define a reduction relation in the obvious way:

Definition 3.54 (Parallel Reduction Relation)
Let $\langle \mathcal{F}, \longrightarrow_R \rangle$ *be an abstract replacement system. The* parallel reduction relation *on* $\mathcal{F}$, $\longrightarrow^{\|}_{R}$, *is defined by*

$$\longrightarrow^{\|}_{R} \;\triangleq\; \{\langle X, X'\rangle \mid X \overset{\mathcal{P}}{\longrightarrow}{}^{\|}_{R} X' \text{ for some } \mathcal{P} \subseteq \|(\longrightarrow_R)\}$$

As usual, we notate the reflexive, transitive closure of $\longrightarrow^{||}_{R}$ by

$$\twoheadrightarrow^{||}_{R}$$

Given Definition 3.54, we get the following lemma:

Lemma 3.17 *Let $\langle \mathcal{F},\ \longrightarrow_R \rangle$ be an abstract replacement system. Then*

$$\longrightarrow_R \subseteq \longrightarrow^{||}_{R}$$

Proof We simply observe that if

$$X \longrightarrow_R X'$$

then $\{X\}$ is a parallel redex of X, and we have

$$X \overset{\{X\}}{\longrightarrow}{}^{||}_{R} X'$$

□

We now show that our notion of parallel reduction yields an abstract replacement system:

Theorem 3.18 *Let $\mathcal{F}$ be the Brouwerian algebra*

$$\mathcal{F} \equiv \langle \mathcal{L}, \sqcup, \sqcap, \dot{-}, \top \rangle$$

and $\langle \mathcal{F},\ \longrightarrow_R \rangle$ be an abstract replacement system. Then $\langle \mathcal{F},\ \longrightarrow^{||}_{R} \rangle$ is an abstract replacement system.

Proof We need only show that $\longrightarrow^{||}_{R}$ is compatible with the algebra, i.e., we must show that for all $A, B \in \mathcal{L}$ such that $A \parallel B$,

$$A \longrightarrow^{||}_{R} A'$$

implies

$$(A \sqcup B) \longrightarrow_R (A' \sqcup B)$$

and

$$A' \parallel B$$

First, we know that if $A \longrightarrow^{||}_{R} A'$, then by Definition 3.53 there exists $\mathcal{P} \subseteq \longrightarrow_R$ such that

$$A \overset{\mathcal{P}}{\longrightarrow}{}^{||}_{R} A'$$

where

$$A' = (A \dot{-} \bigsqcup red(\mathcal{P})) \sqcup \bigsqcup con(\mathcal{P})$$

But $red(\mathcal{P})$ is also a redex of $(A \sqcup B)$, and we have

$$(A \sqcup B) \xrightarrow{\mathcal{P}}{}^{\|}_{R} ((A \sqcup B) \dot{-} \bigsqcup red(\mathcal{P})) \sqcup \bigsqcup con(\mathcal{P})$$

By Brouwerian algebra properties B.14 and B.42, we have

$$\begin{aligned} & ((A \sqcup B) \dot{-} \bigsqcup red(\mathcal{P})) \sqcup \bigsqcup con(\mathcal{P}) \\ = \; & ((A \dot{-} \bigsqcup red(\mathcal{P})) \sqcup \bigsqcup con(\mathcal{P})) \sqcup B && (3.18.1) \\ = \; & (A' \sqcup B) && (3.18.2) \end{aligned}$$

Equation (3.18.2) is the result we seek.

It then remains to show that $A' \parallel B$. By algebra property B.47, we know that

$$(A \dot{-} \bigsqcup red(\mathcal{P})) \quad \| \quad B \tag{3.18.3}$$

and from property B.45 and the fact that $\longrightarrow_R$ is compatible with $\mathcal{F}$, we also have

$$\bigsqcup con(\mathcal{P}) \quad \| \quad B \tag{3.18.4}$$

Combining (3.18.3) and (3.18.4) using another application of property B.45 gives us

$$B \parallel (A \dot{-} \bigsqcup red(\mathcal{P})) \sqcup \bigsqcup con(\mathcal{P})$$

and thus $B \parallel A'$ as desired. □

The definitions of Section 3.4.3 pertaining to reductions can be trivially extended to parallel redexes by replacing the pairs that comprise reduction sequences by the *sets* of pairs comprising parallel redex. The parallel analogue to Definition 3.47 is as follows:

Definition 3.55 (Parallel Reduction)
Let $\langle \mathcal{F}, \longrightarrow_R \rangle$ be an abstract replacement system. A parallel *R-reduction ρ is a list of sets*

$$\langle\!\langle \mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_3 \rangle\!\rangle$$

such that for all i, $\mathcal{P}_i \in \|(\longrightarrow_R)$.

We extend Definition 3.48 in the obvious way to parallel redexes as follows:

Definition 3.56 (Parallel Reduction of an Element)
Let $\langle \mathcal{F}, \longrightarrow_R \rangle$ be an abstract replacement system with $X_0 \in \mathcal{F}$ and $\mathcal{T}_0 \subseteq \mathcal{F}$. Then ρ is a parallel R-reduction of X_0 *iff $\rho = \epsilon$ or ρ is of the form*

$$\rho = \mathcal{P}_1 \mathcal{P}_2 \ldots \mathcal{P}_n$$

and there exist elements

$$X_1, X_2, \ldots, X_n$$

such that

$$\begin{array}{rcl} X_0 & \stackrel{\mathcal{P}_1}{\longrightarrow}_R & X_1 \\ X_1 & \stackrel{\mathcal{P}_2}{\longrightarrow}_R & X_2 \\ & \vdots & \\ X_{n-1} & \stackrel{\mathcal{P}_n}{\longrightarrow}_R & X_n \end{array}$$

We define realizability and coinitiality for parallel reductions (of sets or elements) analogously to Definitions 3.49 and 3.50. We will use $\|(R)^*$ to denote the set of all realizable parallel R-reductions. Every non-parallel reduction $\rho \in R^*$ will be treated as a reduction of $\|(R)^*$ by the trivial injection that takes elements of $\longrightarrow_R$ to singleton sets. Conversely, we will say that an element ρ of $\|(R)^*$ is *serial* iff ρ is also an element of R^* (i.e., can be trivially injected into $\|(R)^*$).

It will be convenient to extend the notion of parallel reduction to apply to *sets* of elements. The requisite definitions are as follows:

Definition 3.57 (Set Applicability)
Let $\langle \mathcal{F}, \longrightarrow_R \rangle$ be an abstract replacement system and $\mathcal{T}$ be a subset of $\mathcal{F}$. Then parallel contraction $\mathcal{P} \in \|(\longrightarrow_R)$ is applicable *to $\mathcal{S}$ iff for all $\mathcal{A} \in \mathcal{P}$, there exists some $X \in \mathcal{T}$ such that $\mathcal{A}$ is applicable to X. If $\mathcal{P}$ is applicable to X, we say that* red$(\mathcal{P})$ *is a redex* of $\mathcal{S}$.

We can now extend the notion of parallel contraction to apply to a *set* of elements as follows:

Definition 3.58 (Parallel Contraction of a Set)
Let $\langle \mathcal{F}, \longrightarrow_R \rangle$ be an abstract replacement system, with $\mathcal{T}, \mathcal{T}' \subseteq \mathcal{F}$, and let $\mathcal{P}$ be a parallel contraction of $\longrightarrow_R$ applicable to $\mathcal{T}$. Then $\mathcal{T}$ reduces to $\mathcal{T}'$ by *contraction $\mathcal{P}$ iff for all $X \in \mathcal{T}$, there exists $\mathcal{P}_X \subseteq \mathcal{P}$ such that*

$$X \stackrel{\mathcal{P}_X}{\longrightarrow}{}^{\|}_{R} X'$$

and

$$\mathcal{T}' \;=\; \bigcup\{X' \mid X \stackrel{\mathcal{P}_X}{\longrightarrow}{}^{\|}_{R} X'\}$$

We also say that $\mathcal{T}$ reduces to $\mathcal{T}'$ by contraction of $\mathcal{P}$. *We refer to $\mathcal{T}'$ as the* reduct *of $\mathcal{T}$.*

We will use either of the following to denote the fact that $\mathcal{T}$ reduces to $\mathcal{T}'$:

$$\begin{array}{rcl} \mathcal{T} & \stackrel{\mathcal{P}}{\longrightarrow}{}^{\|}_{R} & \mathcal{T}' \\ \mathcal{P}{:}\mathcal{T} & = & \mathcal{T}' \end{array}$$

Given Definitions 3.57 and 3.58, it is trivial to extend Definitions 3.49 3.50, 3.54, and 3.55 to reduction on redex sets.

It is important to note that there is a crucial difference between reduction of an element and reduction of a set: Each parallel reduction of a set can be simulated by a set of reductions on the elements of the initial set (we could view this as a parallel set of reduction "streams," one for each element of the initial set). By contrast, it is not generally possible to subdivide an element into a set of sub-elements such that the reduction of the element is also a reduction of the set.

We now define a class of reductions whose members are applicable both to sets and to elements:

Definition 3.59 (Partition-Respecting Reduction)
Let $\langle \mathcal{F}, \longrightarrow_{\mathbf{R}} \rangle$ be an abstract replacement system. Let X be an element of $\mathcal{F}$, $\mathcal{P}_A$ be a partition of X, and $\rho \in ||(R)^*$ be a reduction of X. Then ρ respects *$\mathcal{P}_X$ iff ρ is also a reduction of $\mathcal{P}_X$.*

The idea of the definition above is that a partition-respecting reduction operates entirely on designated disjoint pieces of a given element.

3.4.5 Properties of Parallel Reductions

It should be clear from Definition 3.53 that any parallel contraction of an element corresponds to a set of non-parallel reductions of the same element. To make this more explicit, we require the following definitions:

Definition 3.60 (Development of Disjoint Set)
Let $\langle \mathcal{F}, \longrightarrow_R \rangle$ be an abstract replacement system with $\mathcal{P} \subseteq \longrightarrow_R$ and $||(\mathcal{P})$. Then reduction $\rho \in R^*$ is a development *of $\mathcal{P}$ iff either of the following holds:*

- $\rho = \epsilon$
- *$\rho = \tau Q$ for some reduction τ and τ is a development of $\mathcal{P} - \{Q\}$.*

A development of $\mathcal{P}$ is simply a reduction whose (non-parallel) contractions are a subset of $\mathcal{P}$. ρ is a complete *development of $\mathcal{P}$ iff either of the following holds:*

- *$\rho = \epsilon$ and $\mathcal{P} = \emptyset$*
- *$\rho = \tau Q$ for some reduction τ and τ is a complete development of $\mathcal{P} - \{Q\}$.*

Note that if ρ is a complete development of $\mathcal{P}$, then

$$|\rho| \;=\; |\mathcal{P}|$$

We then have the following theorem:

Theorem 3.19 *Let $\langle \mathcal{F}, \longrightarrow_R \rangle$ be an abstract replacement system with $X \in \mathcal{F}$ and $\mathcal{P} \in \longrightarrow_R$ such that*

$$X \xrightarrow{\mathcal{P}}{}^{\|}_{R} X'$$

Then every complete development ρ of $\mathcal{P}$ is a reduction of X such that

$$X \xrightarrow{\rho}\!\!\!\rightarrow_R X'$$

Proof Trivial, given Definition 3.46 and Definition 3.53, since the operation '$\sqcup$' is commutative and the elements of $\mathcal{P}$ are disjoint. $\square$

Theorem 3.19 states that parallel contraction of a set of disjoint elements can be "simulated" by any sequential reduction of the elements comprising the set, and that the single-step parallel reduction relation is a sub-relation of the non-parallel reduction relation.

Given Theorem 3.19, we can "flatten" parallel reductions to yield non-parallel reductions:

Definition 3.61 (Serialization)
Let $\langle \mathcal{F}, \longrightarrow_R \rangle$ be an abstract replacement system, and let

$$\rho \equiv \mathcal{P}_1 \mathcal{P}_2 \ldots \mathcal{P}_n$$

be a reduction of $\|(R)^*$. Then $\rho' \in R^*$ is a serialization *of ρ iff there exist reductions $\sigma_1, \sigma_2, \ldots, \sigma_n$ such that for all i, $\sigma_i \in R^*$ is a complete development of $\mathcal{P}_i$ and*

$$\rho = \sigma_1 \sigma_2 \ldots \sigma_n$$

It is immediate from Theorem 3.19 that if ρ' is a serialization of ρ and ρ is a reduction of X, then

$$\rho' : X = \rho : X$$

The following is a consequence of Lemma 3.17 and Theorem 3.19, and lets us relate the various reduction relations we have considered thus far:

Corollary 3.20 *Given abstract replacement system $\langle \mathcal{F}, \longrightarrow_R \rangle$, the following holds:*

$$\longrightarrow_R \subseteq \longrightarrow^{\|}_R \subseteq \twoheadrightarrow_R = \twoheadrightarrow^{\|}_R$$

Thus we see that parallel reductions are ultimately no more "powerful" than non-parallel reductions, since the underlying relations are equivalent. However, their *operational* characteristics differ critically, in that a single parallel reduction can simulate several non-parallel reductions in a single step. This fact will be crucial to defining coherent notions of optimality and incrementality in the sequel.

Given an element X of an abstract replacement system, there will generally be more than one contraction that is applicable to X. A *reduction strategy* is a function that maps an element to a non-trivial contraction applicable to that element:

Definition 3.62 (Reduction Strategy)
Let $\langle \mathcal{F},\ \longrightarrow_R \rangle$ be an abstract replacement system Then a partial function

$$s\colon\ \mathcal{F}\ \rightarrow\ \|(\longrightarrow_R)$$

*is an R-*reduction strategy *iff for every $X \in \mathcal{F}$ for which $s(X)$ is defined, $s(X)$ is a contraction of X such that $s(X) \neq \emptyset$.*

*If s is an R-reduction strategy and X is an element of $\mathcal{F}$, we use $s^i(X)$ to denote the i-*fold iteration *of s, a reduction in $\|(R)^*$ defined as follows:*

$$s^i(X)\ \triangleq\ \begin{cases} \epsilon & i = 0 \\ s^{i-1}(X)\ s(\, s^{i-1}(X)\colon X\,) & \text{if } i > 0 \text{ and } s(s^{i-1}(X)\colon X) \text{ is defined} \\ s^{i-1}(X) & \text{otherwise} \end{cases}$$

A reduction strategy that always yields an element of a particular class of reductions is defined as follows:

Definition 3.63 ($\mathcal{C}$-Reduction Strategy)
*Let $\langle \mathcal{F},\ \longrightarrow_R \rangle$ be an abstract replacement system and s be an R-reduction strategy. Let $\mathcal{C} \subseteq \|(R)^*$ be a class of reductions. Then s is a $\mathcal{C}$-*strategy *iff for all $X \in \mathcal{F}$ such that some reduction $\rho \in \mathcal{C}$ is a reduction of X,*

$$s^i(X) \in \mathcal{C}$$

for some $i > 0$. When the value of i is irrelevant (e.g., when it is unique), we will simply use the notation $s^*(X) \in \mathcal{C}$.

A strategy that always yields normal forms is a *normalizing strategy*:

Definition 3.64 (Normalizing Strategy)
*Let $\langle \mathcal{F},\ \longrightarrow_R \rangle$ be an abstract replacement system, $\mathcal{N} \subseteq \mathcal{F}$ be a set of normal forms, and s be an R-reduction strategy. Then s is $\mathcal{N}$-*normalizing *iff for all $X \in \mathcal{F}$ possessing an $\mathcal{N}$-normal form,*

$$s^i(X) \in \mathcal{N}$$

for some i.

In this section, we have laid out a general notion of reduction arising from *replacement* of elements of a Brouwerian algebra. A crucial prerequisite to our theory is that the reduction relation be *compatible* with the structure of the algebra. We were then able to extend various notions of reduction to *parallel* contraction of disjoint sets of sub-elements.

3.5 Residual Maps

Theorem 3.16 shows that disjoint abstract replacement systems are trivially confluent. Unfortunately, most useful abstract replacement systems are not disjoint, and we must resort to more elaborate means to show that they are confluent. Classical results from the lambda calculus [Chu41] identified the notion of *residual* as a means for proving confluence. This idea was subsequently identified in other settings, e.g., term rewriting systems [Ros73], recursive program schemes [Vui74], and combinatory reduction systems [Klo80], in which it has proved equally useful.

Let $\langle \mathcal{F}, \longrightarrow_R \rangle$ be an abstract replacement system. Then given two non-disjoint contractions $\mathcal{P}, \mathcal{Q} \in \|(\longrightarrow_R)$, the residual of $\mathcal{Q}$ by $\mathcal{P}$ (which we notate $\mathcal{Q}/\mathcal{P}$) is defined to be "what's left" of $\mathcal{Q}$ after carrying out contraction $\mathcal{P}$. We want $\mathcal{Q}/\mathcal{P}$ to be a contraction such that by performing the contraction $\mathcal{P}$, followed by $\mathcal{Q}/\mathcal{P}$, we get the same result as performing the contraction $\mathcal{Q}$, then $\mathcal{P}/\mathcal{Q}$. If such a residual operation can be appropriately defined for $\longrightarrow_R$, $\longrightarrow_R$ can be easily shown confluent. Residuals arise quite naturally in systems where redex contraction results in some "rearrangement" of the other redexes. Not only does the concept of a residual allow for relatively simple proofs of confluence, it allows us to define a sensible notion of reduction *equivalence* which will allow various reduction strategies to be compared. We will examine these properties of reduction further in Section 3.6.

In this section, we will generalize the notion of residual to the class of abstract replacement systems using an axiomatic approach. Abstract replacement systems for which a slightly restricted notion of residual can be defined will form *regular replacement systems*. Most of the reduction systems with which we will be concerned in the sequel will be regular replacement systems. The work here is closely related to that of Stark [Sta87], who investigated a general class of systems called *concurrent transition systems*. Our notion of regular replacement system, however, possesses a finer structure that we will exploit in Section 3.6 to generalize the results of Lévy [Lév78,BL79,HL79,Lév80].

We begin by axiomatizing the notion of *residual map*, a function that yields the residual of one contraction after performing another contraction.

Definition 3.65 (Residual Map)

Let $\Psi \equiv \langle \mathcal{F}, \longrightarrow_R \rangle$ be an abstract replacement system. Let '/' be an infix operator representing a function from pairs of elements of $\|(\longrightarrow_R)$ to elements of $\|(\longrightarrow_R)$, i.e.,

$$/ : \; (\, \|(\longrightarrow_R) \times \|(\longrightarrow_R)\,) \;\rightarrow\; \|(\longrightarrow_R)$$

Then '/' is a residual map *for Ψ iff for all coinitial $\mathcal{P}, \mathcal{Q}, \mathcal{T} \in \|(\longrightarrow_R)$, the following axioms hold:*

(1) $\mathcal{T} \subseteq \mathcal{P}$ *implies*

$$(\mathcal{Q}/\mathcal{P} = \emptyset \text{ and } \mathcal{P}/\mathcal{Q} = \mathcal{T}) \quad \text{iff} \quad \mathcal{Q} = \mathcal{P} - \mathcal{T}$$

(2) $\mathcal{P} \parallel \mathcal{Q}$ *implies*

$$\mathcal{P}/\mathcal{T} \parallel \mathcal{Q}/\mathcal{T} \quad \text{and} \quad \mathcal{P} \,/\, (\mathcal{T}/\mathcal{Q}) = \mathcal{P}/\mathcal{T}$$

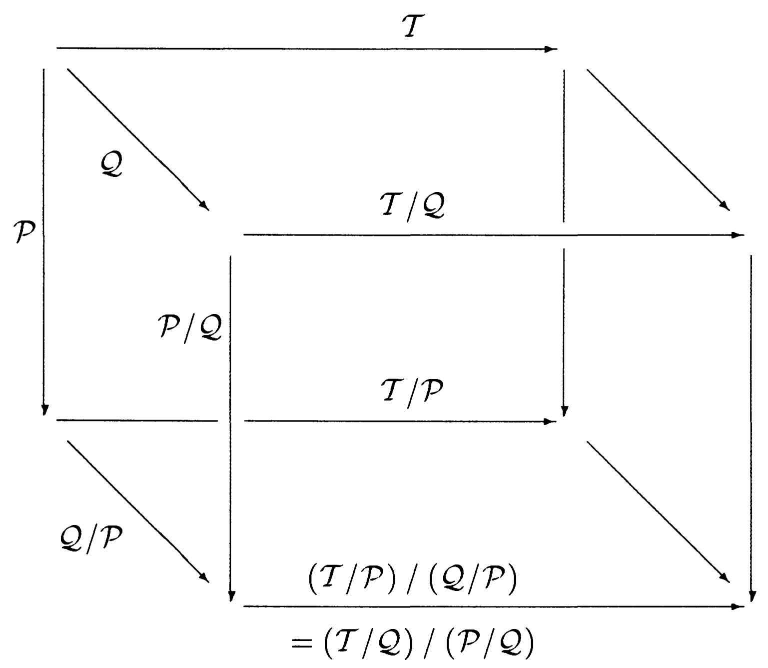


Figure 3.8: Cube Diagram for Residual Maps

(3) $\mathcal{P}\,(\mathcal{Q}/\mathcal{P})$ *and* $\mathcal{Q}\,(\mathcal{P}/\mathcal{Q})$ *are cofinal*

(4) $(\mathcal{T}/\mathcal{P})\,/\,(\mathcal{Q}/\mathcal{P}) = (\mathcal{T}/\mathcal{Q})\,/\,(\mathcal{P}/\mathcal{Q})$

We say that $\mathcal{P}/\mathcal{Q}$ *is the* residual of $\mathcal{P}$ by $\mathcal{Q}$. *If we need to distinguish among residual maps for various abstract replacement systems, we will use a subscript, e.g.,* $/_{\Psi}$.

Clause (1) of Definition 3.65 determines how two contractions, one of which is a subset of the other, behave with respect to residuals. Clause (2) determines how disjoint contractions interact. Finally, clauses (3) and (4) are the heart our requirements for a residual map. Together, they are depicted by the "cube" diagram of Figure 3.8. We thus collectively refer to clauses (3) and (4) as the *cube property* for residuals, after the eponymous lemma of Curry and Feys [CF58] for the lambda calculus.

We first mention some important elementary consequences of Definition 3.65:

Lemma 3.21 *Let* $\langle \mathcal{F},\ \longrightarrow_R \rangle$ *be an abstract replacement system with residual map '/'. Let* $\mathcal{P}$ *and* $\mathcal{Q}$ *be arbitrary coinitial elements of* $\|(\longrightarrow_R)$. *Then the following properties hold:*

(1) $\emptyset/\mathcal{P} = \emptyset$

(2) $\mathcal{P}/\emptyset = \mathcal{P}$

(3) $\mathcal{P}/\mathcal{Q} = \emptyset = \mathcal{Q}/\mathcal{P}$ *iff* $\mathcal{P} = \mathcal{Q}$

(4) *If* $\mathcal{Q} \subseteq \mathcal{P}$*, then* $\mathcal{Q}/\mathcal{P} = \emptyset$ *and* $\mathcal{P}/\mathcal{Q} = \mathcal{P} - \mathcal{Q}$

(5) $\mathcal{P} \parallel \mathcal{Q}$ *implies* $\mathcal{P}/\mathcal{Q} = \mathcal{P}$

Proof (1), (3), and (4) follow from clause (1) of Definition 3.65. (2) follows from clause (2) of Definition 3.65. Finally, we get (5) by noting from clause (2) of Definition 3.65 that for all $\mathcal{T} \in \longrightarrow_R^{\parallel}$ coinitial with $\mathcal{P}$ and $\mathcal{Q}$, $\mathcal{P}/\mathcal{Q} = \mathcal{P} \,/\, (\mathcal{T}/\mathcal{Q})$. Setting $\mathcal{T}$ equal to $\mathcal{Q}$, we have

$$\begin{aligned} \mathcal{P}/\mathcal{Q} &= \mathcal{P} \,/\, (\mathcal{Q}/\mathcal{Q}) && (3.21.1) \\ &= \mathcal{P} \,/\, \emptyset && (3.21.2) \\ &= \mathcal{P} && (3.21.3) \end{aligned}$$

(3.21.1) and (3.21.2) follow from previously proved properties (3) and (2), respectively. □

Properties (1), (2), and (3) of Lemma 3.21 replace clauses (1) and (2) of Definition 3.65 in Stark's axiomatization of *Current Transition Systems* in [Sta87].

We refer to residual maps that are particularly simple as *strong residual maps*:

Definition 3.66 (Strong Residual Map)
Let $\Psi \equiv \langle \mathcal{F}, \longrightarrow_R \rangle$ *be an abstract replacement system and let '/' be a residual map for* Ψ*. Then '/' is a* strong residual map *iff for all* $\mathcal{A}, \mathcal{B} \in \longrightarrow_R$*, either* $\mathcal{A}/\mathcal{B} = \emptyset$ *or there exists* $\mathcal{C} \in \longrightarrow_R$ *such that*

$$\mathcal{A}/\mathcal{B} = \mathcal{C}$$

Given two elements of the (non-parallel) reduction relation $\longrightarrow_R$, a strong residual map yields an element of $\longrightarrow_R$, rather than an element of $\|(\longrightarrow_R)$.

Any disjoint replacement system possesses a strong residual map, as the following theorem shows:

Theorem 3.22 *Let* $\Psi \equiv \langle \mathcal{F}, \longrightarrow_R \rangle$ *be a disjoint abstract replacement system. Then there exists a residual map '/' for* Ψ *such that '/' is strong.*

Proof Given $\mathcal{P}, \mathcal{Q} \in \longrightarrow_R$, simply define '/' by

$$\mathcal{P}/\mathcal{Q} = \begin{cases} \emptyset & \mathcal{Q} = \mathcal{P} \\ \mathcal{P} & \text{otherwise} \end{cases}$$

Since Ψ is disjoint, we know that for all $\mathcal{P}, \mathcal{Q} \in \|(\longrightarrow_R)$, $\mathcal{P} \parallel \mathcal{Q}$. Thus all the clauses of Definition 3.65 are trivially satisfied. We then have for all $\mathcal{A}, \mathcal{B} \in \longrightarrow_R$, either

$$\mathcal{A}/\mathcal{B} = \emptyset$$

or

$$\mathcal{A}/\mathcal{B} = \mathcal{A}$$

and we are done. □

It will turn out that many graph reduction systems are disjoint, and thus have strong residual maps (and that this property is in fact what makes graph reduction such a useful concept).

3.6 Reduction Theory

The remainder of this chapter is a generalization of the work of Lévy [Lév78, BL79,HL79,Lév80] to abstract replacement systems possessing a residual map. We will examine the way that various reductions in such systems interact, with the intention of exploiting these properties in defining optimal and incremental reduction strategies. In section 3.7, we will introduce the notion of *stable* residual map, the basis for defining *regular* abstract replacement systems. Since our primary goal is to *use* Lévy's reduction-theoretic properties, many of the results below will appeal to analogous results in the literature. A complete reiteration of their details would not only be redundant, but require prohibitive amounts of space.

3.6.1 Reduction Residuals

Following Lévy, we begin by extending the notion of residual to arbitrary pairs of coinitial reductions:

Definition 3.67 (Reduction Residual [Lév78])
Let $\langle \mathcal{F}, \longrightarrow_R \rangle$ be an abstract replacement system with residual map '/'. Let σ and ρ be coinitial reductions of $\|(R)^*$. Then the reduction residual *of σ by ρ, notation σ/ρ is the least relation consistent with the axioms of residual maps for singleton reductions (i.e., satisfying $\langle\!\langle \mathcal{P} \rangle\!\rangle / \langle\!\langle \mathcal{Q} \rangle\!\rangle = \langle\!\langle \mathcal{P}/\mathcal{Q} \rangle\!\rangle$ for $\langle\!\langle \mathcal{P} \rangle\!\rangle, \langle\!\langle \mathcal{Q} \rangle\!\rangle \in \|(R)^*$ and $\mathcal{P}, \mathcal{Q} \in \|(\longrightarrow_R))$ that also satisfies the following:*

$$
\begin{array}{lrcl}
(1) & \epsilon/\rho & = & \epsilon \\
(2) & \mathcal{P}/\epsilon & = & \mathcal{P} \\
(3) & \mathcal{P}/(\mathcal{Q}\rho) & = & (\mathcal{P}/\mathcal{Q}) \,/\, \rho \\
(4) & (\sigma\mathcal{P})/\rho & = & (\sigma/\rho)(\mathcal{P} \,/\, (\rho/\sigma))
\end{array}
$$

If ρ and σ are coinitial elements of $\|(R)^*$, we use

$$\rho \vee \sigma$$

to denote

$$\rho(\sigma/\rho)$$

The rather knotty definition above is made clearer if we consider coinitial reductions ρ and σ defined as follows:

$$\rho \;=\; \mathcal{P}\gamma$$

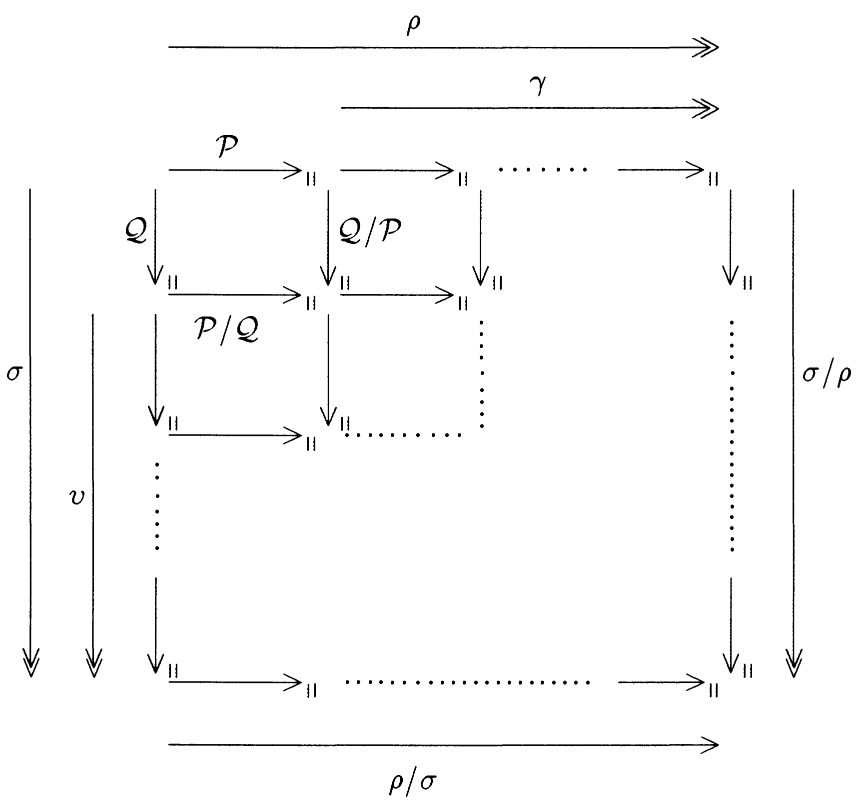


Figure 3.9: Reduction Residuals

$$\sigma \;=\; \mathcal{Q}\upsilon$$

ρ/σ and σ/ρ are then illustrated in Figure 3.9.

We can now generalize clauses (3) and (4) of Definition 3.65 to arbitrary reductions:

Theorem 3.23 (General Lemma of Parallel Moves) *Let $\langle \mathcal{F}, \longrightarrow_R \rangle$ be an abstract replacement system with residual map '/'. Let $\rho, \sigma, \tau \in \|(R)^*$ be coinitial reductions of $X \in \mathcal{F}$. Then the following hold:*

(1)

$$(\rho \vee \sigma){:}X \;=\; (\sigma \vee \rho){:}X$$

(2)

$$\tau/(\rho \vee \sigma) \;=\; \tau/(\sigma \vee \rho)$$

Proof (1) results simply by iterating clause (3) of Definition 3.65 through the square generated by Definition 3.67.

Similarly, we show (2) by iterating clause (4) of Definition 3.65 through the large cube formed by the conjunction of ρ, σ, and τ. The base case is

$$\mathcal{T}/(\mathcal{P} \vee \mathcal{Q}) \; = \; \mathcal{T}/(\mathcal{Q} \vee \mathcal{P}) \tag{3.23.1}$$

We have the following from clause (3) of Definition 3.67:

$$\begin{aligned} \mathcal{T}/(\mathcal{P} \vee \mathcal{Q}) &= (\mathcal{T}/\mathcal{P})/(\mathcal{Q}/\mathcal{P}) \\ \mathcal{T}/(\mathcal{Q} \vee \mathcal{P}) &= (\mathcal{T}/\mathcal{Q})/(\mathcal{P}/\mathcal{Q}) \end{aligned}$$

(3.23.1) then follows from clause (4) of Definition 3.65. □

The proof of Theorem 3.23 thus justifies the commutative notation $\mathcal{P} \vee \mathcal{Q}$, since in a very strong sense the resulting reduction $\mathcal{P} \vee \mathcal{Q}$ is the same as $\mathcal{Q} \vee \mathcal{P}$.

3.6.2 Reduction Equivalence

The preliminaries above enable us to define a notion of reduction *equivalence* and reduction *prefix*, one of Lévy's most important contributions. His idea was to note that in systems with a residual map, if an element contains two redexes $\mathcal{P}$ and $\mathcal{Q}$, the order of their contraction is irrelevant modulo the residual map (i.e., using the notation defined above, the result of $\mathcal{P} \vee \mathcal{Q}$ is the same as $\mathcal{Q} \vee \mathcal{P}$). This fact allows the definition of an equivalence relation on reductions that is oblivious to permutation induced by the operation '$\vee$' on redexes, as follows:

Definition 3.68 (Permutation Equivalence [BL79])
Let $\langle \mathcal{F}, \longrightarrow_R \rangle$ be an abstract replacement system with residual map '/'. Then $\simeq$ is the least equivalence relation satisfying the following properties:

$$\begin{array}{rcll} \epsilon & \simeq & \emptyset & \\ \mathcal{P} \vee \mathcal{Q} & \simeq & \mathcal{Q} \vee \mathcal{P} & \text{if } \mathcal{P}, \mathcal{Q} \in \|(\longrightarrow_R) \text{ are coinitial} \\ \rho\sigma\tau & \simeq & \rho\sigma'\tau & \text{if } \sigma \simeq \sigma' \end{array}$$

If $\mathcal{P} \simeq \mathcal{Q}$, we will say that $\mathcal{P}$ and $\mathcal{Q}$ are equivalent modulo permutation.

The first equation states that we will identify trivial reductions and reductions consisting of the trivial contraction (since the outcome is unaffected by either). The last equation ensures that '$\simeq$' is a congruence for reduction concatenation. We then have an immediate corresponding notion of reduction *prefix*:

Definition 3.69 (Reduction Prefix [Lév78])
Let $\langle \mathcal{F}, \longrightarrow_R \rangle$ be an abstract replacement system with residual map '/', and ρ and σ be reductions of $\|(R)^*$. Then ρ is a prefix *of σ, notated*

$$\rho \lesssim \sigma$$

iff there exists τ *such that*

$$\rho\tau \simeq \sigma$$

The following lemma is immediate from Definition 3.68:

Lemma 3.24 *Let* $\langle \mathcal{F}, \longrightarrow_R \rangle$ *be an and* $\rho, \sigma \in ||(R)^*$ *be reductions of* $X \in \mathcal{F}$ *such that*

$$\rho \quad \simeq \quad \sigma$$

Then ρ *and* σ *are coinitial, and for all* $X \in \mathcal{F}$ *such that* ρ *and* σ *are reductions of* X*,*

$$\rho : X \quad = \quad \sigma : X$$

Proof Easy induction on the definition of $\simeq$, using clause (3) of Definition 3.65. □

We will need the following definition:

Definition 3.70 *For all integers* n*,* $\emptyset^n$ *is defined as:*

$$\emptyset^n \quad \triangleq \quad \underbrace{\emptyset\emptyset\ldots\emptyset}_{n \ times}$$

Note that by definition $\emptyset^n \simeq \epsilon$.

The following lemma is proved in [Lév80], relying only on Theorem 3.23 and the definition of reduction residual:

Lemma 3.25 ([Lév80, Lemma 3.3, p. 167]) *Let* $\langle \mathcal{F}, \longrightarrow_R \rangle$ *be an abstract replacement system with residual map '/', and* $\rho, \sigma \in ||(R)^*$ *be coinitial reductions. Then the following hold:*

(1)

$$\rho \simeq \sigma$$

iff

$$\begin{array}{rcl} \rho/\sigma & = & \emptyset^m \\ \sigma/\rho & = & \emptyset^n \end{array}$$

where $m = |\rho|$ *and* $n = |\sigma|$.

(2)

$$\rho \lesssim \sigma$$

iff

$$\rho/\sigma \quad = \quad \emptyset^m$$

where $m = |\rho|$.

Lemma 3.25 can be viewed as an extension of property 3 of Lemma 3.21. Here, "what's left" of two equivalent reductions is a trivial reduction, i.e., one consisting only of trivial contractions. Thus to determine whether two reductions are equivalent by permutation, it suffices to compute their residuals with respect to one another.

In [BL79], Lévy and Berry show that coinitial reductions of a given element form an upper semilattice, as follows:

Theorem 3.26 ([BL79, Theorem 2.3.2, p. 161]) *Let $\langle \mathcal{F}, \longrightarrow_R \rangle$ be an abstract replacement system with residual map '/', and X be an element of $\mathcal{F}$. Let $R(X)/\simeq$ be the set of equivalence classes of reductions of X in $\|(R)^*$ whose members are ordered by*

$$\rho \lesssim \sigma \quad \textit{implies} \quad [\rho] \lesssim [\sigma]$$

(where the meaning of $\lesssim$ is here overloaded). Then for all coinitial reductions $\rho, \sigma \in \|(R)^*$ of X, there is an element

$$[\rho] \vee [\sigma] \;\in\; R(X)/\simeq$$

such that

$$[\rho] \vee [\sigma] \;=\; [\rho \vee \sigma] \;=\; [\sigma \vee \rho]$$

and $[\rho] \vee [\sigma]$ is the least upper bound of $[\rho]$ and $[\sigma]$ in $R(X)/\simeq$.

Theorem 3.26 thus justifies our use of '$\vee$' in connection with reductions.

As Lévy points out, $\lesssim$ does not necessarily induce a lattice structure on reductions, since the possibility of erasure of redexes implies that there may also exist unbounded increasing chains of reductions whose results are subsequently erased. Thus there may in general be no *greatest lower bound* for a pair of coinitial redexes.

The following properties of $\simeq$, $\lesssim$, and $\vee$ are given in [Lév80] as corollaries of the previous definitions:

Proposition 3.27 (Properties of Reduction Prefix [Lév80, pp. 168-169])

Let $\langle \mathcal{F}, \longrightarrow_R \rangle$ be an abstract replacement system with residual map '/'. Let ρ, σ, and τ and be arbitrary coinitial reductions of $\|(R)^*$. Then the following properties hold:

$$\begin{array}{rcl} \rho \vee \sigma & \simeq & \sigma \vee \rho \qquad (1) \\ \rho \simeq \sigma & \text{iff} & \forall \tau \; (\tau/\rho \equiv \tau/\sigma) \qquad (2) \\ \rho\sigma \simeq \rho\tau & \text{iff} & \sigma \simeq \tau \qquad (3) \\ \rho \simeq \sigma & \text{implies} & \rho/\tau \simeq \sigma/\tau \qquad (4) \\ \rho & \lesssim & \rho \qquad (5) \end{array}$$

$$\begin{array}{rcll} \rho \lesssim \sigma \lesssim \tau & \text{implies} & \rho \lesssim \tau & (6) \\ \rho \lesssim \sigma \lesssim \rho & \text{iff} & \rho \simeq \sigma & (7) \\ \rho \lesssim \sigma & \text{implies} & \rho/\tau \lesssim \sigma/\tau & (8) \\ \rho\sigma \lesssim \rho\tau & \text{iff} & \sigma \lesssim \tau & (9) \\ \rho & \lesssim & \rho \vee \sigma & (10) \\ \rho & \lesssim & \sigma \vee \rho & (11) \\ \rho \lesssim \tau \text{ and } \sigma \lesssim \tau & \text{implies} & (\rho \vee \sigma) \lesssim \tau & (12) \end{array}$$

3.6.3 Properties of Developments

Our definition of residual map is intended to accommodate the idea of residual-as-contraction-in-parallel that occurs in practical settings, e.g., the lambda calculus and term rewriting systems which (as is usually the case) do not possess strong residual maps. However, we want to ensure that, with respect to residuals, a parallel contraction behaves identically to the composite behavior of its constituent components. We demonstrate that our axiomatization of residuals is indeed well-behaved in this sense by a sequence of lemmas and theorems.

The following two lemmas will be useful in the sequel:

Lemma 3.28 *Let $\langle \mathcal{F}, \longrightarrow_R \rangle$ be an abstract replacement system with residual map '$/$', $\mathcal{P}$ be an element of $\|(\longrightarrow_R)$, and let $\rho\sigma$ be an arbitrary reduction of $\|(R)^*$ coinitial with $\mathcal{P}$. Then*

$$\mathcal{P} \,/\, (\rho\sigma) \;=\; (\mathcal{P}/\rho) \,/\, \sigma$$

Proof We proceed by induction on $|\rho|$. If $|\rho| = 0$, then $\rho = \epsilon$, and from clauses (3) and (2) of Definition 3.67,

$$\mathcal{P} \,/\, (\rho\sigma) \;=\; (\mathcal{P}/\epsilon) \,/\, \sigma \;=\; \mathcal{P}/\sigma$$

Otherwise $|\rho| > 0$, and $\rho = \mathcal{Q}\rho_1$ for some $\mathcal{Q} \in \longrightarrow_R^{\|}$ and $\rho_1 \in \|(R)^*$. We then have the following equations:

$$\begin{array}{rcll} \mathcal{P} \,/\, (\rho\sigma) & = & \mathcal{P} \,/\, (\mathcal{Q}\rho_1\sigma) & (3.28.1) \\ & = & (\mathcal{P}/\mathcal{Q}) \,/\, (\rho_1\sigma) & (3.28.2) \\ & = & ((\mathcal{P}/\mathcal{Q}) \,/\, \rho_1) \,/\, \sigma & (3.28.3) \\ & = & (\mathcal{P}/(\mathcal{Q}\rho_1)) \,/\, \sigma & (3.28.4) \\ & = & (\mathcal{P}/\rho) \,/\, \sigma & (3.28.5) \end{array}$$

(3.28.2) follows from clause (3) of Definition 3.67. We get (3.28.3) from the induction hypothesis. (3.28.3) follows once again from clause (4) of Definition 3.67. □

Lemma 3.29 *Let* $\langle \mathcal{F}, \longrightarrow_R \rangle$ *be an abstract replacement system with residual map '/'. Let* $\mathcal{A}\rho$ *be a complete development of some* $\mathcal{P} \in \longrightarrow_R^{||}$. *Then for all* $\mathcal{Q} \in \longrightarrow_R^{||}$ *coinitial with* $\mathcal{P}$,

$$\mathcal{A} \,/\, (\mathcal{Q}/\rho) \;=\; \mathcal{A}/\mathcal{Q}$$

Proof We use induction on $|\rho|$. If $|\rho| = 0$, then $\rho = \epsilon$, and we have from clause (2) of Definition 3.67 that

$$\mathcal{A} \,/\, (\mathcal{Q}/\rho) \;=\; \mathcal{A} \,/\, (\mathcal{Q}/\epsilon) \;=\; \mathcal{A}/\mathcal{Q}$$

Otherwise, $|\rho| > 0$, and $\rho = \rho_1\mathcal{B}$ for some $\mathcal{B} \in \longrightarrow_R$ and $\rho_1 \in R^*$ such that $\mathcal{B} \in \mathcal{Q}$ and ρ_1 is a complete development of $\mathcal{Q} - \{\mathcal{B}\}$. We then get the following equations:

$$\begin{aligned}
\mathcal{A} \,/\, (\mathcal{P}/\rho) &= \mathcal{A} \,/\, (\mathcal{P} \,/\, (\rho_1\mathcal{B})) && (3.29.1)\\
&= \mathcal{A} \,/\, ((\mathcal{P}/\rho_1)/\mathcal{B}) && (3.29.2)\\
&= \mathcal{A} \,/\, (\mathcal{P}/\rho_1) && (3.29.3)\\
&= \mathcal{A}/\mathcal{P} && (3.29.4)
\end{aligned}$$

(3.29.2) follows from Lemma 3.28. (3.29.3) follows from clause (2) of Definition 3.65 (since $\mathcal{A} \parallel \mathcal{B}$), and finally, we get (3.29.4) from the induction hypothesis. □

The next lemma shows how developments of a parallel contraction interact with the contraction itself:

Lemma 3.30 *Let* $\langle \mathcal{F}, \longrightarrow_R \rangle$ *be an abstract replacement system with residual map '/',* $\mathcal{P}$ *be an element of* $\|(\longrightarrow_R)$, *and* $\rho \in R^*$ *be a complete development of* $\mathcal{Q} \subseteq \mathcal{P}$. *Then*

$$\begin{aligned}
\mathcal{P}/\rho &= \mathcal{P} - \mathcal{Q}\\
\rho/\mathcal{P} &= \emptyset^n
\end{aligned}$$

where $n = |\rho|$.

Proof We first show $\mathcal{P}/\rho = \mathcal{P} - \mathcal{Q}$ by induction on n. If $\mathcal{Q} = \emptyset$, then $\rho = \epsilon$, and by clause (2) of Definition 3.67,

$$\mathcal{P}/\epsilon \;=\; \mathcal{P}$$

Otherwise, $\rho = \mathcal{A}\rho_1$ for some $\rho_1 \in R^*$ and $\mathcal{A} \in \longrightarrow_R$, where ρ_1 is a complete development of $\mathcal{Q} - \{\mathcal{A}\}$ and $\mathcal{A} \in \mathcal{Q}$. We then have the following sequence of equations:

$$\begin{aligned}
\mathcal{P}/\rho &= \mathcal{P} \,/\, (\mathcal{A}\rho_1) && (3.30.1)\\
&= (\mathcal{P}/\mathcal{A}) \,/\, \rho_1 && (3.30.2)\\
&= (\mathcal{P} - \{\mathcal{A}\}) \,/\, \rho_1 && (3.30.3)\\
&= (\mathcal{P} - \{\mathcal{A}\}) - (\mathcal{Q} - \{\mathcal{A}\}) && (3.30.4)\\
&= \mathcal{P} - \mathcal{Q} && (3.30.5)
\end{aligned}$$

(3.30.2) follows from clause (3) of Definition 3.67. We then get (3.30.3) from clause (1) of Definition 3.65. Finally, (3.30.4) follows by the induction hypothesis.

Next, we show $\rho/\mathcal{P} = \emptyset^n$, once again using induction on n. If $\mathcal{Q} = \emptyset$ and thus $\rho = \epsilon$, then by clause (1) of Definition 3.67, $\rho/\mathcal{P} = \epsilon$. Otherwise, $\rho = \rho_2\mathcal{B}$, for some $\rho_2 \in R^*$ and $\mathcal{B} \in \longrightarrow_R$, where ρ_2 is a complete development of $\mathcal{Q} - \{\mathcal{B}\}$ and $\mathcal{B} \in \mathcal{Q}$. We then get the following equations:

$$\begin{aligned}
\rho/\mathcal{P} &= (\rho_2\mathcal{B}) \,/\, \mathcal{P} && (3.30.6)\\
&= (\rho_2/\mathcal{P})\ (\mathcal{B} \,/\, (\mathcal{P}/\rho_2)) && (3.30.7)\\
&= \emptyset^{n-1}\ (\mathcal{B} \,/\, (\mathcal{P}/\rho_2)) && (3.30.8)\\
&= \emptyset^{n-1}\ (\mathcal{B} \,/\, \mathcal{P}) && (3.30.9)\\
&= \emptyset^{n-1}\ \emptyset && (3.30.10)\\
&= \emptyset^{n} && (3.30.11)
\end{aligned}$$

(3.30.7) is a consequence of clause (4) of Definition 3.67. (3.30.8) follows from the induction hypothesis. We get (3.30.9) from Lemma 3.29. Finally, since

$$\mathcal{B} \in \mathcal{Q} \subseteq \mathcal{P}$$

we get (3.30.10) from clause (1) of Definition 3.65. □

We then get the following lemma as a corollary:

Lemma 3.31 *Let $\langle \mathcal{F}, \longrightarrow_R \rangle$ be an abstract replacement system with residual map '/', $\mathcal{P}$ be an element of $\|(\longrightarrow_R)$, and let $\rho \in R^*$ be a complete development of $\mathcal{P}$. Then*

$$\mathcal{P} \ \simeq \ \rho$$

Proof We have from Lemma 3.30 that $\mathcal{P}/\rho = \emptyset$ and $\rho/\mathcal{P} = \emptyset^n$ where $n = |\rho|$. Therefore, by Lemma 3.25, we have

$$\mathcal{P} \ \simeq \ \rho$$

□

The following lemma is the converse of Lemma 3.30:

Lemma 3.32 *Let $\langle \mathcal{F}, \longrightarrow_R \rangle$ be an abstract replacement system with residual map '/' Let $\mathcal{P}$ and $\mathcal{Q}$ be coinitial elements of $\longrightarrow^{\|}_R$, and let ρ be a complete development of $\mathcal{P}$ such that $\rho/\mathcal{Q} = \emptyset^n$ and $\mathcal{Q}/\rho = \mathcal{P} - \mathcal{T}$ for some $\mathcal{T} \subseteq \mathcal{P}$. Then*

$$\mathcal{Q} \ = \ \mathcal{T}$$

Proof We know from Theorem 3.31 that

$$\rho \;\simeq\; \mathcal{P}$$

We then have $\mathcal{Q}/\mathcal{P} \simeq \emptyset^n$ and $\mathcal{P}/\mathcal{Q} = \mathcal{P} - \mathcal{T}$ from clauses (2) and (8) of Proposition 3.27. But since $\mathcal{P}/\mathcal{Q} \in \;\longrightarrow^{||}_R$, we must have

$$\mathcal{P}/\mathcal{Q} \;=\; \emptyset$$

From clause 1 of Definition 3.65, we then conclude that $\mathcal{Q} = \mathcal{T}$. □

Putting Lemmas 3.32 and 3.31, we get the following theorem:

Theorem 3.33 *Let $\langle \mathcal{F}, \;\longrightarrow_R \rangle$ be an abstract replacement system with residual map '/'. Let $\mathcal{P}$ and $\mathcal{Q}$ be coinitial elements of $\longrightarrow^{||}_R$, and let ρ be a complete development of $\mathcal{P}$. Then $\mathcal{P} \simeq \rho$ iff $\mathcal{P} = \mathcal{Q}$.*

Proof Immediate from Lemmas 3.32 and 3.31. □

From Theorem 3.33, we can conclude that a complete development of a parallel contraction and the contraction itself behave identically *vis-a-vis* residuals.

Given the results above, we are now in a position to show that the residual of a two parallel contractions can in fact be *defined* in terms of the behaviors of the residuals of their constituent non-parallel contractions. We begin with the following theorem:

Theorem 3.34 *Let $\langle \mathcal{F}, \;\longrightarrow_R \rangle$ be an abstract replacement system with residual map '/'. Let $\mathcal{P}$ and $\mathcal{Q} \neq \emptyset$ be coinitial elements of $\longrightarrow^{||}_R$. Then for all $\mathcal{A} \in \mathcal{Q}$,*

$$\mathcal{P}/\mathcal{Q} \;=\; (\mathcal{P}/\mathcal{A})\,/\,(\mathcal{Q} - \{\mathcal{A}\})$$

Proof Since $\mathcal{Q} \neq \emptyset$, there exists some σ such that $(\mathcal{A}\sigma) \in R^*$ and σ is a complete development of $\mathcal{Q} - \{\mathcal{A}\}$. We then have the following:

$$\begin{aligned}
\mathcal{P}/\mathcal{Q} &\simeq \mathcal{P}/(\mathcal{A}\sigma) && (3.34.1)\\
&= (\mathcal{P}/\mathcal{A})\,/\,\sigma && (3.34.2)\\
&\simeq (\mathcal{P}/\mathcal{A})\,/\,(\mathcal{Q} - \{\mathcal{A}\}) && (3.34.3)\\
&= (\mathcal{P}/\mathcal{A})\,/\,(\mathcal{Q} - \{\mathcal{A}\}) && (3.34.4)
\end{aligned}$$

(3.34.1) follows from Theorem 3.33. (3.34.2) is from clause (3) of Definition 3.67. We get (3.34.3) from another application of Theorem 3.33. Finally, we know from property (3) of Lemma 3.21, and Lemma 3.25 that

$$\mathcal{P}/\mathcal{Q} \;\simeq\; (\mathcal{P}/\mathcal{A})\,/\,(\mathcal{Q} - \{\mathcal{A}\})$$

iff

$$\mathcal{P}/\mathcal{Q} \;=\; (\mathcal{P}/\mathcal{A})\,/\,(\mathcal{Q} - \{\mathcal{A}\})$$

Thus we get (3.34.4). □

Treating now the other element of a pair of parallel contractions, we get:

Theorem 3.35 *Let $\langle \mathcal{F}, \longrightarrow_R \rangle$ be an abstract replacement system with residual map '/'. Let $\mathcal{P}$ and $\mathcal{Q} \neq \emptyset$ be coinitial elements of $\longrightarrow_R^{||}$. Then*

$$\mathcal{P}/\mathcal{Q} \;=\; \bigcup\{\mathcal{A}/\mathcal{Q} \mid \mathcal{A} \in \mathcal{P}\}$$

Proof Let $\rho \in R^*$ be a complete development of $\mathcal{P}$. We proceed by induction on $|\rho|$. If $|\rho| = 0$, then $\rho = \epsilon$ and $\mathcal{P} = \emptyset$, in which case

$$\mathcal{P}/\mathcal{Q} \;=\; \mathcal{P}/\emptyset \;=\; \mathcal{P}$$

and we are done. Otherwise, $|\rho| > 0$ and $\rho = \rho_1\mathcal{A}$ for some $\rho_1 \in R^*$ and $\mathcal{A} \in \longrightarrow_R$ such that $\mathcal{A} \in \mathcal{P}$ and ρ_1 is a complete development of $\mathcal{P} - \{\mathcal{A}\}$. We then have the following:

$$\begin{aligned}
\mathcal{P}/\mathcal{Q} &\simeq (\rho_1\mathcal{A})\,/\,\mathcal{Q} && (3.35.1)\\
&= (\rho_1/\mathcal{Q})(\mathcal{A}\,/\,(\mathcal{Q}/\rho_1)) && (3.35.2)\\
&= (\rho_1/\mathcal{Q})(\mathcal{A}/\mathcal{Q}) && (3.35.3)\\
&\simeq ((\mathcal{P} - \{\mathcal{A}\})/\mathcal{Q})\,/\,(\mathcal{A}/\mathcal{Q}) && (3.35.4)\\
&= \bigcup\{\mathcal{A}/\mathcal{Q} \mid \mathcal{Q} \in \mathcal{P}\} && (3.35.5)
\end{aligned}$$

(3.35.1) follows from Theorem 3.33. We get (3.35.2) from clause 4 of Definition 3.67. (3.35.3) follows from Lemma 3.29. (3.35.4) follows from another application of Theorem 3.33. Finally, we get (3.35.5) from the induction hypothesis and clause (2) of Definition 3.65. As in the proof of Theorem 3.34, the equivalence under '$\simeq$' the proof yields must also be an equality. □

Theorems 3.34 and 3.35 show that our axiomatic definition of residual agrees closely with the behavior of residuals in the actual reduction systems we will consider in the sequel.

3.6.4 The Family Relation

Having defined a coherent notion of reduction equivalence modulo permutation, we would like also to define a related notion of equivalence for certain elements of $\longrightarrow_R$ that occur in coinitial reductions. We want our equivalence to identify those elements that are either residuals of a common contraction or are "created in the same way" by equivalent reductions. In this manner, we can identify contractions that are "essentially the same" across any set of coinitial reductions. This fact will be vital when comparing reduction strategies to extract maximal incrementality: we wish, if possible, to avoid reducing members of the same family more than once. Since such an equivalence is determined by the nature of the reductions in which the contractions are found, we must not only talk of the elements we wish to compare, but also the elements of the reduction that precede them, i.e., their "history." Thus we define the set of reduction/contraction pairs as follows:

Definition 3.71 (Reduction/Contraction Pair)
Let $\langle \mathcal{F},\ \longrightarrow_R \rangle$ be an abstract replacement system with residual map '/'. Then the set of reduction/contraction pairs, $F(R)$, is defined by:

$$F(R) \ \triangleq\ \{\langle \rho,\ \mathcal{A}\rangle \mid \mathcal{A} \in\ \longrightarrow_R \text{ and } (\rho\mathcal{A}) \in \|(R)^*\}$$

We will use the notation

$$\rho\mathcal{A}$$

to denote an element $\langle \rho,\ \mathcal{A}\rangle$ of $F(R)$, since we will frequently be concerned with its properties as a reduction.

Following Lévy, we begin by defining a relation on elements of $F(R)$ called the *progenitor relation*, as follows:

Definition 3.72 (Progenitor [Lév80])
Let $\langle \mathcal{F},\ \longrightarrow_R \rangle$ be an abstract replacement system with residual map '/'. Let $\rho\mathcal{A}$ and $\sigma\mathcal{B}$ be coinitial elements of $F(R)$. Then $\rho\mathcal{A}$ is a progenitor *of $\sigma\mathcal{B}$, notated*

$$\rho\mathcal{A} \ \leqslant\ \sigma\mathcal{B}$$

iff there is a reduction $\tau \in \|(R)^*$ coinitial with ρ and σ such that

$$\rho\tau \ \simeq\ \sigma$$

and

$$\mathcal{B} \in \mathcal{A}/\tau$$

(Lévy uses the term "copy" for the progenitor relation.) We will use the notation

$$\rho\mathcal{A} \ <\ \sigma\mathcal{B}$$

to indicate that $\rho\mathcal{A} \leqslant \sigma\mathcal{B}$ but $\rho \not\simeq \sigma$.

Extending the progenitor relation to an equivalence, we get the notion of *family*:

Definition 3.73 (Family [Lév80])
Let $\langle \mathcal{F},\ \longrightarrow_R \rangle$ be an abstract replacement system with residual map '/'. Let $\rho\mathcal{A}$ and $\sigma\mathcal{B}$ be coinitial elements of $F(R)$. Then $\rho\mathcal{A}$ and $\sigma\mathcal{B}$ are members of the same family*, notated*

$$\rho\mathcal{A} \ \lesseqgtr\ \sigma\mathcal{B}$$

iff one of the following hold:

(1)

$$\rho\mathcal{A} \leqslant \sigma\mathcal{B}$$

(2)

$$\sigma\mathcal{B} \leqslant \rho\mathcal{A}$$

(3) *There exists* $\mathcal{C} \in \longrightarrow_R$ *and* $\tau\mathcal{C} \in \|(R)^*$ *coinitial with* $\rho\mathcal{A}$ *and* $\sigma\mathcal{B}$ *such that*

$$\rho\mathcal{A} \lesseqgtr \tau\mathcal{C} \lesseqgtr \sigma\mathcal{B}$$

We will use the notation

$$[\rho\mathcal{A}]$$

to denote the family of $\rho\mathcal{A}$*, i.e., the equivalence class of* $\rho\mathcal{A}$ *with respect to* $\lesseqgtr$.

Two reduction/contraction pairs $\rho\mathcal{A}$ and $\sigma\mathcal{B}$ are in the same family if either one is a progenitor of the other (cases (1) and (2)) or there is some family member in common (case (3)). There is a rather important subtlety here: the family relation identifies contractions which could be residuals of a common contraction in *some* reduction (which, in the case above, would have τ as a prefix).

Note that if $\rho\mathcal{P} \in \|(R)^*$ is a reduction such that $\mathcal{P} \in \|(\longrightarrow_R)$ and for all $\mathcal{A}, \mathcal{B} \in \mathcal{P}$,

$$\rho\mathcal{A} \lesseqgtr \rho\mathcal{B}$$

then the notation

$$[\rho\mathcal{P}]$$

is unambiguous. This would be the case, e.g., if there existed $\sigma\mathcal{C} \in F(R)$ and $\tau \in \|(R)^*$ such that $\rho = \sigma\tau$ and $\mathcal{P} = \mathcal{C}/\tau$.

Lévy [Lév80] shows that the following properties hold of families (his proofs rely solely on the definition of family and properties of the relation $\lesssim$):

Proposition 3.36 (Properties of Families [Lév80, pp. 172–173])
Let $\langle \mathcal{F}, \longrightarrow_R \rangle$ *be an abstract replacement system with residual map '/'. Then the following properties hold:*

(1) *Let* $\rho\mathcal{A}$*,* $\rho'\mathcal{A}$*,* $\sigma\mathcal{B}$*, and* $\sigma'\mathcal{B}$ *be coinitial elements of* $F(R)$ *such that*

$$\rho' \simeq \rho \text{ and } \sigma' \simeq \sigma$$

Then

$$\rho\mathcal{A} \leqslant \sigma\mathcal{B} \text{ iff } \rho'\mathcal{A} \leqslant \sigma'\mathcal{B}$$
$$\rho\mathcal{A} \lesseqgtr \sigma\mathcal{B} \text{ iff } \rho'\mathcal{A} \lesseqgtr \sigma'\mathcal{B}$$

(2) $\leqslant$ *is a preorder, i.e., if* $\rho\mathcal{A}$*,* $\sigma\mathcal{B}$*, and* $\tau\mathcal{C}$ *are coinitial elements of* $F(R)$*, then the following hold:*

$$\rho\mathcal{A} \leqslant \sigma\mathcal{B} \leqslant \tau\mathcal{C} \quad \text{implies} \quad \rho\mathcal{A} \leqslant \tau\mathcal{C}$$
$$\rho\mathcal{A} \leqslant \sigma\mathcal{B} \leqslant \rho\mathcal{A} \quad \text{iff} \quad \rho \simeq \sigma \text{ and } \mathcal{A} \equiv \mathcal{B}$$

(3) *Let $\rho\mathcal{A}$, σ, and $\tau\mathcal{C}$ be coinitial elements of $F(R)$. Then if*

$$\rho \lesssim \sigma \lesssim \tau \text{ and } \rho\mathcal{A} \leqslant \tau\mathcal{C}$$

there exists $\mathcal{B} \in \longrightarrow_R$ such that $\sigma\mathcal{B}$ is coinitial with $\rho\mathcal{A}$ and $\tau\mathcal{C}$ and

$$\rho\mathcal{A} \leqslant \sigma\mathcal{B} \leqslant \tau\mathcal{C}$$

(4) *Let $\rho_1\mathcal{A}_1$, $\rho\mathcal{A}$, and $\rho_2\mathcal{A}_2$ be coinitial elements of $F(R)$ such that*

$$\rho_1\mathcal{A}_1 \quad \leqslant \quad \rho\mathcal{A} \quad \geqslant \quad \rho_2\mathcal{A}_2$$

Then there exists a unique $\rho_0\mathcal{A}_0 \in F(R)$ such that

$$\rho_1\mathcal{A}_1 \quad \geqslant \quad \rho_0\mathcal{A}_0 \quad \leqslant \quad \rho_2\mathcal{A}_2$$

and

$$\rho_0 \quad \simeq \quad [\rho_1] \vee [\rho_2]$$

3.6.5 Erasure and Generation

Contractions that do not contribute to a reduction, yet "disappear" after the reduction is performed are said to be *erased*:

Definition 3.74 (Erasure)
Let $\langle \mathcal{F}, \longrightarrow_R \rangle$ be an abstract replacement system with residual map '/'. Let $\mathcal{A}$ be an element of $\longrightarrow_R$ and ρ be a reduction of $\|(R)^*$ such that $\mathcal{A}$ and ρ are coinitial. Then $\mathcal{A}$ is erased *by ρ iff*

$$\mathcal{A}/\rho \quad = \quad \emptyset$$

and there exist no *reductions $\sigma, \tau \in \|(R)^*$ and elements $\mathcal{P} \in \|(\longrightarrow_R)$, $\mathcal{A}' \in \longrightarrow_R$ such that $\mathcal{A}' \in \mathcal{P}$,*

$$\rho \quad = \quad \sigma\mathcal{P}\tau$$

and

$$\mathcal{A} \quad \leqslant \sigma\mathcal{A}'$$

A contraction $\mathcal{A}$ is erased by a reduction ρ if the residual of $\mathcal{A}$ by ρ is null and $\mathcal{A}$ is the progenitor of no contraction in ρ.

Contractions that *always* contribute to reductions that yield an element in some set of interest are said to be *needed*:

Definition 3.75 (Needed Contraction)
*Let $\langle \mathcal{F}, \longrightarrow_R \rangle$ be an abstract replacement system with residual map '/', and let $\mathcal{N} \subseteq \mathcal{F}$ be a set of normal forms. Let X be an element of $\mathcal{F}$ and $\mathcal{A} \in \longrightarrow_R$ be a contraction applicable to X. Then $\mathcal{A}$ is $\mathcal{N}$-*needed *iff for every reduction $\rho \in \|(R)^*$ of X such that*

$$\rho : X \ \in \ \mathcal{N}$$

there exist reductions $\sigma, \tau \in \|(R)^*$ and elements $\mathcal{P} \in \|(\longrightarrow_R)$, $\mathcal{A}' \in \longrightarrow_R$ such that $\mathcal{A}' \in \mathcal{P}$,

$$\rho \ = \ \sigma \mathcal{P} \rho$$

and

$$\mathcal{A} \ \leqslant \ \sigma \mathcal{A}'$$

We will say that a contraction $\mathcal{A}$ is simply *needed* if the set $\mathcal{N}$ of normal forms in Definition 3.75 is NF_R (i.e., "$\mathcal{A}$ is needed" means $\mathcal{P}$ is NF_R-needed). Definition 3.75 says a contraction is needed if some member of its family is contracted in every reduction of an element X to some element in the set of normal forms $\mathcal{N}$. Needed contractions have also been called *persistent* by Staples [Sta80a,Sta80b,Sta80c].

Extending the idea of Definition 3.75 to reductions, we get the following:

Definition 3.76 (Call-By-Need Reduction)
*Let $\langle \mathcal{F}, \longrightarrow_R \rangle$ be an abstract replacement system with residual map '/', and let $\mathcal{N} \subseteq \mathcal{F}$ be a set of normal forms. Then a reduction ρ of $X \in \mathcal{F}$ is $\mathcal{N}$-*call-by-need *iff ρ has the form*

$$\rho : X_0 \xrightarrow{\mathcal{P}_1}_R X_1 \xrightarrow{\mathcal{P}_2}_R \cdots \xrightarrow{\mathcal{P}_n}_R X_n$$

and for all $i \geq 1$, $\mathcal{P}_i$ contains a $\mathcal{N}$-needed contraction of X_{i-1}. An R-reduction strategy s is call-by-need iff for all $X \in \mathcal{F}$, $s^i(X)$ is call-by-need for all i.

A contraction that is not a residual of a prefix of some reduction is said to be *created* by that reduction. We make this precise as follows:

Definition 3.77 (Contraction Creation)
Let $\langle \mathcal{F}, \longrightarrow_R \rangle$ be an abstract replacement system with residual map '/'. Let $\rho\mathcal{A}$ be an element of $F(R)$. Then $\mathcal{A}$ is created *by ρ iff there exists no $\mathcal{A}' \in \longrightarrow_R$ such that*

$$\epsilon \mathcal{A}' \ < \rho \mathcal{A}$$

If no contractions were ever created, all reductions would be finite.

Extending the idea of creation inductively to all elements of reduction requires a somewhat more complicated definition. We first define the notion of an *extraction* relation on reduction / contraction pairs.

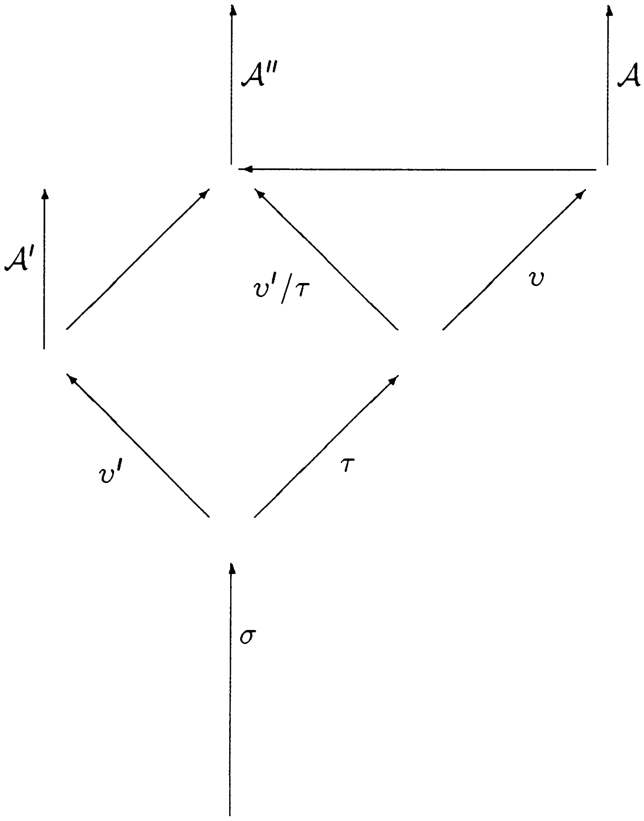


Figure 3.10: $\sigma\tau\nu\mathcal{A} \longrightarrow_{\text{ext}} \sigma\nu'\mathcal{A}'$

Definition 3.78 (Extraction Relation)
Let $\langle \mathcal{F}, \longrightarrow_R \rangle$ be an abstract replacement system with residual map '/'. Let $\sigma\tau\nu\mathcal{A}$ and $\sigma\nu'\mathcal{A}'$ be elements of $F(R)$ such that $\tau \neq \epsilon$. Then $\sigma\nu'\mathcal{A}'$ is derived from $\sigma\tau\nu\mathcal{A}$ by extraction*, notated*

$$\sigma\tau\nu\mathcal{A} \quad \longrightarrow_{\text{ext}} \quad \sigma\nu'\mathcal{A}'$$

iff the following hold:

$$\begin{array}{ccccc} \upsilon & \lesssim & \upsilon'/\tau & & \\ \upsilon'\mathcal{A}' & \leqslant & (\tau \vee \upsilon')\mathcal{A}'' & \geqslant & (\tau\upsilon)\mathcal{A} \end{array}$$

for some $\mathcal{A}'' \in \longrightarrow_R$.

Figure 3.10 depicts the essential elements of the extraction relation. In Definition 3.78, we "extract" a sub-reduction τ which is irrelevant to the creation of $\mathcal{A}$. This notion was defined more concretely in the setting of the lambda calculus by Lévy [Lév80]. The following is a consequence of Definition 3.78:

Lemma 3.37 *Let $\langle \mathcal{F}, \longrightarrow_R \rangle$ be an abstract replacement system with residual map '/'. Let $\rho\mathcal{A}$ and $\rho'\mathcal{A}'$ be elements of $F(R)$ such that $\rho\mathcal{A} \longrightarrow_{\text{ext}} \rho'\mathcal{A}'$. Then*

$$\rho\mathcal{A} \lesseqgtr \rho'\mathcal{A}'$$

Proof Immediate from Definition 3.78. □

We now generalize the notion of contraction creation to reductions as follows.

Definition 3.79 (Generator)
Let $\langle \mathcal{F}, \longrightarrow_R \rangle$ be an abstract replacement system with residual map '/', and let $\rho\mathcal{A}$ be a reduction in $\|(R)^*$ where $\mathcal{A} \in \longrightarrow_R$. Then ρ generates *$\mathcal{A}$ iff there exists no $\rho'\mathcal{A}' \in F(R)$ such that $\rho\mathcal{A} \longrightarrow_{\text{ext}} \rho'\mathcal{A}'$ (i.e., $\rho\mathcal{A}$ is a normal form with respect to the extraction relation). $\rho\mathcal{A}$ is a* generator *iff ρ generates $\mathcal{A}$.*

A generator is a reduction that creates an element $\mathcal{A}$ without performing intermediate contractions not germane to the creation of $\mathcal{A}$.

3.7 Regular Replacement Systems

Our final "system" definition combines the notion of an abstract replacement system and a residual map that obeys the following property:

Definition 3.80 (Stable Residual Maps)
Let $\Psi = \langle \mathcal{F}, \longrightarrow_R \rangle$ be an abstract replacement system with residual map '/'. Then '/' is stable *iff for all $\mathcal{P}, \mathcal{Q} \in \|(\longrightarrow_R)$ and $\mathcal{A} \in \longrightarrow_R$ such that $(\mathcal{P} \vee \mathcal{Q})\mathcal{A} \in F(R)$, there exists unique contractions $\mathcal{P}_0 \subseteq \mathcal{P}, \mathcal{Q}_0 \subseteq \mathcal{Q} \in \longrightarrow_R^{\|}$ and $\mathcal{A}_0 \in \longrightarrow_R$ such that the following hold:*

(1) $(\mathcal{P}_0 \vee \mathcal{Q}_0)\mathcal{A}_0 \in [(\mathcal{P} \vee \mathcal{Q})\mathcal{A}]$

(2) $[\mathcal{P}_0 \vee \mathcal{Q}_0]$ *is the greatest lower bound of the set*

$$\{\, [\tau] \mid \tau\mathcal{A}' \in [(\mathcal{P} \vee \mathcal{Q})\mathcal{A}] \textit{ for some } \mathcal{A}' \}$$

(where $[\mathcal{P}_0 \vee \mathcal{Q}_0]$ and $[\tau]$ are equivalence classes with respect to '$\simeq$', not '$\lesseqgtr$')

The stability property gives some minimal requirements for the creation of non-parallel contractions. In particular, it says that if $\mathcal{P}$ and $\mathcal{Q}$ are coinitial contractions such that $\mathcal{P}\mathcal{A}_1$, $\mathcal{Q}\mathcal{A}_2$, and $(\mathcal{P} \vee \mathcal{Q})\mathcal{A}'$ are all members of the same family, then Definition 3.80 requires that $\mathcal{A}_1$ and $\mathcal{A}_2$ cannot be created separately by $\mathcal{P}$ and $\mathcal{Q}$. Instead, there must exist some common reduction $\mathcal{S}\mathcal{A}_0$ that is the progenitor of both $\mathcal{P}\mathcal{A}_1$ and $\mathcal{Q}\mathcal{A}_2$, such that $\mathcal{S}$ creates $\mathcal{A}_0$. This is a technical condition that will be required to guarantee the existence of optimal reduction strategies (see Section 3.6.4 and Chapter 6).

Our use of the term *stable* is motivated by a similar definition for an important class of functions studied in the semantics of programming languages (see, e.g., [BCL85]). The connection between the property of Definition 3.80 and its analogue in semantics is made concrete in the setting of *monotone input/output automata* by Panangaden, Shanbhogue, and Stark [PSS89], although we will not explore the relationship further here.

As a consequence of Definition 3.80, we have the following lemma:

Lemma 3.38 *Let* $\langle \mathcal{F}, \longrightarrow_R \rangle$ *be an abstract replacement system with stable residual map '/'. Then for all* $\mathcal{PA} \in F(R)$*, if* $\mathcal{P}$ *does not create* $\mathcal{A}$*, there is a unique contraction* $\mathcal{A}_0 \in \longrightarrow_R$ *such that* $\mathcal{A} \in \mathcal{A}_0/\mathcal{P}$

Proof Immediate from Definition 3.80. □

Extending Lemma 3.38 to reductions, we have

Lemma 3.39 *Let* $\langle \mathcal{F}, \longrightarrow_R \rangle$ *be an abstract replacement system with stable residual map '/'. Then for all* $\rho\mathcal{A} \in F(R)$ *such that* ρ *does not create* $\mathcal{A}$*, there exists a unique* $\mathcal{A}_0 \in \longrightarrow_R$ *such that*

$$\mathcal{A} \;=\; \mathcal{A}_0/\rho$$

Proof Simple induction on $|\rho|$ using Lemma 3.38. □

We are now ready to define *regular* abstract replacement systems:

Definition 3.81 (Regular Abstract Replacement System)
Let $\langle \mathcal{F}, \longrightarrow_R \rangle$ *be an abstract replacement system with residual map '/', such that*

- *'/' is stable.*
- *The extraction relation on elements of* $\|(R)^*$, $\longrightarrow_{\mathrm{ext}}$*, is noetherian.*

Then $\langle \mathcal{F}, \longrightarrow_R, / \rangle$ *is a* regular abstract replacement system.

Use of the term "regular" is due to Klop [Klo80], whose restrictions on term-rewriting system are similar to those we require in Definition 3.81. Henceforth, we will refer to regular abstract replacement systems simply as "regular replacement systems" or RRS's. Abstract replacement systems that are RRS's will be the focus of the remainder of our study.

We are now in a position to make use of the stability property required of the residual maps in RRS's to sketch the results of Berry and Lévy in our more abstract setting. First, we have:

Lemma 3.40 *Let* $\langle \mathcal{F}, \longrightarrow_R, / \rangle$ *be an* RRS*. Then for all* $\rho\mathcal{A} \in F(R)$*, there exists* $\rho_0\mathcal{A}_0 \in F(R)$ *such that* $\rho_0\mathcal{A}_0$ *is a generator and*

$$\rho\mathcal{A} \;\lesseqgtr \rho_0\mathcal{A}_0$$

Proof Since $\langle \mathcal{F}, \longrightarrow_R, / \rangle$ is an RRS, its corresponding extraction relation $\longrightarrow_{\text{ext}}$ must be noetherian. Therefore, there must exist a generator $\rho_0\mathcal{A}_0$ such that

$$\rho\mathcal{A} \quad \twoheadrightarrow_{\text{ext}} \quad \rho_0\mathcal{A}_0$$

Then by definition of the extraction relation, $\rho\mathcal{A} \lesseqgtr \rho_0\mathcal{A}_0$. □

From 3.40, we see that every family of an RRS must contain at least one generator. In [BL79,Lév80], effective means are presented for constructing generators for reduction/contraction pairs in recursive program schemes and the lambda calculus, respectively.

We now have:

Lemma 3.41 *Let $\langle \mathcal{F}, \longrightarrow_R, / \rangle$ be an* RRS. *Then for all $\rho\mathcal{A}, \rho_0\mathcal{A}_0 \in F(R)$ such that $\rho_0\mathcal{A}_0$ is a generator and*

$$\rho\mathcal{A} \quad \twoheadrightarrow_{\text{ext}} \rho_0\mathcal{A}_0$$

there exists $\rho'\mathcal{A}' \in F(R)$ such that

$$\rho_0\mathcal{A}_0 \quad \leqslant \quad \rho'\mathcal{A}' \quad \geqslant \quad \rho\mathcal{A}$$

Proof By induction on the length of the reduction $\rho\mathcal{A} \twoheadrightarrow_{\text{ext}} \rho_0\mathcal{A}_0$, using the definition of $\longrightarrow_{\text{ext}}$ as a base case and Lemma 3.38 to prove the inductive step. □

Lemma 3.41 says that the construction of a generator guarantees that there will be be an element $\rho'\mathcal{A}'$ of $F(R)$ such that $\rho\mathcal{A}$ and $\rho_0\mathcal{A}_0$ are both progenitors of $\rho'\mathcal{A}'$.

We can now show that generators are unique up to equivalence under '$\simeq$':

Lemma 3.42 *Let $\langle \mathcal{F}, \longrightarrow_R, / \rangle$ be an* RRS. *Let $\rho\mathcal{A}$ be an element of $F(R)$, and $\rho_0\mathcal{A}_0, \rho_0'\mathcal{A}_0' \in F(R)$ be generators such that*

$$\begin{aligned} \rho\mathcal{A} &\quad \twoheadrightarrow_{\text{ext}} \quad \rho_0\mathcal{A}_0 \\ \rho\mathcal{A} &\quad \twoheadrightarrow_{\text{ext}} \quad \rho_0'\mathcal{A}_0' \end{aligned}$$

Then $\rho_0 \simeq \rho_0'$ and $\mathcal{A}_0 = \mathcal{A}_0'$.

Proof Assume $\rho_0 \not\simeq \rho_0'$. Then either $\rho_0/\rho_0' \not\simeq \epsilon$ or $\rho_0'/\rho_0 \not\simeq \epsilon$. Without loss of generality, assume the former. We then proceed by induction on $|\rho_0|$. If $|\rho_0| = 0$, then $\rho_0 = \epsilon$ and $\rho_0/\rho_0' \simeq \epsilon$, a contradiction. Otherwise, let $\rho_0 = \tau\mathcal{P}$ for some $\tau \in ||(R)^*$ and $\mathcal{P} \in \longrightarrow_R^{||}$. We use the following equalities to yield a contradiction:

$$\rho_0/\rho_0' \quad = \quad (\tau\mathcal{P})/\rho_0' \quad = \quad (\tau/\rho_0')\,(\mathcal{P} \,/\, (\rho_0'/\tau))$$

Since $\rho_0/\rho_0' \not\simeq \epsilon$, then one or both of the following must hold:

$$\begin{aligned} (\mathcal{P} \,/\, (\rho_0'/\tau)) &\quad \not\simeq \quad \emptyset \\ (\tau/\rho_0') &\quad \not\simeq \quad \epsilon \end{aligned}$$

In the first case, either the stability property (Definition 3.80) or the fact that ρ_0 and ρ_0' are generators is violated. In the second case, the result follows by induction. Thus we must have $\rho_0/\rho_0' \simeq \epsilon$ and $\rho_0'/\rho_0 \simeq \epsilon$, from which we conclude that

$$\rho_0 \simeq \rho_0'$$

The fact that $\mathcal{A}_0 = \mathcal{A}_0'$ then follows from Lemma 3.39. $\square$

Berry and Lévy [BL79] show (in the setting of recursive program schemes) that each family class has a *canonical subfamily* which is a unique up to equivalence under $\simeq$. Their proof relies on the existence of generators and (via the stability property) their uniqueness up to equivalence under ' $\simeq$'. More specifically, their proof relies on the results of Lemmas 3.40, 3.41, 3.42 and Proposition 3.36, and can be adapted straightforwardly to the setting of RRS's:

Proposition 3.43 ([BL79, Thm. 3.1.1, Prop. 3.1.7]) *Let*

$$\langle \mathcal{F}, \longrightarrow_R, / \rangle$$

be an RRS. *Then for every reduction $\rho\mathcal{A} \in \|(R)^*$ such that $\mathcal{A} \in \longrightarrow_R$, there corresponds a set of elements of $F(R)$ called the* canonical subfamily *of $[\rho\mathcal{A}]$, notated*

$$[\rho\mathcal{A}]_0$$

such that $[\rho\mathcal{A}]_0 \subseteq [\rho\mathcal{A}]$, all elements of $[\rho\mathcal{A}]_0$ are generators, and the following hold:

(1) *For all $\sigma\mathcal{A}' \in F(R)$,*

$$\rho\mathcal{A} \lesseqgtr \sigma\mathcal{A}' \ \textit{iff}\ [\rho\mathcal{A}]_0 = [\sigma\mathcal{A}']_0$$

(2) *For all $\rho_0\mathcal{A}_0 \in [\rho\mathcal{A}]_0$,*

$$\rho_0 \lesssim \rho \ \textit{iff}\ \rho_0\mathcal{A}_0 \leqslant \rho\mathcal{A}$$

Clause (2) of Proposition 3.43 makes it possible to prove properties of certain families (those for which $\rho_0 \lesssim \rho$) using *only* properties of the progenitor relation '$\leqslant$', i.e., properties of residuals. It will turn out that for certain sorts of reductions, we are guaranteed *a priori* that $\rho_0 \lesssim \rho$.

3.7.1 Relative and Complete Reductions

We now consider reductions that are closely related, not necessarily in the sense of the equivalence defined by $\simeq$, but instead because they contain contractions from the same set of families. We begin with the following definition:

Definition 3.82 (Reduction Families)
Let $\langle \mathcal{F}, \longrightarrow_R, / \rangle$ *be an* RRS, *and let* ρ *be an element of* $\|(R)^*$. *Then the* set of families reduced by ρ, *notation* $\mathrm{FAM}(\rho)$, *is defined as follows:*

$$\mathrm{FAM}(\rho) \triangleq \{[\sigma\mathcal{A}] \mid \rho = \sigma\mathcal{P}\tau \text{ and } \mathcal{A} \in \mathcal{P} \text{ for some } \sigma, \tau, \mathcal{A}, \mathcal{P}\}$$

$\mathrm{FAM}(\rho)$ is simply the set of all families to which some contraction in ρ belongs.

We now define reductions that reduce only members of a given family:

Definition 3.83 (Family-Relative Reduction)
Let $\langle \mathcal{F}, \longrightarrow_R, / \rangle$ *be an* RRS. *Let* $\mathcal{S}$ *be a set of families, Then* ρ *is* relative to $\mathcal{S}$ *iff*

$$\mathrm{FAM}(\rho) \subseteq \mathcal{S}$$

Comparing the families reduced in two reductions, we make the following definition:

Definition 3.84 (Relative Reduction)
Let $\langle \mathcal{F}, \longrightarrow_R, / \rangle$ *be an* RRS. *Let* ρ *and* σ *be coinitial elements of* $\|(R)^*$. *Then* ρ *is* relative to σ *iff*

$$\mathrm{FAM}(\rho) \subseteq \mathrm{FAM}(\sigma)$$

We now define reductions in which only sets of contractions from a single family are performed at each step:

Definition 3.85 (Family-Parallel Reduction)
Let $\langle \mathcal{F}, \longrightarrow_R, / \rangle$ *be an* RRS *and* ρ *be an element of* $\|(R)^*$, *where*

$$\rho \equiv \mathcal{P}_1\mathcal{P}_2 \ldots \mathcal{P}_n$$

Then ρ *is* family-parallel *iff for all* $i \geq 1$, $\mathcal{P}_i \neq \emptyset$, *and for all* $\mathcal{A}, \mathcal{B} \in \mathcal{P}_i$,

$$\rho_{i-1}\mathcal{A} \lesseqgtr \rho_{i-1}\mathcal{B}$$

where for $i \geq 1$, $\rho_i = \rho_{i-1}\mathcal{P}_i$ *and* $\rho_0 = \epsilon$. *The set of all family-parallel reductions of* $\|(R)^*$ *is denoted by*

$$F(R)^*$$

A family-parallel reduction in which only sets of *maximal* size are contracted at each step is *family-complete*:

Definition 3.86 (Family-Complete Reduction)
Let $\langle \mathcal{F}, \longrightarrow_R, / \rangle$ *be an* RRS *and* ρ *be an element of* $\|(R)^*$, *where*

$$\rho \equiv \mathcal{P}_1\mathcal{P}_2 \ldots \mathcal{P}_n$$

Then ρ *is* family-complete *iff* ρ *is family-parallel, and each* $\mathcal{P}_i$ *is a* maximal *set of members of the same family.*

A closely related class of reductions is the following:

Definition 3.87 (Residual-Complete Reduction)
Let $\langle \mathcal{F}, \longrightarrow_R, / \rangle$ be an RRS *and ρ be an element of $\|(R)^*$, where*

$$\rho \equiv \mathcal{P}_1\mathcal{P}_2 \ldots \mathcal{P}_n$$

Then ρ is residual-complete *iff for all $i \geq 1$, $\mathcal{P}_i \neq \emptyset$, and $\mathcal{P}_i$ is a* maximal *element of $\|(\longrightarrow_R)$ such that*

$$\mathcal{P}_i = \{\mathcal{A}' \mid \rho_j\mathcal{A} \leqslant \rho_{i-1}\mathcal{A}' \text{ for some } \mathcal{A}' \text{ and } j \leq i-1\}$$

A residual-complete reduction reduces a maximal set of residuals of some contraction that could have been performed earlier. Note that such a set is always disjoint and finite.

The following lemma is from Berry and Lévy [BL79]:

Lemma 3.44 ([BL79, Lemma 3.2.1]) *Let $\langle \mathcal{F}, \longrightarrow_R, / \rangle$ be an* RRS *and $\rho\mathcal{P}$ be a family-complete reduction of $\|(R)^*$. Then*

$$[\rho\mathcal{P}]_0 \leqslant \rho\mathcal{P}$$

Lemma 3.44 says that every family member contracted in a complete reduction has a *progenitor* in its corresponding canonical family.

We then have the following lemma from [Lév80] relating the two types of complete reductions (its proof relies on Lemma 3.44 and Proposition 3.43):

Lemma 3.45 ([Lév80, Lemma 6.2]) *Let $\langle \mathcal{F}, \longrightarrow_R, / \rangle$ be an* RRS *and ρ be an element of $\|(R)^*$. Then ρ is family-complete iff ρ is residual-complete.*

Thus a reduction that reduces all the members of some family at every step also reduces a complete set of residuals at every step. Lemma 3.45 also implies that the maximal set of members of a given family class at any stage of a reduction is always disjoint, since it corresponds to a maximal set of residuals, which is disjoint by definition. In light of the correspondence between family-completeness and residual-completeness, we will simply speak of *complete* reductions, using the following notation:

Definition 3.88 (Complete Reduction)
Let $\langle \mathcal{F}, \longrightarrow_R, / \rangle$ be an RRS *and ρ be an element of $\|(R)^*$. Then the set of all (family or residual)-complete reductions is notated by*

$$C(R)^*$$

We now have the following theorem from Berry and Levy [BL79], (which also relies on Lemma 3.44 and Proposition 3.43):

Theorem 3.46 ([BL79, Prop. 3.2.3]) *Let $\langle \mathcal{F}, \longrightarrow_R, / \rangle$ be an* RRS *and ρ be an element of $C(R)^*$ such that*

$$\rho \equiv \rho \mathcal{P} \sigma \mathcal{Q}$$

Then

$$[\rho \mathcal{P}] \neq [(\rho \mathcal{P} \sigma) \mathcal{Q}]$$

Theorem 3.46 (which relies for its proof on Lemma 3.44 and clause (2) of Proposition 3.43) says that a complete reduction contracts each family class only once. From this fact, the following corollary is immediate:

Corollary 3.47 *Let $\langle \mathcal{F}, \longrightarrow_R, / \rangle$ be an* RRS *and ρ be an element of $C(R)^*$. Then the following hold:*

(1)

$$|\rho| = |\mathrm{FAM}(\rho)|$$

(2) *If some reduction $\sigma \in \|(R)^*$ coinitial with ρ is relative to ρ, then*

$$|\rho| \leq |\sigma|$$

If ρ is finite, then σ is also finite.

clause (2) of Corollary 3.47 will be exploited in Chapter 6 to show that certain complete reductions are optimal for the class of all family-parallel reductions.

We have following proposition from Berry and Lévy [BL79]:

Proposition 3.48 ([BL79, Prop. 3.2.2]) *Let $\langle \mathcal{F}, \longrightarrow_R, / \rangle$ be an* RRS, *and let X be an element of $\mathcal{F}$. Let $CR(X)/\simeq$ be the set of equivalence classes of complete reductions of X in $C(R)^*$ whose members are ordered by*

$$\rho \lesssim \sigma \quad \textit{implies} \quad [\rho] \lesssim [\sigma]$$

(where the meaning of $\lesssim$ is again overloaded). Then $CR(X)$ is a sublattice of $R(X)$ such that for all coinitial reductions $\rho, \sigma \in C(R)^*$, the least upper bound $[\rho] \vee [\sigma]$ has the property

$$[\rho] \vee [\sigma] = [\rho \vee \sigma] = [\sigma \vee \rho]$$

The proof of Theorem 3.48 relies on clause (2) of Proposition 3.43, Proposition 3.36, and Lemma 3.44. One consequence of this theorem is that complete reductions are closed with respect to the residual operation.

RRS's with strong residual maps are particularly well-behaved with respect to families:

Theorem 3.49 *Let $\langle \mathcal{F},\ \longrightarrow_R,\ /\rangle$ be an* RRS *with a strong residual map. Then all reductions of R^* are complete.*

Proof We simply note that since '/' is a strong residual map, the residual of any contraction in $\longrightarrow_R$ is always an element of $\longrightarrow_R$. Therefore, if any contraction in a reduction of such an RRS is a residual of some previous contraction, it is maximal with respect to the residual map. Therefore, all reductions are complete. □

Given Theorem 3.22, the following corollary to Theorem 3.49 is immediate:

Corollary 3.50 $\langle \mathcal{F},\ \longrightarrow_R,\ /\rangle$ *be a* disjoint RRS. *Then all reductions of R^* are complete.*

3.7.2 Implementations and Complete Reductions

Implementations (see Definition 3.24) which preserve *complete* reductions will be useful. We therefore make the following definition:

Definition 3.89 (Completeness-Preserving Implementation)
Let

$$\begin{aligned}\Phi &= \langle \mathcal{E},\ \longrightarrow_R,\ /_{\Phi}\rangle \\ \Psi &= \langle \mathcal{F},\ \longrightarrow_S,\ /_{\Psi}\rangle\end{aligned}$$

be RRS*'s, such that*

$$\langle \Psi,\ g,\ h\rangle$$

is an implementation of Φ. Then $\langle \Psi,\ g,\ h\rangle$ is family-preserving *iff for all $E \in \mathcal{E}$, $F' \in \mathcal{F}$, and complete reduction $\rho \in C(S)^*$ such that*

$$\rho : h(E) \;=\; F'$$

there exists a complete reduction $\sigma \in C(R)^*$ such that

$$\sigma : E \;=\; g(F')$$

If it is also the case that for every $E \in \mathcal{E}$, $F', F'' \in \mathcal{F}$, and complete reduction $\rho\mathcal{P} \in C(S)^*$ such that

$$\begin{aligned}\rho : h(E) &= F' \\ \mathcal{P} : F' &= F''\end{aligned}$$

there exists reduction $\sigma\mathcal{Q} \in C(R)^*$ such that

$$\begin{aligned}\sigma : E &= g(F') \\ \mathcal{Q} : g(F') &= g(F'')\end{aligned}$$

then we say that $\langle \Psi,\ g,\ h\rangle$ is a strong *family-preserving implementation.*

The difference between (weak) family-preserving implementations and strong ones is that the strong implementations preserve not only completeness, but also the *length* of the complete reduction being simulated.

Having completed our study of regular replacement systems in the abstract, we will examine instances of such systems in the next chapter.

Chapter 4

The Lambda Calculus and Term Rewriting Systems

"A double variation on two contrasted ideas." —Tovey

In this chapter, we will discuss in detail the two formal systems that will concern us for the remainder of this thesis: the lambda calculus and term rewriting systems. We will also discuss the Brouwerian algebras of *labeled trees* and *term graphs*—the means we will use to *implement* the two systems. We will show that a large class of term rewriting systems form regular replacement systems.

In practice, terms may be manipulated for the purposes of computation by a number of different concrete data structures. We will consider two such *representations* of terms: *labeled trees* and *term graphs*.

We begin the chapter by discussing the syntax of "programs" or "functions" defined in these systems, using the notion of a *term*. We will consider a term to be an abstract tree (abstract in the sense that we will not be concerned with the details of its construction) labeled with operation symbols from a fixed *signature* and (possibly) variables.

We then discuss labeled trees. A labeled trees is simply a concrete manifestation of a term to which Brouwerian algebra operations may be applied. By manipulating terms as labeled trees, certain operations that are dependent on tree *location* become much easier to define than they would be in the framework of pure terms.

We then turn to the various formal systems with which we will be concerned, defining first conditional and unconditional term rewriting systems, and then defining several variants of the lambda calculus.

Finally, we return to the issue of concrete structures for implementation. While trees are perhaps the most "obvious" manifestation of terms, it was soon realized that other structures were more appropriate for implementing reduction systems. By using structures in which identical subtrees can be *shared* (i.e., graphs), computations on terms can be performed much more efficiently. Following [BvEG+87a], we call such graphs *term graphs*. We saw in Chapter 2 that the ability to transform such shared subtrees obviates repeated and inefficient transformation of identical subtrees in a non-shared implementation.

Our interest in graphs stems from this: by judiciously sharing subgraphs in appropriate data structures, we can preserve the results of computations that might otherwise be repeated in later computations. This will be the basis for our implementation of incremental reduction. In keeping with this interest in implementation, our discussion below will focus on the syntactic and operational aspects of manipulations of terms, trees, and graphs, rather than their semantic or logical properties.

We conclude the chapter by sketching a natural graph-based interpretation of the lambda calculus that also renders it an RRS. Thus the results from Chapter 3 can be applied to such systems.

4.1 Terms

In this section, we briefly review the well-known notion of a *sorted algebra*. Much of the notation of this section is adapted from Goguen [Gog90], Huet and Oppen [HO80], Huet [Hue80], or Hardin [Har89].

4.1.1 Signature

We will assume that the structures we are interested in manipulating represent one or more disjoint domains of interest. We thus associate with each such domain a *sort*, or name. For instance, in systems manipulating lists, we might have two sorts: *list* and *atom*.

We first require the following bit of notation: If $\mathcal{S}$ is a set, then $\mathcal{S}^n$ will denote the set of n-tuples, lists of elements of $\mathcal{S}$ of length n. $\mathcal{S}^*$ denotes the set of lists of arbitrary length. A particular list of elements will be notated

$$\langle\langle s_1, s_2, \ldots, s_n \rangle\rangle$$

Such a list will often be abbreviated by $\vec{s}$. $\emptyset$ will denote the empty list.

We then define $\mathcal{S}$-*sorted* sets as follows:

Definition 4.1 (Sorted Set)
*Let $\mathcal{S}$ be a set of sorts. Then an $\mathcal{S}$-*sorted set *$\mathcal{A}$ is a family of sets indexed by $\mathcal{S}$ of the form*

$$\mathcal{A} \triangleq \{\mathcal{A}_s \mid s \in \mathcal{S}\}$$

We define $|\mathcal{A}|$ by

$$|\mathcal{A}| \triangleq \bigcup_{s \in \mathcal{S}} \mathcal{A}_s$$

If $\mathcal{A}$ is a sorted set, unless otherwise specified, "$a \in \mathcal{A}$" is to be interpreted as shorthand for "$a \in |\mathcal{A}|$".

We can now define the notion of *signature*:

Definition 4.2 (Signature)
Given a set of sorts $\mathcal{S}$, an $\mathcal{S}$-sorted signature Σ *is a sorted set of the form*

$$\Sigma \triangleq \{\Sigma_{\vec{w},s} \mid \langle \vec{w},\ s\rangle \in \mathcal{S}^* \times \mathcal{S}\}$$

$\Sigma_{\vec{w},s}$ *consists of a set of* operation symbols *of* arity $\vec{w}$, *of* sort s, *and* rank $\langle \vec{w},\ s\rangle$. *An operation symbol f is a* constant symbol *iff $f \in \Sigma_{\emptyset,s}$. Σ is a* ground signature *iff it consists only of distinct constant symbols.*

4.1.2 Terms and Algebras

We define the notion of a Σ-*algebra* as follows:

Definition 4.3 (Σ-Algebra)
Given an $\mathcal{S}$-sorted signature Σ, a Σ-algebra *is a pair $\langle \mathcal{A},\ \mathcal{F}\rangle$ where $\mathcal{A}$, the algebra's* carrier, *is an $\mathcal{S}$-sorted set, and $\mathcal{F}$ is a $|\Sigma|$-indexed family of functions, the* fundamental operations *(or* interpretation functions*) of the algebra, such that for all $f \in \Sigma$, the following hold:*

(1) $f \in \Sigma_{\emptyset,s}$ *implies* $\mathcal{F}_f \in \mathcal{A}_s$.

(2) $f \in \Sigma_{\langle\langle s_1,s_2,\ldots,s_n\rangle\rangle,s}$ *implies*

$$\mathcal{F}_f \in \mathcal{A}_{s_1} \times \mathcal{A}_{s_2} \times \cdots \times \mathcal{A}_{s_n} \to \mathcal{A}_s$$

We will generally refer to a Σ-algebra by its carrier, i.e., if $\langle \mathcal{A},\ \mathcal{F}\rangle$ is a Σ-algebra, we will simply refer to it as $\mathcal{A}$. Also, we will generally use f to denote $\mathcal{F}_f$, when no confusion arises.

We now define *morphisms*, structure-preserving mappings between Σ-algebras:

Definition 4.4 (Σ-Morphism)
Let $\mathcal{A}$ and $\mathcal{B}$ be two Σ-algebras, and for all $f \in \Sigma$, let $f_{\mathcal{A}}$ and $f_{\mathcal{B}}$ denote the corresponding fundamental operations in the two algebras. Then a sort-preserving mapping

$$h: \mathcal{A} \to \mathcal{B}$$

is a Σ-morphism *iff for all $f \in \Sigma_{\langle\langle s_1,s_2,\ldots,s_n\rangle\rangle,s}$, and elements $t_1, t_2, \ldots, t_n \in \mathcal{A}$,*

$$h(\, f_{\mathcal{A}}(t_1, t_2, \ldots, t_n)\,) \;=\; f_{\mathcal{B}}(\, h(t_1), h(t_2), \ldots, h(t_n)\,)$$

If h is an injection, we say that it is a monomorphism. *If h is a bijection, we say that it is an* isomorphism.

Given a signature Σ, we can construct trees labeled with the operation symbols of Σ (the notion of "tree" will, for the time being, be treated rather informally). Such *ground terms* are defined as follows:

Definition 4.5 (Ground Term)
Let Σ be an S-sorted signature. Then the set of ground terms *generated by Σ, notated $GTer(\Sigma)$ is defined as follows:*

$$GTer(\Sigma) = \bigcup_{s \in S} GTer_s(\Sigma)$$

where for each sort s, we define $GTer_s(\Sigma)$ inductively as follows:

(1) *If f is an element of $\Sigma_{\emptyset,s}$ then the tree consisting of a single node labeled f is an element of $GTer_s(\Sigma)$.*

(2) *If f is an element of $\Sigma_{\langle\langle s_1, s_2, \ldots s_n \rangle\rangle, s}$, then the tree with one node labeled f and n successors $t_1, t_2, \ldots, t_n$ such that*

$$t_1 \in GTer_{s_1}(\Sigma),\ t_2 \in GTer_{s_2}(\Sigma), \ldots, t_3 \in GTer_{s_3}(\Sigma)$$

is an element of $GTer_s(\Sigma)$.

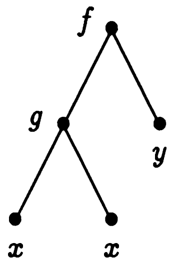
The algebra whose carrier is $GTer(\Sigma)$ and whose operations construct terms from subterms of the appropriate sorts is called the *initial* Σ-algebra. Thus, e.g., the terms from Example 2.1 (equations for group theory) are constructed from a single-sorted signature Γ_g consisting of the nullary operation '0', the unary operation '$-$', and the binary operation '$+$'.

We will use the standard prefix notation for terms, e.g.:

Example 4.1

$$f(g(x,x),\ y)$$

represents the term



4.1.3 Variables and Contexts

Trees that may contain *variables* as well as function symbols from a Σ-algebra are *terms*:

Definition 4.6 (Term)
*Let $\mathcal{V}$ be a denumerable $\mathcal{S}$-sorted set of distinct symbols, e.g., $\mathcal{V}_s = \{x_s, y_s, z_s, \ldots\}$ such that $\mathcal{V}$ and Σ are disjoint. Then ground terms constructed from the signature $\Sigma \cup \mathcal{V}$ will be called $\Sigma(\mathcal{V})$-*terms *(where we use $\Sigma(\mathcal{V})$ as a shorthand for $\Sigma \cup \mathcal{V}$).*

We use $Ter(\Sigma)$ to denote the set of $\Sigma(\mathcal{V}_0)$-terms, where $\mathcal{V}_0$ is a "standard" set of variables containing an infinite number of variables of every sort in $\mathcal{S}$. The set of variables in a term t is given by $V(t)$.

The Σ-algebra whose carrier is the set of $\Sigma(\mathcal{V})$-terms and whose operations construct terms from subterms of the appropriate sort is called the *free* Σ-algebra generated by $\mathcal{V}$.

We will often use a special set of variables indexed by natural numbers, called *holes*. Terms constructed using holes are called *contexts*, defined formally as follows:

Definition 4.7 (Context)
Let Σ be a signature, and $\mathcal{H}$ be the S-sorted set of variables indexed by natural numbers, where for each sort $s \in \mathcal{S}$, $\mathcal{H}_s$ contains:

$$①_s, ②_s, ③_s, \ldots$$

Then the set of contexts, *notated $Con(\Sigma)$, is the set of $\Sigma(\mathcal{H})$-terms such that each hole-labeled tree node has a* unique *index.*

The prohibition on multiple instances of identically indexed holes implies that there is a unique correspondence between hole occurrences and their indices. Holes will usually be used to indicate occurrences in a term that will eventually be replaced by non-hole subterms. We use a special notation for holes to distinguish their "structural" function from the logical or semantic function usually ascribed to variables[1]. We will overload the meaning of ⓘ to refer to the unique occurrence of ⓘ in any term that contains it.

We will use the notation

$$C[①, ②, \ldots, ⓝ]$$

to denote an arbitrary context containing exactly the holes indexed $1, 2, \ldots, n$. When a term contains only a single hole, we will denote it by $\odot$, and C will denote the context $C[\odot]$.

4.2 Labeled Trees and Forests

By refining the notion of context forest introduced in Section 3.3.2, we can formalize the notion of "tree" used informally above to define terms. We begin by allowing

[1]Note, however, that in the lambda calculus, variables can be interpreted both structurally and logically, which accounts for the complexity of variable manipulation in that formalism.

the integer sequences that represent context forest nodes to be augmented with function symbols from some signature:

Definition 4.8 (Σ-Node)

Let Σ be a signature and $\mathbb{P}^*$ be the set of positive integer sequences. Then a Σ-node *is a pair of the form*

$$\langle f,\ s\rangle$$

where $f \in \Sigma$ and $s \in \mathbb{P}^*$. The set of Σ-nodes is notated $TNode(\Sigma)$.

If $n \equiv \langle f,\ s\rangle$ is an element of $TNode(\Sigma)$, we use the following notation to gain access to the components of n:

$$\begin{aligned} op(n) &\triangleq \mathcal{F} \\ addr(n) &\triangleq s \end{aligned}$$

$op(n)$ is n's operation set. *$addr(n)$ is n's* path address. *If $\mathcal{N} \subseteq TNode(\Sigma)$ is a set of nodes, we extend the notation above as follows:*

$$\begin{aligned} op(\mathcal{N}) &\triangleq \bigcup\{op(n) \mid n \in \mathcal{N}\} \\ addr(\mathcal{N}) &\triangleq \bigcup\{addr(n) \mid n \in \mathcal{N}\} \end{aligned}$$

The idea of Definition 4.8 is that a node is defined in the same manner as for (unlabeled) context forests, but now each node may be labeled with sets of function symbol from a signature Σ. We allow nodes to be labeled with sets, rather than single function symbols in order to facilitate the appropriate definition of a Brouwerian algebra.

We now define the set of labeled context forests:

Definition 4.9 (Σ-Labeled Context Forests)

A Σ-labeled context forest f is a set of elements of $TNode(\Sigma)$. The set of all Σ-labeled context forests is notated $\mathcal{F}(\Sigma)$.

A Σ-labeled context forest f is proper *iff for all nodes n_1, n_2 of f,*

$$addr(n_1) = addr(n_2) \quad implies \quad n_1 = n_2$$

f is sort-respecting *iff it is proper, and for all nodes $n_1, n_2 \in f$ such that*

$$addr(n_1) \;=\; s$$

and

$$addr(n_2) \;=\; s \circ i$$

we have that

$$op(n_1) \in \Sigma_{\langle\!\langle s_1, s_2, \ldots, s_n \rangle\!\rangle, s}$$

implies that $i \leq n$ and

$$op(n_2) \in \Sigma_{\vec{w}, s_i}$$

for some $\vec{w}$.

If f is an element of $\mathcal{F}(\Sigma)$, then the set $addr(f)$ is called the skeleton *of f.*

The definition of Σ-labeled context forest is thus identical to that of the unlabeled variety (Definition 3.36), except for the addition of labels. Proper forests are those in for which every node has a unique corresponding address. In the sequel most of the Σ-labeled context forests with which we will be concerned will turn out to be proper.

The set of Σ-labeled trees and subtrees are defined in a manner analogous to their counterparts in Definitions 3.37 and 3.38:

Definition 4.10 **(Σ-Labeled Tree)**
Let t be an element of $\mathcal{F}(\Sigma)$. Then t is a Σ-labeled tree *iff* $\mathsf{addr}(t) \in \mathcal{T}_0$, *i.e., the skeleton of t is a tree. We denote the set of all Σ-labeled trees by $\mathcal{T}_0(\Sigma)$.*

Definition 4.11 **(Σ-Labeled Subtree)**
Let s be an element of $\mathcal{F}(\Sigma)$. Then s is a Σ-labeled subtree *iff* $\mathsf{addr}(s) \in \mathcal{T}$, *i.e., the skeleton of s is a subtree. We denote the set of all Σ-labeled subtrees by $\mathcal{T}(\Sigma)$.*

If s is proper, then there exactly one node $n_0 \in s$ such that $\mathsf{addr}(n_0) = root(\mathsf{addr}(s))$. *We thus refer to n_0 as the* root *of subtree s, notated* $\mathsf{root}(t)$.

As with unlabeled context forests, we can define a trivial Brouwerian algebra on elements of $\mathcal{F}(\Sigma)$ using set operations as follows:

Proposition 4.1 *Let Σ be a signature, and $\mathcal{F}(\Sigma)$ be the set of Σ-labeled context forests. For any two elements $s, t \in \mathcal{F}(\Sigma)$, define the operations '$\sqcup$', '$\sqcap$', and '$\dot{-}$' as follows:*

$$
\begin{aligned}
s \sqcup t &\triangleq s \cup t \\
s \sqcap t &\triangleq s \cap t \\
s \dot{-} t &\triangleq s - t
\end{aligned}
$$

Then $\langle \mathcal{F}(\Sigma), \sqcup, \sqcap, \dot{-}, TNode(\Sigma)\rangle$ is a Boolean (thus Brouwerian) algebra.

In the sequel, any algebraic operation on trees, subtrees, or context forests are assumed to be those of Proposition 4.1.

4.2.1 Trees and Terms

Having defined the Brouwerian algebra of labeled context forests, we observe that the proper elements of $\mathcal{T}(\Sigma(\mathcal{V}_0))$ can be regarded as Σ-terms. By treating terms as elements of $\mathcal{T}(\Sigma(\mathcal{V}_0))$, we will be able to make informal manipulations of terms precise. For instance, we will take "s is a subterm of t" to mean that $s \sqsubseteq t$.

In defining the correspondence between terms and elements of $\mathcal{T}(\Sigma(\mathcal{V}_0))$, the following properties will be useful:

Proposition 4.2 *Let Σ be a signature. Then the following hold:*

(1) *If s and t are proper elements of $\mathcal{F}(\Sigma)$ such that* $\mathbf{addr}(s) \parallel \mathbf{addr}(t)$, *then* $s \sqcup t$ *and* $s \dot{-} t$ *are proper.*

(2) *If t is a proper element of $\mathcal{T}(\Sigma)$ and s is a proper element of $\mathcal{T}(\Sigma)$ such that*

$$s \sqsubseteq t$$

then s is separable from t, and for all proper $s' \in \mathcal{T}(\Sigma)$ such that

$$root(s') = root(s)$$

we have

$$(t \dot{-} s) \parallel s'$$

and

$$(t \dot{-} s) \sqcup s'$$

is a proper element of $\mathcal{T}(\Sigma)$.

Clause (1) of Proposition 4.2 says that as long as two proper elements of a labeled context forest are disjoint, any Brouwerian algebra operation maintains propriety. Clause (2) says that a proper labeled subtree s of a proper subtree t always separates "cleanly" from t, and that replacing s with any proper s' having the same root node also yields a proper subtree. These properties follow easily from the appropriate definitions.

The definition of term makes no formal distinction between term and subterm. However, where elements of $\mathcal{T}(\Sigma(\mathcal{V}_0))$ are concerned, we have to be a bit more careful. We need to be able to "reroot" certain subtrees if we wish to compare them as terms or incorporate them into other terms. For this purpose, we make the following definition:

Definition 4.12 (Rerooting)
Let Σ be a signature. Let s be a proper element of $\mathcal{T}(\Sigma)$ and p be an element of $\mathbb{P}^*$. We then define the rerooting *of s to p, notated*

$$reroot(s, p)$$

as the element of $\mathcal{T}(\Sigma)$ isomorphic to s (in the sense of a Σ-algebra) such that

$$root(reroot(s, p)) = p$$

Thus the rerooting of a subterm s_1 yields a new subterm that has an identical label structure, but a new root equal to p.

We now define a notion of equality for terms:

Definition 4.13 (Term Equality)
Let Σ be a signature, and let s and t be elements of $\mathcal{T}(\Sigma(\mathcal{V}_0))$, i.e., Σ-terms. Then s and t are equal as terms, notated

$$s = t$$

iff

$$\mathrm{re}root(s, root(t)) \equiv t$$

where '$\equiv$' represents Brouwerian algebra equality. In the context of terms, we will use '$\equiv$' for identity.

4.2.2 Operations on Terms

We now define the *replacement* operation on terms, by which a subterm of a given term is replaced by a new subterm:

Definition 4.14 (Replacement)
Let Σ be a signature. Let s, s', and t be proper elements of $\mathcal{T}(\Sigma)$. Then we define the replacement *of s by s' in t by*

$$t[s \leftarrow s'] \triangleq (t \dot{-} s) \sqcup \mathrm{re}root(s', root(s))$$

Note that by clause (2) of Proposition 4.2, $t[s \leftarrow s']$ is a proper element of $\mathcal{T}(\Sigma)$. Definition 4.14 allows us to perform Brouwerian algebra operations on Σ-terms "under the covers," safe in the knowledge that we will always get a proper labeled tree. We will adopt the convention that

$$[s \leftarrow s']$$

is shorthand for

$$s[s \leftarrow s']$$

We note that the replacement operation corresponds exactly to applying a contraction of the form

$$\langle s,\ \mathrm{re}root(s', root(s))\rangle$$

to t. This fact will be exploited in the sequel to yield *term rewriting systems*.

The following operation will also be useful:

Definition 4.15 (Context Completion)
Let Σ be a signature. Let $C[(i_1), (i_2), \ldots, (i_m)]$ be a Σ-context, $\langle\langle t_1, t_2, \ldots, t_n\rangle\rangle$ be a list of Σ-terms. Then for all $k \in 1 \ldots n$, $i_k \in 1 \ldots n$,

$$C[\langle\langle t_1, t_2, \ldots, t_n\rangle\rangle]$$

denotes the term

$$C[\textcircled{i_1} \leftarrow t_{i_1}][\textcircled{i_2} \leftarrow t_{i_2}] \cdots [\textcircled{i_n} \leftarrow t_{i_n}]$$

We will refer to the term above as a completion *of the context* $C[\textcircled{i_1}, \textcircled{i_2}, \ldots, \textcircled{i_n}]$. *We will treat*

$$C[t_i]$$

as shorthand for

$$C[\langle\!\langle t_i \rangle\!\rangle]$$

Definition 4.15 says that each hole in $C[\textcircled{i_1}, \textcircled{i_2}, \ldots, \textcircled{i_n}]$ is replaced with the element of $\langle\!\langle t_1, t_2, \ldots, t_n \rangle\!\rangle$ possessing the same index.

The following little lemma will be useful in the sequel:

Lemma 4.3 *Let* Σ *be a signature. Let*

$$C[\textcircled{i_1}, \textcircled{i_2}, \ldots, \textcircled{i_n}]$$

be a Σ*-context,*

$$\langle\!\langle s_1, s_2, \ldots, s_n \rangle\!\rangle$$

be a list of Σ*-terms, and* t *be the element such that*

$$t \equiv C[\langle\!\langle s_1, s_2, \ldots, s_n \rangle\!\rangle]$$

Then there exists a context forest f *such that*

$$t \equiv f \sqcup s_1 \sqcup s_2 \cdots \sqcup s_n$$

and such that

$$\{f, s_1, s_2, \ldots, s_n\}$$

is a partition of t.

Proof Simply let f be defined by

$$f = C \dot{-} \bigsqcup \{\textcircled{i} \mid \textcircled{i} \sqsubseteq t\}$$

The set

$$\{f, s_1, s_2, \ldots, s_n\}$$

is a partition of t since the subterms s_i are disjoint and f is disjoint from the subterms s_i by definition. $\square$

If we are defining some term using a construction of the form

$$f(t_1, t_2, \ldots, t_n)$$

where the t_i are terms, then we regard the construction as shorthand for

$$C[\langle\!\langle t_1, t_2, \ldots, t_n \rangle\!\rangle]$$

where $C = f(①, ②, \ldots, ⓝ)$.

4.2.3 Term Convention

Using the operations and notation above, we will adopt the convention in the sequel that the set of Σ-terms are treated as sort-respecting elements of $\mathcal{T}(\Sigma(\mathcal{V}_0))$ and vice versa. (Note, however, that while we can treat Σ-terms as elements of $\mathcal{T}(\Sigma(\mathcal{V}_0))$, various *manipulations* of terms will in general yield elements of $\mathcal{F}(\Sigma(\mathcal{V}_0)))$.

We will also treat elements of $\mathcal{V}$ and $\mathcal{V}_0$ as elements of $\mathcal{T}(\mathcal{V})$ and $\mathcal{T}(\mathcal{V}_0)$, respectively.

4.3 Substitution, Unification, and Matching

Substitutions are morphisms on terms that effectively cause a subset of the variables in a given term to be replaced by other terms.

Definition 4.16 (Substitution)
Given a set of variables $\mathcal{V}$, and a signature Σ, a substitution σ *is a mapping from $\mathcal{V}$ to* $\mathrm{Ter}(\Sigma)$, *where $\sigma(x) = x$ for all but a finite number of variables $x \in \mathcal{V}$, and for all $x \in \mathcal{V}$,*

$$root(\sigma x) \quad = \quad root(x)$$

σ is then extended to a morphism on free Σ-algebras by requiring that for all

$$f \in \Sigma_{\langle \langle\!\langle s_1, s_2, \ldots, s_n \rangle\!\rangle,\, s \rangle}$$

and elements

$$t_1 \in \mathrm{Ter}_{s_1}(\Sigma), t_2 \in \mathrm{Ter}_{s_2}(\Sigma), \ldots, t_3 \in \mathrm{Ter}_{s_n}(\Sigma)$$

the following holds:

$$\sigma(\, f(t_1, t_2, \ldots, t_n)\,) \;\equiv\; f(\,\sigma(t_1), \sigma(t_2), \ldots, \sigma(t_n)\,)$$

Note that for any substitution σ and term t, $\sigma(T)$ can be obtained by a finite sequence of replacements of occurrences of variables x by terms $\sigma(x)$.

The set of terms which are substituted for every instance of a variable x in a term by some substitution are called *x-substituends*:

Definition 4.17 (Substituend)
*Let Σ be a signature, t be a Σ-term, $x \in \mathcal{V}_0$ be a variable, and σ be a substitution. Then the set of x-*substituends *induced by σ on t at x, notated $\sigma_x(t)$ is the set defined as follows:*

$$\sigma_x(t) \triangleq \{\sigma(t_x) \mid t_x \sqsubseteq t \text{ and } t_x = x\}$$

It should be clear that if s_1 and s_2 are elements of $\sigma_x(t)$, then $s_1 = s_2$ (but it is *not* necessarily the case that $s_1 \equiv s_2$). Note that without treating Σ-terms as elements of $\mathcal{T}(\Sigma(\mathcal{V}_0))$, the notion of substituend would be difficult to define.

Terms that are substitution instances of one another are said to be *unifiable*, a notion defined formally as follows:

Definition 4.18 (Unifiability)
Let s and t be terms of Ter(Σ). *Then s and t are* unifiable *iff there exist substitutions σ, τ such that*

$$\sigma(s) \equiv \tau(t)$$

*If s is an element of G*Ter(Σ) *and s and t are unifiable, then we say that s* matches *t.*

If s and t are unifiable, then there exists a *most general* instance $s \nabla t$ of s and t such that

$$s \nabla t = \sigma(s) = \tau(t)$$

(see [Hue80, Thm. 3.1] for a proof).

We conclude this section with some definitions applicable to reduction relations defined on terms:

Definition 4.19 (Stability, Compatibility)
Let Σ be a signature. A reduction relation $\longrightarrow_R$ on Ter(Σ) *is*

(1) stable *if for any substitution σ,*

$$M \longrightarrow_R N \text{ implies } \sigma(M) \longrightarrow_R \sigma(N)$$

(2) compatible *if for any context C,*

$$M \longrightarrow_R N \text{ implies } C[M] \longrightarrow_R C[N]$$

A stable relation is oblivious to substitutions, a compatible one is oblivious to the surroundings in which some reduction is performed.

4.4 Term Rewriting Systems

We now turn to term rewriting systems, informally introduced in Section 2.1.3. Here we will give the requisite precise definitions.

Definition 4.20 (Term Rewriting System)
Given a signature Σ, a Σ-term rewriting system is a finite set R of pairs of elements of $\mathcal{T}_0(\Sigma(\mathcal{V}_0))$ of the form

$$\langle T_l,\ T_r\rangle$$

called rewriting rules *such that $V(T_r) \subseteq V(T_l)$ and $T_r \in \mathrm{Ter}_s(\Sigma)$ implies that $T_l \in \mathrm{Ter}_s(\Sigma)$.*

We require that T_l and T_r be elements of $\mathcal{T}_0(\Sigma(\mathcal{V}_0))$ to ensure that they have the same root. The pairs of terms comprising a term rewriting system are more often written in the form $T_l \longrightarrow T_r$, to indicate that terms of the form T_l *rewrite* to terms of the form T_r. We will often associate a unique name with each pair of a term rewriting system, where the pair $\langle T_l,\ T_r\rangle$ with associated name N is denoted by

$$(N)\ \ T_l \ \ \longrightarrow_R \ \ T_r$$

(note that the name is always parenthesized, to indicate its association with a *specific* pair).

Certain rewriting rules serve simply to extract a subterm from a term of the appropriate form:

Definition 4.21 (Projection Rule)
A rewriting rule

$$T_l \longrightarrow T_r$$

is a projection rule *iff T_r is a variable.*

As discussed in Section 2.1.3, term rewriting systems are usually derived from sets of *equations* (e.g., those for groups used in Section 2.1.3), which are then directed from left to right to yield term rewriting rules. We will generally not be concerned with the properties of the equational theory (which is covered, e.g., in [O'D77]), concentrating instead on the computational properties of the resulting rewriting system.

From a term rewriting system R, we get a corresponding reduction relation $\longrightarrow_R$, defined as follows:

Definition 4.22 (Rewriting Relation)
Let R be a Σ-term rewriting system. Then the rewriting relation $\longrightarrow_R$ *induced by R is the smallest stable and compatible reduction relation on* $\mathrm{Ter}(\Sigma)$ *such that if R contains a rule*

$$(N)\ \ T_l \ \ \longrightarrow \ \ T_r$$

then

$$T_l \longrightarrow_R T_r$$

We make the following auxiliary definitions:

(1) *If*

$$(N) \quad T_l \quad \longrightarrow_R \quad T_r$$

is a rewriting rule of system R, then any substitution instance of T_l is a redex. *If we wish to be more specific about the origin of the redex, we will refer to it as an (N)-redex.*

(2) *If P is a redex such that $\sigma(T_l) = P$, then the term*

$$P' \;\equiv\; [P \leftarrow \sigma(T_r)]$$

is called the contractum *of P, and we say that P' results from* contraction *of redex P.*

(3) *If P and M are terms such that $P \sqsubseteq M$, and $\sigma(T_l)$ is a redex such that $P = \sigma(T_l)$, then the term*

$$M' \equiv M[P \leftarrow \sigma(T_r)]$$

is a reduct *of M (resulting from contraction of P).*

(4) *The subrelation of $\longrightarrow_R$ generated by the single rule (N) will be denoted $\longrightarrow_{(N)}$.*

Note that since rewriting rules are defined over elements of $\mathcal{T}_0(\Sigma(\mathcal{V}_0))$, Definition 4.22 makes sense. Since a term rewriting relation $\longrightarrow_R$ is stable and compatible, we know that if M' is a reduct of M,

$$M \longrightarrow_R M'$$

Note that the definitions of redex and contractum in Definition 4.22 are somewhat more restrictive than the corresponding notion for abstract replacement systems.

Given the definition of Rewriting Relation, it is easy to show that in conjunction with the Brouwerian algebra of terms, we get an abstract replacement system:

Theorem 4.4 *Let Σ be a signature. Let R be a Σ-term rewriting system. Then*

$$\langle \mathcal{F}(\Sigma),\; R\rangle$$

is an abstract replacement system.

Proof We need only show that the rewriting relation is compatible with the algebraic structure of terms. This follows immediately from the compatibility requirement of Definition 4.22. □

4.4.1 Superposition Theorem

Knuth and Bendix's seminal paper [KB70] describes a relatively simple procedure for determining if the rewriting relation $\longrightarrow_R$ of a term rewriting system is weakly confluent. It reduces the question of weak confluence to determining whether certain terms, called *critical pairs*, have a common reduct. Critical pairs are defined as follows:

Definition 4.23 (Critical Pair)
Let R be a term rewriting system. Let

$$\begin{array}{lccc} (N_1) & S_l & \longrightarrow & S_r \\ (N_2) & T_l & \longrightarrow & T_r \end{array}$$

be two rewriting rules of R. Let $Q \sqsubseteq S_l$ be a non-variable term such that Q and T_l are unifiable. Let $Q \nabla T_l$ be the most general common instance of Q and T_l, i.e., such that there exist substitutions σ and τ where

$$Q \nabla T_l \;=\; \sigma(Q) \;=\; \tau(T_l)$$

$Q \nabla T_l$ is said to result from the superposition *of S_l and T_l. Then the pair*

$$\Phi \equiv \langle \sigma(S_l[Q \leftarrow \tau(T_r)]),\ [Q \leftarrow \sigma(S_r)] \rangle$$

is a critical pair *for R. Rules (N_1) and (N_2) are said to* have *critical pair Φ.*

Critical pairs represent parts of different rules which "overlap" in a non-trivial way. The following example from [Hue80] illustrates the idea:

Example 4.2 Consider the term rewriting system consisting of the following rules:

$$\begin{array}{lllc} (R_1): & f(x,\, g(x,a)) & \longrightarrow & h(x) \\ (R_2): & g(b,x) & \longrightarrow & k(x) \end{array}$$

The superposition of U of rules (R_1) and (R_2) is given by

$$U \equiv f(b,\, g(b,a))$$

from which we get the critical pair

$$\Phi \equiv \langle f(b,\, k(a)),\ h(b) \rangle$$

Note that term U has both an (R_1) and an (R_2) redex. The critical pair Φ is the pair of reducts of U resulting from application of rules (R_1) and (R_2), respectively. Note in general that two rules may have more than one critical pair, and arbitrary rewriting systems may have many (although there can only be a finite number).

We now have the following theorem of Knuth and Bendix:

Proposition 4.5 (Knuth-Bendix) *Let R be a term rewriting system. Then the rewriting relation $\longrightarrow_R$ is weakly confluent iff, for every critical pair $\langle P_1, P_2\rangle$ of R,*

$$P_1 \downarrow P_2$$

Proposition 4.5 is only sufficient to prove that a rewriting relation is weakly confluent. To prove confluence, other means are required, for instance, showing that $\longrightarrow_R$ is noetherian (which implies confluence by Theorem 3.3).

4.4.2 Regular Term Rewriting Systems

We now define the particularly "well-behaved" class of *regular* term rewriting systems, which are the inspiration for our more general notion of RRS's:

Definition 4.24 (Regular Term Rewriting System)
A Σ-term rewriting system R is regular *iff the following hold:*

(1) *R has no critical pairs.*

(2) *R is* left-linear, *i.e., for every rule $T_l \longrightarrow T_r$ in R, no variable appears in T_l more than once.*

The term "regular" is from Klop [Klo80].

Definition 4.25 (Strongly Regular Rewriting Systems)
A Σ-term rewriting system R is strongly regular *iff it is regular and has no projection rules.*

For regular term rewriting systems, we have the following very important theorem (proved, e.g., in [Hue80]):

Theorem 4.6 (Huet) *All regular term rewriting systems are confluent.*

4.4.3 Conditional Term Rewriting Systems

It will often be the case that we wish to restrict a Σ-term rewriting system R such that the rules of R apply only to members of $\mathrm{Ter}(\Sigma)$ meeting certain criteria. We will call such systems *conditional term rewriting systems*, and define them as follows:

Definition 4.26 (Conditional Term Rewriting System)
A conditional Σ-term rewriting system *is a finite set R of triples of the form*

$$\langle T_l,\ T_r,\ P(x_1, x_2, \ldots, x_n)\rangle$$

called conditional rewriting rules, *such that $\langle T_l, T_r\rangle$ is a rewriting rule, for all x_i in $P(x_1, x_2, \ldots, x_n)$, $x_i \in V(T_l)$, and $P(x_1, x_2, \ldots, x_n)$ is an arbitrary recursive predicate on Σ-terms. We will refer to $P(x_1, x_2, \ldots, x_n)$ as the* guard *for (unguarded) rule N.*

The intention of Definition 4.26 is that the guard *restrict* the set of terms to which the (unguarded) rule applies. As with unconditional term rewriting systems, we will often associate a name with each rule of a conditional term rewriting system. We will usually write a conditional rewriting rule $\langle T_l,\ T_r,\ P(x_1, x_2, \ldots, x_n)\rangle$ (with associated name N) as:

$$(N) \qquad \frac{P(x_1, x_2, \ldots, x_n)}{T_l \quad \longrightarrow_R \quad T_r}$$

We define the notion of redex, contractum, and reduct in exactly the same manner as for unconditional systems. Note that the definition above requires that the predicate $P(x_1, x_2, \ldots, x_n)$ associated with each rule only be concerned with subterms of redexes, thus, e.g. it cannot depend on the context containing the redex.

We define the reduction relation $\longrightarrow_R$ corresponding to a conditional term rewriting system R as follows:

Definition 4.27 (Conditional Rewriting Relation)
Let R be a conditional Σ-term rewriting system. Then the rewriting relation $\longrightarrow_R$ *induced by R is the smallest compatible reduction relation on* $\mathrm{Ter}(\Sigma)$ *such that if there exists a rule*

$$\frac{P(x_1, x_2, \ldots, x_n)}{T_l \quad \longrightarrow_R \quad T_r}$$

and substitution σ such that $S = \sigma(T_l)$, $S' = \sigma(T_r)$, and

$$P(\sigma(x_1), \sigma(x_2), \ldots, \sigma(x_n))$$

holds, then

$$S \longrightarrow_R S'$$

We can easily define the analogue of critical pairs in unconditional term rewriting systems for conditional term rewriting systems:

Definition 4.28 (Conditional Critical Pair)
Let R be a conditional Σ-term rewriting system. Let

$$(N_S) \qquad \frac{P_S(x_1, x_2, \ldots, x_m)}{S_l \quad \longrightarrow_R \quad S_r}$$

and

$$(N_T) \qquad \frac{P_T(y_1, y_2, \ldots, y_n)}{T_l \quad \longrightarrow_R \quad T_r}$$

be rewriting rules of R. Then (N_S) and (N_T) have conditional critical pair Φ *iff there exists a substitution σ such that*

$$P_S(\sigma(x_1), \sigma(x_2), \ldots, \sigma(x_m))$$

and

$$P_T(\sigma(y_1), \sigma(y_2), \ldots, \sigma(y_n))$$

hold, and $\sigma(S_l)$ and $\sigma(T_l)$ have unconditional critical pair Φ.

The gist of the definition above is that two conditional rules have a critical pair only if some substitution causes both of their guards to be satisfied, and the resulting substitution applied to both rules' left hand sides forms a critical pair. One can often transform unconditional term rewriting systems with critical pairs into conditional term rewriting systems without critical pairs by judicious choice of guards that eliminate term instances for which two rules could overlap.

4.4.4 Residuals in Term Rewriting Systems

We are now in a position to show that (conditional or unconditional) regular term rewriting systems form regular replacement systems. To do this, we must define a residual map. Our task is made easier by noting that every redex in a (conditional or unconditional) regular term rewriting system has a unique contractum. Thus we can define residuals directly on redexes, rather than redex/contractum pairs. We can also build up reductions from lists of redexes, rather than lists of contractions. We proceed as follows:

Definition 4.29 (Term Rewriting Residual)
Let R be a (conditional or unconditional) regular Σ-term rewriting system, and M, A_1, and A_2 be elements of $\mathrm{Ter}(\Sigma)$ *such that $A_1 \sqsubseteq M$, $A_2 \sqsubseteq M$, and A_1 and A_2 be (N_1) and (N_2) redexes respectively, where $A_1 \longrightarrow_{(N_1)} A_1'$,*

$$(N_1) \;\; T_l \;\; \longrightarrow \;\; T_r$$

and $A_1 = \sigma(T_l)$ for some substitutions σ. Then we define the set of residuals *of A_2 by A_1, notated A_2/A_1, as follows:*

$$A_2/A_1 \;=\; \begin{cases} \emptyset & A_1 \equiv A_2 \\ (A_2 \dot{-} A_1) \sqcup A_1' & A_1 \sqsubset A_2 \\ \sigma_x([A_1' \leftarrow T_r]) & A_2 \sqsubset A_1,\; x \sqsubseteq [A_1 \leftarrow T_l], \text{ and } root(x) = root(A_2) \end{cases}$$

Note that since R is regular, the three cases in Definition 4.29 are all the situations that can occur. The last two cases in the definition are actually quite simple: If $A_1 \sqsubset A_2$, the residual of redex A_2 is just what remains of it after replacing A_1 by A_1'. In the last case, the residuals are those isomorphic instances of A_2 which are substituted in the instance of the right hand side of A_1.

4.4.5 Regular Conditional Term Rewriting System

Given Definition 4.29, we can now define the notion of regular *conditional* rewriting system analogously to its unconditional counterpart:

Definition 4.30 (Regular Conditional Term Rewriting System)

A conditional term rewriting system R is regular *iff the following hold:*

(1) *R has no conditional critical pairs.*

(2) *R is* left-linear.

(3) *If A is an (N_1)-redex of $A \sqcup B$ and B is an (N_2) redex of $A \sqcup B$, where (N_1) and (N_2) are of the form*

$$(N_1) \qquad \frac{P_1(x_1, x_2, \ldots, x_{n_1})}{T_{l1} \quad \longrightarrow_R \quad T_{r1}}$$

$$(N_2) \qquad \frac{P_2(y_1, y_2, \ldots, y_{n_2})}{T_{l2} \quad \longrightarrow_R \quad T_{r2}}$$

and A/B and B/A are the residuals of A and B with respect to the unguarded *rules*

$$\begin{array}{lll} T_{l1} & \longrightarrow_R & T_{r1} \\ T_{l2} & \longrightarrow_R & T_{r2} \end{array}$$

then

$$P_1(A/B) \text{ and } P_2(B/A)$$

both hold.

Requirement (3) above simply ensures that the residuals of the unguarded rewriting rules remain redexes in the conditional case as well. Put more succinctly, we want unconditional residuals to preserve validity of guard predicates.

We then have:

Theorem 4.7 *All regular conditional term rewriting systems are confluent.*

Given the definition of conditional critical pair, the proof of Theorem 4.7 is a straightforward extension of the standard proof of Theorem 4.6. See [Mar91] for full details.

4.4.6 Parallel Reduction

We now extend the notions of redex and residual to the parallel case in the obvious way:

Definition 4.31 (Parallel Redex for Term Rewriting)
Let R be a regular Σ-term rewriting system. Then $\mathcal{P} \subseteq$ Ter(Σ) *is a* parallel R-redex *iff* $\|(P)$ *and for all $A \in \mathcal{P}$, A is a redex.*

Definition 4.32 (Parallel Residual for Term Rewriting)
Let R be a regular Σ-term rewriting system, and $\mathcal{P}$ and $\mathcal{Q}$ be parallel R-redexes. Then the residual of $\mathcal{P}$ by $\mathcal{Q}$, $\mathcal{P}/\mathcal{Q}$, is defined as follows:

$$\mathcal{P}/\mathcal{Q} \;=\; \begin{cases} \bigcup\{\mathcal{A}/\mathcal{Q} \mid \mathcal{A} \in \mathcal{P}\} & \mathcal{Q} = \{B\} \\ (\mathcal{P}/\mathcal{A})\,/\,(\mathcal{Q} - \{\mathcal{A}\}) & \text{otherwise} \end{cases}$$

We note that the set elements of residual $\mathcal{P}/\mathcal{Q}$ is always disjoint.

The following simple example illustrates the notion of residual map for term rewriting systems:

Example 4.3 Let R be the rewriting system defined by the rules

$$\begin{array}{rcl} D(x) & \longrightarrow & P(x,x) \\ I(x) & \longrightarrow & x \end{array}$$

R is clearly regular.

The term

$$T \equiv D(I(a))$$

then has two redexes: $D(I(a))$ and $I(a)$. The reductions of T possible are illustrated as follows:

$$\begin{tikzcd} & D(I(a)) \arrow[dl] \arrow[dr] & \\ D(a) \arrow[dr] & & P(I(a),I(a)) \arrow[dl] \\ & P(a,a) & \end{tikzcd}$$

The residual of $D(I(a))$ in $D(a)$ is $D(a)$. The residuals of $I(a)$ in $P(I(a),I(a))$ are the two occurrences of $I(a)$ therein. Note that the reduction from $P(I(a),I(a))$ to $P(a,a)$ contracts a parallel redex; thus the diagram above illustrates the fact that regular term rewriting systems possess the parallel moves property.

Both the parallel and non-parallel notion of residual for term rewriting systems have been extensively studied, e.g., in [Ros73,O'D77,HL79,Klo80]. and we refer to these sources for various proofs of the following proposition:

Proposition 4.8 *Let R be a regular (conditional or unconditional) Σ-term rewriting system, and let '$/$' be the residual map for R given by Definition 4.32. Then $\langle \mathrm{Ter}(\Sigma),\ \longrightarrow_R,\ / \rangle$ is an* RRS.

Given Theorem 4.4, to show that regular rewriting systems do indeed form RRS's, we need only show that the residual map defined above agrees with the axioms of Definition 3.65 and that the map is stable. For the former, we refer to any of [Ros73,O'D77,HL79,Klo80]. For the latter, we note that by regularity, no redex can be created by two different sets of coinitial redexes.

4.4.7 Redex Location

We will use some standard terminology for singling out a particular redex in a term, as follows:

Definition 4.33 (Outermost Redex)
Let Σ be a signature, R be a Σ term rewriting system, M be an element of Ter(Σ), *and A be an R-redex of M. Then A is* outermost *iff there exists no redex $B \neq A$ in M such that $A \sqsubseteq B$.*

Note that there may be more than one outermost redex in any term, and that if $A \not\parallel B$, then we must have either $A \sqsubseteq B$ or $B \sqsubseteq A$, thus the set of outermost redexes of M is disjoint.

The following definition will also be useful:

Definition 4.34 (Leftmost Redex)
Let Σ be a signature, R be a Σ term rewriting system, M be an element of Ter(Σ), *and $\mathcal{S}$ be a set of R-redexes of M such that $\|(\mathcal{S})$. Then $A_1 \in \mathcal{S}$ is* to the left of *$A_2 \in \mathcal{S}$ iff there exists $s, t, u \in \mathbb{P}^*$ and $i \in \mathbb{P}$ such that*

$$\begin{aligned} addr(root(A_1)) &= s \circ i \circ t \\ addr(root(A_2)) &= s \circ j \circ u \end{aligned}$$

and $j > i$.

$A \in \mathcal{S}$ is leftmost *iff it is to the left of all $B \in \mathcal{S}$ such that $B \neq A$.*

From the definitions above, we see that the *leftmost outermost* redex of any term M is uniquely defined.

4.5 The Lambda Calculus

Having introduced the lambda calculus informally in section 2.1.1, we now provide the formal details here. Our definitions and notation will usually, although not invariably, follow the conventions of Barendregt [Bar84].

4.5.1 Terms

We begin by defining the term structure of the lambda calculus:

Definition 4.35 (Lambda Terms)
The set of terms in the lambda calculus, designated $\mathrm{Ter}(\lambda)$*, is defined inductively as follows:*

$$\begin{array}{rcl} x \in \mathrm{Ter}(\lambda) & & \\ M, N \in \mathrm{Ter}(\lambda) & \Longrightarrow & (MN) \in \mathrm{Ter}(\lambda) \\ M \in \mathrm{Ter}(\lambda) & \Longrightarrow & (\lambda x.M) \in \mathrm{Ter}(\lambda) \end{array}$$

where x *is an arbitrary variable.*

(MN) is an *application* term; $(\lambda x.M)$ is an *abstraction* term whose *body* is M, whose *bound variable* is x, and whose *abstractor* is λx; $((\lambda x.M)N)$ is a *redex* in which $(\lambda x.M)$ is the *function part* and N is its *argument*. By convention,

$$\lambda xyz.(MNPQ)$$

denotes

$$(\lambda x.(\lambda y.(\lambda z.(((MN)P)Q))))$$

(i.e., application associates to the left, outermost parentheses may be omitted, and nested abstractions can be coalesced).

The syntax used for the operation "symbols" of the lambda calculus differs from the prefix operator and parenthesized operand convention used in term rewriting systems—application is represented by juxtaposition, and abstraction using the '.' notation. This is a mere typographic convention, and causes no formal difficulties when constructing λ-terms. If, however, we wish to use labeled trees to represent λ-terms, it will sometimes be more convenient to use the prefix operator '@' to represent the operation of application, in which case, we could write the λ-term

$$\lambda x.xy$$

as

$$\lambda(x, @(x, y))$$

4.5.2 Variables and Substitution

Since the two operation symbols of the lambda calculus are binary, there are no ground λ-terms. There is, however, a related notion of *closed term*, for which we need the following definitions:

Definition 4.36 (Free Variable, Bound Variable)

(1) *Given a* λ*-term* M*, an occurrence of a variable* x *in* M *is* free *iff there is no occurrence in* M *of a subterm of the form*

$$N \equiv \lambda x.P$$

(2) *An occurrence of a variable is* bound *iff it is not free.*

(3) *x is a member of the set of* free variables *of M, notated $x \in FV(M)$ iff there is a free occurrence of x in M.*

(4) *x is a member of the set of* bound variables *of M, notated $x \in BV(M)$ iff there is a bound occurrence of x in M.*

(5) *M is* closed, *(or a* combinator*) iff $BV(M) = \emptyset$.*

Note that we distinguish above between free *occurrences* of a variable, and the *set* of variables for which free occurrences exist. Thus $FV(M)$ and $BV(M)$ need not be disjoint.

Since variables are an inevitable part of the structure of λ-terms, we must be careful to avoid making distinctions between terms solely on the basis of the names chosen for different variables. We thus make the following definition:

Definition 4.37 (Change of Bound Variable)
Let M and M' be λ-terms such that there is an occurrence of a subterm $N \equiv \lambda x.P$ in M, and let

$$\{x_1, x_2, \ldots, x_n\}$$

be the set of all occurrences of the variable x in N. Finally, let y be a variable with no *occurrences in N. Then M' is obtained from M by a* change of bound variable *from x to y if M' is of the form*

$$M[\, N \leftarrow N[x_1 \leftarrow y][x_2 \leftarrow y] \cdots [x_n \leftarrow y] \,]$$

By changing a bound variable, one produces a new term which is "identical," except for the set of variables chosen. Extending this to multiple bound variable changes yields the notion of *α-congruence*:

Definition 4.38 (α-Congruence)
M and M' are α-congruent, *notated*

$$M \equiv_\alpha M'$$

iff M' is obtained from M by one or more changes of bound variable

By the definition above, we have, e.g.,

Example 4.4

$$\lambda x.xy \;\; \equiv_\alpha \;\; \lambda z.zy \;\; \not\equiv_\alpha \lambda y.yy$$

Rather than fussing over trivial distinctions among terms caused by differing variable names, we will adopt the following convention of Barendregt [Bar84]:

Definition 4.39 (Variable Convention)

(1) *We will identify all terms that are α-congruent, using '$\equiv$' to denote '$\equiv_\alpha$'.*

(2) *Whenever meta-variables $M_1, M_2, \ldots, M_n$ representing λ-terms are used together in the same context, we will assume that*

$$\bigcup_{i \in 1\ldots n} FV(M_i) \;\cap\; \bigcup_{i \in 1\ldots n} BV(M_i) \;=\; \emptyset$$

i.e., the set of bound variables in all the terms is distinct from the set of free variables in all the terms.

Note that convention (2) can always be made to hold by appropriate changes of bound variable.

While the variable convention will suffice to perform formal manipulation of terms on paper, in practice, some mechanism for performing renaming of bound variables is required if we wish to carry out actual *computations*, since we must ensure that the convention is obeyed at each step of the computation. In Section 4.6, we will introduce a variant of the lambda calculus in which problems with variable naming do not occur, and which is thus somewhat better suited for performing practical computation.

4.5.3 Lambda Substitution

Although we dispensed with trivial distinctions caused by variable names above, the distinction between free and bound variables still exists, playing a role in a refinement of replacement which we will refer to as *lambda substitution*:

Definition 4.40 (Lambda Substitution)
Let M and N be λ-terms. Then the substitution *of N for free occurrences of x in M, notation*

$$M[x := N]$$

is defined inductively as follows:

$$\begin{array}{rcll}
x[x := N] & \equiv & N & \\
y[x := N] & \equiv & y & (x \not\equiv y) \\
(\lambda y.M_1)[x := N] & \equiv & \lambda y.(M_1[x := N]) & \\
(M_1 M_2)[x := N] & \equiv & (M_1[x := N])(M_2[x := N]) &
\end{array}$$

The fact that lambda substitution takes into account the binding structure of terms makes it quite different from the "naive" notion of substitution as a simple morphism on terms given by Definition 4.16, in which *all* occurrences, free or otherwise, of a variable in some term are replaced. Thus given a context

$$C \equiv \lambda xy.\odot$$

we have

$$C[(yz)] \equiv \lambda xy.yz$$

But given

$$M \equiv \lambda xy.v$$

we have instead

$$M[v := (yz)] \equiv \lambda xy.wz$$

where we must rename y to w (or any other variable not bound in M) in order to obey the variable convention and perform the substitution.

4.5.4 Beta Contraction

The fundamental rewriting operation for λ-terms is *β-contraction*, defined as follows:

Definition 4.41 (β-Contraction)
Let M be a λ-term of the form

$$M \equiv (\lambda x.M)N$$

*Then M' results from β-*contraction *of M if M' has the form*

$$M' \equiv M[x := N]$$

M is called a β-redex, and M' its contractum.

As with term rewriting systems, the notion of β-redex is somewhat less general than the notion of redex in an RRS (since a term is said to be a redex only if its outermost set of constructors comprise a redex, not if it merely *contains* some redex). Since a β-redex uniquely identifies its corresponding contractum, we can use the redex as a shorthand for the corresponding contraction.

We then define the corresponding reduction relation $\longrightarrow_\beta$ as follows:

Definition 4.42 (Beta Reduction)
The reduction relation $\longrightarrow_\beta$ is the smallest stable and compatible relation on Ter(λ) *such that if M' results from β-contraction of M, then*

$$M \longrightarrow_\beta M'$$

M' is the reduct *of M.*

4.5.5 Reduction

Most of the notation we will use for λ-reduction is identical to that used in our study of abstract replacement systems. Thus

- If R is a redex of λ-term M, then

$$M \xrightarrow{R}_{\beta} M'$$

represents the single-step contraction of R yielding reduct M'.

- A β-reduction sequence ρ is a list of redexes $\langle\langle R_1, R_2, \ldots, R_n \rangle\rangle$ represented as

$$R_1 R_2 \cdots R_n$$

or, when we want to make the initial and final terms explicit, as

$$\rho: M \xrightarrow{R_1}_{\beta} M_1 \xrightarrow{R_2}_{\beta} \cdots \xrightarrow{R_n}_{\beta} M'$$

or just

$$M \xrightarrow{\rho}_{\beta} M'$$

The set of all β-reduction sequences is denoted by β^*.

There are other reduction relations on λ-terms that have been studied. Most prominent is the so called η-rule, which can be used to define a notion of *extensional* equality on λ-terms (treated as functions). Here, however, we will be concerned solely with the more "computational" β-reduction relation.

4.5.6 Normal Forms

As outlined in Chapter 2, we perform β-reduction in order to reduce a term to one of several *normal forms*. Normal forms of interest include the following:

Definition 4.43 (Normal Forms)

(1) *M is a* normal form *(or $M \in NF_\beta$) if and only if it contains no redexes.*

(2) *M is a* head normal form *($M \in HNF_\beta$) if and only if it has the form*

$$\lambda x_1 \cdots x_n.(y\, P_1 \cdots P_m), \qquad n, m \geq 0$$

where y and the x_i are arbitrary variables and the P_j are arbitrary λ-terms.

(3) *M is a* weak head normal form *($M \in WHNF_\beta$) if and only if it has either of the forms*

$$\lambda x.N$$

or

$$(x\, P_1 \cdots P_n), \qquad n \geq 0$$

for arbitrary variable x and arbitrary λ-term N and terms P_i.

A λ-term M *has* a normal form (respectively *head normal form*, *weak head normal form*) if there exists some reduction σ such that

$$M \xrightarrow{\sigma}\!\!\!\!\rightarrow_\beta N$$

and $N \in NF_\beta$ (respectively $N \in HNF_\beta$, $N \in WHNF_\beta$).

We will be primarily interested in the head and weak head normal forms of the lambda calculus, rather than the true normal forms (NF_β). This interest is motivated by the following considerations:

(1) Identifying elements of NF_β produces an inconsistent theory of λ-equality, while consistency is maintained when head normal forms are identified.

(2) Many "useful" λ-terms, such as the well-known Y-combinator used for defining recursive functions, have no normal form, yet have a head normal form.

(3) In the standard models for the untyped lambda calculus, the divergent or *least* semantic element is associated exactly with the set of terms not having a head normal form.

These considerations, among others, motivate Barendregt's [Bar84, p.42] characterization of terms having no head-normal form as "meaningless."

Weak head normal forms are interesting for somewhat more practical reasons: most "lazy" functional programming languages reduce terms to $WHNF_\beta$, rather than HNF_β. This is primarily because in the extensions of the lambda calculus used for programming, the atomic values (i.e., the observable results) are weak head normal forms. Furthermore, since it is undecidable whether two functions are equal, we gain little information of practical value by reducing a term beyond weak head normal form.

A nonstandard semantic investigation of the lambda calculus based on identifying *weak* head normal forms while distinguishing them from head normal forms has been carried out by Abramsky [Abr90]. He argues that in some respects, weak head normal forms are even better behaved from a semantic point of view than head normal forms.

4.5.7 β-Redex Location

If M is not a normal form λ-term, the *leftmost* redex of M is that redex whose abstractor is to the left of the abstractor of every other redex in M.

The leftmost redex R in a non-normal form M is called the *head redex* if its abstractor is only preceded (in the left to right order of symbols) by abstractors of abstraction terms (not redexes). Thus, for example, in $M \equiv \lambda x.(xR)$, where R is a redex, R is the leftmost redex but not the head redex. M has no head redex (and thus is a head normal form). If M's head redex is preceded by no abstraction terms, it will be called the *weak* head redex of M.

4.5.8 Needed Redexes

We have the following useful theorem from Lévy [Lév80]:

Theorem 4.9 ([Lév80, Thm. 6.1]) *The leftmost redex of any λ-term, if it exists, is WHNF$_\beta$-needed, HNF$_\beta$-needed, and NF$_\beta$-needed.*

From this, we can infer the well-known result that the *leftmost* reduction strategy, which always contracts the leftmost redex, is normalizing.

The following is an immediate consequence of a theorem of Barendregt, et al. [BvEG+87a, Thm 4.9]:

Theorem 4.10 ([BvEG+87a]) *The head redex of any λ-term, if it exists, is HNF$_\beta$-needed and WHNF$_\beta$-needed. The weak head redex of any λ-term, if it exists, is WHNF$_\beta$-needed.*

4.5.9 Lambda Residuals

The general notion of residual map for RRS introduced in section 3.5 has its roots in the classical notion of residual in the lambda calculus. As with RRS's, classical λ-residuals induce corresponding notions of permutation, reduction equivalence, and family. Nonetheless, the lambda calculus in its classical form is *not* an RRS, since classical residual map for the lambda calculus does not yield disjoint sets.

We will argue in Section 4.9.5, however, that unlike term rewriting systems, the lambda calculus is *fundamentally* a formalism based on graphs, rather than trees. We will sketch an operational semantics for the lambda calculus based on graph, rather than term rewriting, which yields an RRS. The classical notion of residual then appears when the graphs are mapped back to trees.

Let us now review the definition of λ-residual, which is taken from [Lév80].

Definition 4.44 (Lambda Residual)
Let M be a λ-term, and let R and S be (not necessarily disjoint) redexes of M such that

$$M \xrightarrow{S}_\beta M'$$

Then the set of λ-residuals *of R by S, notated R/S, is defined (informally) by inspection of the relative positions of R and S as follows:*

- *If it is not the case that $R \sqsubseteq S$, then*

$$R/S = \{R'\}$$

  where R' is the redex of M' which is at the same place as R in M.

- *If $R \equiv S$, then*

$$R/S = \emptyset$$

- *If $S = (\lambda x.P)Q$ and $R \sqsubseteq P$, then*

$$R/S = \{R'\}$$

where R' is the redex which corresponds to R in the contractum $P[x := Q]$ of S in M'.

- *Finally, if $S = (\lambda x.P)Q$ and $R \sqsubseteq Q$, then*

$$R/S = \{R_1, R_2, \ldots, R_n\}$$

where n is the number of occurrences of the free variable x in P and every R_i corresponds to R in the i-th instance of Q in the contractum $P[x := Q]$ of S in M'.

Example 4.5 Consider the following λ-term M:

$$M \equiv (\lambda x.((\lambda y.xxy)((\lambda z.P_1)Q_1)))\ ((\lambda z.P_2)Q_2)$$

M contains four redexes: M itself, the term $(\lambda y.xxy)R_1$, and the terms $((\lambda z.P_1)Q_1)$ and $((\lambda z.P_2)Q_2)$. If we contract the redex M, we have the reduction

$$M \ \xrightarrow{M}_\beta\ (\lambda y.((\lambda x.P_2)Q_2)((\lambda x.P_2)Q_2)y)((\lambda x.P_1)Q_1) \equiv M'$$

Given Definition 4.44 we have the following residuals:

- $M/M = \emptyset$
- $((\lambda x.P_1)Q_1)/M$ consists of the single occurrence of $((\lambda x.P_1)Q_1)$ in M'
- $((\lambda x.P_2)Q_2)/M$ consists of the pair of occurrences of $((\lambda x.P_2)Q_2)$ in M'
- $((\lambda y.xxy)R_1)\ /\ M$ consists of M'

As a somewhat simpler alternative to Definition 4.44, we can define residuals by *labeling* the abstraction terms of redexes, and tracing the progress of the terms thus labeled in the course of the reduction. The idea is illustrated by the following example:

Example 4.6 Let M^l be a λ-term of Example 4.5, with each abstraction annotated by a different symbolic *label*.

$$M \equiv (\lambda x.((\lambda y.xxy)^t((\lambda x.P_1)^u Q_1)))^s\ ((\lambda x.P_2)^v Q_2)$$

If we reduce M to M' in a manner that is oblivious to labeling, we get

$$M \ \xrightarrow{M}_\beta\ (\lambda y.((\lambda x.P_2)^v Q_2)((\lambda x.P_2)^v Q_2)y)^t((\lambda x.P_1)^u Q_1) \equiv M'$$

where we note that the residuals of the redexes whose abstraction terms have label α in M are exactly those redex subterms of M' having the same label, if any exists.

The idea of using labels to determine residuals, and, in fact, denote entire redex *families* will explained in detail in Section 4.7.

We can extend Definition 4.44 to arbitrary reduction sequences as follows:

Definition 4.45 (Lambda Reduction Residual)
Let M be a λ-term, ρ be a β-reduction sequence of the form

$$M \stackrel{\rho}{\twoheadrightarrow}_{\beta} M'$$

and let R be a of redex of M. Then the set of λ-residuals *of R by ρ, notated R/ρ, is defined inductively on the length of ρ as follows:*

$$\begin{aligned} R/\epsilon &= \{R\} \\ R/S\rho &= \{U \mid U \in T/S,\ T \in R/\rho\} \end{aligned}$$

We cannot define *parallel* β-reduction as reduction of disjoint subterms, since we do not thereby get a consistent notion of residual commutativity. Instead, we use the notion of *development:*

Definition 4.46 (Relative Reduction, Lambda Development)

(1) *A reduction*

$$\rho = R_1 R_2 \ldots R_n \ldots$$

is relative *to F iff for all $n \geq 1$,*

$$R_n \in F/(R_1 R_2 \ldots R_{n-1})$$

(2) *ρ is a* development *of F iff ρ is relative to F and*

$$F/\rho = \emptyset$$

We then have the following theorem from Curry [CF58]:

Theorem 4.11 (Finite Developments [CF58]) *Let $\mathcal{S}$ be a set of redexes in a λ-term M. Then*

(1) *There is no infinite reduction relative to $\mathcal{S}$.*

(2) *All developments end at the same term.*

(3) *For all redexes R in M, if ρ and σ are two developments of $\mathcal{S}$, then*

$$R/\rho = R/\sigma$$

The theorem above lets us define a consistent notion of "parallel" reduction even when the terms in question are not disjoint.

Definition 4.47 (Parallel Lambda Contraction, Reduction)
Let $\mathcal{S}$ be a set of redexes in a term M. Then M reduces in parallel *to M' (by contraction of $\mathcal{S}$), notated*

$$M \xrightarrow{\mathcal{S}}{}^{\|}_{\beta} M'$$

iff there exists a complete development ρ of $\mathcal{S}$ such that

$$M \xrightarrow{\rho}\!\!\!\rightarrow_{\beta} M'$$

*As usual, the reflexive, transitive closure of the parallel contraction relation yields the parallel β-*reduction *relation, notated $\twoheadrightarrow^{\|}_{\beta}$. The set of all parallel reduction sequences is denoted by $\|(\beta)^*$.*

Given Theorem 4.11, Definition 4.47 is self-consistent. The notions of residual and reduction sequence then extend trivially to *sets* of redexes.

The following are also from [CF58], and are the counterparts of clauses (3) and (4), respectively of the definition of residual map (3.65):

Theorem 4.12 (Lemma of Parallel Moves [CF58]) *Let $\mathcal{S}$ and $\mathcal{T}$ be two sets of redexes in a term M. Define the operation $\vee$ by*

$$\mathcal{A} \vee \mathcal{B} \triangleq \mathcal{A}(\mathcal{B}/\mathcal{A})$$

Then $\mathcal{S} \vee \mathcal{T}$ and $\mathcal{T} \vee \mathcal{S}$ end at the same term.

Theorem 4.13 (Cube Lemma [CF58]) *Let $\mathcal{S}$, $\mathcal{T}$, and $\mathcal{U}$ be sets of redexes in a term M. Then*

$$\mathcal{U}/(\mathcal{S} \vee \mathcal{T}) = \mathcal{U}/(\mathcal{T} \vee \mathcal{S})$$

Given Theorems 4.12 and 4.13, all the reduction theoretic definitions and results of Section 3.6 hold. In particular, the we can define the notions of reduction equivalence and family in exactly the same manner as for RRS's

4.6 The de Bruijn Lambda Calculus

We now consider the de Bruijn lambda calculus [dB72,dB78], a more "practical" variant of the lambda calculus in which variables are replaced by *de Bruijn numbers* denoting their binding depth in the term in which they are contained. This facilitates reduction without concern for variable "capture," which can occur during conventional λ-reduction even when the initial term of a reduction contains no bound variables with the same name. By providing a variable substitution mechanism that appropriately adjusts the de Bruijn numbers of substituted terms, the de Bruijn lambda calculus eliminates the need for α-conversion. We will provide only a very brief review of the de Bruijn lambda calculus, since we will soon define a system that is even "better behaved" than the de Bruijn lambda calculus for the purposes of implementing practical reduction schemes.

The definitions that follow are from [Cur86a]

Definition 4.48 *The set of ground terms in the* de Bruijn lambda calculus, *designated* $GTer(\lambda^{DB})$, *is defined inductively as follows:*

$$\begin{array}{rcl} n \in \mathbb{N} \textit{ (the set of natural numbers)} & \Longrightarrow & n \in GTer(\lambda^{DB}) \\ M, N \in GTer(\lambda^{DB}) & \Longrightarrow & (MN) \in GTer(\lambda^{DB}) \\ M \in GTer(\lambda^{DB}) & \Longrightarrow & \lambda.M \in GTer(\lambda^{DB}) \end{array}$$

The terms $n \in \mathbb{N}$ can be considered an infinite set of constant operations.

Definition 4.49 *For any* $M \in Ter(\lambda)$ *such that* $FV(M) \subseteq \{x_0, \ldots, x_n\}$, *define its de Bruijn translation,* $M_{DB(x_0,\ldots,x_n)} \in GTer(\lambda^{DB})$, *as follows:*

$$\begin{array}{rcl} x_{DB(x_0,\ldots,x_n)} & = & i, \textit{ where } i \textit{ is minimum s.t. } x = x_i \\ (\lambda y.M)_{DB(x_0,\ldots,x_n)} & = & \lambda.M_{DB(y,x_0,\ldots,x_n)} \\ (MN)_{DB(x_0,\ldots,x_n)} & = & M_{DB(x_0,\ldots,x_n)} N_{DB(x_0,\ldots,x_n)} \end{array}$$

(We will usually write M_{DB} rather than $M_{DB(x_0,\ldots,x_n)}$ when the free variable ordering is irrelevant).

The following theorem is from Curien [Cur86a]

Theorem 4.14 ([Cur86a]) *For all* $N, M \in Ter(\lambda)$ *such that* $M \equiv_\alpha N$ *and* $FV(N) = FV(M) \subseteq \{x_0, \ldots, x_n\}$, $N_{DB(x_0,\ldots,x_n)} \equiv M_{DB(x_0,\ldots,x_n)}$, *(where "$\equiv$" here means "identically equal to" and "$\equiv_\alpha$" means "equivalent modulo changes of bound variables").*

Substitution and β-reduction can be suitably redefined on λ^{DB}such that

$$M \twoheadrightarrow_\beta N \iff M_{DB} \twoheadrightarrow_\beta N_{DB}$$

For a concise exposition of the details of β-reduction and the substitution process,, see [Cur86a]

4.7 Lévy's Labeled Lambda Calculus

We consider here a final, rather specialized variant of the lambda calculus: Lévy's *labeled lambda calculus* [Lév75]. Lévy's extension allows λ-terms to be annotated with symbolic strings of characters, or *labels*. Such annotations are a refinement of those introduced informally to "track" residuals in Example 4.6. Most importantly for our purposes, however, it will turn out that identically labeled redexes are members of the same family.

We will use a slightly simplified labeling system proposed by Klop [Klo80]. A concise summary of Lévy's labeled lambda calculus is given in [Bar84, p. 382, Ex. 14.5.5], and a summary of a number of useful properties is given in [BKKS87, Appendix].

First we must define what constitutes a *label:*

Definition 4.50 *The set of* Lévy-labels, *designated* L, *is defined inductively as follows:*

$$\begin{aligned} l \in S &\implies l \in \mathrm{L} \\ w, v \in \mathrm{L} &\implies wv \in \mathrm{L} \\ w \in \mathrm{L} &\implies \underline{w} \in \mathrm{L} \end{aligned}$$

where $S = \{a, b, c, \ldots\}$ *is an infinite set of symbols and* wv *is the concatenation of labels* w *and* v.

An *atomic* label is a label consisting of a single symbol. Note that nested underlinings, e.g. $\underline{a\underline{bc}d}$, may occur.

The set of *labeled* λ-terms consists of the regular λ-terms and terms annotated with labels:

Definition 4.51 *The set of terms in Lévy's* labeled lambda calculus, *designated* $\mathrm{Ter}(\lambda^{\mathrm{L}})$, *is defined as follows:*

$$\begin{aligned} M \in \mathrm{Ter}(\lambda) &\implies M \in \mathrm{Ter}(\lambda^{\mathrm{L}}) \\ M \in \mathrm{Ter}(\lambda^{\mathrm{L}}), w \in \mathrm{L} &\implies (M^w) \in \mathrm{Ter}(\lambda^{\mathrm{L}}) \end{aligned}$$

where w *is an arbitrary variable.*

If M is a meta-variable referring to a labeled term, M^w denotes the concatenation of w to the label of the term to which M refers. We will often refer to terms "with" or "having" label w. A term M *has* label w if M is of the form N^w and N is not of the form P^u for non-null label u. The parentheses surrounding a labeled term will often be omitted for the sake of clarity if no confusion would arise. (If, however, a parenthesized term is *itself* labeled, a formal reduction rule is required to eliminate the parentheses; see below.)

In contexts where a labeled term is expected, unlabeled terms will be treated as having the *null label*, ϵ. We define label concatenation and underlining to behave on the null label as follows:

$$\begin{aligned} \epsilon w &= w \\ w\epsilon &= w \\ \underline{\epsilon} &= \epsilon \end{aligned}$$

The label of the abstraction part of a redex is called the *degree* of the redex. Thus the degree of the (Ix) redex in $((\lambda x.(I^a x^b)^c)^d z^e)^f$ is a (not c).

We now define an extension of β-reduction for labeled terms:

Definition 4.52 *Labeled β-contraction, denoted by $\longrightarrow_{\beta\mathrm{L}}$, is the smallest stable and compatible relation on members of* Ter(λ^{L})*such that*

$$((\lambda x.M)^w N)^v \quad \longrightarrow_{\beta\mathrm{L}} \quad (M^{\underline{w}}[x := N^{\underline{w}}])^v$$

where M and N are arbitrary members of Ter(λ^{L})*.*

Note that with the null label convention, labeled β-contraction is exactly the same as regular β-contraction on unlabeled terms.

Though the labeled β-contraction rule looks a bit formidable, the idea is quite simple: Whenever a redex is contracted, the underlined form of the label of the redex's abstraction (w) is attached both to the body of the abstraction (M) and to all instances of the argument (N) substituted into the body. Any label attached to the application term (v) is left intact. The attachment to a label of an underlined substring, say ($\underline{w}$), is an indication that the term was effectively generated by contraction of a redex having degree w (this assumes, as we always will, that any labeled reduction has an initial term with no underlined labels). One can thus view labels as a sort of genetic code, in the sense that by knowing the labels of the initial term of a reduction (it's "matriarch," in a sense), the lineage of a subsequent term in the reduction may be traced by inspection of the labels.

The formation rules of Ter(λ^{L}) allow multiple labelings of parenthesized terms, which can be created as a result of labeled β-contraction. This requires an auxiliary reduction rule for labels:

Definition 4.53 *The* label simplification reduction, $\longrightarrow_{\mathrm{lab}}$, *is the following smallest stable and compatible relation on members of* Ter(λ^{L})*such that:*

$$(M^w)^v \quad \longrightarrow_{\mathrm{lab}} \quad M^{wv}$$

where M^w is a term of Ter(λ^{L})*.*

We then have:

Definition 4.54 Labeled β-reduction, $\twoheadrightarrow_{\beta\mathrm{L}}$, *is the reflexive, transitive closure of* ($\longrightarrow_{\mathrm{lab}} \cup \longrightarrow_{\beta\mathrm{L}}$)*, where '$\cup$' denotes relational union.*

The label simplification rule is a technical necessity, but a practical nuisance. Without loss of generality, when referring to a labeled term, we will assume it has been simplified as much as possible using $\twoheadrightarrow_{\beta\mathrm{L}}$. This assumption is technically justified by the following theorem :

Theorem 4.15 (**[Lév75]**) $\twoheadrightarrow_{\beta\mathrm{L}}$ *is confluent, i.e., if*

$$M \twoheadrightarrow_{\beta\mathrm{L}} N_1 \quad \textit{and} \quad M \twoheadrightarrow_{\beta\mathrm{L}} N_2$$

then there exists P such that

$$N_1 \twoheadrightarrow_{\beta\mathrm{L}} P \quad \textit{and} \quad N_2 \twoheadrightarrow_{\beta\mathrm{L}} P$$

Thus labeled λ-reduction is as "well-behaved" as its unlabeled counterpart, and, in a sense, is a strict refinement of the regular λ-reduction. Ignoring the labels, it is simply regular λ-reduction. Depending on the initial labeling, however, it can give a great deal more information about the reduction process.

We can now define transformations from the unlabeled to the labeled world and vice versa:

Definition 4.55 *Let M^l be a term of* $\mathrm{Ter}(\lambda^{\mathrm{L}})$*. Then the* erasure *of M^l,*

$$Er(M^l)$$

is the same term with all the labels erased.

Definition 4.56 *Let M be a term of* $\mathrm{Ter}(\lambda)$*. Then $M^l \in \mathrm{Ter}(\lambda^{\mathrm{L}})$ is a* labeling *of M if and only if*

$$Er(M^l) \;=\; M$$

We can also define the erasure of a reduction (overloading the meaning of '$Er()$'):

Definition 4.57 *Let σ^l be a labeled reduction. Then the* erasure *of σ^l, $Er(\sigma^l)$, is the unlabeled reduction obtained by erasing the labels of all the terms in the reduction and replacing all labeled β-contractions by unlabeled β-contractions.*

Finally, we can "lift" reductions on unlabeled terms to their labeled counterparts:

Definition 4.58 *Let M be a term of* $\mathrm{Ter}(\lambda)$*, M^l be some labeling of M such that*

$$M \overset{\sigma}{\twoheadrightarrow}_{\beta} N$$

Then the lifted reduction

$$Lift(\sigma, M^l)$$

is defined as the labeled reduction with initial term M^l in which the redexes contracted are the labeled counterparts of those contracted in σ.

4.7.1 Labels, Residuals, and Families

Having invested some effort to understand the rather arcane machinery of the labeled lambda calculus, we are able to reap the benefits: the definition of residual and, more importantly, redex *family* becomes particularly straightforward. We have the following theorems from Lévy [Lév78]:

Theorem 4.16 (**[Lév78]**) *Let ρ be a β-reduction of a λ-term M of the form*

$$M \xrightarrow{\rho}\!\!\!\!\rightarrow_{\beta} N$$

Let R be a redex of M, and let l be a labeling of M such that R^l has degree w. Finally, let

$$\rho^l \colon M^l \twoheadrightarrow_{\beta\mathrm{L}} N^l = \mathit{Lift}(\rho, M^l)$$

be the labeled version of ρ. Then a redex S of N is a residual of R by ρ iff the corresponding redex R^l of N^l has degree w.

Although the connection between labels and residuals may seem self-evident, the following theorem is a much deeper result:

Theorem 4.17 (**[Lév78]**) *Let*

$$\rho \colon M \xrightarrow{R_1}_{\beta} M_1 \xrightarrow{R_2}_{\beta} \cdots \xrightarrow{R_n}_{\beta} M'$$

be a β-reduction of M. Let l be a labeling of M such that each subterm of M^l has a unique *atomic label. Let*

$$\rho^l \colon M^l \twoheadrightarrow_{\beta\mathrm{L}} N^l = \mathit{Lift}(\rho, M^l)$$

be the labeled version of ρ. Then a redex S_j in any intermediate term of ρ (not necessarily a redex contracted by ρ) is a member of the same family *F^l_w if and only if the corresponding redex R^l_j in ρ^l has degree w.*

Thus labeling makes evident on inspection of terms a property that might seem from its definition (3.73) to require enumeration of all reductions.

4.8 Graphs and Trees

Having completed our exposition of the details of the formal systems with which we will be concerned, we turn now to add a new set of structures to our repertoire of implementation tools. The structures we will consider are another class of Brouwerian algebras defined over a signature Σ called *Σ-term graphs*. These are graphs that arise naturally in implementation schemes for term rewriting and the lambda calculus that allow identical subterms to be shared. While the set of

Σ-term graphs contains a class that is isomorphic to the set $\mathcal{T}(\Sigma)$ we used to represent terms, in general, there will only be a *homomorphic* mapping from a term graph to the term it represents. The flexibility afforded by structures whose images are terms, but which are not themselves isomorphic to terms, makes a correct implementation of sharing possible.

Term graphs were first treated extensively by Staples in [Sta80a, Sta80b, Sta80c]. Our discussion, however, will primarily follow that of Barendregt, et al. [BvEG+87a], whose exposition of the issues is somewhat less complex. We will refer to their results for detailed proofs. However, we will make a number of changes to their ideas to adapt them to the framework of regular reduction systems. Related studies of the connections between term rewriting and graph rewriting have also been carried out by Wadsworth (who seems to have invented the idea of graph reduction) [Wad71], Vuillemin [Vui73], and Farmer, et al. [FRW90].

We begin with the definition of term graph. A term graph is a set of *nodes*, each of which corresponds to a labeled node in the abstract tree representing a term. Term graph nodes are defined as follows:

Definition 4.59 (Σ-Term Graph Nodes)
Let Σ be an $\mathcal{S}$-sorted signature and let $\mathcal{I}$ be any set. The set of Σ-nodes $GNode(\Sigma)_{\mathcal{I}}$ *is a sorted set of the form*

$$\{N_{\vec{w},s} \mid \langle \vec{w},\, s\rangle \in \mathcal{S}^* \times \mathcal{S}\}$$

where each element

$$\nu \in N_{\langle\!\langle s_1, s_2, \ldots, s_m\rangle\!\rangle, s}$$

is a triple of the form

$$\langle i_0,\, f,\, \langle\!\langle i_1, i_2, \ldots, i_m\rangle\!\rangle\rangle$$

such that

$$f \in \Sigma_{\langle\!\langle s_1, s_2, \ldots, s_m\rangle\!\rangle, s}$$

- $i_0 \in \mathcal{I}$ *is called the* node identifier *of* ν
- $\langle\!\langle i_1, i_2, \ldots, i_m\rangle\!\rangle \in \mathcal{I}^*$ *is a list of* successor identifiers *(or simply* successors*) of* ν
- $f \in \Sigma$ *is* ν*'s* operation symbol
- ν *is a* node *of* arity $\langle\!\langle s_1, s_2, \ldots, s_m\rangle\!\rangle$, sort s, *and* rank $\langle\langle\!\langle s_1, s_2, \ldots, s_m\rangle\!\rangle,\, s\rangle$

We will generally drop the subscript from $GNode(\Sigma)_{\mathcal{I}}$ denoting the underlying set, when it is clear from context.

Term graphs are simply sets of graph nodes, as follows:

Definition 4.60 (Σ-Term Graph)
Let Σ be a signature and $\mathcal{I}$ be a set. Then any subset $G \in GNode(\Sigma)_{\mathcal{I}}$ is a term

graph. *The set of all term graphs with signature* Σ *and underlying set* $\mathcal{I}$ *is denoted by* $Graph(\Sigma)_{\mathcal{I}}$.

G *is* proper *iff for all nodes* ν *of* G *such that* $id(\nu) = i$, *there exists no* $\pi \neq \nu \in G$ *such that* $id(\pi) = i$.

G *is* sort-respecting *iff* G *is proper, and for all nodes* $\nu, \pi \in G$ *such that*

$$\nu \;\equiv\; \langle i_0,\, f,\, \langle\!\langle i_1, i_2, \ldots, i_m \rangle\!\rangle \rangle$$

and

$$\pi \;\equiv\; \langle i_k,\, g,\, \vec{\jmath} \rangle$$

for some $k \in 1 \ldots m$,

$$f \in \Sigma_{\langle\!\langle s_1, s_2, \ldots, s_m \rangle\!\rangle, s}$$

implies that

$$sort(g) = s_k$$

To a first approximation, our definition of term graph looks very much like a machine implementation of a graph: node identifiers are like cell addresses, operation symbols a sort of tag, and successors may be considered pointers to other cells. This analogy is exact for proper graphs, but breaks down when we consider arbitrary graphs. We will give some examples of term graphs in the next section.

In a proper term graph, there is a one-to-one correspondence between nodes and node identifiers. Improper graphs are not particularly useful for our purposes, but we include them in our definition in order to allow operations on graphs to be specified easily. Unless specifically stated otherwise, all graphs in subsequent discussion are assumed to refer to proper graphs.

We now define some notation for term graphs:

Definition 4.61 (Graph Notation)
Let $Graph(\Sigma)_{\mathcal{I}}$ *be a set of term graphs. Then for any graph* $G \in Graph(\Sigma)_{\mathcal{I}}$, *we will use the following notation:*

(1) *If* $\nu \in G_{\langle\!\langle s_1, s_2, \ldots, s_m \rangle\!\rangle, s}$ *is a node such that*

$$\nu \equiv \langle i_0,\, f,\, \langle\!\langle i_1, i_2, \ldots, i_m \rangle\!\rangle \rangle$$

then we define the following operations on ν:

$$\begin{aligned}
succ(\nu) &\triangleq \langle\!\langle i_1, i_2, \ldots, i_m \rangle\!\rangle \\
succ_k(\nu) &\triangleq i_k \\
id(\nu) &\triangleq i_0 \\
op(\nu) &\triangleq f \\
sort(\nu) &\triangleq s \\
arity(\nu) &\triangleq \langle\!\langle s_1, s_2, \ldots, s_m \rangle\!\rangle \\
size(\nu) &\triangleq m
\end{aligned}$$

(2) *If G is a proper graph and ν is an element of G such that $id(\nu) = i$, then we define* node(i) *by*

$$\mathit{node}(i) \;\triangleq\; \nu$$

(3) *A node ν of G is a* root *if there exists no node $\pi \in G$ such that for some k, $succ_k(\pi) = id(\nu)$.*

(4) *A* path *in G is a (heterogeneous) list of the form*

$$\langle\!\langle \nu_0, n_0, \nu_1, n_1, \ldots, \nu_{m-1}, n_{m-1}, \nu_m \rangle\!\rangle$$

where $m \geq 0$, $\nu_0, \ldots, \nu_m \in G$, $n_0, \ldots, n_{m-1} \in \mathbb{N}$, and

$$id(\nu_k) = succ_{n_{k-1}}(\nu_{k-1})$$

i.e., $id(\nu_k)$ is the n_{k-1}-th successor of ν_{k-1}. This path is said to be from ν_0 *to ν_m, and to have* length m.

(5) *A* cycle *is a path of length greater than 0 from a node ν to itself, in which case ν is a* cyclic *node. A graph is* cyclic *iff it contains a cyclic node, and* acyclic *otherwise.*

(6) *Let ν and π be nodes of G. Then π is* reachable *from G iff there is a path from ν to π.*

(7) *A path in a graph is* rooted *iff it begins with a root of the graph.*

(8) *H is a* subgraph *of G iff $H \subseteq G$.*

(9) *G is* total *iff for all nodes ν, and for all $k \in 1 \ldots \mathsf{size}(\nu)$, there exists a node $\pi \in G$ such that*

$$succ_k(\nu) = id(\pi)$$

G is partial *otherwise.*

(10) *A node ν of G is a* leaf *iff $\mathsf{size}(\nu) = 0$, i.e., its operation symbol is nullary.*

When depicting a proper graph G, we will draw a directed edge from each node ν to each node with an identifier in $succ(\nu)$, provided that such a successor node exists in G. We will label the nodes with their operation symbols, and order outgoing edges from left to right with increasing successor index. We will often omit the node identifier from the depiction of G.

Note that even proper graphs may be partial, i.e., it is not the case that if $\nu \in G$, $succ_k(\nu) = i$ implies that there exists a node $\pi \in G$ of the form $\langle i,\ f,\ \vec{s}\rangle$. In other words, the successor of a node need not exist in the graph (which makes such graphs a bit different from the usual notion of directed graph).

Since a term graph is an unrestricted set of nodes, it forms a trivial Brouwerian algebra:

Proposition 4.18 *Let Σ be a signature and $\mathcal{I}$ be a set. Define the operations '$\sqcup$', '$\sqcap$', and '$\dot{-}$' on elements G and H of $Graph(\Sigma)_\mathcal{I}$ by:*

$$\begin{aligned} G \sqcup H &\triangleq G \cup H \\ G \sqcap H &\triangleq G \cap H \\ G \dot{-} H &\triangleq G - H \end{aligned}$$

Then the structure

$$\langle Graph(\Sigma)_\mathcal{I}, \sqcup, \sqcap, \dot{-}, GNode(\Sigma)_\mathcal{I} \rangle$$

forms a Boolean (thus Brouwerian) algebra.

As usual, we will use $Graph(\Sigma)_\mathcal{I}$ to refer to the Brouwerian algebra of Σ-term graphs.

The following definition will be useful:

Definition 4.62 (Downward Closure)
Let G and H be term graphs such that $H \sqsubseteq G$. Then the downward closure *of H relative to G, notated $DC(H,G)$ is the graph defined inductively as follows:*

(1) *If $\nu \in H$, then $\nu \in DC(H,G)$.*

(2) *If $\nu \in DC(H,G)$, $succ_k(\nu) = j$ for some $k \in 1 \dots size(\nu)$, and there exists some node $\pi \in G$ such that $id(\pi) = j$, then*

$$\pi \in DC(H,G)$$

The downward closure of a H relative to G consists of H and all of its transitive successors in G. We note that if G is a total graph, then $DC(H,G)$ is a total graph.

4.8.1 Forests and Trees

We define here certain interesting subclasses of term graphs. We will be particularly interested in the subclasses of labeled graphs representing partial and total *forests*, defined as follows:

Definition 4.63 (Forest Graph)
Let G be a Σ-term graph and $\mathcal{I}$ be a set. Then $G \in Graph(\Sigma)_\mathcal{I}$ is a forest graph *iff for every root node ρ of G, there is exactly one path from ρ to every node of G reachable from ρ. The set of all partial Σ-forest graphs is denoted by $PForest(\Sigma)$, and the set of all total Σ-forest graphs (or simply "forests") is denoted by $Forest(\Sigma)$.*

We see from the definitions above that a partial forest is a graph in which no nodes are shared, but for which some successors may be missing. A total forest has no shared nodes as well, but also has no missing successors. Note that (partial and total) forests must be acyclic, since otherwise there would be more than one path from the root to any node in the cycle, gotten by going around the cycle a different number of times.

We then have the following definitions:

Definition 4.64 (Tree Graph)
Let Σ be a signature and $\mathcal{I}$ be a set. A graph $G \in Graph(\Sigma)_{\mathcal{I}}$ is a partial tree graph *iff it is a partial forest graph with one root, and a* total tree graph *(or just "tree") iff it is a total forest graph with one root. The set of partial Σ-tree graphs is denoted by* $P\mathrm{Tree}(\Sigma)$*, and total Σ-tree graphs by* $\mathrm{Tree}(\Sigma)$.

We finally define the class of graphs that are intended to represent complete or incomplete single terms:

Definition 4.65 (Rooted Graph)
Let Σ be a signature and $\mathcal{I}$ be a set. A rooted *Σ-graph is a pair*

$$\langle H,\ \rho\rangle$$

where $H \in \mathrm{Graph}(\Sigma)_{\mathcal{I}}$ and ρ is a node of H. ρ is called the designated root *of H, or, when there is no confusion, simply* the *root of H. The set of partial rooted Σ-graphs is denoted by $P\mathrm{Graph1}(\Sigma)$. The set of total rooted Σ-graphs is denoted by $\mathrm{Graph1}(\Sigma)$.*

The designated root of a rooted graph $G \in \mathrm{Graph1}(\Sigma)$ is denoted by $root(G)$. We will adopt the convention that any reference to a rooted graph

$$\langle H,\ \rho\rangle$$

in a context requiring a graph is a reference to H, rather than the pair $\langle H,\ \rho\rangle$.

Note that the *designated* root of a rooted graph need not be an actual root (no predecessors in the graph). Usually, however the designated root will be a root as well.

Example 4.7 Figures 4.1, 4.2, 4.3, and 4.4 depict respectively graphs G_1, G_2, G_3, and G_4. G_1, G_2, and G_3 are all proper, and all total. G_1 and G_2 are intended to represent the same *term*. The former is a tree, the latter is a rooted graph with root $\rho \equiv \langle j_1,\ f,\ \langle\langle j_2, j_3\rangle\rangle\rangle$. G_3 has *two* roots, and is thus a forest. G_4 is a partial tree.

When depicting a graph in textual form, we will be concerned primarily with the operation symbols, not the node identifiers or the identity of particular nodes. Thus, when referring to total trees, we will generally use the same prefix notation as that adopted for terms, e.g., as before,

$$f(g(x, x),\ y)$$

denotes the the tree G_1 of Figure 4.1. If a tree is partial, we will use ϵ to denote any missing successors, e.g.,

$$f(g(x, \epsilon),\ y)$$

denotes the graph G_4 of Figure 4.4.

$$G_1 \equiv \left\{ \begin{array}{c} \langle i_1,\ f,\ \langle\!\langle i_2, i_3 \rangle\!\rangle \rangle \\ \langle i_2,\ g,\ \langle\!\langle i_4, i_5 \rangle\!\rangle \rangle \\ \langle i_3,\ y,\ \langle\!\langle \rangle\!\rangle \rangle \\ \langle i_4,\ x,\ \langle\!\langle \rangle\!\rangle \rangle \\ \langle i_5,\ x,\ \langle\!\langle \rangle\!\rangle \rangle \end{array} \right\}$$

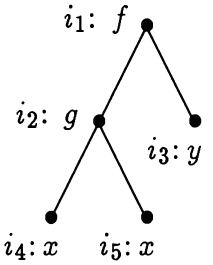


Figure 4.1: Tree Graph G_1

$$G_2 \equiv \left\{ \begin{array}{c} \langle j_1,\ f,\ \langle\!\langle j_2, j_3 \rangle\!\rangle \rangle \\ \langle j_2,\ g,\ \langle\!\langle j_4, j_4 \rangle\!\rangle \rangle \\ \langle j_3,\ y,\ \langle\!\langle \rangle\!\rangle \rangle \\ \langle j_4,\ x,\ \langle\!\langle \rangle\!\rangle \rangle \end{array} \right\}$$

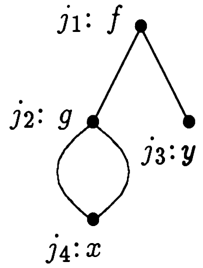


Figure 4.2: Rooted Graph G_2

$$G_3 \equiv \left\{ \begin{array}{c} \langle k_1,\ g,\ \langle\!\langle k_3, k_4 \rangle\!\rangle \rangle \\ \langle k_2,\ y,\ \langle\!\langle\rangle\!\rangle \rangle \\ \langle k_3,\ x,\ \langle\!\langle\rangle\!\rangle \rangle \\ \langle k_4,\ x,\ \langle\!\langle\rangle\!\rangle \rangle \end{array} \right\}$$

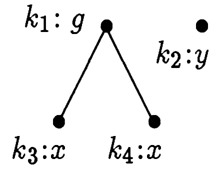


Figure 4.3: Forest Graph G_3

$$G_4 \equiv \left\{ \begin{array}{c} \langle l_1,\ f,\ \langle\!\langle l_2, l_3 \rangle\!\rangle \rangle \\ \langle l_2,\ g,\ \langle\!\langle l_4, l_5 \rangle\!\rangle \rangle \\ \langle l_3,\ y,\ \langle\!\langle\rangle\!\rangle \rangle \\ \langle l_4,\ x,\ \langle\!\langle\rangle\!\rangle \rangle \end{array} \right\}$$

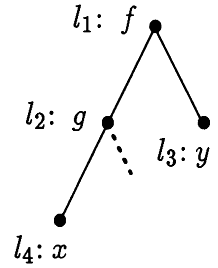


Figure 4.4: Partial Tree Graph G_4

When referring to more general graphs, we will use a hybrid notation, decomposing a graph into pieces which are partial trees. Any separate pieces comprising the graph will be enclosed in set braces. We will prefix the roots of some subtrees with meta-variables representing the node of their root (or its node identifier when we have to be extremely precise). A repeated instance of a meta-variable in an expression, e.g., ν, then refers to the shared subgraph whose root is assumed to be prefixed with that meta-variable. This notation is similar to that used in the graph-rewriting language CLEAN of Barendregt, et al. [BvEG+87b]. Thus

$$\{f(g(\nu,\nu),\ y),\ \ \nu : x\}$$

(or equivalently $\{f(g(j_4,j_4),\ y),\ \ j_4\!: x\}$) represents the graph G_2 of Figure 4.2 and

$$\{g(x,x),\ y\}$$

represents the graph G_3 of Figure 4.3.

4.8.2 Graph Transformations

From the examples above, it should be clear that there are a number of different term graphs that can be used to represent the same term (as well as term graphs that do not in fact represent terms at all). It is thus useful to define mappings between graphs that are intended to represent equivalent terms. We begin with the following definition from [BvEG+87a]:

Definition 4.66 (Term Graph Homomorphism)
*Given two Σ-term graphs G_1 and G_2, a Σ-*term graph homomorphism h *from (nodes of) G_1 to (nodes of) G_2 is a mapping*

$$h\colon G_1 \rightarrow G_2$$

such that for all nodes $\nu \in G_1$,

$$\begin{aligned} op(h(\nu)) &= op(\nu) \\ succ(h(\nu)) &= h^*(succ(\nu)) \end{aligned}$$

where h^* is defined by

$$h^*(\langle\!\langle \nu_1,\nu_2,\ldots,\nu_m \rangle\!\rangle) \;=\; \langle\!\langle h(\nu_1),h(\nu_2),\ldots,h(\nu_m)\rangle\!\rangle$$

A Σ-term graph homomorphism $h\colon G_1 \rightarrow G_2$ is an isomorphism *if there exists an inverse homomorphism*

$$h^{-1}\colon G_2 \rightarrow G_1$$

We will say that term graphs G_1 and G_2 are equivalent *if they are isomorphic.*

Σ-term graph homomorphisms may be viewed as defining a sort of "pattern matching" from one graph to another. They will be useful as a means of singling out term graphs that are homomorphic images of trees or contexts. Term graph homomorphisms preserve the order of operation symbols, but are *not* necessarily Brouwerian algebra homomorphisms. We have to be careful to distinguish between transformations that preserve the *label* structure (e.g., a term graph homomorphism), and those that preserve the *connectivity* structure of the graph (e.g., a Brouwerian Algebra homomorphism).

It will be useful to generalize Definition 4.66 a bit to allow certain operation symbols to be "universal," in the sense that they may be mapped to any other operation symbol by a homomorphism. We proceed as follows:

Definition 4.67 (Generalized Term Graph Homomorphism)
Let Σ be a signature, and let Γ be a subset of Σ containing some set of universal *operation symbols. Then given two Σ-term graphs G_1 and G_2, a mapping*

$$h: G_1 \to G_2$$

is a generalized Σ-homomorphism with universal set Γ *iff for all nodes $\nu \in G_1$,*

$$op(h(\nu)) = f \text{ and } f \notin \Gamma$$

implies

$$op(\nu) = f$$

and

$$succ(h(\nu)) = h^*(succ(\nu))$$

where h^* is defined as in Definition 4.66.

The set of holes $\mathcal{H}$ (Definition 4.7) is a natural candidate for a set of universal operation symbols. Thus we will refer to a generalized $\Sigma(\mathcal{H})$-term graph homomorphism (relative to $\mathcal{H}$) simply as a $\Sigma(\mathcal{H})$ homomorphism.

Example 4.8 Figures 4.5 and 4.6 contain two examples of graph homomorphisms. The former (taken from [BvEG+87a] is a homomorphism from a tree to a graph. The latter is a generalized homomorphism from a context to a graph (recall from Definition 4.7 that a context is a term (i.e., tree) whose leaves are labeled with uniquely indexed *holes*).

Following [BvEG+87a], we can define a straightforward mapping from an arbitrary term graph G and a node ρ of G to a corresponding term graph $U(G,\rho)$ that is a partial forest. We do so by using rooted paths in the graph (each of which begins with ρ) to define nodes of the resulting forest. This mapping is thus called the *unraveling* of G with respect to ρ, and is defined formally as follows:

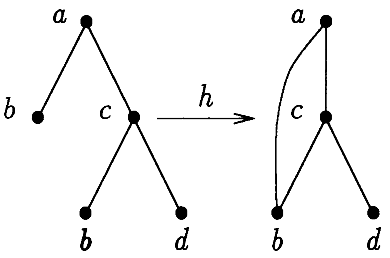


Figure 4.5: Homomorphism h from Tree T to Graph G

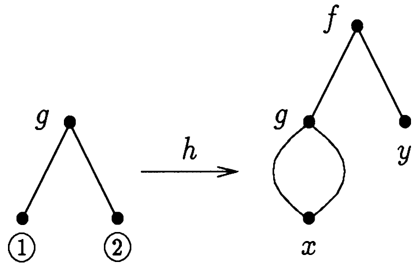


Figure 4.6: (Generalized) Homomorphism h from Context C to Graph G

Definition 4.68 (Unraveling)

Let Σ be a signature and $\mathcal{I}$ be a set. Let $\mathcal{P}$ be the set of paths in graphs of $\mathrm{Graph}(\Sigma)_{\mathcal{I}}$*. Finally, let G be an element of* $\mathrm{Graph}(\Sigma)_{\mathcal{I}}$*, and ρ be a node of G. Then the* unraveling mapping

$$U(G, \rho)$$

is the set of nodes of the form

$$\langle p_0,\ f,\ \vec{p}\rangle$$

such that for each such node, the following hold:

(1)

$$p_0 \equiv \langle\langle \rho, i_0, \nu_1, \ldots, i_{m-1}, \nu_m \rangle\rangle$$

is a path in G.

(2)

$$f = \mathrm{op}(\nu_m)$$

(3)

$$\vec{p} = \langle\langle p_1, \ldots, p_l \rangle\rangle$$

where each $p_k \in \vec{p}$ *is the result of appending*

$$\langle\!\langle k, \pi \rangle\!\rangle$$

to p_0*, where*

$$id(pi) = succ_k(\nu_m)$$

If G *is an element of* $PGraph1(\Sigma)$*, we use*

$$U(G)$$

to denote

$$U(G, root(G))$$

We will feel free to regard $U(G)$ *as a Brouwerian algebra homomorphism from* $Graph1(\Sigma)$ *to* $\mathcal{T}_0(\Sigma)$ *by interpreting the tree term graph resulting from* $U(G)$ *as a labeled tree in the obvious way.*

The result of an unraveling is clearly a partial tree, since there is only one instance of any successor value. Note that if G is proper, $U(G, \rho)$ is also proper, and if G is acyclic, $U(G, \rho)$ must be finite. Term graph G_1 in Example 4.7 is the unraveling of G_2, where the node identifiers of G_1 may be defined in terms of paths in G_2 as follows:

$$\begin{aligned}
i_1 &\triangleq \langle\!\langle \langle j_1,\ f,\ \langle\!\langle j_2, j_3 \rangle\!\rangle \rangle \rangle\!\rangle \\
i_2 &\triangleq \langle\!\langle \langle j_1,\ f,\ \langle\!\langle j_2, j_3 \rangle\!\rangle \rangle, 1, \langle j_2,\ g,\ \langle\!\langle j_4, j_4 \rangle\!\rangle \rangle \rangle\!\rangle \\
i_3 &\triangleq \langle\!\langle \langle j_1,\ f,\ \langle\!\langle j_2, j_3 \rangle\!\rangle \rangle, 2, \langle j_3,\ y,\ \langle\!\langle\rangle\!\rangle \rangle \rangle\!\rangle \\
i_4 &\triangleq \langle\!\langle \langle j_1,\ f,\ \langle\!\langle j_2, j_3 \rangle\!\rangle \rangle, 1, \langle j_2,\ g,\ \langle\!\langle j_4, j_4 \rangle\!\rangle \rangle, 1, \langle j_4,\ x,\ \langle\!\langle\rangle\!\rangle \rangle \rangle\!\rangle \\
i_5 &\triangleq \langle\!\langle \langle j_1,\ f,\ \langle\!\langle j_2, j_3 \rangle\!\rangle \rangle, 1, \langle j_2,\ g,\ \langle\!\langle j_4, j_4 \rangle\!\rangle \rangle, 2, \langle j_4,\ x,\ \langle\!\langle\rangle\!\rangle \rangle \rangle\!\rangle
\end{aligned}$$

By careful examination of results of Barendregt, et al. [BvEG+87a], we note that the unraveling mapping has the following properties:

Proposition 4.19 ([BvEG+87a, Prop. 2.8]) *Let* Σ *be a signature. Let* G *be an element of* $Graph(\Sigma)$*. Then the following hold:*

(1) *For every node* $\nu \in G$*, there is a term graph homomorphism from* $U(G, \nu)$ *to* G*.*

(2) *For every node* $\nu \in G$*, there is a Brouwerian algebra homomorphism from the Brouwerian sub-algebra generated by* G *to the Brouwerian sub-algebra generated by* $U(G, \nu)$ *(see Theorem 3.9).*

(3) *If* G *is an element of* $Graph1(\Sigma)$*, then* $U(G)$ *is an element of* $\mathrm{Tree}(\Sigma)$*.*

From Proposition 4.19, we note that G preserves the *label* structure of $U(G)$, while $U(G)$ preserves the *graph connectivity* of G.

4.8.3 Term Graph Convention

We note from Proposition 4.19, clause (3) that (proper) elements of $Graph1(\Sigma(\mathcal{V}_0))$ can be treated as (proper) elements of $Tree(\Sigma(\mathcal{V}_0))$. We also note that elements of $Tree(\Sigma(\mathcal{V}_0))$ can be treated as elements of $\mathcal{T}(\Sigma(\mathcal{V}_0))$ via appropriate trivial Brouwerian algebra isomorphisms. We will thus be able to use sort-respecting elements of $Graph1(\Sigma(\mathcal{V}_0))$, $Tree(\Sigma(\mathcal{V}_0))$, $\mathcal{T}(\Sigma(\mathcal{V}_0))$, and $Ter(\Sigma)$ interchangeably in many contexts.

4.8.4 Operations on Proper Term Graphs

We now define operations on proper term graphs analogous to those defined in Section 4.2.2 for terms. All graphs in the sequel are assumed to be proper, and the operations that we will define on graphs maintain propriety.

We first define what it means to *redirect* an edge of the graph to point to a new node, a generalization of the notion of rerooting of Definition 4.12:

Definition 4.69 (Redirection)
Let ν be a graph node such that $\nu \in GNode(\Sigma)_{\mathcal{I}}$, and let $i \in \mathcal{I}$ be a node identifier. Then given $k \in 1 \ldots size(\nu)$, we define

$$\nu[k \mapsto i]$$

to be the node which is identical to ν, except that $succ_k(\nu) = i$. We say that $\nu[k \mapsto i]$ results from redirection *of the k-th successor of ν to i.*

We now generalize the notion of *replacement* given in Definition 4.14:

Definition 4.70 (Graph Replacement)
Let Σ be a signature. Let G be an element of $Graph(\Sigma)$, let H be an element of $Graph1(\Sigma)$ such that $H \sqsubseteq G$, and let K be an element of $Graph1(\Sigma)$ such that $K \sqcup (G \dot{-} H)$ is proper. Then the replacement *of H by K in G, notated $G[H \leftarrow K]$, is given by:*

$$G[H \leftarrow K] \;\triangleq\; (G' \dot{-} H) \sqcup K$$

where G' is the graph defined by

$$G' \triangleq \{\pi[1 \mapsto \dot{i}_1][2 \mapsto \dot{i}_2]\cdots[n \mapsto \dot{i}_n] \mid \pi \in G\}$$

and for all $k \in 1 \ldots size(\pi)$, $\dot{i}_k$ is defined by

$$\dot{i}_k \triangleq \begin{cases} id(root(K)) & i_k = id(root(H)) \\ i_k & \text{otherwise} \end{cases}$$

If ν is a node of $G \in Graph1(\Sigma)$, we will use

$$G[\nu \leftarrow K]$$

as shorthand for

$$G[\{\nu\} \leftarrow K]$$

Thus we get $G[H \leftarrow K]$ by redirecting those successors in G that point to the root of H to the root of K. Note, however, that if

$$id(root(H)) = id(root(K))$$

then the redirection operation has no effect, and we have

$$G[H \leftarrow K] = (G \dot{-} H) \sqcup K$$

In this case, we can think of the root of H as being *overwritten* by K, which is precisely the means by which we will attempt to implement the replacement operation whenever possible. Note also that this simplification doesn't preclude the possibility of the root of K being quite different from the root of H—we only require that they have the same node *identifier*, not that they are the same *node*.

We can now generalize the notion of context given in Definition 4.7:

Definition 4.71 (Graph Context)
Let Σ be a signature and $\mathcal{H}$ be the set of Σ-holes. Then $G \in Graph(\Sigma(\mathcal{H}))$ is a (generalized) Σ-context iff for any two nodes $\nu, \pi \in G$, $op(\nu) = (i)$ and $op(\pi) = (j)$ imply that $\nu \equiv \pi$. The set of all graph Σ-contexts is denoted by $GCon(\Sigma)$. Contexts that are elements of $PTree(\Sigma)$ will be called Σ-tree contexts.

As before,

$$C[(1), (2), \ldots, (m)]$$

will denote a context containing exactly the holes indexed $1, 2, \ldots, n$.

We now generalize Definition 4.15:

Definition 4.72 (Graph Context Completion)
Let Σ be a signature. Let

$$C[(i_1), (i_2), \ldots, (i_m)]$$

be an element of $GCon(\Sigma)$, and let Π be the set

$$\Pi \triangleq \{\pi_{i_k} \mid op(\pi_{i_k}) = (i_k)\}$$

Let $\langle\!\langle G_1, G_2, \ldots, G_n \rangle\!\rangle$ be a list of n elements of $Graph1(\Sigma)$, such that for all $i \neq j$, $root(G_i) \neq root(G_j)$, and such that

$$(\bigsqcup_i G_i) \sqcup (C[(i_1), (i_2), \ldots, (i_m)] \dot{-} \Pi)$$

is proper. Then, by analogy the notation used for terms,

$$C[\langle\!\langle G_1, G_2, \ldots, G_n \rangle\!\rangle]$$

will denote the graph

$$C[\pi_{i_1} \leftarrow G_{i_1}][\pi_{i_2} \leftarrow G_{i_2}] \cdots [\pi_{i_n} \leftarrow G_{i_n}]$$

where $i_k \in 1 \ldots n$. *We will refer to the graph above as a (generalized) completion of the context* $C[(i_1), (i_2), \ldots, (i_m)]$.

We then define what it means for a context to *match* a graph:

Definition 4.73 (Context Matching)
Let Σ *be a signature. Let*

$$C[(1), (2), \ldots (n)]$$

be an element of $GCon(\Sigma)$ *and* G *be an element of* $Graph1(\Sigma)$. *Then we say that* C matches G *iff there is a generalized homomorphism* h *(with the set of holes* $\mathcal{H}$ *as the universal set) from* C *to* G.

$$h(G)$$

is the element matched *by* C.

If for each node ν *of* G *there exists a node* π *of* C *such that* $h(\pi) = \nu$, *then we say the match is* exact.

The list of bindings induced by a match of C and G, *notated* $match_h(C, G)$, *is defined by*

$$match_h(C, G) \triangleq \langle\langle DC(h(\pi_1), G), DC(h(\pi_2), G), \ldots, DC(h(\pi_n), G)\rangle\rangle$$

Where π_i *denote the node of* C *whose operation symbol is* (i).

We see that $match_h(C, G)$ is a list of those subgraphs of G corresponding (via the homomorphism h) to the holes in C. We will say that a graph G has the *form of* some context C if there is an exact match between C and G (note that the existence of an exact match between C and G does not imply that C and G are isomorphic). In Figure 4.6 of Example 4.8, the match between context C and graph G is not exact, since the nodes labeled f and g are not the images of any node of C. However, we can say that there is a *sub*graph of G having the form of $g((1), (2))$. The set of bindings induced by the match of C to G is

$$\langle\langle \nu_x, \nu_x \rangle\rangle$$

where ν_x is the node of G labeled x.

The act of matching a context C to a graph G has the effect of singling out those subgraphs of G that correspond to the holes in C. We make this relationship a bit more precise as follows:

Lemma 4.20 *Let Σ be a signature. Let*

$$C[①, ②, \dots ⓝ]$$

be an element of $GCon(\Sigma)$ and G be an element of $Graph1(\Sigma)$ such that C matches G via homomorphism h. Then there exists a context

$$C'[①, ②, \dots ⓝ]$$

such that

$$G = C'[match_h(C, G)]$$

We will say that $C'[match_h(C, G)]$ is the decomposition *of G induced by the match of C to G.*

Proof Let

$$\langle\!\langle B_1, B_2, \dots, B_n \rangle\!\rangle \quad = \quad match_h(C, G)$$

We let C_1 be the graph given by taking the difference of G and the set

$$\bigsqcup_i B_i$$

of elements of $match_h(C, G)$. C' is then defined by redirecting, for all $i \in 1 \dots n$, all of those successors identifiers of nodes in C_1 which point to $root(B_i)$ to point to a new hole node ⓘ. □

We note that the elements of the list of bindings $match_h(C, G)$ above are not necessarily disjoint, (as shown, e.g., by figure 4.6 of Example 4.8), nor are they necessarily disjoint from C.

When a *tree* context matches a graph, we get an even nicer notion of decomposition:

Lemma 4.21 *Let*

$$C[①, ②, \dots ⓝ]$$

an element of $(GCon(\Sigma) \cap \mathrm{Tree}(\Sigma))$ (i.e., a tree Σ-context). Let G be an element of $Graph1(\Sigma)$ such that C matches G via homomorphism h. Then there exists contexts

$$C_1[\odot]$$

and

$$C_2[①, ②, \dots ⓝ]$$

such that $C[\odot]$ matches $C_2[①, ②, \dots ⓝ]$ and

$$G = C_1[\, C_2[match_h(C, G)]\,]$$

We will say that $C_1[\, C_2[match_h(C, G)]\,]$ is the tree decomposition *of G induced by the match of C to G.*

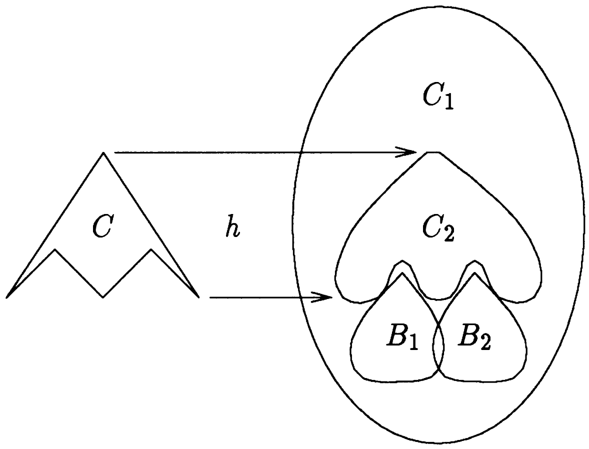


Figure 4.7: Tree Decomposition

Proof We note that if C is a tree, there must be a unique node of ρ_C of G to which the root of C is mapped. Thus C matches $DC(\rho_C, G)$, and by Lemma 4.21, there exists C_2 such that

$$DC(\rho_C, G) \quad = \quad C_2[match_h(C, G)]$$

Let G' be the graph

$$G' \quad \equiv \quad G \dot{-} C_2[match_h(C, G)]$$

We then get $C_1[\odot]$ by redirecting those successors of nodes of G'w that point to ρ_C to a new node with operation symbol $\odot$, from which we get

$$G \quad = \quad C_1[C_2[match_h(C, G)]]$$

$\square$

Figure 4.7 illustrates the tree decomposition induced by a match of some context C schematically.

In Figure 4.6 of Example 4.8, the match of C to G yields the decomposition

$$C_1[\,C_2[\langle\!\langle x, x\rangle\!\rangle]\,]$$

where

$$C_1 \equiv f(①, y)$$

and

$$C_2 \equiv \{g(\pi, \pi), \pi : x\}$$

4.9 Term Rewriting and Graph Rewriting

In Sections 4.2 and 4.4, we showed that elements of $\mathcal{T}(\Sigma(\mathcal{V}_0))$ could be viewed as concrete manifestations of $Ter(\Sigma)$ for the purpose of implementing term rewriting. Using the machinery we have developed for abstract replacement systems, we will here sketch the results of Barendregt, et al. [BvEG+87a] showing that graphs can be used to implement term rewriting as well, while allowing identical subterms to be shared.

From our perspective, the most important aspect of graph rewriting implementations of regular term rewriting systems is that *each graph contraction effectively contracts complete sets of residuals in the corresponding term rewriting system*, which we will enable us to show in Chapter 6 that such implementations are optimal.

We begin with the following definition:

Definition 4.74 (Term Graph Rewriting Systems)
Let Σ be a signature. A Σ-term graph rewriting system $\longrightarrow_R$ *is a rewriting relation on proper elements of $Graph1(\Sigma)$.*

An abstract replacement system

$$R \equiv \langle Graph(\Sigma(\mathcal{V}_0)), \longrightarrow_R \rangle$$

is a $\Sigma(\mathcal{V}_0)$-term graph replacement *system iff R is a Σ-term graph rewriting system and for all proper graphs $(H \sqcup G) \in Graph1(\Sigma(\mathcal{V}))$,*

$$G \longrightarrow_R G'$$

implies $H \sqcup G'$ is proper.

If R coupled with an appropriate residual map '/' is a regular replacement system, then we say that R is a regular $\Sigma(\mathcal{V}_0)$-term graph rewriting system

We require that the underlying reduction relation for graph replacement systems be restricted to *rooted graphs*, rather than arbitrary graphs, since it is the rooted graphs that correspond to terms.

The work of Barendregt, et al., and Staples shows that regular term rewriting system can be implemented as graph rewriting systems. We will briefly outline their ideas in an informal manner, adapting their terminology to our notion of an RRS, and refer the reader to [BvEG+87a] or [Sta80a,Sta80b,Sta80c] for details.

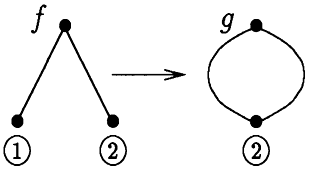
The general idea was discussed in conjunction with the lambda calculus in Section 2.3.1, and amounts to sharing subgraph corresponding to repeated instances of a variable in a rule's right-hand side. More particularly, we view each rewriting rule as a graph rewriting rule that affects *only* the subgraph to which the non-variable nodes of the rule's left-hand side corresponds.

In a term rewriting system, each redex of a term is a subterm that is a substitution instance of some left-hand side of a rewriting rule. To get a corresponding graph replacement system, we replace the variables in a term rewriting rule by holes, sharing holes that correspond to repeated variables. Each graph rewriting rule is then given by a pair of (graph) contexts. The following simple example should give a feel for the correspondence between a term rewriting rule and its corresponding graph rewriting rule.

Example 4.9 Consider the rewrite rule

$$f(x, y) \quad \longrightarrow \quad g(y, y)$$

The analogous graph rewriting rule is given by:



Or, in textual form:

$$f(\textcircled{1}, \textcircled{2}) \quad \longrightarrow \quad \{g(\nu, \nu),\ \nu : \textcircled{2}\}$$

Thus to get a term graph rewriting rule from a term rewriting rule, we replace corresponding variables by correspondingly numbered holes. Repeated instances of right hand side variables in a term rewriting rule correspond to shared instances of the appropriate hole. Each side of a term graph rewriting rule is thus a proper element of $GCon(H) \cap Graph1(\Sigma)$.

4.9.1 Graph Rewriting

Making the discussion above a bit more formal, we have the following definition:

Definition 4.75 (Term Graph Rewriting Rule)
Let R be a regular term rewriting system and

$$(N) \;\; P_l \quad \longrightarrow \quad P_r$$

be a rule of R containing variables $x_1, x_2, \ldots, x_n$. Then the term graph rewriting rule *N^G corresponding to (N) is a pair of the form*

$$(N^G) \;\; G_l \quad \longrightarrow \quad G_r$$

where G_l is the proper element of $(GCon(\Sigma) \cap \mathrm{Tree}(\Sigma))$ such that there exists a Σ-algebra isomorphism from G_l to P_l mapping $\textcircled{i}$ to x_i for $i \in 1 \ldots n$.

G_r is then a proper element of $(GCon(\Sigma) \cap Graph1(\Sigma))$ such that there exists a (Σ-algebra) homomorphism from G_r to P_r mapping $\textcircled{i}$ to x_i for $i \in 1 \ldots n$, and such that

$$G_r' \quad = \quad G_r \dot{-} \{\pi \mid op(\pi) = \textcircled{i} \text{ for some } i\}$$

is an element of $\mathrm{Tree}(\Sigma)$.

Note that by definition of a graph context, the holes in G_r must be unique, hence the requisite sharing follows automatically.

We now must define term graph *redex*. We proceed as follows:

Definition 4.76 (Term Graph Redex)
Let H be an element of $Graph1(\Sigma)$. Let (P) be a term graph rewriting rule of the form

$$(P) \;\; G_l \;\; \longrightarrow \;\; G_r$$

Let G_l' be the maximal subgraph of G_l not containing any elements of $\mathcal{H}$. Then $K \sqsubseteq H$ is a (P)-redex *of H iff G_l matches H via homomorphism h, and*

$$K \;\; = \;\; h(G_l')$$

Definition 4.76 says that a term graph redex K of H consists of the image of the non-hole (i.e., non-variable) parts of the left hand side of the graph rewriting rule G_l.

We now define the *contraction* of a term graph redex as follows:

Definition 4.77 (Term Graph Redex Contraction)
Let (P) be a graph rewriting rule of the form

$$(P) \;\; G_l \;\; \longrightarrow \;\; G_r$$

Let H be an element of $Graph1(\Sigma)$, K be a (P)-redex of H, and h be the homomorphism induced by the match of G_l to H. Let G_r' be a copy *of G_r, i.e., the image of a Σ-algebra isomorphism in $Graph1(\Sigma)$ such that $G_r' \parallel (H \dot{-} K)$ and*

$$id(root(G_r')) \;\; = \;\; id(root(K))$$

Let

$$\langle\!\langle B_1, B_2, \ldots, B_n \rangle\!\rangle \;\; = \;\; match_h(G_l, H)$$

be the set of bindings induced by the match of G_l to K.

Let H' be defined by

$$H' \;\; \triangleq \;\; H[K \;\leftarrow\; G_r'[① \leftarrow B_1][② \leftarrow B_2] \cdots [ⓝ \leftarrow B_n]]$$

Then we say that H reduces to H' by contraction *of term graph redex K.*

4.9.2 Graph Replacement

Note that unlike our previous definitions for contraction operations, Definition 4.77 supplies not just with a transformation on the redex in isolation, but also the graph in which the redex is contained (H). We would like our graph rewriting system

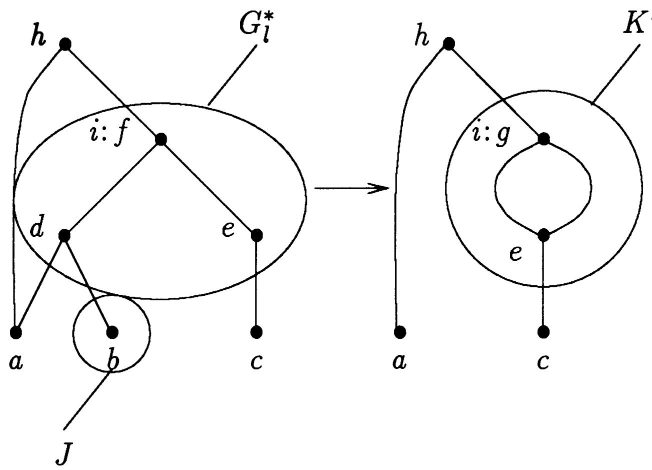


Figure 4.8: Reduction of Graph Redex $K = (G_l^* \sqcup J)$ to K'

to form an abstract replacement system. That is, for every (P)-term graph redex K of H, we would like there to exist K' such that the reduction of H to H' is achieved by *replacing* K by its contractum K', i.e.,

$$H \sqcup K \quad \longrightarrow \quad (H \dot{-} K) \sqcup K'$$

The problem stems from projection rules, which require that the successors of nodes in H that point to the redex be redirected to the subgraph of H that is being projected out. However, in all other cases, the redirection operation has no effect (see the comment after Definition 4.70), since the root of the copy of the reduction rule's right hand side has the same node identifier as the redex did prior to contraction. Therefore, if the term rewriting system being simulated is *strongly* regular, the contraction operation has no effect on the graph containing the redex, and the resulting system is an abstract replacement system.

Example 4.10 Figure 4.8 shows how a subgraph replacement is induced by the application of the graph rewrite rule of Example 4.9. In particular, note that the node identifiers of the root of the redex and the root of its contractum are the same.

We have glossed over a small formal problem in our rush to proclaim that strongly regular term rewriting systems yield abstract replacement systems: When a copy of a reduction rule's right hand side is created by the contraction operation, we must have a way to generate new node identifiers disjoint from the rest of the graph, in order to maintain propriety. This can be done with an identifier creation scheme that forms new redex identifiers from sets of redex identifiers in the replaced redex. Using this scheme, it is easy to generate new node identifiers in such a way as to guarantee their disjointness from old identifiers in the remaining graph. There is also a slight complication with the node identifier of the root of the contractum, since (in order to avoid redirection) we require it be the same as the identifier of the root of the redex. One could thus add "generation tags" to the node definition, which are used to distinguish among various incarnations of nodes with the same identifier, and which can be used to generate disjoint node identifiers for copied nodes.

4.9.3 Regular Graph Replacement Systems

From a careful reading of [BvEG+87a, Thm. 5.12], we get the following theorem (rephrased in our terminology):

Theorem 4.22 ([BvEG+87a, Thm. 5.12]) *Let R be a regular Σ-term rewriting system. Let* **id** *be the Σ-algebra isomorphism from $\mathcal{T}(\Sigma)$ to Graph1(Σ), and U be the unraveling mapping. Then the following hold:*

(1) *Then there exists a term graph rewriting system R^G such that*

$$\langle R^G, \mathbf{id}, U\rangle$$

is a complete implementation of R.

(2) *If R is strongly regular, then there exists a disjoint regular term graph replacement system R^G such that*

$$\langle R^G, \mathbf{id}, U\rangle$$

is a strong family-preserving complete implementation of R.

The corresponding graph rewriting system is the one whose reduction rules are defined by Definition 4.75 and whose notion of reduction follows from Definition 4.77. The "completeness-preserving" part of clause (2) is a consequence of the observation in [BvEG+87a, Thm. 5.12] that so-called Gross-Knuth reductions are preserved by term graph rewriting systems.

4.9.4 Term Graph Redex Location

By analogy with the terminology of Section 4.4.7, we can single out particular redexes in a graph:

Definition 4.78 (Outermost Term Graph Redex)
Let Σ be a signature, R^G be a Σ term rewriting system, G be an element of Graph1(Σ), and A be an R^G-term graph redex of G. Then A is outermost *iff the image of A under the unraveling mapping $U(G)$ is outermost in any element of* Ter(Σ) *isomorphic to $U(G)$.*

Definition 4.79 (Leftmost Term Graph Redex)

Let Σ be a signature, R^G be a Σ term rewriting system, G be an element of Graph1(Σ), and $\mathcal{S}$ be a set of term graph redexes of G such that their images under the unraveling mapping $U(G)$ are disjoint. Then $A \in \mathcal{S}$ is leftmost *iff the image of A under the unraveling mapping $U(G)$ is leftmost among the images of $\mathcal{S}$ in any element of* Ter(Σ) *isomorphic to $U(G)$.*

4.9.5 The Lambda Calculus and Graph Replacement

In Section 4.7.1, we noted that the classical definitions of residual in the lambda calculus and the consequent notion of parallel reduction are not expressible in the framework of an abstract replacement system, although the resulting reduction theory is the same. However, if we view the lambda calculus as *fundamentally* a reduction relation on graphs, with terms as a convenient means of *notating* graphs rather than the primary objects of interest, we can explain residuals and parallel reduction in the framework of an RRS.

We will call the graph replacement analogue of β-reduction *naive* graph reduction, notated β^G, to distinguish it from more sophisticated (and efficient) variants that have been proposed, e.g., by Wadsworth [Wad71] (since we only wish to justify treating the lambda calculus as an RRS, concerns of efficiency are of no consequence here). We will call the set of graphs defined on the signature of λ-terms *λ-graphs*.

In outlining an appropriate notion of graph replacement for the lambda calculus, our approach is quite similar to that of the term graph rewriting analogues to regular term rewriting systems systems, except that instead of having "fixed" rules in the latter case, the rules are variable depending on the structure of the body of an abstraction.

As with β-reduction on terms, any subgraph of the form

$$((\lambda x.H)K)$$

is a redex.

Let G be a λ-graph containing a redex subgraph

$$R \equiv ((\lambda x.H)K)$$

To construct the contractum R' of R in G, we proceed as follows:

Let H^* be a graph context which is a copy of H such that

$$id(root(H^*)) \quad = \quad id(root(R))$$

all non-root nodes of H^* are disjoint from G, and all free instances of variable x in H are replaced in H^* by n uniquely indexed holes. Then R' is given by

$$R' \triangleq H^*[① \leftarrow K][② \leftarrow K] \cdots [ⓝ \leftarrow K]$$

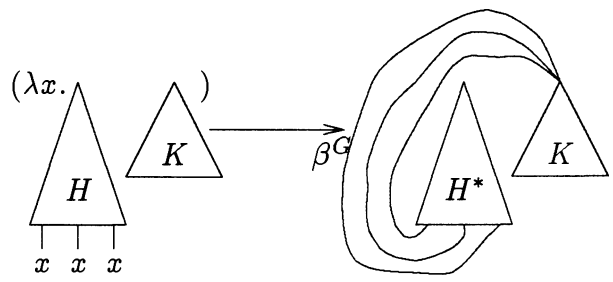


Figure 4.9: β^G Contraction

Figure 4.9 illustrates the β^G-contraction operation.

It is not difficult to define an appropriate notion of residual for naive graph reduction and thus show that naive graph reduction yields an RRS, although we will not do so here. One can also show that classical β-residuals result from the unraveling of the graph residuals. Note that in contrast to the situation with strongly regular term rewriting systems, the graph replacement version of β-reduction is *not* disjoint, and thus families do not have a single representative (it will turn out that this fact makes a practical *optimal* implementation of the β-reduction very difficult to devise).

The advantage of the graph-based approach to β-reduction is that residuals are now always disjoint, and thus the many complications caused by defining parallel reduction in terms of (finite) developments, disappear.

Chapter 5

Reduction with Environments and Closures

"Sequential continuation on a new running theme."

"If [he] uses an old convention, we must find out how it fits the use he makes of it, instead of imagining that its origin elsewhere explains its presence here."

—Tovey

A number of reduction schemes for the lambda calculus have been proposed using the notions of *environment* and *closure*. These include those of Landin [Lan64] using applicative order evaluation, and updated versions devised by Henderson and Morris [HM76] and Aiello and Prini [AP81] to accommodate leftmost evaluation. Each of these systems avoids immediate substitution of an argument for all instances of bound variables in the body of a λ-abstraction after β-contraction, constructing a closure instead.

In this chapter, we introduce a new term rewriting system, $\mathbf{\Lambda CCL}$, which is intended as a formal tool for understanding reduction using closures and environments. There are two reasons to formalize these ideas. First, if we are to develop a theory of incremental lambda-reduction that accurately reflects realistic computations, we must devise a model of reduction that incorporates the substitution operation *explicitly*, rather than relegating it to a vague meta-theoretical level. Second, variants of closures and environments are used in most implementations of programming languages (particularly functional ones), and a correct formal treatment of these ideas is thus more likely to have immediate practical applications.

We will show that $\mathbf{\Lambda CCL}$ is a confluent monomorphic extension (Definition 3.21) of the notion of β-reduction on λ-terms, and thus can be used in place of λ-reduction while yielding essentially the same results. While $\mathbf{\Lambda CCL}$ loses none of the computational power of the untyped lambda calculus, it has the advantage of making the manipulations required by the substitution operation in the lambda calculus *explicit*, a fact that can be used to implement alternative substitution mechanisms in $\mathbf{\Lambda CCL}$-based implementations. In particular, this flexibility with regard to performing substitutions will be used to advantage in implementing incremental λ-reduction.

5.1 Environments and Closures

An environment consists of sets of mappings between variable names and values, or *bindings*. The result of a β-contraction is then a *closure* consisting of the body of the abstraction part of the redex, paired with an environment updated to contain the binding of the abstraction's bound variable to the argument of the redex. The idea is illustrated below:

$$(\lambda x.(xx))N \longrightarrow [(xx),\ \langle\!\langle x := N\rangle\!\rangle]$$

In general, $[T,\ E]$ will represent a closure consisting of term T and environment E. An environment is denoted thus:

$$\langle\!\langle B_1, B_2, \ldots\rangle\!\rangle$$

where B_1, B_2, etc. are bindings.

The following example (using the same term as in Example 2.3) shows that sharing of λ-terms can be achieved indirectly through shared bindings:

Example 5.1

$$
\begin{array}{rcl}
(\lambda y.(yy))(Iz) & \longrightarrow & [(yy),\ \langle\!\langle y := (Iz)\rangle\!\rangle] \\
 & \longrightarrow & ([y,\ \bullet][y,\ \bullet]) \\
 & & \qquad\qquad\quad \langle\!\langle y := (Iz)\rangle\!\rangle \\
 & \longrightarrow & ([y,\ \bullet][y,\ \bullet]) \\
 & & \qquad\qquad\quad \langle\!\langle y := z\rangle\!\rangle \\
 & \vdots & \\
 & \longrightarrow & (zz)
\end{array}
$$

Use of closures obviates copying any part of the body of an abstraction after β-contraction. By contrast, Wadsworth's graph reduction scheme copies the parts of the body of an abstraction containing the abstraction's bound variable, in order to avoid incorrect substitutions in pieces of the abstraction's body that might be shared by other terms. By using environments, the body of the abstraction term, and hence any redexes contained therein, have the potential to be shared, avoiding redundant reductions.

The reduction given below of the term of Example 2.4 shows that shared environments can be used to minimize the number of redex contractions performed in a nominally leftmost strategy. Let $N \equiv (N_1 N_2)$, where $N_1 \equiv \lambda x.(xw)(xz)$ and $N_2 \equiv \lambda y.(Iy)$. Then, using shared environments, we have:

Example 5.2

$$
\begin{aligned}
N &\longrightarrow [(xw)(xz),\ \langle\!\langle x := N_2 \rangle\!\rangle] \\
&\twoheadrightarrow (([x,\ \bullet][w,\ \bullet])\ [(xz),\ \bullet]) \\
&\qquad\qquad \langle\!\langle x := N_2 \rangle\!\rangle \\
&\longrightarrow ((\bullet\ [w,\ \bullet])\ [(xz),\ \bullet]) \\
&\qquad\qquad \langle\!\langle x := \bullet \rangle\!\rangle \\
&\qquad\qquad\quad \lambda y.(Iy) \\
&\longrightarrow (([\bullet,\ \langle\!\langle y := [w,\ \bullet] \rangle\!\rangle])\ [(xz),\ \bullet]) \\
&\qquad\qquad \langle\!\langle x := \lambda y.\bullet \rangle\!\rangle \\
&\qquad\qquad\quad (Iy) \\
&\longrightarrow (([\bullet,\ \langle\!\langle y := [w,\ \bullet] \rangle\!\rangle])\ [(xz),\ \bullet]) \\
&\qquad\qquad \langle\!\langle x := \lambda y.\bullet \rangle\!\rangle \\
&\qquad\qquad\quad y \\
&\longrightarrow \\
&\ \vdots
\end{aligned}
$$

Note that the (Iy) redex in N_2 is reduced in a shared environment, *independently* of the substitution for free variable y in closures that refer to N_2.

The question as to whether some *combination* of shared environments, closures, and terms could be used to achieve an optimal reduction scheme, or at least improve on Wadsworth's method will be examined in Chapter 6. To proceed any further, however, we will need a more formal system to study reduction using environments and closures.

5.2 ΛCCL

In [Cur86a], P.-L. Curien defines a number of equational theories based on Cartesian Closed Categories (CCCs) using terms from the *Pure Categorical Combinatory Logic*, **CCL**. Curien observed that the CCC axioms could model *reduction* in the lambda calculus, i.e., its operational semantics as well as its denotational semantics. Treated as combinators, Curien's axioms have the advantage of avoiding the difficulties with variables and substitution normally encountered in the lambda calculus, and thus has aspects in common with the *de Bruijn lambda calculus* [dB72, dB78].

One set of equational axioms, deemed *Weak Categorical Combinatory Logic*, is the basis for the Categorical Abstract Machine ([CCM87]). However, Curien proposed no confluent system strong enough to simulate arbitrary β-reductions in the lambda calculus that could itself be simulated using only β-reduction. If such a system were available, it would provide an immediate proof of correctness for any reduction scheme for the lambda calculus based on it (since any combinator reduction would correspond to a β-reduction). λ-reduction methods based on Categorical Combinators proposed thus far, such as the Categorical Abstract Machine and schemes by Lins [Lin87], have heretofore required ad-hoc proofs of correctness.

To provide a more sophisticated tool for modeling and analyzing λ-reduction using environments, we will define a new 2-sorted equational theory, $\mathbf{\Lambda CCL}$, akin to Curien's theory $\mathbf{CCL\beta}$. Its sort structure makes possible relatively simple proofs of close correspondence between β-reduction and $\mathbf{\Lambda CCL}$ reduction not possible in Curien's original theory. More importantly, $\mathbf{\Lambda CCL}$ has a very natural interpretation in terms of structures and operations commonly used to implement λ-reduction, as well as providing a sound framework for entirely new reduction strategies. $\mathbf{\Lambda CCL}$ is not a regular term rewriting system, thus its reduction-theoretic properties cannot be described using the theory we have developed for RRS's. However, in the sequel, we will describe a subsystem of $\mathbf{\Lambda CCL}$ that *does* form an RRS. The properties of $\mathbf{\Lambda CCL}$ are nonetheless quite useful for describing the behavior of reduction using environments.

In the sequel, we will assume that any λ-terms under consideration are actually terms of the de Bruijn lambda calculus (see Section 4.6 for a brief review), although we will feel free to give examples using named variables.

5.2.1 Term Structure

Definition 5.1 ($\mathbf{\Lambda CCL}$ Terms)
The terms of $\mathbf{\Lambda CCL}$ *are built from a set of variables and constructors over a two-sorted signature. The sorts are as follows:*

- $\mathcal{L}$*, the sort of* lambda-like expressions
- $\mathcal{E}$*, the sort of* environments

The constructors are listed below. Each constructor is given with the sort of the term constructed and the sorts of its argument(s) specified in the corresponding argument positions.

$$\begin{array}{rl}
\mathrm{Var} : \mathcal{L} & \textit{(variable reference)} \\
\mathrm{Apply}(\mathcal{L},\, \mathcal{L})\colon \mathcal{L} & \textit{(application)} \\
\Lambda(\mathcal{L})\colon \mathcal{L} & \textit{(abstraction)} \\
[\mathcal{L},\, \mathcal{E}]\colon \mathcal{L} & \textit{(closure)} \\
\emptyset\colon \mathcal{E} & \textit{(null environment)} \\
\Box\colon \mathcal{E} & \textit{(shift)} \\
\langle \mathcal{E},\, \mathcal{L}\rangle\colon \mathcal{E} & \textit{(expression list)} \\
\mathcal{E} \circ \mathcal{E}\colon \mathcal{E} & \textit{(environment composition)}
\end{array}$$

The terms of $\mathbf{\Lambda CCL}$ *will be denoted by* $\mathrm{Ter}(\Lambda\mathrm{CCL})$ *and the* ground terms, *those terms containing no variables, by* $G\mathrm{Ter}(\Lambda\mathrm{CCL})$. *The set of terms of sort* $\mathcal{L}$ *will be denoted by* $\mathrm{Ter}_{\mathcal{L}}(\Lambda\mathrm{CCL})$; *the terms of sort* $\mathcal{E}$ *will be denoted by* $\mathrm{Ter}_{\mathcal{E}}(\Lambda\mathrm{CCL})$.

The following notation (for "de Bruijn" numbers) will be used:

Definition 5.2

$$n! \equiv \begin{cases} \mathrm{Var} & n = 0 \\ [\mathrm{Var},\ \square^n] & n > 0 \end{cases}$$

where

$$\square^n \equiv \begin{cases} \square & n = 1 \\ \underbrace{\square \circ (\square \circ (\cdots (\square \circ \square) \cdots))}_{n\ times} & n > 1 \end{cases}$$

The intuition behind the term structure of **ΛCCL** is fairly straightforward: Terms of sort $Ter_{\mathcal{L}}(\Lambda\mathrm{CCL})$ are analogous to terms in the de Bruijn lambda calculus, after variable numbers are encoded as above. Closures are created by the **ΛCCL** equivalent of β-contraction. Environments are essentially lists of terms, the association between bound variables and the terms to which they are bound being represented implicitly by position in the list. An environment informally presented as

$$\langle\langle x_1 := M_1, x_2 := M_2, \ldots, x_n := M_n \rangle\rangle$$

is represented in **ΛCCL** as

$$\langle\langle\langle \cdots \langle \emptyset,\ M_n \rangle \cdots \rangle,\ M_2 \rangle,\ M_1 \rangle.$$

"$\circ$" allows separate environments to be merged. The only perhaps mysterious term present is "$\square$", which when composed on the left with an arbitrary environment effects the "shifting" of de Bruijn numbers required when environments are moved inside abstractions, and when composed on the right with an environment causes the outermost piece of the list to be stripped away in the course of variable lookup. All these operations are embodied in the axioms below:

5.2.2 Axioms

Definition 5.3 (ΛCCL Axioms)
The axioms of **ΛCCL** *are as follows:*

(Beta)	$\mathrm{Apply}(\Lambda(A),\ B) = [A,\ \langle \emptyset,\ B \rangle]$
(AssC)	$[[A,\ E_1],\ E_2] = [A,\ E_1 \circ E_2]$
(NullEL)	$\emptyset \circ E = E$
(NullER)	$E \circ \emptyset = E$
(ShiftE)	$\square \circ \langle E,\ A \rangle = E$
(VarRef)	$[\mathrm{Var},\ \langle E,\ A \rangle] = A$
(DΛ)	$[\Lambda(A),\ E] = \Lambda([A,\ \langle E \circ \square,\ \mathrm{Var} \rangle])$
(DE)	$\langle E_1,\ A \rangle \circ E_2 = \langle E_1 \circ E_2,\ [A,\ E_2] \rangle$
(DApply)	$[\mathrm{Apply}(A,\ B),\ E] = \mathrm{Apply}([A,\ E],\ [B,\ E])$
(AssE)	$(E_1 \circ E_2) \circ E_3 = E_1 \circ (E_2 \circ E_3)$
(NullC)	$[A,\ \emptyset] = A$

We define a related equational theory, **ECCL**, as follows:

Definition 5.4 (ECCL)
The axioms of **ECCL** *are those of* **ΛCCL** *without rule* **(Beta)**.

The following derived rule will be useful in the sequel:

Definition 5.5 ((Beta′))

$$\textbf{(Beta}'\textbf{)}\quad \mathrm{Apply}([\Lambda(A),\ E],\ B) = [A,\ \langle E,\ B\rangle]$$

(Beta′) is easily derived from the axioms of **ΛCCL**; we omit the details.

It will be useful to consider **ΛCCL** as the union of two systems intended for different purposes: **ECCL**, which governs manipulation of environments, and **(Beta)**, which models β-reduction.

5.3 Properties of ΛCCL

By orienting the equations of **ΛCCL** from left to right, they can be treated as a term rewriting system. We will restrict ourselves in the sequel to the *ground terms* of **ΛCCL**, $GTer(\Lambda\mathrm{CCL})$. Since we are interested in using **ΛCCL** to model λ-reduction rather than to prove theorems, this restriction will be of no concern. More importantly, in conjunction with the 2-sorted term structure of **ΛCCL**, the restriction to ground terms makes it possible to prove properties of **ΛCCL** that did not hold for arbitrary terms of Curien's system **CCLβ**. We will refer to the formal theories and their corresponding rewriting systems by the same name. We first show that **ECCL** and **(Beta)** each yield noetherian term rewriting systems:

5.3.1 Noetherian Subsystems

Theorem 5.1 **ECCL** *is noetherian.*

Proof We can orient the rules of **ECCL** using Kamin and Lévy's *extended recursive path ordering* technique (described in [Hue86, p. 56]).

We first define a quasi-ordering $\succeq_o$ on the operators of **ΛCCL** by:

$$f \equiv g \quad \text{implies} \quad f \approx_o g$$

and

$$\circ \approx_o [\cdot,\ \cdot] \succ_o \langle \cdot,\ \cdot\rangle \succ_o \mathrm{Apply}(\cdot,\ \cdot) \succ_o \Lambda(\cdot) \succ_o \mathrm{Var} \succ_o \Box \succ_o \emptyset$$

Next, for any **ΛCCL** term T, we define a size measure $|T|_{\mathrm{est}}$. $|T|_{\mathrm{est}}$ is intended to serve as a rough upper bound on the ultimate size of T when reduced to normal form (if $T \in Ter_{\mathcal{L}}(\Lambda\mathrm{CCL})$), or, in the case where $T \in Ter_{\mathcal{L}}(\Lambda\mathrm{CCL})$, an upper bound on the size of the largest term "contained" in the environment list, once reduced. $|\cdot|_{\text{est}}$ is defined inductively on the structure of **ΛCCL** terms as follows:

$$\begin{aligned}
|\text{Var}|_{\text{est}} &= 1 \\
|\Lambda(A)|_{\text{est}} &= |A|_{\text{est}} + 1 \\
|\text{Apply}(A,\, B)|_{\text{est}} &= |A|_{\text{est}} + |B|_{\text{est}} + 1 \\
|[A,\, E]|_{\text{est}} &= |A|_{\text{est}} \cdot |E|_{\text{est}} \\
|\emptyset|_{\text{est}} &= 1 \\
|\Box|_{\text{est}} &= 1 \\
|\langle E,\, A\rangle|_{\text{est}} &= \max(|E|_{\text{est}}, |A|_{\text{est}}) \\
|E_1 \circ E_2|_{\text{est}} &= |E_1|_{\text{est}} \cdot |E_2|_{\text{est}}
\end{aligned}$$

Let $A \equiv f(s_1, \ldots, s_m)$ and $B \equiv g(t_1, \ldots, t_n)$ be terms of **ΛCCL** (where f and g are the outermost operators of the terms). We use the definitions above to define a "basic" quasi-ordering $\succeq_t$ on terms as follows:

$$A \succeq_t B$$

if

$$f \succ_o g$$

or

$$f \approx_o g \quad \text{and} \quad |A|_{\text{est}} \geq |B|_{\text{est}}$$

In other words, $\succeq_t$ is simply the lexicographic combination of the ordering $\succeq_o$ on operators and the size estimate $|\cdot|_{\text{est}}$. $\succeq_t$ is clearly a well quasi-ordering .

Finally, following Kamin and Lévy, we extend $\succeq_t$ to a *simplification ordering* (see [Der87, p. 80]), $\succeq$, as follows:

$$A \equiv f(s_1, \ldots, s_m) \succeq B \equiv g(t_1, \ldots, t_n)$$

if

$$s_i \succeq B \quad \text{for some} \quad i \in 1 \ldots m$$

or

$$A \succ_t B \quad \text{and} \quad A \succ t_j \quad \text{for all} \quad j \in 1 \ldots n$$

or

$$A \approx_t B, \quad (s_1, \ldots, s_m) \succeq_* (t_1, \ldots t_n), \quad \text{and} \quad A \succ t_j \quad \text{for all} \quad j \in 1 \ldots n$$

where $\succeq_*$ is the standard lexicographic extension of $\succeq$ to sequences (see [Der87, p. 97] for details of this extension).

Kamin and Lévy give 11 axioms that must be satisfied in order to ensure that $\succeq$ is indeed a simplification quasi-ordering. Most of them are concerned with properties of the lexicographic functional $\succeq_*$ on $\succeq$, and are shown by Kamin and Lévy to hold. The only non-trivial axiom that must be satisfied is the following:

[Hue86, Axiom 11] If $s \longrightarrow t$, and for some i, A, and B,

$$A \equiv f(u_1, u_2, \ldots, u_{i-1}, s, \ldots, u_n)$$

and

$$B \equiv f(u_1, u_2, \ldots, u_{i-1}, t, \ldots, u_n)$$

then $A \succeq_t B$.

This axiom is easy (though tedious) to verify for each rule, by noting that if $A \longrightarrow B$, then $|A|_{\mathsf{est}} \geq |B|_{\mathsf{est}}$. Since $|\cdot|_{\mathsf{est}}$ is defined inductively and monotonically on the structure of terms, we then have that

$$|f(\ldots, A, \ldots)|_{\mathsf{est}} \;\geq\; |f(\ldots, B, \ldots)|_{\mathsf{est}}$$

For each **ECCL** axiom of the form $L = R$, it is then straightforward to show that $L \succ R$. Since $\succ$ is a simplification ordering, this implies that if $A \longrightarrow_{\mathrm{ECCL}} B$, $A \succ B$ and thus that **ECCL** is noetherian[1]. □

We also have the following:

Theorem 5.2 **(Beta)** *is noetherian.*

Proof Trivial, since each application of **(Beta)** in a term reduces the number of **(Beta)** redexes in that term by one. □

Finally, we consider the combination of rules **(Beta)** and **(DΛ)**:

Theorem 5.3 *The subsystem of* **ΛCCL** *composed of rules* **(Beta)** *and* **(DΛ)** *is noetherian.*

Proof We use Kamin and Lévy's (non-extended) recursive path ordering technique [Hue86, p. 55]. We once again begin by defining (the strict part of) a quasi-ordering $\succeq_o$ on the operators of **ΛCCL** as follows:

$$\mathrm{Apply}(\cdot,\,\cdot) \succ_o [\cdot,\,\cdot] \succ_o \Lambda(\cdot) \succ_o \text{(all other operators)}$$

We now extend $\succ_o$ to a simplification ordering $\succeq_t$ as follows:

$$A \equiv f(s_1, \ldots, s_m) \;\succeq\; B \equiv g(t_1, \ldots, t_n)$$

if

$$f \succ_o g \;\text{ and }\; A \succ t_i \quad \text{for all} \quad i \in 1 \ldots n$$

or

$$s_i \succeq B \quad \text{for some} \quad i \in 1 \ldots m$$

[1]The author gratefully acknowledges the assistance of Thérèse Hardin in pointing out flaws in a previous attempt at a simplified proof of this theorem.

or

$$f \approx_o g, \quad (s_1, \ldots, s_m) \succeq_* (t_1, \ldots t_n), \text{ and } A \succ t_i \text{ for all } i \in 1 \ldots n$$

where $\succeq_*$ is once again the lexicographic extension of $\succeq$ to sequences. It is very easy to verify that for rule **(Beta)**, we have for all $A, B \in \mathbf{\Lambda CCL}$,

$$\text{Apply}(\Lambda(A),\, B) \;\succ\; [A,\, \langle \emptyset,\, B \rangle]$$

and for rule **(DΛ)**, we have for all $A, E \in \mathbf{\Lambda CCL}$

$$[\Lambda(A),\, E] \;\succ\; \text{Apply}([A,\, E],\, [B,\, E])$$

Since $\succeq$ is a simplification ordering, this implies that if $A \longrightarrow_{(\text{Beta})} B$ or $A \longrightarrow_{(\text{D}\Lambda)} B$, then $A \succ B$ and thus that the combination of rules **(Beta)** and **(DΛ)** is noetherian. □

The combination of **(Beta)** and **(DΛ)** are useful for simulating *developments*, since they are sufficient to simulate all the β-contractions in a given term without allowing any of the substitutions that would create new **(Beta)** redexes.

5.3.2 Normal Forms

It will be useful in the sequel to define several normal-form subsets of $\mathbf{\Lambda CCL}$ terms (technically, some of these sets of terms are normal forms only with respect to a subrelation of $\mathbf{\Lambda CCL}$):

Definition 5.6 (*LNF*)
The set of lambda normal forms *(LNF) is a subset of* $GTer_{\mathcal{L}}(\Lambda\text{CCL})$*, defined inductively as follows:*

$$\begin{array}{rcl} n! \in LNF & & \\ A \in LNF & \Longrightarrow & \Lambda(A) \in LNF \\ A \in LNF, B \in LNF & \Longrightarrow & \text{Apply}(A,\, B) \in LNF \end{array}$$

Lambda normal forms are intuitively those terms that "look like" terms of the (de Bruijn) lambda calculus.

Theorem 5.4 *All elements of* $GTer_{\mathcal{L}}(\Lambda\text{CCL})$ *are reducible to a lambda normal form, using the rules of* **ECCL**, *i.e., for all* $A \in GTer_{\mathcal{L}}(\Lambda\text{CCL})$, *there exists* $B \in LNF$ *such that*

$$A \quad \twoheadrightarrow_{\text{ECCL}} \quad B$$

Proof Simply note that any term of sort $\mathcal{L}$ that is not in *LNF* contains an **ECCL** redex. Keep reducing such redexes using rules of **ECCL** until a term in *LNF* is reached, which must happen eventually since **ECCL** is noetherian. □

We now define some normal forms for environments:

Definition 5.7 (*PENF*, *ENF*)
The set of partial environment normal forms *(PENF) is a subset of* $G\mathrm{Ter}_{\mathcal{E}}(\Lambda\mathrm{CCL})$, *defined inductively as follows:*

$$\begin{aligned} \emptyset &\in PENF \\ \square^n &\in PENF \\ E \in PENF \quad &\Longrightarrow \quad \langle E,\, A\rangle \in PENF \end{aligned}$$

If $E \in PENF$, then we define the set of elements of E, *notated* $\mathrm{elt}(E)$ *inductively by:*

$$\mathrm{elt}(E) \quad \triangleq \quad \begin{cases} elt(E') \cup A & E = \langle E',\, A\rangle \\ \emptyset & otherwise \end{cases}$$

The set of full environment normal forms *(ENF) is a subset of PENF with the property that $E \in PENF$ iff for all $A \in \mathrm{elt}(E)$, $A \in LNF$.*

Theorem 5.5 *All elements of* $G\mathrm{Ter}_{\mathcal{E}}(\Lambda\mathrm{CCL})$ *are reducible to a (full or partial) environment normal form using the rules of* **ECCL**, *i.e., for all* $E_1 \in G\mathrm{Ter}_{\mathcal{E}}(\Lambda\mathrm{CCL})$, *there exists* $E_2 \in PENF$ *such that*

$$E_1 \quad \twoheadrightarrow_{\mathrm{ECCL}} \quad E_2$$

Proof Once again, we can observe that every term of sort $\mathcal{E}$ that is not in *PENF* or *ENF* must contain an **ECCL** redex. Such redexes can be reduced until the normal form is reached. □

The following less restrictive normal form for environments will also be useful:

Definition 5.8 (*WPENF*)
The set of weak partial environment normal forms *(WPENF) is a subset of* $G\mathrm{Ter}_{\mathcal{E}}(\Lambda\mathrm{CCL})$, *defined as follows:*

$$\begin{aligned} \emptyset &\in WPENF \\ \square^n &\in WPENF \\ \langle E,\, A\rangle &\in WPENF \end{aligned}$$

We now define the set of environment terms which will be produced by the process of simulated λ-reduction:

Definition 5.9 (*LEF*)
The set of lambda environment forms *(LEF) is a subset of* $G\mathrm{Ter}_{\mathcal{E}}(\Lambda\mathrm{CCL})$, *defined*

inductively as follows:

$$\begin{array}{rcl} \emptyset \in LEF & & \\ \square \in LEF & & \\ E \in LEF & \Longrightarrow & \langle E,\ A\rangle \in LEF \\ E_1, E_2 \in LEF,\ E_1 \neq \emptyset & \Longrightarrow & E_1 \circ E_2 \in LEF \end{array}$$

We then have the following:

Theorem 5.6 *All elements of LEF are reducible to a weak partial environment normal form using the rules of* **ECCL** *without rule* **(NullEL)**, *i.e., for all $E_1 \in$ LEF, there exists $E_2 \in$ WPENF such that*

$$E_1 \quad \twoheadrightarrow_{\mathrm{ECCL-(NullEL))}} \quad E_2$$

Proof We observe that for every term E_1' in *LEF* not in *WPENF*, there is a rule (R) of **ECCL−(NullEL)** that is applicable to E_1' such that

$$E_1' \quad \twoheadrightarrow_{(R)} \quad E_1''$$

and $E_1'' \in LEF$. Since **ECCL** is noetherian, **ECCL−(NullEL)** is noetherian as well, and therefore the reduction of E_1 must terminate with a term $E_2 \in WPENF$. □

The following two definitions allow us to define the **ΛCCL** analogue of a weak head normal form ($WHNF_\beta$) in the lambda calculus:

Definition 5.10 (*AF*)
The set of abstraction forms *(AF) is a subset of $GTer_{\mathcal{L}}(\Lambda\mathrm{CCL})$, defined inductively as follows:*

$$\begin{array}{rcl} \Lambda(A) \in AF & & \\ A \in AF & \Longrightarrow & [A,\ E] \in AF \end{array}$$

Definition 5.11 ($WHNF_{\Lambda\mathrm{CCL}}$)
The set of weak head normal forms *($WHNF_{\Lambda\mathrm{CCL}}$) is a subset of* $\mathrm{Ter}(\Lambda\mathrm{CCL})$, *defined inductively as follows:*

$$\begin{array}{rcl} n! \in WHNF_{\Lambda\mathrm{CCL}} & & \\ A \in AF & \Longrightarrow & A \in WHNF_{\Lambda\mathrm{CCL}} \\ A \in WHNF_{\Lambda\mathrm{CCL}}, A \notin AF & \Longrightarrow & \mathrm{Apply}(A,\ B) \in WHNF_{\Lambda\mathrm{CCL}} \end{array}$$

5.3.3 Confluence of Subsystems

We are now in a position to show that both **ECCL** and **(Beta)** are confluent subsystems.

In order to show **ECCL** confluent, the following definition and lemma will be required:

Definition 5.12 *The terms* Null^i *are defined by*

$$\begin{array}{rcl} \mathrm{Null}^0 & \equiv & \emptyset \\ \mathrm{Null}^i & \equiv & \langle\langle\cdots\langle\langle\Box^i,\ i!\rangle,\ (i-1)!\rangle,\cdots 1!\rangle,\ 0!\rangle \quad (i>0) \end{array}$$

The terms Null^i are behaviorally equivalent to $\emptyset$, in the sense that for any context $C[\ \ ]{:}\,\mathcal{L}$, $C[\mathrm{Null}^i]$ is reducible to the same term as $C[\emptyset]$. In particular, we have the following:

Lemma 5.7 *For all* $A \in GTer_{\mathcal{L}}(\Lambda\mathrm{CCL})$ *and* n*, there exists* B *such that*

$$[A,\ \mathrm{Null}^n] \quad \twoheadrightarrow_{\mathrm{ECCL}} \quad B \quad \twoheadleftarrow_{\mathrm{ECCL}} \quad A$$

Proof By Theorem 5.4, we know that there exists $A' \in LNF$ such that

$$A \twoheadrightarrow_{\mathrm{ECCL}} A'$$

We can then show

$$[A',\ \mathrm{Null}^n] \twoheadrightarrow_{\mathrm{ECCL}} A'$$

by structural induction on A'. Since $A' \in LNF$, there are only three cases:

Case 1: $A' \equiv \mathrm{Apply}(B,\ C)$. We then have

$$[\mathrm{Apply}(B,\ C),\ \mathrm{Null}^n] \longrightarrow_{\mathrm{ECCL}} \mathrm{Apply}([B,\ \mathrm{Null}^n],\ [C,\ \mathrm{Null}^n])$$

and use the induction hypothesis on $[B,\ \mathrm{Null}^n]$ and $[C,\ \mathrm{Null}^n]$.

Case 2: $A' \equiv \Lambda(B)$. It is then easy to see that

$$[\Lambda(B),\ \mathrm{Null}^n] \twoheadrightarrow_{\mathrm{ECCL}} \Lambda([B,\ \mathrm{Null}^{n+1}])$$

We then use the induction hypothesis on $[B,\ \mathrm{Null}^{n+1}]$.

Case 3: $A' \equiv i!$. We then have

$$[i!,\ \mathrm{Null}^n] \twoheadrightarrow_{\mathrm{ECCL}} i!$$

by a simple induction on n.

We thus have

$$[A,\ \mathrm{Null}^n] \quad \twoheadrightarrow_{\mathrm{ECCL}} \quad [A',\ \mathrm{Null}^n] \quad \twoheadrightarrow_{\mathrm{ECCL}} \quad A' \quad \twoheadleftarrow_{\mathrm{ECCL}} \quad A$$

which completes the proof. $\Box$

Theorem 5.8 **ECCL** *is confluent (thus Church-Rosser) on ground terms i.e., if* $A \twoheadrightarrow_{\mathrm{ECCL}} B_1$ *and* $A \twoheadrightarrow_{\mathrm{ECCL}} B_2$*, then there exists* C *such that* $B_1 \twoheadrightarrow_{\mathrm{ECCL}} C$ *and* $B_2 \twoheadrightarrow_{\mathrm{ECCL}} C$.

Proof Since **ECCL** is noetherian (by Theorem 5.1), to show confluence, we need only prove that it is weakly confluent. It then suffices to show that each critical pair has a common reduct. This is straightforward, except for the critical pair induced by the rules **(DΛ)** and **(NullC)**, for which we must show

$$[A,\ \langle\square,\ \mathrm{Var}\rangle] \twoheadrightarrow_{\mathrm{ECCL}} B \twoheadleftarrow_{\mathrm{ECCL}} A$$

for some B. This follows from Lemma 5.7 above (since $\langle\square,\ \mathrm{Var}\rangle \equiv \mathrm{Null}^1$). $\square$

Since **ECCL** is noetherian, we can make the following definitions:

Definition 5.13 (*LNF*(), *ENF*())
For any term $A \in GTer_{\mathcal{L}}(\Lambda\mathrm{CCL})$*, we will refer to the term* A' *such that*

$$A \quad \twoheadrightarrow_{\mathrm{ECCL}} \quad A'$$

and $A \in LNF$ *by* $LNF(A)$. *Since* **ECCL** *is confluent and noetherian,* $LNF(A)$ *is unique.*

For any term $E \in GTer_{\mathcal{E}}(\Lambda\mathrm{CCL})$*, we will refer to the term* E' *such that*

$$E \quad \twoheadrightarrow_{\mathrm{ECCL}} \quad E'$$

and $E' \in ENF$ *by* $ENF(E)$. $ENF(E)$ *is also unique.*

We also have the following:

Theorem 5.9 **(Beta)** *is confluent, i.e., if* $A \twoheadrightarrow_{(\mathrm{Beta})} B_1$ *and* $A \twoheadrightarrow_{(\mathrm{Beta})} B_2$*, then there exists* C *such that* $B_1 \twoheadrightarrow_{(\mathrm{Beta})} C$ *and* $B_2 \twoheadrightarrow_{(\mathrm{Beta})} C$.

Proof **(Beta)** has no critical pairs, thus it is trivially weakly confluent. Since it is also noetherian (by Theorem 5.2), it is confluent. $\square$

5.3.4 Confluence of ΛCCL

We can now show **ΛCCL** confluent by a technique similar to the classic Tait/Martin-Löf proof of the Church-Rosser property for the lambda calculus. The following reduction relation will be useful:

Definition 5.14

$$\rightarrow_{\mathrm{Dev}} \;\equiv\; \twoheadrightarrow_{\mathrm{ECCL}} \cdot \twoheadrightarrow_{(\mathrm{Beta})} \cdot \twoheadrightarrow_{\mathrm{ECCL}}$$

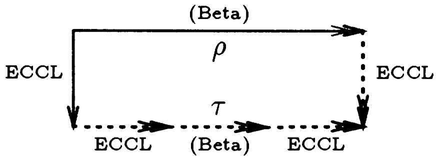
where '$\cdot$' denotes relational composition.

$\longrightarrow_{\mathrm{Dev}}$ is intended to correspond roughly to the notion of a *development* in the lambda calculus. As usual, $\twoheadrightarrow_{\mathrm{Dev}}$ represents the reflexive, transitive closure of $\longrightarrow_{\mathrm{Dev}}$.

First, we need the following sequence of lemmas, each represented as a commuting diagram (dotted arrows denote reductions existentially dependent on the arbitrary reductions represented by solid arrows).

The proof of the theorem hinges on the following lemma:

Lemma 5.10 *(given by the diagram below)*

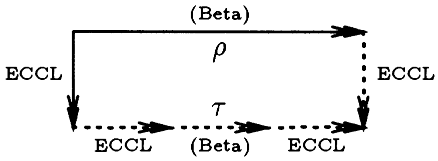


Let ρ and τ be the **(Beta)** *reductions as marked in the diagram above. Then if the redexes in ρ are disjoint, the redexes in τ are also disjoint.*

Proof See appendix A.1. □

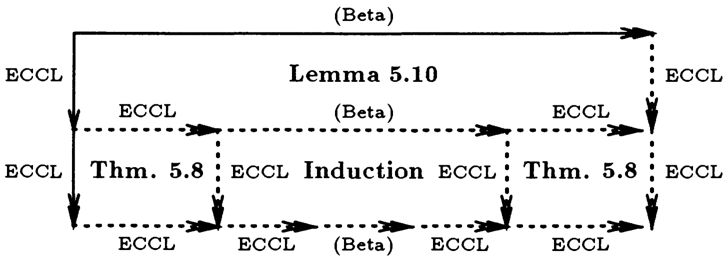
The following lemma states that **(Beta)** and **ECCL** *almost* commute, however, the intervention of additional **ECCL** reductions may be required:

Lemma 5.11 *(given by the diagram below)*



Let ρ and τ be the **(Beta)** *reductions as marked in the diagram above. Then if the redexes in ρ are disjoint, the redexes in τ are also disjoint.*

Proof Follows by noetherian induction on the left-hand **ECCL** reduction using Lemma 5.10 and Theorem 5.8. The required construction is given by the following diagram:



□

The following lemma is also needed:

Lemma 5.12 *(given by the diagram below)*

$$\begin{tikzcd}
{} \arrow[r, two heads, "\text{ECCL}"] \arrow[d, two heads, "\text{ECCL}"'] & {} \arrow[r, two heads, "\text{(Beta)}"] & {} \arrow[r, two heads, "\text{ECCL}"] & {} \arrow[d, dotted, two heads, "\text{ECCL}"] \\
{} \arrow[r, dotted, two heads, "\text{ECCL}"'] & {} \arrow[r, dotted, two heads, "\text{(Beta)}"'] & {} \arrow[r, dotted, two heads, "\text{ECCL}"'] & {}
\end{tikzcd}$$

Proof The required diagram is constructed as follows:

$$\begin{tikzcd}
{} \arrow[r, two heads, "\text{ECCL}"] \arrow[d, two heads, "\text{ECCL}"'] \arrow[dr, phantom, "\textbf{Thm. 5.8}"] & {} \arrow[rrr, two heads, "\text{(Beta)}"] \arrow[d, dotted, two heads, "\text{ECCL}"] \arrow[drrr, phantom, "\textbf{Lemma 5.11}"] & & & {} \arrow[r, two heads, "\text{ECCL}"] \arrow[d, dotted, two heads, "\text{ECCL}"'] \arrow[dr, phantom, "\textbf{Thm. 5.8}"] & {} \arrow[d, dotted, two heads, "\text{ECCL}"] \\
{} \arrow[r, dotted, two heads, "\text{ECCL}"'] & {} \arrow[r, dotted, two heads, "\text{ECCL}"'] & {} \arrow[r, dotted, two heads, "\text{(Beta)}"'] & {} \arrow[r, dotted, two heads, "\text{ECCL}"'] & {} \arrow[r, dotted, two heads, "\text{ECCL}"'] & {}
\end{tikzcd}$$

□

The next lemma shows that **ECCL–(Beta)–ECCL** sequences commute:

Lemma 5.13 *(given by the diagram below)*

$$\begin{tikzcd}
{} \arrow[r, two heads, "\text{ECCL}"] \arrow[d, two heads, "\text{ECCL}"'] & {} \arrow[r, two heads, "\text{(Beta)}"] & {} \arrow[r, two heads, "\text{ECCL}"] & {} \arrow[d, dotted, two heads, "\text{ECCL}"] \\
{} \arrow[d, two heads, "\text{(Beta)}"'] & & & {} \arrow[d, dotted, two heads, "\text{(Beta)}"] \\
{} \arrow[d, two heads, "\text{ECCL}"'] & & & {} \arrow[d, dotted, two heads, "\text{ECCL}"] \\
{} \arrow[r, dotted, two heads, "\text{ECCL}"'] & {} \arrow[r, dotted, two heads, "\text{(Beta)}"'] & {} \arrow[r, dotted, two heads, "\text{ECCL}"'] & {}
\end{tikzcd}$$

Proof Follows by noetherian induction on the left-hand **ECCL** reduction, using Lemmas 5.11 and 5.12 and Theorem 5.8 as base cases. The construction in Figure 5.1 is used. □

We then have:

Lemma 5.14 $\longrightarrow_{\text{Dev}}$ *is confluent on ground terms, i.e.,*

$$\begin{tikzcd}
{} \arrow[r, two heads, "\text{Dev}"] \arrow[d, two heads, "\text{Dev}"'] & {} \arrow[d, dotted, two heads, "\text{Dev}"] \\
{} \arrow[r, dotted, two heads, "\text{Dev}"'] & {}
\end{tikzcd}$$

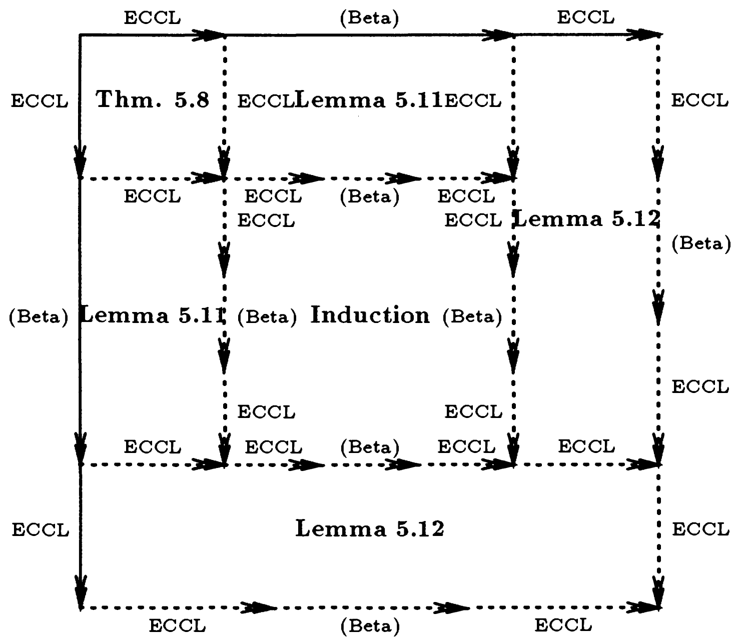


Figure 5.1: Construction for Lemma 5.13

Proof The reductions used in Lemma 5.13 are $\longrightarrow_{\text{Dev}}$ contractions, and the theorem thus follows by induction on the lengths of the reductions in the premise. □

Theorem 5.15 **ΛCCL** *is confluent on ground terms.*

Proof $\twoheadrightarrow_{\text{Dev}}$ and $\twoheadrightarrow_{\Lambda\text{CCL}}$ are relationally equivalent. Thus from Lemma 5.14, we must conclude that $\twoheadrightarrow_{\Lambda\text{CCL}}$ is confluent. □

The principal idea used in the proof above is that one can always find a common reduct for two diverging **(Beta)** reductions surrounded by **ECCL** reductions by using reductions of the same sort. Intuitively, *new* **(Beta)** redexes can only be created by intervening **ECCL** reductions. Infinite **ΛCCL** reductions can thus only occur when an infinite number of (necessarily) finite **ECCL** or **(Beta)** subreductions are alternated.

Theorem 5.15 is a principal result; Curien did not exhibit a confluent system strong enough to model arbitrary reductions in the lambda calculus. However, independent work of Hardin [Har87,Har89] and Yokouchi [Yok89] has led to a characterization of *subsets* of Curien's original **CCL** terms for which confluence of the system **CCL**β can be proven. By contrast, the 2-sorted term structure of **ΛCCL** rules out the construction of "uninteresting" terms that Hardin and Yokouchi's **CCL** subsets explicitly omit.

Yokouchi's technique for proving the confluence of **CCLβ** on subsets of terms is similar to the confluence proof given here. Hardin's proof of confluence relies on the confluence of a subsystem of **CCLβ** that simulates β-reduction.

Hardin and Yokouchi's proofs of confluence both rely on the fact that a "substitutive" subset of **CCL**, similar to **ECCL**, is noetherian. This was shown to be the case by Hardin and Laville [HL86] using an ingenious, but complicated term ordering. The proof of Theorem 5.1 is considerably simpler.

5.4 Conservative Extension

We denote the *ground theory* (i.e., the set of equivalences on ground terms) of elements of $GTer_{\mathcal{L}}(\Lambda\mathrm{CCL})$ by **ΛCCL($\mathcal{L}$)**. We can now show that **ΛCCL($\mathcal{L}$)** is equivalent to the theory of β-reduction on elements of $GTer(\lambda^{\mathrm{DB}})$.

5.4.1 Translation

We begin by defining a translation between terms of the de Bruijn lambda calculus and terms of **ΛCCL**.

Definition 5.15 ($[\![\cdot]\!]_{\Lambda\mathrm{CCL}}$)
*For any term $M \in GTer(\lambda^{\mathrm{DB}})$, we can define a corresponding term $[\![M]\!]_{\Lambda\mathrm{CCL}} \in$ **ΛCCL** inductively as follows:*

$$\begin{aligned} [\![i]\!]_{\Lambda\mathrm{CCL}} &= i! \\ [\![(\lambda.N)]\!]_{\Lambda\mathrm{CCL}} &= \Lambda([\![N]\!]_{\Lambda\mathrm{CCL}}) \\ [\![(N_1 N_2)]\!]_{\Lambda\mathrm{CCL}} &= \mathrm{Apply}([\![N_1]\!]_{\Lambda\mathrm{CCL}}, [\![N_2]\!]_{\Lambda\mathrm{CCL}}) \end{aligned}$$

The reverse transformation, $[\![\cdot]\!]_{\lambda}$, is defined in the obvious way on members of *LNF*. $[\![\cdot]\!]_{\Lambda\mathrm{CCL}}$ is clearly a monomorphic mapping from λ-trees to **ΛCCL**-trees.

5.4.2 Extension Theorems

The following lemma is proved by Curien for **CCLβ** [Cur86a]; an isomorphic construction works for **ΛCCL**:

Lemma 5.16 ([Cur86a]) *Let M and N be elements of $GTer(\lambda^{\mathrm{DB}})$ such that $M \longrightarrow_{\beta} N$. Then*

$$[\![M]\!]_{\Lambda\mathrm{CCL}} \twoheadrightarrow_{\Lambda\mathrm{CCL}} [\![N]\!]_{\Lambda\mathrm{CCL}}$$

The **ΛCCL** equivalent of Curien's simulation of β-reduction in **CCLβ** has the following property:

Corollary 5.17 *Let M and N be elements of $G\mathrm{Ter}(\lambda^{\mathrm{DB}})$ such that*

$$M \longrightarrow_\beta N$$

Then there exists $B \in G\mathrm{Ter}(\Lambda\mathrm{CCL})$ such that

$$[\![M]\!]_{\Lambda\mathrm{CCL}} \xrightarrow{\rho}_{(\mathbf{Beta})} B$$

and

$$B \overset{\sigma}{\twoheadrightarrow}_{\mathrm{ECCL}} [\![N]\!]_{\Lambda\mathrm{CCL}}$$

σ effectively carries out the substitution required by a β-contraction.

We then have the following:

Lemma 5.18 *Let A, B, and C be elements of $G\mathrm{Ter}(\Lambda\mathrm{CCL})$ such that $A, C \in$ LNF,*

$$A \overset{\rho}{\twoheadrightarrow}_{(\mathbf{Beta})} B$$

and

$$B \overset{\sigma}{\twoheadrightarrow}_{\mathrm{ECCL}} C$$

where the redexes in ρ are disjoint. Then

$$[\![A]\!]_\lambda \twoheadrightarrow_\beta [\![C]\!]_\lambda$$

Proof Let $R_1, R_2, \ldots, R_n$ be the set of redexes contracted in ρ. Consider the set of simulated β-reductions $\gamma_i = \rho_i \tau_i$ carried out on **(Beta)** redexes R_i as follows:

$$\begin{array}{lll} \rho_i\colon & A_{i-1} & \xrightarrow{R_i}_{(\mathbf{Beta})} & B_i \\ & B_i & \overset{\tau_i}{\twoheadrightarrow}_{\mathrm{ECCL}} & A_i \end{array}$$

where $A_0 \equiv A$ and τ_i is the **ECCL** reduction given by Lemma 5.17 that simulates the substitution operation within R_i.

So we have for all i

$$[\![A_{i-1}]\!]_\lambda \longrightarrow_\beta [\![A_i]\!]_\lambda$$

Concatenating the reductions gives us

$$[\![A]\!]_\lambda \twoheadrightarrow_\beta [\![A_n]\!]_\lambda$$

Since the reductions γ_i are disjoint, we can trivially reorder their components as follows:

Let $\tau = \tau_1 \tau_2 \cdots \tau_n$. We then have

$$B \overset{\tau}{\twoheadrightarrow}_{\mathrm{ECCL}} A_n$$

Recalling that $\rho = \rho_1\rho_2\cdots\rho_n$, we have

$$\begin{array}{lll} A & \xrightarrow{\rho}\!\!\!\!\rightarrow_{\text{(Beta)}} & B \\ B & \xrightarrow{\tau}\!\!\!\!\rightarrow_{\text{ECCL}} & A_n \end{array}$$

We now note that the final term C of reduction σ in the premise of the lemma, and the final term A_n of reduction τ above, are terms in *LNF*. Thus, since **ECCL** is confluent and σ and τ are coinitial, we must have

$$A_n \equiv C$$

from which we get

$$\llbracket A \rrbracket_\lambda \twoheadrightarrow_\beta \llbracket C \rrbracket_\lambda$$

□

We can now prove the converse of Lemma 5.16:

Lemma 5.19 *Let A be an element of $GTer_{\mathcal{L}}(\Lambda\text{CCL})$, and*

$$A \twoheadrightarrow_{\Lambda\text{CCL}} B$$

Then

$$\llbracket LNF(A) \rrbracket_\lambda \twoheadrightarrow_\beta \llbracket LNF(B) \rrbracket_\lambda$$

Proof Let $A_0 \equiv A$ and $A_n \equiv B$. Divide the **ΛCCL** reduction into subreductions alternating (possibly null) **ECCL** reductions and **(Beta)** contractions as follows:

$$\begin{array}{l} A \equiv A_0 \twoheadrightarrow_{\text{ECCL}} \hat{A}_0 \longrightarrow_{\text{(Beta)}} A_1 \longrightarrow_{\text{(Beta)}} \cdots \\ \quad \longrightarrow_{\text{(Beta)}} A_{n-1} \twoheadrightarrow_{\text{ECCL}} \hat{A}_{n-1} \longrightarrow_{\text{(Beta)}} A_n \equiv B \end{array}$$

Note that since **ECCL** is confluent,

$$LNF(A_i) \equiv LNF(\hat{A}_i)$$

We can then show that

$$\begin{array}{l} \llbracket LNF(A) \rrbracket_\lambda \equiv \llbracket LNF(A_0) \rrbracket_\lambda \twoheadrightarrow_\beta \llbracket LNF(A_1) \rrbracket_\lambda \twoheadrightarrow_\beta \cdots \\ \quad \twoheadrightarrow_\beta \llbracket LNF(A_{n-1}) \rrbracket_\lambda \twoheadrightarrow_\beta \llbracket LNF(A_n) \rrbracket_\lambda \equiv \llbracket LNF(B) \rrbracket_\lambda \end{array}$$

using Lemma 5.11, Lemma 5.18, and the construction in the following diagram:

$$\begin{tikzcd}[column sep=small]
A_0 \equiv A \arrow[rr, two heads, "\text{ECCL}"] \arrow[dr, two heads, "\text{ECCL}"'] & & \hat{A}_0 \arrow[dl, two heads, "\text{ECCL}"] \arrow[rr, "\text{(Beta)}"] \arrow[drr, phantom, "\textbf{Lemma 5.11}"] & & A_1 \arrow[d, two heads, "\text{ECCL}"] \arrow[r, dotted, no head] & A_{n-1} \arrow[dr, two heads, "\text{ECCL}"'] \arrow[rr, two heads, "\text{ECCL}"] & & \hat{A}_{n-1} \arrow[dl, two heads, "\text{ECCL}"] \arrow[rr, "\text{(Beta)}"] \arrow[drr, phantom, "\textbf{Lemma 5.11}"] & & A_n \equiv B \arrow[d, two heads, "\text{ECCL}"] \\
& LNF(A_0) \arrow[rrr, dashed, two heads, "\text{(Beta)}" near start, "\text{ECCL}" near end] \arrow[d, phantom] & & & LNF(A_1) \arrow[r, dotted, no head] & LNF(A_{n-1}) \arrow[rrrr, dashed, two heads, "\text{(Beta)}" near start, "\text{ECCL}" near end] & & & & LNF(A_n) \\
& \begin{array}{c}\llbracket LNF(A_0) \rrbracket_\lambda \\ \equiv \llbracket LNF(A) \rrbracket_\lambda\end{array} \arrow[rrr, two heads, "\textbf{Lemma 5.18}", "\beta"'] & & & \llbracket LNF(A_1) \rrbracket_\lambda \arrow[r, dotted, no head] & \llbracket LNF(A_{n-1}) \rrbracket_\lambda \arrow[rrrr, two heads, "\textbf{Lemma 5.18}", "\beta"'] & & & & \begin{array}{c}\llbracket LNF(A_n) \rrbracket_\lambda \\ \equiv \llbracket LNF(B) \rrbracket_\lambda\end{array}
\end{tikzcd}$$

□

Putting the results from Lemma 5.16 and Lemma 5.19 together yields:

Theorem 5.20 $\mathbf{\Lambda CCL}(\mathcal{L})$ *is a monomorphic extension of* β*, and* β *is a monomorphic extension of* $\mathbf{\Lambda CCL}(\mathcal{L})$.

Proof We have from Lemmas 5.16 and 5.19 that for all $M, N \in GTer(\lambda^{\mathrm{DB}})$,

$$M \twoheadrightarrow_\beta N \quad \text{iff} \quad [\![M]\!]_{\Lambda\mathrm{CCL}} \twoheadrightarrow_{\Lambda\mathrm{CCL}} [\![N]\!]_{\Lambda\mathrm{CCL}}$$

and

$$[\![M]\!]_{\Lambda\mathrm{CCL}} \twoheadrightarrow_{\Lambda\mathrm{CCL}} B \quad \text{implies} \quad B \twoheadrightarrow_{\Lambda\mathrm{CCL}} [\![P]\!]_{\Lambda\mathrm{CCL}}$$

for some $P \in GTer(\lambda^{\mathrm{DB}})$. We also have for all $A, B \in GTer(\Lambda\mathrm{CCL}(\mathcal{L}))$

$$A \twoheadrightarrow_{\Lambda\mathrm{CCL}} B \quad \text{iff} \quad [\![LNF(A)]\!]_\lambda \twoheadrightarrow_\beta [\![LNF(B)]\!]_\lambda$$

From these facts, the result follows immediately. □

This result shows that *any* reduction of a $\mathbf{\Lambda CCL}$ term $A \in LNF$ effectively simulates a reduction in the lambda calculus. In [Har89], Hardin proves a similar result for Curien's Categorical Combinators. However, the proof there is considerably complicated by the presence in **CCL** of computationally uninteresting terms. Hardin was, however, able to show that in fact such terms do not arise in the process of simulating β-reduction.

We then have the following corollaries to Theorem 5.20:

Corollary 5.21 $\mathbf{\Lambda CCL}(\mathcal{L})$ *is a conservative extension of* β *and* β *is a conservative extension of* $\mathbf{\Lambda CCL}(\mathcal{L})$.

Proof Immediate by Lemma 3.6 since β and $\mathbf{\Lambda CCL}$ are confluent. □

Corollary 5.22

$$\langle \mathbf{\Lambda CCL},\ [\![LNF(\cdot)]\!]_\lambda,\ [\![\cdot]\!]_{\Lambda\mathrm{CCL}} \rangle$$

is an complete implementation *of* β *(on* $GTer(\lambda^{\mathrm{DB}})$*) in the sense of Definition 3.24.*

Proof Immediate from the definitions and Theorem 5.20. □

The construction of Lemma 5.19 shows that a single **(Beta)** contraction can effectively simulate multiple β-contractions. It is this property, together with the fact that the substitution operation can be deferred except when necessary to yield an outermost redex, that makes an environment-based interpreter an attractive implementation tool for the lambda calculus.

Any reduction scheme for the lambda calculus implemented using **ΛCCL** would have to perform **ECCL** reductions as well as **(Beta)** contractions, but it is not unreasonable to count the former as "overhead," if we ensure that the number of **ECCL** reductions required is at worst proportional to the number of **(Beta)** reductions and the size of the initial term (which is not difficult to do).

The following example illustrates how the reductions of a λ-term and its **ΛCCL** equivalent compare:

Example 5.3 Let $M \equiv \lambda y.((\lambda x.x)y)$. We then have

$$M \twoheadrightarrow_{\beta} \lambda y.y$$

The equivalent term in **ΛCCL**, after encoding variables, is given by

$$[\![M]\!]_{\Lambda\mathrm{CCL}} \equiv \Lambda(\mathrm{Apply}(\Lambda(0!),\ 0!)) \equiv \Lambda(\mathrm{Apply}(\Lambda(\mathrm{Var}),\ \mathrm{Var}))$$

We then have

$$\begin{aligned} &\Lambda(\mathrm{Apply}(\Lambda(\mathrm{Var}),\ \mathrm{Var})) \\ &\qquad \longrightarrow_{(\mathbf{Beta})} \Lambda([\mathrm{Var},\ \langle\emptyset,\ \mathrm{Var}\rangle]) \\ &\qquad \longrightarrow_{(\mathbf{VarRef})} \Lambda(\mathrm{Var}) \end{aligned}$$

and

$$[\![\Lambda(\mathrm{Var})]\!]_{\lambda} \equiv \lambda y.y$$

In essence, **ΛCCL** is just a formalization of the informal notions of closure and environment given in the introduction, coupled with a mechanism for indexing environments.

5.4.3 Weak Head Redex

It will be important to be able to single out the *weak head* **(Beta)** or **(Beta′)** redex of any **ΛCCL** term. The weak head **(Beta)** or **(Beta′)** redex will play role similar to that of the weak head redex in a λ-term—in order to produce a weak head normal form, the weak head redex must always be contracted. We thus make the following definition:

Definition 5.16 (Weak Head Redex)
Let A be an element of $G\mathrm{Ter}_{\mathcal{L}}(\Lambda\mathrm{CCL})$. Then the weak head **(Beta)** *or* **(Beta′)**

redex of A, notated whead(A), is the subterm of A defined inductively as follows:

$$whead(A) \;=\; \begin{cases} A & A \text{ is a } \textbf{(Beta)} \text{ or } \textbf{(Beta}'\textbf{)} \text{ redex} \\ whead(B) & A = \text{Apply}(B,\, C), \\ & \quad A \text{ is not a } \textbf{(Beta)} \text{ or } \textbf{(Beta}'\textbf{)} \text{ redex} \\ whead(B) & A = [\text{Var},\, \langle E,\, B\rangle] \\ \perp & \text{otherwise} \end{cases}$$

If $whead(A) \neq \perp$, *then we say that* A has *a weak head redex.*

The only interesting case in Definition 5.16 that distinguishes it from the definition of weak head redex in a λ-term is the last one. Since we have

$$[\text{Var},\, \langle E,\, B\rangle] \quad \longrightarrow_{(\text{VarRef})} \quad B$$

we allow the weak head redex of the element of the environment extracted by rule **(VarRef)** (if it exists) to be considered the weak head redex of the entire term. It would then immediately become the weak head redex upon application of rule **(VarRef)**.

We have the following lemma:

Lemma 5.23 *Let* A *be a term of* $GTer_{\mathcal{L}}(\mathbf{\Lambda CCL})$. *Then if* A *has a weak head redex,* $[\![LNF(A)]\!]_\lambda$ *has a weak head redex.*

Proof We proceed by structural induction on A. If A has a weak head redex and $A \neq [\text{Var},\, \langle E,\, B\rangle]$, then we note that there exists context C and terms D, D' such that C has a weak head redex,

$$A \;=\; C[D]$$

and

$$LNF(A) \;=\; C[D']$$

Thus $LNF(A)$ also has a weak head redex. But then by the definition of $[\![\cdot]\!]_\lambda$, $[\![LNF(A)]\!]_\lambda$ also has a weak head redex.

If $A = [\text{Var},\, \langle B,\, E\rangle]$, then we note that

$$LNF(A) \;=\; LNF(B)$$

Thus $LNF(A)$ has a weak head redex if and only if $LNF(B)$ does, and the result follows by induction. $\square$

5.5 A Simple Lambda Interpreter

To illustrate the utility of **ΛCCL** both as a notation for specifying manipulations of environments, closures, and λ-terms and as a vehicle for implementing graph reduction, we will describe simple interpreter for the lambda calculus: *rwhnf*(). *rwhnf*() is an algorithm that transforms a term of the form $[A,\ E]$ to the **ΛCCL** equivalent of weak head normal form, $\mathit{WHNF}_{\Lambda\mathrm{CCL}}$ (Definition 5.11).

5.5.1 A Graph Rewriting Language

It will be convenient to use a notation for specifying graph rewriting strategies that is similar to that used in a number of strict functional programming languages, e.g., ML. However, since we are interested in specifying *rewriting* strategies, the rewriting operation is not itself functional—we are explicitly substituting one subgraph for another, and will thus make the sequencing of certain transformations explicit. The algorithms given below should be treated as recursive, applicative-order specifications of sequences of term graph redex contractions. Our algorithmic notation is as follows:

- The left hand side of a term graph rewriting rule will be specified using contexts, where holes are labeled with unique identifiers rather than integers.
- The statement "**case** A **of** ..." causes one of a set of sub-statements to be executed depending on which of a (non-overlapping) set of graph redex left hand sides matches the downward closure of the subgraph whose root matches A. The notion of matching contexts to graphs was discussed in Section 4.73. Every pattern variable on the right-hand side of a pattern then refers to a subgraph specified by the right hand side of some graph rewriting rule.
- Any term constructed on the right side of a match is assumed to be constructed as if it were the contractum of a term graph rewriting rule (i.e., repeated instances of a variable result in sharing of terms referred to by variables; the node identifier of its root is set to be the same as the redex).
- ":=" causes a matched subgraph to be replaced by a graph constructed as above. No right hand sides of any :=s will be a simple variable, i.e., no projection rules will be applied; thus no redirection is required to perform the replacement operation.
- Statements inside "**seq**...**endseq**" are executed in strict sequence. "**skip**" causes no action to be performed when the corresponding pattern matches.

- *copyTop*(A) copies the root node of A, i.e., creates a node which is identical to the root of A, then sets the copy's node identifier to be the same as that of the redex being replaced.

The notation above is quite similar to the language CLEAN of Barendregt, et al. [BvEG+87b].

5.5.2 Normalization Algorithms

Figure 5.5 depicts the reduction algorithm *rwhnf*(). The algorithm simply applies graph rewriting rules derived from **ΛCCL** using a reduction strategy specified by recursion on the term structure. Figures 5.2, 5.3, and 5.4 consist of auxiliary algorithms for normalizing environments (to *PENF*).

Our goal is to use *rwhnf*() to effectively reduce a term $M \in \mathit{Ter}(\lambda^{\mathrm{DB}})$ to yield a term N such that

$$M \twoheadrightarrow_\beta N \in \mathit{WHNF}_\beta$$

if N exists.

rwhnf() is limited to reducing only *closure* terms, which has the effect of greatly simplifying the interpreter's structure. This design is accommodated by using the derived rule **(Beta′)**. We must also construct an initial term A having the form of a closure:

$$A \equiv [\,[\![M]\!]_{\Lambda\mathrm{CCL}},\, \emptyset] \in \mathit{Ter}(\Lambda\mathrm{CCL})$$

A is thus a closure whose term part is the **ΛCCL** equivalent of M, and whose environment is null. This construction has no effect on the outcome of the reduction, since by Lemma 5.7, we have

$$[A,\, \emptyset] =_{\Lambda\mathrm{CCL}} A$$

Applying *rwhnf*() to A then yields $B \in \mathit{WHNF}_{\Lambda\mathrm{CCL}}$, from which we can extract N:

$$N \equiv [\![\mathit{LNF}(B)]\!]_\lambda \in \mathit{WHNF}_\beta$$

Since *rwhnf*() simply applies **ΛCCL** rules to a term in a fixed order, Theorem 5.20 shows it to be correct, that is, given M, N, A, and B as above, we have

$$\mathit{rwhnf}(A) = B \in \mathit{WHNF}_{\Lambda\mathrm{CCL}}$$

implies

$$M \twoheadrightarrow_\beta N \in \mathit{WHNF}_\beta$$

The normalization properties of reduction schemes using **ΛCCL** depend on whether or not applications of the rule **(Beta′)** are *needed* (Definition 3.75). We therefore have the following theorem:

Theorem 5.24 *Let M be a term of $G\mathit{Ter}(\lambda^{\mathrm{DB}})$. Then $\mathit{rwhnf}([\![M]\!]_{\Lambda\mathrm{CCL}})$ terminates with a term $A \in G\mathit{Ter}(\Lambda\mathrm{CCL})$ such that $\mathit{LNF}(A) \in \mathit{WHNF}_{\Lambda\mathrm{CCL}}$ iff M has a weak head normal form.*

Proof We note from the structure of *rwhnf*() that all **(Beta′)** redexes contracted are weak head redexes, and that a **(Beta′)** redex must always be contracted after a finite number of **ECCL** contractions, unless no weak head redex exists. Thus by Lemma 5.23, we know that if *rwhnf*() does not terminate, there is a nonterminating weak head reduction of M, and thus M has no weak head normal form. Conversely, if *rwhnf*() terminates, $LNF(A)$ has no weak head redex and therefore is in weak head normal form. □

We then have the following corollary:

Corollary 5.25 *Let M be a term of $G\mathrm{Ter}(\lambda^{\mathrm{DB}})$. Then if $rwhnf([\![M]\!]_{\Lambda\mathrm{CCL}})$ terminates, there exists $N \in G\mathrm{Ter}(\lambda^{\mathrm{DB}})$ such that $N \in WHNF_\beta$ and*

$$M \twoheadrightarrow_\beta N$$

Proof Follows from Theorem 5.24 and Theorem 5.19. □

rwhnf(·) illustrates the simplicity with which interpreters can be specified using **ΛCCL**, and functions as a starting point for developing other interpreters (including ones that are "lazier" with respect to sharing of **(Beta)** or **(Beta′)** redexes). Another, and quite different, **ΛCCL**-based λ-interpreter is described in Chapter 8.

It should also be noted that the environment-normalizing algorithm *rpenf*(·) is unnecessarily general for use with *rwhnf*(·); a careful check of the environment terms actually produced by *rwhnf*(·) shows that only terms in *LEF* are actually generated by the reduction process. Therefore, a simpler normalization algorithm would suffice, though we include the more general algorithm here for completeness. In Chapter 8, we will be more fastidious about the efficiency of the reduction process and present a simpler environment normalization algorithm.

5.5.3 Related Work

A system almost identical to **ΛCCL** has been independently proposed by Abadi, et al. [ACCL90]. Its term structure is isomorphic to that of **ΛCCL**, and its axioms are the same with two minor exceptions. They propose to use their system to study properties of substitutions, to describe type-checking algorithms, and as the basis for machine-oriented implementations of reduction schemes.

5.6 Summary

In this chapter we have described a new formal system, **ΛCCL**, with which one can describe a wide variety of reduction methods for the lambda calculus using environments, closures, and shared terms. We have shown that terms and reductions in **ΛCCL** correspond in a natural way to their counterparts in the lambda calculus; λ-reduction schemes using **ΛCCL** may thus be proved correct trivially. However, by making the "micro-manipulations" required to implement substitution explicit, **ΛCCL** makes evident a broad range of options for implementation of efficient interpreters. Given the equivalence result of theorem 5.20, and the fact that **ΛCCL** removes difficulties of substitution and α-conversion present in the lambda calculus, we will henceforth concern ourselves almost exclusively with **ΛCCL** and its variants as a sound means for implementing the lambda calculus.

```
rpenf(E) ≡
    case E of
        ∅: skip;                                   {E ∈ PENF}
        □ⁿ: skip;                                  {E ∈ PENF}
        ⟨E₁, A⟩: rpenf(E₁);
        E₁ ∘ E₂: seq
            rpenf(E₁);
            rpenf(E₂);
            composeEnvs(E)
            endseq
```

Figure 5.2: Algorithm *rpenf*$(\cdot)$

```
composeEnvs((E₁ ∘ E₂): E) ≡                        {E₁, E₂ ∈ PENF}
    case E₁ of
        ∅: E := E₂;                                {rule (NullEL)}
        □: distribShiftL(E);
        ((□: E₃) ∘ E₄): seq
            E := E₃ ∘ ((E₄ ∘ E₂): E′);             {rule (AssE)}
            composeEnvs(E′);
            distribShiftL(E)
            endseq
        ⟨E₃, A⟩: seq
            E := ⟨(E₃ ∘ E₂): E′, [A, E₂]⟩;         {rule (DE)}
            composeEnvs(E′)
            endseq
```

Figure 5.3: Algorithm *composeEnvs*$(\cdot)$

$distribShiftL(((\Box\colon E_1) \circ E_2)\colon E) \equiv$ $\{E_2 \in PENF\}$
  case E_2 **of**
    $\emptyset$: $E := E_1$; $\{rule\ \textbf{(NullER)}\}$
    $\Box^n$: **skip**; $\{E = \Box^{n+1} \in PENF, n > 0\}$
    $\langle E_3,\ A\rangle$: $E := E_3$ $\{rule\ \textbf{(ShiftE)}\}$

Figure 5.4: Algorithm $distribShiftL(\cdot)$

$rwhnf([L,\ E]\colon C) \equiv$
  case L **of**
    Apply$(A,\ B)$: **seq**
      $C :=$ Apply$([A,\ E]\colon A',\ [B,\ E]\colon B')$; $\{rule\ \textbf{(DApply)}\}$
      $rwhnf(A')$;
      if $A' \equiv [\Lambda(A''),\ E']$
      then seq
        $C := [A'',\ \langle E',\ B'\rangle]$; $\{rule\ \textbf{(Beta')}\}$
        $rwhnf(C)$;
        endseq
      else skip;
    $\Lambda(A)$: **skip**; $\{C \in WHNF_{\Lambda\text{CCL}}\}$
    $[L_1,\ E_1]$: **seq**
      $C := [L_1,\ E_1 \circ E]$; $\{rule\ \textbf{(AssC)}\}$
      $rwhnf(C)$
      endseq;
    (Var: L_1): **seq** $\{C \equiv 0!\}$
      $rpenf(E)$; $\{transform\ E\ to\ PENF\}$
      case E **of**
        $\emptyset$: $C := L_1$; $\{rule\ \textbf{(NullC)}\}$
        $\Box^n$: **skip**; $\{E \equiv \Box^n,\ thus\ C \in WHNF_{\Lambda\text{CCL}}\}$
        $\langle E,\ A\rangle$: **seq**
          $rwhnf(A)$;
          $C := copyTop(A)$ $\{rule\ \textbf{(VarRef)},\ C \in WHNF_{\Lambda\text{CCL}}\}$
          endseq
      endseq

Figure 5.5: Algorithm $rwhnf(\cdot)$

Chapter 6

The Optimality Problem

"With delicious effrontery an entirely new melody enters."

"This Countersubject takes remarkable advantage of the fact that it is largely a decoration of the Subject."

—Tovey

6.1 Background

There has been much recent interest in efficient implementations of lazy functional programming languages whose semantics are based on normalizing reduction schemes for the lambda calculus [Pey87,FH88]. Most such implementations have made use of some combination of Wadsworth's graph reduction, e.g., [Aug84, Joh84], environments and closures [Lan64,HM76,AP81,FW87,CCM87] or *combinators* [Tur79,Hug84b,Joh85]. The first two are perhaps best seen as means to allow certain redexes to be effectively shared during reduction; the latter can be considered a restricted form of λ-expression for which certain implementation techniques are more efficient.

While all these techniques effectively yield weak head normal forms, all end up performing more β-contractions than are absolutely necessary by effectively *copying* redexes that could in principal be shared, as was illustrated in Example 2.4 of Section 2.3.1. In some cases, this lack of sufficient laziness can result in considerable unnecessary additional computation. Concern for this phenomenon led to the introduction of methods allowing "fully-lazy" reduction [Hug84b]. Unfortunately, the exact nature of laziness has apparently been heretofore something of a mystery. For example, in [Pey87, p. 400], Peyton Jones states that

> ...it is by no means obvious how lazy a function is, and...we do not at present have any tools for reasoning about this. Laziness is a delicate property of a function, and seemingly innocuous program transformations may lose laziness.

However, Lévy's analysis in [Lév78,Lév80] made clear that there was a wide range of laziness possible for reduction in the lambda calculus—ranging from profligately wasteful to optimal—with full-laziness actually lying somewhere in between. The mystery, such as it is, lies not so much in *reasoning* about laziness as finding ways to *implement* reduction schemes that are able to exploit it fully.

We thus begin this chapter by reviewing the optimality results of Lévy (also pursued jointly by Berry and Lévy [BL79] for recursive program schemes), where we regard optimality as the goal that any lazy implementation should strive to achieve. Lévy's results are presented in the setting of regular replacement systems. We then address the issue of implementation. We view ΛCCL as a formal system that is able to represent most of the techniques exploited by graph, closure, and combinator-based reduction techniques for lazy languages. Unfortunately, we show that ΛCCL is nonetheless insufficient to implement optimal λ-reduction, in a sense we will make precise in the sequel.

6.2 Redex Sharing and Environments

Let us begin by reviewing the Example 2.4, which shows that Wadsworth's graph reduction is not optimal:

Example 2.4

$$\begin{array}{l} \rho_1: N \longrightarrow (\bullet w)(\bullet z) \longrightarrow (Iw)(\bullet z) \longrightarrow w(\bullet z) \longrightarrow w(Iz) \longrightarrow wz \\ \qquad\qquad\qquad\quad \lambda y.(Iy) \quad\ \lambda y.(Iy) \qquad \lambda y.(Iy) \end{array}$$

$$\begin{array}{l} \rho_2: N \longrightarrow (\bullet w)(\bullet z) \longrightarrow (\bullet w)(\bullet z) \longrightarrow w(\bullet z) \longrightarrow wz \\ \qquad\qquad\qquad\quad \lambda y.(Iy) \qquad\qquad \lambda y.y \qquad \lambda y.y \end{array}$$

By examining the reductions above we can see that Wadsworth left the door open to further improvements by not taking advantage of all conceivable opportunities for redex sharing. Note in ρ_1 that as $\lambda y.(Iy)$ is applied in sequence to w and to z, the inner redex (Iy) is effectively copied (after each substitution for y). If there were some means to *parametrically* share the (Iy) redex while still substituting w and z separately for y, more efficient, and perhaps optimal, reductions might still be achievable. This suggests the use of the notions of environment and closure embodied in ΛCCL.

6.3 Optimality Criteria

In examining the question of optimal reduction in the lambda calculus, Lévy noted that by sharing redexes through graph structures, Wadsworth was essentially contracting multiple β-redexes in parallel. Lévy was able to define a natural class of parallel reductions—the *family-parallel* reductions—for which relatively simple criteria could be stated that guarantee a reduction's optimality. This subclass of optimal family-parallel reductions consists of those that are *complete* and *call-by-need*. Since a serial (non-parallel) reduction is a degenerate family-parallel reduction, an optimal complete reduction is also optimal for the class of serial reductions.

Lévy's critical observation was that the notion of family defines not only which contractions are residuals of some common contraction, but also which contractions *could* be residuals of a common contraction in some alternative reduction. He also noted that since family-completeness and residual-completeness correspond, the set of contractions that comprises a family is relatively easy to ascertain at each step of the reduction.

The question was then whether any *practical* reduction scheme could be implemented that would ensure that all copies in a family class were represented by a single structure. Contraction of such a structure would then effectively contract all copies. Lévy speculated that some scheme using shared closures, which permit contractions independent of substitutions for free variables (i.e., environments) might be a means for performing optimal λ-reduction. We will argue here, however, that such devices alone, even in conjunction with graph replacement systems, are insufficient to allow optimal λ-reduction.

Staples [Sta80a,Sta80b,Sta80c][1] has also investigated optimal reduction strategies in a number of different settings. His work, however, treats graph reduction as an end in itself, rather than as a means for implementing parallel reduction. It thus does not allow a precise comparison to be made between parallel and serial reduction strategies in the same system. For that reason, we regard his results as less general than those of Lévy. In addition, the much more algebraic character of Lévy's results also renders them more amenable to generalization and further exploitation.

6.4 Optimality Theorem

We begin with a formal definition of optimality:

Definition 6.1 (Optimal Reduction Strategy)
*Let $\langle \mathcal{F}, \longrightarrow_R, / \rangle$ be an RRS. Let $\mathcal{N} \subseteq \mathcal{F}$ be some set of normal forms of interest, and $\mathcal{C} \subseteq \|(R)^*$ be a designated class of (parallel) reductions. Then a reduction strategy s is $\mathcal{C},\mathcal{N}$-*optimal *iff the following hold:*

(1) *s is an $\mathcal{N}$-normalizing strategy*

(2) *For all $\mathcal{N}$-normalizable elements $X \in \mathcal{F}$, let i be minimal such that*

$$\rho_f^i(X){:}X \;\in\; \mathcal{N}$$

[1] [Ken84] contains as a very readable summary of Staples' results.

Then there exists no *reduction sequence* $\sigma \in \mathcal{C}$ *such that*

$$\sigma : X \ \in \ \mathcal{N}$$

and

$$|\sigma| < |\rho^i_f(X)|$$

By combining a reduction strategy with an implementation, we can define a reasonable notion of *optimal* implementation:

Definition 6.2 (Optimal Implementation)
Let

$$\begin{array}{rcl} \Phi & = & \langle \mathcal{E}, \ \longrightarrow_R, \ /_{\Phi} \rangle \\ \Psi & = & \langle \mathcal{F}, \ \longrightarrow_S, \ /_{\Psi} \rangle \end{array}$$

be RRS*'s and* $\mathcal{N} \subseteq \mathcal{E}$ *be a set of normal forms such that*

$$\langle \Psi, \ g, \ h \rangle$$

is a complete implementation of Φ *with respect to* $\mathcal{N}$. *Let* s *be an* S*-reduction strategy. Let* $\mathcal{N}_{\mathcal{F}}$ *be the set defined by*

$$\mathcal{N}_{\mathcal{F}} \ \triangleq \ \{X \mid g(X) \in \mathcal{N}\}$$

(Given the definition of implementation, $\mathcal{N}_{\mathcal{F}}$ *is clearly also a set of normal forms). Finally, let* C *be a class of reductions in* $\|(R)^*$.

Then $\langle \Psi, g, h, s \rangle$ *is a* $\mathcal{C}$,$\mathcal{N}$-optimal implementation *of* Φ *iff* s *is* $\mathcal{N}_{\mathcal{F}}$*-normalizing, and for every* $E \in \mathcal{E}$, $\rho \in \mathcal{C}$, *and minimal* i *such that*

$$s^i(h(E)) \in \mathcal{N}_{\mathcal{F}}$$

and

$$\rho : E \in \mathcal{N}$$

then the following holds:

$$|s^i(h(E))| \ \leq \ |\rho|$$

An optimal implementation of RRS Φ is one whose reductions are always shorter than any of those of Φ that they simulate.

We now state Lévy's "fundamental theorem of optimality," which is given in various forms in each of [Lév78,BL79,Lév80]:

Theorem 6.1 (Fundamental Theorem of Optimality [Lév78,BL79,Lév80])
Let $\langle \mathcal{F}, \ \longrightarrow_R, \ / \rangle$ *be an an* RRS. *Let* $\mathcal{N} \subseteq \mathcal{F}$ *be a set of normal forms of* $\mathcal{F}$. *Let* $F(R)^*$ *be the set of family-parallel R-reductions. Then any parallel reduction strategy* s *which is*

(1) *Complete*

(2) $\mathcal{N}$*-call-by-need*

is $F(R)^*, \mathcal{N}$*-optimal.*

The proof of Theorem 6.1 relies on the fact that if ρ is a reduction that yields an $\mathcal{N}$-normal form and σ is complete and call-by-need, then

$$\mathrm{FAM}(\sigma) \;\subseteq\; \mathrm{FAM}(\rho)$$

But then by clause (2) of Corollary 3.47, we know that $|\sigma| \leq |\rho|$, and thus we can conclude that any strategy that yields complete, call-by-need reductions is optimal. Since serial reductions are degenerate family-parallel reductions, a complete, call-by-need reduction is also optimal for the class of all serial reductions.

We then get the following corollaries to Theorem 6.1:

Corollary 6.2 *Let*

$$\begin{aligned} \Phi &= \langle \mathcal{E},\ \longrightarrow_R,\ /_{\!\Phi} \rangle \\ \Psi &= \langle \mathcal{F},\ \longrightarrow_S,\ /_{\!\Psi} \rangle \end{aligned}$$

be RRS*'s and* $\mathcal{N} \subseteq \mathcal{E}$ *be a set of normal forms such that*

$$\langle \Psi,\ g,\ h \rangle$$

is a strong family-preserving implementation of Φ *that is complete with respect to* $\mathcal{N}$*. Let* $\mathcal{N}_{\mathcal{F}}$ *be the set defined by*

$$\mathcal{N}_{\mathcal{F}} \;\triangleq\; \{X \mid g(X) \in \mathcal{N}\}$$

Finally, let s *be a complete,* $\mathcal{N}_{\mathcal{F}}$*-call-by-need strategy for* Ψ *such that for all* $E \in \mathcal{E}$*,* $F', F'' \in \mathcal{F}$*, and complete reductions* $\rho\mathcal{P} \in C(S)^*$*,*

$$s^i(\,h(E)\,)\!:\!h(E) \;=\; F'$$

and

$$s(F')\!:\!F' \;=\; F''$$

together imply that there exists a complete reduction $\sigma\mathcal{Q} \in C(R)^*$ *such that*

$$\sigma\!:\!E \;=\; g(F')$$

and

$$\mathcal{Q}\!:\!g(F') \;=\; g(F'')$$

and $\mathcal{Q}$ *contains a* $\mathcal{N}$*-needed contraction. Then* $\langle \Psi, g, h, s\rangle$ *is a* $F(R)^*, \mathcal{N}$*-optimal implementation of* Φ*.*

Proof Since Ψ is a strong family-preserving implementation, and since s effectively simulates $\mathcal{N}$-call-by-need reductions we know that for every element $E \in \mathcal{E}$ and minimal i such that

$$s^i(h(E)) : h(E) \in \mathcal{N}_{\mathcal{F}}$$

there corresponds a reduction $\rho \in F(R)^*$ such that ρ is complete, $\mathcal{N}$-call-by-need

$$\rho : E \in \mathcal{N}$$

and

$$|s^i(h(E))| = |\rho|$$

Thus $\langle \Psi, g, h, s\rangle$ is an optimal implementation. □

From Theorem 3.49, the following is immediate:

Corollary 6.3 *Let $\langle \mathcal{F}, \longrightarrow_S, /\rangle$ be a* disjoint RRS. *Let $\mathcal{N} \subseteq \mathcal{F}$ be a set of normal forms. Then any reduction strategy s which is $\mathcal{N}$-call-by-need is $F(R)^*, \mathcal{N}$-optimal.*

(This is essentially the result proved by Staples in [Sta80b]).

From Corollary 6.2, we get the following:

Theorem 6.4 *Let R be a (conditional or unconditional) strongly regular Σ-term rewriting system, and let the corresponding graph replacement system (given by Proposition 4.22) be R^G. Let $\mathcal{N}$ be a set of normal forms of R, and $\mathcal{N}_G$ be the set defined by*

$$\mathcal{N}_G \triangleq \{X \mid U(X) \in \mathcal{N}\}$$

Let s be an $\mathcal{N}_G$-call-by-need reduction strategy for R^G. Then $\langle R^G, U, \mathbf{id}, s\rangle$ is an $\mathcal{N}$-optimal implementation of R, where U is the unraveling mapping, and **id** *is the Σ-algebra isomorphism from $\mathcal{T}(\Sigma)$ to $Graph1(\Sigma)$.*

Proof Since by definition, R^G is disjoint, by Corollary 3.50, it is complete. By Theorem 4.22, we have that $\langle R^G, U, \mathbf{id}\rangle$ is a strong family-preserving complete implementation of R. By definition, s preserves $\mathcal{N}_G$-neededness. Therefore, by Corollary 6.2, s is relatively optimal. □

Note that Theorem 6.1 does not specify means of performing family complete reductions. With strongly regular term rewriting systems, we can use family-preserving graph replacement implementations to yield optimal reductions. However, the families of the lambda calculus are not as readily implemented as those of term rewriting systems. We will thus develop in the next section a *labeled* variant of ΛCCL to make the connection between λ-families and parallel ΛCCL redexes.

$$
\begin{array}{ll}
\textbf{(Beta)} & \mathrm{Apply}(\Lambda(A)^{w},\ B)^{u} = [A^{u\underline{w}},\ \langle \emptyset,\ B^{\underline{w}}\rangle] \\
\textbf{(AssC)} & [[A,\ E_1]^{u},\ E_2]^{v} = [A^{uv},\ E_1 \circ E_2] \\
\textbf{(NullEL)} & \emptyset \circ E = E \\
\textbf{(NullER)} & E \circ \emptyset = E \\
\textbf{(ShiftE)} & \Box \circ \langle E,\ A\rangle = E \\
\textbf{(VarRef)} & [\mathrm{Var}^{u},\ \langle E,\ A\rangle]^{v} = A^{uv} \\
\textbf{(D}\Lambda\textbf{)} & [\Lambda(A)^{u},\ E]^{v} = \Lambda([A,\ \langle E \circ \Box,\ \mathrm{Var}\rangle])^{uv} \\
\textbf{(DE)} & \langle E_1,\ A\rangle \circ E_2 = \langle E_1 \circ E_2,\ [A,\ E_2]\rangle \\
\textbf{(DApply)} & [\mathrm{Apply}(A,\ B)^{u},\ E]^{v} = \mathrm{Apply}([A,\ E],\ [B,\ E])^{uv} \\
\textbf{(AssE)} & (E_1 \circ E_2) \circ E_3 = E_1 \circ (E_2 \circ E_3) \\
\textbf{(NullC)} & [A^{u},\ \emptyset]^{v} = A^{uv} \\
\textbf{(DLabel)} & [A,\ E]^{u} = [A^{u},\ E]
\end{array}
$$

Figure 6.1: $\Lambda\mathrm{CCL}^{\mathrm{L}}$ Axioms

6.5 Labeled ΛCCL

Since the members of a λ-family class may consist of different substitution instances of the same term, it is essential that a practical reduction scheme allow such sets of differing terms to be effectively shared. ΛCCL provides a means to do this: the closure. Two closures with different environment subterms (representing different free variable substitutions) may share a common subterm: e.g., $[A, E_1]$ and $[A, E_2]$, where the two instances of A are shared. This idea was exploited in Example 5.2.

To make a precise connection between redex families in the lambda calculus and shared terms with closures and environments in ΛCCL, we will define a *labeled* variant of ΛCCL, $\Lambda\mathrm{CCL}^{\mathrm{L}}$. $\Lambda\mathrm{CCL}^{\mathrm{L}}$ is intended to be analogous to Lévy's labeled lambda calculus.

Definition 6.3 $\Lambda\mathrm{CCL}^{\mathrm{L}}$ *is the equational theory of terms of* $\mathrm{Ter}(\Lambda\mathrm{CCL}^{\mathrm{L}})$ *whose axioms are given in Figure 6.1.*

Note that the **(DLabel)** has no analogue in unlabeled ΛCCL. It is the $\Lambda\mathrm{CCL}^{\mathrm{L}}$ equivalent of the the convention allowing the removal of parentheses in multiply-labeled parenthesized terms of λ^{L}. By analogy with the labeled lambda calculus, the *degree* of a labeled **(Beta)** redex is the label of its abstraction term; e.g.,

$$\mathrm{Apply}(\Lambda(A^{u})^{w},\ B^{v})^{z}$$

has degree w. As with the labeled-lambda calculus, we can define the *erasure* of labeled $\Lambda\mathrm{CCL}^{\mathrm{L}}$ terms and reductions, yielding their unlabeled counterparts. Likewise, we can define the *lifting* of unlabeled terms and reductions to labeled versions.

The translations $[\![\cdot]\!]_{\lambda^{\mathbf{L}}}$ and $[\![\cdot]\!]_{\Lambda\mathrm{CCL}^{\mathbf{L}}}$ are defined in the obvious way analogous to their unlabeled counterparts. Theorems 5.15, 5.4, 5.19, and 5.20 all apply to $\Lambda\mathbf{CCL}^{\mathbf{L}}$ and $\lambda^{\mathbf{L}}$; the proofs are analogous and are omitted.

We also wish to consider *parallel* labeled reductions, in which sets of identical (including labeling) terms are contracted at each step. The parallel analogues of the aforementioned theorems are straightforward to define. We will make use in particular of the following refinement of Lemma 5.19:

Lemma 5.19′ *Let $A{:}\mathcal{L}$ be a term of* $\Lambda\mathbf{CCL}^{\mathbf{L}}$ *such that*

$$A \xrightarrow{\rho}\!\!\!\!\twoheadrightarrow^{\shortparallel}_{\Lambda\mathrm{CCL}^{\mathrm{L}}} B$$

Let R_i be the set of redexes contracted in the ith parallel (**Beta**) *contraction of ρ, w_i be the degree of the redexes contracted in R_i, and L_ρ be the multiset consisting of the labels w_i.*

Then there exists reduction σ such that

$$[\![LNF(A)]\!]_{\lambda^{\mathrm{L}}} \xrightarrow{\sigma}\!\!\!\!\twoheadrightarrow^{\shortparallel}_{\beta} [\![LNF(B)]\!]_{\lambda^{\mathrm{L}}}$$

where all the members of the set S_j of redexes contracted in the jth parallel β-contraction of σ have the same degree, v_j.

Furthermore, if we let L_σ be the multiset consisting of the labels v_j, then we have

$$L_\rho = L_\sigma$$

Proof Since the redexes in each parallel (**Beta**) contraction of ρ are by definition disjoint, we may use a construction analogous to that used in the proof of Lemma 5.19 to yield σ, where the reduction of disjoint β-redexes at each stage of the construction is replaced by a single parallel β-contraction.

We note that the degrees of the (**Beta**) redexes in the premise of the labeled analogue of Lemma 5.11 are preserved in the degrees of the (**Beta**) redexes in its conclusion. From this we can conclude that $L_\rho = L_\sigma$. □

We can then make the following definitions:

Definition 6.4 *Let A contain* (**Beta**) *redex R and let ρ be the reduction*

$$A \xrightarrow{\rho}\!\!\!\!\twoheadrightarrow_{\Lambda\mathrm{CCL}} B$$

Let A^l be a labeling of A such that R has degree w and all other subterms of A^l have the null label. Finally, let ρ^l be the labeled counterpart of ρ such that

$$A^l \xrightarrow{\rho^l}\!\!\!\!\twoheadrightarrow_{\Lambda\mathrm{CCL}^{\mathrm{L}}} B^l$$

Then (**Beta**) *redex S in B is a* λ-residual *of R (relative to reduction ρ) if its labeled counterpart S^l in B^l has degree w.*

If R is a **(Beta)** redex in $A\colon\mathcal{L}$, we will refer without ambiguity to the residuals of R in $LNF(A)$, since $\mathbf{\Lambda CCL^L}$ is confluent and any **ECCL** reduction of A to $LNF(A)$ must yield the same set of residuals.

Definition 6.5 *Let ρ be the following parallel reduction of $A\colon\mathcal{L}$:*

$$A \xrightarrow{\rho}\!\!\twoheadrightarrow_{\overset{\shortparallel}{\Lambda \mathrm{CCL}}} B$$

Then ρ is λ-optimal *if the number of (parallel)* **(Beta)** *contractions in ρ is less than or equal the number of parallel β-contractions in an optimal (in the sense of Lévy) λ-reduction from $[\![LNF(A)]\!]_\lambda$ to $[\![LNF(B)]\!]_\lambda$.*

Definition 6.6 *A* **(Beta)** *redex R in a term $A\colon\mathcal{L}$ is* λ-needed *if and only if some λ-residual of R in $LNF(A)$ is needed.*

We can now apply Lévy's optimality criteria directly to reductions in $\mathbf{\Lambda CCL}$, using $\mathbf{\Lambda CCL^L}$. The idea is to consider each **(Beta)** contraction in a term A as representing a parallel β-contraction on the corresponding λ-term $[\![LNF(A)]\!]_\lambda$.

Theorem 6.5 *Let ρ be the following parallel reduction of $A\colon\mathcal{L}$:*

$$A \xrightarrow{\rho}\!\!\twoheadrightarrow_{\overset{\shortparallel}{\Lambda \mathrm{CCL}}} B$$

Let $A^l \in \mathbf{\Lambda CCL^L}$ be a labeling of A such that all of A's subterms have unique labels, and let ρ^l be the labeled counterpart of ρ such that

$$A^l \xrightarrow{\rho^l}\!\!\twoheadrightarrow_{\overset{\shortparallel}{\Lambda \mathrm{CCL^L}}} B^l$$

Let S_i be the set of (identical) **(Beta)** *redexes contracted in the ith parallel* **(Beta)** *contraction in ρ and w_i be the degree of the redexes contracted in by the labeled counterpart of S_i in ρ^l, S_i^l. Then the reduction ρ is λ-optimal only if*

- *For all i, j, $w_i \neq w_j$.*
- *For all i, some redex in R_i is λ-needed.*

Proof Consider the reduction $\hat{\rho}^l$ constructed in Lemma 5.19$'$ such that

$$\hat{\rho}^l\colon [\![LNF(A^l)]\!]_{\lambda^{\mathrm{L}}} \twoheadrightarrow_\beta [\![LNF(B^l)]\!]_{\lambda^{\mathrm{L}}}$$

If a redex with degree w is contracted twice in ρ, a β-redex with degree w must be contracted at least twice in $\hat{\rho}^l$ (since the multisets of redex degrees in the two reductions are equal). But then there must be some *shorter* family-complete parallel β-reduction σ such that

$$\sigma\colon [\![LNF(A^l)]\!]_{\lambda^{\mathrm{L}}} \twoheadrightarrow_{\shortparallel\beta} [\![LNF(B^l)]\!]_{\lambda^{\mathrm{L}}}$$

in which the redex labeled w is reduced only once (since σ is family-complete), thus ρ is not λ-optimal. Similarly, if ρ contracts an unneeded β-redex, then an unneeded β-redex is contracted in $\hat{\rho}^l$, and there is a shorter reduction in which no unneeded β-redex is contracted. $\square$

If we construct a λ-interpreter whose action can be expressed in terms of some application of the rules of **ΛCCL**, we can determine how close to optimality any such interpreter can come by showing how many **(Beta)** redexes in the corresponding labeled reductions have the same label.

6.6 Non-Optimality of Reduction with Shared Closures

We can now show that there is *no* λ-optimal reduction possible in the term graph rewriting system corresponding to **ΛCCL** in the sense discussed in Section 4.9, and formalized in [BvEG+87a]. We do so by exhibiting a λ-term Q for which *every* **ΛCCL** term graph reduction causes more than one **(Beta)** redex to be contracted in the corresponding labeled form. The term Q is as follows:

$$(\lambda x.((xA^d)(xB^e)))(\lambda y.((\lambda z.(z^a t)(z^b u))(\lambda w.y^c v)))$$

where A and B are arbitrary λ-abstractions. Not all subterms are given labels for the sake of clarity.

We will not enumerate all possible reductions of Q's corresponding $\mathbf{\Lambda CCL^L}$ translation. However, the crux of the matter is embodied in the following term, which must be produced in any reduction of the **ΛCCL** equivalent of Q if no prior **(Beta)** redexes with the same label are to be reduced twice:

$$(([\bullet,\ \langle\!\langle y := A^d\rangle\!\rangle]\colon C_1)([\bullet,\ \langle\!\langle y := B^e\rangle\!\rangle]\colon C_2))$$
$$[((z^a t)(z^b u)),\ \langle\!\langle z := \lambda w.(y^c v)\rangle\!\rangle]\colon C_3$$

As before, $\langle\!\langle\cdot\rangle\!\rangle$ represents a **ΛCCL** environment with bound variables indicated explicitly. The notation $T\colon N$ is used to give names to subterms. One is forced here to choose between reducing closure C_3 or one of closures C_1 or C_2. Choosing C_3 yields:

$$(([\bullet,\ \langle\!\langle y := A^d\rangle\!\rangle]\colon C_1)([\bullet,\ \langle\!\langle y := B^e\rangle\!\rangle]\colon C_2))$$
$$((y^c v)(y^c v))$$

which reduces to

$$(((A^{dc}v)(A^{dc}v))([\bullet,\ \langle\!\langle y := B^e\rangle\!\rangle]\colon C_2))$$
$$((y^c v)(y^c v))$$

in which two redexes of the form $(A^{dc}v)$ are created, thus yielding a non-optimal reduction (since they have the same degree and are no longer shared).

To avoid the copying that occurs above, one could alternately first reduce closure C_1 (or C_2, for which the argument to follow is symmetric), which would eventually yield a term of the following following form:

$$(([\bullet,\ \langle\!\langle z := \lambda w.(A^{dc}v)\rangle\!\rangle]\colon C_1{}')([\bullet,\ \langle\!\langle z := \lambda w.(B^{ec}v)\rangle\!\rangle]\colon C_2{}'))$$
$$((z^a t)(z^b u))$$

which reduces to

$$\underbrace{((\lambda w.\bullet)^a t)((\lambda w.\bullet)^b t)}_{(A^{dc}v)}\underbrace{((\lambda w.\bullet)^a t)((\lambda w.\bullet)^b t)}_{(B^{ec}v)}$$

The term above has two (actually, two sets) of unshared redexes with the same degree, e.g., $((\lambda w.(A^{dc}v))^a t)$ and $((\lambda w.(B^{ec}v))^a t)$. If both are needed (which depends on the particular abstractions chosen for A and B), a non-optimal reduction will once again result. In the end, no matter what choice is made, a non-optimal reduction occurs.

The informal observation that term graph rewriting implementations of closures and environments are inadequate for implementing optimal reduction schemes was also made independently by Curien in [Cur86b]. He did not, however, provide a formal connection (such as that made above using labels) between redex families in the lambda calculus and their equivalents in a formal system using environments, nor was the system he was using as general as the one proposed here.

6.7 Related Work

[AKP84] provides an analysis of the differences between various lazy and fully-lazy λ-interpreters without examining the issue of optimality.

There have been several proposed implementations of optimal λ-reduction: by Staples [Sta82] and more recently by Lamping [Lam90] and Kathail [Kat90]. These schemes seem to allow more terms to be shared than are possible using traditional environment or substitution mechanisms. However, they are notable for their extreme complexity, and it is not clear that the overhead incurred by these schemes in order to ensure that family classes are always shared is not prohibitive. Such complexity may be inherent, however, in the peculiar nature of redex families in the lambda calculus.

6.8 Comments

We note from the counterexample of Section 6.6 that sharing of redex families requires that existing sharing in a graph be *respected* during the process of substitution, in addition to being *created* as a consequence of the application of rewrite rules with multiple instances of meta-variables. In order to implement optimal reduction, it thus seems that more powerful rewriting systems (i.e., non-left-linear systems that can "test" for equality of subterms) or graph rewriting systems with capabilities beyond those of term graph rewriting (such as those used by Lamping and Kathail) are required.

In this chapter we have described a labeled variant of Λ**CCL**, $\Lambda\mathbf{CCL}^{\mathbf{L}}$, which can be used as a precise tool to analyze environment and closure-based implementations of the lambda calculus to determine the extent to which the implementation is lazy. We have shown, however, that the standard term graph rewriting implementation of Λ**CCL** is insufficient for implementing *optimal* reduction schemes.

Chapter 7

A Theory of Incremental Reduction

"...the effect of a correct analysis can only be to inculcate a broader view."

"[It] is short, and the stress on its formal aspects is a positive æsthetic gain."

—Tovey

7.1 Preliminaries

Our aim in this chapter is to provide a rigorous treatment of incremental reduction that is applicable to *any* regular replacement system. Many of the ideas informally presented in Chapter 2 with regard to the λ-calculus will be generalized here. We will thus make extensive use of the reduction theory developed in Chapter 3.

Consider an element X_1 of an RRS that is the join of two disjoint elements A and B. We want to be able to determine the effect of a reduction ρ of X_1 on a *sub-element*, e.g., A, of X_1. We will therefore define the notion of a *projection* of ρ on A, ρ^A, as a maximal reduction of A that is consistent with ρ.

Let A' be the result of applying the projection of ρ on A to A, i.e., such that

$$\rho^A : A \quad = \quad A'$$

We can now replace B in X_1 with $C \parallel A$, to yield

$$X_2 = A \sqcup C$$

Since A and C are disjoint, we know that ρ^A is a reduction of X_2 such that

$$\rho^A : X_2 \quad = \quad A' \sqcup C$$

Thus a reduction of $A' \sqcup C$ can also be treated as a reduction of X_2.

How does all of this apply to incremental computation? The idea is to think of $\rho^A : A \equiv A'$ as a repository for "storing" all the computations of ρ that apply to A. Thus when we replace B by C, the subsequent reduction of $A' \sqcup C$ will avoid repeating contractions of A that had already been performed by ρ^A, instead performing only those contractions required by the presence of C. The trick, however, is to be able to reduce $A \sqcup B$ while preserving the result of the projection of ρ on A. We must also define a notion of projection for an appropriate class of reductions that ensures we can legitimately equate "computation of the projection" with "avoidance of subsequent redundant computations."

7.2 Incremental Optimality

To formalize the notion of *optimal* incremental reduction we wish to capture, we make the following definition:

Definition 7.1 (Optimal Incremental Reduction)
Let $\langle \mathcal{B}, \longrightarrow_R, / \rangle$ be an RRS*, $\mathcal{C}$ be a class of reductions of $\|(R)^*$, and $\mathcal{N}$ be a set of normal forms of $\mathcal{B}$.*

Let X_1, X_2, and A be elements of $\mathcal{B}$ such that $X_1 \overset{A}{\sim} X_2$. Let $\rho \in \mathcal{C}$ be a reduction of X_1 such that $\rho : X_1 \in \mathcal{N}$, $\sigma \in \mathcal{C}$ be a reduction of A such that $\sigma \lesssim_R \rho$, and $\tau \in \mathcal{C}$ be such that $\sigma\tau \in \mathcal{C}$ is a reduction of X_2 and $\sigma\tau : X_2 \in \mathcal{N}$.

Then $\sigma\tau$ is incrementally $\mathcal{C},\mathcal{N}$-optimal *with respect to ρ iff there exists no reductions $\sigma', \tau' \in \mathcal{C}$ such that σ' is a reduction of A, $\sigma' \lesssim_R \rho$, $\sigma'\tau'$ is a reduction of X_2 where $\sigma'\tau' : X_2 \in \mathcal{N}$, and*

$$|\tau'| \;<\; |\tau|$$

Definition 7.1 has two premises: First, that for every reduction ρ of X_1 to a normal form, we compute some sub-reduction σ of ρ on common element A. Second, we assume that we also perform the reduction of σ on A.

Given a subsequent reduction τ such that $\sigma\tau$ reduces X_2 to a normal form, we then say that $\sigma\tau$ is optimal with respect to ρ iff there is no other pair of reductions $\sigma'\tau'$ which satisfies the same requirements such that τ' yields a normal form in fewer steps.

In the sequel, we will show that the conditions above are met if ρ and τ are optimal reductions, and σ is a projection of ρ on A.

7.3 Relative Equivalence

We begin our formal exposition of the ideas discussed above by defining the following strengthened version of Lévy's notion of reduction prefix:

Definition 7.2 (Relative Prefix)
Let $\langle \mathcal{B}, \longrightarrow_R, / \rangle$ be an RRS*. Let ρ and σ be coinitial elements of $\|(R)^*$. Then*

ρ is a relative prefix *of σ, notation*

$$\rho \ \lesssim_R \ \sigma$$

iff

(1) $\rho \leq \sigma$

(2) *ρ is relative to σ.*

Extending the notion of relative prefix to an equivalence yields:

Definition 7.3 (Relative Equivalence)
Let $\langle \mathcal{B}, \longrightarrow_R, / \rangle$ be an RRS. *Let ρ and σ be coinitial elements of $\|(R)^*$. Then ρ is* relatively equivalent *to σ, notation*

$$\rho \ \simeq_R \ \sigma$$

iff $\rho \lesssim_R \sigma$ and $\sigma \lesssim_R \rho$.

$\simeq_R$ is clearly an equivalence relation, since $\simeq$ is an equivalence relation and relatively equivalent reductions reduce exactly the same set of families.

In the sequel, we will use the relative prefix relation ('$\lesssim_R$') rather than the Lévy's prefix relation ('$\lesssim$') when comparing reductions. We prefer the former to the latter since '$\lesssim_R$' better captures the idea of comparing the amount of "work" done by two reductions. Here, we regard the number of families contracted in a reduction as an intuitively appealing measure of the amount of work performed by a reduction.

If ρ and σ are coinitial reductions such that

$$\rho \ \lesssim \sigma$$

it may be the case that ρ performs many contractions (perhaps an infinite number) that are subsequently erased, while σ is relatively short (see [Lév78,] for an example of such a pair of reductions in the lambda calculus). By contrast, if the contractions in ρ that were erased were not in members of any families contracted by σ, then it would *not* be the case that

$$\rho \ \lesssim_R \ \sigma$$

7.4 The Lattice of Complete Relative Reductions

By restricting our attention to *complete* reductions that are related by the relative prefix relation, we will be able to define a sensible notion of projection. We begin by showing that the set of coinitial complete relative reductions forms a lattice:

Theorem 7.1 *Let* $\langle \mathcal{B}, \longrightarrow_R, / \rangle$ *be an* RRS. *Let* $CR(X)$ *be the set of finite complete reductions of some element* $X \in \mathcal{B}$. *Let* $R(X)/\simeq_R$ *be the set of equivalence classes of* $CR(X)$ *defined by the relation* $\simeq_R$, *whose members are ordered by:*

$$\rho \lesssim_R \sigma \quad \Longrightarrow \quad [\rho] \lesssim_R [\sigma]$$

(where we once again overload meaning of $\lesssim_R$*). Then for all complete reductions* $\rho, \sigma \in C(R)^*$ *of* X, $[\rho]$ *and* $[\sigma]$ *have a greatest lower bound*

$$[\rho] \sqcap_R [\sigma] \ \in \ R(X)/\simeq_R .$$

and the set $R(X)/\simeq_R$ *ordered by* $\lesssim_R$ *forms a lattice.*

Proof Let FAM(ρ) and FAM(σ) be the (necessarily finite) sets of families reduced in ρ and σ, respectively. By Corollary 3.47, clause (2), we know that the set of all complete reductions relative to ρ and σ is finite. Therefore, let $\mathcal{T} \subseteq CR(X)$ be the set of complete reductions of X relative to both ρ and σ, i.e.,

$$\mathcal{T} \equiv \{\tau \mid \tau \lesssim_R \sigma, \ \tau \lesssim_R \rho\}$$

Since $\mathcal{T}$ is finite, it has a least upper bound $\bigsqcup_R(\mathcal{T})$, which by Proposition 3.48 is an element of $CR(X)$. Thus

$$[\bigsqcup_R(\mathcal{T})]$$

is a greatest lower bound for $[\sigma]$ and $[\rho]$. □

Thus we see that for any pair of complete reductions, there exists a "maximal overlap" (modulo $\simeq_R$) given by $[\rho] \sqcap_R [\sigma]$. Lévy [Lév78] has shown that such a greatest lower bound *does not* in general exist for reductions ordered by '$\lesssim$' (rather than '$\lesssim_R$'). The set of equivalence classes of coinitial reductions with respect to '$\simeq$' thus forms only a semilattice (by Theorem 3.26), rather than a lattice.

We can now define what it means for two reductions to *overlap*:

Definition 7.4 (Overlapping Reductions)
Let $\langle \mathcal{B}, \longrightarrow_R, / \rangle$ *be an* RRS. *Let* ρ *and* σ *be coinitial elements of* $\|(R)^*$. *Then* ρ *and* σ overlap, *notation* $\rho \asymp \sigma$, *iff*

$$[\rho] \sqcap_R [\sigma] \ \neq \ [\epsilon]$$

By Definition 7.4, two reductions overlap iff they have a non-null common prefix.

7.5 Projections

Recall that our goal is to determine the effect of a reduction on some sub-element of the element being reduced. By refining the means by which we compare two reductions, we were able to extract a greatest common prefix from two coinitial reductions. With these preliminaries behind us, we can now give a formal definition of projection:

Definition 7.5 (Projection)
Let $\langle \mathcal{B}, \longrightarrow_R, / \rangle$ be an RRS. *Let $X \equiv A \sqcup B \in \mathcal{B}$ be such that $A \parallel B$, and let $\rho \in C(R)^*$ be a complete reduction of X such that*

$$\rho : X \;=\; X'$$

Then $\rho^A \in C(R)^*$ is a projection *of ρ on A iff:*

$$[\rho^A] \;=\; \bigsqcup_R \{\sigma \mid \sigma \in CR(X) \text{ and } \sigma \lesssim_R \rho\}$$

The idea of Definition 7.5 is that the projection of

$$X \xrightarrow{\rho}\!\!\!\!\rightarrow X'$$

on A should be the the maximal reduction of A that is a relative prefix of ρ.

Extending the idea of projection to a *partition*, we have the following definition:

Definition 7.6 (Projection on Partition)
Let $\langle \mathcal{B}, \longrightarrow_R, / \rangle$ be an RRS. *Let $\mathcal{S}_X \subseteq \mathcal{B}$ be a partition of $X \in \mathcal{B}$ (Definition 3.33). Let $\rho \in C(R)^*$ be a complete reduction of X. Then $\rho^{\mathcal{S}}$ is a projection of ρ on $\mathcal{S}$ iff there exists a set of projections $\mathcal{T}$ such that*

$$\mathcal{T} \;=\; \{\rho^A \mid A \in \mathcal{S}\}$$

and

$$\rho^{\mathcal{S}} \;\simeq_R\; \rho^{A_1}\rho^{A_2}\cdots\rho^{A_n}$$

where for all $i \in 1\ldots n$, $A_i \in \mathcal{T}$, and for all $i,j \in 1\ldots n$, $T_i \neq T_j$.

Note that the order in which the reductions ρ^{T_1} above are concatenated is irrelevant, since (by disjointness of the elements of a partition) all of their permutations are equivalent under $\simeq_R$.

7.6 Complete Reductions and the Computation of Projections

Contractions of sub-elements that contract maximal subsets of the families contracted in the larger element yield projections:

Theorem 7.2 *Let $\langle \mathcal{B}, \longrightarrow_R, / \rangle$ be an* RRS. *Let $X = A \sqcup B$ be an element of $\mathcal{B}$ such that $A \parallel B$, $\rho \in C(R)^*$ be a complete reduction of X, and* FAM(ρ) *be the set of families contracted in ρ. Then if $\sigma \in C(R)^*$ is a complete reduction of A such that* FAM(σ) *is a maximal subset of* FAM(ρ) *(i.e.,* FAM$(\sigma) \subseteq$ FAM(ρ)*, and there exists no reduction τ of A such that* FAM$(\tau) \subseteq$ FAM(ρ) *and* FAM$(\sigma) \subset$ FAM(τ)*), then*

$$\sigma = \rho^A$$

i.e., σ is a projection of ρ on A.

Proof Since σ is complete and relative to ρ, we have from Proposition 3.48

$$\sigma \lesssim_R \rho$$

(otherwise $(\sigma \vee \rho) \not\simeq_R \rho$, from which we would get FAM$(\rho) \subset$ FAM(σ), which would in turn violate Theorem 3.46). We then note that since σ is maximal with respect to families, it has no extension which is relative to ρ. Thus it is maximal with respect to $\lesssim_R$ and therefore a projection. □

The above theorem will turn out to be the key to actually *computing* a projection.

7.7 Incremental Optimality Theorem

We are now in a position to show that complete and call-by-need reductions in conjunction with projections yield optimal incremental reductions:

Theorem 7.3 (Fundamental Theorem of Incremental Optimality)
Let $\langle \mathcal{B}, \longrightarrow_R, / \rangle$ be an RRS, *$\mathcal{C}$ be a class of reductions of $\|(R)^*$, and $\mathcal{N}$ be a set of normal forms of $\mathcal{B}$.*

Let $X_1 \equiv A \sqcup B$, $X_2 \equiv A \sqcup C$, and A be elements of $\mathcal{B}$ such that $X_1 \overset{A}{\sim} X_2$. Let $\rho \in C(R)^*$ such that be a complete and call-by-need reduction of X_1 such that $\rho : X_1 \in \mathcal{N}$, and $\rho^A \in C(R)^*$ be the projection of ρ on A. Let $\tau \in C(R)^*$ be a complete and call-by-need reduction such that $\rho^A\,\tau$ is a reduction of X_2 and

$$(\rho^A\,\tau) : X_2 \in \mathcal{N}$$

Then $\rho^A\,\tau$ is $\mathcal{C}, \mathcal{N}$ optimal with respect to ρ.

Proof Assume that there exists some pair of reductions $\sigma\tau'$ such that σ is a reduction of A,

$$|\sigma\tau'| \;<|\rho^A\,\tau|$$

and $\sigma \lesssim_R \rho$. Then since τ is complete and call-by-need, it must be the case that

$$\mathrm{FAM}(\sigma)\cap\mathrm{FAM}(\tau) \;\neq\; \emptyset$$

and

$$\mathrm{FAM}(\rho^A) \;\subset\; \mathrm{FAM}(\sigma)$$

But then $\rho^A \lesssim_R \sigma$ and $\rho^A \not\approx_R \sigma$, which contradicts the assumption that ρ^A was a projection and thus maximal with respect to '$\lesssim_R$'. □

Theorem 7.3 shows that projections are indeed the key requirement for performing optimal incremental reductions.

Chapter 8

Incremental Reduction

"[It is] an art manifested in works of art, not an exact *a priori* science like Palmistry or Judicial Astrology..."

"...and if we are not impatient, the contrast of the episodic development and the novel strokes...in the recapitulation will gain by repetition, welding the total diversity into a more lyric unity."

—Tovey

In this chapter, we gather up the the vari-colored threads that have been spun in chapters past, knitting them together to produce our principal result: Λ^{inc}, an incremental reduction algorithm for ΛCCL, and, by translation, the lambda calculus. Our aim is to combine the formal theory of incremental reduction developed in Chapter 7, the notions of reduction on graphs and trees covered in Chapters 3 and 4, the theory of optimality discussed in Chapter 6, and the practical advantages of the notion of reduction using environments and closures given in Chapter 5 to yield an incremental reduction algorithm that is provably efficient *as well as* practical.

We will begin with a relatively informal introduction that outlines our approach to the problem, appealing directly to the λ-calculus and the informal notion of graph reduction used in Chapter 2. We will then formalize our ideas using several related variants of ΛCCL. We will show that one of these variants, ΛCCL$'$, yields an optimal, weak head normalizing reduction algorithm. We then introduce a final variant of ΛCCL, $\triangle\Lambda$CCL, which yields an optimal *incremental* weak head normalizing reduction algorithm, Λ^{inc}.

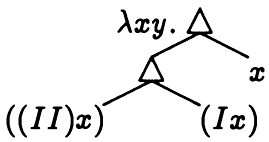
8.1 Informal Presentation of Incremental Reduction

8.1.1 Data Structures

In representing λ-terms for the purpose of incremental reduction, we will add to our repertoire of devices—already consisting of graphs, environments, and closures—the new notion of a *fork* node: $\triangle$. Forks allow certain *sets* of λ-terms to be represented by a single graph. For example, let:

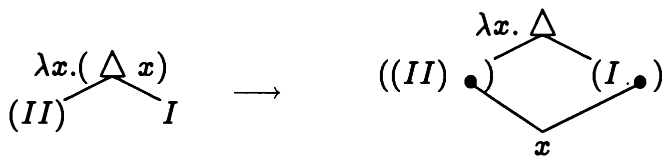
$$\begin{aligned} Q_1 &\equiv \lambda xy.(I\ I\ x) \\ Q_2 &\equiv \lambda xy.Ix \\ Q_3 &\equiv \lambda xy.x \end{aligned}$$

These terms could be represented simultaneously as follows:



Depending on whether the right or left branches of the two $\triangle$ nodes are inspected, different terms may be read off. In the sequel, the two branches will always represent β-equivalent terms (as is the case above).

Note that the arguments of an application whose function part is a $\triangle$ node can be "distributed" through the $\triangle$ to yield an alternate representation of the same set of terms, e.g.:



8.1.2 Example

Let us return to the example of Section 2.5 to illustrate how the reduction algorithm Λ^{inc} works, where, as before, we are given the mutually similar terms

$$\begin{aligned} M_1 &\equiv N[z := \lambda xy.xy] \\ M_2 &\equiv N[z := \lambda xy.x] \\ M_3 &\equiv N[z := \lambda xy.y] \end{aligned}$$

We begin by reducing M_1 to normal form, using the following graph to represent the initial term:

$$M_1 \equiv \quad ((\underset{\lambda xy.xy}{\triangle}(II))(II))$$

Note that the two terms represented by the graph above are the same, and that the subterm $\lambda xy.xy$ is *shared* by both arms of the $\triangle$.

We then reduce M_1 as shown in Figure 8.1, using a variant of the outermost graph-reduction strategy of Wadsworth, where all steps are either β-contractions (on possibly shared terms) or the distribution of arguments of applications ("dist." in Figure 8.1) through $\triangle$ nodes. The node being reduced at each step is boxed. At all times, the graph represents a pair of β-equivalent terms, depending on whether only left branches or only right branches are inspected. The pair of terms represented by the graph at each stage of the reduction process is listed next to each graph.

When reduction of M_1 is complete, we are left with a term headed by a $\triangle$ node whose left subtree is the normal form of M_1 and whose right subtree is the term

$$(z\,I\,I)[z := P_1] \equiv N'[z := P_1]$$

If we disregard the substituend P_1, we can regard the right subtree as representing N', the desired final term of the *projection* of the reduction on N.

In general, Λ^{inc} will operate by reducing an initial term whose substituend is replaced by a $\triangle$ node (the branches of which both point to the substituend). When a $\triangle$ node is encountered in the function part of an application during reduction, its argument is distributed into each branch of the $\triangle$, effectively creating two copies of the application. We can view the $\triangle$ node as a sort of "checkpoint" in the reduction process, which preserves the redex whose contraction depends on the value of the substituend (i.e., a redex whose function part is the substituend). Reduction then proceeds only on the *left* branch of the $\triangle$. However, contraction of a redex in the left subterm of a $\triangle$ may have the beneficent side-effect of contracting a redex in the *right* subterm as well, if the redex is shared. The trick is to perform the reduction in such a way that the maximal number of such shared contractions take place.

The combination of graph reduction and appropriate use of the $\triangle$ node is crucial to performing incremental reduction: The $\triangle$ node enables a subterm whose reduction is *dependent* upon the substituend to be preserved, while graph reduction enables redex contractions that are *disjoint* of the substituend to affect both terms reachable from a $\triangle$.

Continuing with the example above, if we remove the topmost $\triangle$ node, discard

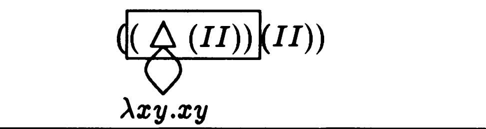
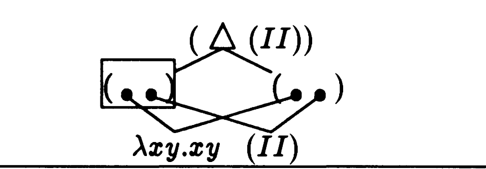
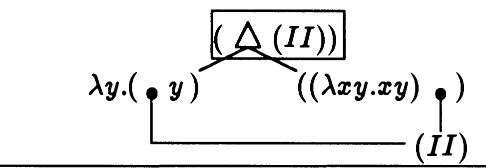
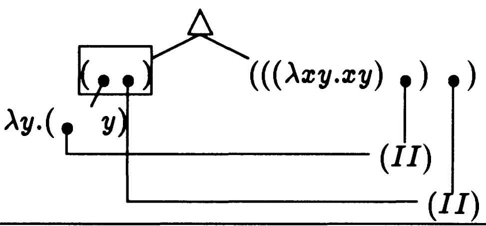
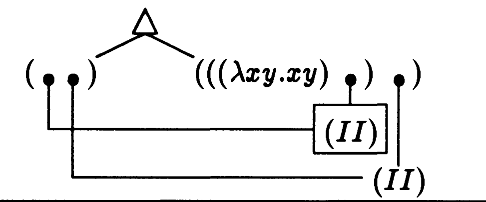
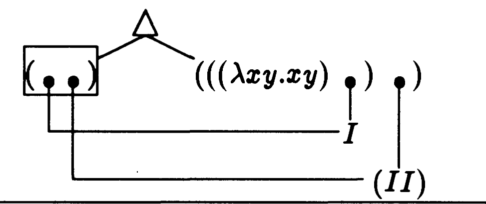
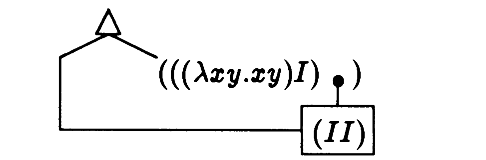
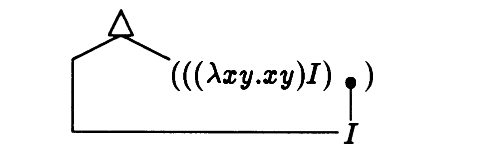
Graph	Op.	Corresponding Terms	
		Left Δ-Subterms	Right Δ-Subterms
	dist.	$((\lambda xy.xy)(II)(II))$	$((\lambda xy.xy)(II)(II))$
	β	$((\lambda xy.xy)(II)(II))$	$((\lambda xy.xy)(II)(II))$
	dist.	$((\lambda y.(II)\,y)(II))$	$((\lambda xy.xy)(II)(II))$
	β	$((\lambda y.(II)\,y)(II))$	$((\lambda xy.xy)(II)(II))$
	β	$((II)(II))$	$((\lambda xy.xy)(II)(II))$
	β	$(I\,(II))$	$((\lambda xy.xy)\,I\,(II))$
	β	(II)	$((\lambda xy.xy)\,I\,(II))$
	β	I	$((\lambda xy.xy)\,I\,I)$

Figure 8.1: Reduction of M_1

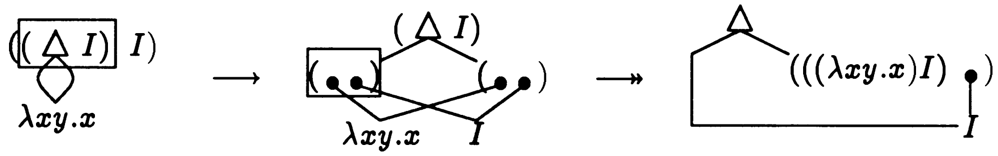
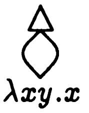


Figure 8.2: Contraction of $N'[z := P_2]$

its left subterm, and replace P_1 by



in the (former) right subterm of the topmost $\triangle$, we form the following new term (disregarding the $\triangle$ node)

$$N'[z := P_2]$$

We then reduce the new term, taking advantage of the fact that N' embodies the reduction already performed in N above, and avoiding (as was our aim) contraction of overlapping redexes. The new reduction is given in Figure 8.2. We can repeat the process as required for additional substituends P_3, P_4.... Note that in the example above, the final graph is the same as that of the reduction in Figure 8.1. In general, however, each substituend may induce additional reductions in the common term (which was not possible above since the common term $N' \equiv (z\,I\,I)$ was already reduced to normal form by the first reduction).

Each time reduction is to be performed in the presence of a new substituend, the initial term is reconstructed by replacing each reference to a fork node with a reference to its *right* subterm, thus preserving the "checkpoints" on which the substituend depended. A new substituend (along with an appropriately configured $\triangle$ node) replaces the old one and the reduction process begins anew.

Unfortunately, Wadsworth-style graph reduction is not completely satisfactory in the case illustrated in Figure 8.3. There we are unable to take advantage of the fact that the redex Ia is a residual of the Iy redex in $\lambda y.Iy$. The Iy redex is left uncontracted in the final term. This phenomenon is related to the issue of "full-laziness" in functional language implementations. Although we could have contracted the Iy redex *before* applying $\lambda y.Iy$ to a, we wish to retain an outermost reduction strategy in order to guarantee termination.

By using shared closures and environments, the (Iy) redex can be contracted independently of the substitution of a for y, even though the reduction is nominally outermost, as we see in Figure 8.4. In addition, by using environments, substitution can be implemented efficiently.

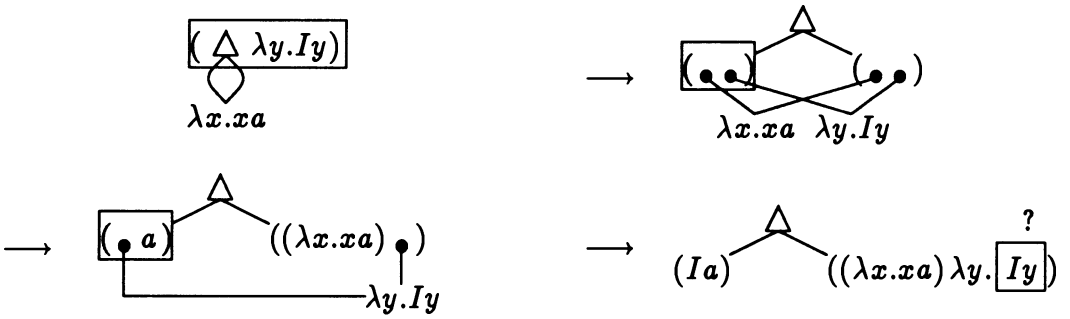


Figure 8.3: Shortcomings of Outermost Graph Reduction

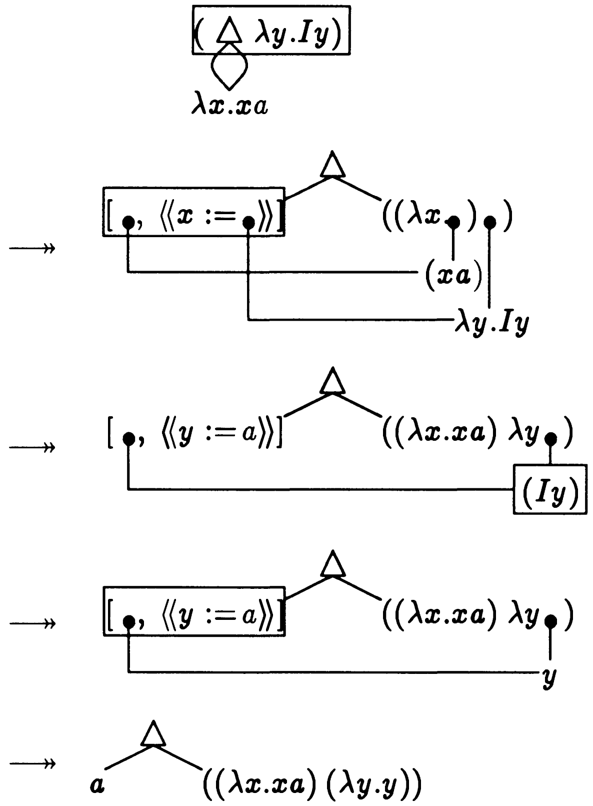


Figure 8.4: Reduction using Closures and Environments

8.1.3 Multiple Substituends

The discussion above was concerned with the incremental reduction of terms differing in a single substituend. However, as the example of Section 2.6 illustrates, it is much more useful to consider the possibility of multiple, and perhaps nested, substituends. It turns out that $\Lambda^{\mathbf{inc}}$ can be extended almost trivially to deal with this new model. In this section, we will provide an overview of the concepts involved.

We will now assume that we are given a λ-term M such that

$$M \equiv N[z_1 := M_1, \ldots, z_n := M_n],\ n \geq 0$$

where each M_i has the form

$$M_i \equiv N_i[z_{i1} := M_{i1}, \ldots, z_{in_i}, := M_{in_i}],\ n_i \geq 0$$

and each of the M_{ij} are assumed have a similar form, etc.

After reducing M to $M' \in \mathit{WHNF}_\beta$, we would like to replace some set of disjoint substituends

$$M_{\vec{v}_1}, \ldots, M_{\vec{v}_m}$$

in M, and reduce the resulting new term.

Example 8.1 Consider, for example, the term $M \equiv (\lambda y.yy\ (I\,(Ia)))$, in which we designate two substituends z_1 and z_2 as follows:

$$\begin{aligned} M &\equiv (z_1\,(I\,z_2))[z_1 := \lambda y.yy, z_2 := (Ia)] \\ &\equiv N[z_1 := P_1, z_2 := P_2] \end{aligned}$$

To reduce this term incrementally, we start by replacing the substituends z_1 and z_2 by terms with $\triangle$ nodes, as before. However, we now *index* the $\triangle$ nodes by a number corresponding to the index of the substituends, as shown by the term A in Figure 8.5. We then proceed to reduce it as before. The indices in $\triangle$ nodes only come into play when a subterm is replaced and reduction is to be performed anew. The result of the reduction is summarized in Figure 8.5 (using our informal notation for environments and closures). Though the final term of the reduction, A', is a bit complicated, its interpretation is straightforward: If we read off only the left branches of the $\triangle$s, the result is the term

$$(aa)$$

Reading off only right branches yields

$$(z_1\,z_2)[z_1 := \lambda y.yy, z_2 := a] \equiv N'[z_1 := P_1', z_2 := P_2']$$

Note that the reduction above preserves the contraction of the (Iz_1) redex in N and the (Ia) redex in P_2 (there are no redexes in P_1 to preserve).

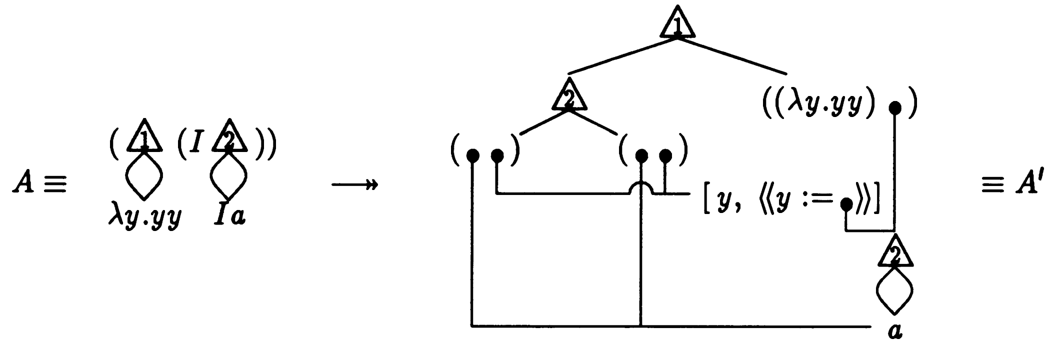


Figure 8.5: Reduction with Indexed $\triangle$ nodes

8.1.4 Overlapping Contexts

Consider the term

$$\begin{aligned} M &\equiv (z_1\,((\lambda x.xa)\ z_2))[z_1 := I, z_2 := I] \\ &\equiv N[z_1 := P_1, z_2 := P_2] \end{aligned}$$

where we reduce M by $\rho: M \twoheadrightarrow a$. In this case, not only do we want Λ^{inc} to compute the projection of ρ on N, P_1, and P_2 as before, but since we are now allowed to replace arbitrary combinations of substituends, we need to compute the projection of ρ on *every* context that could contain some subset of the substituends. For example, if we replace only substituend P_1, we need to ensure that we compute the projection of ρ on Q, where we write M as

$$M \equiv (z_1\,((\lambda x.xa)\ I))[z_1 := I] \equiv Q[z_1 := P_1]$$

Note that the projection ρ^Q is *not* the concatenation of projections ρ^N and ρ^{P_2}, since together, N and P_2 create a redex, (Ia), that does not exist separately in either term.

It will turn out that by choosing the appropriate path through the graph according to the values of the indices encountered in $\triangle$s, we can reconstruct the (reduced) context containing that set of substituends. Thus with very little additional effort, we not only get the projection of the reduction on a single context, but the projection on *all* possible contexts simultaneously.

In the sequel, we will work directly with terms of **ΛCCL**, rather than λ-terms. Furthermore, the notion of substituend will be defined using Brouwerian algebra operations, rather than the substitution operation used informally above.

8.2 Regular Variants of ΛCCL

Example 8.1 illustrated the desirability of using closures and environments in implementing incremental reduction so as to fully preserve the sharing required to capture the contractions in a projection. It should by now be clear that ΛCCL is the right formal framework for capturing these ideas. However, ΛCCL turns out to be a bit too general for our needs. In particular, ΛCCL is not an RRS, since it contains many critical pairs. Thus our analysis of incremental reduction in Chapter 7 using projections does not apply. Furthermore, the set of environment operations in ΛCCL is broader than necessary to reduce environments produced in by simulated β-reduction—we want to restrict the set of operations on environments to those that actually arise when implementing leftmost λ-reduction.

To answer these concerns, we will define a variant of ΛCCL we will call ΛCCL$'$. ΛCCL$'$ is a *conditional* regular term rewriting system, and thus an RRS. However, since its rewriting relation is simply a restriction of that of ΛCCL, most of the (desirable) properties of ΛCCL apply to ΛCCL$'$ as well.

8.2.1 ΛCCL$'_0$

In order to motivate the rules of ΛCCL$'$, we first define a slightly simpler version of the system, which we will call ΛCCL$'_0$:

Definition 8.1 (ΛCCL$'_0$)
ΛCCL$'_0$ is the conditional rewriting system whose terms are Ter(ΛCCL)*, and whose rules are given by Figure 8.6.*

We note that since all the rules of ΛCCL$'_0$ are either identical to those of ΛCCL, or are *restricted instances* of ΛCCL rules, ΛCCL is trivially a conservative extension of ΛCCL$'_0$. Note that the restricted rules are **(AssC$'$)**, **(NullER$'$)**, **(DApply$'$)**, **(AssE$'$)**, and **(NullC$'$)**.

The most important restricted rule is **(DApply$'$)**. It effectively disallows substitution inside the body of a term not yet in weak head normal form. This restriction has the effect of forbidding the contraction of *created* **(Beta)** redexes whose creator is not yet in weak head normal form. This restriction turns out to be quite natural, and is not by any means an impediment to practical application.

We note that ΛCCL$'_0$ is well-behaved, in the following sense:

Lemma 8.1 ΛCCL$'_0$ *is a regular conditional term rewriting system*

Proof ΛCCL$'_0$ is left-linear, and it is easily seen that ΛCCL$'_0$ has no conditional critical pairs. By Definition 4.30, clause (3), it remains to show that the unconditional residuals of ΛCCL$'_0$ commute, i.e., that the guards for those residuals always hold. The only conditional rules are **(AssC$'$)**, **(AssE$'$)**, and **(DApply$'$)**. For the first two, the guard requires that the redex be a normal form; once it holds,

$$
\begin{array}{ll}
\textbf{(Beta)} & \mathrm{Apply}(\Lambda(A),\ B) \longrightarrow_{\Lambda\mathrm{CCL}_0'} [A,\ \langle\emptyset,\ B\rangle] \\[2ex]
\textbf{(AssC}\,'\textbf{)} & \dfrac{E_1 \equiv \Box^n}{[[\mathrm{Var},\ E_1],\ E_2] \longrightarrow_{\Lambda\mathrm{CCL}_0'} [\mathrm{Var},\ E_1 \circ E_2]} \\[2ex]
\textbf{(NullER}\,'\textbf{)} & \Box \circ \emptyset \longrightarrow_{\Lambda\mathrm{CCL}_0'} \Box \\[2ex]
\textbf{(ShiftE)} & \Box \circ \langle E,\ A\rangle \longrightarrow_{\Lambda\mathrm{CCL}_0'} E \\[2ex]
\textbf{(VarRef)} & [\mathrm{Var},\ \langle E,\ A\rangle] \longrightarrow_{\Lambda\mathrm{CCL}_0'} A \\[2ex]
\textbf{(D}\Lambda\textbf{)} & [\Lambda(A),\ E] \longrightarrow_{\Lambda\mathrm{CCL}_0'} \Lambda([A,\ \langle E \circ \Box,\ \mathrm{Var}\rangle]) \\[2ex]
\textbf{(DE)} & \langle E_1,\ A\rangle \circ E_2 \longrightarrow_{\Lambda\mathrm{CCL}_0'} \langle E_1 \circ E_2,\ [A,\ E_2]\rangle \\[2ex]
\textbf{(DApply}\,'\textbf{)} & \dfrac{A \in \mathit{WHNF}_{\Lambda\mathrm{CCL}},\ A \equiv \mathrm{Apply}(B,\ C)}{[A,\ E] \longrightarrow_{\Lambda\mathrm{CCL}_0'} \mathrm{Apply}([B,\ E],\ [C,\ E])} \\[2ex]
\textbf{(AssE}\,'\textbf{)} & \dfrac{E_1 \equiv \Box^n}{(\Box \circ E_1) \circ E_2 \longrightarrow_{\Lambda\mathrm{CCL}_0'} \Box \circ (E_1 \circ E_2)} \\[2ex]
\textbf{(NullC}\,'\textbf{)} & [\mathrm{Var},\ \emptyset] \longrightarrow_{\Lambda\mathrm{CCL}_0'} \mathrm{Var}
\end{array}
$$

Figure 8.6: $\Lambda\mathrm{CCL}_0'$ Reduction Rules

nothing can invalidate it. Rule (**DApply**$'$) is a bit trickier since the guard does not require its unconditional redex to be a strict normal form. However, we note that members of $WHNF_{\Lambda\mathrm{CCL}}$ have a similar property: once a ΛCCL term is in $WHNF_{\Lambda\mathrm{CCL}}$, it remains in $WHNF_{\Lambda\mathrm{CCL}}$ after any subsequent $\Lambda\mathrm{CCL}'_0$ reduction. Therefore, once the guard holds, it holds for all residuals as well. □

As a corollary, we then get:

Corollary 8.2 *$\Lambda\mathrm{CCL}'_0$ is confluent.*

Proof Immediate from Lemma 8.1 and Theorem 4.7. □

Note that unlike the unconditional rewriting system ΛCCL, $\Lambda\mathrm{CCL}'_0$ is confluent on *non-ground terms*, as well as ground terms.

8.2.2 $\Lambda\mathrm{CCL}'$

$\Lambda\mathrm{CCL}'_0$ is regular, but due to projection rules (**ShiftE**), and (**VarRef**), it is not strongly regular. Since we want to implement *complete* reductions using term graph rewriting, we need a term rewriting system that can be implemented by a regular graph rewriting system. Therefore, we know from Theorem 4.22, a good starting point would be a *strongly* regular term rewriting system.

To get a strongly regular system, we replace the projection rules of $\Lambda\mathrm{CCL}'_0$ with sets of rules that contain instances of each operator that might occur in the term projected out. These new rules are also appropriately restricted to ensure that regularity is maintained. The result is the system $\Lambda\mathrm{CCL}'$, defined as follows:

Definition 8.2 ($\Lambda\mathrm{CCL}'$)
The rules of $\Lambda\mathrm{CCL}'$ are those of $\Lambda\mathrm{CCL}'_0$, with

(1) (**ShiftE**) *replaced by the rules* (**ShiftE**$'_1$), (**ShiftE**$'_2$), *and* (**ShiftE**$'_3$) *depicted in Figure 8.7.*

(2) (**VarRef**) *replaced by the rules* (**VarRef**$'_1$), (**VarRef**$'_2$), (**VarRef**$'_3$), *and* (**VarRef**$'_4$) *depicted in Figure 8.8.*

We define the $\Lambda\mathrm{CCL}'$ analogue of ΛCCL's "substitutive" subsystem ECCL as follows:

Definition 8.3 (ECCL')
The axioms of ECCL' are those of $\Lambda\mathrm{CCL}'$ without rule (**Beta**).

8.2.3 Properties of $\Lambda\mathrm{CCL}'$

We can now show that $\Lambda\mathrm{CCL}'$ has a number of desirable properties, beginning with:

$$(\textbf{ShiftE}'_1) \qquad \Box \circ \langle\langle E,\ A_1\rangle,\ A_2\rangle \ \longrightarrow_{\Lambda\mathrm{CCL}'}\ \langle E,\ A_1\rangle$$

$$(\textbf{ShiftE}'_2) \qquad \Box \circ \langle \emptyset,\ A\rangle \ \longrightarrow_{\Lambda\mathrm{CCL}'}\ \emptyset$$

$$(\textbf{ShiftE}'_3) \qquad \frac{E \equiv \Box^n}{\Box \circ \langle E,\ A\rangle \ \longrightarrow_{\Lambda\mathrm{CCL}'}\ E}$$

Figure 8.7: Alternate Rules for **(ShiftE)**

$$(\textbf{VarRef}'_1) \qquad [\mathrm{Var},\ \langle E,\ \mathrm{Var}\rangle] \ \longrightarrow_{\Lambda\mathrm{CCL}'}\ \mathrm{Var}$$

$$(\textbf{VarRef}'_2) \qquad [\mathrm{Var},\ \langle E,\ \Lambda(A)\rangle] \ \longrightarrow_{\Lambda\mathrm{CCL}'}\ \Lambda(A)$$

$$(\textbf{VarRef}'_3) \qquad \frac{A \in \mathit{WHNF}_{\Lambda\mathrm{CCL}}}{[\mathrm{Var},\ \langle E,\ \mathrm{Apply}(A,\ B)\rangle] \ \longrightarrow_{\Lambda\mathrm{CCL}'}\ \mathrm{Apply}(\boldsymbol{A},\ \boldsymbol{B})}$$

$$(\textbf{VarRef}'_4) \qquad \frac{E_2 \equiv \Box^n}{[\mathrm{Var},\ \langle E,\ [\mathrm{Var},\ E_2]\rangle] \ \longrightarrow_{\Lambda\mathrm{CCL}'}\ [\mathrm{Var},\ E_2]}$$

Figure 8.8: Alternate Rules for **(VarRef)**

Lemma 8.3 $\mathbf{\Lambda CCL'}$ *is a strongly regular conditional term rewriting system.*

Proof $\mathbf{\Lambda CCL'}$ is left-linear, has no projection rules, and no conditional critical pairs. The argument for the commutativity of residuals is similar to that of Lemma 8.1. □

As with $\mathbf{\Lambda CCL'_0}$, we then have:

Corollary 8.4 $\mathbf{\Lambda CCL'}$ *is confluent.*

Since $\mathbf{\Lambda CCL'}$ is conservatively extended by $\mathbf{\Lambda CCL}$, the following is trivial:

Theorem 8.5 $\mathbf{ECCL'}$ *is noetherian.*

Proof **ECCL** is noetherian by Theorem 5.1. Since Every $\mathbf{ECCL'}$ rule is an instance of a **ECCL** rule, $\mathbf{ECCL'}$ must be noetherian as well. □

We recall from Chapter 5 that the set of environment forms *LEF* generated by simulated λ-reduction can be reduced to *WPENF* with a limited set of rules. The analogous result of $\mathbf{ECCL'}$ is as follows:

Theorem 8.6 *All elements of LEF are reducible to a weak partial environment normal form using the rules of* ECCL'. *That is, for all* $E_1 \in LEF$, *there exists* $E_2 \in WPENF$ *such that*

$$E_1 \quad \twoheadrightarrow_{\mathrm{ECCL}'} \quad E_2$$

Proof We observe that for every term E_1' such that $E_1' \in LEF$, $E_1' \notin WPENF$, and

$$E_1 \quad \twoheadrightarrow_{\mathrm{ECCL}'} \quad E_1'$$

there is a rule (R) of $\mathbf{ECCL'}$ that is applicable to E_1' such that

$$E_1' \quad \twoheadrightarrow_{(R)} \quad E_1''$$

and $E_1'' \in LEF$. Since we know from Theorem 8.5 that $\mathbf{ECCL'}$ is noetherian, the reduction of E_1 must terminate with a term $E_2 \in WPENF$. □

In order to pursue an optimal reduction strategy, we must ensure that each set of parallel redexes contracted at any step contains some needed redex. Thus we will need the following.

Lemma 8.7 *The leftmost outermost* $\mathbf{\Lambda CCL'}$ *redex of any term of* $GTer_{\mathcal{L}}(\Lambda\mathrm{CCL})$ *is* $WHNF_{\Lambda\mathrm{CCL}}$*-needed. Furthermore, if A is a term of the form*

$$A \equiv [\mathrm{Var},\ \langle E,\ B\rangle]$$

then the leftmost outermost $\mathbf{\Lambda CCL'}$ *redex of B is* $WHNF_{\Lambda\mathrm{CCL}}$*-needed.*

Proof By careful examination of the rules, we note that every term

$$E \in GTer_{\mathcal{E}}(\Lambda\text{CCL})$$

such that $E \notin WPENF$ has a *unique* outermost redex. We also note that for any term $A \in GTer_{\mathcal{L}}(\mathbf{\Lambda CCL})$ such that $A \notin WHNF_{\Lambda\text{CCL}}$, the leftmost outermost redex is always needed (since $WHNF_{\Lambda\text{CCL}}$ requires that leftmost subterms also be elements of $WHNF_{\Lambda\text{CCL}}$). Finally, we note for that any term $A \equiv [\text{Var}, E]$ such that $E \notin WPENF$, a redex B of A is $WHNF_{\Lambda\text{CCL}}$-needed iff B is a $WPENF$-needed redex of E. Therefore, every leftmost outermost redex is needed. $\square$

We then have the following corollary:

Corollary 8.8 *Any leftmost outermost* $\mathbf{\Lambda CCL'}$ *reduction strategy for* $\mathbf{\Lambda CCL}$ *elements of* $GTer_{\mathcal{L}}(\Lambda\text{CCL})$ *is* $WHNF_{\Lambda\text{CCL}}$*-call-by-need.*

8.2.4 Optimal Reduction in $\mathbf{\Lambda CCL'}$

Gathering up the facts from the previous section, we first note that $\mathbf{\Lambda CCL'}$ can simulate weak head normalizing β-reductions:

Theorem 8.9

$$\langle \mathbf{\Lambda CCL'},\ [\![\cdot]\!]_{\Lambda\text{CCL}},\ [\![LNF(\cdot)]\!]_{\lambda} \rangle$$

is an implementation of β *(on* $GTer(\lambda^{\text{DB}})$*), and is complete with respect to* $WHNF_{\beta}$.

Proof By Lemma 5.19, we know that $\Lambda\text{CCL}'$ is sound, i.e., that for all $M \in GTer(\lambda^{\text{DB}})$ and $B' \in GTer_{\mathcal{L}}(\Lambda\text{CCL})$ such that

$$[\![M]\!]_{\Lambda\text{CCL}} \quad \twoheadrightarrow_{\Lambda\text{CCL}'} \quad B'$$

we have

$$M \quad \twoheadrightarrow_{\beta} \quad [\![LNF(B')]\!]_{\lambda}$$

To show that $\mathbf{\Lambda CCL'}$ is complete with respect to $WHNF_{\beta}$, we must show that for all $M, M' \in GTer(\lambda^{\text{DB}})$ such that

$$M \quad \twoheadrightarrow_{\beta} \quad M'$$

there exists $B' \in GTer_{\mathcal{L}}(\Lambda\text{CCL})$ such that

$$[\![M]\!]_{\Lambda\text{CCL}} \quad \twoheadrightarrow_{\Lambda\text{CCL}'} \quad B'$$

and $[\![LNF(B')]\!]_{\lambda} \in WHNF_{\beta}$.

Assume then that M has a weak head normal form. We then note that the reduction strategy for $[\![M]\!]_{\Lambda\text{CCL}}$ that always contracts the leftmost outermost redex must eventually either yield a term $B' \in \mathit{WHNF}_{\Lambda\text{CCL}}$, or infinitely often contract a weak head **(Beta)** redex (since the strategy is leftmost outermost and $\mathbf{ECCL}'$ is noetherian). In the first case, we have $[\![\mathit{LNF}(B')]\!]_\lambda \in \mathit{WHNF}_\beta$, as desired. In the second case, we have from the proof of Theorem 5.24 that M must have no weak head normal form, contradicting the assumption to the contrary. Therefore, a leftmost outermost strategy always yields an element of $\mathit{WHNF}_{\Lambda\text{CCL}}$, and via the translation $[\![\mathit{LNF}(\cdot)]\!]_\lambda$, an element of WHNF_β, if one exists. □

8.2.5 Term Graph Reduction and $\mathbf{\Lambda CCL}'$

Since $\mathbf{\Lambda CCL}'$ is a strongly regular term rewriting system, we have by Theorem 4.22 that there exists a corresponding regular graph rewriting system that is a complete implementation of $\mathbf{\Lambda CCL}'$. We thus make the following definition:

Definition 8.4 **($\mathbf{\Lambda CCL}^{G'}$)**
$\mathbf{\Lambda CCL}^{G'}$ *will denote the regular graph rewriting system given by Theorem 4.22 that is a complete, family-preserving implementation of* $\mathbf{\Lambda CCL}'$.

8.2.6 An Optimal Reduction Algorithm for $\mathbf{\Lambda CCL}'$

Now that we have a family preserving implementation of $\mathbf{\Lambda CCL}'$, we need a reduction strategy that is call-by-need in order to get an *optimal* implementation. For this, we use two algorithms that implement the desired strategy: *rwhnf2*$(\cdot)$ and *rwpenf*$(\cdot)$, depicted respectively in Figure 8.9 and Figure 8.10. *rwhnf2*$(\cdot)$ takes as argument a term of $G\mathrm{Ter}_{\mathcal{L}}(\Lambda\text{CCL}')$, applies a leftmost outermost reduction strategy, and terminates with a term in $\mathit{WHNF}_{\Lambda\text{CCL}}$, if any exists. *rwpenf*$(\cdot)$ is an auxiliary algorithm for reducing terms of $G\mathrm{Ter}_{\mathcal{E}}(\Lambda\text{CCL})$ to *WPENF*, and also embodies a leftmost outermost strategy. Note that *rwhnf2*$(\cdot)$ and *rwpenf*$(\cdot)$ together use every rule of $\mathbf{\Lambda CCL}'$. The notation used for these algorithms is that discussed in Section 5.5.1.

We now have the following theorem:

Theorem 8.10 *The implementation/strategy pair*

$$\langle \mathbf{\Lambda CCL}^{G'}, \mathit{rwhnf2}(\cdot)\rangle$$

is a $\mathit{WHNF}_{\Lambda\text{CCL}}$*-optimal implementation of* $\mathbf{\Lambda CCL}'$ *for terms of*

$$G\mathrm{Ter}_{\mathcal{L}}(\Lambda\text{CCL}')$$

Proof $\mathbf{\Lambda CCL}^{G'}$ is a regular term graph rewriting systems that strongly implements the system $\mathbf{\Lambda CCL}'$. Therefore, every $\mathbf{\Lambda CCL}^{G'}$ contraction simulates

rwhnf2$(T) \equiv$
case T **of**
  $\Lambda(A)$: **skip**; $\{T \in \mathit{WHNF}_{\Lambda\mathrm{CCL}}\}$
  Var: **skip**; $\{T \in \mathit{WHNF}_{\Lambda\mathrm{CCL}}\}$
  Apply(A, B): **seq**
    rwhnf2(A);
    case A **of**
      $\Lambda(A')$: **seq**
        $T := [A', \langle\emptyset, B\rangle]$; $\{\mathit{rule}$ **(Beta)**$\}$
        rwhnf2(T)
        endseq;
      otherwise: **skip** $\{T \in \mathit{WHNF}_{\Lambda\mathrm{CCL}}\}$
    endseq;
  $[L, E]$: **seq**
    rwhnf2(L);
    case L **of**
      Apply(A, B): **seq**
        $T :=$ Apply$([A, E], [B, E])$; $\{\mathit{rule}$ **(DApply$'$)**$\}$
        rwhnf2(T)
        endseq;
      $\Lambda(A)$: $T := \Lambda([A, \langle E \circ \square, \mathrm{Var}\rangle])$;
        $\{\mathit{rule}$ **(DΛ)**; $T \in \mathit{WHNF}_{\Lambda\mathrm{CCL}}\}$
      $[L_1, E_1]$: **seq**
        $T := [L_1, E_1 \circ E]$; $\{\mathit{rule}$ **(AssC$'$)**$\}$
        rwhnf2(T)
        endseq;
      Var: **seq**
        rwpenf(E);
        case E **of**
          $\emptyset$: $T :=$ Var; $\{\mathit{rule}$ **(NullC$'$)**$\}$
          $\square^n$: **skip**; $\{T \equiv n!;\ T \in \mathit{WHNF}_{\Lambda\mathrm{CCL}}\}$
          $\langle E, A\rangle$: **seq**
            rwhnf2(A);
            $T := \mathit{copyTop}(A)$ $\{\mathit{rules}$ **(VarRef$'_i$)**, $i \in 1 \ldots 4\}$
            endseq
        endseq
    endseq

Figure 8.9: Algorithm *rwhnf2*$(\cdot)$

$$
\begin{array}{ll}
\textit{rwpenf}(E) \equiv & \{E \in \textit{LEF}\} \\
\quad \textbf{case } E \textbf{ of} & \\
\qquad \emptyset\text{: } \textbf{skip}; & \{E \in \textit{WPENF}\} \\
\qquad \Box^n\text{: } \textbf{skip}; & \{E \in \textit{WPENF}\} \\
\qquad \langle E_1,\, A\rangle\text{: } \textbf{skip}; & \{E \in \textit{WPENF}\} \\
\qquad E_1 \circ E_2\text{: } \textbf{seq} & \\
\qquad\quad \textit{rwpenf}(E_1); & \\
\qquad\quad \textbf{case } E_1 \textbf{ of} & \\
\qquad\qquad \Box\text{: } \textbf{case } E_2 \textbf{ of} & \\
\qquad\qquad\quad \emptyset\text{: } E := \emptyset; & \{\textit{rule } (\mathbf{NullER}')\} \\
\qquad\qquad\quad \langle E_3,\, A\rangle\text{: } \textbf{seq} & \\
\qquad\qquad\qquad \textit{rwpenf}(E_3); & \\
\qquad\qquad\qquad E := \textit{copyTop}(E_3) & \{\textit{rule } (\mathbf{ShiftE}'_i),\ i \in 1\ldots 3\} \\
\qquad\qquad\qquad \textbf{endseq} & \\
\qquad\qquad\quad E_3 \circ E_4\text{: } \textbf{seq} & \\
\qquad\qquad\qquad \textit{rwpenf}(E_2); & \\
\qquad\qquad\qquad \textit{rwpenf}(E) & \\
\qquad\qquad\qquad \textbf{endseq}; & \\
\qquad\qquad E_3 \circ E_4\text{: } E := E_3 \circ (E_4 \circ E_2); & \\
 & \{E_3 \equiv \Box;\ E_4 \equiv \Box^n;\ \textit{rule } (\mathbf{AssE}')\} \\
\qquad\qquad \langle E_3,\, A\rangle\text{: } E := \langle E_3 \circ E_2,\, [A,\, E_2]\rangle & \{\textit{rule } (\mathbf{DE})\} \\
\qquad\quad \textbf{endseq} &
\end{array}
$$

Figure 8.10: Algorithm $\textit{rwpenf}(\cdot)$

a complete contraction of $\mathbf{\Lambda CCL}'$. By Corollary 8.8 *rwhnf2*$(\cdot)$ implements a $\mathit{WHNF}_{\Lambda\mathrm{CCL}}$-call-by-need strategy. Therefore, by Theorem 6.4,

$$\langle \mathbf{\Lambda CCL}^{G'},\ \mathit{rwhnf2}(\cdot)\rangle$$

is a $\mathit{WHNF}_{\Lambda\mathrm{CCL}}$-optimal implementation of $\mathbf{\Lambda CCL}'$ for terms of $\mathit{GTer}_{\mathcal{L}}(\Lambda\mathrm{CCL}')$. $\square$

Despite all of its nice properties, we note that $\mathbf{\Lambda CCL}'$ still does not provide a vehicle for implementation of optimal *β-reduction*: While $\mathbf{\Lambda CCL}^{G'}$ implements optimal reduction *within* the system $\mathbf{\Lambda CCL}'$, the families of β are not preserved when mapped to reductions (even complete reductions) of $\mathbf{\Lambda CCL}'$. Thus while $\mathbf{\Lambda CCL}^{G'}$ optimally implements $\mathbf{\Lambda CCL}'$, and $\mathbf{\Lambda CCL}'$ implements β-reduction, the two together do *not* imply that $\mathbf{\Lambda CCL}^{G'}$ optimally implements β. Nonetheless, we believe that $\mathbf{\Lambda CCL}^{G'}$ is at least as efficient as any existing λ-interpreter based on graph reduction, since amount of sharing of **(Beta)** redexes is, if anything, greater than in most graph reduction schemes.

8.3 Simultaneous Representation of Multiple Partitions

To get an optimal incremental reduction algorithm from an optimal reduction algorithm, we know from Chapter 7 that we must preserve *projections* of reductions on elements of a term that are unchanged by some incremental update.

Let us recall from Chapter 7 how our model for incremental reduction works: Assume that ρ is a reduction of some element $A \sqcup B$ such that $A \parallel B$,

$$\rho\colon(A \sqcup B) \;\; = \;\; X'$$

and such that one of A or B may later be updated.

If we subsequently wish to reduce, say $C \sqcup B$ or $A \sqcup D$ to some normal form, we want to take advantage of the previous reduction ρ by preserving the *projection* of ρ on *both* A and B (since either one might later be replaced). If we decide subsequently to reduce $C \sqcup B$, we can use the preserved result of the projection of ρ on B, i.e., B' such that

$$\rho^{B}\colon B \;\; = \;\; B'$$

is the starting point for a new reduction of $C \sqcup B'$. We then know from Theorem 6.1 that the resulting reduction will be as short as possible (provided we are performing complete, call-by-need reductions).

Ideally, we would like to be able to designate a set of (perhaps non-disjoint) sub-elements of the initial term that may be later be replaced—in the parlance of

Section 8.1, a set of *substituends*. Every set of substituends of a term A partitions the term into two pieces: the join of the substituends, and the context containing them. We then require the ability to preserve the projection of a reduction on the context into which a substitution may later be made. This means that we must find a way to represent *simultaneously* the partitions induced by each possible set of substituends. If we can preserve the projection of any reduction on each such partition, we can perform *optimal* incremental computation (provided, as usual, that the reductions involved are complete and call-by-need).

8.3.1 $\triangle\mathbf{\Lambda}$CCL

In order to implement a mechanism for preserving projections and their results, we will extend **ΛCCL′** with two new constructors, and add several new axioms defining its behavior. The resulting system is called $\triangle$**ΛCCL**:

Definition 8.5 ($\triangle$ΛCCL Terms)
The terms of $\triangle$**ΛCCL**, *Ter($\triangle$ΛCCL), are those of* **ΛCCL**, *along with those built from the following new constructors:*

$$\begin{array}{rl} \triangle_{\mathcal{S}}(\mathcal{L},\, \mathcal{L}) : \mathcal{L} & \textit{(fork)} \\ \mathrm{Dup}_{\mathcal{S}}(\mathcal{L}) : \mathcal{L} & \textit{(duplicator)} \end{array}$$

where $\mathcal{S}$ is any denumerable set and $\mathcal{L}$ is the sort of lambda-like expressions.

The annotation s on any constructor of the form $\triangle_s(\cdot,\, \cdot)$ or $\mathrm{Dup}_s(\mathcal{L})$ will be called the constructor's *index*. In general, s will be an element of $\mathbb{P}^*$. Indices can be regarded as a means for providing an infinite set of constructors from a common family.

$\triangle(\cdot,\, \cdot)$ is the formalization of the informal "fork" node introduced in Section 8.1. Dup($\cdot$) will be used to *create* $\triangle(\cdot,\, \cdot)$ nodes, each of whose branches shares a subterm.

Definition 8.6 ($\triangle$ΛCCL Axioms)
The axioms of $\triangle$**ΛCCL** *are those of* **ΛCCL′**, *augmented with the following:*

$$\begin{array}{lr} (\triangle\mathbf{Apply}) & \mathrm{Apply}(\triangle_s(A,\, B),\, C) = \triangle_s(\mathrm{Apply}(A,\, C),\, \mathrm{Apply}(B,\, C)) \\ (\triangle\mathbf{C}) & [\triangle_s(A,\, B),\, E] = \triangle_s([A,\, E],\, [B,\, E]) \\ (\mathbf{Duplicate}) & \mathrm{Dup}_s(A) = \triangle_s(A,\, A) \end{array}$$

The rules of Definition 8.6 constitute a scheme describing an *infinite* set of rules, one for each operator index value. Since the rules are oblivious to the value of fork indices, we will usually omit the index value in any reference to an application of the rules.

($\triangle$**Apply**) and ($\triangle$C) are intended to formalize the informal distribution operations on $\triangle$s discussed in Section 8.1. Although one might think it necessary, it turns out that no distribution rule for the $\Lambda(\cdot)$ operator is required, since there is no $\mathbf{\Lambda CCL'}$ rewriting rule that operates directly on terms with $\Lambda(\cdot)$ as their root operator. The rule **(Duplicate)** will be used to create shared copies of certain terms, the result of the reduction of which is to be preserved. Such sharing was introduced informally in Section 8.1 without the use of a special operator. Note that since the terms and rules of $\mathbf{\Lambda CCL'}$ are a strict subset of those of $\triangle\mathbf{\Lambda CCL}$, $\triangle\mathbf{\Lambda CCL}$ is trivially a conservative extension of $\mathbf{\Lambda CCL'}$.

The set $\mathit{WHNF}_{\Lambda\mathrm{CCL}}$ is generalized to incorporate $\triangle$s as follows:

Definition 8.7 **(**$\mathit{WHNF}_{\triangle\Lambda\mathrm{CCL}}$**)**
The generalized set $\mathit{WHNF}_{\Lambda\mathrm{CCL}}$ *of elements of* $G\mathrm{Ter}_{\mathcal{L}}(\triangle\Lambda\mathrm{CCL})$ *is defined inductively as follows:*

$$
\begin{array}{rcl}
n! \in \mathit{WHNF}_{\triangle\Lambda\mathrm{CCL}} & & \\
A \in \mathit{AF} & \Longrightarrow & A \in \mathit{WHNF}_{\triangle\Lambda\mathrm{CCL}} \\
A \in \mathit{WHNF}_{\triangle\Lambda\mathrm{CCL}}, A \notin \mathit{AF} & \Longrightarrow & \mathrm{Apply}(A,\ B) \in \mathit{WHNF}_{\triangle\Lambda\mathrm{CCL}} \\
A \in \mathit{WHNF}_{\triangle\Lambda\mathrm{CCL}} & \Longrightarrow & \triangle_{s}(A,\ B) \in \mathit{WHNF}_{\triangle\Lambda\mathrm{CCL}}
\end{array}
$$

8.3.2 Indexed Interpretations

Recall that the purpose of the $\triangle$ node is to represent by a single term *sets* of compatible $\mathbf{\Lambda CCL}$ terms. By specifying a path to be taken through the $\triangle$ nodes in the term (i.e., for each $\triangle$, whether to inspect the right or the left subterm), we determine a unique $\mathbf{\Lambda CCL}$ term. In the sequel, we will ensure that the set of all $\mathbf{\Lambda CCL}$ terms simultaneously represented by a single $\triangle\mathbf{\Lambda CCL}$ term will be $\mathbf{\Lambda CCL}$-equivalent.

To make this precise, we define the notion of an *indexed interpretation* of a $\triangle\mathbf{\Lambda CCL}$ term as follows:

Definition 8.8 (Indexed Interpretation)
Let A be an element of $\mathrm{Ter}(\triangle\Lambda\mathrm{CCL})$ *and $\mathcal{S}$ be a subset of $P(\mathbb{N})$. Then an $\mathcal{S}$-*indexed interpretation *of A, $\mathit{interp}_{\mathcal{S}}(A)$ is any Brouwerian algebra monomorphism on* $\mathrm{Ter}(\triangle\Lambda\mathrm{CCL})$ *satisfying the following:*

$$
\begin{array}{rcll}
\mathit{interp}_{\mathcal{S}}(\triangle_{s}(A,\ B)) & \triangleq & \mathit{interp}_{\mathcal{S}}(B) & s \in \mathcal{S} \\
\mathit{interp}_{\mathcal{S}}(\triangle_{s}(A,\ B)) & \triangleq & \mathit{interp}_{\mathcal{S}}(A) & s \notin \mathcal{S} \\
\mathit{interp}_{\mathcal{S}}(\mathbf{Dup}_{s}(A)) & \triangleq & \mathit{interp}_{\mathcal{S}}(A) & \\
\mathit{interp}_{\mathcal{S}}(\mathrm{Apply}(A,\ B)) & \triangleq & \mathrm{Apply}(\mathit{interp}_{\mathcal{S}}(A),\ \mathit{interp}_{\mathcal{S}}(B)) & \\
\mathit{interp}_{\mathcal{S}}(\Lambda(A)) & \triangleq & \Lambda(\mathit{interp}_{\mathcal{S}}(A)) & \\
\mathit{interp}_{\mathcal{S}}(\mathrm{Var}) & \triangleq & \mathrm{Var} & \\
 & \vdots & &
\end{array}
$$

(similarly for other $\triangle\mathbf{\Lambda CCL}$ *operators)*

The set of all $\triangle$ indices in a term A is given by

$$\mathit{ForkIndices}(A)$$

We can extend the notion of indexed interpretation to $\triangle\mathbf{\Lambda CCL}$-term graphs as follows:

Definition 8.9 (Generalized Indexed Interpretation)

Let G be an element of $\mathit{Graph1}(\triangle\Lambda\mathrm{CCL})$ and $\mathcal{S}$ be a subset of $\boldsymbol{P}(\mathbb{N})$. Then a generalized $\mathcal{S}$-indexed interpretation of G, $\mathit{interp}_{\mathcal{S}}(G)$ is any Brouwerian algebra monomorphism on $\mathit{Graph1}(\triangle\Lambda\mathrm{CCL})$ satisfying

$$U(\mathit{interp}_{\mathcal{S}}(G)) \;=\; \mathit{interp}_{\mathcal{S}}(U(G))$$

(where U is the unraveling mapping).

Note that since $\mathit{interp}_{\mathcal{S}}(\cdot)$ is a monomorphism, it must preserve any sharing present in G, as well as disjointness.

An indexed interpretation of a $\triangle\mathbf{\Lambda CCL}$ term A by some set of indices $\mathcal{S}$ serves not only to extract a $\mathbf{\Lambda CCL}$ term, it also has the effect of *partitioning* the term into two context forests, which we designate $\mathit{left}_{\mathcal{S}}(A)$ and $\mathit{right}_{\mathcal{S}}(A)$, respectively. Their definitions are as follows:

Definition 8.10 (Partitioned Interpretation)

Let A be an element of $\mathrm{Ter}(\triangle\Lambda\mathrm{CCL})$ *and $\mathcal{S}$ be a subset of $P(\mathbb{N})$. Then we define $\mathcal{S}$-*indexed left and right partitions *of A, notated $\mathit{left}_{\mathcal{S}}(A)$ and $\mathit{right}_{\mathcal{S}}(A)$, respectively, as any pair of Brouwerian algebra monomorphisms that satisfy the following:*

(1)

$$\begin{array}{rcll}
\mathit{right}_{\mathcal{S}}(\overset{s}{\triangle}(A,\, B)) & \triangleq & \mathit{right}_{\mathcal{S}}(B) & s \in \mathcal{S} \\
\mathit{right}_{\mathcal{S}}(\overset{s}{\triangle}(A,\, B)) & \triangleq & \bot & s \notin \mathcal{S} \\
\mathit{right}_{\mathcal{S}}(\mathrm{Dup}_s(A)) & \triangleq & \mathit{right}_{\mathcal{S}}(A) & \\
\mathit{right}_{\mathcal{S}}(\mathrm{Apply}(A,\, B)) & \triangleq & \mathrm{Apply}(\mathit{right}_{\mathcal{S}}(A),\, \mathit{right}_{\mathcal{S}}(B)) & \\
\mathit{right}_{\mathcal{S}}(\Lambda(A)) & \triangleq & \Lambda(\mathit{right}_{\mathcal{S}}(A)) & \\
\mathit{right}_{\mathcal{S}}(\mathrm{Var}) & \triangleq & \mathrm{Var} & \\
 & \vdots & &
\end{array}$$

(similarly for other $\triangle\mathbf{\Lambda CCL}$ operators)

(2)

$$
\begin{array}{rcll}
\mathit{left}_{\mathcal{S}}(\triangle_s(A,\ B)) & \triangleq & \bot & s \in \mathcal{S} \\
\mathit{left}_{\mathcal{S}}(\triangle_s(A,\ B)) & \triangleq & \mathit{left}_{\mathcal{S}}(A) & s \notin \mathcal{S} \\
\mathit{left}_{\mathcal{S}}(\mathrm{Dup}_s(A)) & \triangleq & \mathit{left}_{\mathcal{S}}(A) & \\
\mathit{left}_{\mathcal{S}}(\mathrm{Apply}(A,\ B)) & \triangleq & \mathrm{Apply}(\mathit{left}_{\mathcal{S}}(A),\ \mathit{left}_{\mathcal{S}}(B)) & \\
\mathit{left}_{\mathcal{S}}(\Lambda(A)) & \triangleq & \Lambda(\mathit{left}_{\mathcal{S}}(A)) & \\
\mathit{left}_{\mathcal{S}}(\mathrm{Var}) & \triangleq & \mathrm{Var} & \\
 & \vdots & & \\
\multicolumn{4}{c}{\textit{(similarly for other } \triangle\mathbf{\Lambda CCL} \textit{ operators)}}
\end{array}
$$

(3)

$$\mathit{left}_{\mathcal{S}}(A) \parallel \mathit{right}_{\mathcal{S}}(A)$$

(4)

$$\mathit{left}_{\mathcal{S}}(A) \sqcup \mathit{right}_{\mathcal{S}}(A) = \mathit{interp}_{\mathcal{S}}(A)$$

$\mathit{left}_{\mathcal{S}}(\cdot)$ and $\mathit{right}_{\mathcal{S}}(\cdot)$ merely divide up the term produced by $\mathit{interp}_{\mathcal{S}}(\cdot)$ into the part derived by traversing left children of $\triangle$s, and the part derived by traversing right children.

The extension of $\mathit{left}_{\mathcal{S}}(\cdot)$ and $\mathit{right}_{\mathcal{S}}(\cdot)$ to term graphs is analogous to that of Definition 8.9.

In the sequel we will require the following *left removal operation*:

Definition 8.11 (Left Removal)
Let A be an element of Ter($\triangle\Lambda$CCL) *and $\mathcal{S}$ be a subset of $P(\mathbb{N})$. Then we define* $\mathcal{S}$-left removal operation *on A, notated* $\mathrm{remove}L_{\mathcal{S}}(A)$ *as any Brouwerian algebra monomorphism that satisfies the following:*

$$
\begin{array}{rcll}
\mathrm{remove}L_{\mathcal{S}}(\triangle_s(A,\ B)) & \triangleq & \mathrm{remove}L_{\mathcal{S}}(B) & s \in \mathcal{S} \\
\mathrm{remove}L_{\mathcal{S}}(\triangle_s(A,\ B)) & \triangleq & \triangle_s(\mathrm{remove}L_{\mathcal{S}}(A),\ \mathrm{remove}L_{\mathcal{S}}(B)) & s \notin \mathcal{S} \\
\mathrm{remove}L_{\mathcal{S}}(\mathrm{Dup}_s(A)) & \triangleq & \mathrm{remove}L_{\mathcal{S}}(A) & \\
\mathrm{remove}L_{\mathcal{S}}(\mathrm{Apply}(A,\ B)) & \triangleq & \mathrm{Apply}(\mathrm{remove}L_{\mathcal{S}}(A),\ \mathrm{remove}L_{\mathcal{S}}(B)) & \\
\mathrm{remove}L_{\mathcal{S}}(\Lambda(A)) & \triangleq & \Lambda(\mathrm{remove}L_{\mathcal{S}}(A)) & \\
\mathrm{remove}L_{\mathcal{S}}(\mathrm{Var}) & \triangleq & \mathrm{Var} & \\
 & \vdots & & \\
\multicolumn{4}{c}{\textit{(similarly for other } \triangle\mathbf{\Lambda CCL} \textit{ operators)}}
\end{array}
$$

The left removal operation replaces every node of the form

$$\triangle_s(\cdot,\ \cdot)$$

such that $s \in \mathcal{S}$ with the node's right child. It also removes any $\mathrm{Dup}_s(\cdot)$ node such that $s \in \mathcal{S}$. All other nodes are essentially left alone.

$removeL_{\mathcal{S}}(\cdot)$ can be extended in the usual way to term graphs.

We note that all of the interpretation operations defined above are homomorphic with respect to the operators of **ΛCCL**: they preserve the operator structure of all terms not of the form $\triangle(\cdot, \cdot)$ or $\mathrm{Dup}(\cdot)$.

8.3.3 Properties of △ΛCCL

Ignoring for the time being the whys and wherefores of indexed △s, we enumerate the salient properties of **△ΛCCL**, all of which follow immediately from the analogous results for **ΛCCL′**:

The new rules added to yield **△ΛCCL** are not projection rules, nor do not introduce critical pairs to **ΛCCL′**. Thus we have:

Lemma 8.11 **△ΛCCL** *is a strongly regular conditional term rewriting system.*

Proof Analogous to that of Lemma 8.3 □

As usual, we get:

Corollary 8.12 **△ΛCCL** *is confluent.*

Lemma 8.13 *The leftmost outermost* **△ΛCCL** *redex of any term of*

$$GTer_{\mathcal{L}}(\triangle\Lambda\mathrm{CCL})$$

is $WHNF_{\triangle\Lambda\mathrm{CCL}}$*-needed. Furthermore, if A is a term of the form*

$$A \equiv [\mathrm{Var},\ \langle E,\ B\rangle]$$

then the leftmost outermost **ΛCCL′** *redex of B is* $WHNF_{\triangle\Lambda\mathrm{CCL}}$*-needed.*

Proof Similar to that of Lemma 8.7. □

Corollary 8.14 *Any leftmost outermost* **△ΛCCL** *reduction strategy for* **△ΛCCL** *terms of* $GTer_{\mathcal{L}}(\triangle\Lambda\mathrm{CCL})$ *is* $WHNF_{\triangle\Lambda\mathrm{CCL}}$*-call-by-need.*

Since **△ΛCCL** is strongly regular, its term graph rewriting version can be used to yield an optimal implementation of **△ΛCCL**. We thus make the following definition:

Definition 8.12 ($\triangle\mathbf{\Lambda CCL}^G$)
$\triangle\mathbf{\Lambda CCL}^G$ *will denote the regular graph rewriting system given by Theorem 4.22 that is a complete, strong, family-preserving implementation of* **△ΛCCL**.

As with $\mathbf{\Lambda CCL}'$, to get an optimal implementation, we need a reduction strategy. For this, we will use $\triangle$*rwhnf2*$(\cdot)$, depicted in Figures 8.11 and 8.12. $\triangle$*rwhnf2*$(\cdot)$ is based on *rwhnf2*$(\cdot)$; the only differences between the two algorithms are found in those lines annotated with asterisks. (which are exactly those concerned with the $\triangle(\cdot,\cdot)$ or $\mathrm{Dup}(\cdot)$ operators). As with *rwhnf2*$(\cdot)$, $\triangle$*rwhnf2*$(\cdot)$ always contracts only leftmost outermost redexes. Therefore, we have:

Theorem 8.15 *The implementation/strategy pair*

$$\langle \triangle\mathbf{\Lambda CCL}^G,\ \triangle \mathit{rwhnf2}(\cdot)\rangle$$

is a $\mathit{WHNF}_{\triangle\Lambda\mathrm{CCL}}$*-optimal implementation of* $\triangle\mathbf{\Lambda CCL}$ *for terms of*

$$\mathit{GTer}_{\mathcal{L}}(\triangle\mathbf{\Lambda CCL})$$

Proof As before, by definition, $\triangle\mathbf{\Lambda CCL}^G$ is a family-preserving implementation of $\triangle\mathbf{\Lambda CCL}$. $\triangle$*rwhnf2*$(\cdot)$ is $\mathit{WHNF}_{\triangle\Lambda\mathrm{CCL}}$-call-by-need. Therefore, by Theorem 6.4,

$$\langle \triangle\mathbf{\Lambda CCL}^G,\ \triangle \mathit{rwhnf2}(\cdot)\rangle$$

is $\mathit{WHNF}_{\triangle\Lambda\mathrm{CCL}}$-optimal. □

8.3.4 $\triangle\mathbf{\Lambda CCL}$ and $\mathbf{\Lambda CCL}'$

Our primary interest in $\triangle\mathbf{\Lambda CCL}$ comes not from its intrinsic properties, but from its ability to simulate $\mathbf{\Lambda CCL}'$ reductions on multiple terms simultaneously.

We first note that $\triangle\mathbf{\Lambda CCL}$ can be used to implement $\mathbf{\Lambda CCL}'$, as follows:

Theorem 8.16 *Let $\mathcal{S}$ be a set of indices, and let*

$$g\colon \mathit{GTer}(\Lambda\mathrm{CCL}) \quad\rightarrow\quad \mathit{GTer}(\triangle\Lambda\mathrm{CCL})$$

such that for all $A \in \mathit{GTer}(\Lambda\mathrm{CCL})$,

$$\mathit{interp}_{\mathcal{S}}(g(A)) \quad=\quad A$$

Then

$$\langle \triangle\mathbf{\Lambda CCL},\ g,\ \mathit{interp}_{\mathcal{S}}(\cdot)\rangle$$

is a complete implementation of $\mathbf{\Lambda CCL}'$.

Proof We must show that if for some $A \in \mathit{GTer}(\Lambda\mathrm{CCL})$

$$g(A) \quad\twoheadrightarrow_{\triangle\Lambda\mathrm{CCL}}\quad B'$$

then

$$A \quad\twoheadrightarrow_{\Lambda\mathrm{CCL}'}\quad \mathit{interp}_{\mathcal{S}}(B')$$

If a redex is contracted in any intermediate element B_i of a reduction of $g(A)$, it must be either a $\triangle\mathbf{\Lambda CCL}$ redex or a **($\triangle$Apply)**, **($\triangle$C)**, or **(Duplicate)** redex. If it is a $\mathbf{\Lambda CCL'}$ redex, we note that all the nodes that comprise any $\triangle\mathbf{\Lambda CCL}$ redex are either mapped by the interpretation to a a redex in the image of the interpretation, or are mapped to bottom. Therefore, for every contraction of a $\mathbf{\Lambda CCL'}$ redex, there is either a corresponding contraction in its interpretation, or the interpretation is left unaltered by the contraction. If one of rules **($\triangle$Apply)**, **($\triangle$C)**, or **(Duplicate)** is used, the result of the interpretation is unchanged. In either event, there is some reduction from $interp_{\mathcal{S}}(B_1)$ to $interp_{\mathcal{S}}(B_2)$.

To show that the implementation is complete, it must be the case that for all A, A' such that

$$A \twoheadrightarrow_{\mathbf{\Lambda CCL'}} A'$$

we have

$$g(A) \twoheadrightarrow_{\triangle\mathbf{\Lambda CCL}} g(A')$$

The only problem with simulating arbitrary $\mathbf{\Lambda CCL'}$ reductions in $\triangle\mathbf{\Lambda CCL}$ comes when $\triangle(\cdot, \cdot)$ or $\mathrm{Dup}(\cdot)$ nodes intervene between components of a term mapped to a redex by $interp_{\mathcal{S}}(\cdot)$. But then after a finite number of applications of rules **($\triangle$Apply)** or **($\triangle$C)**, the subterm whose image is a $\mathbf{\Lambda CCL'}$ redex must become a $\mathbf{\Lambda CCL'}$ redex as well. □

We then have the following:

Corollary 8.17 *For any leftmost outermost $\triangle\mathbf{\Lambda CCL}$ reduction ρ of the form*

$$\hat{A} \xrightarrow{\rho}\!\!\!\!\rightarrow_{\triangle\Lambda\mathrm{CCL}} \hat{A}'$$

and index set $\mathcal{S}$, there exists a leftmost outermost $\mathbf{\Lambda CCL'}$ reduction σ such that

$$interp_{\mathcal{S}}(\hat{A}) \xrightarrow{\sigma}\!\!\!\!\rightarrow_{\Lambda\mathrm{CCL'}} interp_{\mathcal{S}}(\hat{A}')$$

Proof Since the mapping $interp_{\mathcal{S}}(\cdot)$ is a homomorphism, it maps leftmost outermost sub-elements to leftmost outermost sub-elements. We then observe that application of a leftmost outermost **($\triangle$Apply)**, **($\triangle$C)**, or **(Duplicate)** redex contraction preserves the leftmost or outermost state of any $\mathbf{\Lambda CCL'}$redexes created by the mapping $interp_{\mathcal{S}}(\cdot)$. □

When we consider the term graph rewriting systems $\mathbf{\Lambda CCL}^{G'}$ and $\triangle\mathbf{\Lambda CCL}^{G}$, we get a stronger implementation result:

Theorem 8.18 *Let S be a set indices, and let*

$$g\colon \mathit{Graph1}(\Lambda\mathrm{CCL}^{G'}) \;\; \rightarrow \;\; \mathit{Graph1}(\triangle\Lambda\mathrm{CCL})$$

such that for all $A \in G\mathrm{Ter}(\Lambda\mathrm{CCL})$,

$$\mathit{interp}_{\mathcal{S}}(g(A)) \;\; = \;\; A$$

Then

$$\langle \triangle\mathbf{\Lambda CCL},\, g,\, \mathit{interp}_{\mathcal{S}}(\cdot)\rangle$$

is a complete, family-preserving implementation of $\mathbf{\Lambda}\mathrm{CCL}^{G'}$.

Proof Since $\triangle\mathbf{\Lambda CCL}^G$ is disjoint, each of its families has only one redex. We know from Theorems 8.16 and 4.22 that the implementation must be sound and complete. Since $\mathit{interp}_{\mathcal{S}}(\cdot)$ is a Brouwerian algebra monomorphism, no $\triangle\mathbf{\Lambda CCL}^G$ redex is mapped to more than one $\mathbf{\Lambda}\mathrm{CCL}'$ redex. Let G be a $\triangle\mathbf{\Lambda CCL}$ graph, and let R be a redex of G. We then note that every redex *created* by the contraction of R using a $\mathbf{\Lambda CCL}^{G'}$ rule maps to a single redex under the interpretation. Thus contraction of any redex in G corresponds to contraction of 0 or 1 redexes in $\mathit{interp}_{\mathcal{S}}(G)$, and each created redex maps to a single redex. Therefore, the mapping preserves families. □

Given Theorem 8.18 and Theorem 8.16, the following corollary is immediate:

Corollary 8.19 *Let S be a set of indices, and let*

$$g\colon G\mathrm{Ter}(\Lambda\mathrm{CCL}) \;\; \rightarrow \;\; \mathit{Graph1}(\triangle\Lambda\mathrm{CCL})$$

such that

$$U(\mathit{interp}_{\mathcal{S}}(g(A))) \;\; = \;\; A$$

(where $U(\cdot)$ is the unraveling mapping). Then

$$\langle \triangle\mathbf{\Lambda CCL},\, g,\, U(\mathit{interp}_{\mathcal{S}}(\cdot))\rangle$$

is a complete, family-preserving implementation of $\mathbf{\Lambda}\mathrm{CCL}'$.

Theorem 8.18 will be very important in the sequel. It says that $\triangle\mathbf{\Lambda CCL}^G$ implements *complete* $\mathbf{\Lambda}\mathrm{CCL}'$ reductions for any interpretation of the graph being reduced.

$\triangle rwhnf2(T) \equiv$
case T **of**
    $\Lambda(A)$: **skip**; $\{T \in WHNF_{\triangle\Lambda\mathrm{CCL}}\}$
    Var: **skip**; $\{T \in WHNF_{\triangle\Lambda\mathrm{CCL}}\}$
*   Dup(A): **seq**
* $T := \triangle(A,\ A)$; $\{$*rule* **(Duplicate)**$\}$
* $\triangle rwhnf2(A)$
*   **endseq**;
* $\triangle(T_1',\ T_2')$: $\triangle rwhnf2(T_1')$;
    Apply$(A,\ B)$: **seq**
        $\triangle rwhnf2(A)$;
        case A **of**
* $\triangle(A_1',\ A_2')$: **seq**
* $T := \triangle(\mathrm{Apply}(A_1',\ B){:}\hat{T},\ \mathrm{Apply}(A_2',\ B))$;
* $\{$*rule* **($\triangle$Apply)**$\}$
* $\triangle rwhnf2(\hat{T})$
*   **endseq**
            $\Lambda(A')$: **seq**
                $T := [A',\ \langle\emptyset,\ B\rangle]$; $\{$*rule* **(Beta)**$\}$
                $\triangle rwhnf2(T)$
                endseq;
            otherwise: **skip** $\{T \in WHNF_{\triangle\Lambda\mathrm{CCL}}\}$
        endseq;

⋮
{See Part II: Figure 8.12}
⋮

Figure 8.11: Algorithm $\triangle rwhnf2(\cdot)$, Part I

⋮
{See Part I: Figure 8.11}
⋮

$[L,\ E]$: **seq**
  $\triangle rwhnf2(L)$;
  case L **of**
* $\triangle(L_1',\ L_2')$: **seq**
* $T := \triangle([L_1',\ E]\colon\hat{T},\ [L_2',\ E])$; {*rule* $(\triangle\mathbf{C})$}
* $\triangle rwhnf2(\hat{T})$
* **endseq**;
    $\mathrm{Apply}(A,\ B)$: **seq**
      $T := \mathrm{Apply}([A,\ E],\ [B,\ E])$; {*rule* $(\mathbf{DApply}')$}
      $\triangle rwhnf2(T)$
      endseq;
    $\Lambda(A)$: $T := \Lambda([A,\ \langle E \circ \Box,\ \mathrm{Var}\rangle])$;
      {*rule* $(\mathbf{D}\Lambda)$; $T \in WHNF_{\triangle\Lambda\mathrm{CCL}}$}
    $[L_1,\ E_1]$: **seq**
      $T := [L_1,\ E_1 \circ E]$; {*rule* $(\mathbf{AssC}')$}
      $\triangle rwhnf2(T)$
      endseq;
    Var: **seq**
      $rwpenf(E)$;
      case E **of**
        $\emptyset$: $T := \mathrm{Var}$; {*rule* $(\mathbf{NullC}')$}
        $\Box^n$: **skip**; {$T \equiv n!$; $T \in WHNF_{\triangle\Lambda\mathrm{CCL}}$}
        $\langle E,\ A\rangle$: **seq**
          $\triangle rwhnf2(A)$;
          $T := copyTop(A)$ {*rules* $(\mathbf{VarRef}_i')$, $i \in 1 \ldots 4$}
          endseq
      endseq
  endseq

Figure 8.12: Algorithm $\triangle rwhnf2(\cdot)$, Part II

8.4 Incremental Reduction Using $\triangle\mathbf{\Lambda CCL}$

We have now reached the point where we are able to give a formal account of the incremental reduction ideas outlined in Section 8.1. We outline the approach we will take as follows:

- First, we must ensure that we perform complete reductions. Thus our starting point is the strongly regular system $\mathbf{\Lambda CCL}'$. As shown by Theorem 8.9, the term graph rewriting counterpart of $\mathbf{\Lambda CCL}'$, $\mathbf{\Lambda CCL}^{G'}$, along with the leftmost outermost reduction strategy of *rwhnf2*$(\cdot)$ yields an optimal weak head normalizing reduction system.

- Next, we assume that each term to be reduced is decomposed into nested sets of *substituends*—those subterms that may be later replaced. Any set of substituends determines a *partition* of the term into disjoint pieces. This decomposition will be implemented by inserting Dup$(\cdot)$ nodes at each "join point" of elements in the partition. Any subset of the set of substituends in the term, provided its elements are disjoint, may later be replaced by a new set of subterms. These new subterms in turn also may take the form of nested sets of substituends.

- Having inserted Dup$(\cdot)$ nodes in appropriate places to demarcate the boundaries between elements of the partition induced by the substituend structure, we proceed to reduce the term using the rules of $\triangle\mathbf{\Lambda CCL}$ and algorithm $\triangle$*rwhnf2*$(\cdot)$. Whenever a contraction needs to be performed that involves two different substituends, reduction of a node of the form Dup(A) that demarcates the boundary has the effect of creating a node of the form $\triangle(A, A)$. When we perform the reduction using graph rewriting, the effect is to cause two references to a shared subgraph A to be created.

- As the initial term is reduced, we must preserve the "boundaries" between elements of the initial partition in such a way as to capture in each subpartition the *projection* of the reduction on the initial term. The $\triangle$s have the effect of preserving boundaries between partition elements on one branch, and allowing reductions that cross the partition boundary on the other branch. Those elements that are present in two subpartitions of the initial partition are guaranteed to be *shared*, since $\triangle\mathbf{\Lambda CCL}^G$ is a disjoint RRS.

- When $\triangle\mathbf{\Lambda CCL}$ reduction terminates with an element in $\mathit{WHNF}_{\triangle\Lambda\mathrm{CCL}}$, we must be able to interpret the resulting graph to yield a term graph (in $\mathit{WHNF}_{\Lambda\mathrm{CCL}}$) that represents a term of $\mathbf{\Lambda CCL}'$. For this, we will use $\mathit{interp}_{\mathcal{S}}(\cdot)$.

- Finally, in order to reuse the results of a projection of a reduction, we must be able to *replace* substituends in some element of the initial partition with new terms, taking advantage of the result of the projection of the old reduction.

8.4.1 The Problem

In order to define the problem of incremental reduction formally, we first require the following auxiliary definitions:

Definition 8.13 (Substituend Decomposition)
Let Σ be a signature. Let B be an element of Ter(Σ) *and s be an element of $\mathbb{P}^*$. Then a set of pairs $\mathcal{B}^s \in \mathbb{P}^* \times \mathrm{Ter}(\Sigma)$ is an s-rooted* substituend decomposition *of B iff the following hold:*

$$\begin{array}{rcl} \langle s,\ B\rangle \in \mathcal{B}^s & & \\ \langle t,\ B^t\rangle \in \mathcal{B}^s & \Longrightarrow & t = s \circ u \quad \textit{for some } u \in \mathbb{P}^* \\ \langle t \circ i,\ B^{t\circ i}\rangle \in \mathcal{B}^s \text{ and } i > 1 & \Longrightarrow & \langle t \circ (i-1),\ B^{t\circ(i-1)}\rangle \in \mathcal{B}^s \end{array}$$

and for all $\langle t,\ B^t\rangle \in \mathcal{B}^s$ such that there exists $\langle t \circ i,\ B^{t\circ i}\rangle \in \mathcal{B}^s$,

$$B^t \;=\; C^t[B^{t\circ 1}, B^{t\circ 2}, \ldots, B^{t\circ n_t}]$$

for some context C^t, where

$$n_t = \max_i \{i \mid \langle t \circ i,\ B^{t\circ i}\rangle \in \mathcal{B}^s\}$$

If $\langle t,\ B^t\rangle \in \mathcal{B}^s$, then we will refer to t as the index *of B^t, and B^t as a substituend (of $\mathcal{B}^s$). If the root of the decomposition is ϵ, we will use $\mathcal{B}$ to refer to the decomposition $\mathcal{B}^\epsilon$. We define Ind($\mathcal{B}^s$) to be the set of all substituend indices of $\mathcal{B}^s$, i.e.,*

$$\mathit{Ind}(\mathcal{B}^s) \triangleq \{t \mid \langle t,\ B^t\rangle \in \mathcal{B}^s\}$$

An s-rooted substituend decomposition of B divides B into a set of nested subtrees, with B indexed by s. For the sake of brevity, we will usually use B^t to refer to an element $\langle t,\ B^t\rangle$ of some substituend decomposition $\mathcal{B}^s$. Note that any substituend decomposition $\mathcal{B}^s$ uniquely determines the element that it decomposes, i.e., B such that $\langle s,\ B\rangle \in \mathcal{B}^s$. Also note that by Lemma 4.3, any substituend decomposition $\mathcal{B}^s$ of B determines a *partition* of B into disjoint nested context forests.

We begin by defining for any set of substituend indices $\mathcal{S}$ and decomposition $\mathcal{A}$, the *substituend element designated by $\mathcal{S}$* and the *context element designated by $\mathcal{S}$* as follows:

Definition 8.14 (Designated Context, Substituend)
*Let A be an element of G*Ter(ΛCCL)*, $\mathcal{A}$ be a substituend decomposition of A, and*

let $\mathcal{S} \subseteq Ind(\mathcal{A})$. *Then the* substituend element designated by $\mathcal{S}$ *and the* context element designated by $\mathcal{S}$, *notated* $substs_{\mathcal{S}}(\mathcal{A})$ *and* $conts_{\mathcal{S}}(\mathcal{A})$, *respectively, are defined by:*

$$\begin{aligned} substs_{\mathcal{S}}(\mathcal{A}) &\triangleq \bigsqcup\{A^s \mid A^s \in \mathcal{A},\ s \in \mathcal{S}\} \\ conts_{\mathcal{S}}(\mathcal{A}) &\triangleq A \dot{-} substs_{\mathcal{S}}(\mathcal{A}) \end{aligned}$$

The idea of Definition 8.14 is to partition the term A into two pieces: the join of all the substituends designated by $\mathcal{A}$, and the "rest" of A, that is, the context containing the join of the substituends. It is immediate from the definitions that for all A, $\mathcal{A}$, and $\mathcal{S}$,

$$A = conts_{\mathcal{S}}(\mathcal{A}) \sqcup substs_{\mathcal{S}}(\mathcal{A})$$

Note also that $cont_{\emptyset}(\mathcal{A}) = A$.

We can now state formally the problem of incremental reduction in **ΛCCL**: We assume we wish to reduce a sequence of **ΛCCL** terms to $WHNF_{\Lambda\text{CCL}}$. Let these terms be

$$A_0, A_1, A_2, \ldots$$

Let the elements resulting from each reduction be denoted respectively by

$$A'_0, A'_1, A'_2, \ldots$$

where for all i, $A'_i \in WHNF_{\Lambda\text{CCL}}$.

We also assume that associated with each term A_i that is to be reduced is a substituend decomposition $\mathcal{A}_i$. *Each substituend* $A_i^s \in \mathcal{A}_i$ *should be viewed as a pre-designated subterm of* A_i *that may later be replaced by some new substituend decomposition.* If A_i^s is a substituend of $\mathcal{A}_i$, we will also say that it is a substituend of A_i.

Each element A_i is then derived from its predecessor A_{i-1} by replacement of some disjoint set of its substituends by elements of the form $B_i^{s_j}$, each of which also has an associated substituend decomposition $\mathcal{B}_i^{s_j}$ (so that $B_i^{s_j}$'s elements may later be replaced). A_i is thus defined by:

$$A_i \equiv A_{i-1}[A_{i-1}^{s_1} \leftarrow B_i^{s_1}][A_{i-1}^{s_2} \leftarrow B_i^{s_2}] \cdots [A_{i-1}^{s_m} \leftarrow B_i^{s_m}]$$

where for all $j \in 1 \ldots m$, $A_{i-1}^{s_j} \in \mathcal{A}_{i-1}$, and for all $j \neq k$, $A_{i-1}^{s_j} \parallel A_{i-1}^{s_k}$.

After the rerooting operation performed by the replacement operation, the images of the terms $B_i^{s_j}$ become substituends $A_i^{s_j}$ for the next phase of the reduction process. The set of substituends $\mathcal{B}_i^{s_j}$ associated with each $B_i^{s_j}$ is mapped by the rerooting operation to $\mathcal{A}_i^{s_j}$.

The substituend decomposition of $\mathcal{A}_i$ associated with A_i is then given by

$$\mathcal{A}_i \equiv \mathcal{A}_{i-1} - \{A_{i-1}^s \mid s \in \mathcal{S}_i\} \cup (\bigcup_{s \in \mathcal{S}_i} \mathcal{A}_i^s)$$

We will refer to the set of indices of substituends replaced at step i as $\mathcal{S}_i$. The *substitution set* for step i, i.e., the set of all decompositions $\mathcal{B}_i^s$ such that $s \in \mathcal{S}_i$, is denoted by Δ_i.

From an algebraic point of view, we note that

$$A_i \equiv (A_{i-1} \dot{-} D_i^-) \sqcup D_i^+$$

where the *difference elements* D_i^- and D_i^+ at each step are defined by:

$$\begin{aligned} D_i^- &\triangleq cont_{\mathcal{S}}(\mathcal{A}_{i-1}) \\ D_i^+ &\triangleq \bigsqcup_{s \in \mathcal{S}_i} A_i^s \end{aligned}$$

Note that the problem of incremental reduction can be phrased solely in terms of some initial term A_0 and substituend set $\mathcal{A}_0$, along with and a sequence of pairs

$$\langle \Delta_1, \mathcal{S}_1 \rangle, \langle \Delta_2, \mathcal{S}_2 \rangle, \ldots$$

where the Δ_i and $\mathcal{S}_i$ are the substitution set and corresponding index set, respectively, for step i. Note that this formulation of the problem fits in nicely with the framework for incremental computation of Section 1.2.

The description of the the problem above may initially seem frightfully complicated, but the essential points are quite straightforward. Each element A_i that is to be reduced must be formed by substituting some disjoint subset of $\mathcal{A}_{i-1}$ in A_{i-1} for the elements of Δ_i.

Our goal will be to preserve the *projection* of the earlier reduction of A_{i-1} on

$$A_{i-1} \dot{-} D_i^-$$

so that the subsequent reduction of

$$A_i \equiv (A_{i-1} \dot{-} D_i^-) \sqcup D_i^+$$

will, by Theorem 7.3, be incrementally optimal.

8.4.2 Λ^{inc}

In order to define Λ^{inc}, we first require the following auxiliary definition:

Definition 8.15 (Explicit Partitioning, Root Map)
Let A be an element of GTer(ΛCCL), $\mathcal{A}$ be a substituend decomposition of A, G be an element of Graph1($\triangle\Lambda$CCL), and let ν be an element of G. Then the explicit partitioning function *on A, $prt(\mathcal{A}, \nu, G)$ constructs a tree graph element of Graph1($\triangle\Lambda$CCL) such that the following hold:*

(1)

$$\begin{aligned} prt(A^s, \nu, G) \quad = \quad C^s[\; & \mathrm{Dup}_{so1}(prt(\mathcal{A}^{so1}, \nu, G)), \\ & \mathrm{Dup}_{so2}(prt(\mathcal{A}^{so2}, \nu, G)), \\ & \vdots \\ & \mathrm{Dup}_{son_s}(prt(\mathcal{A}^{son_s}, \nu, G)) \quad] \end{aligned}$$

where

$$A^s \;=\; C^s[A^{so1}, A^{so2}, \ldots, A^{son_s}]$$

and A^s, A^{soi} *are elements of* $\mathcal{A}$.

(2)

$$id(root(prt(\mathcal{A}, \nu, G))) = id(root(\nu))$$

(3)

$$prt(\mathcal{A}, \nu, G) \parallel (G \dot{-} \nu)$$

For any element $\hat{A}$ *such that*

$$\hat{A} \;=\; prt(\mathcal{A}, \nu, G)$$

we define its associated root map, *notated* $rootMap[\hat{A}](\cdot)$, *to be a function mapping substituend indices to graph nodes such that*

$$rootMap[\hat{A}](s) = \nu$$

implies that $\nu = root(\hat{A}^s)$, *where*

$$\mathrm{Dup}_s(\hat{A}^s) \sqsubseteq prt(\mathcal{A}, \nu, G)$$

If $\hat{A}$ *and* $\hat{B}$ *are disjoint graphs such that*

$$\begin{aligned} \hat{A} &= prt(\mathcal{A}, \nu, G) \\ \hat{B} &= prt(\mathcal{B}, \nu, H) \end{aligned}$$

then we can speak of their combined root map

$$rootMap[\hat{A}](\cdot) \cup rootMap[\hat{B}](\cdot)$$

without confusion.

The explicit partitioning operation makes the "joins" between substituends and the context containing them explicit via insertion of appropriately indexed $\mathrm{Dup}_s(\cdot)$ nodes at the join points. It also guarantees that the graph thus constructed will have a root identifier that matches ν, and that it will be independent of G (except for node ν). The root map maps a substituend index to the root of the subgraph representing that substituend.

Given the lengthy preliminaries, our incremental reduction algorithm, Λ^{inc}, is remarkably short:

Definition 8.16 (Algorithm Λ^{inc})
Let A_0 be an element of $GTer_{\mathcal{L}}(\mathbf{\Lambda CCL})$, and $\mathcal{A}_0$ be its associated substituend decomposition. Let

$$\langle \Delta_1,\, \mathcal{S}_1\rangle, \langle \Delta_2,\, \mathcal{S}_2\rangle, \ldots$$

be a sequence of substitution sets and corresponding index sets. Then the algorithm Λ^{inc} is defined in Figure 8.13.

Lines (1) and (2) define the initial term $\hat{A}_0$ with appropriate $\mathrm{Dup}(\cdot)$ nodes inserted, along with its corresponding root map. Line (3) of Λ^{inc} causes $\triangle rwhnf2(\cdot)$ to be applied to the last term encountered, $\hat{A}_{i-1}$. In Line (4), we define the result of the reduction as A'_{i-1} as the null interpretation of $\hat{A}'_{i-1}$ (i.e., with respect to no substituends). Given a substitution set and corresponding index set, in Line (5), those $\triangle(\cdot,\,\cdot)$ nodes relevant to substituends in $\mathcal{S}_i$ are deleted to yield an initial value for $\hat{A}_i$ (since their corresponding substituends are to be replaced). In Line (6), the current member of the substitution set Δ_i has appropriate $\mathrm{Dup}(\cdot)$ nodes inserted. In Line (7), the term $\hat{A}_i$ is updated by replacing the root of the appropriate substituend of $\hat{A}_{i-1}$ with the current member of the substitution set (as updated in Line (6)). Note that since the node identifiers of the old root and new subterm are the same, the replacement operation requires no redirection. Finally, in Line (8), we update the current root map appropriately.

8.4.3 Analysis of Λ^{inc}

Our analysis of the behavior of Λ^{inc} will hinge on showing that it not only simulates a complete and call-by-need (and thus optimal) reduction ρ of any term $\hat{A}_i$ to which $\triangle rwhnf2(\cdot)$ is applied, but it also computes a *projection* of ρ on the context in which a substitution of elements from Δ_{i+1} is being performed at each step of an incremental computation.

We will need the following lemma:

Lemma 8.20 *Let $\hat{B}$ and $\hat{B}'$ be elements of $Graph1(\triangle\Lambda\mathrm{CCL}^G)$ such that $\triangle rwhnf2(\cdot)$ performs contraction*

$$\hat{B} \;\longrightarrow_{\triangle\Lambda\mathrm{CCL}}\; \hat{B}'$$

Then for all index sets $\mathcal{S}$,

$$\begin{array}{rcl} \mathit{left}_{\mathcal{S}}(\hat{B}) & \longrightarrow^{\equiv}_{\Lambda\mathrm{CCL}'} & \mathit{left}_{\mathcal{S}}(\hat{B}') \\ \mathit{right}_{\mathcal{S}}(\hat{B}) & \longrightarrow^{\equiv}_{\Lambda\mathrm{CCL}'} & \mathit{right}_{\mathcal{S}}(\hat{B}') \end{array}$$

Proof By definition, we know that

$$\mathit{interp}_{\mathcal{S}}(\hat{B}) \;=\; \mathit{left}_{\mathcal{S}}(\hat{B}) \sqcup \mathit{right}_{\mathcal{S}}(\hat{B})$$

Furthermore, since $\triangle\mathit{rwhnf2}(\cdot)$ implements reductions in $\mathbf{\Lambda CCL'}$, we know that

$$\mathit{interp}_{\mathcal{S}}(\hat{B}) \;\twoheadrightarrow_{\Lambda\mathrm{CCL}'}\; \mathit{interp}_{\mathcal{S}}(\hat{B}')$$

Therefore, we need only show that the contraction performed do not involve both $\mathit{left}_{\mathcal{S}}(\hat{B})$ and $\mathit{right}_{\mathcal{S}}(\hat{B})$. Note that this could only occur with either rule **(Beta)** or one of the rules concerning closures, since all other rules are concerned only with reductions in environments. However, in order for any such contraction to involve both partitions, it must first contract a **($\triangle$Apply)** or **($\triangle$C)** redex, which creates a copy of the the **(Beta)** redex or closure redex in the right child of a $\triangle(\cdot, \cdot)$ node. But then since $\triangle\mathit{rwhnf2}(\cdot)$ only contracts leftmost outermost redexes, this "preserved" redex is never contracted, and thus no contraction can involve both partitions. □

Lemma 8.20 says that the contractions performed by $\triangle\mathit{rwhnf2}(\cdot)$ respect the partition induced by the index structure of the substituends.

Next, we show that *complete reductions* also respect the partition induced by the index structure, and furthermore, that the partition-respecting reductions are maximal with respect to the number of families contracted.

Lemma 8.21 *Let $\hat{B}$ and $\hat{B}'$ be elements of Graph1$(\triangle\Lambda\mathrm{CCL}^G)$ and*

$$\rho \in C(\triangle\Lambda\mathrm{CCL}^G)^*$$

be a complete reduction performed by $\triangle$rwhnf2$(\cdot)$ such that

$$\hat{B} \;\stackrel{\rho}{\twoheadrightarrow}_{\triangle\Lambda\mathrm{CCL}}\; \hat{B}'$$

Then for all index sets $\mathcal{S}$, there exist complete reductions $\sigma, \tau \in C(\triangle\Lambda\mathrm{CCL}^G)^*$ such that

$$\begin{array}{rcl} \mathit{left}_{\mathcal{S}}(\hat{B}) & \stackrel{\sigma}{\twoheadrightarrow}_{\Lambda\mathrm{CCL}'} & \mathit{left}_{\mathcal{S}}(\hat{B}') \\ \mathit{right}_{\mathcal{S}}(\hat{B}) & \stackrel{\tau}{\twoheadrightarrow}_{\Lambda\mathrm{CCL}'} & \mathit{right}_{\mathcal{S}}(\hat{B}') \end{array}$$

and such that the FAM(σ) *and* FAM(τ) *are* maximal *subsets of* FAM(ρ).

Proof We have from Lemma 8.18 that $\triangle\mathbf{\Lambda CCL}^G$ is family-preserving. We then must show that the maximal number of families are preserved in each partition. Λ^{inc} establishes the invariant that shared families in more than one partition have a unique representative redex, by insertion of $\text{Dup}(\cdot)$ nodes. The only redexes that are subsequently copied by any contraction performed by $\triangle rwhnf2(\cdot)$are those (from the proof of Lemma 8.20) that involve more than one partition, which therefore could not be shared. Since the maximal number of redexes is shared among overlapping partitions, the maximal number of families must also be shared as well. $\square$

The following lemma is crucial to establishing the correctness and incremental optimality of Λ^{inc}:

Lemma 8.22 (Master Invariant for Λ^{inc}) *Let A_i be the ith term that is to be reduced by Λ^{inc} and $\mathcal{A}_i$ be its corresponding substituend decomposition, such that*

$$A_i \;=\; (A_{i-1} \dot{-} D_i^-) \sqcup D_i^+$$

Let $\hat{A}_i$ be the element of Graph1$(\triangle\mathbf{\Lambda CCL})$ created by Λ^{inc} at iteration i. Let $\hat{A}'_i$ be the graph such that

$$\hat{A}'_i \;=\; \triangle rwhnf2(\hat{A}_i)$$

Let $\mathcal{S}$ be a set of substituend indices. Finally, let

$$\begin{array}{rcl} C' &=& \mathit{left}_{\mathcal{S}}(\hat{A}'_i) \\ S' &=& \mathit{right}_{\mathcal{S}}(\hat{A}'_i) \end{array}$$

Then

$$\begin{array}{rcl} \mathit{cont}_{\mathcal{S}}(\mathcal{A}_i) & \twoheadrightarrow_{\Lambda\text{CCL}'} & C' \\ \mathit{subst}_{\mathcal{S}}(\mathcal{A}_i) & \twoheadrightarrow_{\Lambda\text{CCL}'} & S' \end{array}$$

Proof The invariant is clearly established by the initialization of $\hat{A}_0$, by the construction performed by $prt(\mathcal{A}_0, \nu, G)$. From Lemma 8.21, we know that the invariant is maintained by any reduction performed by $\triangle rwhnf2(\cdot)$. Finally, the invariant is re-established at each iteration by the left-removal operation, $removeL_{\mathcal{S}}(\cdot)$. $\square$

Lemma 8.22 says that the partition structure induced by $\triangle$ indices matches the partition structure of the substituend decompositions.

We can now show that $\triangle rwhnf2(\cdot)$ does indeed compute projections, and is thus incrementally optimal modulo ($\triangle$**Apply**), ($\triangle$C), and (**Duplicate**) contractions:

Theorem 8.23 *Let $\hat{A}_{i-1}$ and $\hat{A}_i$ be the $\triangle\mathbf{\Lambda CCL}$ terms to which $\triangle rwhnf2(\cdot)$ is applied at adjacent iterations of Λ^{inc}, such that*

$$\begin{aligned} \hat{A}'_{i-1} &= \triangle rwhnf2(\hat{A}_{i-1}) \\ \hat{A}'_i &= \triangle rwhnf2(\hat{A}_i) \end{aligned}$$

and ρ and σ be elements of $C(\Lambda\mathrm{CCL}')^*$ such that

$$\begin{aligned} interp_{\emptyset}(\hat{A}_{i-1}) &\xrightarrow{\rho}\!\!\!\!\rightarrow_{\Lambda\mathrm{CCL}'} interp_{\emptyset}(\hat{A}'_{i-1}) \\ interp_{\emptyset}(\hat{A}_i) &\xrightarrow{\sigma}\!\!\!\!\rightarrow_{\Lambda\mathrm{CCL}'} interp_{\emptyset}(\hat{A}'_i) \end{aligned}$$

Then σ is $C(\Lambda\mathrm{CCL}')^*, WHNF_{\Lambda\mathrm{CCL}}$ incrementally optimal relative to ρ.

Proof We first note that since Λ^{inc} uses $\triangle rwhnf2(\cdot)$, it simulates complete and call-by-need $\mathbf{\Lambda CCL}'$ reductions, and is $WHNF_{\Lambda\mathrm{CCL}}$-normalizing.

We also know that

$$A_i = (A_{i-1} \dot{-} D_i^-) \sqcup D_i^+$$

for some set of substitution indices $\mathcal{S}_i$. From Lemma 8.22, we have that

$$left_{\mathcal{S}_i}(\hat{A}_i) = cont_{\mathcal{S}_i}(\mathcal{A}_i) = B_i$$

where there exists reduction τ such that

$$(A_{i-1} \dot{-} D_i^-) \xrightarrow{\tau}\!\!\!\!\rightarrow_{\Lambda\mathrm{CCL}'} B_i$$

By Lemma 8.21, we know that τ is the projection of ρ on $(A_{i-1} \dot{-} D_i^-)$ or some term to which it reduces.

Therefore, since σ is complete and $WHNF_{\Lambda\mathrm{CCL}}$-call-by-need, and τ is the projection of a complete and call-by-need reduction ρ, we have by Theorem 7.3 that σ is incrementally optimal. □

8.5 Remarks

It is perhaps worthwhile to pause for a moment to reflect on how the ideas from previous sections have been assembled to produce an incremental reduction algorithm. First, we needed to define a notion of *computation*, and a notion of *structure* from which an increment could be derived. For this, we defined abstract replacement systems, combining reduction relations and Brouwerian algebras. However, in order to allow "optimal" to be well defined, we augmented abstract replacement systems with various axioms about the behavior of *residuals*, resulting in *regular* replacement systems. Our definition of regular replacement system, in concert with Lévy's notion of complete reduction, led to an abstract notion of optimal incrementality via the *projections* of a reduction on a substructure.

Since we wanted a notion of reduction whose computational power was effectively equal to that of the lambda calculus, yet was "better behaved" with respect to substitution (and thus more readily analyzed and implemented), we introduced the system $\mathbf{\Lambda CCL}$. We then added restrictions to the rules of $\mathbf{\Lambda CCL}$ to yield the conditional rewriting system $\mathbf{\Lambda CCL}'$, which (unlike $\mathbf{\Lambda CCL}$) forms an RRS. Graph rewriting equivalents of $\mathbf{\Lambda CCL}'$ and $\triangle\mathbf{\Lambda CCL}$ made possible *implementations* of complete, and thus optimal, reductions.

Finally, in order to simultaneously represent various partitions of the initial term, and thus compute the projection of a reduction on the context in which some set of substituends might be replaced, we defined $\triangle\mathbf{\Lambda CCL}$.

Our model of incrementality requires that the possible points of for replacement be designated in advance. While at first glance this might seem a severe restriction, we note that one can, if desired, specify *every* subterm of a term as a possible point of replacement. However, the overhead required to maintain the attendant $\triangle$ node "checkpoints" is of course increased accordingly.

$\Lambda^{\text{inc}} \equiv$

seq;

{1} $\hat{\boldsymbol{A}}_0 \triangleq \boldsymbol{prt}(\mathcal{A}_0, \nu, \perp);$ $\{\nu\ \textit{is arbitrary}\}$

{2} $\boldsymbol{currRootMap} := rootMap[\hat{A}_0](\cdot);$

for $i \in 1 \ldots$ **do**

seq

{3} $\hat{A}'_{i-1} \triangleq \triangle rwhnf2(\hat{A}_{i-1});$

{4} $A'_{i-1} \triangleq interp_{\emptyset}(\hat{A}'_{i-1});$

{5} $\hat{A}_i \triangleq removeL_{\mathcal{S}_i}(\hat{A}'_{i-1});$

for all $s_j \in \mathcal{S}_i$ and $\mathcal{B}_i^{s_j} \in \Delta_i$ **do**

seq

{6} $\hat{\boldsymbol{B}}_i^{s_j} \triangleq prt(\mathcal{B}_i^{s_j}, \boldsymbol{currRootMap}(s_j), \hat{A}_{i-1});$

{7} $\hat{A}_i := \hat{A}_i[currRootMap(s_j) \leftarrow \hat{B}_i^{s_1}];$

{8} $currRootMap :=$

$(\boldsymbol{currRootMap} - rootMap[\hat{A}_{i-1}^{s_j}](\cdot)) \cup rootMap[\hat{\boldsymbol{B}}_i^{s_j}](\cdot);$

endseq

endseq

Figure 8.13: Algorithm Λ^{inc}

Chapter 9

Related Work

> "A lady once told Sir William Cummings that his adored Purcell was only an imitator of Handel. When Sir William suggested the slight difficulty that Handel was only six years old when Purcell died, the lady said 'Oh, if you drag in dates you can prove anything' "
>
> —Tovey

In this chapter, we will review work by other authors concerned with general systems for incremental computation. Since it is extremely difficult to make precise comparisons of performance among the widely differing models of computation employed by other systems, our review here will necessarily be of a qualitative, rather than a quantitative nature. One of the greatest impediments to making meaningful comparisons is that in most of the general systems of which we are aware, there is no obvious way to to determine whether the incremental computations performed *within* the systems are the most efficient possible, much less how their performance can be compared in general to that of other systems. Indeed, one of the advantages of the incremental reduction approach we have taken here is that a notion of optimality exists at all. Nonetheless, there are numerous aspects of other work in comparison to our own that merit mention.

9.1 Lombardi's Pioneering Work

The term "incremental computation" was apparently coined by Lombardi in his 1967 paper of the same name [Lom67]. The paper contains a rather elaborate proposal[1] for a complete system (including hardware) to evaluate a LISP-like language in incremental manner. Lombardi's idea was to allow expressions containing free variables to be partially (and thus lazily) evaluated to yield a new expression. By later binding values to the free values in a partially evaluated expression, the

[1]The paper is subtitled "The Preliminary Design of a Programming System Which Allows for Incremental Data Assimilation in Open-Ended Man-Computer Information Systems."

expression could be evaluated further. Lombardi's notion of increment-as-binding thus only allowed for a monotonic increase in the amount of information available to the computation; it did not incorporate any notion of computation on input created by deletion or alteration of previous values. Lombardi's goal was apparently less one of avoiding redundant computation than allowing input to be specified lazily by the programmer.

One might nonetheless reasonably describe Lombardi's ideas as visionary, since his system design incorporated a form of functional programming, an interactive mode of computation much like that of a "spreadsheet" program, and a form of partial evaluation—none of which have acquired wide currency until quite recently. Lombardi also proposed no formal analysis of performance or computational capability of his methods. Nonetheless, his ideas are remarkably close in basic form, if not in their details, to those developed here.

9.2 Incremental Attribute Evaluation and Value Propagation Algorithms

The work of Reps, Teitelbaum, and Demers on incremental evaluation of attribute grammars [RTD83,Rep84] is the only other notion of general-purpose incremental computation of which we are aware that possesses a formal optimality property. Incremental attribute evaluation has proven to be particularly useful in interactive systems for program development (e.g., the Synthesizer Generator system [RT88]), although in principal it can be used in other situations as well.

9.2.1 Attribute Grammars

An attribute grammar consists of three fundamental components:

- An abstract syntax specification, which can be viewed as a mutually recursive datatype definition defining a set of sorted terms.
- A specification, for each term sort, of a fixed set of *attributes*: annotations intended to represent computed values that characterize the term.
- A set of *attribute equations*, defining the values of attributes of a given term sort based on values of attributes of constituent or containing terms.

In the Synthesizer Generator system, the terms defined by the grammar are used to describe the structural form of editable objects. The attributes are used for computing and displaying various sorts of "derived information" about the editable object; e.g., if the editable object were a program, attributes could be used to compute and display type information about the program.

The equations of an attribute grammar are best thought of as a network of functions defined inductively on the structure of the abstract syntax. Input and output values from attribution functions are connected via edges in the network. The attribution functions can then be viewed as jointly specifying a single function from terms in the grammar to sets of output attribute values at each node in the network[2]. An *attribute evaluator* [DJL86] is then provided to evaluate the attribute functions in the network, determining consistent values for each attribute instance.

Depending on how the attribute grammar is to be used, it is often the case that only a subset of the attribute instance values are actually needed at any one time. Efficient attribute evaluators only evaluate those attribution functions whose values are required by a given application, and moreover avoid evaluating the same function more than once. The Synthesizer Generator is capable of producing a variety of attribute evaluators, which exploit various time/space trade-offs and different modes of use.

In the incremental case, the network of functions may change as a result of changes to the abstract syntax tree. Any incremental attribute evaluator must account for these changes and avoid computing the results of more attribute functions than are necessary to correctly account for the changes to the tree.

The attribute evaluators produced by the Synthesizer Generator take advantage of the fact that many, if not most, of the attribute values are unaffected by such an editing change, and endeavor to calculate the new attribute values with a minimum of work.

9.2.2 Quiescence Propagation

The primary incremental attribute evaluation mechanism employed by the Synthesizer Generator is *quiescence propagation*. Quiescence propagation relies on the following invariants maintained by the system:

- The dependence structure of the attribution function network is guaranteed to be acyclic.
- Every attribute value computed by an attribution function is stored with the abstract syntax tree.
- A topological sort of the input/output dependence edges in the attribution network is dynamically maintained as the underlying tree is edited.

When a subtree in the abstract syntax tree is replaced, the attribution functions are evaluated in an order consistent with the topological sort of the dependence edges, beginning with the point of change. Values of attribution functions are

[2]Mayoh [May81] shows that attribute grammars indeed represent mathematically well-defined functions. For an example of a relatively straightforward translation of an attribute grammar into an algorithm consisting of a set of recursive procedures, see [Kat84]

thus *propagated* along dependence edges. Before each attribution function is to be evaluated, the current values of its inputs are compared with their previous values (which have been stored with the syntax tree). If none of the values have changed, propagation (for that part of the dependence structure) cease, or *quiesces*. Since the network is acyclic, propagation eventually ceases.

9.2.3 Optimality Properties of Quiescence Propagation

The incremental evaluator of [RTD83] guarantees that the number of attribution functions evaluated is proportional to the number of functions whose input values are altered, or *influenced* as a result of the subtree replacement. Since this number is proportional to the number of attributes whose values change, the evaluator is optimal with respect to this measure.

9.2.4 Other Frameworks for Change Propagation

Starting with the basic idea of [RTD83], several other systems have been proposed that perform incremental computation by propagating changed values throughout function networks. Work in this area includes that of Hoover [Hoo87] and Alpern, et.al. [ACR+87].

9.2.5 Limitations of Change Propagation

While change propagation strategies have amply demonstrated their practical value, their limitations make them much less well-suited to some applications than others. We outline some of these drawbacks below:

Granularity

In existing change propagation frameworks, the attribution functions are considered to be "atomic" computations. Thus the optimality results of [RTD83] are only expressed in terms of the *number* of attribution functions evaluated, rather than taking into account the internal complexity of any of the functions themselves. When the computation represented by a change propagation framework is defined inductively on the size of the dependence graph structure, quiescence propagation works rather well. If, however, some of the attribution functions perform non-trivial tasks, the cost of a computing a single attribution function can dominate the the time required to perform quiescence propagation.

Acyclicity

Most of the change propagation frameworks require that the network dependence structure be acyclic. This is at odds with the iterative nature of many useful algorithms, particularly those that perform dataflow analysis for program compilation. Walz and Johnson, however, have proposed methods for evaluating certain classes of circular attribute grammars [WJ88].

Aggregates

Finally, we come to the problem of *aggregates*. Even when a problem has a natural inductive definition on the dependence structure of an attribution network, such a structure may come at the expense of propagating large aggregate values between functions. For instance, many applications in which attribute grammars are used require that symbol tables be propagated throughout the abstract syntax tree. Even if these aggregate values are shared to as great an extent possible, the very fact that they are used makes it less likely that quiescence will occur quickly, since small changes within an aggregate require that any function that uses its value be re-evaluated. Hoover has developed special aggregate types for use in change-propagation frameworks [Hoo87]. However, a completely general solution to the "aggregate problem" in change propagation frameworks is not yet at hand.

It is clearly the case that in general, incremental reduction will preserve results of intermediate computations of much finer grain than possible with change-propagation frameworks. Incremental reduction is much more general, and does not require special methods (as does change propagation) to handle situations in which values are computed on demand—in fact, incremental reduction is by definition as "lazy" as possible.

9.3 Differential Propagation

Several authors have proposed an interesting twist on the idea of change propagation. Instead of allowing general changes to the network of attribution functions used in change propagation, they assume a more-or-less *fixed* network of functions, and instead investigate changes only to user-supplied *inputs* to the network. These inputs take the form of aggregate datatypes, and changes by the user are made to sub-elements of the aggregates.

Rather than propagating entire aggregates, *differential*, or "delta" values are propagated instead. The language in question is designed so that functions defined over aggregates have predefined variants that operate on delta values. Differential values then propagate throughout the fixed function network until the resulting differences are null or the outputs of the network are reached. Sometimes intermediate values are stored in the network so that quiescence propagation can be performed.

Examples of differential propagation include Bengelloun's incremental SCHEME [Ben82], in which a change to a list element in a function's input propagates to produce a consequent change to the result of the function, and Yellin and Strom's language INC [YS89], in which the aggregate datatype in question is the *bag* or multiset.

9.3.1 Remarks

Differential propagation works well when the "derivative" functions are well defined for the problems and data structures at hand. It is particularly good at taking advantage of "domain-specific" knowledge about derivative values, which the much more general framework of incremental reduction is less effective at exploiting. However, differential propagation does not work well when there is no obvious notion of "delta" value.

9.4 Pure Function Caching

Function caching is a straightforward idea that has labored under the unfortunate reputation of being something of a "programmer's trick," and therefore has apparently escaped much systematic study. The concept was first discussed in print by [Mic68].

The idea is quite simple, and not at all inelegant: Start with a program written in a functional, or nearly functional language. A "memo" is made of every function call performed in the course of a program's execution (hence such functions are often referred to as *memo functions*). The name of the function, the value of its arguments, and the result of the function on those arguments is recorded and stored. Every invocation of a function with the same arguments as used in a previous call will make use of the result recorded on the "memo" made of the previous invocation, rather than computing the same result again from scratch. We will call this idea *pure function caching* to distinguish from other schemes in which certain values are cached in concert with other incremental techniques.

Many efficient programs go to great pains to *explicitly* "cache" the values of computations that are likely to be reused. For instance, the programming technique known as "dynamic programming," used for certain optimization problems [Bel57], maintains a table of computations on successively larger subproblems, using it to search an exponentially large solution space in quadratic time. A much more naive solution to the same problems normally solved by dynamic programming can achieve a solution in the same amount of time with dynamic function caching.

Function caching can yield incremental behavior without an explicit representation of a Δ set: If a function is computed on a composite value by decomposing it into subproblems on component pieces of the original value, computing the function on a value that differs only slightly from a previous value will take advantage of the fact that subproblems that have been computed on invariant pieces of the input will not be re-computed. The only significant computation performed will be that required for the part of the original value that was altered.

In implementing function caching, the greatest difficulty is managing the amount of information remembered in the cache, since the number of recorded values can increase rapidly (using the framework of section 1.2, we can view the cache as the "baggage", Z, carried from one function call to the next). Special techniques (usually making use of hashing) are required to implement rapid lookup and retrieval of key/value pairs, where the keys may be large composite values used as function arguments. For function caching to yield acceptable incremental behavior, the original algorithm must be defined inductively on the structure of its arguments, so that small changes to the argument will be reflected by a large number of calls to cached functions. New function computations will only be required on that part of the original input that was altered. Pugh's work [Pug88] discusses special datatypes that are particularly simple to decompose, and thus for which function caching is especially effective [Pug88].

9.4.1 Remarks

Pure function caching has the virtue of great conceptual simplicity. Like incremental lambda reduction, it works for arbitrary recursive functions. Like change propagation, it works best when functions are defined inductively on the structure of inputs and when arbitrary aggregate values are avoided. It also does not lend itself naturally to application in settings requiring higher-order values (since they cannot be readily compared). Nonetheless, Pugh has shown that in *conjunction* with change propagation, caching can be quite beneficial. Function caching also has the virtue of capturing "equivalent" computations that are not equivalent in the sense of the notion of residual. One is essentially exploiting the fact that function equality is easy to determine when the function's names are the same. On the other hand, with function caching, one may end up storing numerous values that are in fact not used in subsequent computations, and in fact *cannot* be used subsequently, in the vain hope that they may eventually reappear.

9.5 Partial Evaluation and Residual Function Algebras

The work of Sundaresh and Hudak in [SH90] is perhaps the closest to that we present here. Their framework assumes a program written in a functional language. They also assume that there is a predefined *partition* of the inputs, defined in terms of a Brouwerian algebra of *projection functions* (rather than a data algebra as is used here). By composing the function to be computed with the projection function used to select part of the input, we get a function that has been "specialized" with respect to that part of the input.

Given an actual input, their idea is to partially evaluate each specialized version of the function to be computed. Since the projection function ignores all but one element of a given partition, the result of the partial evaluation will have free variables representing each of the other elements in the partition. The result of a partial evaluation is a so-called *action tree*, which is essentially a term representing the result of the partial evaluation. Action trees have the property that there is a combining algebra of trees representing projections of the function to be evaluated.

When one element of the input partition is changed, the partial evaluation is re-invoked only for the projection corresponding to the altered input. The new action tree for the altered partition is combined with the trees for the unaltered partitions, and the resulting composite partition is reduced until a normal form is reached.

9.5.1 Remarks

In contrast to the methods proposed here, Sundaresh and Hudak require that the specialized functions for the various partitions be evaluated in advance of any change, regardless of whether the computations performed therein will ever be used to produce any output. Note also that computations which are created by combining several partition elements whose inputs do not change are repeatedly re-evaluated every time the specialized functions are combined. Thus while their notion of incrementality captures computations that are specific to a single element of a partition, they have no mechanism for avoiding recomputation of any results that arise from combinations of partitions. As with pure function caching techniques, unless the partition is chosen very carefully, the computations in the joins of the partition elements may dominate those that are avoided by pre-evaluating the specialized functions. By contrast, the performance of incremental reduction depends only on the nature of the substituend that is replaced, not on all the other substituends that are left unchanged.

9.6 Computational Complexity Issues

We conclude this chapter with a mention of work examining the connection between incremental algorithms and computational complexity. Not only are questions about this subject interesting in their own right, but answers to such questions also serve to bound the effectiveness of techniques applicable to the automatic generation of incremental algorithms.

[BPR85], using a model of incremental computation that differs from ours, prove some interesting results about the *relative* time complexity of incremental algorithms. They define a measure of "improvement potential" for incremental algorithms that they call an *incremental relative lower bound.* For certain classes of problems, they show that the incremental version of an algorithm can improve on the time complexity of its non-incremental counterpart by only so much. This improvement potential is expressed as $1/\rho(n)$, where $\rho(n)$ is a function of input size. They show that if a problem has an incremental relative lower bound of $1/\rho(n)$ and we let $T(n)$ be the lower bound for the time complexity of the non-incremental problem, then the time complexity of any incremental (under their model) algorithm for the problem can have complexity *no better than* $T(n)/\rho(n)$. Thus for problems with known lower bounds (such as sorting), their techniques give an immediate bound for the complexity of an incremental algorithm used to solve the problem. As alluded to earlier, [PCB84] show (using the same model as [BPR85]) that there are no polynomial incremental algorithms for *NP-complete* problems, unless $P = NP$, although for some problems, a polynomial *improvement* can still be achieved.

Unfortunately, the computational model of [BPR85] does not allow history information from previous computations to be used in computing new results incrementally.

Chapter 10

Conclusions

"The criticism that [it] was 'too long' merely annoyed [the author], and was the last thing in the world that would have induced him to alter his work."

"Here the coda is to be... retrospective and quiet."

"Though the manner of the end is abrupt, the matter of the coda is conspicuous for its breadth."

—Tovey

10.1 Future Work

The work discussed in this thesis lends itself to numerous additional investigations, and we outline some of our thoughts for possible directions below:

- Our investigation of incremental reduction has been on a rather theoretical level in this thesis. Nonetheless, we believe there is a reasonable potential for the ideas herein to be applied to more practical languages. One reason for believing this to be the case is that a great deal of progress has been made in efficient implementations of functional languages. We would hope to take further advantage of this work.

- We have considered only the untyped λ-calculus in our investigations here, since it is oblivious to issues of well-formedness of functions or terms. However, by considering various typed λ-calculi, one might be able to extract more interesting or efficient incremental algorithms.

- By restricting the set of λ-terms considered, e.g., to those of the λI-calculus or to those having the form of a "supercombinator" [Hug84b], one might be able to simplify the substitution operations required by Λ^{inc}, and perhaps facilitate some compiled form of it.

- An investigation of the time/space tradeoffs in storing partially evaluated graphs and their connection with projections would be extremely valuable. It seems that one could define a notion of computational complexity based on the "density" of an incrementally evaluated graph with embedded "fork" nodes.

- It would be interesting to investigate complete reductions in regular replacement systems as a model of computational complexity. Time complexity would be defined in the obvious way by the number of complete contractions performed in a reduction to a normal form. Unlike automata-based models of complexity, RRS's have the interesting property that there are many complete call-by-need reductions of the same element, all possessing the same length. This property might be exploited to prove lower bounds results based on adversary arguments.

- Klop has defined a notion of *combinatorial reduction system* [Klo80] which combines most of the essential features of the λ-calculus and regular term rewriting systems. It would be interesting to try to couch his work in terms of regular reduction systems.

- Plotkin's notion of *structural operational semantics* (see, e.g., [Kah87]) is closely related to the idea of a conditional rewriting system. It seems a particularly appropriate vehicle for investigation of the potential for incremental evaluation of arbitrary languages, particularly those such as ML whose semantics have already been defined.

- The so-called program dependence graphs used in program optimization and parallelization, also investigated by Reps in his study of program integration [Rep89] appears to provide a basis for investigating incrementality in arbitrary languages. Determining its connection with structural operational semantics might also prove fruitful.

- Our use of "fork" nodes in **ΛCCL** would seem to be generalizable to other term/graph rewriting systems. A generalization of the notion of fork, and the consequent generalization of implementation techniques for incremental reduction to a broader class of rewriting systems would clearly be useful.

- The definition of Regular Replacement System (Definition 3.81) required that the extraction relation on families, $\longrightarrow_{\text{ext}}$, be noetherian. While this condition in fact holds for every practical reduction system of which we are aware, it would be desirable to replace it with a strictly *local* property (i.e., one based on residuals of single contractions) which would imply that $\longrightarrow_{\text{ext}}$ is noetherian.

- We strongly suspect that the "auxiliary" system $\mathbf{ACCL}_0'$ defined in Chapter 8 is in fact a weak monomorphic extension of β, and thus conservative extension of β; it would be interesting to prove it.

- Several authors, particularly Ehrig, et al. [EKMS+81], have investigated category-theoretic approaches to reduction. It would be useful to try to couch our theory of regular abstract replacement systems in category-theoretic terms.

- It would be interesting to investigate the algebraic, i.e., lattice-theoretic properties of projections, and their interaction with the properties of relatively equivalent reductions.

- Between the entirely ad-hoc, problem-dependent kind of incremental algorithm and the automatically generated, problem-independent variety, we can envision a hybrid of sorts: A programmer might give an initial specification of an algorithm using a fairly conventional programming notation. One could then generate an equivalent incremental version. Before doing so, however, we could first augment the original algorithm specification with an indication of the intended semantics of the more *abstract* structures that the programmer is implementing in the language's base constructs, e.g., if the program implements stacks using linked lists, it might be annotated with predicates describing the semantics of stacks (which are considerably more restricted than those of arbitrary lists). Such annotations could be used to improve the generated incremental algorithm, taking advantage of the intended semantics of the abstract objects.

10.2 Final Words

The goal of this thesis has been to define a completely general model of computation in which precise notions of incrementality and optimality can be defined. In defining the notion of regular replacement system, we believe that we have succeeded in providing such a model. Furthermore, we have demonstrated that there are RRS's with computational power equivalent to the lambda calculus. We believe that many of the results obtained along the way to the definition of Λ^{inc} will yield interesting insights of their own, independent of any application to incremental computation.

As for incremental reduction itself, we believe it can be applied quite directly to many applications. In particular, we believe that there are many applications for which computation may naturally be viewed as reduction, and for which the ability to rapidly recompute results after dynamic changes to input would be extremely valuable, if it were available transparently and performed efficiently. Such applications include:

- Rewriting transformations performed by optimizing compilers.
- Interpreters for shell languages.
- "Macro" definitions in editors and programming environments.
- Symbolic algebra and equation solving systems.
- Theorem-proving and heuristic problem-solving systems.

Each of these applications can either be viewed as some specialized reduction system, or as a language whose semantics can be encoded functionally, e.g., via the lambda calculus.

In addition to the applications above, we believe there are many other applications not obviously of a "reductionist" character for which incremental computation will prove extremely valuable *if they do not require the programmer to be concerned with incrementality*. In order for this to be possible, incremental systems must be allow as broad a class of computations as possible. This has been our primary goal: to define a completely general system in which incremental computation can be precisely characterized and efficiently carried out. We believe this has been done, and look forward to applying these ideas in practice.

"...and on the last quaver of its 2nd bar the Fugue ends. Yes, ends! One-and-two-and-three-and——"

—Tovey

Appendix A

Supplementary Proofs

A.1 Proof of Lemma 5.10

In this section, we give the proof for Lemma 5.10, which will first require some preliminary Lemmas and definitions.

We first define two subsystems of **ECCL** that will be useful in the sequel.

Definition A.1 *The axioms of* **FCCL** *consist of those of* **ECCL** *without rule* **(DApply)**.

Definition A.2 *The axiom* **(VarRef′)** *is as follows:*

$$[\mathrm{Var},\ \langle E_1,\ [A,\ E_2]\rangle] = [A,\ E_2]$$

Note that **(VarRef′)** is simply **(VarRef)** restricted to a smaller set of terms.

Definition A.3 *The axioms of* **GCCL** *are as follows:*

(AssC), **(DE)**, **(AssE)**, **(ShiftE)**, **(NullER)**, **(VarRef′)**

Using these new systems, we now obtain the following results:

Lemma A.1 **FCCL** *and* **(Beta)** *commute, i.e.,*

$$\begin{tikzcd} {} \arrow[r, "\text{(Beta)}"] \arrow[d, "\text{FCCL}"'] & {} \arrow[d, dotted, "\text{FCCL}"] \\ {} \arrow[r, dotted, "\text{(Beta)}"'] & {} \end{tikzcd}$$

Proof **FCCL** and **(Beta)** have no critical pairs and therefore commute weakly. Since both are noetherian, the result follows by noetherian induction. □

Lemma A.2 *Let* $\mathbf{GCCL}^{-1}$ *be the rewrite system obtained by orienting the equations of* **GCCL** *from* right to left. *Then* $\mathbf{GCCL}^{-1}$ *and* **(Beta)** *commute, i.e.,*

$$\begin{tikzcd}
\cdot \arrow[r, twoheadrightarrow, "\text{(Beta)}"] \arrow[d, twoheadrightarrow, "\text{GCCL}^{-1}"'] & \cdot \arrow[d, twoheadrightarrow, dotted, "\text{GCCL}^{-1}"] \\
\cdot \arrow[r, twoheadrightarrow, dotted, "\text{(Beta)}"'] & \cdot
\end{tikzcd}$$

Proof $\mathbf{GCCL}^{-1}$ and **(Beta)** have no critical pairs, thus commute weakly. The result follows by noetherian induction on the **(Beta)** reduction. □

Lemma A.3 **GCCL** *and* **(Beta)** *reductions may be* permuted, *that is, the following diagram holds:*

$$\begin{tikzcd}
 & \cdot \arrow[dr, twoheadrightarrow, "\text{(Beta)}"] & \\
\cdot \arrow[ur, twoheadrightarrow, "\text{GCCL}"] \arrow[dr, twoheadrightarrow, dotted, "\text{GCCL}"'] & & \cdot \\
 & \cdot \arrow[ur, twoheadrightarrow, dotted, "\text{(Beta)}"'] &
\end{tikzcd}$$

Proof Simply reverse the arrows of the $\mathbf{GCCL}^{-1}$ reductions used in Lemma A.2 to obtain **GCCL** reductions and the desired result (this technique was suggested by Proposition 5.5 of [Klo80, p. 46]). □

We are now in a position to prove Lemma 5.10:

Lemma 5.10 *(given by the diagram below)*

$$\begin{tikzcd}
\cdot \arrow[rrr, twoheadrightarrow, "\text{(Beta)}", "\rho"'] \arrow[d, "\text{ECCL}"', "\sigma"] & & & \cdot \arrow[d, twoheadrightarrow, dotted, "\text{ECCL}"] \\
\cdot \arrow[r, twoheadrightarrow, dotted, "\text{ECCL}"'] & \cdot \arrow[r, twoheadrightarrow, dotted, "\text{(Beta)}"', "\tau"] & \cdot \arrow[r, twoheadrightarrow, dotted, "\text{ECCL}"'] & \cdot
\end{tikzcd}$$

Let ρ *and* τ *be the* **(Beta)** *reductions as marked in the diagram above. Then if the redexes in* ρ *are disjoint, the redexes in* τ *are also disjoint.*

Proof Let σ and ρ be the **ECCL** contraction and **(Beta)** reduction, respectively, in the premise of the Lemma as shown in the diagram above.

We first note that if neither σ nor any of its residuals (relative to ρ) creates an instance of a critical pair with one of the redexes in ρ, then the diagram can be closed trivially (using residuals), which must also be disjoint if the **(Beta)** redexes in ρ are disjoint. Otherwise, the only rule that can create an instance of a critical pair is **(DApply)**.

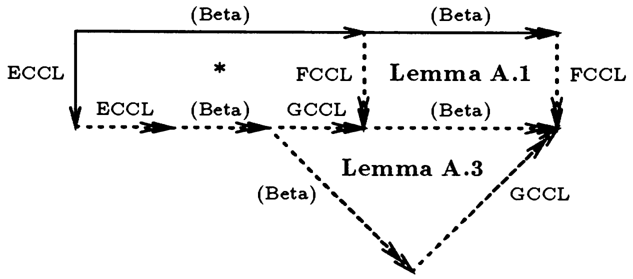
Since rule **(Beta)** can create no new **(Beta)** redexes, the redexes in ρ can be permuted arbitrarily. Thus if σ is an application of rule **(DApply)** that creates an instance of a critical pair with some redex in ρ, we can reorder the redexes in ρ so that the contraction of the overlapping redex occurs first.

Therefore, we assume the following without loss of generality:

(1) σ is an application of rule **(DApply)**.

(2) There is an instance of a critical pair between the first redex contracted in ρ and σ.

Given the assumptions above, we can construct the following diagram using Lemmas A.1 and A.3:



The square in the diagram marked '$*$' is easy to construct using rules of the appropriate sort starting from the critical pair of terms created by an overlap of rules **(DApply)** and **(Beta)**. By treating rules of **FCCL** and **GCCL** as **ECCL** rules, we obtain the required diagram. We finally note that the constructions in '$*$', Lemma A.1, and A.3 each yield **(Beta)** reductions in their conclusions with disjoint redexes if the redexes in the **(Beta)** reductions of their premises are also disjoint. □

A.2 Proof of Lemma 3.10

In this section, we give the proof for Lemma 3.10.

Lemma 3.10 *Let $\langle \mathcal{L}, \sqcup, \sqcap, \dot{-}, \top \rangle$ be a Brouwerian algebra with $a, b, c, d \in \mathcal{L}$ and a, b, and c be mutually disjoint, i.e., $a \parallel b$, $a \parallel c$, and $b \parallel c$. Then the following hold:*

$$a \dot{-} b = a \tag{1}$$
$$(a \sqcup b) \dot{-} a = b \tag{2}$$
$$a \parallel (b \sqcap c) \tag{3}$$
$$a \parallel (b \sqcup c) \tag{4}$$
$$(a \sqcup b) \sqcap (b \sqcup c) = b \tag{5}$$
$$a \dot{-} d \parallel b \tag{6}$$

Proof

(1) We wish to show $a \dot{-} b = a$. From $a \parallel b$, we get the following:

$$
\begin{aligned}
a \dot{-} (a \dot{-} b) &\sqsubseteq a \sqcap b && \text{(A.3.1)} \\
a \dot{-} (a \dot{-} b) &\sqsubseteq \bot && \text{(A.3.2)} \\
a \dot{-} (a \dot{-} b) &= \bot && \text{(A.3.3)}
\end{aligned}
$$

(A.3.1) follows from Brouwerian algebra property B.27. Since $a \parallel b$, we have by definition $a \sqcap b = \bot$, from which we get (A.3.2). From property B.2 we can conclude that

$$(a \dot{-} b) \sqsupseteq a \qquad \text{(A.3.4)}$$

But by property B.11, we also have that

$$(a \dot{-} b) \sqsubseteq a \qquad \text{(A.3.5)}$$

Combining (A.3.4) and (A.3.5) we get

$$a \dot{-} b = a \qquad \text{(A.3.6)}$$

as desired.

(2) We wish to show $(a \sqcup b) \dot{-} a = a$. Since $a \parallel b$, we have:

$$
\begin{aligned}
(a \sqcup b) \dot{-} a &\sqsupseteq b \dot{-} a && \text{(A.3.7)} \\
(a \sqcup b) \dot{-} a &\sqsupseteq b && \text{(A.3.8)}
\end{aligned}
$$

(A.3.7) follows from Brouwerian algebra property B.10. (A.3.8) is a consequence of property (1) above. However, we also have by property B.20 that

$$(a \sqcup b) \dot{-} a \sqsubseteq b \qquad \text{(A.3.9)}$$

Combining (A.3.8) and (A.3.9) we get

$$(a \sqcup b) \dot{-} a = b \qquad \text{(A.3.10)}$$

as desired.

(3) We wish to show $a \parallel (b \sqcap c)$. Since $b \parallel c$, we have $b \sqcap c = \bot$. Thus

$$a \sqcap (b \sqcap c) = \bot$$

and thus by definition of disjointness, $a \parallel (b \sqcap c)$.

(4) We wish to show $a \parallel (b \sqcup c)$. Since $a \parallel b$ and $a \parallel c$, we have

$$a \sqcap b = \bot \quad \text{and} \quad a \sqcap c = \bot \tag{A.3.11}$$

From Brouwerian algebra property B.41, we have

$$\begin{aligned} a \sqcap (b \sqcup c) &= (a \sqcap b) \sqcup (a \sqcap c) && \text{(A.3.12)} \\ &= \bot \sqcup \bot && \text{(A.3.13)} \\ &= \bot && \text{(A.3.14)} \end{aligned}$$

We then have

$$a \parallel (b \sqcup c) \tag{A.3.15}$$

which follows from (A.3.14) by definition of disjointness, and we are done.

(5) We wish to show $(a \sqcup b) \sqcap (b \sqcup c) = b$. We have the following sequence of equations:

$$\begin{aligned} (a \sqcup b) \sqcap (b \sqcup c) &= ((a \sqcup b) \sqcap b) \sqcup ((a \sqcup b) \sqcap c) && \text{(A.3.16)} \\ &= b \sqcup ((a \sqcup b) \sqcap c) && \text{(A.3.17)} \\ &= b \sqcup \bot && \text{(A.3.18)} \\ &= b && \text{(A.3.19)} \end{aligned}$$

(A.3.16) follows from property B.41. (A.3.18) is a consequence of property (4) above. (A.3.19) is the result we seek.

6 We wish to show $(a \dot{-} d) \parallel b$. By property B.11, we know that

$$(a \dot{-} d) \sqsubseteq a$$

from which we get the following equations:

$$\begin{aligned} a \sqcap b &= \bot && \text{(A.3.20)} \\ (a \dot{-} d) \sqcap b &\sqsubseteq \bot && \text{(A.3.21)} \\ (a \dot{-} d) \sqcap b &= \bot && \text{(A.3.22)} \end{aligned}$$

(A.3.20) is by definition of disjointness. (A.3.21) follows from property B.39. (A.3.22) implies $(a \dot{-} d) \parallel b$, the result we seek.

□

The properties of disjoint elements given by Lemma 3.10 are summarized in Appendix B.2.

Appendix B

Elementary Properties of Brouwerian Algebras

The following are some elementary identities of Brouwerian Algebras, taken primarily from Reps' excellent survey in [Rep89], as well as from [RS63] and [MT46].

B.1 Brouwerian Algebra Properties

Let

$$\mathcal{B} \equiv \langle \mathcal{L}, \sqcup, \sqcap, \dot{-}, \top \rangle$$

be a Brouwerian algebra. Then the following hold for all elements of $\mathcal{L}$:

$$a \dot{-} b = \min\{z \mid a \sqsubseteq (b \sqcup z)\} \tag{1}$$

$$b \dot{-} a = \bot \quad \text{iff} \quad a \sqsupseteq b \tag{2}$$

$$a = b \quad \text{iff} \quad b \dot{-} a = \bot = a \dot{-} b \tag{3}$$

$$a \dot{-} a = \bot \tag{4}$$

$$\bot \dot{-} a = \bot \tag{5}$$

$$b \dot{-} \bot = b \tag{6}$$

$$(a \dot{-} a) \sqcup b = b \tag{7}$$

$$a \sqcup (b \dot{-} a) \sqsupseteq b \tag{8}$$

$$a_1 \sqsupseteq a_2 \quad \text{implies} \quad b \dot{-} a_2 \sqsupseteq b \dot{-} a_1 \tag{9}$$

$$b_1 \sqsupseteq b_2 \quad \text{implies} \quad b_1 \dot{-} a \sqsupseteq b_2 \dot{-} a \tag{10}$$

$$b \sqsupseteq b \dot{-} a \tag{11}$$

$$a \sqcup (b \dot{-} a) = a \sqcup b \tag{12}$$

$$(b \dot{-} a) \sqcup b = b \tag{13}$$

$$(b \dot{-} a) \sqcup (c \dot{-} a) = (b \sqcup c) \dot{-} a \tag{14}$$

$$(c \dot{-} a) \sqcup (c \dot{-} b) = c \dot{-} (a \sqcap b) \tag{15}$$

$$
\begin{array}{rcll}
(c \dot{-} b) \dot{-} a & = & c \dot{-} (a \sqcup b) \quad = \quad (c \dot{-} a) \dot{-} b & (16) \\
a \dot{-} c & \sqsupseteq & (b \dot{-} c) \dot{-} ((b \dot{-} a) \dot{-} c) & (17) \\
(b \dot{-} a) \sqcup (c \dot{-} b) & \sqsupseteq & c \dot{-} a & (18) \\
b \dot{-} a & \sqsupseteq & (c \dot{-} a) \dot{-} (c \dot{-} b) & (19) \\
a & \sqsupseteq & (a \sqcup b) \dot{-} b & (20) \\
c \sqcup ((c \sqcup b) \dot{-} (c \sqcup a)) & = & c \sqcup (b \dot{-} a) & (21) \\
(a \dot{-} b) \dot{-} (a \sqcap b) & = & a \dot{-} b & (22) \\
(b \dot{-} a) \sqcup (b \sqcap a) & = & b & (23) \\
(c \dot{-} b) \dot{-} a & = & (c \dot{-} a) \dot{-} (b \dot{-} a) & (24) \\
(a \sqcap b) \dot{-} c & \sqsubseteq & a \sqcap (b \dot{-} c) & (25) \\
(a \sqcap b) \dot{-} c & \sqsubseteq & (a \dot{-} c) \sqcap (b \dot{-} c) & (26) \\
b \dot{-} (b \dot{-} a) & \sqsubseteq & a \sqcap b & (27) \\
a \dot{-} b \sqsubseteq b & \text{implies} & a \sqsubseteq b & (28) \\
c \dot{-} (a \sqcup b) & \sqsubseteq & (c \dot{-} a) \sqcap (c \dot{-} b) & (29) \\
(a \dot{-} b) \dot{-} (b \dot{-} a) & = & a \dot{-} b & (30) \\
(a \sqcup b) \dot{-} (a \sqcap b) & = & (a \dot{-} b) \sqcup (b \dot{-} a) & (31) \\
a \dot{-} (a \sqcap b) & = & a \dot{-} b & (32) \\
b \sqsupseteq a & \text{implies} & a \sqcup (b \dot{-} a) = b & (33) \\
a & \sqsubseteq & a \sqcup b & (34) \\
a & \sqsupseteq & a \sqcap b & (35) \\
a \sqsubseteq c \quad \text{and} \quad b \sqsubseteq c & \text{imply} & a \sqcup b \sqsubseteq c & (36) \\
c \sqsubseteq a \quad \text{and} \quad c \sqsubseteq b & \text{imply} & c \sqsubseteq a \sqcap b & (37) \\
a \sqsubseteq c \quad \text{and} \quad b \sqsubseteq d & \text{imply} & a \sqcup b \sqsubseteq c \sqcup d & (38) \\
a \sqsubseteq c \quad \text{and} \quad b \sqsubseteq d & \text{imply} & a \sqcap b \sqsubseteq c \sqcap d & (39) \\
a \sqcup (b \sqcap c) & = & (a \sqcup b) \sqcap (a \sqcup c) & (40) \\
a \sqcap (b \sqcup c) & = & (a \sqcap b) \sqcup (a \sqcap c) & (41)
\end{array}
$$

B.2 Additional Properties of Disjoint Elements

Let $\langle \mathcal{L}, \sqcup, \sqcap, \dot{-}, \top \rangle$ be a Brouwerian algebra with $a, b, c, d \in \mathcal{L}$ and a, b, and c be mutually disjoint, i.e., $a \parallel b$, $a \parallel c$, and $b \parallel c$. Then the following hold:

$$
\begin{array}{rcll}
a \dot{-} b & = & a & (42) \\
(a \sqcup b) \dot{-} a & = & b & (43) \\
a & \parallel & (b \sqcap c) & (44) \\
a & \parallel & (b \sqcup c) & (45)
\end{array}
$$

$$
\begin{aligned}
(a \sqcup b) \sqcap (b \sqcup c) &= b && (46)\\
(a \dot{-} d) &\parallel b && (47)
\end{aligned}
$$

Bibliography

[Abr90] Samson Abramsky. The lazy lambda calculus. In David A. Turner, editor, *Research Topics in Functional Programming*, chapter 4, pages 65–116. Addison-Wesley, Reading, Massachusetts, 1990.

[ACCL90] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. In *Proc. Seventeenth ACM Symposium on Principles of Programming Languages*, San Francisco, 1990.

[ACR+87] B. Alpern, A. Carle, B. Rosen, P. Sweeney, and K. Zadeck. Incremental evaluation of attributed graphs. Technical Report RC 13205, IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598, October 1987.

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Massachusetts, 1974.

[AKP84] Arvind, Vinod Kathail, and Keshav Pingali. Sharing of computation in functional language implementations. In *Proc. International Workshop on High-Level Computer Architecture*, Los Angeles, 1984.

[AP81] Luigia Aiello and Gianfranco Prini. An efficient interpreter for the lambda calculus. *Journal of Computer and System Sciences*, 23:383–424, 1981.

[Aug84] L. Augustsson. A compiler for Lazy ML. In *Proc. ACM Symp. on Lisp and Functional Programming*, Austin, 1984.

[Bar84] H.P. Barendregt. *The Lambda Calculus*, volume 103 of *Studies in Logic and the Foundations of Mathematics.* North-Holland, Amsterdam, 1984.

[BBKV76] H.P. Barendregt, J. Bergstra, J.W. Klop, and H. Volken. Degrees, reductions, and representability in the lambda calculus. Preprint 22, Department of Mathematics, University of Utrecht, The Netherlands, 1976.

[BCL85] G. Berry, P.-L. Curien, and J.-J. Lévy. Full abstraction for sequential languages: The state of the art. In M. Nivat and J. Reynolds, editors, *Algebraic Methods in Semantics*, chapter 3, pages 89–132. Cambridge University Press, Cambridge, 1985.

[Bel57] R. E. Bellman. *Dynamic Programming.* Princeton University Press, Princeton, New Jersey, 1957.

[Ben82] S. A. Bengelloun. *Aspects of Incremental Computing.* Ph.D. dissertation, Yale University, 1982.

[BKKS87] H.P. Barendregt, J.R. Kennaway, J.W. Klop, and M.R. Sleep. Needed reduction and spine strategies for the lambda calculus. *Information and Computation*, 75:191–231, 1987.

[BL79] Gérard Berry and Jean-Jacques Lévy. Minimal and optimal computations of recursive programs. *Journal of the Association for Computing Machinery*, 26(1):148–175, 1979.

[BPR85] A. M. Berman, M. C. Paull, and B. G. Ryder. Proving relative lower bounds for incremental algorithms. Technical Report DCS-TR-154, Department of Computer Science, Rutgers University, New Brunswick, New Jersey 08903, 1985.

[Bra69] W.S. Brainerd. Tree generating regular systems. *Information and Control*, 14:217–231, 1969.

[BvEG+87a] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, and M.R. Sleep. Term graph rewriting. In *Proc. PARLE Conference, Vol. II: Parallel Languages*, pages 141–158, Eindhoven, The Netherlands, 1987. Springer-Verlag. Lecture Notes in Computer Science 259.

[BvEG+87b] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, and M.R. Sleep. Towards an intermediate language based on graph rewriting. In *Proc. PARLE Conference, Vol. II: Parallel Languages*, pages 159–173, Eindhoven, The Netherlands, 1987. Springer-Verlag. Lecture Notes in Computer Science 259.

[CCM87] G. Cousineau, P.-L. Curien, and M. Mauny. The Categorical Abstract Machine. *Science of Computer Programming*, 8:173–202, 1987.

[Ce86] R. L. Constable and et. al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewoods Cliff, N.J., 1986.

[CF58] H.B. Curry and R. Feys. *Combinatory Logic*, volume 1. North Holland, Amsterdam, 1958.

[CH78] Francis Chin and David Houck. Algorithms for updating minimal spanning trees. *Jounral of Computer and Systems Sciences*, 16(3):333–344, June 1978.

[Chu41] A. Church. *The Calculi of Lambda Conversion.* Princeton University Press, Princeton, NJ, 1941.

[Cur77] Haskell B. Curry. *Foundations of Mathematical Logic.* Dover, New York, 1977.

[Cur86a] P.-L. Curien. Categorical combinators. *Information and Control*, 69:188–254, 1986.

[Cur86b] P.-L. Curien. De la difficulté d'implémenter le partage optimal au sens de Lévy. Unpublished Note, Université de Paris VII, 1986.

[dB72] N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Proc. of the Koninklijke Nederlandse Akademie van Wetenschappen*, 75(5):381–392, 1972.

[dB78] N.G. de Bruijn. Lambda calculus with namefree formulas involving symbols that represent reference transforming mappings. *Proc. of the Koninklijke Nederlandse Akademie van Wetenschappen*, 81(3):348–356, 1978.

[Der87] Nachum Dershowitz. Termination of rewriting. *J. Symbolic Computation*, 3:69–116, 1987.

[DGKS76] J. Daniel, W.B. Gregg, L. Kaufman, and G.W. Stewart. Reorthogonalization and stable algorithms for updating the Gram Schmidt QR-factorization. *Math. Comp.*, 30:772–795, 1976.

[DJL86] Pierre Deransart, Martin Jourdan, and Bernard Lorho. A survey on attribute grammars. Rapport de Recherche 485, Institut National de Recherche en Informatique et en Automatique, Domaine de Voluceau, Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France, 1986.

[Ear76] J. Earley. High level iterators and a method for automatically designing data structure representation. *Journal of Computer Languages*, 1:321–342, 1976.

[EKMS⁺81] H. Ehrig, H.-J. Kreowski, A. Maggiolo-Schettini, B.K. Rosen, and J. Winkowski. Transformations of structures: an algebraic approach. *Mathematical Systems Theory*, 14:305–334, 1981.

[FH88] Anthony J. Field and Peter G. Harrison. *Functional Programming.* Addison-Wesley, Wokingham, England, 1988.

[FRW90] William M. Farmer, John D. Ramsdell, and Ronald J. Watro. A correctness proof for combinator reduction with cycles. *ACM Trans. on Programming Languages and Systems*, 12(1):123–134, January 1990.

[FW87] Jon Fairbairn and Stuart Wray. TIM: A simple, lazy abstract machine to execute supercombinators. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 34–45, Portland, 1987. Springer-Verlag. Lecture Notes in Computer Science 274.

[Gog90] Joseph A. Goguen. Proving and rewriting. In *Proc. Second International Conference on Algebraic and Logic Programming*, Nancy, France, Oct 1990.

[Har87] Thérèse Hardin. *Résultats de Confluence pour les Règles Fortes de la Logique Combinatoire Catégorique et Liens avec les Lambda-Calculs.* Ph.D. dissertation, Université de Paris VII, 1987. (Thèse de Doctorat).

[Har89] Thérèse Hardin. Confluence results for the pure strong categorical logic CCL. λ-calculi as subsystems of CCL. *Theoretical Computer Science*, 65:291–342, 1989.

[Hen80] P. Henderson. *Functional Programming: Application and Implementation.* Prentice Hall International, Englewood Cliffs, New Jersey, 1980.

[HL79] G. Huet and J.-J. Lévy. Call by need computations in non-ambiguous linear term rewriting systems. Rapport de Recherche 359, Institut National de Recherche en Informatique et en Automatique, Domaine de Voluceau, Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France, 1979.

[HL86] Thérèse Hardin and Alain Laville. Proof of termination of the rewriting system SUBST on CCL. Report de Recherche 560, Institut National de Recherche en Informatique et en Automatique, Domaine de Voluceau, Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France, August 1986.

[HM76] P. Henderson and J.H. Morris. A lazy evaluator. In *Proc. Third ACM Symposium on Principles of Programming Languages*, pages 95–103, 1976.

[HMT88] R. Harper, R. Milner, and M. Tofte. The definition of standard ML (version 2). Technical Report ECS-LFCS-88-62, LFCS, University of Edinburgh, 1988.

[HO80] G. Huet and D.C. Oppen. Equations and rewrite rules: A survey. In R.V. Book, editor, *Formal Language Theory, Perspectives, and Open Problems*, pages 349–405. Academic Press, London, 1980.

[Hoo87] R. Hoover. Incremental graph evaluation. Technical Report 87-836, Department of Computer Science, Cornell University, Ithaca, New York 14853, May 1987. (Ph.D. Thesis).

[HP90] Paul Hudak and Philip Wadler, eds. Report on the programing language HASKELL, (version 1.0). Technical Report YALEU/DC/RR-777, Department of Computer Science, Yale University, New Haven, April 1990.

[HS86] J.R. Hindley and J.P. Seldin. *Introduction to Combinators and Lambda-Calculus*, volume 1 of *London Mathematical Society Student Texts*. Cambridge University Press, Cambridge, 1986.

[Hue80] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980.

[Hue86] G. Huet. Formal structures for computation and deduction (preliminary edition). Unpublished Course Notes, Carnegie-Mellon University, 1986.

[Hug84a] J. Hughes. Why functional programming matters. Report 16, Programming Methodology Group, Department of Computer Sciences, Chalmers University of Technology and University of Göteborg, S-412 96 Göteborg, Sweden, 1984.

[Hug84b] R.J.M. Hughes. *The Design and Implementation of Programming Languages*. Ph.D. dissertation, Oxford University, September 1984. (PRG-40).

[Joh84] T. Johnsson. Efficient compilation of lazy evaluation. In *Proc. ACM Conf. on Compiler Construction*, Montreal, 1984.

[Joh85] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 190–203. Springer-Verlag, 1985. Lecture Notes in Computer Science 201.

[JSS89] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. MIX: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2, 1989.

[Kah87] Gilles Kahn. Natural semantics. Rapport de Recherche 601, Institut National de Recherche en Informatique et en Automatique, Domaine de Voluceau, Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France, February 1987.

[Kat84] T. Katayama. Translation of attribute grammars into procedures. *ACM Transactions on Programming Languages and Systems*, 6(3):345–369, July 1984.

[Kat90] Vinod Kathail. *An Optimal Interpreter for the λ-calculus.* Ph.D. dissertation, Massachusetts Institute of Technology, 1990.

[KB70] D. Knuth and P. Bendix. Simple word problems in universal algebra. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, Elmsford, NY, 1970.

[Ken84] J.R. Kennaway. An outline of some results of Staples on optimal reduction orders in replacement systems. Report CSA/19/1984, School of Information Systems, University of East Anglia, Norwich, U.K., 1984.

[Klo80] J.W. Klop. *Combinatory Reduction Systems*, volume 127 of *Mathematical Centre Tracts.* Mathematical Centre, Kruislaan 413, Amsterdam 1098SJ, The Netherlands, 1980.

[Lam90] John Lamping. An algorithm for optimal lambda calculus reduction. In *Proc. Seventeenth ACM Symposium on Principles of Programming Languages*, San Francisco, 1990.

[Lan64] P.J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.

[Lév75] Jean-Jacques Lévy. An algebraic interpretation of the $\lambda\beta$K-calculus and a labelled λ-calculus. In C. Böhm, editor, *Proc. Symp. on λ-Calculus and Computer Science Theory.* Springer-Verlag, 1975. Lecture Notes in Computer Science 37.

[Lév78] Jean-Jacques Lévy. *Réductions correctes et optimales dans le lambda-calcul.* Ph.D. dissertation, Université de Paris VII, 1978. (Thèse d'Etat).

[Lév80] Jean-Jacques Lévy. Optimal reductions in the lambda-calculus. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism.* Academic Press, London, 1980.

[Lin87] R.D. Lins. On the efficiency of categorical combinators as a rewriting system. *Software—Practice and Experience*, 17(8):547–559, August 1987.

[Lom67] L.A. Lombardi. Incremental computation. In F.L. Alt and M. Rubinoff, editors, *Advances in Computers, Vol. 8*, pages 247–333. Academic Press, New York, 1967.

[Mar89] T.J. Marlowe. Data flow analysis and incremental iteration. Technical Report DCS-TR-255, Department of Computer Science, Rutgers University, October 1989. Ph.D. Thesis.

[Mar91] Luc Maranget. Optimal derivations in weak lambda-calculi and in orthogonal term rewriting systems. In *Proc. Eighteenth ACM Symp. on Principles of Programming Languages*, pages 255–269, Orlando, FL, January 1991.

[May81] B. H. Mayoh. Attribute grammars and mathematical semantics. *SIAM Journal on Computing*, 10(3):503–518, August 1981.

[Mic68] D. Michie. "Memo" functions and machine learning. *Nature*, 218:19–22, April 1968.

[MT46] J.C.C. McKinsey and A. Tarski. On closed elements in closure algebras. *Annals of Mathematics*, 47(1):122–162, January 1946.

[O'D77] Michael J. O'Donnell. *Computing in Systems Described by Equations.* Springer-Verlag, Berlin, 1977. Lecture Notes in Computer Science 58.

[PCB84] M. C. Paull, Ching-an Cheng, and A. M. Berman. Exploring the structure of incremental algorithms. Technical Report DCS-TR-139, Department of Computer Science, Rutgers University, New Brunswick, New Jersey 08903, 1984. (Preliminary Version).

[Pey87] S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall International, Englewood Cliffs, New Jersey, 1987.

[PK82] R. Paige and S. Koenig. Finite differencing of computable expressions. *Transactions on Programming Languages and Systems*, 4(3):402–454, July 1982.

[PSS89] Prakash Panangaden, Vasant Shanbhogue, and Eugene W. Stark. Stability and sequentiality in dataflow networks. Technical Report TR 89-1055, Department of Computer Science, Cornell University, Ithaca, NY 14853-7501, November 1989.

[PT89] W. Pugh and T. Teitelbaum. Incremental computation by function caching. In *Proc. Sixteenth ACM Symp. on Principles of Programming Languages*, 1989.

[Pug88] W. W. Pugh. *Incremental Computation and the Incremental Evaluation of Functional Programs.* Ph.D. dissertation, Cornell University, 1988.

[RD87] B. G. Ryder and M. D.Carroll. Incremental data flow analysis via attributes. Technical Report LCSR-TR-93, Department of Computer Science, Rutgers University, June 1987.

[Rep82] T. Reps. Optimal-time incremental semantic analysis for syntax-directed editors. *Proc. Ninth ACM Symp. on Principles of Programming Languages*, pages 169–176, 1982.

[Rep84] T. W. Reps. *Generating Language-Based Environments.* The MIT Press, Cambridge, Massachusetts, 1984.

[Rep89] Thomas Reps. On the algebraic properties of program integration. Technical Report 856, Computer Sciences Department, University of Wisonsin, 1210 W. Dayton St., Madison, WI 53706, June 1989.

[Ros73] Barry K. Rosen. Tree-manipulating systems and Church-Rosser theorems. *Journal of the ACM*, 20(1):160–187, January 1973.

[Ros81] B. Rosen. Linear cost is sometimes quadratic. In *Proc. Eighth ACM Symp. on Principles of Programming Languages*, pages 117–124, June 1981.

[RS63] Helena Rasiowa and Roman Sikorski. *The Mathematics of Metamathematics.* Polish Scientific Publishers, Warsaw, 1963.

[RT88] T. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors.* Springer-Verlag, New York, 1988.

[RTD83] T. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependent analysis for language-based editors. *Transactions on Programming Languages and Systems*, 5(3):449–477, July 1983.

[SH90] R.S. Sundaresh and P. Hudak. A theory of incremental computation and its application. Technical Report YALEU/DCS/RR-770, Department of Computer Science, Yale University, New Haven, CT, March 1990.

[SP75] P.M. Spira and A. Pan. On finding and updating spanning trees and shortestpaths. *SIAM Journal of Computing*, 4(3):375–380, September 1975.

[SS71] D.S. Scott and C. Strachey. Toward a mathematical semantics for computer languages. In J. Fox, editor, *Proc. Symp. on Computers and Automata*, pages 19–46, New York, 1971. Polytechnic Institute of Brooklyn Press.

[Sta80a] John Staples. Computation on graph-like expressions. *Theoretical Computer Science*, 10:171–185, 1980.

[Sta80b] John Staples. Optimal evaluations of graph-like expressions. *Theoretical Computer Science*, 10:297–316, 1980.

[Sta80c] John Staples. Speeding up subtree replacement systems. *Theoretical Computer Science*, 11:39–47, 1980.

[Sta82] John Staples. Two-level expression representation for faster evaluation. In *Proc. Second International Workshop on Graph Grammars and Their Applications*, pages 392–404. Springer-Verlag, 1982. Lecture Notes in Computer Science 153.

[Sta87] Eugene W. Stark. Concurrent transition system semantics of process networks. In *Proc. Fourteenth ACM Symp. on Principles of Programming Languages*, pages 291–299, Munich, January 1987.

[Tov76] Donald Francis Tovey. *A Companion to Beethoven's Pianoforte Sonatas.* AMS Press, New York, 1976. From the edition of 1931, London.

[Tur79] D.A. Turner. A new implementation technique for applicative languages. *Software—Practice and Experience*, 9:31–49, 1979.

[Tur81] D.A. Turner. The semantic elegance of applicative languages. In *Proceedings of the ACM Symposium on Functional Languages and Computer Architecture*, 1981.

[Van89] B. Vander Zanden. *Incremental Constraint Satisfaction and its Application to Graphical Interfaces.* Ph.D. dissertation, Cornell University, 1989.

[Vui73] J. Vuillemin. *Proof Techniques for Recursive Programs.* Ph.D. dissertation, Computer Science Dept., Stanford University, Stanford, CA, 1973.

[Vui74] Jean Vuillemin. Correct and optimal implementations of recursion in a simple programming language. *Journal of Computer and System Sciences*, 9:332–354, 1974.

[Wad71] C.P. Wadsworth. *Semantics and Pragmatics of the Lambda-Calculus.* Ph.D. dissertation, Oxford University, 1971.

[WJ88] J. A. Walz and G. F. Johnson. Incremental evaluation for a general class of circular attribute grammars. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1988.

[Yel88] Daniel Yellin. A dynamic transitive closure algorithm. Technical report, IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598, February 1988.

[Yok89] Hirofumi Yokouchi. Church-Rosser theorem for a rewriting system on categorical combinators. *Theoretical Computer Science*, 65:271–290, 1989.

[YS89] D. Yellin and R. Strom. INC: A language for incremental computations. Technical report, IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598, October 1989.

[Zad84] Frank Kenneth Zadeck. Incremental data flow analysis in a structured program editor. In *Proc. ACM SIGPLAN Symposium on Compiler Construction*, 1984. Appeared as *SIGPLAN Notices* 19(6), June 1984.